BOUNDED PARALLELISM IN COMPUTER ALGEBRA

by

Stephen Michael Watt

A thesis
presented to the University of Waterloo
in fulfillmentof the
thesis requirement of the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, 1985
© S.M. Watt 1985

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-29618-6

I hereby delcare that I am the sole author if this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Signature

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means in total or in part, at the request of other institutions of individuals for the purpose of scholarly research.

Signature

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

This thesis examines the performance improvements that can be made by exploiting parallel processing in symbolic mathematical computation. The study focuses on the use of high-level parallelism in the case where the number of processors is fixed and independent of the problem size, as in existing multiprocessors.

Since seemingly small changes to the inputs can cause dramatic changes in the execution times of many algorithms in computer algebra, it is not generally useful to use static scheduling. We find it is possible, however, to exploit the high-level parallelism in many computer algebra problems using dynamic scheduling methods in which subproblems are treated homogeneously.

Our investigation considers the reduction of execution time in both the case of AND-parallelism, where all of the subproblems must be completed, and the less well studied case of OR-parallelism, where completing any one of the subproblems is sufficient. We examine the use of AND and OR-parallelism in terms of the *problem heap* and *collusive* dynamic scheduling schemes which allow a homogeneous treatment of subtasks. A useful generalization is also investigated in which each of the subtasks may either succeed or fail and execution completes when the first success is obtained.

We study certain classic problems in computer algebra within this framework. A collusive method for integer factorization is presented. This method attempts to find different factors in parallel, taking the first one that is discovered. Problem heap algorithms are given for the computation of multivariate polynomial GCDs and the computation of Gröbner bases. The GCD algorithm is based on the sparse modular GCD and performs the interpolations in parallel. The Gröbner basis algorithm exploits the independence of the reductions in basis completion to obtain a parallel method.

In order to make evaluations in concrete terms, we have constructed a system for running computer algebra programs on a multiprocessor. The system is a version of Maple able to distribute processes over a local area network. The fact that the multiprocessor is a local area network need not be considered by the programmer.

Acknowledgements

I would like to thank my supervisor, K.O. Geddes, for his guidance and support throughout my Ph.D. studies. The Symbolic Computation Group at the University of Waterloo provided the stimulating environment in which this research was conducted. I would like to express my gratitude to the Natural Sciences and Engineering Research Council of Canada for financial support. In addition, I would like to acknowledge many fruitful discussions with my friends and colleagues both in Waterloo and in Yorktown Heights.

Contents

1 INTRODUCTION **1**
- 1.1 Motivation 1
- 1.2 Views of Parallelism 2
- 1.3 Scope 4
- 1.4 Related Work 5
- 1.5 Outline 8

2 PARADIGMS FOR PARALLELISM IN COMPUTER ALGEBRA **10**
- 2.1 Considerations for Computer Algebra 10
- 2.2 Considerations for High Level Parallelism 11
- 2.3 The Problem Heap and the Administrator Process 11
- 2.4 Collusion and the Don Process 12

3 COLLUSION **14**
- 3.1 Colluding Processes 14
- 3.2 Execution Duration 16
- 3.3 Parallelism 18
- 3.4 Modelling Collusion 19
- 3.5 Expected Execution Times 20
- 3.6 Example: Satisficing Search 24
- 3.7 The Existence of Extremely Collusive Densities 28

4 INTEGER FACTORIZATION **33**
- 4.1 The Role of Integer Factorization 33
- 4.2 A Brief Survey 33
- 4.3 A Parallel Algorithm: llifactor 36
- 4.4 Some Empirical Results 37

5 POLYNOMIAL GREATEST COMMON DIVISORS **42**
- 5.1 The Role of Greatest Common Divisors 42
- 5.2 A Survey of GCD Algorithms 42
- 5.3 A Parallel Algorithm: llgcd 44
- 5.4 Analysis 46

6 GRÖBNER BASES **52**
- 6.1 The Role of Gröbner Bases 52
- 6.2 Buchberger's Algorithm 52

7 DEVELOPING A PARALLEL COMPUTER ALGEBRA SYSTEM 59
7.1 Chapter Overview 59
7.2 Parallel Processing Primitives 59
7.3 Message Passing in the Administrator and the Don 64
7.4 Shared Memory in Maple 65
7.5 A Multiprocessing Prototype 67
7.6 A Multiprocessing Version of Maple 70

8 IMPLEMENTATION ASPECTS 73
8.1 Chapter Overview 73
8.2 The Architecture of the Multiprocessing Maple System 73
8.3 AND Parallelism 79
8.4 OR Parallelism 83
8.5 Broadcasting 83

9 CONCLUDING REMARKS 85
9.1 Summary 85
9.2 Contributions 86
9.3 Directions for Further Work 86

BIBLIOGRAPHY 88
General References 88
Parallelism in Computer Algebra 91
Scheduling 94
Collusion 95
Integer Factorization 97
Greatest Common Divisors 98
Gröbner Bases 100

INDEX 101

List of Figures

2.1 The Administrator Process 11
2.2 The Don Process 13

3.1 $\langle t \rangle_{\mathrm{ser}} / \langle t \rangle_{\mathrm{par}}$ for various values of $\bar{p}$, N and t/T. 32

4.1 A Collusive Factor Finding Method 35
4.2 A Parallel Factorization Algorithm 36
4.3 Empirical Probability for Success 38
4.4 Empirical Probability for Failure 39
4.5 Tabulation of Probabilities at Certain Values 41

5.1 GCDs in Rational Function Arithmetic 43
5.2 The Sparse Modular GCD 45
5.3 A Parallel GCD Algorithm 47

6.1 Buchberger's Algorithm 55
6.2 A Parallel Gröbner Basis Algorithm 56
6.3 A Parallel Gröbner Basis Using Administration 57

7.1 Message Passing for the Proprietor 63
7.2 Message Passing for the Administrator 64
7.3 Message Passing for the Don 65

8.1 Layers in the Implementation of the Multiprocessing Maple System . . 73
8.2 Library Used by Kernel of Multiprocessing Maple System 78
8.3 Implementation of AllOf using the Administrator Construct 80
8.4 Implementation of OneOf using the Don Construct 81
8.5 Implementation of Broadcast 82

1.0 INTRODUCTION

1.1 Motivation

Small problems in computer algebra can be solved with moderate time and storage. However, even simply stated problems with moderately sized inputs, can be very expensive to compute. Many such problems run out of storage or take excessive time. Large problems, if they can be done at all, can take hours or even days. As in many other fields, we run into the question of what is the best way to obtain raw processing power.

For many years now, it has been apparent that multiprocessing will ultimately be the solution to the problem of reducing execution time. There are at least four reasons for this. The first is that at any given time, manufacturing technology places limits on how fast a uniprocessor can be. The second reason is cost effectiveness. It appears that it will always be less expensive to produce several slow processors than a single very fast one. If these slow processors can be combined in such a way that they perform work together, then they may have a combined power that exceeds that of the single fast processor. The third reason is reliability. If a failure occurs in a multiprocessor then, with the proper software support, the affected element can be shut down while the system continues to run. Finally, the fourth reason is extensibility. With certain multiprocessor architectures it is possible to incrementally increase the performance by adding processors. With a uniprocessor the entire unit needs to be replaced in order to obtain an increase in speed.

From the point of view of the computer algebra community, it is natural to ask what performance improvements can be made by exploiting parallelism in symbolic mathematical computation. Many problems in this field seem suitable for division into parallel processes. For example, when a sum is integrated, the first thing that is usually tried is to integrate each of the terms separately. At this high level of problem description, the tasks to be performed are quite substantial. One would therefore expect the overhead of managing multiple processes to be relatively insignificant.

Although it is clear that it would be advantageous to exploit parallel processing, there is the consideration of how to make the processors work together. There are problems in both hardware and software that must be addressed in order to successfully take advantage of multiprocessing. First there are abundant issues in the design of suitable multiprocessor hardware. Then, given a particular multiprocessor, there is the requirement of developing software that can utilize it. None of these problems is trivial.

In addition to these practical considerations, the problems exposed in examining parallel algorithms for computer algebra are interesting in their own right.

1.2 Views of Parallelism

There are many possible reasons why one may wish to structure a program as multiple processes. For example, it is often desirable to make inherent parallelism explicit (as in a simulation) or to maintain program modularity (as in an operating system). As far as this thesis is concerned, however, the aim of using parallelism in computing is to decrease execution time.

The use of parallelism for performance improvement may be approached from several different views. We outline four independent aspects of parallelism which can be used to characterize investigations.

SIMD versus MIMD Parallelism

There is a classification of parallel computer architectures by Flynn [**Flynn66**] that is based on the number of "instruction streams" and "data streams" (the number of instruction streams is the number of processors.) The class to which a processor belongs is determined by whether the processor accepts *single* or *multiple* streams of each variety. This yields four classes, abbreviated SISD, SIMD, MISD and MIMD. In this classification the traditional von Neumann machine would be classified as a single instruction stream, single data stream (SISD) processor.

In the SIMD class a single program operates upon multiple data objects simultaneously (usually as vectors). Processors such as ILLIAC IV fall into this category. This sort of parallel processing is most useful in numerical computation where the same procedure must be applied at each one of a number of points.

Certain problems in computer algebra can also take this approach. For example, certain operations on dense univariate polynomials can be done as vector operations. The same is true for dealing with arbitrary precision integers upon which most computer algebra systems rely. Other candidates are problems solved by multiple homomorphisms. However, the problems which can take advantage of this sort of parallelism are fairly limited because of the inhomogeneity of the operations in computer algebra.

The category of multiple instruction stream, single data stream processors does not seem to be widely useful.

In the most general form of parallel processor, several independent processes work upon different sets of data. This is known as a MIMD processor in Flynn's classification and includes both SIMD and MISD as special cases. In this thesis we consider MIMD processing and we shall use the term *multiprocessor* to refer to a MIMD machine.

Bounded versus Unbounded Parallelism

The first question that must be addressed is the number of processors that are available. This divides investigations into two categories — those that consider a fixed number of processors, independent of the problem size, and those that consider the number of processors available to grow with the size of the problem. We call the former *bounded parallelism* and the latter *unbounded parallelism*.

Questions in which the number of processors does not have an *a priori* bound lie in the domain of parallel complexity. Here the problem is to determine lower bounds on the parallel time, allowing either an arbitrary number of processors or a number of processors determined by a function of the problem size. For example, in sorting n items what time is required if there are $O(n)$ processors available? What time is required if $O(n \log n)$ processors are available? Such questions are asked to give bounds on performance, since in practice there are always only a fixed number of processors available.

It is possible on the other hand, to ask how to use properly a given, fixed number of processors. Here the objective is to schedule tasks on processors so as to minimize (or approximately minimize) one of several measures. When the scheduling is done statically, before execution, this problem is known in operations research as *job-shop scheduling* (see, for example, [**Salvador78**]).

Some quantities which are of interest in job-shop scheduling are the *completion time* (the time at which processing of all tasks is completed), the *flow-time* (the total time from start to completion of a task) and the *waiting time* (that portion of the flow time that a task spends inactive). It is usual to minimize either the expected- or worst-case of one of these quantities. If there are deadlines on the tasks, then lateness may also be a consideration.

In parallel processing applications, completion time is of the greatest interest. For a problem where there is a certain fixed amount of work to be done, the objective is to share it amongst the processors. If the time to perform the work on a single processor is denoted by T, the goal is to take advantage of the structure of the data or of an algorithm so that if there are N processors the completion time is reduced to as close to T/N as possible.

High-level versus Low-level Parallelism

Another aspect is the level at which the parallelism is exploited. The division of a problem for parallel execution can be done at many levels. If an algorithm is divided at a level of abstraction close to the problem description, we say that it uses *high-level parallelism*. If an algorithm is described in a serial fashion at the high level but

makes use of parallelism in its primitives, we say that it uses *low-level parallelism*. Investigations into the use of high-level parallelism address issues which arise from the domain of the problem. The questions that arise in the use of low-level parallelism tend to have a more general, system oriented flavour.

AND versus OR Parallelism

The most straightforward way to take advantage of parallelism is to divide a problem into subtasks and to distribute the subtasks amongst the collection of processors. When all of the subtasks have been completed then the original problem is solved. Alternatively, it is sometimes possible to split a problem in such a way that only *one* of the subtasks needs to be completed. We call the former *AND parallelism* and the latter *OR parallelism*.

It is obvious how AND parallelism can be used to decrease execution time. What is less clear is how OR parallelism can be exploited. If only one of the subtasks needs to be completed, then how can executing it on one of the processors of a multiprocessor have any advantage over simply executing that one subtask on a uniprocessor? The advantage comes from the fact that the subtasks can have widely differing execution times which cannot be determined in advance.

It is commonly assumed that, when a single processor is used, the structuring of a program with processes always leads to a deterioration of performance (due to the overhead of process management). This assumption is invalid. In fact, there are problems where the execution time even on a uniprocessor can be reduced by taking advantage of OR parallelism. When multiple processes work independently toward a common goal we call them *colluding processes* after Hoare [**Hoare76**]. We use the term *collusion* to describe the use of OR parallelism to decrease execution time.

1.3 Scope

Surveying previous and on-going work shows that the studies of parallelism in computer algebra may be split into two categories. The first category encompasses work on the suitability of certain key primitives to take advantage of low-level parallelism on special purpose hardware. The second category contains the work on the parallel complexity of algebraic problems.

Between these two lies the important but largely unstudied domain of parallel algorithms for the classic problems in computer algebra that are suitable for existing architectures. The hardware needed to take advantage of parallelism for these problems is widely available in the form of local area networks, and, less widely, in the form of tightly coupled multiprocessors. This thesis attempts to fill this gap partially, by

exploring the use of bounded, high-level parallelism for problems in computer algebra. Since we are concerned with OR parallelism as well as the more often studied AND parallelism, we begin by studying the use of collusion.

In our study of bounded parallelism, we maintain the overall objective that the algorithms we develop be actually implementable. Since no multiprocessing computer algebra systems, upon which the the algorithms could be tested, existed, the decision was made to construct such a system. The issues involved in the design of such systems subsequently formed a significant part of this thesis.

Certain algorithms are capable of using a particular number of processors but do not generalize to the use of more. Although a speedup by a factor of two or three, for example, is certainly worthwhile, the most valuable parallel algorithms are those general enough to effectively take advantage of an arbitrary number of processors. We restrict our attention to algorithms of this type.

We do not address reliablility issues such as handling processor or connection failures.

1.4 Related Work

There have been numerous investigations into the use of parallelism in other fields, in parallel complexity theory, in utilizing a fixed number of processors, and in parallel computer architecture. Although some of the references occur in the computer algebra literature, they do not pertain specifically to computer algebra (e.g., [**Wise84**] [**Legendi85**]).

This section presents a summary of the work that has been done on the use of parallelism in computer algebra. Since the body of literature is rather small, we do not restrict our attention to bounded, high-level parallelism. The use of OR parallelism for performance improvement has not received much attention and so we also present a summary of the results which pertain to collusion.

Implementation Language Support

Since many computer algebra systems are based on LISP, developments in LISP for concurrent programming and parallel LISP machines (e.g. the Bath concurrent LISP machine [**Marti83**]) are of interest to the computer algebra community. Fortran has also been an implementation language for computer algebra systems (e.g. Altran, Aldes/SAC-II). The existence of vectorizing Fortran compilers provides an avenue for Fortran based systems to take some advantage of low-level parallelism on super computers. Indeed, there is significant interest in the potential of super computers in computer algebra [**Arnon84**] [**Berman84**].

Low-Level Parallelism

There has been some work investigating the use of low-level parallelism in computer algebra based on special-purpose hardware. Kung suggests VLSI designs for polynomial multiplication and division based on systolic arrays [**Kung81**]. His systolic design improves upon previous work which required broadcasting of data to all cells at every cycle. Systolic arrays are also used in recent work by Yun and Zhang [**Yun85**] for residue arithmetic. Davenport and Robert take advantage of the parallelism inherent in the Modular GCD in their proposal for a VLSI implementation [**Davenport84**]. In addition, Smit gives some preliminary thoughts on a single chip multiprocessor design for computer algebra [**Smit83**].

The researchers working on the L-Network project at the University of Linz, Austria, are designing an interconnection network system intended to be used in computer algebra and logic inference [**Bibel84**] [**Buchberger85a**]. This system is based on the concept of L-modules. Each L-module consists of a processor, a shared memory component and bus switches. Buchberger has presented arguments for the suitability of these structures to the needs of symbolic computing. A prototype consisting of 8 L-modules was exhibited at Eurocal'85.

Beardsworth has investigated the application of array processors to the manipulation of polynomials [**Beardsworth81**]. He describes a system for the manipulation of univariate polynomials with single word integer coefficients. This experimental system runs on the I.C.L. Distributed Array Processor, a SIMD machine with 64 x 64 simple processors.

Parallel Complexity

Recently von zur Gathen, Kaltofen and colleagues have initiated investigations into the parallel complexity of algebraic algorithms [**Borodin82**] [**Gathen83a, 83b, 84**] [**Kaltofen 85a-d**]. These papers give constructions for a number of algebraic problems based on an abstract model of parallel computation. The results are given in high-level descriptions which assume an underlying model of parallel computation equivalent to an algebraic PRAM (a parallel RAM with arithmetic operations and tests over an appropriate ground field). These results can also be expressed in terms of an equivalent model of parallel computation, namely a parallel algebraic computation graph. In this model each node processor can perform arithmetic and tests over a ground field. The parallel time is given by the depth of the graph. The literature on algebraic computation graphs is well established (see [**Reif83**], [**Ben-Or83**], [**Valiant83**] and references cited therein). We summarize below the published results which follow this line of pursuit. In each case the result can be viewed either as the construction of a parallel algorithm

using a polynomial number of processors and requiring poly-logarithmic time or as a circuit of polynomial size and poly-logarithmic depth.

The constructions are based on a result of Valiant *et al* for converting certain sequential programs to parallel ones [**Valiant83**]. The paper shows that a sequential straight-line program that computes a polynomial of degree n in t steps using only the operations of addition, subtraction and multiplication can be converted to a fast parallel program requiring $O(t^3n^6)$ processors and parallel time $O(\log n\ (\log n + \log t))$. In order to take advantage of this result it is necessary to recast the algebraic problems in a form free of division.

Borodin, von zur Gathen and Hopcroft present constructions for determinants, characteristic polynomials of matrices and the GCD of univariate polynomials [**Borodin82**]. In addition, a probabilistic method is given for computing the rank of matrices. Von zur Gathen presents constructions for converting between various representations of rational functions [**Gathen83a**]. In another paper he gives a number of constructions for dealing with polynomials [**Gathen83b**]: an extended Euclidean algorithm, GCD and LCM of many polynomials, factoring polynomials over finite fields and square free decomposition of polynomials over fields of characteristic zero or finite fields. The algorithms are for univariate polynomials and are deterministic for certain fields (including the rational numbers and the reals). These constructions are based on a reduction to solving systems of linear equations. In a subsequent paper von zur Gathen treats the parallel powering of one integer modulo another [**Gathen84**] and constructions for Chinese remaindering and polynomial powering are given as secondary results. Kaltofen shows how multivariate polynomial GCDs can be constructed from straight-line programs [**Kaltofen85a**] and he conjectures that the irreducible factors of a polynomial given by a straight-line program can also be represented by straight-line programs [**Kaltofen85b**]. In another paper, he uses Padé approximants to show how to obtain a straight-line program for computing a rational function if degree bounds are known in advance for the numerator and denominator [**Kaltofen85c**]. Along slightly different lines, Kaltofen, Krishnamoorthy and Saunders give constructions for computing the Hermite and Smith normal forms for polynomial matrices [**Kaltofen85d**].

Parallel Algorithms

Sasaki and Kanada present parallel algorithms for symbolic determinants and linear systems [**Sasaki81**]. They observe the opportunity for parallelism in minor expansion and exploit this in computing determinants and in solving linear equations by Cramer's method. Their empirical study is based on a serial version of the determinant algorithm.

Collusion

It has been previously noted that the use of parallel processes can lead to a decrease in execution time by taking advantage of differing execution times of similar tasks. In a paper on heuristic search, Kornfeld remarks that a program for puzzle solving which is structured into parallel processes on a uniprocessor runs faster than the sequential version [**Kornfeld81**]. Subsequently, he expanded upon these observations and developed language primitives to take advantage of this phenomenon [**Kornfeld82**]. In this paper such algorithms are termed *combinatorially implosive*. Kornfeld's observations are qualitative and are based on extensive empirical evidence. In his thesis, Baudet observed that it was possible to take advantage of fluctuations in processor loads to obtain an average speedup of a chess game tree search [**Baudet78**].

1.5 Outline

This chapter has discussed the idea of using parallelism in computer algebra. Although there are several ways to examine parallelism, we have presented reasons why we restrict our view to the problem of utilizing a bounded number of processors for high-level parallelism.

Parallel processing can be used in many application areas and the requirements in each application vary. In Chapter 2 we discuss the issues specific to the use of parallelism in the field of computer algebra. If parallelism is to be used at a high level of abstraction we wish to formulate its usage in general paradigms that can be applied in many situations. We describe the so-called *problem heap* paradigm, in which all of the subproblems must be performed, and the *collusion* paradigm, where the completion of any single subtask is sufficient. The parallel algorithms described in this thesis can be formulated in terms of these paradigms. The problem heap paradigm has been used by others and its suitability for parallel processing is obvious. (The subproblems can be executed in parallel on separate processors.) The suitability of collusion is less obvious and we therefore examine it in greater detail.

Chapter 3 presents a mathematical model of collusion. We view the running time of a program as a random variable. In this framework we determine the relationship between the expected execution times of running several collusive tasks serially versus running them in parallel.

Each of the next three chapters presents a parallel algorithm for a specific problem. They are integer factorization, the calculation of polynomial greatest common divisors and the computation of Gröbner bases. These problems were chosen because of their general importance in computer algebra and because they seemed representative. They

were not chosen because they exhibited either any more or any less opportunity for parallelism than other problems in computer algebra.

Chapter 4 examines integer factorization. After briefly surveying existing methods we conclude that none of these methods can be expressed in parallel form except at a very low level. We then formulate a collusive solution to this problem as an example illustrating how collusion can be used to take advantage of high level parallelism in cases such as this. The key idea is that when factoring an integer the order of discovery of the factors is not important. In fact an integer factorization algorithm proceeds by attempting to find *any* factor and if one is found proceeding from there. We use collusion to take advantage of this.

Chapter 5 examines the computation of greatest common divisors. Many algorithms in computer algebra, such as rational function arithmetic and square free decomposition, pre-suppose the existence of a polynomial GCD algorithm. Because GCDs play a fundamental role in computer algebra systems, it is worth while to study parallel algorithms to compute them. We examine a parallel modular algorithm that takes advantage of sparseness of the input polynomials.

Chapter 6 examines the determination of Gröbner bases, a relatively new topic in computer algebra. They can be used to solve many of the traditional problems such as solution of equations, Hensel lifting, GCD computations and simplification. We examine how the problem heap paradigm can be applied to Buchberger's algorithm to obtain a Gröbner basis in parallel.

Chapter 7 begins by describing the selection of the parallel processing primitives and how they can be used to implement the high-level constructs we desire for computer algebra. Following this we discuss the steps that led to the development of our multiprocessing computer algebra system.

Chapter 8 describes the multiprocessing version of the Maple system upon which we implemented and tested our algorithms.

Finally, in Chapter 9, we conclude by summarizing our results and indicating possible directions for future work.

2.0 PARADIGMS FOR PARALLELISM IN COMPUTER ALGEBRA

2.1 Considerations for Computer Algebra

We begin by questioning whether the application of parallelism in computer algebra is different than in other areas of mathematical computation (such as numerical computation) and if so, why?

The traditional methods of exploiting parallelism in numeric mathematical computations rely on homogeneity of execution time. In computer algebra execution times can be very inhomogeneous so we must use more sophisticated methods to take advantage of parallelism.

The algorithms of computer algebra work upon a wide variety of objects. The information contained in a polynomial is much greater than the information contained in a single precision floating point number and the data structure for it is correspondingly more complex. Whereas all single precision floating point numbers are represented by the same layout of bit fields in a machine word, a sparse multivariate polynomial over the ring of two by two complex matrices could be represented naturally in a number of different ways. The usual notion of the *size* of an integer or the precision of a floating point number is given by its absolute value (or alternatively by its precision). Since the floating point numbers are of a fixed precision all additions will take the same time, likewise, each multiplication will take exactly as long as any other. Under such circumstances it is natural to talk about vectorized algorithms where a number of these operations all taking the same time are performed in parallel. This line of development has lead all the way to present day supercomputers which attain their speed in part by heavy reliance on vector operations.

Now on the other hand what can be said about operations on our polynomial in $GL\ (2,\ C)\ [x, y, z]$? There are many possible different measures of size for elements of this domain and algorithms which work upon these elements will have execution times which can vary considerably for inputs that appear to be very similar. The reason for this is that the latitude introduced by the extra information content allows for a subtle underlying mathematical structure. For example, in one computer algebra system (Maple 3.3 on a Vax 11/780) factoring the polynomial

$$f_1 = x^{64} - 3$$

over the integers takes 3 seconds and factoring

$$f_2 = x^{64} - 5$$

takes 2 seconds. This compares with over 200 seconds for the polynomial

$$f_3 = x^{64} - 4.$$

```
Administrator(nWorkers, severalProblems) ==
      for i in 1..nWorkers repeat
            spawnWorker()
      while not allDone() repeat
            waitForWorkRequestFromAnyWorker()
            if problemsRemain() then
                  giveProblemToWorker()
      return results
```

Figure 2.1: The Administrator Process

The only difference between these polynomials is a small change in one of the coefficients. In other problems, equally small changes can have an equally great impact while having a less directly obvious connection with the final form of the answer.

It is possible to characterize the classical methods in computer algebra as being algorithmic yet having execution times which can not always be predicted in advance.

2.2 Considerations for High Level Parallelism

If we are to exploit high level parallelism successfully, then we must be able to keep each one of a set of processors as busy as possible. Additionally we will want the algorithms to be described in such a way that they are independent of the number of processors. As stated earlier, the traditional parallel methods in mathematical computing may be inappropriate because of the variance in execution time. For example, in Wallach's scheme of alternating parallel dispatch with serialized synchronization points for algorithms in linear algebra, computations in a symbolic domain would leave many of the processors idle while waiting for the longest running subprocess to complete. [**Wallach82**]

2.3 The Problem Heap and the Administrator Process

One finds in the study of operating systems much discussion on how to structure parallel programs. Although these operating systems do not necessarily run on a multiprocessor, many of the concerns they address are similar to ours. A survey of the operating system

literature presents us with the concepts of a "problem heap" and an "Administrator" process.

To obtain parallel execution, a problem must be split into a number of tasks which can be performed simultaneously. Sometimes the splitting can be performed *statically* before execution. Often, static splitting is not possible and splitting of the tasks can only be determined *dynamically*. In this case there will be a number of (perhaps identical) processes cooperating in the solution of the problem. The data structure that contains the partial results and the parts of the problem that remain has been called a *problem heap* by Moller-Nielsen and Staunstrup [**Moller-Nielsen84**]. They assume this data structure is shared between the processes in common memory and is accessed via some mechanism such as monitors.

Often it is not desirable to presume the existence of shared memory. It is possible to extend the problem heap concept to a message passing environment. When message passing is used, the problem heap can belong to one process, with which the others communicate to obtain tasks. The concept of an *Administrator* process, as described by Gentleman, fills this role [**Gentleman81**]. The Administrator concept described in Gentleman's paper is based on the semantics of certain communication primitives used in a highly stylized way. Leaving the communication details until Chapter 7, we may express the idea of an Administrator process as follows. There is a certain amount of work to be done and there are a certain number of worker processes to perform the work. The Administrator begins by assigning a task to each worker process. It then sits and waits for its workers to finish these assigned tasks. Whenever a given worker finishes its job it communicates the result back to the Administrator. At this point the Administrator assigns a second task to that worker. This is continued until there exists no more work for the worker processes to perform. (This termination condition is usually not appropriate in operating systems.) A sketch of the Administrator process is given in Figure 2.1.

The Administrator process can be used as a generalization of the parallel begin/parallel end which is often found in the literature. It is appropriate to use it whenever there are several independent computations that can be performed in parallel.

2.4 Collusion and the Don Process

There does not always exist high level splitting to which the Administrator construct can be applied. This does not mean that parallelism must be foregone. Since execution times can be quite variable between *different methods* or even between the same method

```
Don(nWorkers, oneProblem) ==
      for i in 1..nWorkers repeat
             spawnWorker(variant(i, oneProblem))
      result := waitForResultFromAnyWorker()
      for i in 1..nWorkers-1 repeat
             terminateRemainingWorker()
      return result
```

Figure 2.2: The Don Process

on *different inputs*, we can use several processes to try to find the answer independently. This is the notion of *collusion* which is examined in detail in Chapter 3.

Just as the Administrator is the name given to a process which supervises a collection of worker processes, we shall say that a process which dispatches a collection of colluding processes is called a “Don”. When any one of the colluding processes yields a result the Don process may decide to terminate the others. This will be the case if the result returned makes the results of the others unnecessary. Sometimes, though, the results of the first n colluding processes, when combined, yield the result. A sketch of the Don process is shown in Figure 2.2.

The effectiveness of the Administrator construct and the Don construct depend on how many processors they can keep performing useful work at any given time. It is clear that the Administrator construct will be able to keep all processors busy if there are enough independent quantities to be computed. The effectiveness of the Don construct depends on how much of the work performed by the collusive processes is useful. This question is addressed in the next chapter, where a mathematical model of collusion is used to analyze the problem.

3.0 COLLUSION

In this chapter we develop a quantitative framework in order to understand how OR parallelism can be used to reduce execution times. In order to best understand the issues that pertain to OR parallelism, we restrict the investigation to its use on a single processor: How does running the tasks in parallel on a uniprocessor affect the execution time? Once this question has been addressed, then using a multiprocessor to exploit OR parallelism is not substantially different than using it to exploit AND parallelism.

3.1 Colluding Processes

In order to perform several tasks simultaneously on a single processor we use *time-slicing*. That is, a little bit of work is done on each task as the processor is switched rapidly between them. Rather than viewing the execution of a task as being constantly interrupted, suspended, and restarted, we may abstractly view the machine as simultaneously performing all the tasks, albeit each more slowly than if performed individually. We call the execution of a task a *process*.

The practical implementation of the process abstraction is usually an operating system function. However, it is possible for a single program to interlace the performance of several functions independently of an operating system. It is in the sense that logically distinct tasks are to be performed that we ask the questions we do.

Hoare has categorized the relationships between parallel processes based on the exchange of information between them [**Hoare76**]. We shall outline his classification here.

Disjoint processes are completely independent. They do not communicate and they do not share data. *Competing* processes also neither share data nor communicate, however, they do contend for resources such as disks and line printers. It is clear that on a uniprocessor these two forms of parallelism cannot lead to a speed up because the computations that must be performed for any task are independent of the results obtained by others. *Cooperating* processes are allowed to update common data but are not allowed to read it. Again, with this type of relationship between processes a certain amount of work must be done and the use of parallelism cannot possibly decrease the total execution time.

Communicating processes pass information between one another. This is done through shared variables which may be both updated and read or some other sort of message passing.

If several tasks are to be performed, it could be that the necessity of executing (or even completing, if execution has commenced) some of the tasks is determined by the results of other tasks. In this case, the order of execution will influence the total

amount of work that is done. For example, one process may tell the others that it has "the result" and they may terminate.

A particular case of this would be processes which follow alternate strategies for attaining a common goal. Hoare calls such processes *colluding*. Colluding processes work together toward a common goal; when one process succeeds in accomplishing that which was desired, all the processes are terminated. The colluding processes may communicate to share partial results but aside from this the work spent on the processes which do not "succeed" is wasted.

We shall call tasks *collusive* if they may be executed as colluding processes. If, in performing one of a group of collusive tasks, the common goal is attained by a particular task, then we say that task *succeeds*. If in the execution of a group of collusive tasks, a task terminates without having attained the common goal, then we say that task *fails*.

If colluding processes can be exploited to give a decrease in computation time, then this fact will be of practical import only if there are real problems which take advantage of collusion. Kornfeld has reported timings in which colluding processes more than doubled the speed in a particular heuristic search program [**Kornfeld81**]. We give three broad problem categories which use collusion in essentially different ways:

- problems in which there are alternate methods for attaining the common goal
- problems for which there are several equally acceptable solutions
- divide and conquer problems.

We shall discuss each of these classes in turn.

Alternate Methods

In this category, a problem has a single ultimate goal and there is more than one known method of achieving it. In many problems it is not possible to determine which alternate method is the least expensive for given data without performing a costly analysis. The cost of the analysis may well outweigh the savings gained from using the most economical method.

This situation can occur if the operation to be performed can be done cheaply using special methods for certain types of input. This is illustrated in the following example.

Example: We consider two methods of computing the GCD of a pair of polynomials, each of which is much cheaper than the other under particular circumstances.

The first method is a generalization of Euclid's method for computing the GCD of integers, the subresultant PRS algorithm (see, for example, [**Brown71b**]). If the GCD

of two polynomials is large, then this method finds it quickly, since only a few iterations are required. However, if the GCD of the polynomials is small, then this method becomes extremely costly due to the exponential growth of intermediate results.

The second method, the EZGCD finds the GCD of related polynomials in one variable and with coefficients in a finite field. From this GCD, the GCD of the original polynomials can be constructed [**Moses73**]. The cost of this construction depends on the size of the final GCD — the larger the GCD the greater the cost.

Comparing these two methods we find the first is less costly when the GCD is large and the second is less costly when the GCD is small. We may take advantage of collusion in the following way: To compute the GCD of two polynomials, two processes are used — one using each method. When either of the processes produces the GCD, the goal has been attained. (On a uniprocessor, it would be desireable for one of the methods to give up gracefully when it discovered a problem was not one of its good cases, otherwise the parallel algorithm could take twice as long as the quicker method.) □

Alternate Goals

Another class of problems well suited for collusion are those for which there are equally acceptable different solutions. An example of this would be to find a divisor of a large integer — any integer that exactly divides the given number is as good as any other. A more detailed example is the "satisficing search" problem which is analyzed later in this chapter.

Divide and Conquer

The *divide and conquer* approach is to divide a problem into subproblems, solve the subproblems, and combine the results [**Bentley80**]. In some divide and conquer algorithms the subproblems contribute different amounts toward the final solution, depending on the problem instance. This type of problem can utilize collusion in situations where not all the subproblems' results are needed to determine the final answer. In cases such as this, OR parallelism is used in conjunction with AND parallelism.

3.2 Execution Duration

The first step in building our mathematical model is to incorporate the expected execution duration for tasks. Exactly what do we mean by the "expected" duration? It is clear that any given program with particular input data will either require a certain fixed amount of execution time, if it terminates, or it will require an infinite amount

of execution time, if it does not. However, even when a task is guaranteed to halt, to determine the exact time needed for execution may be just as costly as performing the execution in the first place. Therefore, it is not reasonable to assume that, in practice, the execution time required can be known prior to performing the task. Rather than assigning to each task an exact assessment of the required time, we shall treat it as a random variable, based on the behavior of the program over many inputs.

Both in theoretical models of computation and in real machines, computation proceeds in discrete steps. To perform a serious computation takes very many basic machine operations. In view of this, we can simplify the calculations that arise in using our model by taking execution time to be a continuous, rather than a discrete, variable.

To each task we will assign a probability density for the execution time. The choice of the density will be based on the overall behavior of the algorithm for the domain of input. From this density, we get the expected execution time.

Example: Consider the following Pascal procedure:

```
procedure action (x: real);
var k : integer;
begin
        k := trunc(1000 * sin(x)) mod 100;
        if k <> 37 then
                subaction1(x);
        subaction2(x)
end
```

It can be seen that the routine *subaction1* is called for roughly ninety-nine out of every one hundred inputs, while the routine *subaction2* is always called. Suppose that the computation of k takes time T_k and that the routines *subaction1* and *subaction2* take times T_1 and T_2 respectively. Then roughly 1% of the valid inputs will take time $T_k + T_2$ and the remaining 99% will take time $T_K + T_1 + T_2$.

If the inputs to this routine are uniformly distributed in the input domain, then the probability density for execution time is

$$p(t) = .01\ \delta(T_k + T_2 - t) + .99\ \delta(T_k + T_1 + T_2 - t)\ \ .$$

Here δ is the Dirac delta function, defined by

$$\delta(x) = \frac{1}{2\pi}\int_{-\infty}^{\infty} e^{ikx} \mathrm{d}k$$

and having the properties

$$\delta(x - x') \quad = \quad 0, \qquad \text{if } \ x \neq x'$$

$$\int_b^a \delta(x - x')\, \mathrm{d}x' \;=\; \begin{cases} 0, & \text{if } x < a \text{ or } b < x \\ 1, & \text{if } a < x < b \end{cases}$$

□

As in the above example, it is sometimes possible that the input domain can easily be seen to be divided into a number of disjoint subsets where the time required for an instance of the problem depends only on the subset to which the input belongs. In fact, for *any* procedure the input domain may be partitioned based on the criterion of execution duration; there exists a partition of the input domain into classes such that all the elements (i.e. inputs) in a particular class require the same amount of execution time. As noted before, sufficiently analyzing a given element of the input domain to determine this membership may be exactly as costly as performing the operation. Instead of examining each input, we assign a weight to each of the classes in the partition. This gives the probability density for the execution time. The assignment of weights can be done either theoretically through analysis of the application, or empirically through simulation or collection of data on actual usage.

3.3 Parallelism

There is a broad range of possible degrees of parallelism in executing processes for a set of tasks. At one extreme, we could execute the tasks completely serially (no parallelism at all). Another possibility would be to give all the tasks an equal share of the available processing time (complete parallelism). In the general case, it must be decided for each time interval what portion of the processor time each process should receive. This budgeting of time can be done either statically, before execution begins, or dynamically, with the time allotments based on the processes' dynamic behavior.

In this section, for simplicity of the model, we shall use static time allotment. We do this by assigning a *time allotment function* $\nu_i(t)$, to each task $\mathbf{T}_i$. The function $\nu_i(t)$ has as a value the amount of processor time that the process for task $\mathbf{T}_i$ will have received after a total time t has passed. Suppose that the set of tasks to be executed is $\{\mathbf{T}_1, ..., \mathbf{T}_n\}$. The $\nu_i(t)$ may be any non-decreasing functions such that

$$\sum_{i=1}^{n} \nu_i(t) \leq t \tag{3.1}$$

and

$$\nu_i(0) = 0. \tag{3.2}$$

We allow the inequality in the definition so that overhead may be accounted for, if desired. So far, what we have said about the functions $\nu_i(t)$ allows the time variable to

be either discrete or continuous. In this thesis we are using a continuous variable for time. In this case, the derivative ${\nu'}_i(t)$ indicates the instantaneous proportion of the processor time which the process for task $\mathbf{T}_i$ is receiving at time t.

Example: If N tasks are all to receive an equal share of time, then we let

$$\nu_i(t) = \frac{t}{N} \,, \quad i = 1, ..., N.$$

□

Example: Suppose we have two tasks $\mathbf{T}_1$ and $\mathbf{T}_2$ which require times T_1 and T_2 to complete, respectively. If we execute task $\mathbf{T}_1$ and when it is done we execute task $\mathbf{T}_2$, then

$$\begin{aligned} \nu_1(t) &= \min(t, T_1) \\ \nu_2(t) &= \max(0, t - T_1) \,. \end{aligned}$$

□

3.4 Modelling Collusion

In our model of collusion we start with a set of tasks $\{\mathbf{T}_1, \mathbf{T}_2, ..., \mathbf{T}_n\}$. The execution of each of the tasks can result in one of three possibilities:

1. It succeeds; the execution terminates and the remaining tasks need not be executed (or completed if execution has commenced).

2. It fails; the execution of the task terminates and the remaining tasks are unaffected.

3. It does not halt.

To each task $\mathbf{T}_i$ we assign two probability density functions, $p_i(t)$ and $q_i(t)$. The density $p_i(t)$ gives the probability that the task will succeed when it has consumed a total time t. The density $q_i(t)$ gives the probability that the task $\mathbf{T}_i$ will fail when it has consumed a total time t. If the execution of task $\mathbf{T}_i$ halts, then

$$\int_0^\infty [p_i(t) + q_i(t)] \; \mathrm{d}t = 1.$$

3.5 Expected Execution Times

In this section we develop formulas for the expected completion times of groups of collusive tasks as we have modelled them. We first examine the time for serial execution of the tasks and then the time for parallel execution. After this, examples are given to compare the expected execution duration of serial versus parallel computations.

Collusive Tasks Executed Serially

We have a set of N collusive tasks that are to be executed one after another until one of them succeeds or until they all have failed. Let the tasks be labelled $\mathbf{T}_1, \mathbf{T}_2, ..., \mathbf{T}_N$, according to the order in which they would be executed if none were to succeed. Now for each task $\mathbf{T}_i$ let $p_i(t)$ and $q_i(t)$ denote the probability densities for success and failure, respectively, as a function of the time consumed by the process for the task.

We will now derive formulas for the probability densities, over time, for success of any task or failure of all tasks in the group. Let $p_{1..m}(t)$ denote the probability that one of the first m tasks (i.e. $\mathbf{T}_1, ..., \mathbf{T}_m$) succeeds when a total time t has been spent on all the tasks together. Let $q_{1..m}(t)$ denote the probability that the m-th task fails at time t.

By definition, we have

$$p_{1..1}(t) = p_1(t) \tag{3.3}$$
$$q_{1..1}(t) = q_1(t)\ . \tag{3.4}$$

One of the first m ($m \geq 2$) tasks succeeds at time t if either (i) one of the first $m-1$ of them succeeds at this time or (ii) all of the first $m-1$ tasks fail and the m-th task succeeds after consuming the remaining time to t. Thus,

$$p_{1..m}(t) = p_{1..m-1}(t) + \int_0^t q_{1..m-1}(t')\ p_m(t-t')\ \mathrm{d}t'. \tag{3.5}$$

If the m-th task fails at time t, the task $\mathbf{T}_{m-1}$ must have failed at some time prior to t. Then task $\mathbf{T}_m$ will have failed after consuming the time remaining to time t. We therefore have

$$q_{1..m}(t) = \int_0^t q_{1..m-1}(t')\ q_m(t-t')\ \mathrm{d}t'. \tag{3.6}$$

Taking Laplace transforms of (3.3) through (3.6), we obtain the expressions

$$\begin{aligned}
\tilde{p}_{1..1}(s) &= \tilde{p}_1(s) \\
\tilde{q}_{1..1}(s) &= \tilde{q}_1(s) \\
\tilde{p}_{1..m}(s) &= \tilde{p}_{1..m-1}(s) + \tilde{q}_{1..m-1}(s) \cdot \tilde{p}_m(s) \\
\tilde{q}_{1..m}(s) &= \tilde{q}_{1..m-1}(s) \cdot \tilde{q}_m(s),
\end{aligned} \tag{3.7}$$

where $\tilde{f}(s)$ denotes the Laplace transform of $f(t)$. Solving these recurrences and putting $m = N$, we have

$$\begin{aligned}\tilde{p}_{1..N}(s) &= \tilde{p}_1(s) + \tilde{q}_1(s) \cdot \tilde{p}_2(s) + \cdots + \tilde{q}_1(s) \cdots \tilde{q}_{N-1}(s) \cdot \tilde{p}_N(s) \qquad (3.8)\\ &= \sum_{i=1}^{N} \tilde{p}_i(s) \cdot \prod_{j=1}^{i-1} \tilde{q}_j\ (s)\\ \tilde{q}_{1..N}\ (s) &= \prod_{i=1}^{N} \tilde{q}_i\ (s). \qquad (3.9)\end{aligned}$$

Taking the inverse transform gives the probability densities for $p_{1..N}(t)$ and $q_{1..N}(t)$.

The performance of the N collusive tasks will be complete under either one of two mutually exclusive conditions: (i) one of them has succeeded or (ii) they have all failed. The expected execution time is therefore

$$\langle t \rangle_{\text{ser}} = \int_0^\infty t\ [p_{1..N}\ (t) + q_{1..N}\ (t)]\ \mathrm{d}t. \qquad (3.10)$$

Example: Suppose we have two tasks, $\mathbf{T}_1$ and $\mathbf{T}_2$, with

$$\begin{aligned}p_i(t) &= a_i\ \lambda_i\ e^{-\lambda_i\ t}\\ q_i(t) &= (1 - a_i)\ \lambda_i\ e^{-\lambda_i\ t}\end{aligned}$$

for $0\ \leq\ a_i\ \leq\ 1,\ \lambda_i > 0$. This gives

$$\begin{aligned}\tilde{p}_i(s) &= \frac{\lambda_i\ a_i}{s + \lambda_i}\\ \tilde{q}_i(s) &= \frac{\lambda_i(1 - a_i)}{s + \lambda_i}\end{aligned}$$

so that

$$\begin{aligned}\tilde{p}_{1..2}(s) &= \frac{\lambda_1\ a_1}{s + \lambda_1} + \frac{\lambda_1(1 - a_1)}{s + \lambda_1} \cdot \frac{\lambda_2\ a_2}{s + \lambda_2}\\ \tilde{q}_{1..2}(s) &= \lambda_1\lambda_2\frac{(1 - a_1)(1 - a_2)}{(s + \lambda_1)(s + \lambda_2)}\end{aligned}$$

and

$$p_{1..2}(t) + q_{1..2}(t) = a_1\lambda_1 e^{-\lambda_1 t} - (1 - a_1)\ \lambda_1\ \lambda_2\frac{e^{-\lambda_1 t} - e^{-\lambda_2 t}}{\lambda_1 - \lambda_2}\ .$$

Then the expected execution time is

$$\langle t \rangle_{\rm ser} = \int_0^\infty t \, [p_{1..2}(t) + q_{1..2}(t)] \, {\rm d}t = \frac{1}{\lambda_1} + \frac{1 - a_1}{\lambda_2}$$

□

Collusive Tasks Executed In Parallel

Here we have N collusive tasks $\mathbf{T}_1, \mathbf{T}_2, ..., \mathbf{T}_N$, which are to be performed in parallel. The execution of these colluding processes will continue until one of them succeeds or until all of them have failed. As for the serial case, let $p_i(t)$ and $q_i(t)$ denote the probability densities for success and failure respectively with respect to the amount of time consumed by the process for task $\mathbf{T}_i$. Let $\nu_i(t)$ denote the time allotment function for task $\mathbf{T}_i$.

Ignoring the other processes for now, the probability that the process for task $\mathbf{T}_i$ succeeds before a total time t is spent (on all processes) is given by

$$P_i(t) = \int_0^{\nu_i(t)} p_i(t') \, {\rm d}t' \; . \tag{3.11}$$

Similarly, ignoring the other processes, the probability that the process for task $\mathbf{T}_i$ fails by time t is

$$Q_i(t) = \int_0^{\nu_i(t)} q_i(t') \, {\rm d}t' \; . \tag{3.12}$$

We now derive formulas for the probability densities for the success of any process or the failure of all processes. Let $P_*(t)$ denote the probability of success in any of the processes by time t and let $Q_*(t)$ denote the probability of failure of all the processes by time t.

The probability that a success occurs by a given time is given by

$$P_*(t) = P_1(t) \; \cup \; P_2(t) \cup \; \cdots \; \cup \; P_N(t) \; . \tag{3.13}$$

Here, the associative operator $\cup$ is the inclusive *or*, defined to be $a + b - ab$. That is, if a and b are probabilities, then $a \cup b$ is the probability of a or b or both. It is simple to prove

$$\bigcup_{i=1}^{n} \; a_i = 1 - \prod_{i=1}^{n} (1 - a_i)$$

Therefore the probability density for success at time, t, $p_*(t)$, is

$$p_*(t) = \frac{d}{dt} P_*(t) = \frac{d}{dt} \left[1 - (1 - P_1(t)) \, (1 - P_2(t)) \cdots (1 - P_N(t))\right] \tag{3.14}$$

The probability that all tasks have failed by time t is

$$Q_*(t) = Q_1(t) \cdot Q_2(t) \cdots Q_N(t). \tag{3.15}$$

This directly gives us the probability density function for failure at time t, $q_*(t)$:

$$q_*(t) = \frac{d}{dt}\, Q_*(t) = \frac{d}{dt}\left[Q_1(t) \cdot Q_2(t) \cdots Q_N(t)\right]. \tag{3.16}$$

The execution of the N colluding processes will be complete if one of the processes succeeds or if all of them have failed. These two conditions are mutually exclusive. The probability density for execution completion is $p_*(t) + q_*(t)$. The expected execution duration is therefore given by

$$\langle t\rangle_{\text{par}} = \int_0^\infty t\ (p_*(t) + q_*(t))\ dt = \int_0^\infty t\ \frac{d}{dt}\ (P_*(t) + Q_*(t))\ \mathrm{d}t.$$

Integrating by parts, we obtain the expression

$$\langle t\rangle_{\text{par}} = \lim_{L\to\infty} t\ (P_*(t) + Q_*(t))\Big|_0^L - \int_0^L (P_*(t) + Q_*(t))\ \mathrm{d}t. \tag{3.17}$$

Example: Suppose again that we have two tasks, $\mathbf{T}_1$ and $\mathbf{T}_2$, with

$$p_i(t) = a_i\ \lambda_i\ e^{-\lambda_i\, t} \qquad q_i(t) = (1 - a_i)\ \lambda_i\ e^{-\lambda_i\, t}$$

for $0 \le a_i \le 1$, $\lambda > 0$. For both tasks let the time allotment function be $\nu_i(t) = t/2$. Then we have

$$P_i(t) = a_i(1 - e^{-\lambda_i\, t/2}) \quad Q_i(t) = (1 - a_i)\ (1 - e^{-\lambda_i\, t/2})$$

which implies

$$\begin{aligned} P_*(t) + Q_*(t) \quad &= \quad 1 - (1 - a_1)\ e^{-\lambda_2 t/2} - (1 - a_2)\ e^{-\lambda_1 t/2} \\ &\quad\ +(1 - a_1 - a_2)\ e^{t(\lambda_1+\lambda_2)/2} \end{aligned}$$

Using (3.17), the above expression yields

$$\langle t\rangle_{\text{par}} = 2\left[\ \frac{1 - a_2}{\lambda_1} + \frac{1 - a_1}{\lambda_2} - \frac{1 - (a_1 + a_2)}{\lambda_1 + \lambda_2}\ \right].$$

□

3.6 Example: Satisficing Search

Several types of search problems may be distinguished based on the aim of the search. In a *satisficing search* [**Simon75**] there is a set of items, a subset of which have some particular property, and the goal of the search is to find any element of the set with that property. An example is the search for a block of storage in a *first-fit* storage allocation algorithm. Several classes of satisficing search are treated in the literature. In this section we model the type of satisficing search in which the items may be examined in any order. This is known as *unrestricted satisficing search* or *satisficing search without order constraints*.

We can model satisficing search without order constraints in the following way: for each element (e_i) there is a probability (p_i) that the element has the goal property and there is a fixed time (t_i) required to examine the element. If we have N elements, then we have the tasks $\mathbf{T}_1 ... \mathbf{T}_N$ of examining the elements $e_1, ..., e_N$, respectively.

From the above description, we see that the density functions for the probabilities of success and of failure with the *i-th* task are

$$p_i(t) = p_i \; \delta(t - t_i) \tag{3.18}$$

$$q_i(t) = \bar{p}_i \; \delta(t - t_i) \tag{3.19}$$

where δ is the Dirac delta function and $\bar{p}_i = 1 - p_i$.

We now find the expected time for serial and parallel execution of the tasks $\mathbf{T}_1, ..., \mathbf{T}_N$. Because the time dependence is given by delta functions, it is quite possible to derive the expected times using discrete methods. However, to illustrate the use of the formulas derived in the previous sections we will use the more general method.

Serial Execution

Taking Laplace transforms of (3.18) and (3.19), we obtain

$$\tilde{p}_i(s) = p_i \; e^{-st_i} \qquad \tilde{q}_i(s) = \bar{p}_i \; e^{-st_i}.$$

Using (3.8) and (3.9), we therefore have

$$\begin{aligned}\tilde{p}_{1..N}(s) &= p_1 e^{-st_1} + \bar{p}_1 p_2 e^{-s(t_1+t_2)} + \cdots + \bar{p}_1 \cdots \bar{p}_{N-1} p_N e^{-s(t_1+\cdots+t_N)} \\ \tilde{q}_{1..N}(s) &= \bar{p}_1 \bar{p}_2 \cdots \bar{p}_N e^{-s(t_1+\cdots+t_N)}\end{aligned}$$

Taking the inverse Laplace transforms we find

$$\begin{aligned}p_{1..N}(t) &= p_1 \delta(t_1 - t) + \bar{p}_1 p_2 \delta(t_1 + t_2 - t) + \cdots + \bar{p}_1 \cdots \bar{p}_{N-1} p_N \delta(t_1 + \cdots + t_N - t) \\ q_{1..N}(t) &= \bar{p}_1 \bar{p}_2 \cdots \bar{p}_N \delta(t_1 + \cdots + t_N).\end{aligned}$$

The expected execution time is therefore

$$\begin{aligned}
\langle t \rangle_{\mathrm{ser}} &= \int_0^\infty t\,(p_{1..N}(t) + q_{1..N}(t))\,\mathrm{d}t \\
&= p_1 t_1 + \bar{p}_1 p_2 (t_1 + t_2) + \cdots + \bar{p}_1 \cdots \bar{p}_{N-1} p_N (t_1 + \cdots + t_N) \\
&\quad + \bar{p}_1 \cdots \bar{p}_N (t_1 + \cdots + t_N) \\
&= \sum_{i=1}^{N-1} \left[p_i \cdot \prod_{j=1}^{i-1} \bar{p}_j \cdot \sum_{j=1}^{i} t_j \right] + \prod_{j=1}^{N-1} \bar{p}_j \cdot \sum_{j=1}^{N} t_j \qquad (3.20)
\end{aligned}$$

Not surprisingly, the expected execution time depends on the values of the p's, the t's, and the order in which the tasks are executed. If we know the values for the p's and t's it is natural to ask in what order the tasks should be executed to minimize the expected processing time. This is known as the *least cost testing sequence* problem **[Price59]**.

To solve this problem, consider the effect of exchanging the order of two tasks, $\mathbf{T}_k$ and $\mathbf{T}_{k+1}$. In the original order we have

$$\langle t \rangle_{\mathrm{ser}_{k,k+1}} = A + \rho p_k (\tau + t_k) + \rho \bar{p}_k p_{k+1} (\tau + t_k + t_{k+1}) + B,$$

where

$$\begin{aligned}
A &= \sum_{i=1}^{k-1} \left(p_i \cdot \prod_{j=1}^{i-1} \bar{p}_j \cdot \sum_{j=1}^{i} t_j \right) \\
B &= \sum_{i=k+2}^{N-1} \left(p_i \cdot \prod_{j=1}^{i} \bar{p}_j \cdot \sum_{j=1}^{i} t_j \right) + \prod_{j=1}^{N-1} \bar{p}_j \cdot \sum_{j=1}^{N} t_i \\
\rho &= \prod_{i=1}^{k-1} \bar{p}_i \qquad \tau = \sum_{i=1}^{k-1} t_i.
\end{aligned}$$

With the interchange we have

$$\langle t \rangle_{\mathrm{ser}_{k+1,k}} = A + \rho p_{k+1} (\tau + t_{k+1}) + \rho \bar{p}_{k+1} p_k (\tau + t_{k+1} + t_k) + B$$

Therefore

$$\begin{aligned}
\langle t \rangle_{\mathrm{ser}_{k,k+1}} - \langle t \rangle_{\mathrm{ser}_{k+1,k}} &= \rho\, [p_k(\tau + t_k) - p_{k+1}(\tau + t_{k+1}) \\
&\quad + \bar{p}_k p_{k+1} (\tau + t_k + t_{k+1}) - \bar{p}_{k+1} p_k (\tau + t_{k+1} + t_k)] \\
&= \rho\, [p_{k+1} t_k - p_k t_{k+1}]
\end{aligned}$$

This shows that the task with the smaller value of t_i/p_i should be executed first. Since any permutation can be obtained from successive transpositions, the optimal order for sequentially executing the tasks will be $\mathbf{T}_1, \mathbf{T}_2, ..., \mathbf{T}_N$ when

$$\frac{t_1}{p_1} \le \frac{t_2}{p_2} \le \cdots \le \frac{t_N}{p_N}\,. \qquad (3.21)$$

If any of the ratios are in fact equal, then more than one ordering is optimal. This solution to the least cost testing sequence problem has been given by a number of authors, the earliest apparently being Mitten [**Mitten60**].

Parallel Execution

For simplicity, we shall use the time allotment function $\nu_i(t) = t/N$ for all processes. This is less than optimal, since once some of the tasks have failed there is more processor time available. However, taking advantage of this available time adds to the complexity of the analysis without significantly affecting the results.[1]

We label the tasks in such a way that

$$t_1 \le t_2 \le \cdots \le t_N \tag{3.22}$$

Now, using (3.18) in (3.11), we find that

$$P_i(t) = \int_0^{t/N} p_i \delta(u - t_i) \mathrm{d}u = p_i \mathbf{U}(t/N - t_i)$$

where $\mathbf{U}$ is the Heavyside unit step function. Similarly, using (3.19) in (3.12), we see

$$Q_i(t) = \bar{p}_i \mathbf{U}(t/N - t_i).$$

Therefore, we have

$$\begin{aligned} P_*(t) &= 1 - [1 - p_1 \mathbf{U}(t/N - t_1)] \cdots [1 - p_N \mathbf{U}(t/N - t_N)] \\ Q_*(t) &= \bar{p}_1 \cdots \bar{p}_N \mathbf{U}(t/N - t_N), \end{aligned}$$

the latter justified by (3.22). The expected execution time is

$$\langle t \rangle_{\mathrm{par}} = \lim_{L \to \infty} t(P_*(t) + Q_*(t))|_0^L - \int_0^L (P_*(t) + Q_*(t)) \mathrm{d}t.$$

When L exceeds Nt_N, we have

$$\begin{aligned} \langle t \rangle_{\mathrm{par}} &= \lim_{L \to \infty} L(1 - \bar{p}_1 \cdots \bar{p}_N + \bar{p}_1 \cdots \bar{p}_N) \\ &\quad - \int_0^{Nt_1} (1-1)\mathrm{d}t - \int_{Nt_1}^{Nt_2} (1 - \bar{p}_1)\mathrm{d}t - \int_{Nt_2}^{Nt_3} (1 - \bar{p}_1\bar{p}_2)\mathrm{d}t - \cdots \\ &\quad - \int_{Nt_{N-1}}^{Nt_N} (1 - \bar{p}_1 \cdots \bar{p}_{N-1})\mathrm{d}t - \int_{Nt_N}^{L} (1 - \bar{p}_1 \cdots \bar{p}_N + \bar{p}_1 \cdots \bar{p}_N)\mathrm{d}t \\ &= Nt_1 p_1 + Nt_2 \bar{p}_1 p_2 + \cdots + Nt_N \bar{p}_1 \cdots \bar{p}_{N-1} p_N + Nt_N \bar{p}_1 \cdots \bar{p}_N \end{aligned}$$

[1] In particular, we could use a model for time allocation where, after a process fails, the time formerly allocated to it is split equally amongst the remaining processes. Call this model M. We give the results for this model in footnotes for comparison.

This may be written as[2]

$$\langle t\rangle_{\text{par}} = \sum_{i=1}^{N-1}\left(p_i \cdot \prod_{j=1}^{i-1}\bar{p}_j \cdot Nt_i\right) + \prod_{j=1}^{N-1}\bar{p}_j \cdot Nt_N \tag{3.23}$$

Comparison

We now compare the expected execution duration for serial and parallel execution of a satisficing search. First, we demonstrate that a non-optimal ordering of serial execution can have an expected execution time *greater* than for parallel execution. Then we show that the optimal order for serial execution gives an expected execution time *less* than for parallel execution.[3]

We show that a non-optimal ordering of serial execution may be expected to require more time than a parallel execution by giving a simple example. Consider the case where $N = 2$ and $t_1 = t, t_2 = 10t, p_1 = p_2 = 39/40$. The expected time required for parallel execution is less than $5t/2$ while the expected time for serial execution is greater than $10t$ if task $\mathbf{T}_2$ is performed before $\mathbf{T}_1$.

To show that *optimal* serial execution is better than parallel execution, for this problem we assume that the time required for each task is greater than zero and that each task has a non-zero probability of success. Let the tasks be labelled $\mathbf{T}_1, \mathbf{T}_2, ..., \mathbf{T}_N$ so that $t_1 \leq t_2 \leq ... \leq t_N$. We define the following notation:

P_k = the expected time to execute $\mathbf{T}_1, ..., \mathbf{T}_k$ in parallel;
$\tilde{S}_k$ = the expected time to execute $\mathbf{T}_1, \mathbf{T}_2, \mathbf{T}_3, ..., \mathbf{T}_k$ serially and in that order;
S_k^0 = the expected time to execute $\mathbf{T}_1, ..., \mathbf{T}_k$ serially in an optimal order.

Since $S_k^0 \leq \tilde{S}_k$, we have

$$P_k - S_k^0 \geq P_k - \tilde{S}_k.$$

Therefore, we can prove P_N is greater than S_N^0 by showing $P_N - \tilde{S}_N > 0$. This we do by induction.

For the basis of the induction, consider the case when $N = 2$:

$$\begin{aligned} P_2 &= 2(t_1 p_1 + t_2 \bar{p}_1) \\ \tilde{S}_2 &= p_1 t_1 + \bar{p}_1(t_1 + t_2). \end{aligned}$$

Subtracting, we find

$$P_2 - \tilde{S}_2 = p_1 t_1 + \bar{p}_1(t_2 - t_1) > 0.$$

[2]For model M, replace Nt_k with $\sum_{i=1}^{k-1} t_j + (N-k+1)t_k$ in (3.23).
[3]Both these results still hold if parallel execution is based on model M.

For the inductive step, we note that

$$\begin{aligned} P_N &= \frac{N}{N-1} P_{N-1} + N \bar{p}_1 \cdots \bar{p}_{N-1}(t_N - t_{N-1}) \\ \tilde{S}_N &= \tilde{S}_{N-1} + t_N \bar{p}_1 \cdots \bar{p}_{N-1} \end{aligned}$$

Taking the difference and using 3.23, we see that

$$\begin{aligned} P_N - \tilde{S}_N &= (P_{N-1} - \tilde{S}_{N-1}) + \frac{P_{N-1}}{N-1} + \bar{p}_1 \cdots \bar{p}_{N-1}((N-1)t_N - Nt_{N-1}) \\ &> (P_{N-1} - \tilde{S}_{N-1}) + \bar{p}_1 \cdots \bar{p}_{N-1}\, t_{N-1} + \bar{p}_1 \cdots \bar{p}_{N-1}\, ((N-1)\, t_N - N\, t_{N-1}) \\ &= (P_{N-1} - \tilde{S}_{N-1}) + (N-1)\bar{p}_1 \cdots \bar{p}_{N-1}(t_N - t_{N-1}). \end{aligned}$$

So, for all $N \geq 2$,

$$P_N - S_N^0 \ \geq\ P_N - \tilde{S}_N \ >\ 0.$$

This completes the proof.

3.7 The Existence of Extremely Collusive Densities

In the previous sections we have shown how to determine the expected times for serial and parallel execution of collusive tasks running on a uniprocessor. The expressions derived in these sections are quite general, allowing for each task to have a distinct probability density. Quite often, however, we can expect the tasks to share the same probability density, while maintaining independence. This leads us to consider the following question:

Given a number of collusive tasks with a common probability density for execution time, can the expected time for performing the tasks in parallel be *less* than the expected time for performing the tasks serially, *even on a uniprocessor*?

We define an *extremely collusive density* to be one for which the expected time of two tasks using parallel execution on a single processor is less than for serial execution. In this section, we show that densities with this property do indeed exist.

We demonstrate the existence of extremely collusive densities by explicitly exhibiting an example.

Let us begin with a problem for which N identical tasks must be performed in the worst case. Call these tasks $\mathbf{T}_1 ... \mathbf{T}_N$. With each task, $\mathbf{T}_i$, we associate a probability (p_i) that it may find the solution to the overall problem and a time (t_i) which it would take to do this. We also associate with each task the time (T_i) that would be required to compute a partial result if it does not solve the entire problem. For each task, we

assume that $T_i > t_i$. :p. From this description, we see that the probability distribution functions for success and for "failure" are, respectively given by

$$\begin{aligned} p_i(t) &= p_i\delta(t - t_i) && (3.24) \\ q_i(t) &= \bar{p}_i\delta(t - T_i). && (3.25) \end{aligned}$$

In our analysis we will ignore the time required to determine the appropriate subproblems and to combine the results. This is not because the time is necessarily negligible, but because it is exactly the same regardless of whether the tasks are executed serially or in parallel.

We now find the expected time for serial and for parallel execution of the tasks. After this we compare the results for the special case when the tasks share the same distribution. Doing this we find that for certain ranges of p_i and t_i/T_i these densities are extremely collusive.

Serial Execution: The Laplace transforms of (3.24) an (3.25) are

$$\tilde{p}_i(s) = p_i e^{-st_i} \qquad \tilde{q}_i(s) = \bar{p}_i e^{-sT_i}.$$

Using the formulas (3.8) and (3.9), this gives us

$$\begin{aligned} \tilde{p}_{1..N}(s) &= p_1 e^{-st_1} + \bar{p}_1 p_2 e^{-s(T_1+t_2)} + \cdots + \bar{p}_1 \cdots \bar{p}_{N-1} p_N e^{-s(T_1+\cdots+T_{N-1}+t_N)} \\ \tilde{q}_{1..N}(s) &= \bar{p}_1 \cdots \bar{p}_N e^{-s(T_1+\cdots+T_N)}. \end{aligned}$$

Taking the inverse Laplace transforms yields

$$\begin{aligned} p_{1..N}(t) &= p_1\delta(t_1 - t) + \bar{p}_1 p_2 \delta(T_1 + t_2 - t) + \cdots \\ &\quad + \bar{p}_1 \cdots \bar{p}_{N-1} p_N \delta(T_1 + \cdots + T_{N-1} + t_N - t) \\ q_{1..N}(t) &= \bar{p}_1 ... \bar{p}_N \delta(T_1 + \cdots + T_N - t). \end{aligned}$$

This gives an expected execution time of

$$\begin{aligned} \langle t \rangle_{\text{ser}} &= p_1 t_1 + \bar{p}_1 p_2 (T_1 + t_2) + \cdots \\ &\quad + \bar{p}_1 \cdots \bar{p}_{N-1} p_N (T_1 + \cdots + T_{N-1} + t_N) \\ &\quad + \bar{p}_1 ... \bar{p}_N (T_1 + \cdots + T_N) \\ &= \sum_{i=1}^{N} \left[p_i \cdot \prod_{j=1}^{i-1} \bar{p}_j \cdot (t_j + \sum_{j=1}^{i-1} T_j) \right] + \prod_{j=1}^{N} \bar{p}_j \cdot \sum_{j=1}^{N} T_j. && (3.26) \end{aligned}$$

As before, it is natural to ask in what order the tasks should be executed to minimize the expected execution time. Using a method similar to that employed in section 3.6, we find that the optimal ordering of the tasks is to have

$$\phi(1) \leq \phi(2) \leq ... \leq \phi(N),$$

where

$$\phi(i) = (t_i - T_i) + \frac{T_i}{p_i}.$$

Parallel Execution: Again, for simplicity, we shall use the time allotment function $\nu_i(t) = t/N$ for all processes. Then, using (3.24) and (3.25) in (3.11) and (3.12), we have

$$\begin{aligned} P_i(t) &= p_i \mathbf{U}(\frac{t}{N} - t_i) \\ Q_i(t) &= \bar{p}_i \mathbf{U}(\frac{t}{N} - T_i) \end{aligned}$$

Without loss of generality, let $t_1 \leq t_2 \leq \cdots \leq t_N$. Also let

$$T_{MAX} = \max(T_1, ..., T_N).$$

Then we have

$$\begin{aligned} P_*(t) &= 1 - [1 - p_1 \mathbf{U}(\frac{t}{N} - t_1)] \cdots [1 - p_N \mathbf{U}(\frac{t}{N} - t_N)] \\ Q_*(t) &= \bar{p}_1 \cdots \bar{p}_N \mathbf{U}(\frac{t}{N} - T_{MAX}). \end{aligned}$$

Assuming $L > NT_{MAX}$, the expected execution time is

$$\begin{aligned} \langle t \rangle_{\text{par}} &= \lim_{L \to \infty} t\ (P_*(t) + Q_*(t))\ |_0^L \\ &\quad - \int_{NT_{MAX}}^{L} 1 \cdot \mathrm{d}t - \int_{Nt_N}^{NT_{MAX}} [P_*(t) + Q_*(t)]\mathrm{d}t \\ &\quad - \int_0^{Nt_1} P_*(t)\mathrm{d}t - \int_{Nt_1}^{Nt_2} P_*(t)\mathrm{d}t - \cdots - \int_{Nt_{N-1}}^{Nt_N} P_*(t)\mathrm{d}t \\ &= \lim_{L \to \infty} \quad L - [1]_{NT_{MAX}}^{L} - [1 - \bar{p}_1]_{Nt_1}^{Nt_2} - \cdots \\ &\quad - [\ 1 - \bar{p}_1 \cdots \bar{p}_{N-1}\ ]_{Nt_{N-1}}^{Nt_N} - [\ 1 - \bar{p}_1 \cdots \bar{p}_N\ ]_{Nt_N}^{NT_{MAX}} \\ &= N\ t_1\ p_1 + N\ t_2\ \bar{p}_1\ p_2 + \cdots + N\ t_N\ \bar{p}_1 \cdots \bar{p}_{N-1}\ p_N + NT_{MAX}\ \bar{p}_1 \cdots \bar{p}_N. \end{aligned}$$

This may be expressed as

$$\langle t \rangle_{\text{par}} = \sum_{i=1}^{N} \left(p_i \cdot \prod_{j=1}^{i-1} \bar{p}_j \cdot Nt_i \right) + \prod_{j=1}^{N} \bar{p}_j \cdot NT_{MAX}. \tag{3.27}$$

Comparison: We now compare the expected serial execution time to the expected parallel execution time for the special case of this example where

$$\begin{array}{lllllllll} p_1 & = & p_2 & = & \cdots & = & p_N & = & p \\ t_1 & = & t_2 & = & \cdots & = & t_N & = & t \\ T_1 & = & T_2 & = & \cdots & = & T_N & = & T. \end{array} \tag{3.28}$$

We show that even in this special case parallel execution may have a better expected execution time than serial execution.

Substituting from (3.28), the formulas (3.26) and (3.27) reduce to

$$\begin{aligned} \langle t \rangle_{\text{ser}} &= \frac{1-\bar{p}^N}{1-\bar{p}} \cdot [(1-\bar{p})\,t + \bar{p}\,T] \\ \langle t \rangle_{\text{par}} &= N \cdot \left[(1-\bar{p}^N)\,t + \bar{p}^N T\right]. \end{aligned}$$

Serial execution of the tasks is expected to take longer than parallel execution when the ratio

$$\frac{\langle t \rangle_{\text{ser}}}{\langle t \rangle_{\text{par}}} = \frac{1}{N} \cdot \frac{1-\bar{p}^N}{1-\bar{p}} \cdot \frac{(1-\bar{p})\,t + \bar{p}\,T}{(1-\bar{p}^N)\,t + \bar{p}^N\,T} \tag{3.29}$$

is greater than unity.

Examining this expression, we see that when p approaches zero, the value of the ratio approaches one and when p is one the value of the ratio is $1/N$, as would be expected. For small values of t/T, we find that as p increases from zero the value of the ratio increases from 1, reaches a maximum, decreases back past 1, and eventually reaches the minimum value of $1/N$ when $p = 1$. The cross-over point, where the value of the ratio is one, may be given in terms of $\bar{p}$:

$$\bar{p} = \frac{(N-1)\,t}{(N-1)\,t + T} + O\,(\bar{p}^N).$$

For example, when $T = 10t$ and $N \geq 5$, ignoring the $O(\bar{p}^N)$ term gives the value of p correct to within 1%.

Thus when t is small compared to T, the expected parallel time is less than the serial time when $\bar{p}$ is between 0 and $\sim (N-1)t/[(N-1)t+T]$. When $\bar{p}$ is between $\sim (N-1)t/[(N-1)t+T]$ and 1, the expected time for serial execution is the smaller.

To be very explicit, let

$$p(t) = \frac{1}{2}\delta(t-1)$$

$$q(t) = \frac{1}{2}\delta(t-4).$$

This is an extremely collusive density. We give graphs to show the ratio $\langle t \rangle_{\text{ser}}/\langle t \rangle_{\text{par}}$ as a function of $\bar{p}$ for various values of N and t/T. (See Figure 3.1.)

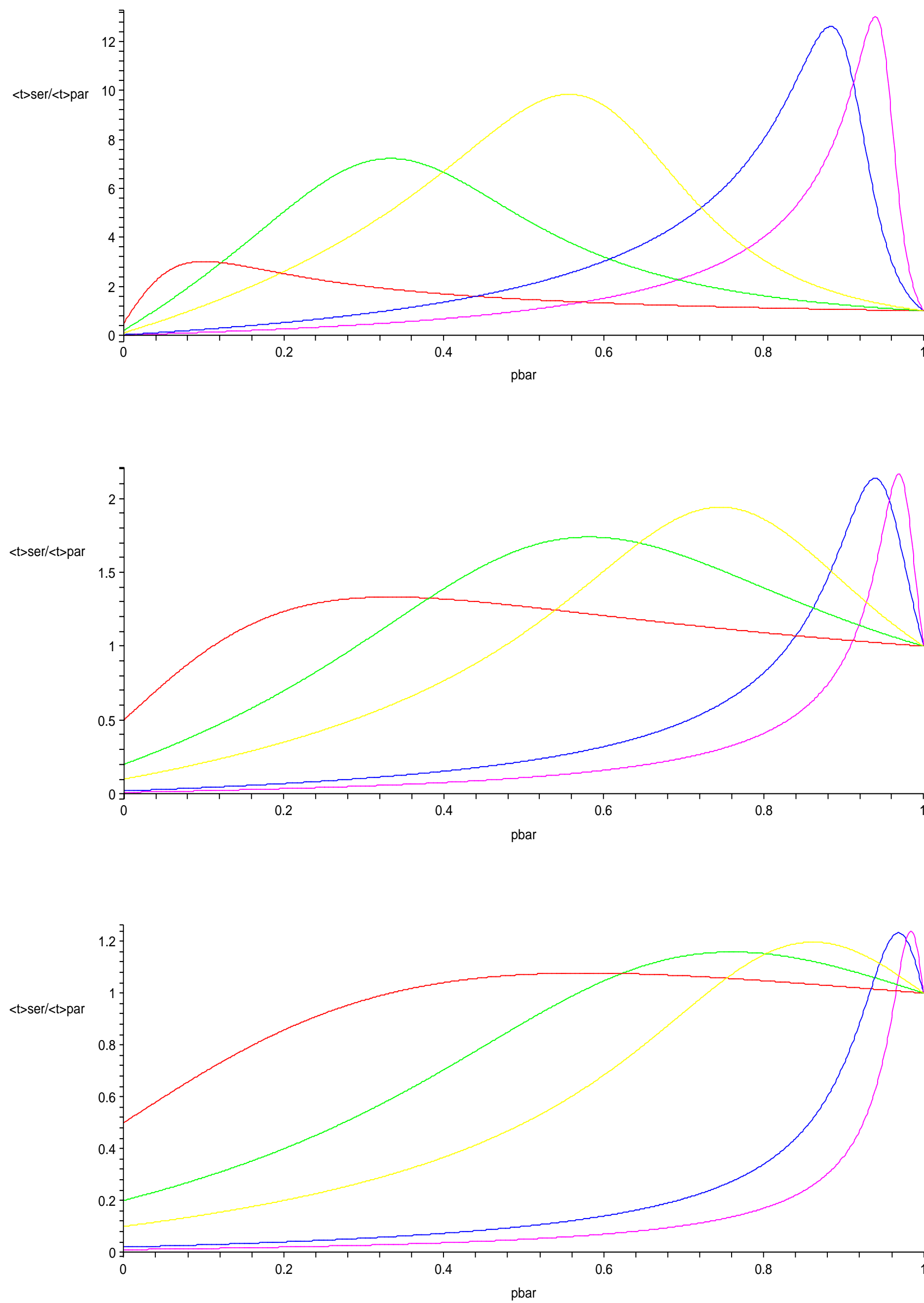

Figure 3.1: $\langle t \rangle_{\text{ser}} / \langle t \rangle_{\text{par}}$ for various values of $\bar{p}$, N and t/T.

4.0 INTEGER FACTORIZATION

4.1 The Role of Integer Factorization

The task of finding the prime factorization of a given integer has a long history. By the time of Gauss, integer factorization was already a well established problem. In fact, in his *Disquisitiones Arithmeticae*, he says that it is one of the most important and useful procedures in arithmetic. Integer factorization, and the associated task of primality testing have formed one of the cornerstones of number theory. Recently, there has been a renewed interest in the topic from the viewpoint of cryptography. The RSA cryptographic system makes use of the fact that factoring a large integer is a task of significant computational complexity. [**Rivest78**]

Integer factorization is a problem that properly belongs to the domain of number theory. The present state of the art is the factorization of arbitrary numbers of 70 to 75 digits, or of longer integers of special forms. To do this requires the use of a multiprecision arithmetic package with support for various number theoretic procedures. In many cases, the work on sophisticated factorization packages has been implemented in the form of a stand-alone program. On the other hand, a computer algebra system is a natural environment in which to pursue such work. Typically, in these systems, considerable effort has already been invested into implementing an efficient large integer package. For example, the decision of the Chudnovskys to use Scratchpad II in their work on integer factorization [**Chudnovsky85a**][**Chudnovsky85b**] was based on the efficiency and ease of use of big integer arithmetic.

Integer factorization certainly falls within the realm of computer algebra. In addition, users of a computer algebra system often require the use of a built-in integer factorization function. Once a reasonable factorization package is available, a computer algebra system can make use of it internally. An example of this is Maple's use of integer factorization in a single point evaluation heuristic which is used as an initial attempt in *polynomial* factorization.

In this chapter we survey the currently employed methods of integer factorization and show how certain of them may be used in a parallel setting.

4.2 A Brief Survey

This section provides a brief sketch of integer factorization methods with attention to those we have considered suitable for use with parallelism.

Trial Division

Perhaps the most obvious way of discovering factors of integers is by trial division. This is usually used as the very first step in multi-stage integer factorization codes. With a little effort it is possible to screen out numbers that are multiples of certain previous trial divisors so that not every integer is actually used in trial division.

GCDs with Highly Composite Numbers

Another strategy which is used as an early stage is to take the GCD of the number to be factored together with certain other numbers. These numbers are chosen to contain many distinct factors so that taking the GCD will yield the factors they have in common.

Legendre's Congruence

There are a number of integer factorization methods based on Legendre's congruence. If the number to be factored, N, can be written as $N = pq$ with $GCD(p, q) = 1$, then the congruence

$$x^2 \equiv a^2 \pmod{N}$$

has at least four solutions. These arise from the two combinations,

$$\begin{array}{llll} x \equiv \pm a \pmod{p} & \text{and} \quad x \equiv \pm a \pmod{q} & \Rightarrow & x \equiv \pm a \pmod{N} \\ x \equiv \mp a \pmod{p} & \text{and} \quad x \equiv \pm a \pmod{q} & \Rightarrow & x \equiv \pm b \pmod{N} \end{array}$$

The factorization methods use the fact that $(a^2 - b^2) = (a - b)(a + b)$ is congruent to zero (mod N) and so $a - b$ and $a + b$ separately divide N.

Pollard's Rho Method

Pollard's rho method finds factors by detecting cycles in a specially constructed sequence of integers. The sequence x_i is generated as iterates of a polynomial and is periodic (mod q) for some unknown factor q of the number N. The value of the factor is obtained as a a non-trivial GCD, $q = GCD(x_i - x_j, N)$, using a cycle-finding algorithm. Sometimes a non-trivial GCD cannot be found and the algorithm fails. **[Pollard75]** **[Brent81]**

Shank's SQUFOF

Shank's SQUFOF factorization is based on finding a square denominator in the continued fraction expansion of $\sqrt{N}$. The method is to computer the continued fraction

```
llfindfactor(n) ==
       if isPrime(n) then return n
       m := easyFactor(n)
       if m <> FAIL then return m

       for i from 1 to branchingFactor repeat
               spawn(serialFindFactor(smallPrime[i] * n))
       do
               m := waitForAnyResult()
               if not trivialFactor(m, n) then break
               i := i + 1
               giveWorkerNewProblem(smallPrime[i] * n)
       for i from 1 to branchingFactor - 1 do
               terminateWorker()
       return m
```

Figure 4.1: A Collusive Factor Finding Method

expansion until, on an even step, a square denominator is found, the square root of which has not yet occured as a denominator. [**Riesel85a**]

Morrison and Brillhart's Method

This method also searches for squares in the continued fraction expansion of $\sqrt{N}$, but attempts to construct a quadratic residue out of previously encountered square forms. [**Morrison75**]

Lenstra's Method

Lenstra's method finds a divisor of N by examining multiples of a point on an elliptic curve. One chooses a random elliptic curve E over $\mathbf{Z}/N\mathbf{Z}$ and a random rational point P on that curve. The multiples $n\ P,\ \ n = 2, 3, ...$ are computed until a suitable coordinate of $n\ P$ has a non-trivial GCD with N or until some cut-off is reached. [**Lenstra85**] [**Chudnovsky85a**]

```
llifactor(n) ==
      if isPrime(n) then return n

      compositeFactors := [n]
      primeFactors := []

      while compositeFactors <> [] do
              f := first(compositeFactors)
              compositeFactors := rest(compositeFactors))

              m := llfindFactor(f)

              for k in [m, f/m] do
                    if isPrime(k) then
                          primeFactors := [k, op(primeFactors)]
                    else
                          compositeFactors := [k, op(compositeFactors)]
      return primeFactors
```

Figure 4.2: A Parallel Factorization Algorithm

4.3 A Parallel Algorithm: llifactor

Having examined the existing algorithms for integer factorization we come to the conclusion that the level of parallelism inherent in these algorithms is on a low enough level that interprocess communication costs would become a dominant factor if the algorithms were implemented on a general purpose multiprocessor. To side-step this problem we use collusion.

For a particular integer, the different factorization algorithms can have very different execution times. One can take advantage of this fact by running the methods in parallel and taking the first answer. We call such an approach a *parallel polyalgorithm*. This method has a limited generality, however. No advantage can be obtained if there are more processors than algorithms. In this section a collusive integer factorization method is presented which can utilize any number of processors.

The first observation about integer factorization is that most methods proceed by discovering one factor at a time. When a factor is discovered it is divided out and, if

the result is not prime, the factorization process is begun anew on the result. Rather than waiting for a complete factorization by one of the colluding processes, as soon as one process discovers any factor all processes can be restarted on the deflated integer. Whether or not this is advantageous depends on how close to discovering a factor the other processes are expected to be. It is not necessarily wasteful to let them continue running because they may discover a *different* factor. In principal this decision could be made based on the size of the number to be factored and the size of the factor that has been discovered. However it is simplest to restart all processes right away.

The second observation to be made is that the factorization methods do not necessarily discover factors in any particular order by size. The same algorithm may discover a different factor or discover the same factor more quickly if presented with a *multiple* of the original number. This fact allows effective use to be made of additional processors, by attempting to factor several small multiples in parallel.

Combining the idea of taking one factor at a time with the idea of factoring multiples of the input yields the algorithm shown in Figure 4.1.[1] Here, ***serialfindfactor*** is a sequential program which returns a factor of its second argument. It uses its first argument to select a small prime multiplier which is good for the method. If desired, ***serialfindfactor*** can be used to select the *method* as well.

When a factor is found, it and its cofactor are tested for primality. If both are prime, the complete factorization has been obtained. If one of them is composite, then llfindfactor is reapplied. See Figure 4.2. If both were composite, one could imagine applying a divide and conquer strategy to factorize the composite factors in parallel.

4.4 Some Empirical Results

When examining various factorization methods for use in the framework we have described, it is necessary to consider the degree of collusion that can be expected. If a factorization method uniformly requires a certain execution time depending only on the length of the number to be factored, and if the factors are discovered in a particular order, then the method is not very useful in this context. That is if all workers are guaranteed to find the same factor, in the same time, then one might as well use a uniprocessor. The parallel algorithm exploits the *variance* in the time to find a factor and in the order in which the factors are discovered. If the distribution of execution times is extremely collusive, then we can expect a speedup greater than T/N when N processors are used. Even when the distribution of execution times is not extremely collusive, we can expect a significant speedup, say $3T/2N$. That we can get any speedup

[1] We use the “ll” prefix to indicate a parallel algorithm.

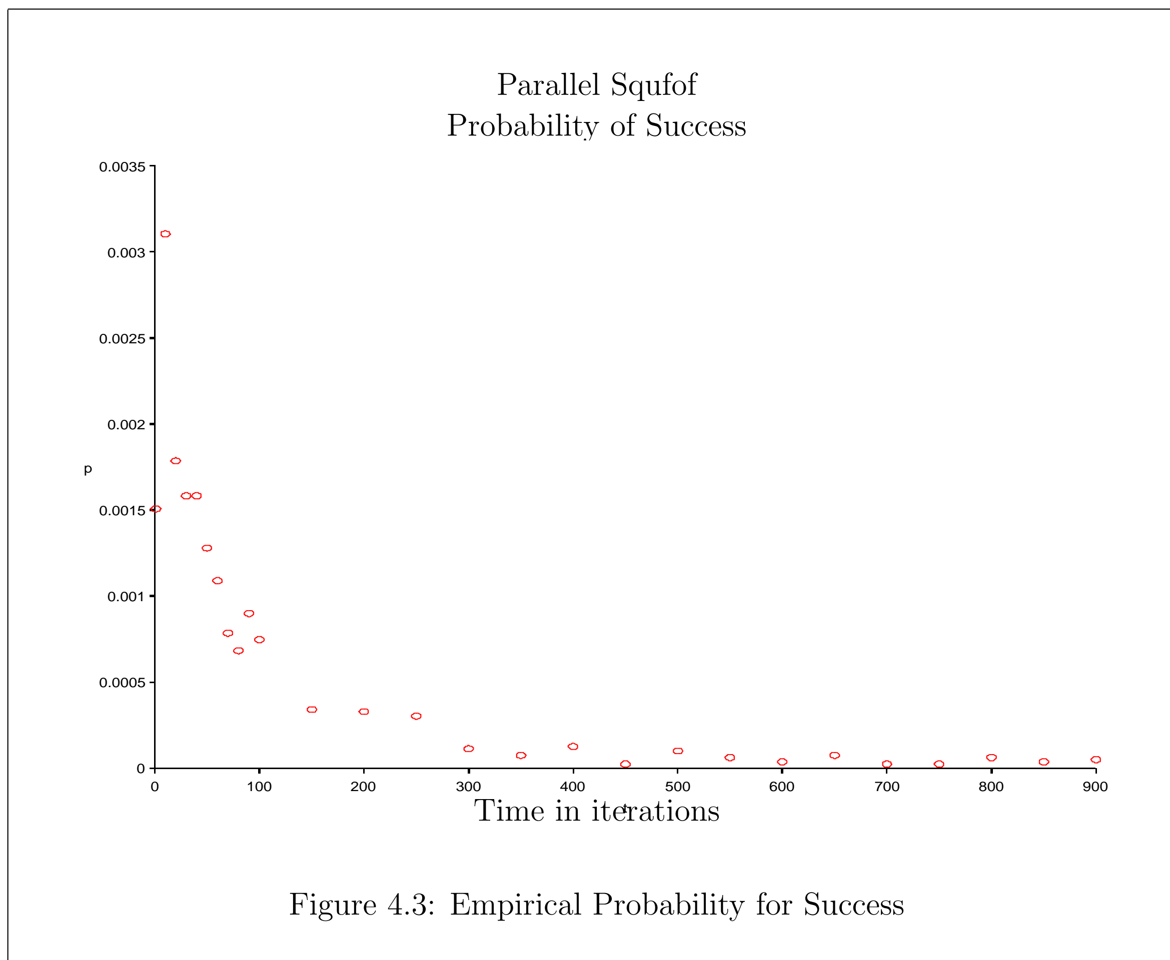

Figure 4.3: Empirical Probability for Success

at all is remarkable because we are not parallelizing the method used, but rather we are running several different versions of the serial method.

The methods which we selected for closer investigation were Pollard's ρ method and Shank's SQUFOF. Although the order of discovery of the factors in the ρ method is not very random, the running time has a wide variance. Since there is a correlation in the order of the discovery of the factors, it is better to vary the factorization by selecting different iteration polynomials and starting values for the iteration. However, it appears that the longer the ρ method runs, the more likely it is to discover a factor. This is the exact opposite of what we desire for use with collusion.

The second method we investigated was SQUFOF. This method has quite a wide variance in running time and the order of discovery of the factors is quite random. However, whereas certain statistics are known for the running time of the ρ method (e.g. the mean and variance have been derived **[Pollard75]** **[Brent80]**), analytically deriving

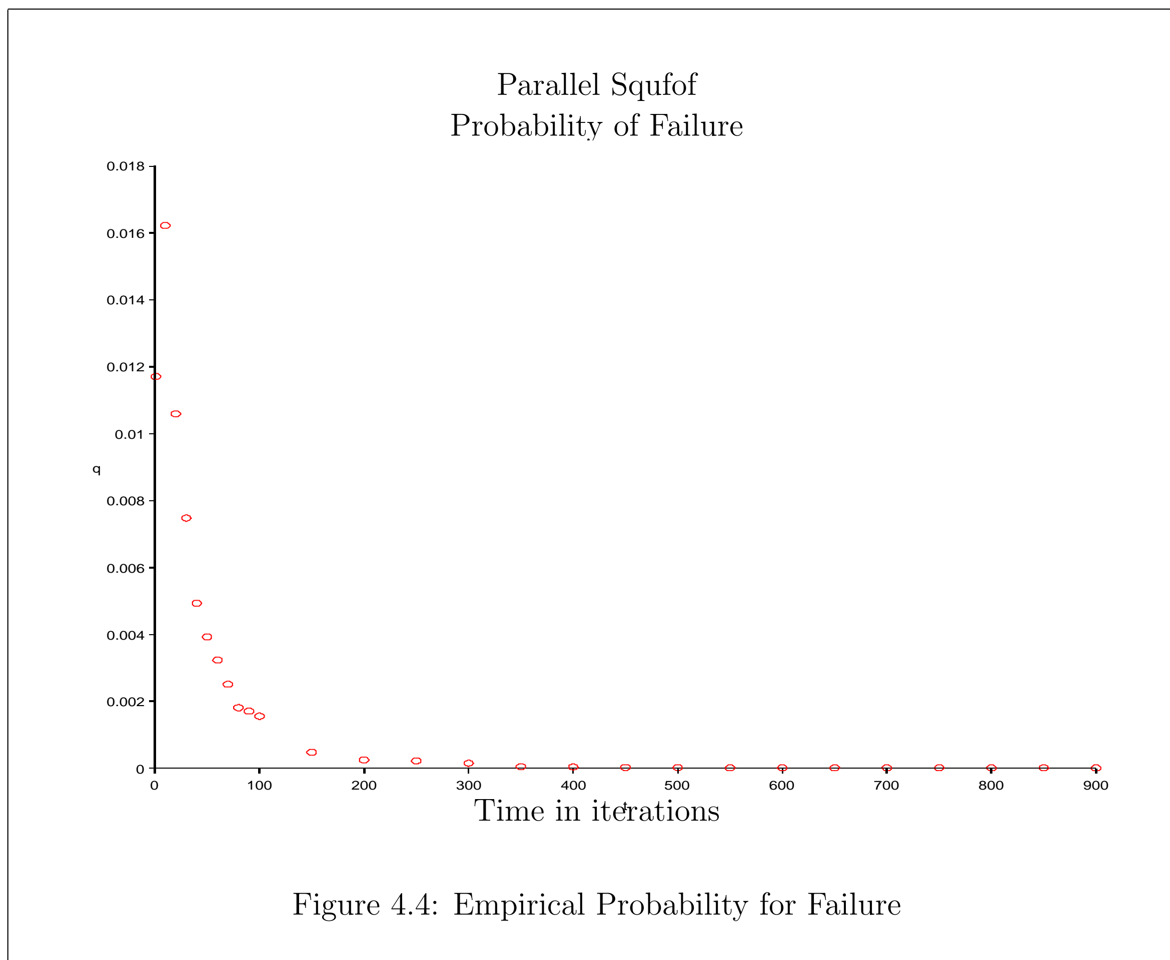

Figure 4.4: Empirical Probability for Failure

the statistics for SQUFOF's has not been accomplished. To investigate SQUFOF's suitability for collusion, we compiled an empirical distribution. To arrive at this distribution we measured the time required to find the first factor of four thousand random eight digit integers. As well as the first-factor time for the number itself, we recorded the first-factor time for the random number multiplied by each of the first fifty primes. The discovery of a proper factor of the random number was counted as a success, while a failure of the SQUFOF algorithm or the rediscovery of the prime multiplier was counted as a failure.

Analysis of the resulting data provided the empirical distributions for $p(t)$ and $q(t)$ shown in Figure 4.3 and Figure 4.4. Some numerical values for the distributions are given in Figure 4.5.

We can measure how good an algorithm is for collusion by the ratio of the expected serial time to the expected parallel time. It is this ratio that determines how close to a

T/N speedup can be obtained. For our sample data, the ratio of expected serial time to expected parallel time is

$$\frac{\langle t \rangle_{\text{ser}}}{\langle t \rangle_{\text{par}}} = 0.76$$

Incidentally, we note that parallel SQUFOF is expected to be better than serial SQUFOF roughly one third of the time on a uniprocessor.

t	lnt	p	-lnp	t / lnp	q	-lnq	t / lnq
1	0.00	.001508	6.49	.15	.011712	4.44	0.22
10	2.30	.003105	5.77	1.73	.016224	4.12	2.42
20	2.99	.001787	6.32	3.16	.010596	4.54	4.39
30	3.40	.001584	6.44	4.65	.007478	4.89	6.12
40	3.68	.001584	6.44	6.20	.004931	5.31	7.52
50	3.91	.001280	6.66	7.50	.003929	5.53	9.02
60	4.09	.001090	6.82	8.79	.003232	5.73	10.46
70	4.24	.000786	7.14	9.79	.002510	5.98	11.69
80	4.38	.000684	7.28	10.97	.001813	6.31	12.67
90	4.49	.000900	7.01	12.83	.001711	6.37	14.12
100	4.60	.000748	7.19	13.89	.001559	6.46	15.47
150	5.01	.000342	7.98	18.79	.000482	7.63	19.63
200	5.29	.000330	8.01	24.94	.000253	8.28	24.15
250	5.52	.000304	8.09	30.87	.000228	8.38	29.81
300	5.70	.000114	9.07	33.04	.000152	8.79	34.12
350	5.85	.000076	9.48	36.90	.000051	9.88	35.39
400	5.99	.000127	8.97	44.57	.000038	10.17	39.30
450	6.10	.000025	10.58	42.52	.000025	10.58	42.52
500	6.21	.000101	9.19	54.36	.000025	10.58	47.24
550	6.30	.000063	9.66	56.89	.000013	11.27	48.77
600	6.39	.000038	10.17	58.95	.000013	11.27	53.21
650	6.47	.000076	9.48	68.53	.000013	11.27	57.64
700	6.55	.000025	10.58	66.14	.000013	11.27	62.07
750	6.62	.000025	10.58	70.87	.000013	11.27	66.51
800	6.68	.000063	9.66	82.76	.000013	11.27	70.94
850	6.74	.000038	10.17	83.51	.000013	11.27	75.38
900	6.80	.000051	9.88	91.00	.000013	11.27	79.81

p = probability of success
q = probability of failure
t = time (iterations)

Figure 4.5: Tabulation of Probabilities at Certain Values

5.0 POLYNOMIAL GREATEST COMMON DIVISORS

5.1 The Role of Greatest Common Divisors

According to Knuth, the computation of the greatest common divisor (GCD) of two natural numbers is the oldest recorded non-trivial algorithm [**Knuth81**]. It is found in Book 7 of Euclid's *Elements* which dates to approximately 300 B.C.

GCD calculations can be related to a number of important mathematical problems. For example, quantities produced as by-products of Euclid's algorithm play a role in the mathematics of Sturm sequences, cylindrical algebraic decomposition and continued fractions.

In addition to its theoretic importance, the computation of GCDs is central to the operation of a computer algebra system One of the basic types of objects which a computer algebra system must manipulate is the rational functions of multivariate polynomials and rational function arithmetic requires the heavy use of GCDs. A computer algebra system which uses a canonical form for rational functions must perform GCD calculations in all rational function arithmetic operations. If a/b and c/d are two rational functions in canonical form, then computing their product requires $\gcd(a,\ d)$ and $\gcd(c,\ b)$. Their sum may be computed by extracting the single $\gcd(ad + bc, bd)$ but it is usually more advantageous to compute two related GCDs for polynomials of smaller degree. This is shown in Figure 5.1.

Besides their use in rational function arithmetic, GCDs are used in other algorithms in computer algebra systems. For example, they are used in square free decomposition which is a key step in factorization and integration. Davenport reports an application where up to 95% of the total time was being spent in GCD calculation [**Davenport81**].

5.2 A Survey of GCD Algorithms

The first algorithms for polynomial GCDs were generalizations of Euclid's algorithm to polynomial domains. These algorithms are called polynomial remainder sequence (PRS) methods, describing the sequence of intermediate results. In order to apply these methods to multivariate polynomials, the isomorphism $R[x_1, ..., x_n] \cong R[x_2, ..., x_n][x_1]$ is used to view polynomials recursively as univariate with coefficients in a smaller multivariate domain. Euclid's algorithm cannot be applied directly to this domain because the polynomial remainder operation is not defined if the coefficient domain is not a field. Rather than extending the coefficient domain to the rational functions $R\ (x_2, ..., x_n)$, the algorithm is modified to use a *pseudo-remainder* operation. However, in its simplest form the resulting method (the *Euclidean PRS* algorithm) is totally impractical because of explosive coefficient growth. A potential solution would be to divide out the

$$\frac{a}{b} \cdot \frac{c}{d} = \frac{(a \div u) \cdot (c \div v)}{(b \div v) \cdot (d \div u)}$$

where $u = \gcd(a,\ d)$, $v = \gcd(b,\ c)$.

$$\frac{a}{b} + \frac{c}{d} = \frac{(a\tilde{d} + \tilde{b}c) \div v}{(\tilde{b}d) \div v}$$

where $u = \gcd(b,\ d)$, $\tilde{b} = b \div u$, $\tilde{d} = d \div u$, $v = \gcd(a\tilde{d} + \tilde{b}c,\ \tilde{b}d)$.

Figure 5.1: GCDs in Rational Function Arithmetic

content of the pseudo-remainder at each step (the *primitive PRS* algorithm), however the recursive GCD calculations are very expensive. Rather then performing content calculations, it is more practical to divide out a smaller quantity which can be computed with less cost. The *reduced PRS* algorithm and the *subresultant PRS* algorithm are two such methods which were incorporated into several computer algebra systems. The main body of work on PRS algorithms is due to Collins. (See [**Collins67**].) Additional contributions were made by Brown and Traub [**Brown71b**]. Hearn considered the use of trial divisions to further improve PRS algorithms [**Hearn79**].

Independently, Brown and Collins developed a totally different approach to the computation of GCD's. In this method, known as the *modular GCD* algorithm, the GCDs of several homomorphic images are combined to construct the GCD of the input polynomials [**Brown71a**]. Shortly afterwords, Moses and Yun introduced the *EZGCD* algorithm which uses Hensel's lemma to compute the GCD using only a single homomorphic image [**Moses73**]. One negative feature of these algorithms is that they take no advantage of the sparseness of the input polynomials. Since multivariate polynomials tend in practice to be sparse, this can be a severe problem. Wang has introduced several improvements for Hensel lifting [**Wang78**]. Zippel has given probabilistic versions of the modular and EZGCD algorithms which can take advantage of sparse inputs [**Zippel79**], [**Zippel81**].

Recently two new methods for GCD calculation have been introduced. Char, Geddes and Gonnet have presented a fast heuristic GCD [**Char84**]. This method uses an evaluation homomorphism (which is usually invertible) to compute polynomial GCDs by applying an integer GCD operation to single point evaluation. An improvement to this algorithm is suggested by Davenport and Padget [**Davenport85**]. Gianni and Trager show how GCDs can be computed as the least degree member of a Gröbner basis for an ideal defined in terms of the input polynomials [**Gianni85**].

All of the above methods are sequential. The history of parallel methods is much briefer. Brown's Modular GCD algorithm was the first to exhibit a high degree of inherent parallelism. Davenport and Robert argue that this parallel structure makes this algorithm suitable for VLSI implementation [**Davenport84**].

The main results to date on parallel GCD computation are upper bounds for the complexity. Borodin, von zur Gathen and Hopcroft reduce univariate GCD calculation to the solution of linear systems [**Borodin82**]. They then show that an asymptotically fast parallel solution exists, using the construction of Valiant *et al* [**Valiant83**]. Kaltofen presents a straight-line GCD and a sparse conversion algorithm which can be used to formulate Zippel's sparse modular GCD as a random polynomial time algorithm [**Kaltofen85a**]. Valiant's construction can be used to parallelize this as well.

5.3 A Parallel Algorithm: llgcd

In designing a parallel GCD algorithm we wish to identify a method which can take advantage of the problem structure at a high level. It has been suggested by others that the modular GCD algorithm would be suitable as a starting point. In the modular GCD there is a very obvious way to take advantage of the problem structure. This algorithm decomposes the computation of multivariate GCDs over the integers into several GCD calculations in homomorphic image domains. The image GCDs are then combined to yield the final answer. Although this construction is ripe with parallelism it performs a lot of extra work in not taking advantage of sparsity. In VLSI, a number of simple processing elements is usually preferred over a larger, more complex processor, so a direct implementation of the modular GCD is more feasible than the implementation of a more sophisticated method which takes advantage of sparsity. Our goal, however, is to take advantage of a general multiprocessor.

Zippel's sparse modular GCD algorithm does take advantage of the sparse structure in constructing its result. At first glance it does not explicitly exhibit the same degree of inherent parallelism. Upon closer examination, however, we see that there is a rather simple way to express this method in a parallel form. We shall now describe the sparse modular algorithm in enough detail to see how it may be transformed.

The Sparse Modular GCD

The sparse modular methods introduced by Zippel are based on the observation that evaluating a polynomial at a random point will almost never yield zero if the point is chosen from a large enough set. The construction of the GCD in the sparse modular algorithm is an alternating sequence of dense and sparse interpolations. Here, the word "dense" is used to describe the usual notion of interpolation which is applied to $d + 1$ point-value pairs to uniquely construct a polynomial of degree d. A "sparse"

```
SparseModularGCD(p, q, goodTuple) ==
   d := boundGCDdegree(p, q)
   b := boundRandomPointSelection(p, q)
   vars := variablesOf(p, q)
   g := imageGCD(p, q, goodTuple)
   for i in 1..nops(vars) repeat
        # Introduce vars[i] by collecting d+1 points, r, and values, p.
        r [0] := op(i, goodTuple)
        p [0] := g
        for j in 1..d do
             # Obtain another polynomial by sparse interpolation.
             r [j] := random(b)
             L := {}
             while nops(L) ≠ nops(monomials(g)) repeat
                  tuple := randomTuple(b, i-1), r [j], goodTuple [i+1..]
                  L := L ∪ {g(tuple) = imageGCD(p, q, tuple)}
             coeffs := solve(L, monomials(g))
             p [j] := construct(coeffs, monomials(g))
        new := 0
        for x in monomials(g) repeat
             vals := map(coef, p, x)
             new := new + x * interpolate(vals, r, vars [i])
        g := new
   return g
```

Figure 5.2: The Sparse Modular GCD

interpolation takes a description of which coefficients are a non-zero and a number of point-value pairs equal to the number of non-zero coefficients. The coefficients are then determined by solving a linear system.

Example: Given that the only non-zero coefficients of $p(x)$ are those of x^4 and x^0, the two evaluations $p(2) = 7$ and $p(5) = -196$ imply

$$\begin{aligned} a_4(2)^4 + a_0(2)^0 &= 7 \\ a_4(5)^4 + a_0(5)^0 &= -196 \end{aligned}$$

Solving the system yields $p(x) = -\frac{1}{3}\,x^4 + \frac{37}{3}$

□

The sparse modular GCD determines the goal GCD, say $g(x, y, z)$, by recovering one variable at a time. Given bound d for the degree in each variable, the algorithm recovers x by interpolating the $d + 1$ images $g(x_0, y_0, z_0), ..., g(x_d, y_0, z_0)$ to give $g(x, y_0, z_0)$. This polynomial will have some number of terms, say t. Those terms which are zero for $x, y_0, ...z_0$ are assumed to be identically zero. The algorithm then performs d sparse interpolations to produce the images $g(x, y_1, z_0), ..., g(x, y_d, z_0)$. Dense interpolation yields $g(x, y, z_0)$. Again, d sparse interpolations provide the values for the dense interpolation to give $g(x, y, z)$. The algorithm is shown in Figure 5.2.

This algorithm is probabilistic. Each sparse interpolation is based on the highly probable assumption that coefficients which are zero in the one dense interpolation are identically zero. The probability that the computed GCD is incorrect can be made arbitrarily small by using a large enough range for the evaluation points. For details of the probabilistic argument see [**Zippel79**].

The Sparse Parallel Modular GCD

At each stage of the sparse modular algorithm, sparse interpolations must be performed. These sparse interpolations are independent and may be performed in parallel. Furthermore, each of these sparse interpolations requires the computation of a number of image GCDs equal to the number of terms in the last dense interpolation. The image GCD calculations are not very expensive on an individual basis, however. Each consists of (1) the generation of a random number, (2) two polynomial evaluations, and (3) an application of Euclid's algorithm to integers. If there are enough terms the image calculations can be grouped together to keep the overhead ratio down. Between one dense interpolation and the next, td image GCDs must be evaluated, and systems of t equations over the rationals must be solved. In the parallel algorithm these are solved as an inhomogeneous problem heap.

Next let us look at the problem of dense interpolation. Rather than performing the interpolation with the polynomials $p(x, y_0, z_0)$, $p(x, y_1, z_0)...p(x, y_n, z_0)$ as coefficients it is better to apply the interpolation algorithm to each integer coefficient and to add the results. These coefficient interpolations are independent and can be performed as a problem heap. Alternating the parallel pairs of dense and sparse interpolations yields the algorithm shown in Figure 5.3.

5.4 Analysis

We now investigate how effectively the parallel algorithm can make use of multiple processors. Although the algorithm takes direct advantage of the multiplicity of subproblems in both the sparse and dense interpolation stages, there are some inherent losses with this method that keep it from obtaining an ideal linear speedup. The first is

```
llGCD(p, q) ==
   d := boundGCDdegree(p, q)
   b := boundRandomPointSelection(p, q)
   goodTuple := findGoodTuple(p, q)
   vars := variablesOf(p, q)
   g := imageGCD(p, q, goodTuple)

   for i in 1..nops(vars) repeat
        # Introduce vars[i]
        r [0] := op(i, goodTuple)
        p [0] := g
        # This would be done with AllOf -- see section 8.2
        for i in 1..d lldo
             r [j] := random(b)
             p [i] := sparseInterpolation(p, q, g, i, r[j], goodTuple)
        new := 0
        for x in monomials(g) lldo
              vals := map (coeff, p, x)
              term := x * interpolate(vals, T, vars [i])
              new := new + term  # could return set
        g := new
   return g
```

Figure 5.3: A Parallel GCD Algorithm

that there can be an initial starvation phase in which there are not enough subproblems to keep all processors busy. The second loss comes from the discreteness of the division of the problem into subtasks. At each stage there are several interpolations which must be completed before the algorithm proceeds to the next step. Since the number of interpolations will not generally be divisible by the number of processors, at each stage some processors will be idle while the remainder of the interpolations are completed. In addition, there is the overhead of initiating the processes for the subtasks and the communication of the intermediate results. In this section we quantify these losses and see how closely a linear speedup may be approached.

We begin by noting that the amount of work to be performed at each stage of the algorithm is not constant. First of all, the number of terms increases as more variables

are introduced. This means that the size of the linear system increases and that more image GCDs are needed. Secondly, the degree bound for each of the variables will be different, so the number of sparse interpolations required is different at each stage.

Cost of One Stage

Let us examine a typical stage of the algorithm. Suppose that we are computing the GCD, g, of two polynomials p and q in $\mathbf{Z}\ [x_1, x_2, ..., x_n]$ and that we are about to introduce the variable, x_{k+1}. This means that we have already constructed the image of g with the variables $x_{k+1}, ..., x_n$ evaluated at some values, say $a_{k+1}, ..., a_n$. Let this image polynomial be denoted by g_k, where

$$g_k \in \mathbf{Z}\ [x_1, ..., x_k] \cong \mathbf{Z}[x_1, ..., x_n] \ / \ (x_{k+1} - a_{k+1}, ..., x_n - a_n)$$

Let t_k be the number of terms in g_k and let $d_1, ..., d_k$ denote the degrees of the variables $x_1, ..., x_k$ in g_k, and hence in g. In addition, let D_{k+1} be the degree bound for the variable x_{k+1} in g.

In order to introduce the variable x_{i+1}, we must perform D_{k+1} sparse interpolations. Each sparse interpolation consists of solving a system of t_k linear equations $Ax = b$. Setting up the system involves generating t_k random k-tuples, $\Lambda^{(k)}_{i,\ i=1,...,t_k}$ and evaluating each of the t_k monomials of g_k at each of these t_k points. This gives the matrix A. To obtain the right hand side, b, requires the computation of the integer GCD of p and q evaluated at $(\Lambda^{(k)}_i, a_{k+1}, ..., a_n)$ for $i = 1, ..., t_k$. If a linear congruential random number generator is used, then each random number requires 3 arithmetic operations for a total of $3kt_k$ operations. Each monomial may be evaluated with k powerings and $(k-1)$ multiplications. The number of arithmetic operations to set up the matrix is at most

$$(k-1)t_k^2(\log d_1 + ... + \log d_k) = (k-1)t_k^2 \log \prod_{i=1}^{k} d_k$$

Let T be the time to compute an image of g in $\mathbf{Z}$. Then to determine b requires time $t_k T$. Solving the system requires $2/3\ t_k^3 + O(t_k^2)$ operations. Thus the number of arithmetic operations for one sparse interpolation is

$$(T + 3k)\ t_k + (k-1)\ t_k^2 \log \prod_{i=1}^{k}\ d_k + \frac{t_k^3}{3} + O(t_k^2)$$

Next we perform a series of t_k dense interpolations. Each dense interpolation requires

$$\frac{7D_{k+1}^2 + 9D_{k+1} - 16}{2}$$

operations.

Thus the total number of arithmetic operations to introduce x_{i+1} is

$$D_{k+1}\left(t_k(\mathbf{T} + 3k) + (k-1)t_k^2 \log \prod_{i=1}^{k} d_k + t_k^3\right) + \frac{7}{2}\ D_{k+1}^2 t_k$$

plus lower order terms in t_k. If we have p processors, the time required is

$$\left\lceil \frac{D_{k+1}}{p} \right\rceil \left(t_k(\mathbf{T} + 3k) + (k-1)t_k^2 \log \prod_{i=1} kd_k + t_k^3 \right) + \left\lceil \frac{t_k}{p} \right\rceil \frac{7}{2} D_{k+1}^2$$

plus the communication costs.

Communication Costs

The communication costs associated with each sparse interpolation consist of the administrating process sending out a skeleton polynomial description of length t_k and the subordinate process returning a list of coefficients of the same length. If the communication of a numeric coefficient requires C times the time for an arithmetic operation, then the total is $2Ct_k$ for each sparse interpolation.

A dense interpolation requires the Administrator process to deliver D_{k+1} point/value pairs. In return, D_{k+1} coefficients are returned. The communication costs associated with the dense interpolation is $3CD_{k+1}$. Combining these, the total communication performed by the Administrator process requires time $5Ct_kD_{k+1}$. This cost is dominated by the cost of the arithmetic.

Worst and Best Case

Before we examine the expected degree of parallelism let us consider the best-case and worst-case behavior.

With extreme luck, t_k and D_{k+1} will be exactly divisible by p at each stage or be slightly *less* than an exact multiple. This would yield a nearly linear speedup.

On the other hand, if the GCD consists of a single constant term, then the algorithm degenerates to a serial one. This can be avoided by removing the integer content of p and q and performing an initial test for co-primality.

The Initial Starvation Phase

The algorithm computes the GCD by introducing one variable at a time. It starts with a single term with no variables and terminates with the full set of terms involving all of the variables. As each variable is introduced, the coefficients of the existing terms become polynomials in the new variable. This can split an existing term into several new terms, with the number of terms increasing throughout the course of the algorithm until the full number is reached. At stage k of the algorithm, t_k dense interpolations are performed. The value of t_k determines the number of processors that can be used.

Since the number of terms starts off small, there is less opportunity for parallelism in the initial stages.

In order to assess this initial starvation phase, it is necessary to determine the number of terms expected at each stage.

Expected Number of Terms

Before introducing the variable x_{k+1}, the image g_k of the goal polynomial has t_k out of a possible $\prod_{i=1}^{k}(d_i+1)$ terms. The expected value of t_k depends on the degrees $d_1, ..., d_n$ and the total number of terms t_n in the final result. Let us assume that the t_n terms of g are distributed randomly amongst the $\prod_{i=1}^{k}(d_i+1)$ possible terms in g_k. Then some terms of g_k will get exactly one term of g, other terms of g_k will get two or more terms of g, while the terms which do not appear in g_k (i.e. those which have zero coefficients) are those which did not get any. Each term of g_k can contain at most $\prod_{i=k+1}^{n}(d_i+1)$ of the t_n terms of g. Determining the expected number of non-zero terms in g_k is an occupancy problem.

The problem of finding the expected value of t_k can be expressed in the following way: Given b balls distributed randomly in N boxes, each of which can hold at most m balls, what is the expected number of empty boxes? (Given t_n terms of the goal polynomial distributed randomly in the $\prod_{i=1}^{k}(d_i+1)$ terms of the polynomial lifted to k variables, each term representing at most $\prod_{i=k+1}^{n}(d_i+1)$ terms of the goal polynomial, what is the expected number of zero terms of the polynomial lifted to k variables.) This is Romanovsky's box restraint problem. It is fairly straightforward to show that the probability that exactly s boxes are empty (that exactly s terms are zero) is

$$p(s) = \frac{\binom{N}{s}}{\binom{Nm}{b}} \sum_{j=0}^{N-s-b/m} (-1)^j \binom{N-s}{j} \binom{m(N-s-j)}{b}$$

The expected number of empty boxes is

$$N\frac{(Nm-b)^{\underline{m}}}{(Nm)^{m}} \simeq N\left(1-\frac{b}{Nm}\right)^{m} \qquad \text{for small } \frac{b}{N}.$$

Here, $x^{\underline{m}}$ denotes the descending factorial $x(x-1)\cdots(x-m+1)$. Details of the derivation of the expectation may be found, for example, in **[David62]**.

Example: Suppose the goal polynomial has 1000 terms and four variables of degrees 4, 29, 16 and 4. Then the expected number of non-zero terms at each stage is

$$\begin{aligned}
\langle t_1\rangle &\simeq 5\cdot\left[1-(1-\tfrac{1000}{5\cdot 30\cdot 17\cdot 5})^{30\cdot 17\cdot 5}\right] &\simeq 5\\
\langle t_2\rangle &\simeq 5\cdot 30\cdot\left[1-(1-\tfrac{1000}{5\cdot 30\cdot 17\cdot 5})^{17\cdot 5}\right] &\simeq 150\\
\langle t_3\rangle &\simeq 5\cdot 30\cdot 17\cdot\left[1-(1-\tfrac{1000}{5\cdot 30\cdot 17\cdot 5})^{5}\right] &\simeq 855\\
\langle t_4\rangle &\simeq 5\cdot 30\cdot 17\cdot 5\cdot\left[1-(1-\tfrac{1000}{5\cdot 30\cdot 17\cdot 5})\right] &\simeq 1000
\end{aligned}$$

Using the exact expression for the expected value shows $\langle t_3\rangle$ is actually closer to 856.
□

6.0 GRÖBNER BASES

6.1 The Role of Gröbner Bases

This chapter examines the computation of a particular kind of polynomial basis which is of interest in computer algebra. In his Ph.D. dissertation and subsequent work (see bibliography), Buchberger developed an algorithm to generate a particular type of basis for a multivariate polynomial ideal. He named these "Gröbner bases", after his thesis advisor Professor W. Gröbner.

Before giving a formal definition we shall briefly outline some of the uses of Gröbner bases in computer algebra. The reason that they are important in this field is that polynomial algorithms lie at the core of computer algebra systems and Buchberger's Gröbner basis algorithm is one of the few constructive tools in polynomial ideal theory.

One problem that can be solved using Gröbner bases is that of polynomial simplification. Simplifying a polynomial with respect to several side relations consists of determining the equivalence class of the polynomial modulo the ideal generated by the side relations. Given a Gröbner basis for this ideal, the simplification can be performed by a simple term rewriting procedure [**Buchberger82**].

Gröbner bases may be used to answer questions about the solvability of systems of polynomial equations [**Buchberger85b**]. They can be used to determine whether or not a system is consistent and, if so, whether the number of solutions is finite or infinite. If a system of equations has a finite number of solutions, then all solutions can be obtained by using Gröbner bases to "triangularize" the system.

Using Gröbner bases, certain of the classic problems in computer algebra can be solved in an elegant way. It has recently been shown how to do polynomial GCD's, factorization and Hensel lifting using Gröbner bases [**Gianni85**]. Although these constructions may not provide the most time-efficient solutions for these problems, they are compact and extremely interesting from a theoretical point of view.

6.2 Buchberger's Algorithm

Terminology

Let $R = K[x_1, ..., x_n]$ denote the ring of polynomials in n variables over the field K. A polynomial in R is a sum of terms where each term is the product of a *coefficient* in K and a *power product* $x_1^{i_1} \cdots x_n^{i_n}$.

Given a total order on the set of power products, the *head term* of a polynomial is defined to be the term with the power product which is greatest with respect to the order. Two commonly used orderings are *lexicographic order*, $<_L$, and *total degree order*, $<_T$, defined by

$$x_1^{i_1} \cdots x_n^{i_n} <_L x_1^{j_1} \cdots x_n^{j_n} \quad \Leftrightarrow \quad \exists l \ni i_l < j_l \text{ and } i_k = j_k \text{ for } k > l$$

$$x_1^{i_1} \cdots x_n^{i_n} <_T x_1^{j_1} \cdots x_n^{j_n} \quad \Leftrightarrow \quad \sum_{k=1}^{n} i_k < \sum_{k=1}^{n} j_k \text{ or}$$

$$\sum_{k=1}^{n} i_k = \sum_{k=1}^{n} j_k \text{ and } x_1^{i_1} \cdots x_n^{i_n} <_L x_1^{j_1} \cdots x_n^{j_n}$$

Example: In $K\ [x_1, x_2]$ the power products of degree two and less are ordered as follows:

$$1 <_L x_1 <_L x_1^2 <_L x_2 <_L x_1x_2 <_L x_2^2$$

$$1 <_T x_1 <_T x_2 <_T x_1^2 <_T x_1x_2 <_T x_2^2$$

□

Let f and g be two polynomials in R. We say that g is *reducible to* h *using* f, written $g \bullet\!\!>_f h$, if some term t in g is a non-zero multiple of the headterm of f and

$$h = g - \frac{t}{\text{headterm}(f)} \cdot f$$

Given a finite set of polynomials F, we say g *reduces to* h *with respect to* F, written $g \bullet\!\!>_F h$, if there exists $f \in F$ such that $g \bullet\!\!>_f h$. If there is no h such that $g \bullet\!\!>_F h$ then we say that g *is reduced with respect to* F.

If there is a sequence of reductions

$$g \bullet\!\!>_F k_1 \bullet\!\!>_F \cdots \bullet\!\!>_F k_n \bullet\!\!>_F h$$

where h is reduced with respect to F, then we say that h is a *normal form* of g and write $g \bullet\!\!>_F^* h$.

Gröbner Basis Definition

Gröbner bases may be defined in several ways. One intuitive definition is

Definition: The finite set F of polynomials in R is a *Gröbner basis* if and only if for all $f, g, h \in R$,

$$f \bullet\!\!>_F^* g \text{ and } f \bullet\!\!>_F^* h \quad \Rightarrow \quad g = h$$

That is, F is a Gröbner basis if and only if its normal forms are canonical.

Given F, we require some way of determining whether it is a Gröbner basis. This may be done using the notion of an S-polynomial. The S-polynomial of f and g, is defined to be

$$\frac{\text{headterm}(g)}{u} f - \frac{\text{headterm}(f)}{u} g$$

where u is the GCD of the head terms of f and g. (Sometimes the S-polynomial is defined to be a multiple of this.) The following result characterizes Gröbner bases.

Theorem [Buchberger79]:
F is a Gröbner basis if and only if for all pairs $(f, g) \in F \times F$,

$$S\text{-polynomial}(f, g) \bullet\!\!>^*_F 0$$

□

Buchberger's Basic Algorithm

Buchberger's algorithm is a completion procedure which takes a finite set F and adds elements until all pairs of elements have S-polynomials which reduce to zero. The elements added are exactly those S-polynomials which do not reduce to zero, taking advantage of the fact

$$g \bullet\!\!>^*_F h \;\Rightarrow\; g \bullet\!\!>^*_{F \cup \{h\}} 0$$

The basic form of Buchberger's algorithm is shown in Figure 6.1. In the simplest form, ***Criterion(f1, f2, G, B)*** is identically true so all pairs of polynomials are examined. This is correct but inefficient because after a certain point most of the reductions yield zero. One aspect of the work on Gröbner bases is centered on giving criteria which screen out these useless reductions. Buchberger gives an inexpensive but useful criterion[**Buchberger85b**]:

```
Criterion(f1, f2, G, B)  ==
    not Criterion1(f1, f2, G, B) and not Criterion2(f1, f2)

Criterion1(f1, f2, G, B) ==
    there exists p in G - {f1, f2} such that
        {p, f1} not in B and {p, f2} not in B and
        lcm(headterm(f1), headterm(f2)) is a multiple of headterm(p)

Criterion2(f1, f2) ==
    lcm(pp1, pp2) = pp1 * pp2  where
        pp1 := headterm(f1)/lcoef(f1); pp2 := headterm(f2)/lcoef(f2)
```

```
Gröbner(F) ==
     G := F
     B := {{f1, f2} | f1, f2 ∈  G, f1 ≠ f2}
     while B ≠ { } repeat
           {f1, f2} := choseElement B
           B := B - {f1, f2}
           if Criterion(f1, f2, G, B) then
               h := S-polynomial(f1, f2)
               h := normalForm(h, G)
               if h ≠ 0 then
                    G := G ∪ {h}
                    B := B ∪ {{g, h} | g∈G, g≠h}
     return G
```

Figure 6.1: Buchberger's Algorithm

Efficiency Considerations

There are several ways in which the basic algorithm has been improved. The first, as explained previously, is the application of a screening criterion to determine which pairs will reduce to zero without actually computing the S-polynomial or performing the reduction.

A second efficiency consideration concerns the selection of the pair to be used. Buchberger recommends selecting f_1 and f_2 in such a way that the least common multiple of the headterms is minimal with respect to the power-product ordering.

Whenever a polynomial is added to the basis, it may make possible the further reduction of some of those previously introduced. After adding a new polynomial it is usually desirable to reduce each element of the basis with respect to the others. This can cause several polynomials to be dropped and the resulting basis to be simpler. If G is a Gröbner basis and each element $g \in G$ is reduced with respect to $G - \{g\}$, then G is defined to be a *reduced Gröbner basis*. For any ideal, there is a unique reduced Gröbner basis. [**Buchberger85b**]

A Parallel Algorithm: ||Gröbner

In Buchberger's algorithm, the ideal basis is extended one element at a time. This construction has considerable flexibility in the order of selecting the pairs, in deciding

```
llGröbner(F) ==
    G' := { }
    G := F
    while G ≠ G' do
        G' := G
        B := {{f1, f2} | f1, f2 ∈ G, f1 ≠ f2}
        H := G
        forall {f1, f2} ∈ B lldo
            if Criterion(f1, f2, G, B) then
                h := S-polynomial(f1, f2)
                h := normalForm(h, G)
                if h ≠ 0 then H := H ∪ {h}
        G := {}
        forall h ∈ H do                                          |
            h := normalForm(h, H - {h})
            if h ≠ 0 then G := G ∪ {h}
  return G
```

Figure 6.2: A Parallel Gröbner Basis Algorithm

whether or not to reduce the basis, and if so when. In computing a reduced Gröbner basis several new basis elements can be computed at a time, before being added to the basis used to compute normal forms. This follows immediately from the uniqueness of reduced bases.

This observation allows us to formulate a parallel algorithm for computing Gröbner bases; rather than examining pairs of polynomials one at a time, several pairs can be examined simultaneously. Given $O(N^2)$ processors it would be possible to examine all pairs in one iteration. The resulting non-zero polynomials would then be adjoined to the basis. The resulting set would then be reduced before proceeding with the next iteration.[1] Such a parallel algorithm would terminate on the iteration when no new elements were generated.

[1] **ERRATUM [1986]:** Redundant elements may occur when treating S-polynomials in parallel. If these elements are simultaneously added to the basis, auto-reduction may drop them all. We did not originally take this into account. Figures 6.2 and 6.3 have been modifed to correct this, as indicated by the change bars.

```
llGröbner(F) ==
    G := F; H := F
    workers := SpawnWorkers()
    repeat
        Broadcast(WorkerMessage, [NewBasis, G], workers)

        # reduce all pairs
        B := {{f1, f2}|f1 ∈ G, f2 ∈ H, f1 ≠ f2}
        L := AllOf(WorkerMessage, [ReducePair, 'B[i]'], i=1..nops(B))

        # add new polynomials to basis; return if there are none
        H := convert(L, set) - {0}
        H':= { }                                                    |
        forall h in H do                                            |
            h := normalForm(h, H' - {h})                            |
            if h ≠ 0 then H' := H' ∪ {h}                            |
        H := H'                                                     |
        G := G ∪ H
        if H = { } then return G

        # auto-reduce the basis
        Broadcast(WorkerMessage, [NewBasis, G], workers)
        L := AllOf(WorkerMessage, [ReduceWithout, 'G[i]'], i=1..nops(G))
        G := convert(L, set) - {0}
```

Figure 6.3: A Parallel Gröbner Basis Using Administration

The basis reduction that is performed in each iteration is itself inherently parallel. Here, each of the basis elements is reduced to its normal form with respect to all the others, and the leading coefficient is divided through to make the result monic: Given a set F, the reduced forms

$$f \bullet\!\!>^{*}_{F-\{f\}} g$$

are computed for each $f \in F$. Since each of these reductions is completely independent of the others, the basis reduction is an ideal algorithm for parallel implementation.

These two components are combined to give the parallel Gröbner basis algorithm shown in Figure 6.2. In practice we do not have an arbitrarily large number of pro-

cessors and in many cases we will not be able to share directly the basis as it is being constructed.

In the implementation of the algorithm we use an Administrator to pass out the pairs and to coordinate the auto-reduction of the basis. The worker processes are capable of accepting three sorts of messages:

```
[ NewBasis, G ]
      Install G as the basis for normal

[ ReducePair, f1, f2 ]
      Return the reduced S-polynomial of f1 and f2

[ ReduceWithout, f ]
      Reduce f modulo the current basis without f.
```

In our implementation it is the worker process' responsibility to decide whether it will actually perform a critical pair reduction or immediately return 0 on the basis of some criterion.

A worker can also accept a sequence of these at once. Using these task descriptions as messages the implementation using the Administrator and message passing is shown in Figure 6.3. The precise meaning of *AllOf* and *Broadcast* are given in Chapter 8.

The desirability of broadcasting an entire basis twice per iteration depends on how tightly coupled the processors are. When the communication cost is high, an alternative is for the administrating process to maintain a record of which polynomials each worker has in its basis and to send updates. The update message tells the worker which polynomials to drop from its basis and contains new polynomials to include.

7.0 DEVELOPING A PARALLEL COMPUTER ALGEBRA SYSTEM

7.1 Chapter Overview

This chapter describes certain design considerations and the software prototypes that led to the development of a parallel computer algebra system. To begin, we discuss the issues in the choice of parallel processing primitives. We have an idea of certain high-level paradigms that are suitable for use in computer algebra, namely the Administrator construct and the Don construct. However, it is most unlikely that these will be sufficient for all needs. It is necessary to decide on a set of low-level primitives with which the high-level constructs can be implemented, and which can be used on the occasions when an escape hatch is necessary. This is the topic of section 7.2. In section 7.3 we show how these primitives are sufficient to implement the Administrator and Don constructs.

Section 7.4 discusses considerations for the use of multiple processes with shared memory and describes an experimental version of Maple in which data structures were shared between processes.

Section 7.5 describes an experimental system for running processes on multiple hosts. This system served as a prototype and provided early experience in running distributed computer algebra programs. The main consideration in the construction of the prototype was flexibility, with efficiency being a secondary consideration. The system consisted of two loosely connected components: a "message passing filter" and a collection of Maple functions.

Although the initial approach provided maximum flexibility, certain information on resource usage could only be obtained by modifying the Maple kernel. At this stage the opportunity was taken to directly incorporate multiprocessing facilities in a special version of Maple, creating a true multiprocessing computer algebra system. The external characteristics of this system are discussed in section 7.6. The discussion of the implementation details is left until Chapter 8.

7.2 Parallel Processing Primitives

There are many decisions that must be made in the selection of a set of primitives for writing parallel programs. Some of these are:

1. Are the processes fixed or dynamically created and destroyed?

2. Is it possible to share memory between processes on the same processor? On different processors?

3. Is process synchronization done with semaphores, monitors, message passing or some other method?
4. Is interprocess communication by message passing or shared data structures?
5. If message passing is used, are messages buffered by the system?
6. Do communicating processes block, and if so, what are the blocking semantics?
7. With which processes may a given process communicate?
8. If there is more than one processor, how is load balancing done?

For our purposes, the best combination was one that allows the widest range of experimentation. We chose a set of primitives to allow dynamic process creation and termination and to allow any process to communicate with any other.[1] Furthermore, we desired a set of primitives that could readily be implemented on a multiprocessor without requiring that the component processors be tightly coupled. Message passing generalizes most readily to multiprocessing, whereas schemes such as monitors, that require the use of shared memory (or its simulation), are more difficult to implement in a distributed environment. For this reason we chose a message passing scheme for interprocess communication.

The blocking behaviour of message passing primitives is one of the most significant design decisions in choosing a set of parallel processing primitives. First of all, the blocking behaviour must not inhibit the parallelism of programs. Secondly, how the message passing primitives block determines whether a hidden layer of message buffering is required. Finally, the choice of blocking behaviour directly determines the clarity of the conceptual model and understandability of programs.

The primitives we selected are described in the following paragraphs. In the exposition we talk about each primitive's "arguments" and "return value". The exact method of passing arguments and accepting the return value vary according to the parallel processing implementation. Also, exactly what constitutes a process id or a message varies depending on the context of the implementation.

[1]Unix pipes, for example, do *not* provide such a facility. In order for two processes to communicate, they must have a common ancestor, the pipe must have existed in that ancestor, and the ends of the pipe must have been passed down from the ancestor process to the processes that wish to communicate. That is, which processes a given process can communicate with are at least partially determined at the time it is created.

Process allocation and identification

Each process is identified by its *process id*. A process id is guaranteed to be unique within the set of processes ever used by the currently running parallel programs. A process id is used to refer to a process when using the parallel processing primitives.

The primitive for process allocation is *spawn*. It accepts one argument which determines what the new process is to execute. For example, depending on the context, this might be the name of a load module or of a Maple function. The value returned by *spawn* is the process id of the newly created process. In subsequent discussions we shall often call the process which does the *spawn* the "parent" and the spawned process the "child".

There is not a separate primitive for the parent process to ready the child for execution. The child process is started without any further intervention by the parent. If it is desired to create several child processes before letting any of them commence working, then the process synchronization primitives should be used. An example where this situation occurs is when each child must be set up to communicate with all the others.

The primitive for process termination and de-allocation is *kill*. It takes as an argument, the process id of the process to be terminated. To terminate itself, a process would pass its own process id as the argument.

Two primitives allow a process to find its own process id and the process id of its parent. The primitive *idself* takes no argument and returns the calling process' process id. The primitive *idparent* likewise takes no argument and returns the process id of the calling process' parent.

Interprocess communication

Interprocess communication is achieved by a message passing scheme. Depending on the implementation, processes may also be able to interact via shared memory.

The message passing primitives are *send, receive,* and *reply*. The semantics of these are taken from the Thoth operating system [**Cheriton79a**].

A process may communicate with any process for which it knows the process id. Since process ids may be sent as messages, this means that, in principle, any process can be made to talk to any other.

The *send* primitive takes as arguments (1) a process id, and (2) a message to be sent to the indicated process. The process doing the *send* blocks until the destination process accepts the message with *receive* and responds with *reply*. The returned value is normally a pair consisting of the process id of the replying process and the message sent as the reply.

The *receive* primitive takes as its single argument the process id from which to accept a message. This is called a *receive-specific*. If a null process id is given as the argument, then the first available message from any process will be accepted. This is called a *receive-any*. The value returned by the *receive* primitive is normally a pair consisting of the process id of the sending process and the message sent. If necessary, the receiving process blocks until a message is available from the indicated process.

The *reply* primitive takes as arguments (1) a process id, and (2) a message to be given as a response to the (blocked) sending process indicated by the first argument. The value returned by *reply* is normally the process id of the process which was unblocked. The *reply* primitive does not block since it is known that the sender is waiting.

The blocking *send*, blocking *receive* and non-blocking *reply* form the complete set of communication primitives. There are several possible variants on the blocking scheme described above. The most often used are various types of non-blocking *send*s and *receive*s. The proper use of the primitives we have chosen eliminates the need for other non-blocking primitives or any other process synchronization mechanisms [**Gentleman81**].

Note that the blocking semantics imply that messages need not be buffered when communication is between processes on processors sharing the same memory. For a *send-receive* rendezvous to transpire, one process will already be blocked and the other will have just issued the complementing request. At that time, the message can be copied directly from the sender to the recipient. Similarly, for a *reply* the sender is already blocked so the reply can be copied directly to the sender. In a loosely coupled multiprocessor, however, the sender and the receiver may be on different processors, in which case direct copying cannot be done.

Since processes may be terminated dynamically, it is possible that the process specified to a communication primitive might not exist any more. If the indicated process for a *send*, *receive* or *reply* does not (or ceases to) exist, then the primitive returns and the process id in the return value is null.

Process synchronization

The blocking semantics of the message passing scheme provide a mechanism for process synchronization. No other primitives are necessary.

For example, to provide mutually exclusive access to a resource a process would be set up to own the resource and perform all the operations upon it. (Notice the similarity to the intent of monitors.) Such a process is called a "proprietor". Any process that desired the use of that resource would *send* a message to the proprietor.

There are many possible implementations for the proprietor process. The simplest scheme would be to make the proprietor consist of a loop which accepts and services

```
Proprietor() ==
      Initialize()
      repeat
         (clientpid, request) := receive(ANY)
         result := PerformService(request)
         reply(clientpid, result);
```

Figure 7.1: Message Passing for the Proprietor

requests (see Figure 7.1). This form of proprietor implements mutual exclusion among processes wishing to access the resource. Of all the requests for operations upon a resource, only one request is acted upon at a time.

It is important to note that the receiving process determines when the sending process is unblocked. It would be possible to have a much more complicated proprietor by having it store up requests (received messages) and replying to them on a priority basis.

Load balancing

On a multiprocessor it is desirable to keep the load of the processors roughly evenly balanced. One way to achieve load balancing is to somehow make available information about the processor loads and to spawn processes on the least loaded processors. The *spawn* primitive would have an extra argument to indicate which processor should host the new process.

A simpler method of load balancing also spawns processes on particular hosts but does so without requiring information on processor loads. The idea is to specifically spawn worker processes on each processor and the processes on the least loaded hosts will complete their tasks first and request more work. (This idea is exploited by the Administrator construct.) For this reason, the process creation primitive we have chosen accepts an optional argument to specify the host. Information about host loads, however, is not provided. If *spawn* is used without the optional argument it should create the process on the least loaded processor (perhaps allowing a slight preference for local process creation because of reduced overhead).

```
Administrator() ==
       Initialize ()
       repeat
          (workerpid, result) := receive (ANY)
          request := use(result)
          reply (workerpid, request)
```

Figure 7.2: Message Passing for the Administrator

7.3 Message Passing in the Administrator and the Don

We now show how the low-level primitives can be used to implement the high level constructs we desire.

The Administrator

Gentleman describes the idea of an Administrator process and shows how to dispatch tasks to worker processes using the *send*, *receive* and *reply* primitives [**Gentleman81**]. The central idea behind the Administrator concept is that it avoids blocking by inverting the usual *send*/*receive*/*reply* cycle. The Administrator must not *send* work to its workers because that could cause it to become send-blocked, awaiting a *reply* from the worker. Instead, it *receive*s requests for work from its workers and issues work to them by *reply*ing.

The Administrator starts by *spawn*ing its worker processes. Each worker begins by *send*ing a request for work to its parent, the Administrator. The worker then remains send-blocked on its parent, the Administrator, waiting for work. The Administrator always uses the *receive-any* primitive. Each *receive-any* will yield either a request from the outside for the Administrator's services or it will be a request for work from one of its worker processes. See Figure 7.2.

The paramount consideration of the Administrator is that it never performs an action itself that would cause it to block when there is a work request. If the Administrator's job involved sending a message to another process, for example, it would not send the message itself because that could cause it to block. Instead it would delegate the task of sending the message to a worker. In our context, we use the Administrator for load balancing and it is this non-blocking property that guarantees that it will

```
Don() ==
    Initialize ()
    repeat
       (workerpid, msg) := receive (ANY)
       if (isNonFailureResult(msg)) then
          killRemainingWorkers()
          return msg
       else
          reply(workerpid, getNextTask())
```

Figure 7.3: Message Passing for the Don

always be able to receive work requests as they arrive from the more lightly loaded processors.

The Don

The message passing for the Don construct is similar to that of the Administrator. It also starts with one worker process per processor. Initially, the worker processes are reply-blocked awaiting tasks. The Don receives the requests and replies with tasks. The difference between the Don and the Administrator is that when the Don process receives the results of the completed tasks, it checks for a non-*FAIL* result. As soon as a non-*FAIL* result is obtained, the remaining active workers are *kill*ed and fresh worker processes are initiated in their place. See Figure 7.3 on page 65.

7.4 Shared Memory in Maple

One issue in parallel processing is whether the processes will be able to access common data in shared memory. For problems in many domains, this question is not of very great performance significance. The programs tend to operate on a small number of basic objects, which may be passed as messages or accessed as shared data structures. Typically the amount of data is small and whether or not the data is shared is not a performance issue. Alternatively, if the amount of data is large but not heavily used, an application consisting of multiple processes may share the data through a file.

In a computer algebra system, however, there will potentially be megabytes of heavily used data in main storage. There is clearly a potential performance gain possible in sharing common data. A chief area of gain is in saving the communication overhead of passing the data between processes. Although no shared memory multiprocessor capable of supporting a computer algebra system was available, experimenting with shared memory seemed worthwhile, if only to examine the semantics.

We call a group of processes that share the same address space a *team*, following the Thoth terminology [**Cheriton79**].hile it is not necessary that the entire address space be common to share data, this is the simplest case. On some systems it is possible to share only desired segments, allowing some degree of protection.

If a number of processes update common data then extreme care must be taken that the data is always left in a safe state. Critical sections must enclose updates to commonly operated upon data and the updates must make sense to all of the processes that use the data.

Maple's implementation allows these conditions to be guaranteed fairly easily. The implementation of Maple's basic simplification guarantees that any simplified expression has a unique instance in memory. Common sub-expressions are always shared. Since no reference counts are kept, all simplified expressions must be treated as read-only. If a modification to a data structure is required, it is a copy that is modified, not the original. (Table objects are the sole exception to these copy semantics.) From this point of view, Maple is an ideal computer algebra system to adapt for parallel processing with shared data.

None of the operating systems within Maple's domain of portability allows for processes with shared memory. In order to implement teams, it was necessary to add a layer of software on top of the operating system. This consisted of first implementing a coroutine package in C. (A detailed discussion of coroutines may be found elsewhere, e.g. [**Knuth73**].) Then, using coroutines, we implemented the ability to run a team of sub-processes that share the address space of a single (operating system) process. This functionality was then used to produce a special version of the Maple kernel that provided multiple processes to user-level Maple programs.

In practice, the read-only property of Maple's data structures worked out well and there were no major obstacles in getting the shared data version running. However, to create a shared data version of Maple for a *multiprocessor* with common memory, care would be needed in garbage collection.

7.5 A Multiprocessing Prototype

The next stage in the experimentation leading to the development of a parallel system was a prototype for running multiprocessing computer algebra programs. The multiprocessor on which the prototype runs is a local area network of Vax 11/780's running Berkeley Unix version 4.2 [**Leffler83**]. This program uses the Internet socket support for datagrams provided by this version of Unix. The fact that the multiprocessor is a local area network is transparent to user programs.

This prototype allows multiple processes to communicate with each other in a manner independent of the processor upon which they reside. In fact, a user program has to go to some effort to determine that its processes are in fact being run on separate processors.

The most obvious design for a multiprocessing computer algebra system at Waterloo is to produce a special version of the Maple kernel incorporating parallel processing primitives. However, for prototyping, it was desired that the system be as loosely coupled as possible to give maximum flexibility. If the multiprocessing facilities could be provided completely externally to the kernel, then experiments could be performed more rapidly.

Having the multiprocessing facilities packaged in a separate program gives an additional bonus. It means that the facilities can be made to be language independent. The individual processes would be programs written in C, Maple, Macsyma, Fortran, Prolog, or any other language. A parallel program would be able to have the processes written in whatever language was best suited. The same multiprocessing tool could be used to write distributed Macsyma programs as to write distributed Maple programs. In fact, the processes need not be programs written in the same language. For example, if one was to write a program that used both asymptotic analysis and Laplace transforms, then one could use Maple for the asymptotic analysis and Macsyma for the Laplace transforms. If a distributed Maple program had the need of an "inference server" then that portion of the code could be written in Prolog. To do this by installing corresponding multiprocessing facilities in each interpreter would be impossible.

These considerations led to the implementation of such a program. The program was called *mpf* for "message passing filter". [**Watt85**]

The Message Passing Filter as an Agent

The *message passing filter* is a program which allows another program to participate as a member of a distributed system. To each user process, there corresponds a process which is an instance of the mpf program. This mpf process acts as an agent for the user process and performs all of that processes' interprocess communication. It also

allows the user process to spawn new processes, kill other processes and perform several miscellaneous functions. In the remainder of this section we describe the facilities provided by the message passing filter.

A parallel program using the mpf consists of several user processes, each with an mpf process acting as an agent. When the user process is created its standard output is redirected to go to the mpf process. Likewise, its standard input unit is received from the mpf process. The user process sends messages to other user processes and issues various commands by writing on its standard output.

The output from the user process is interpreted on a line-by-line basis by the agent process. Lines beginning with a user-definable prefix (usually "`#] `") are taken to be commands. Other lines are taken to be normal output. Messages that are deliverable to the user process are given to it via its standard input. They cannot be confused with normal input because the user process must make a special request to obtain a message.

Commands

Lines beginning with the specified prefix are taken to be *mpf* commands. The default prefix is `"#] "`. The delivery of messages uses printf(format, id, message). The default format is `"%s: %s\n"`. An erroneous call is indicated by a return of *idnull*.

The available *mpf* commands are:

```
Command form                            Id and  message returned
--------------------------------------- -----------------------------------
<prefix> debug                          idself   "Debug on." | "Debug off."
<prefix> spawn <host | any> <command>   idchild  "Spawned."
<prefix> kill  <mpfid>                  idvictim "Killed."
<prefix> send  <mpfid> <message>        fromid   <message>
<prefix> receive <mpfid | any>          fromid   <message>
<prefix> reply <mpfid> <message>        recipid  "Replied."
<prefix> idself                         idself   "Identification."
<prefix> idparent                       idparent "Identification."
<prefix> idnull                         idnull   "Identification."
<prefix> quit                           N/A
<prefix> <other>                        idnull   "Invalid command."
```

Here `<message>` must be one token. In particular, a quoted string can be used. The `<command>` may be many tokens.

The meaning of these commands is described below.

Each user process is assigned a process id which is guaranteed to be unique within all currently executing mpf programs. Three process ids are available via *mpf* commands. A process may find out its own process id, that of its parent, and the null process id.

Two formats that are useful in Maple are:

```
"#%s: \n%s\n"
```

and

```
"[`%s`, %s];\n"
```

To create a new process, the following is used:

```
<prefix> spawn <command>
```

All text after the keyword spawn is taken to be a Unix shell command. The command is executed on the available host with the lowest load. A new user process with its own agent is created on the selected host. The process id of the spawned process is placed as a message on the standard input of the process issuing the spawn command. The message delivery format is used, and the process id in it is that of the spawned process. The body of the message is the text "Spawned".

The available hosts are those which are currently "up". They are listed in the *.rhosts* file in the user's home directory. The login name to use on the different hosts is also determined from the .rhosts file.

```
<prefix> kill <process id>
```

This command kills the specified process.

A process' own id, its parent's id and the null id may be obtained via the following commands:

```
<prefix> idself
<prefix> idparent
<prefix> idnull
```

In all three cases the return value is placed as a line on the standard input according to the specified format, with the process id being that requested and the body of the message being the text “Identification”.

The following are used for interprocess communication and synchronization:

```
<prefix> send    <process id> <message>
<prefix> receive <process id>
<prefix> reply   <process id> <message>
```

The values are returned as the next input line to the invoking process.

Writing Maple Programs

A set of Maple procedures was written to provide an interface with the message passing filter. The separation of the multiprocessing facilities from the Maple kernel allowed a number of different ideas to be tried before converging on the following. To pass Maple expressions between processes a two-level message passing scheme was developed. At the first level, the messages exchanged were file names. At the second level, written in Maple, the message expression would be saved in a “.m” file and a Unix command would be used to transfer the file to the host of the destination process where it could be read. A similar two-level scheme was used to provide the spawning of Maple processes running selected Maple functions. The first action of a newly spawned Maple process was to receive a file from its parent containing the expression to be evaluated.

Although the initial approach provided flexibility, information on resource usage could not easily be obtained, especially the statistics on the time and space used by a process killed by a Don. This information could only be obtained by modifying the Maple kernel to trap an interrupt.

7.6 A Multiprocessing Version of Maple

At this stage the opportunity was taken to directly incorporate multiprocessing facilities in a special version of Maple, creating a true multiprocessing computer algebra system. Although quite inefficient in the form used in the prototype, the two-level methods developed there suggested the ideas used in this multiprocessing version of the Maple kernel. In this section we describe the external features of the system. A discussion of the implementation is left until Chapter 8.

The Maple functions which allow the writing of multiprocessing programs are *spawn*, *kill*, *send*, *receive*, *reply*, *idself*, *idparent*, *idnull*, *hosts* and *stats*. The calling sequences of these functions are described below.

```
processid := spawn('expr');
processid := spawn('expr', hostname);
```

The *spawn* function takes as its first argument an unevaluated expression. This expression is evaluated in a new Maple process. (The new Maple process will be passed copies of all objects pointed to by the expression.) The id of the spawned process is returned. If a second argument is given, it is taken to be a host name and the new process is created on that host. Otherwise the process is created on the least loaded host listed in the file *.rhosts* in the home directory.

```
statlist := kill(processid);
```

This function terminates the process with the given process id. A number of statistics are returned, including the words used, CPU time, real time, real time blocked, real time spawning and killing, and the total number of messages sent and received.

```
statlist := stats();
```

The *stats* function returns the statistics (listed above) at the time of calling for the current process.

```
hostlist := hosts();
```

This function returns a list of host names upon which processes may be spawned.

```
response := send(processid, msg);
```

This function takes a process id as its first parameter and any Maple object as its second parameter. The Maple object is delivered as a message to the indicated process when it does a corresponding *receive* call. The call to the *send* function returns when a *reply* to the message arrives. The reply is returned as the value of the function call.

```
pair := receive(processid);
pair := receive();
```

This function receives a message from a process. If a parameter is given, then the call will wait if necessary for a message from the process with the indicated process id. If no parameter is given, or if the process id is *idnull*, then the first message to become available from any sender is accepted. This function returns a two element expression sequence, the first element of which is the process id of the sending process and the second element of which is the Maple object constituting the message.

```
reply(processid, response);
```

This function is used to reply to a received message. The first parameter is the process id to which the reply is delivered. The second parameter is the Maple object which is delivered as the reply.

```
id := idself();
id := idparent();
id := idnull();
```

These functions return the indicated process ids. If *idparent* is called from the top level process, then *idnull* is returned. The function *idnull* is provided to allow programs to be written which do not depend on the representation of process ids.

8.0 IMPLEMENTATION ASPECTS

8.1 Chapter Overview

In this chapter we describe the more interesting aspects of the implementation of the high-level parallel processing facilities and of the computer algebra programs.

8.2 The Architecture of the Multiprocessing Maple System

We begin by describing the architecture of the parallel computer algebra system. The multiprocessing facilities have been designed in a number of layers. We begin with an overview of the entire structure and then provide details about each layer.

In all, there are seven levels in the implementation of the multiprocessing Maple system (see Figure 8.1).

7. High-Level Multiprocessing Constructs
6. External Maple Support
5. Internal Maple Support
4. Process Allocation
3. Very Large Messages
2. Blocking Interprocess Communication
1. Reliably Delivered Messages
0. Operating System

Figure 8.1: Layers in the Implementation of the Multiprocessing Maple System

The operating system is viewed as the lowest level because the multiprocessing software relies critically on certain facilities it provides. The next level provides reliably delivered short messages between processes on any host. On top of this there is a set of functions which provides message buffering and implements the *send/receive/reply* blocking semantics for the short messages. The next layer provides the ability to transmit arbitrarily large messages. This is crucial for computer algebra where the messages could have sizes in the megabytes. The next level provides the functions to create and destroy processes dynamically and functions for monitoring their resource usage. The functions at this level and below are packaged as a library which may be used by any program.

The remaining layers incorporate this library in Maple and use it to provide high level multiprocessing constructs.

The Operating System

The multiprocessor upon which the multiprocessing Maple system runs is a local area network of Vaxes.[1] The system relies upon certain functions in the 4.2 BSD release of Vax Unix.[2] It makes use of the ability to have multiple processes and the Ethernet support for Internet sockets. Both of these facilities are provided by the Unix kernel. In addition it makes use of the *rsh* program which allows the execution of commands on remote processors (and is itself implemented using the Internet socket support in a privileged mode).

There are certain restrictions in this version of Unix that have an impact on the design of a system such as ours. In particular, it is necessary to work around the restriction of having a relatively small number of open files (or more accurately I/O descriptors) in any given process. There is also a relatively low limit to the number of processes that may be simultaneously running under any particular user id.

The Reliably Delivered Message Layer

The Internet socket support provided by 4.2 BSD Unix allows two different communication methods although it is planned that more shall eventually be provided. The two which are currently available are datagram sockets and stream sockets.

After a socket is created it is necessary to bind a name to it. This name is called the Internet socket address and there is a Unix system call for this purpose. The name is itself a data structure and can be transmitted as part of a message if desired. Using datagram sockets a process can send messages to any socket for which it has the Internet address. Exactly this functionality is needed at the base level of our multiprocessing software. We could use datagrams to transmit short messages, however, it is part of their definition that they are not guaranteed to be delivered. What we require is a reliably delivered message (RDM) scheme upon which we can build software for sending large expressions.

The second variety of socket is the stream socket. Stream sockets provide the ability for two processes to set up a two way communication channel. The advantage of this is that large amounts of data can be passed reliably through this channel and be delivered in order of sending. The disadvantage is that the connection takes time to establish and uses up one of the limited supply of I/O descriptors in a permanent connection.

[1] Vax is a trademark of Digital Equipment Corporation.

[2] Unix is a trademark of AT&T Bell Laboratories.

Because of this a process cannot maintain a large number of connected stream sockets. It is possible however, to use stream sockets effectively by having all processes connect with a message server. This method is used in the multiprocessing Maple System.

When the multiprocessing Maple system is started up the first thing that it does is create a message server. All processes which are to communicate using RDMs must connect to this server. Upon connection, the first message that is sent is used to identify the process to the server. Subsequent messages contain three fields of fixed length. The first field identifies the sending process by giving an Internet address, the name of the host upon which it is running and its Unix process id on that host. The second field identifies the process to which the message is being sent in the same way. The third field may contain arbitrary data. Because all of the RDMs must go through the message server, the messages are limited to a fixed size (150 bytes) to prevent potential bottlenecks.[3]

The Blocking Internet Process Communication Layer

RDMs satisfy the basic communication needs but are hard to use for synchronization. As discussed before in order to provide for process synchronization we wish to have blocking interprocess communication semantics based on a *send/receive/reply* cycle. This functionality is provided by the blocking interprocess communication (IPC) layer, which endows the fixed sized messages with these semantics.

When a process wishes to *send* a message to another it writes it to the RDM server and fills in a data structure indicating that it is send-blocked on the destination process. It then reads messages from the RDM server until a reply is obtained. Usually the process will remain blocked for some time while awaiting messages from the server. Any messages that are obtained before the reply must be *send*s from other processes. These messages are maintained in a list until such time as they are needed for a corresponding *receive*.

A process may wish to *receive* a message from a specific other process or it may wish to receive the first message from any process that is available. In either case, the list of sent messages is first checked and if a suitable message is not found the process marks itself as receive blocked, awaiting a *send* from the named process. It then awaits messages from the RDM server until such a time as it obtains a message from the desired sender. Any messages obtained before this are *send*s from other processes. When the desired message is obtained it is returned as the value of the *receive* function.

[3]The message server initially consists of one process. If very many processes wish to communicate using RDMs, the message server must be prepared to spawn subordinate processes to get around the limitation on the number of active I/O descriptors.

To *reply*, a message is passed via the RDM server back to the sender which is awaiting it. Spurious replys are ignored by the server.

In summary the only messages which a process may unexpectedly obtain from the RDM server are sends from other processes. Each process maintains its own queue of unreceived messages and its own blocking status.[4] Although this functionality could be provided by the RDM server, maintaining this distinction allows the low level messages to be passed in other ways. For example, the message passing filter used this same collection of functions with datagram sockets to pass the messages.

The Very Large Message Layer

This is the first layer in which considerations for computer algebra play a dominant role. In computer algebra it is quite common to be dealing with data structures that can be extremely large. Passing such objects as messages is handled at a level above the basic blocking IPC for two reasons. The first reason is that if a central switch is used for the basic IPC then very large messages would cause the switch to become a bottleneck with even moderate use. Secondly it would be a serious problem for a process to buffer very many unreceived messages. We have previously mentioned that the *send/receive/reply* semantics do not require message buffering on a uniprocessor. Although in our distributed implementation short messages are buffered, it is not necessary to buffer very large messages. With a two level message passing scheme it is possible to guarantee that the recipient is ready to receive a message before it is sent.

In the multiprocessing Maple system, the second level of the two level message passing scheme uses stream sockets directly for the transfer of messages.[5] The synchronization for the *send/receive/reply* of the very large message layer is achieved by passing messages in the blocking IPC layer.

In the very large message layer the initialization function creates a stream socket upon which to accept connections. The Internet address of this socket is the address which is used in the identification fields for the lower level message passing.

When a process wishes to *send* a very large message it first sends a short message (via the blocking IPC layer) which is a request to create a stream connection. When the reply is obtained the sender first initiates a stream connection and then calls a user specified function with a pointer to the message and the I/O descriptor as parameters. This user function is expected to write the message out on the unit specified by the I/O descriptor. When the user specified function returns, the sender closes the stream

[4] An additional list of messages that have been received but not replied to is maintained in order to check for and avoid simple deadlocks.

[5] In the prototype described in section 7.5 this second level was implemented using network file transfers.

connection. This may at first seem curious because a stream connection is needed for the reply. However, since the receiving end may have a large number of processes awaiting replies, it cannot maintain a stream socket open to each one of them. After closing the stream connection the first process sends a second request to connect (again via the lower level blocking IPC layer). When the receiver is ready to reply with a very large message it replies to the connect request and a second connection is established. This time a second user supplied function is called with the I/O descriptor as a parameter to read the reply. It is the responsibility of the user supplied functions to have some mutually intelligible message format which allows the matching of message lengths.

The Process Allocation Layer

This layer provides the ability to allocate processes on other processors and to deallocate them. The basic tool is the *rsh* command provided by Unix. The only subtlety in process creation is in obtaining the child's process identification in such a way that it can be passed back to the parent process. This can be achieved as follows.

The *rsh* command takes as arguments a host name, the name of an executable file and parameters to be passed. When this file is executed, *rsh* maintains a communication stream between the remote command and the caller. In its usual use the remote command terminates and *rsh* returns. When the *spawn* function is creating the argument list for the remote command, it includes as arguments its own process id, the id of the RDM server and the name of the executable file for the user process. The command that is invoked on the remote host adds the two process ids to the environment[6] and overlays itself with the user specified command.

The initialization function which must be called by the spawned program checks the process environment. If it finds these ids then it knows it is a spawned process and performs certain functions. It first creates the stream socket to be used for very large messages and using this then determines its own process id. After this it then writes this id on the standard output which it then closes. (Further initializations are performed such as connection to the RDM server.) The parent process is expecting to read a single line from the child which it interprets as the child's id. This procedure enables both the parent and child, knowing each others ids, to proceed independently.

The exact mechanism for killing remote processes is straightforward. In order to collect statistics on resource usage special care is taken to trap the "quit" interrupt. Before dying a process passes a message containing resource use to its parent. If this information is desired at intermediate intervals it is available to be incorporated in any or all messages.

[6]Each process in Unix has an "environment" which is passed on to all of its descendant processes. The environment itself is nothing more than a vector of character strings.

The functions provided at this level (see Figure 8.2) form a complete multiprocessing library. This library is used in the multiprocessing Maple system and can be used by other programs as well.

```
id    =  mpfidSelf;
id    =  mpfidParent;
id    =  mpfidNull;

         mpffinalize();
         mpfinitialize(getfn, putfn, puserstat1, pu2, pu3, failValue);

id    =  mpfspawn(idArea, hostnameOrAny, progFile, infile, outfile);
stats =  mpfkill(id);

msg   =  mpfsend(id, msg);
msg   =  mpfreceive(idorAny, msg);
         mpfreply(id, msg);

stats =  mpfstats(statsArea);
l     =  mpfhostsAndLoads();
```

Figure 8.2: Library Used by Kernel of Multiprocessing Maple System

Internal Maple Support

The functions in the multiprocessing library are, not surprisingly, exactly those primitives we wish to provide in the multiprocessing Maple system. In order to minimize the modifications to the Maple kernel, these functions are all accessed via a single, built-in function *multiprocess*. The first argument to *multiprocess* selects the desired action and the remaining arguments are passed on to the corresponding internal function.

The data structures upon which these functions operate are process ids and messages. Process ids are converted to Maple names and the messages can be arbitrary expression trees.

The functions that read and write the expression trees as very large messages are simply those that read and write internal format data and which are normally used for the creation of ".m" files.

External Maple Support

There is a library of Maple functions that provide the interface to the *multiprocess* function. In order to use the multiprocessing Maple system, the user must have in his home directory a *.mapleinit* file containing commands to load this small library and invoke the *dmapleBegin* function.

Except for *spawn* and *dmapleBegin* the functions in this library are all simple functions providing a more palatable calling sequence than *multiprocess* provides.

The process spawning function provided by the multiprocessing library can only execute a single command. It cannot pass data to the command except in the form of arguments. Somehow the Maple expression which is to be evaluated by the newly created Maple process must be passed. To do this the *dmapleBegin* function is used. When called by a top level Maple process or by a version of Maple that does not support multiprocessing, this function simply returns without doing anything. When *dmapleBegin* is called by a child process it attempts to *receive* a message from the parent. The communication functions are very careful not to cause extra evaluations. When the child receives a message from the parent it immediately replies and then *evaluates* the message it has received. It is the evaluation of this message that constitutes the execution of the spawned process. The *spawn* function must *send* the expression to be evaluated to the child. Usually the expression is an unevaluated function call and its evaluation will involve calculations and perhaps interprocess communication. After evaluating the message, *dmapleBegin* executes a *quit* statement.

High-Level Multiprocessing Constructs

The process allocation and communication primitives provided by the interface of the multiprocessing Maple system are very flexible and can be used to create arbitrary multiprocessing configurations. Quite often though, higher level constructs are more suitable.

In Chapter 2 we described two high level constructs for dynamic scheduling, namely the Administrator and the Don. These constructs are implemented as two functions in the library for use with the multiprocessing Maple system. These functions, called *AllOf* and *OneOf*, are described in the next sections.

8.3 AND Parallelism

In Chapter 7 we described the parallel processing primitives provided in the testbed software. While they provide a great deal of flexibility, the use of these primitives requires an attention to detail which need not be of direct concern in the writing of

```
AllOf(f, args, indexEquation) ==
    breakupEquation(indexEquation, 'index', 'lo', 'hi')
    InitializeIfNecessary()
    outstandingWorkerIDs := {}
    for i in lo... while i < hi or outstandingWorkerIDs <>{} repeat
        pair := receiveAny()
        id := pair[1];  result := pair[2]
        outstandingWorkerIDs := outstandingWorkerIDs - {id}
        if result <> Not_A_Result_Just_A_Request then
             # result is [i, f(i)]
             resultTable[result[1]] := result[2]
        if i < hi then
             reply(id, makeTask(f, args, index, i))
             outstandingWorkerIDs :=  outstandingWorkerIDs + {id}
        else
             reply(id, Refresh_Worker)
    [seq(resultTable['i'], 'i' = lo .. hi)]
```

Figure 8.3: Implementation of AllOf using the Administrator Construct

computer algebra code. In most cases, the high level view of the problem appeals to AND or OR parallelism. For these cases it is appropriate to have procedures providing functionality at this high level of abstraction.

In order to make use of AND-parallelism, the multiprocessing Maple system provides the *AllOf* function.

```
l := AllOf (f, args, i = lo .. hi);
```

Here f is a function to be applied to the argument list *args*. For each value of i, f is applied to the argument list and with i replaced by its value. The results obtained are returned together as a list.

The *AllOf* function uses the Administrator paradigm to allocate the computations to the available processors.

The first step is the initialization of one worker process on each processor. Each worker process sends a message to the Administrator (its parent) requesting work. Because of the semantics of the *send* primitive, the workers remain blocked until they are presented with tasks to perform. This initialization step is not always necessary,

```
OneOf(f, args, indexEquation) ==
    breakupEquation(indexEquation, 'index', 'lo', 'hi')
    InitializeIfNecessary()
    outstandingWorkerIDs := {}
    result := FAIL
    for i in lo... while i < hi or outstandingWorkerIDs <>{} do
          pair := receiveAny()
          id := pair[1];  result := pair[2]
          outstandingWorkerIDs := outstandingWorkerIDs - {id}
          if result <> Not_A_Result_Just_A_Request and
                 result <> FAIL then exitloop
          if i < hi then
                 reply(id, makeTask(f, args, index, i))
                 outstandingWorkerIDs :=  outstandingWorkerIDs + {id}
          else
                 reply(id, Refresh_Worker)
    for i to nops outstandingWorkerIDs do
          kill(outstandingWorkerIDs[i])
    InitializeIfNecessary()
    result
```

Figure 8.4: Implementation of OneOf using the Don Construct

because a worker process may already exist from a previous use. The details of this are discussed later.

After initialization, the Administrator process receives work requests from the worker process using *receive any* and replies with descriptions of the tasks. Since *receive any* returns immediately if there is any worker available, this strategy keeps all processors busy. The use of *reply* to present a task to a worker ensures that the Administrator does not block.

If there are fewer tasks than workers, then some of the processors are not utilized. If there are more tasks than workers, then as workers return they are presented with new tasks until there are no tasks left. The first message received from each worker is a simple work request. Subsequent messages, as well as being work requests, contain the results of completed tasks. This is illustrated in Figure 8.3.

```
Broadcast(f, args, workerIDs) ==
    outstandingWorkers  := convert(workerIDs, set)
    waitingBroadcastees := {}
    innocentBystanders  := {}

    while outstandingWorkers ≠ {} do
        pair := receiveAny()
        id := pair[1]
        if id ∈ outstandingWorkers then
             outstandingWorkers := outstandingWorkers - {id}
             waitingBroadcastees := waitingBroadcastees ∪ {id}
        else
             innocentBystanders := innocentBystanders ∪ {id}
    for id in waitingBroadcastees do
        reply(id, args)
    for id in innocentBystanders do
        reply(id, Refresh_Worker)
```

Figure 8.5: Implementation of Broadcast

Although the *AllOf* construct treats the tasks to be performed homogeneously, it is quite possible for the tasks themselves to be quite different. This can be done by having the function f dispatch to other functions based on the value of i.

As a final note, we point out that the arguments to which f is applied are often selector forms, for example

```
AllOf(f, ['op(i, giantPolynomial)'],  i = 1 .. nops(giantPolynomial))
```

:pc. This is the reason that the function and the argument list are passed separately to *AllOf*. Doing this allows the arguments to be evaluated *before* passing the task to the worker process, cutting down considerably on the communication costs.

We mentioned earlier that there may be workers around from previous multiprocessing activity, and that it may not be necessary to initialize the workers for each *AllOf* operation. Whether it is desirable to keep workers around between calls to *AllOf* depends on the relative costs of process spawning, message passing, and the size of the tasks given to the workers. In the environment in which the multiprocessing Maple system was developed, process creation was much more expensive than message pass-

ing (the dominating cost was initiating the remote shell across the network). In this situation it was desirable for an administrating *AllOf* to leave its workers alive for the next *AllOf* to utilize. Before exiting, an *AllOf* sends a *Refresh_Worker* message to its blocked workers and leaves the set of worker process ids in a global variable. The workers recognize this message as a null task and simply turn around and immediately send another work request. They remain blocked awaiting a *receive*. In particular, the next *AllOf*, *OneOf* or *Broadcast* will be able to use them.

8.4 OR Parallelism

The multiprocessing Maple system provides the *OneOf* function to allow the use of OR parallelism:

```
result := OneOf(f, args, i = lo .. hi)
```

The calling sequence is the same as for *AllOf* but only a single value is returned. Each call to f can produce either a result or the value *FAIL*. If any of the calls to f produces a non-*FAIL* result, then *Oneof* returns one of these values, otherwise *OneOf* returns *FAIL*. This is illustrated in Figure 8.4.

The *OneOf* function uses the Don construct to distribute the tasks over the available processors. The value returned is the *first* non-*FAIL* result.

The two functions *AllOf* and *OneOf* implement pure AND parallelism and OR parallelism respectively. Sometimes it is the case that a certain number of results from a set of computations are desired but it does not matter which ones. For such cases the multiprocessing Maple system provides the *SomeOf* function.

```
results := SomeOf(f, args, i = lo .. hi, howmany)
```

Here f is evaluated with i taking values from lo to hi until *howmany* non-*FAIL* results are obtained. This function generalizes both *OneOf* and *AllOf* in the case where *howmany* is 1 and $hi - lo + 1$ respectively.

8.5 Broadcasting

It is often necessary to broadcast a message to all workers. An example would be in letting all workers know the new value of some variable. To do this the multiprocessing Maple system provides the *Broadcast* function. Its calling sequence is simillar to that of the functions disussed so far, except a list of worker process ids is used in place of an index equation.

```
results := Broadcast(f, args, workerids)
```

The *Broadcast* function collects work requests until a request has been received from each worker on the broadcast list. These workers are sent the specified message. The requests from workers not on the list are given Refresh_Worker response, which simply tells them to ask for work again. This is illustrated in Figure 8.5.

9.0 CONCLUDING REMARKS

9.1 Summary

Foremost,we have seen that parallel computing methods form a useful set of tools for applications in computer algebra. We have seen that some forms of inherent parallelism in computer algorithms are more useful than other forms. The ideal situation from the point of view of an algorithm with bounded parallelism is when there is a large number of high-level tasks that can be performed independently. It is usual in computer algebra that these high-level tasks require differing execution times. We have seen that the Administrator concept is very useful in this situation. This was the situation with the parallel GCD algorithm that we developed.

Sometimes there is a large number of tasks that could be performed in parallel but which are not independent. For example, in the calculation of Gröbner bases using Buchberger's algorithm, the results at one stage affect the results at later stages by modifying a global variable. In this case we were able to convert the algorithm to one which does not use a global variable but instead performs some small number of redundant calculations.

Sometimes the level of parallelism available is on a low enough level that directly exploiting it on a general purpose multiprocessor is impractical because of the ratio of the overhead to the useful work. In this situation, depending on distribution of execution times, a collusive algorithm can yield speedups. Furthermore, we have seen that the implementation of these algorithms can be quite simple in the parallel setting. This situation is exemplified by the parallel integer factorization algorithm presented in Chapter 4.

It seems that each problem in computer algebra has some inherent parallelism. In each of the problems we examined, we found clear sources of parallelism. Although in each problem it took a slightly different form, we saw that it was possible to code the algorithms using two primitives for high-level parallelism: *AllOf* and *OneOf*. These express AND-parallelism and OR-parallelism, respectively.

Our implementation of *AllOf* and *OneOf* uses a fixed set of processes working together on a *problem heap* or in *collusion*. Internally, blocking message passing is used and synchronization and load sharing are based on the blocking semantics. The communication scheme used for the problem heap is the *Administrator* concept of Gentleman. We call the corresponding communication scheme for collusion the *Don* concept. We found that it was not necessary for the algebraic algorithms to use general facilities for interprocess communication. The Administrator and Don communication paradigms embodied in the parallel processing primitives were capable of handling the desired interactions.

A general purpose computer algebra system requires many man years to construct. The time to develop the programming language and to implement an interpreter or compiler, although substantial, is dwarfed by the amount of effort that is expended in developing the library of mathematical software. With such a body of code, it is a very real concern not to become locked in to a particular processor or architecture. If the language provides operators to express parallelism at a high level, then the dependencies are minimized.

If a system is to be able to take advantage of parallelism, it is better able to do so when explicitly shown where parallelism exists. Even if the computing system upon which the computer algebra system rests does not support multiprocessing, there is no harm in explicitly expressing the high-level parallelism inherent in the algebraic algorithms.

We argue that it is desirable to explicitly indicate high-level parallelism in the source both for algorithmic clarity and for portability of any newly developed computer algebra systems.

9.2 Contributions

The original material presented in this thesis includes:

1. A mathematical formulation of collusion and a demonstration that identical colluding processes can yield a speedup even on a single processor.

2. A high-level construct for OR-parallelism which is analogous to the use of the Administrator construct used with AND-parallelism.

3. Parallel algorithms for specific computer algebra problems. Specifically a collusive algorithm for integer factorization and problem heap algorithms for the calculation of multivariate polynomial GCDs and the computation of Gröbner bases.

4. A distributed programming environment for the implementation of parallel computer algebra programs.

9.3 Directions for Further Work

A very exciting direction for work in integer factorization would be to invesitgate various methods using collusion on a SIMD processor. It should be possible to get good results exploiting a vector architecture with Lenstra's method.

For some time much research activity has been devoted to the complexity of Gröbner basis calculation [**Mayr82**]. It would be useful to determine how many reductions are required by the parallel algorithm that are not required by the serial algorithm.

Our investigation of Gröbner bases calculation was in a message passing context. It would also be interesting to see what would be possible given a shared memory multiprocessor.

Our multiprocessing system was obtained by modifying an existing serial system. It would be useful to build a parallel computer system from the ground up.

There are a host of problems in computer algebra of which we have examined only a small number. In each we saw opportunities to take advantage of parallelism. Given this, the whole body of computer algebra literature is open for review.

BIBLIOGRAPHY

General References

[Aczel66] J. Aczel, *Lectures on Functional Equations and their Applications*, Academic Press, New York (1966).

[Ben-Or83] M. Ben-Or, “Lower Bounds for Algebraic Computation Trees”, pp. 80-86 in *Proc. 15th Annual ACM Symposium on the Theory of Computing*, (April 1983).

[Bentley80] J.L. Bentley, “Multidimensional Divide and Conquer”, *Comm. ACM* **23** (4) pp. 214-229 (1980).

[Caferra85] R. Caferra and P. Jorrand, “Unification in Parallel with Refined Linearity Test”, pp. 539-540 in *Proc. Eurocal’85, Vol. 2* ed. B.F. Caviness, European Computer Algebra Conference, Linz, Austria (April 1985), Springer-Verlag Lecture Notes in Computer Science No. 204.

[Char85] B.W. Char, K.O. Geddes, G.H. Gonnet and S.M. Watt, *Maple User’s Guide*, Watcom Publications, Waterloo (1985).

[Cheriton79a] D.R. Cheriton, M.A. Malcolm, L.S. Melen, and G.R. Sager, “Thoth, a Portable Real-Time Operating System”, *Comm. ACM* **22** (2) pp. 105-115 (1979).

[Cheriton79b] D.R. Cheriton, “Multi-process Structuring and the Thoth Operating System”, University of Waterloo, Ph.D. Thesis (1979).

[Coffman80] E.G. Coffman and K. So, “On the Comparison Between Single and Multiple Processor Systems”, *Proc. 7th Annual Symposium on Computer Architecture* pp. 72-79, IEEE (1980).

[David62] F.N. David and D.E. Barton, *Combinatorial Chance*, Hafner Publishing Company, New York (1962).

[Fich85a] F.E. Fich, F.M. auf der Heide, P. Ragde and A. Wigderson, “One, Two, Three ... Infinity: Lower Bounds for Parallel Computation”, pp. 48-58 in *Proc. 17th Annual ACM Symposium on the Theory of Computing*, Providence, R.I. (May 1985).

[Flynn66] M.J. Flynn, "Very High-Speed Computing Systems", *Proc. IEEE* **54** pp. 1901-1909 (1966).

[Friebiblist85] K. Friebiblist and L. Ronyai, "Polynomial Time Solutions of Some Problems in Computational Algebra", pp. 153-162 in *Proc. 17th Annual ACM Symposium on the Theory of Computing*, Providence, R.I. (May 1985).

[Gentleman81] W.M. Gentleman, "Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept", *Software—Practice and Experience*, **11** pp. 435-466 (1981).

[Gentleman83] W.M. Gentleman, "Using the Harmony Operating System", Report NRCC No. 23030, National Research Council, Canada (December 1983).

[Gottlieb82] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph and M. Shir, "The NYU Ultracomputer—Designing a MIMD, Shared-Memory Parallel Machine", *Proc. 9th Annual Symposium on Computer Architecture*, *ACM SIGARCH Newsletter* **10** (3) pp. 27-42 (April 1982).

[Hibino80] Y. Hibino, "A Practical Parallel Garbage Collection Algorithm and It's Implementation", *Proc. 7th Annual Symposium on Computer Architecture*, IEEE (1980).

[Hoare76] C.A.R. Hoare, "Parallel Programming: An Axiomatic Approach", pp.11-42 in *Language Hierarchies and Interfaces*, ed. F. L. Bauer and K. Samelson, Springer-Verlag Lecture Notes in Computer Science No. 46, Berlin (1976).

[Hoare78] C.A.R. Hoare, "Communicating Sequential Processes", *Comm. ACM* **21** (8) pp. 666-677 (1978).

[Knuth73] Donald E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms, 2nd Edition*, Addison-Wesley, Reading, Massachusetts (1973).

[Knuth81] Donald E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms, 2nd Edition*, Addison-Wesley, Reading, Massachusetts (1981).

[Leffler83] S.J. Leffler, W.N. Joy and M.K. McKusick, *Unix Programmer's Manual, 4.2 Berkeley Software Distribution*, Virtual VAX-11 Version, University of California, Berkeley (August 1983).

[Legendi85] T. Legendi, “Cellular Algorithms and Machines: An Overview”, no manuscript, *Eurocal'85*, European Computer Algebra Conference, Linz, Austria (April 1985).

[Moller-Nielsen83] P. Moller-Nielsen and J. Staunstrup, “Saturation in a Multiprocessor”, pp. 383-388 in *Information Processing 83*, ed. R.E.A. Mason, Elsevier Science Publishers, North-Holland (1983).

[Moller-Nielsen84] P. Moller-Nielsen and J. Staunstrup, “Experiments with a Multiprocessor”, Report DAIMI PB-185, Computer Science Department, Aarhus University, Aarhus, Denmark, Nov. 1984.

[Rivest78] R. Rivest, A. Shamir and L. Adleman, “A Method for Obtaining Digital Signatures and Public Key Cryptosystems”, *Comm. ACM*, **21** (2), (1978).

[Salvador78] M.S. Salvador, “Scheduling and Sequencing”, in *Handbook of Operations Research: Models and Applications*, ed. J. J. Moder and S. E. Elmaghraby, Van Nostrand Reinhold, New York (1978).

[Smith78] R. G. Smith, *A Framework for Problem Solving in a Distributed Processing Environment*, Report STAN-CS-78-700, Stanford University (1978).

[Wallach82] Y. Wallach, “Alternating Sequential/Parallel Processing”, Springer-Verlag Lecture Notes in Computer Science No. 127, Berlin, (1982).

[Wang78] P. S. Wang, "An Improved Multivariate Polynomial Factoring Algorithm", *Math. Comp.* **32** (144) pp. 1215-1231 (1978).

[Wise84] D.S. Wise, "Representing Matrices as Quad Trees for Parallel Processes", *ACM SIGSAM Bulletin* **18** (3) pp. 24-25 (1984).

[Yao83] A.C. Yao, "Lower Bounds by Probabilistic Arguments", pp. 420-428 in *Proc. 24th Annual IEEE Symposium on Foundations of Computer Science* (1983).

Parallelism in Computer Algebra

[Arnon84] Dennis Arnon, Editor, "Report Of Workshop on Supercomputers and Symbolic Computation", Purdue University, West Lafayette, Indiana, May 1984.

[Beardsworth81] R. Beardsworth, "On the Application of Array Processes to Symbol Manipulation", pp. 126-130 in *Proc. SYMSAC '81*, ed. Paul S. Wang, 1981 ACM Symposium on Symbolic and Algebraic Computation, Snowbird, Utah (August 1981).

[Bibel84] W. Bibel and B. Buchberger, "Towards a Connection Machine for Logical Inference", Technical Report of Computer Aided Mathematical Problem Solving Group, *CAMP-Publ.-Nr. 84-19.0*, Johannes Kepler University, Linz, (October 1984).

[Berman84] Robert Berman, "Measuring the Performance of a Computational Physics Environment", pp. 244-290 in *Proc. 1984 Macsyma User's Conference*, Schenectedy, New York (July 1984).

[Borodin82] A. Borodin, J. von zur Gathen and J. Hopcroft , "Fast Parallel Matrix and GCD Computations", *Information and Control* **52** pp. 241-256 (1982) also available as pp. 65-71 in *Proc. 23th Annual IEEE Symposium on Foundations of Computer Science*, (November 1982).

[Buchberger85a] B. Buchberger, "The Parallel L-Machine for Symbolic Computation", pp. 541-542 in *Proc. Eurocal'85, Vol. 2* ed. B.F. Caviness, European Computer Algebra Conference, Linz, Austria

(April 1985), Springer-Verlag Lecture Notes in Computer Science No. 204.

[Davenport81] James H. Davenport, *On the Integration of Algebraic Functions*, Springer-Verlag Lecture Notes in Computer Science No. 102, Springer-Verlag, Berlin (1981).

[Davenport84] J.H. Davenport and Yves Robert, "VLSI et Calcul Formel: l'example du PGCD" in Comportement des automates et applications (Proceedings), Springer-Verlag (to appear).

[Eberly84] W. Eberly, "Very Fast Parallel Matrix and Polynomial Arithmetic", pp. 21-30 in *Proc. 25th Annual IEEE Symposium on Foundations of Computer Science*, Sugar Island, Florida, (1984).

[Fateman84] R.J. Fateman, "My View of the Future of Symbolic and Algebraic Computation", *ACM SIGSAM Bulletin* **70** pp.10-11 (1984).

[Fich85b] F.E. Fich and M. Tompa, "The Parallel Complexity of Exponentiating Polynomials over Finite Fields", pp. 38-47 in *Proc. 17th Annual ACM Symposium on the Theory of Computing*, Providence, R.I. (May 1985).

[Gathen83a] J. von zur Gathen, "Representations of Rational Functions", pp. 133-137 in *Proc. 24th Annual IEEE Symposium on Foundations of Computer Science* (November 1983).

[Gathen83b] J. von zur Gathen, "Parallel Algorithms for Algebraic Problems", pp. 17-23 in *Proc. 15th Annual ACM Symposium on the Theory of Computing*, (April 1983) also available as *SIAM J. Comp.* **13** (4) pp. 802-824 (1984).

[Gathen84] J. von zur Gathen, "Parallel Powering", pp. 31-36 in *Proc. 25th Annual IEEE Symposium on Foundations of Computer Science*, Singer Island, FA. (1984).

[Kaltofen85a] Erich Kaltofen, "Computing with Polynomials Given by Straight-Line Programs I: Greatest Common Divisors", pp.

131-142 in *Proc. 17th Annual ACM Symposium on the Theory of Computing*, Providence, R.I. (May 1985).

[Kaltofen85b] Erich Kaltofen, "Computing with Polynomials Given by Straight-Line Programs II: Sparse Factorization", Rensselaer Polytechnic Institute, Department of Computer Science, Troy, NY, Preprint (1985).

[Kaltofen85c] Erich Kaltofen, "Parallelizing Computations of Rational Functions", Rensselaer Polytechnic Institute, Department of Computer Science, Troy, NY, Preprint (1985).

[Kaltofen85d] Erich Kaltofen, M. Krishnamoorthy and B. David Saunders, "Fast Parallel Computation of Hermite and Smith Forms of Polynomial Matrices", Rensselaer Polytechnic Institute, Department of Computer Science, Troy, NY, Preprint (1985).

[Kung81] H.T. Kung "Use of VLSI in Algebraic Computation: Some Suggestions", pp. 218-222 in *Proc. SYMSAC '81*, ed. Paul S. Wang, 1981 ACM Symposium on Symbolic and Algebraic Computation, Snowbird, Utah (August 1981).

[Marti83] J. Marti and J. Fitch, "The Bath Concurrent LISP Machine", pp. 78-90 in *Proc. Eurocal'83*, ed. J.A. van Hulzen, European Computer Algebra Conference, London, England (March 1983).

[Pan85] V. Pan and J. Reif, "Efficient Parallel Solution of Linear Systems", pp. 153-162 in *Proc. 17th Annual ACM Symposium on the Theory of Computing*, Providence, R.I. (May 1985).

[Reif83] John Reif, "Logarithmic Depth Circuits for Algebraic Functions", pp. 138-145 in *Proc. 24th Annual IEEE Symposium on Foundations of Computer Science* (1983).

[Sasaki81] T. Sasaki and Y. Kanada "Parallelism in Algebraic Computation and Parallel Algorithms for Symbolic Linear Systems", pp. 155-159 in *Proc. SYMSAC '81*, ed. Paul S. Wang, 1981 ACM Symposium on Symbolic and Algebraic Computation, Snowbird, Utah (August 1981).

[Smit83] J. Smit "Computer Algebra and VLSI, Prospects for Cross Fertilization", pp. 275-285 in *Proc. Eurocal'83*, ed. J.A. van Hulzen, European Computer Algebra Conference, London, England (March 1983).

[Valiant81] L.G. Valiant and S. Skyum, "Fast Parallel Computation of Polynomials Using Few Processors", pp. 132-139 in Springer Verlag Lecture Notes in Computer Science No. 118, New York (1981).

[Valiant83] L.G. Valiant, S. Skyum, S. Berkowitz and C. Rackoff, "Fast Parallel Computation of Polynomials Using Few Processors", *SIAM J. Comp.* **12** (4) pp. 641-644 (1983).

[Watt85] S.M. Watt, "A System for Parallel Computer Algebra Programs", pp. 537-538 in *Proc. Eurocal'85, Vol. 2* ed. B.F. Caviness, European Computer Algebra Conference, Linz, Austria (April 1985), Springer-Verlag Lecture Notes in Computer Science No. 204.

[Yun85] Y.Y. Yun and C.N. Zhang "Binary Paradigm and Systolic Array Implementation for Residue Arithmetic", no manuscript, *Eurocal'85* Euoropean Computer Algebra Conference, Linz, Austria (April 1985).

Scheduling

[Coffman78] E.G. Coffman, Jr., M.R. Garey and D.S. Johnson, "An Application of Bin-Packing to Multiprocessor Scheduling", *SIAM J. Comp.* **7** (1) pp. 1-17 (1978).

[Ecker78] Klaus Ecker, "Analysis of a Simple Strategy for Resource Constrained Task Scheduling", pp.181-183 in *Proc. 1978 International Conference on Parallel Processing* , (August 1978).

[Garey73] M.R. Garey, "Optimal Task Sequencing with Precedence Constraints", *Discrete Math.* **4** pp. 37-56 (1973).

[Garey75a] M.R. Garey and R.L. Graham, "Bounds for Multiprocessing Scheduling with Resource Constraints", *SIAM J. Comp.* **4** (2) pp. 187-190 (1975).

[Garey75b] M.R. Garey and D.S. Johnson, "Complexity Results for Multiprocessor Scheduling Under Resource Constraints", *SIAM J. Comp.* **4** (4) pp.397-400 (1975).

[Graham66] R.L. Graham, "Bounds for Certain Multiprocessing Anomalies", *Bell System Technical Journal* pp.1563-1581 (November 1966).

[Graham69] R.L. Graham, "Bounds on Multiprocessing Timing Anomalies", *SIAM J. Appl. Math.* **17** (2) pp.416-429 (1969).

[Gottlieb83] A. Gottlieb, B.D. Lubachevsky and L. Rudolph, "Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors", *ACM Transactions on Programming Languages and Systems* **6** (2) pp. 165-189 (1983).

[Lai84] Ten-Hwang and Sartaj Sahni, "Anomalies in Parallel Branch-and-Bound Algorithms", *Comm. ACM* **27** (6) pp.594-602 (1984).

[Lawler66] E.L. Lawler and D.E. Wood, "Branch-and-Bound Methods: A Survey", *Operations Research* **14** pp.699-719 (1966).

[Mehrotra84] R. Mehrotra and S.N. Talukdar, "Scheduling of Tasks for Distributed Processors", *ACM SIGARCH Newsletter* **12** (3) pp. 263-270 (1984).

[Simons80] B. Simons, "A Fast Algorithm for Multiprocessor Scheduling", pp. 50-53 in *Proc. of the 21st Annual Symposium on Foundations of Computer Science*, Syracuse, N.Y. (October 1980).

Collusion

[Baudet78] G.M. Baudet, "The Design and Analysis of Algorithms for Asynchronous Multiprocessors", Carnegie-Mellon University,

Ph.D. Thesis (1978).

[Floyd67] R.W. Floyd, "Nondeterministic Algorithms", *JACM* **14** (4) pp. 636-644 (1967).

[Kadane69] J.B. Kadane, "Quiz Show Problems", *J. Math. Anal. Appl.* **26** pp. 609-623 (1969).

[Kornfeld81] W.A. Kornfeld, "The Use of Parallelism to Implement a Heuristic Search", pp. 575-580 in *The Proceedings of the Seventh International Joint Conference on Artificial Intelligence (IJCAI-81)*, (August 1981).

[Kornfeld82] W.A. Kornfeld, "Combinatorially Implosive Algorithms", *Comm. ACM* **25** (10) pp. 734-738 (1982).

[Mitten60] L.G. Mitten, "An Analytic Solution to the Least Cost Testing Sequence Problem", *J. Indust. Engineering* p. 17 (January-February 1960).

[Nau82] D.S. Nau, V. Kumar and L. Kanal, "A General Paradigm for A.I. Search Procedures", pp. 120-123 in *Proc. Nat'l Conf. on AI, (AAAI-82)* CMU (1982).

[Price59] H.W. Price, "Least-Cost Testing Sequence", *J. Indust. Engineering* (July-August 1959).

[Reif84] J.H. Reif, "On Synchronous Parallel Computations with Independent Probabilistic Choice", *SIAM J. Comp.* **11** (1) pp. 46-56 (1984).

[Simon75] H.A. Simon and J.B. Kadane, "Optimal Problem-Solving Search: All-or-None Solutions", *AI* **6** pp. 235-247 (1975).

[Vishkin84] U. Vishkin, "Randomized Speed-ups in Parallel Computation", pp. 230-239 in *Proc. 16th Annual ACM Symposium on the Theory of Computing*, Washington, D.C. (May 1984).

Integer Factorization

[Brent80] R.P. Brent , "An Improved Monte Carlo Factorization Algorithm", *BIT* **20**, pp. 176-184 (1980).

[Brent81] R.P. Brent and J.M. Pollard, "Factorization of the Eighth Fermat Number", *Math. Comp.* **36** (4) pp. 627-630 (1981).

[Buell84] D.A. Buell, "The Expectation of Success Using a Monte Carlo Factoring Method - Some Statistics on Quadratic Class Numbers", *Math. Comp.* **43** (167) pp. 313-327 (1984).

[Chudnovsky85a] D.V. Chudnovsky and G.V. Chudnovsky, "Sequences of Numbers Generated by Addition in Formal Groups and New Primality and Factorization Tests", Report RC 11262, IBM Research Division, San Jose/Yorktown/Zurich, (1985).

[Chudnovsky85b] D.V. Chudnovsky and G.V. Chudnovsky, "Elliptic Curve Calculations in Scratchpad II", in *Scratchpad II Newsletter* **1** (1) (1985).

[Davis84] J.A. Davis and D.B. Holdridge, "Most Wanted Factorizations Using the Quadratic Sieve", Report SAND84-1658, SANDIA National Laboratories, Livermore (1984).

[Dixon81] John D. Dixon, "Asymptotically Fast Factorization of Integers", *Math. Comp.* **36** (153) pp.255-260 (1981).

[Galambos76] J. Galambos "The Sequence of Prime Divisors of Integers", *Acta Arithmetica* **31** pp. 213-218 (1976).

[Knuth76] Donald E. Knuth and Luis Trabb Pardo, "Analysis of a Simple Factorization Algorithm", *Theor. Comp. Sci.* **3** pp. 321-348 (1976).

[Lenstra85] H.W. Lenstra, *Letter to A.M. Odlyzko*, (February 14, 1985).

[Morrison75] M.A. Morrison and J. Brillhart, "A Method of Factoring and the Factorization of F_7", *Math. Comp.* **29** (129) pp.183-205 (1975).

[Pollard75] J.M. Pollard, "A Monte Carlo Method for Factorization", *BIT* **15** pp. 331-334 (1975).

[Pollard78] J.M. Pollard, "Monte Carlo Methods for Index Computation (mod p)" *Math. Comp.* **32**, (143) pp. 918-924 (1978).

[Pomerance84] Carl Pomerance, *Lecture Notes on Primality Testing and Factoring: A Short Course at Kent State University*, The Mathematical Association of America Notes No. 4 (1984).

[Riesel85a] Hans Riesel, *Prime Numbers and Computer Methods for Factorization*, Birkhauser, Boston (1985).

[Riesel85b] Hans Riesel, "Modern Factorization Methods", *Bit* **25** pp. 205-222 (1985).

[Schnorr84] C.P. Schnorr and H.W. Lenstra, Jr., "A Monte Carlo Factoring Algorithm with Linear Storage", *Math. Comp.* **43** (167) pp. 289-311 (1984).

[Williams81] H.C. Williams, "A Numerical Investigation into the Length of the Period of the Continued Fraction Expansion of $\sqrt{D}$", *Math. Comp.* **36** (154) pp.593-601 (1981).

Greatest Common Divisors

[Brown71a] W.S. Brown, "On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors", *J. ACM* **18** (4) pp.478-504 (1971).

[Brown71b] W.S. Brown and J.F. Traub, "On Euclid's Algorithm and the Theory of Subresultants", *J. ACM* **18** (4) pp.505-514 (1971).

[Bryant83] M. Bryant, "The Sparse Modular GCD in Maple", University of Waterloo, December 1983, M.Math. Thesis.

[Char84] B.W. Char, K.O. Geddes and G.H. Gonnet, "GCDHEU: Heuristic Polynomial GCD Algorithm Based on Integer GCD Computation", pp. 285-296 in *Eurosam '84*, ed. John Fitch, Springer Verlag Lecture Notes in Computer Science No. 174 (1984).

[Collins67] G.E. Collins, "Subresultants and Reduced Polynomial Remainder Sequences", *J. ACM*, **14** (1), pp. 128-142, (1967).

[Collins71] G.E. Collins, "The Calculation of Multivariate Polynomial Resultants", *J. ACM* **18** (4) pp.515-532 (1971).

[Davenport84] J.H. Davenport and Yves Robert, "VLSI et Calcul Formel: l'example du PGCD" in Comportement des automates et applications (Proceedings), Springer-Verlag (to appear).

[Davenport85] J.H. Davenport and J.A. Padget, "HEUGCD: How Elementary Upperbounds Generate Cheaper Data", in pp. 18-28 in *Proc. Eurocal'85, Vol. 2* ed. B.F. Caviness, European Computer Algebra Conference, Linz, Austria (April 1985), Springer-Verlag Lecture Notes in Computer Science No. 204.

[Hearn79] A.C. Hearn, "Non-Modular Computation of Polynomial GCDs Using Trial Division", pp. 227-239 in *Symbolic and Algebraic Computation*, ed. E.W. Ng, Proc. EUROSAM '79, An International Symposium on Symbolic and Algebraic Manipulation, Marseille, France (June 1979), Springer Verlag Lecture Notes in Computer Science No. 72.

[Kaltofen85a] Erich Kaltofen, "Computing with Polynomials Given by Straight-Line Programs I: Greatest Common Divisors", pp. 131-142 in *Proc. 17th Annual ACM Symposium on the Theory of Computing*, Providence, R.I. (May 1985).

[Knuth81] Donald E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms, 2nd Edition*, Addison-Wesley, Reading, Massachusetts (1981).

[Moses73] J. Moses and D.Y.Y. Yun, "The EZ-GCD Algorithm", pp. 159-166 in *Proceedings of the A.C.M. Annual Conference*, Atlanta (1973).

[Zippel79] Richard Zippel, *Probabilistic Algorithms for Sparse Polynomials*, Massachusetts Institute of Technology, PhD. Thesis, (1979).

[Zippel81] Richard Zippel, "Newton's Iteration and the Sparse Hensel Algorithm", pp. 68-72 in *Proc. SYMSAC '81*, ed. Paul S. Wang, 1981 ACM Symposium on Symbolic and Algebraic Computation, Snowbird, Utah (August 1981).

Gröbner Bases

[Buchberger76] B. Buchberger, "Some Properties of Gröbner Bases for Polynomial Ideals", *ACM SIGSAM Bulletin* **10** (4) pp. 19-36 (1976).

[Buchberger79] B. Buchberger, "A Criterion for Detecting Unnecessary Reductions in the Construction of Gröbner Bases", pp. 3-21 in *Symbolic and Algebraic Computation*, ed. E.W. Ng, Proc. EUROSAM '79, An International Symposium on Symbolic and Algebraic Manipulation, Marseille, France (June 1979), Springer Verlag Lecture Notes in Computer Science No. 72.

[Buchberger82] B. Buchberger and R. Loos, "Algebraic Simplification", in *Computer Algebra: Symbolic and Algebraic Computation* ed. B. Buchberger, G.E. Collins and R. Loos, Springer-Verlag, New York (1982).

[Buchberger83] B. Buchberger, "A Critical-Pair/Completion Algorithm for Finitely Generated Ideals in Rings", in *Logic and Matrices: Decision Problems and Complexity, Proc. Symposium Kamsive Kombinatork*, (May 1983).

[Buchberger85b] B. Buchberger, "Gröbner Basis: An Algorithmic Method in Polynomial Ideal Theory", in *Recent Trends in Multidimensional Systems Theory*, ed. N.K. Bose, D. Reidel Publishing Co. (1985).

[Gianni85] P. Gianni and B. Trager, "GCD's and Factoring Multivariate Polynomials using Gröbner Bases", pp. 409-410 in *Proc. Eurocal'85*, ed. B.F. Caviness, European Computer Algebra Conference, Linz, Austria (April 1985), Springer-Verlag Lecture Notes in Computer Science No 204.

[Kandri-Rody84] A. Kandri-Rody and D. Kapur, "Algorithms for Computing Gröbner Bases of Polynomial Ideals Over various Euclidean Rings", pp.195-206 in *Proc. Eurosam'84*, ed. John Fitch, International Symposium on Symbolic and Algebraic Computation, Cambridge, England (July 1984), Springer Verlag Lecture Notes in Computer Science No. 174.

[Kandri-Rody85] A. Kandri-Rody and D. Kapur and P. Narendran, "An Ideal Theoretic Approach to Word Problems and Unification Problems over Finitely Presented Commutative Algebras", pp. 243-262 in *Rewriting Techniques and Applications*, ed. J.-P. Jouannaud, Proc. 1st International Conference, Universite de Dijon, France (May 1985), Springer Verlag Lecture Notes in Computer Science No. 202.

[Mayr82] E.W. Mayr and A.R. Meyer, "The Complexity of the Word Problems for Commutative Semigoups and Polynomial Ideals", *Advances in Mathematics,* **46** pp. 305-329 (1982).

[Winkler83] Franz Winkler, "On the Complexity of the Gröbner-Basis Algorithm over K[x,y,z]", pp. 184-194 in *Proc. Eurosam'84*, ed. John Fitch, International Symposium on Symbolic and Algebraic Computation, Cambridge, England (July 1984), Springer Verlag Lecture Notes in Computer Science No. 174.