A PARALLEL LISP
AND
MULTIVARIATE POLYNOMIAL GCD
IMPLEMENTATION

A thesis presented to the
University of Waikato
in fulfillment of the thesis requirements
for the degree of

Doctor of Philosophy

By

Diane M. Koorey Willcock

University of Waikato
1991

Abstract

This thesis is concerned with the intersection of two fields of study - computer algebra and parallel processing. The operation of multivariate polynomial greatest common divisor (gcd), an important and fundamental computer algebra operation, is investigated for amenability to parallelisation following the investigation of a suitable environment for implementation of a parallel computer algebra algorithm.

After researching the requirements of a parallel computer algebra system, these were matched as closely as possible to available hardware. The choice of an array of transputers as the hardware system lead to the development of a parallel lisp for such a system as there was none readily available. The result was a useful, experimental, coarse-grained, message-passing parallel lisp (Tlisp) based on the eval-server model where each processor in the array (configured in a tree) executes its own lisp process, reading, evaluating and printing s-expressions independently of the others, but with facilities for passing s-expressions to and from its neighbours.

Following an appraisal of algorithms available for calculating multivariate polynomial gcds, Zippel's sparse modular gcd algorithm was selected as the most suitable candidate for parallelisation and implementation in Tlisp.

Evaluation of the performance of both Tlisp and the parallel gcd algorithm was achieved by detailed examination of how they performed during execution for a range of test data.

The results of the performance experiments and the practical experience gained from a working parallel lisp system and its implementation of an important computer algebra algorithm clearly show that there is much potential for a medium- to coarse-grained message-passing parallel system in the quest for a suitable system for parallel computer algebra.

Acknowledgements

Many people have contributed in various ways to the completion of this thesis, I thank them all. Special thanks are due to my supervisor, Associate Professor Kevin Broughan, for his support and encouragement; to my fellow graduate students, especially Wayne Schou, for inspiration and help; to Associate Professor Ian Graham for continually changing and updating the transputer hardware and software but keeping a status quo just long enough for me to finish Chapter 5; to Terry Robb for his implementation of Zippel's sparse modular gcd algorithm; to the University of Waikato for their Postgraduate Scholarship and the DSIR (and later the MoRST) for their Women's Study Award. Finally, thanks to my husband, Mark, for his support.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1. Overview

This thesis is concerned with the intersection of two fields of study - computer algebra and parallel processing. After examining issues of hardware, programming models and language, a working parallel lisp system is developed and used to investigate the possibilities for a parallel multivariate polynomial greatest common divisor algorithm which in turn tests the parallel lisp system. Valuable insights into parallel computer algebra are gained from the practical experience with the parallel lisp system and with algebraic algorithms as important as those for finding polynomial gcds.

1.2. Computer Algebra

Computer algebra may be defined as "that part of computer science which designs, analyses, implements and applies algebraic algorithms" (Loos, 1983).

Algebraic algorithms differ from other mathematical algorithms in that the algebraic objects can be represented exactly, so that there is no loss of precision or significance during execution. When implemented as part of a system which supports input and output in the symbolic notation of algebra, algebraic algorithms can be used to solve problems exactly in many fields with a minimum of effort, and delivering results with a maximum of insight.

Applications for computer algebra have appeared in such research areas as genetics, chemical engineering, energy physics, celestial mechanics, general relativity and other areas of physics, number theory, projective geometry, Pade-approximants, ordinary differential equations, the study of fields, rings and groups, and other areas of mathematics. For further examples see Calmet and van Hulzen (1983).

Clearly computer algebra systems are important to the scientific community. Any improvements to such systems are therefore worthy of investigation. Much effort has gone into improving the solutions to important computer algebra problems. For example, polynomial factorisation was first achieved with Kronecker's algorithm dating from the eighteenth century but with the advent of computer algebra several new and more practical algorithms have been developed (Willcock, 1987). Another basic

algebraic problem which has received much attention is that of calculating the greatest common divisor of two arbitrary multivariate polynomials with integer coefficients. Several mathematical approaches to solving the gcd problem have yielded useful algorithms. Another approach, investigated in this thesis, is the possibility of using several processors simultaneously to solve the problem thus speeding execution time. Such a strategy comes under the general term of *parallel processing*.

1.3. Parallel Processing

Parallel processing is that part of computer science that deals with concurrently using more than one processing unit to solve a problem. As an area of research, parallel processing is rapidly gaining momentum as suitable hardware becomes more readily available (less expensive) and more diverse and flexible. It offers several advantages over sequential processing. On a sequential machine many computations, even those with moderately sized inputs, are too expensive in terms of time and memory requirements to execute. By distributing a computation over several processors both constraints of time and memory may be greatly reduced. Thus parallel processing has the potential to allow the solution of problems which were either not possible or took too long on a sequential processor.

Cost efficiency is also a factor. In many cases it is more cost effective to use many relatively slow but cheap processors in a cooperative way to achieve greater processing power than to use one fast but expensive serial computer.

Parallel processing is not without its problems. Many of these are faced by the system designer. For example, a design which offers the largest number of arithmetic operations per second might offer very limited inter-processor communication. A design which offers good communication might not compute as fast other designs and might not scale easily to very large numbers of processors. After the system designer has made the necessary decisions and compromises, the programmer is faced with the intricacies of designing and fine-tuning suitable parallel algorithms for the hardware.

To apply the benefits of parallel processing to any algorithm, an investigation into the amenability of that algorithm to parallelisation is necessary.

All algorithms consist of a number of tasks (groups of one or more instructions). Some of these tasks will be dependent on others (that is, if a task T_1 must occur before T_2 for correct program execution, then T_2 is said to be dependent on T_1). Any two tasks which are independent of each other may be executed concurrently or in parallel (for the purposes of this thesis the terms *concurrently* and *in parallel* will

be used interchangeably although some authors define a subtle difference between the two).

The size, (relative execution time), of the tasks determines the *grain size* of the (parallel) algorithm. If the tasks are small (for example, a single instruction), then the algorithm is said to be *fine-grained.* If the tasks are large the algorithm is said to be *coarse-grained.*

The grain size of a parallel algorithm will influence the type of hardware best suited to its execution. Parallel hardware comes in a variety of forms. These include pipelines, vector or array processors and networked processors. In a pipeline architecture, data is passed through a succession of processors one after the other with each performing a different set of instructions on the data, the computations being overlapped to exploit temporal parallelism. In a vector or array processor (single-instruction-multiple-data or SIMD processor), several tightly-coupled processors arranged in a rectangular grid execute the same instructions, each on a disjoint set of data under a centralised control thus achieving spatial parallelism. In a networked processor (multiple-instruction-multiple-data or MIMD processor), several processors are loosely coupled in some network configuration, each having its own storage and instructions thus achieving asynchronous parallelism. There are all manner of variations, modifications and combinations of these (the interested reader is directed to Krishnamurthy, 1989).

Other considerations to be taken into account when choosing a parallel system include whether or not the processing units have shared memory space and if not, how communications between processors, particularly non-neighbouring processors, are achieved; what memory and communication requirements the application has; how many processing units the system can support and of how many the application can make effective use.

1.4. Computer Algebra and Parallel Processing

There are two general approaches to finding a parallel solution to a problem. One is to start with a sequential algorithm (not necessarily the best) and modify (*parallelise*) it for some parallel system; the other is to start from scratch and design a parallel algorithm.

Algebraic algorithms present special problems for parallelisation in that being so diverse they are not easily classified as being suitable for any one particular type of parallel machine. For example, matrix manipulation algorithms may be very fine-grained and suitable for SIMD hardware, but algorithms for polynomial factorisation

or symbolic integration are very irregular and not suitable for that type of system.

Further, the behaviour of an algebraic algorithm is often determined by the structure of the output rather than the input (for example, the performance of the parallel greatest common divisor algorithm presented in Chapter 4 is primarily determined by the number of variables and terms in the output). Thus for a given problem and a given algorithm to solve it, the optimal hardware parameters (for example, how many processors to use and how to connect them) cannot often be predetermined.

The distinction between designing a parallel algorithm from scratch and modifying an existing one is often blurred. A *new* parallel algorithm may in fact be a parallelisation of a sequential algorithm that was never fully developed due to being inappropriate or inefficient for a sequential computer; or it may be partly a parallelisation of an sequential algorithm and partly new. To literally start from scratch with the design of parallel algebraic algorithms would in many cases mean to develop a new mathematical approach to the particular problem.

Thus in view of the years and effort already (and in most cases successfully) spent in finding sequential algorithms for algebraic problems, the approach of designing parallel algorithms for algebraic problems from scratch is possibly best left as a backup for particular problems when the first approach does not yield satisfactory results.

1.5. Structure of the Thesis

The next chapter of this thesis, Chapter 2, examines the problem of choosing a suitable grain size and architecture for a parallel computer algebra system. A transputer array is recognised as suitable hardware (within certain constraints) and a parallel lisp developed for use on that hardware following an investigation of existing parallel lisps.

The remainder of the thesis is concerned with an important algebraic problem and a parallel computer algebra solution. The problem is the calculation of the greatest common divisor of two arbitrary multivariate polynomials with integer coefficients. Chapter 3 looks at various sequential solutions that have been developed up until the present time and selects one, an adaptation of Zippel's sparse modular algorithm with improvements by Kaltofen and Yagati, as a candidate for parallelisation. The algorithm is presented and parallelised in Chapter 4. Chapter 5 investigates how well the parallel algorithm performs on the parallel lisp system.

Chapter 6 concludes the thesis with a summary and discussion of the research presented.

Chapter 2

A Parallel Functional Language

2.1. The Need for a Parallel Functional Language

Computer algebra systems have frequently been written in functional languages, in particular lisp which has features of dynamic storage allocation, garbage collection and the ability to refer to symbols by their names. Thus to investigate parallelism in algebraic algorithms, a parallel lisp (a lisp for programming a parallel computer) would be expedient. An array of transputers was chosen as the parallel machine for this research as they satisfy most of the theoretical requirements of the application. As there was no parallel lisp readily available for transputer systems, a decision was made to develop one.

In parallelising any language the nature of the intended application, the characteristics of the target hardware and the need for demonstrable performance enhancement through concurrency must be taken into account. In this chapter these aspects are looked at in some detail, existing parallel lisps surveyed and finally the development and design of a parallel lisp with basic constructs for parallelism is discussed.

2.2. The Intended Application: Computer Algebra

Several attempts, some of which are outlined below, have been made to identify the requirements of a parallel computer algebra system but with inconclusive results. It remains difficult to choose among the many parallel computation models and select the optimal one for computer algebra.

Ponder (1988a and 1988b) points out that few algorithms for algebraic manipulation are suitable for vector or systolic array computations due to lack of regularity in the algorithmic structure. Also, few are suitable for grid and message-passing architectures as these call for specific and unusual program organisations and have no global memory. His conclusion is that shared memory processors with both availability of global data and few restrictions on program organisation seem to have the most potential.

Recognising that different requirements are necessary for exploitation of parallelism in different algorithms, Buchberger (1985), after examining the alternatives, gives the

following objectives for a parallel machine for symbolic computation (which includes computer algebra):

(i) distributed control mechanism,

(ii) asynchronous cooperation of processors,

(iii) flexible interconnection topology,

(iv) each processor able to access the memories of its neighbours as easily as its own,

(v) processor modules all of the same type (homogeneous structure),

(vi) flexible synchronisation mechanism,

(vii) large number of low cost processor/memory modules (as opposed to a small number of expensive ones),

(viii) universal microprocessors as the core of the processor modules (rather than special purpose modules),

(ix) the machine as a whole universally programmable (rather than special purpose).

Buchberger's objectives are well reasoned and supported by algorithmic examples. After experience with the Tlisp system discussed later in this chapter the objectives appear commendable. It is notable that Buchberger reaches a similar conclusion about shared memory as Ponder.

One early parallel computer algebra system that has been developed is based on a simple design strategy reported by Watt (1985a, 1985b). A local area network of Vax 11/780s and the Maple system were combined to provide a prototype multiprocessor computer algebra system. When first invoked, the system set up a normal Maple session. The user could spawn Maple processes which were run on other processors. Spawned processes terminated when their evaluation was complete. The system provided facilities for processes to communicate with each other via message-passing (send, receive, and reply functions were reported to have been implemented).

Watt's system seems to be largely an illustration of how the parallelism constructs used in his research (AND-Parallelism where results are required from all the tasks and OR-Parallelism where only one result is required) might be programmed. He makes no comment as to the performance of the system, ease of its use or indeed if it was ever tested with parallel algebraic algorithms. From Watt's description of how the multiprocessing Maple system worked it is difficult to imagine that the overheads required to set up remote processes and to communicate with them would be outweighed by the gains of using remote processes, especially for the fine- to medium-grained applications (integer factorisation, sparse polynomial interpolation and Gröbner bases calculation) discussed in Watt's thesis.

The shortcomings of the parallel Maple system imposed by the hardware may be overcome by a better choice of hardware. To this end, Watt designed the system in two parts, keeping the multiprocessing facilities separate from the Maple kernel, thus allowing for a certain degree of portability.

Portability is an important feature of a computer algebra system (*cf.* Mayers and Hammerling, 1987), however the inherent differences between different types of parallel systems render portability of parallel software almost impossible without some loss of parallelism (for example, software for a system with shared memory is unlikely to be as efficient when adapted in a straight forward manner for hardware which relies on message-passing to share information between processors).

As yet the field of research into parallel processing has not yielded a truly portable model of parallelism. Since language design is portable, the practice of keeping the details of the implementation of the multiprocessing facilities separate from the programming language allows for as much portability as can reasonably be expected, at the current time.

2.3. The Hardware Choice: A Transputer Array

The transputer (Graham and King, 1990) is a VLSI microcomputer incorporating processor, memory, and four inter-computer data links on a single chip. It was developed to serve as a node computer in a regularly connected network of autonomous and nominally identical, concurrent computing elements. Communication between adjacent transputers is by synchronous message-passing, that is, messages are not passed until both sender and receiver are ready. It has potential for providing low-cost solutions to high performance processing demands across a wide range of applications.

Each transputer in a network is autonomous and it may communicate directly with up to four other transputers. There is no shared memory accessible by more than one transputer.

Thus a system of transputers is most suited to exploitation of medium- to coarse-grained parallelism where there is a high computation to communication time ratio for each job executed on a remote processor.

An array of transputers satisfies most of Buchberger's objectives: each transputer is autonomous and there is no central control mechanism; the processors' clocks are asynchronous; the interconnection topology is flexible as each transputer may be physically connected to four nearest neighbours and the Helios operating system (see

below) facilitates through-routing of communications to give almost any virtual configuration; processor modules are of the same type; synchronisation of processes is by message-passing, thus the synchronisation mechanism is flexible as any type of message can be passed (for example, data may be passed to effect data driven synchronisation); the processor/memory modules, transputers, are relatively inexpensive and there are no theoretical limits to the size of a transputer array; the processor modules are not special purpose and a transputer array is universally programmable.

There is only one objective, (iv), in Buchberger's list not satisfied by a transputer array. Transputers are not able to access the memories of their neighbours as easily as their own, but messages may be passed between processors. Using Buchberger's objectives as a standard and although one objective is compromised, a transputer array appears to be a suitable parallel machine for investigating parallel algorithms for computer algebra.

The transputer array used for this research consisted of a board of eight 25 MHz T800's and board of four 20 MHz T800's. The operating system was Helios (Perihelion, 1991), a distributed operating system which makes no assumptions about the topology of the transputers on which it runs, allowing logical configurations of the transputers. Helios provides an infrastructure for processes to locate and communicate with each other.

One advantage of using Helios is its ability to handle indirect communications without explicit control from the programmer. The programmer may assume any two processors have a direct link and can therefore communicate directly – Helios automatically handles any necessary indirect routing of messages. As a consequence the restriction that each transputer may only directly communicate with up to four nearest neighbours is relaxed as far as the programmer is concerned. In the interests of efficiency the programmer should, however, pay some attention to the physical configuration of the processors.

2.4. A Brief Survey of Parallel Lisps

Purely functional programs have a high potential for parallelism: since there is an absence of side-effects, all expressions are independent of each other and may be evaluated in any order or in parallel. However, lack of side-effects is considered by many to be an unnecessary restriction on a language. In particular, not being able to use variables to store temporary data leads to both inefficiencies of having to recompute and to a non-intuitive programming style. As a consequence most functional programming languages in widespread use (*e.g.* lisp) are not purely functional, in particular they include facilities for writing to memory locations (*i.e.* assignment of

variables).

Introducing parallelism into a functional language with side-effects is not without its problems. The main problem is that subexpressions lose their assumed independence resulting in a need for synchronisation of memory accesses and order of evaluations. Such synchronisation controls can be complex, resulting in complicated code.

As a method of synchronisation Burton (1987) suggests *annotating* (attaching to s-expressions labels or notes pertaining to their evaluation) a functional language to keep control of evaluation separate from program logic. In this way efficiency may be increased by giving the programmer explicit control of parallelism without affecting the clarity of an unannotated program or its semantics.

Burton's annotated functional programs are targeted at a system of loosely coupled processors viewed as a tree machine but could also be run on most other parallel machines.

Annotations are introduced to an existing sequential language such as lisp to control communication, synchronisation and placement of work with the effect of producing a collection of communicating processes. Burton defines the following annotations:

- *name*, *value* and *speculation* to override the default strategy for parameter passing (*speculation* denotes data driven parameter passing),
- *at* and *anywhere* to denote at which node a subexpression is to be evaluated.

Burton claims that in most cases annotation is not necessary – reasonable defaults ensure that the best evaluation strategy is selected in most situations. Only a few simple annotations are required to give a programmer a considerable degree of control over the runtime behaviour of a parallel functional program (further mechanisms for controlling evaluation may be created by encapsulating annotations within functions).

The idea of using annotations has been implemented in the design of most parallel lisps to date. The major parallel construct which has arisen is the *future* (Halstead, 1985). This annotation was first used in Multilisp (a parallel extension of Scheme) and is used in Qlisp (Goldman and Gabriel, 1988), Mul-T (Kranz, 1989), Concurrent Scheme (Kessler and Swanson, 1989) and other parallel lisps.

The expression (*future* X) where X is an arbitrary expression, immediately returns a future (*promise to deliver* or *place-holder*) for the value of X and concurrently begins evaluation of X. The future is said to be initially *undetermined*. When its value has been computed, that value replaces the future and the future becomes *determined*. When an operation requires the value of an undetermined future, then that operation is suspended until the future is determined (note that the value of a future will not

always be required, for example the s-expression *(or x y z)* may be evaluated by making x, y and z futures, then if the first to finish evaluating has value T, the others will not be required). There are many operations (such as assignment, parameter passing or placement in a data structure) that may be performed on undetermined futures thus increasing the degree of concurrency.

Qlisp (Goldman and Gabriel, 1988) is a dialect of Common Lisp implemented on the Alliant FX/8. As well as the *future*, Qlisp has procedures *QLET* and *QLAMBDA* which exploit features already in lisp which lend themselves to parallel evaluation. *QLET* is used to evaluate, in parallel, a number of arguments to a *LET* form, spawning a new process for each argument. *QLAMBDA* is used to create a closure and associate a new process with it. Subsequent invocations of that closure causes the calling process to perform parameter evaluation, and the closure process to do the appropriate lambda-binding, evaluate its body and return the result to the calling process. The closure process has a queue; multiple invocations of the same closure process will not create multiple copies of it. Programming experiments with Qlisp have found that *QLAMBDA* may be too low level to be easily used.

Qlisp also provides functions to create, acquire, release and test locks, as well as futures of the same type as in Multilisp. There are several constructs for the creation and forced evaluation of futures.

Mul-T (Kranz, 1989) is a parallel lisp system that has been developed to run on Encore Multimax NS32332 processors. It uses Multilisp's *future* but incorporates a parallel extension of the stopped computation idea for program development. A collection of tasks resulting from the evaluation of a single expression is called a *group*. Groups can be started, stopped, resumed or killed independently of other groups.

A project at University of California, Berkeley, has been involved with the building of the SPUR multiprocess workstation which was developed to study different styles of concurrent programming (Zorn *et al.*, 1988). SPUR Lisp, an extension of Common Lisp, was designed to run on the SPUR multiprocessor and be used as a research vehicle to understand and experiment with the many programming abstractions that have been proposed for parallel processing. Consequently SPUR Lisp has a set of multiprocessing features that could be used to implement a wide range of high-level abstractions including the *process*, a lisp object that embodies an independent thread of control; *mailboxes* for communication and synchronisation between processes; *signals* to allow processes to asynchronously interrupt other processes; and *futures* as in Multilisp. SPUR Lisp has a shared memory in that each process shares the heap with all other processes but has its own private stack and registers.

All the parallel lisps discussed above are designed for shared memory MIMD architectures. Parallel lisps for a system of autonomous processing units with no shared memory do not seem to have attracted the same amount of interest as those for shared memory systems. Consequently there are currently fewer of them under development.

Concurrent Scheme (Kessler and Swanson, 1989) is aimed at a distributed-memory MIMD machine, the Mayfly at the Hewlett Packard Research Laboratories. It has a function *make-thread* which is similar to *future*. However its main parallel construct is the *domain*, a small dynamically-created, monitor-like object which provides the basic mutual exclusion mechanism. It has the property that at most one thread of control can execute within it at any time thus providing a guarantee of mutual exclusion. *Domains* are intended as an improvement over Qlisp style locks for control over accesses to shared data.

Another approach for parallelisation of lisp on an architecture with no shared memory is the *eval-server* idea where a complete lisp system runs on each computing node and facilities are added to enable the nodes to communicate with each other in much the same way as they might read from or write to files. Some of the lisp processors serve as evaluators (*eval-servers*) for other lisp processors (*clients*) which control which s-expressions should be evaluated at another processing node. The eval-servers may themselves act as clients for their own eval-servers.

A project at Edinburgh University employed the eval-server model to implement a parallel lisp system on a transputer array using a Scheme interpreter with an interface to the Linda kernel[†] (Dourish, 1989).

The eval-server idea has also been used at INRIA in France, targeted at a transputer array. Starting with LeLisp they have installed a thread manager which allows process creation/communication and have added communication primitives. The main idea is to execute complete lisp systems on each transputer and allow these to communicate through low level messages (Piquer, 1989).

PMLisp and mUtilisp are two parallel lisps under development in Japan, both are for architectures with no global memory but have different features.

PMLisp (Yuasa and Kanawa, 1989) is a dialect of Scheme which runs on the PM1, a

[†] Linda is a memory model for distributed computers developed at Yale University. Processors have access to their own local memory and to a global memory called the *tuple space*. Communications between processors is by objects called *tuples* which pass through the tuple space (Gelernter, 1985).

MIMD, scalable architecture comprised of a network of low-cost, single board computers which act as processing elements. PMLisp uses a variant of the eval-server model, the key feature is that any data object including expressions, procedures and continuations may be passed between processing elements. PMLisp provides only primitive mechanisms for communication as built-in functions; it is up to the user to decide how to use them, typically by defining a function to intercept evaluations and apply the primitive mechanisms as appropriate. This flexibility of communication is coupled with the notion of *network models* which allow the user to implement programs that depend on a particular network configuration independently of the physical network, thus providing a suitable tool for experimentation with massively parallel computation.

Like PMLisp, mUtilisp (Iwasaki, 1989) has explicit parallelism and excludes sharing of lisp objects and environments. Parallelism is achieved through named processes. Each process is named when created so that the user can handle processes explicitly. The process name is used in functions related to processes (such as interprocess communication). There are three basic constructs, namely *create-process, send* and *receive*, each with the obvious meaning. Processes are created dynamically and interprocess communication is by asynchronous message-passing. Because of the explicit function calls needed to control processes, mUtilisp emphasises the procedural aspects of lisp rather than the functional aspects. mUtilisp simulates multiprocessing on a Unix machine with an MC68000 series microprocessor but it is not, at the present time, available for a multiprocessor.

There are other parallel lisp projects currently in various stages of development but not reported here due to such factors as the main features being superseded by later research, the project being shelved, little or no information in the available literature, or the language being too specific to a machine or purpose not relevant to this research.

Clearly there is much research and development underway in the field of parallel lisps, but as yet no standards have emerged and no system stands out as using the best approach. None of the parallel lisps discussed above are readily available for a transputer system, therefore development of a simple parallel lisp for experimental purposes on transputer architectures was appropriate for research into parallelising algebraic algorithms.

2.5. The Tlisp System

Tlisp is an experimental concurrent lisp based on Xlisp[†] (version 2.0) and running on a transputer network. Motivation for development of Tlisp was the need for a concurrent lisp for investigation of coarse-grained parallelism in algebraic algorithms, targeted at message-passing architectures.

The name Tlisp will be used to refer to the extended Xlisp language, a single process executing the Tlisp language and to the whole system of communicating Tlisp processes. Which of the meanings is intended will be clear from context.

The eval-server model of parallelism (see section 2.4) is employed with each transputer in the network independently reading, evaluating and printing s-expressions. Ideally this model suits a network of processors where each may serve as an "evaluator" for any other as the need arises. This ideal has been compromised in the design of Tlisp; the processors are configured in a tree so that each processor may use only its children processors as *eval–servers*. This is discussed in more detail in section 2.5.2.

Tlisp has a small set of functions which allow any processor in the network to send an s-expression to one of its eval-server nodes for evaluation and receive an s-expression sent back as a result of the "remote" evaluation.

Conceptually Tlisp can be thought of as one processor executing Xlisp with added features to enable the user to specify that some s-expression evaluations are to take place on another (remote) processor while local execution resumes normally until the remote value is required.[††] The remote processors act as "evaluation servers", running a version of Tlisp which is essentially Xlisp with no I/O other than the *read* and *print* of the *read-eval-print* loop. They each execute a *read-eval-print* loop reading input from and printing output to its client processor. These evaluation servers also have added features for creating remote jobs (s-expressions which are to be evaluated on another processor). Note that the Tlisp system does not have a shared memory but it is possible to create read-only variables that are global to all the processors.

In the next two subsections (2.5.1 and 2.5.2) an overview of the system, its

[†] Xlisp is a public domain experimental programming language, developed by David Betz. It was originally implemented to allow experimentation with object-oriented programming. It is a readily available lisp dialect written in C and has the advantages of being portable, small and easily extended.

[††] The remote evaluation discussed in this thesis is similar to, but not as general or flexible as, the remote evaluation formalised and discussed by Stamos and Gifford (1990). The differences between the two models are outlined in Stamos and Gifford's paper.

components and configuration is given. Section 2.5.3 looks at the concurrent functions of Tlisp in more detail, describing how they work and are used. The details of how communications between a client and its eval-servers are coordinated are described in section 2.5.4. An example program is given in section 2.5.5 and finally section 2.5.6 gives a discussion of the performance and short-comings of Tlisp.

2.5.1. Tlisp System Components

The Tlisp system is comprised of three basic components – the root process, the eval-server processes and the communications coordinator processes. The root process and the eval-server processes will both be referred to as Tlisp processes when no distinction is necessary.

The root process executes the top-level *read-eval-print* loop of Tlisp performing input and output as normal with the user-interface and includes several functions which allow it to send messages to and receive messages from its child processes.

Each child process is called an *eval-server* since it serves as an s-expression evaluator for its parent process (the *client*). Eval-servers have limited side-effects, for example there is no I/O other than the *read* and *print* of the *read-eval-print* loop. If an error is created during evaluation then a *remote-error* datum is returned, a message sent to the user interface and the *read-eval-print* loop is reset and resumes execution at *read*. The eval-servers may also act as client processes with their own eval-server processes.

Note that there is a set number (user-defined, but restricted by the number of processors available) of eval-server processes created and logically connected at startup time, they are not dynamically created.

All traffic between a client process and its eval-server processes is routed through a multiplexing/load-balancing light-weight process called the communications coordinator process (CCP). This strategy keeps communication and computation separate (see Figure 2.1). The communications coordinator process is described in detail in section 2.5.4.

S-expressions are passed between processes as print-strings (null-terminated character strings which give a linear representation of the s-expressions). Each eval-server reads print-strings from its client and prints print-strings back to the client (via the CCP). The pros and cons of passing print-strings is discussed in section 2.5.6.

2.5.2. Tlisp System Configuration

Tlisp is designed for a network of transputers configured as a farm (set of replicated structures) of an arbitrary number of small trees as shown in Figure 2.1 where each node on the tree represents a transputer executing the named process(es). Each processor in the tree may communicate with its parent or with any of its child processors; the root processor also interacts with the user-interface.

With the simple communication strategy adopted for Tlisp (each processor in direct communication with its own eval-servers and client, if any) the configuration possibilities are limited. The ideal would be to have a more complex method of distributing jobs amongst eval-servers such that any job awaiting evaluation could be sent to any eval-server awaiting input regardless of where the job originated.

Consideration was given to the configuration of Figure 2.2. The central process queues jobs from the peripheral eval-servers and the client process, and distributes them to idle eval-servers. Jobs are queued until an eval-server becomes available. This model is flawed with the probable event of deadlock as in the following instance.

Consider a simple case of there being only two eval-servers and the first receives a job which creates a remote job. Both eval-servers are now busy, with the first unable to complete before the second. If the second eval-server creates a remote job it will be put in the queue until an eval-server becomes available. Clearly no eval-server is ever going to become available for the waiting job – the first eval-server is waiting on the second which is waiting on the job in the queue. A deadlock has occurred.

As a possible alternative the central process could act merely as a supervisor with no facilities for storing waiting jobs. When a process (the client or an eval-server) creates a remote job the supervisor could either direct the job to an idle eval-server or, when there are no idle eval-servers, tell the process where the job originated to evaluate it locally. This model is also flawed. For example, if the client has several independent jobs to be evaluated at once and when the first is created all the eval-servers are busy, then the client sets about locally evaluating the job thus blocking the creation of the remaining jobs. Meanwhile some other eval-servers may become idle resulting in processes waiting for work and jobs waiting to be processed (but as yet uncreated).

The configuration of Figure 2.2. requires a more complex model for job distribution than the two simple models discussed here. Implementing such a complex job distribution strategy is beyond Tlisp's major design objective of simplicity.

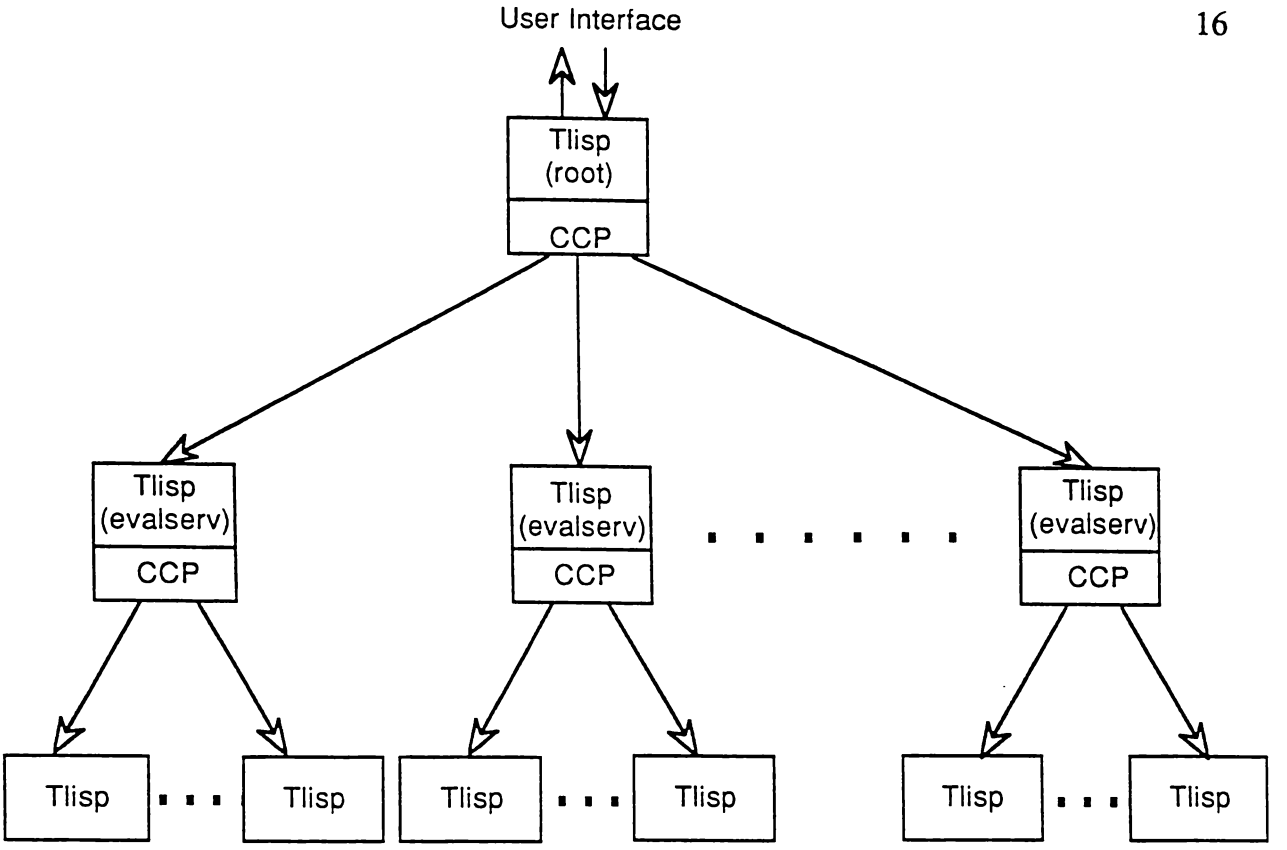

Figure 2.1 *Configuration of the Tlisp System*

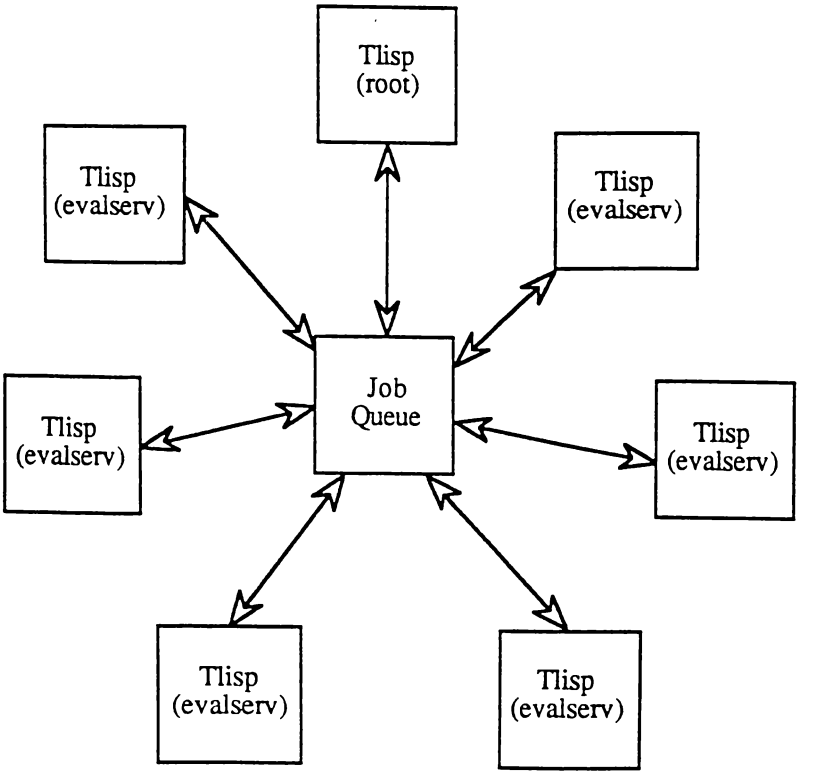

Figure 2.2 *A Possible Configuration*

The farm-of-trees configuration was chosen for Tlisp after experimental observations with other configurations. Initially a full binary tree was used to take advantage of the recursive nature of many algebraic algorithms (in particular, the greatest common divisor algorithms investigated in Chapters 3 and 4). This often resulted in those processors distant from the root being idle while jobs waited further up the tree. A simple farm configuration (a master processor with an arbitrary number of slaves as eval-servers) was not entirely satisfactory either – the applications generally did not have enough parallelism to use more than a few slaves at once while the jobs being sent to the slaves contained parallelism which could not be exploited.

The farm of small trees seems to be a satisfactory compromise. It is flexible enough to accommodate as special cases a binary tree of depth three or a simple farm configuration (a farm of trees each of depth one). The user may specify what size and how many trees depending on the application and the number of processors available.

2.5.3. The Modifications and Additions to Xlisp

The main modification to Xlisp required to implement Tlisp was the addition of four functions to support concurrent computation. The functions are *remote*, *request*, *broadcast* and *evalp*. The functions *remote* and *request* are used, respectively, to force remote evaluation and retrieve the results of such evaluation. The function *broadcast* is used to propagate function definitions and s-expressions throughout the network. The predicate *evalp* is used to check if a remote evaluation is finished. All the functions are described in more detail below and an example of a program using *remote* and *request* is given in section 2.5.5.

Other modifications to Xlisp include the addition of matrix and multiple precision integer (big number) data types as these are needed by the algebraic algorithms to be implemented in Tlisp. Only the additions for concurrent computation will be discussed in this section.

Based on Multilisp's *future* (Halstead, 1985) the function *remote* is used to send s-expressions to eval-servers. It takes three arguments:

(remote 'symbol pred 'sexpr)

symbol – a symbol to be given the value of *sexpr* after evaluation (the address of *symbol* is used internally as a reference key),

pred – a predicate which indicates whether evaluation should take place on a remote Tlisp process or locally (for example, *pred* might depend on the number of eval-server processes the local process is in communication with, or on the computation size of some part of *sexpr*),

sexpr – the s-expression to be evaluated. If this is a function call (and it usually will be) the arguments are evaluated locally and their values passed to the eval-server. Note that since there is no shared memory between processors any function definitions or (read-only) global variables used during evaluation of *sexpr* must be known to all eval-server processes before the *remote* call is made.

If *pred* is non-nil, *remote* passes the print string of *sexpr* with the address of *symbol* (as a reference key) to the CCP and immediately returns a *placeholder* datum which is bound to *symbol*. Otherwise *sexpr* is evaluated locally and its value bound to *symbol* and returned.

As an example consider the s-expression

(remote 'x (> n 4) '(factorial n))

where *n* has some value in the environment. If *n* is greater than 4 then *(factorial n)* will be sent to an eval-server for evaluation. The *remote* call returns a placeholder value which is bound to *x*. When *n* is not greater than 4 the evaluation takes place locally and the resultant value returned and bound to *x*.

The function *request* is used to retrieve, from the CCP, values of expressions evaluated by eval-server processes. It takes one argument:

(request 'symbol)

symbol – the symbol associated with the *remote* call whose value is now required.

If *symbol* is not a symbol then it is returned unevaluated. If the value of *symbol* is not a placeholder datum then its value is returned. Otherwise, a message is passed to the CCP and the calling process waits until the requested value has been received by the CCP (execution may be held up indefinitely waiting for this value). Once available the requested value destructively replaces the placeholder datum bound to *symbol* and is returned by *request* (*NIL* is treated as a special case since it has a fixed address making destructive replacement impossible). Thus any data structure containing the placeholder before the call to *request* will contain the placeholder's value after the call.

The function *request* checks returned s-expressions and enters an error state if a *remote-error* datum is returned, printing the error message which was generated at the remote process.

As an example consider the s-expression

(request 'x)

at some time after the remote call in the above example. Regardless of whether *(factorial n)* was evaluated locally or remotely, the *request* will return its resultant value

and destructively bind it to *x*.

In order to avoid long delays waiting for a remote value to become available (when there is some alternative action that can be taken if the required value is not yet available) the predicate *evalp* may be used before *request*. It is called in the same manner:

(evalp 'symbol)

but returns *T* or *NIL* indicating whether or not the remote value is currently available at the CCP. To increase efficiency *evalp* has a side-effect: when *T* is returned, the sought after value is also acquired from the CCP and bound to *symbol*. Thus after a successful *evalp* call a subsequent *request* call will not require further communication with the CCP.

The function *broadcast* is used to broadcast s-expressions and function and macro definitions from the root Tlisp process to all eval-server processes in the network:

(broadcast s1 s2 ...)

si – a closure (user-defined function) or macro, or an s-expression.

Function/macro definitions are propagated throughout the entire network via the communication coordinator processes. For example

(broadcast #'f)

where *f* has been defined as a function (or macro), sends the definition of *f* to each of the eval-servers. Eval-server processes receiving function definitions which have been broadcast, automatically broadcast the definition to their own eval-server processes (if any) thus propagating the definition through the entire network.

S-expressions are broadcast to the eval-servers then evaluated locally. For example

(broadcast '(setq global_x 0))

sends the entire s-expression *(broadcast '(setq global_x 0))* to each of the eval-servers, then sets *global_x* to be zero locally. At each eval-server the broadcast procedure is repeated thus propagating the value of *global_x* throughout the network. The variable *global_x* may now be treated as a *read-only* global variable.

The function *broadcast* returns *T* after the CCP has sent back an acknowledgement signal indicating that the broadcast has started (the broadcast does not start until any backlog of s-expressions awaiting remote evaluation has been cleared); *NIL* is returned if there are no eval-servers.

Three new symbols added to Xlisp are **nr_evalservers**, **processor_id** and **remote_monitor**. As suggested by their names, **nr_evalservers** is the number of

eval-servers a Tlisp process has and **processor_id** is a number unique to each Tlisp process in the network, where the numbering starts at 0 at the root and is incremented in a breadth-first manner through the tree of lisp processes. Both these symbols are initialised at start-up and should be treated as read-only.

The symbol **remote_monitor** is used as a flag for monitoring input and output at eval-server processes. If **remote_monitor** has nonzero value on eval-server *n* then all input and output for that evalserver will be recorded in a file called *iofile*n. The purpose of **remote_monitor** is to facilitate debugging and analysis of the behaviour of Tlisp programs. This symbol is zero by default but may be initialised at start-up or modified using the *broadcast* function.

The Tlisp system does not have any facilities for fault tolerance – if a processor or inter-processor link fails then Tlisp does not make any allowances for the failure and the behaviour of the system becomes unpredictable.

2.5.4. The Communications Coordinator Process

The communications coordinator process (CCP) is one of the three basic components of the Tlisp system. Each client process has a CCP for coordinating communications between that client and its eval-servers.

Messages to eval-server processes from the client process can take one of four forms:

1. an s-expression to be evaluated and its reference key,
2. a function/macro definition or s-expression to be broadcast,
3. a reference key for an evaluated s-expression to be returned (*request*),
4. a reference key to check if a form has been evaluated (*evalp*).

Each message consists of a null-terminated character string and an integer. The first character of the string is a code number indicating the type of message and the remainder of the string is an s-expression, function definition or empty. The integer part of the message is always a reference key.

The CCP deals with each message according to its code number. Three threads[†] are required to handle client-to-eval-server messages as shown in Figure 2.3. – one (A) for handling all client to eval-server traffic, one (B) for queuing remote jobs and one (C) for handling s-expressions that have been returned from the eval-servers. Another thread (D) is needed for handling eval-server to client traffic.

[†] A thread is a process created (forked) by another process. Both run on the same processor which time-slices between them, and share the same memory space.

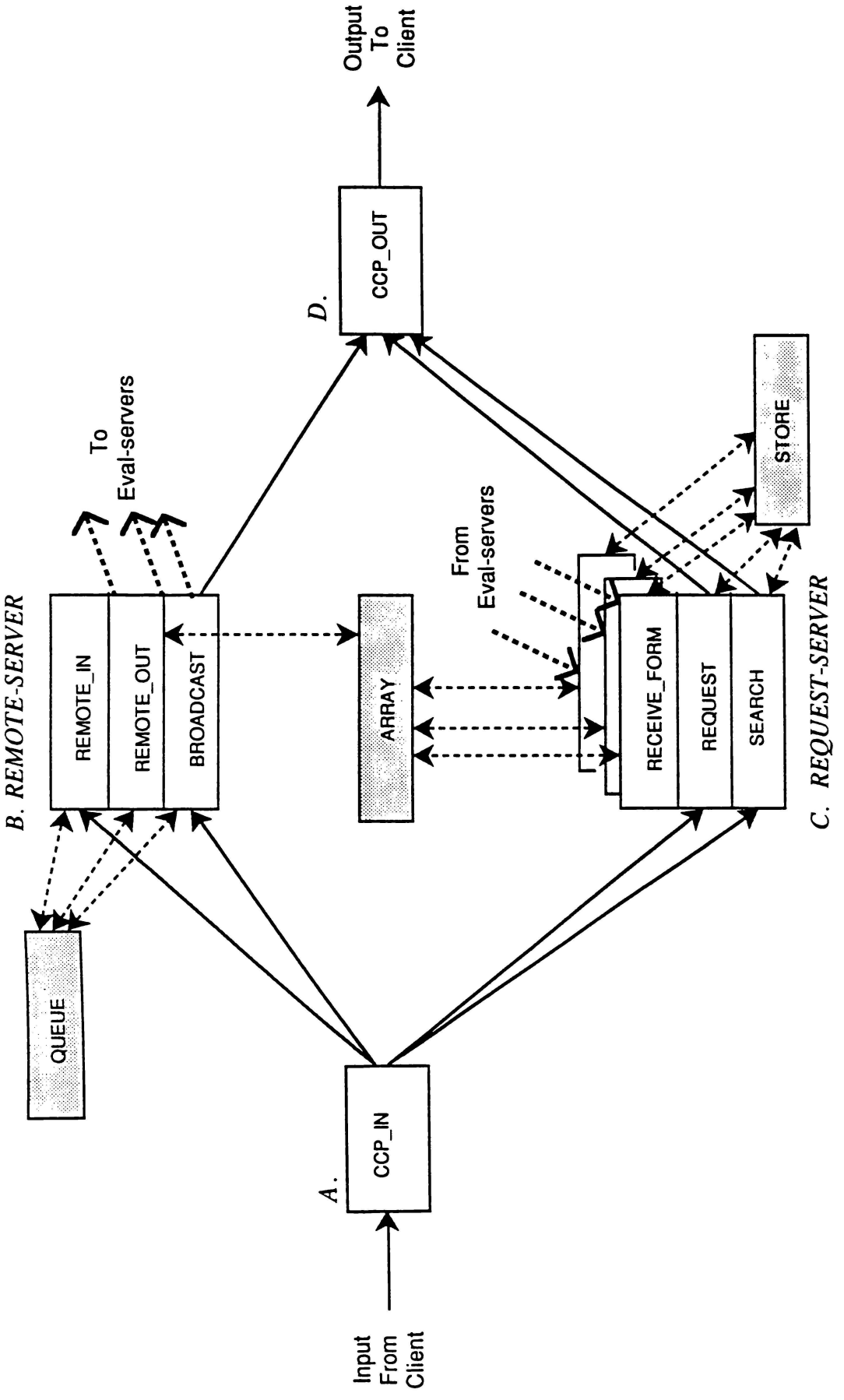

Figure 2.3 *The Communications Coordinator Process*

The threads which make up the CCP use three main data structures. Thread B uses a FIFO queue for queuing s-expressions awaiting remote evaluation. Thread C has a store for storage of s-expressions (indexed by their reference keys) which have been returned by the eval-servers. There is an array shared by threads B and C for keeping a record of which jobs are currently being evaluated on which eval-servers and thus a record of which eval-servers are waiting for input.

The threads communicate via *soft-links*. A soft-link is implemented as a memory location in the shared heap space. It is protected from simultaneous accesses by semaphores.

Thread A (*ccp_in*) is responsible for receiving all communication from the client process. It waits for incoming messages and sends them to either thread B or thread C along the appropriate soft-link according to the type of message.

S-expressions for remote evaluation are sent to thread B where they are placed in a FIFO queue. Thread B (*remote_server*) sends the queued s-expressions to eval-servers as they become available for input. A reference key to identify the job created by the remote evaluation is stored in an array indexed by the number of the eval-server to which it has been sent. Function/macro definitions to be broadcast are passed from A to B where they are held until the queue is empty indicating all waiting remote jobs have been sent out. The definition is then sent to each of the eval-servers and an acknowledge message returned to the client (via thread D). Note that since the client is waiting for the acknowledge signal, while a broadcast is waiting at B no messages are being added to the queue at B.

Thread C (*request_server*) waits for output (being results from remote calls) from the eval-servers. There is one *receive_form* thread for each eval-server. On receipt of a message from an eval-server the *receive_form* thread retrieves the appropriate key number from the array where the *remote_server* stored it. The returned s-expressions and their reference keys are stored by thread C until requested. When the store is given an s-expression for which it already has an entry with the same reference key (this occurs when two *remote* calls are made using the same symbol but the first is not *requested* before the second is made), the existing entry is overwritten.

Any *request* and *evalp* messages from the client are passed from A to C where the store is consulted. In the case of *request* the store is searched continuously until the key is found – the corresponding s-expression is then passed back to the client (via D) and the entry removed from the store. In the case of *evalp*, the store is searched once only and '1' or '0' returned to the client (via D) depending on whether or not the required entry was found in the store. If the search was successful the required s-

expression is also passed back to the client at the same time.

Thread D (*ccp_out*) simply acts as a common point for directing messages from threads B and C to the client.

The main pieces of source code, written in C, for the threads described above are given in Appendix A.

2.5.5. An Example Tlisp Program

The design of Tlisp imposes certain constraints on the design of algorithms which will perform well on the system. There are at least two types of problem solution strategies that are especially suitable for the Tlisp system. One is the divide and conquer strategy where a problem is divided into subproblems recursively until the solution is simple. Each subproblem may be sent to an eval-server where the divide and conquer algorithm is applied recursively.

The other strategy is a special case of divide and conquer. When a solution involves a number of iterations of a loop where each iteration is independent of the others it can be parallelised by making a remote job out of each iteration or a group of iterations such that the job size is sufficient to warrant sending it to an eval-server. A farm of eval-servers is an appropriate configuration for such a problem unless the iterations may themselves be parallelised, in which case a tree makes better use of the number of processors available.

In this section the divide and conquer strategy and its applicability to Tlisp is investigated. As an illustration of how *remote* and *request* might be used in practice and to gain some insight into the behaviour of Tlisp and the overheads associated with making *remote* calls, a simple divide and conquer algorithm to calculate Fibonacci numbers is investigated.

A general divide and conquer function definition may be given as follows:

```
(defun  Divide&Conquer (problem)
   (cond ((conquered? problem) solution)
         (t (apply #'Conquer
                   (map #'Divide&Conquer (Divide problem))))))
```

Parallelisation of such a function is simply a matter of using a parallel map which creates a remote job for each subproblem (and which would be defined in Tlisp using *remote* and *request*).

Consider, for example, a parallel Fibonacci number function:

```
; parallel Fibonacci number generator
; using the general divide and conquer technique
(defun pfib (n)
   (cond ((or (= n 1) (= n 2)) 1)
         (t (apply #'+
                   (parallel-map #'pfib (list (- n 1) (-n 2)))))))
```

On a Tlisp binary tree (clearly for *pfib* a binary tree will be the best configuration) the remote jobs would propagate as illustrated in Figure 2.4. Unless the tree were sufficiently large enough to accommodate the entire problem, the leaf nodes will clearly perform the bulk of the work. The non-leaf nodes on the tree do very little work and so the system is not efficiently utilised.

Redefining *pfib* for Tlisp as below without the overheads of using *apply* and *parallel-map* the overheads of making *remote* calls may be studied.

```
; parallel Fibonacci number generator
(defun pfib (n)
   (prog (x1 x2)
      (return (cond ((or (= n 1) (= n 2) 1)
                     (t (remote (setq x1 (gensym)) (> n 10) '(pfib (- n 1)))
                        (remote (setq x2 (gensym)) (> n 10) '(pfib (- n 2)))
                        (+ (request x2) (request x1)))))))
```

The function *gensym* is used to generate unique names for each of the placeholders *x1* and *x2* at each recursive call of *pfib* so that the CCP can distinguish between them. Using *(> n 10)* as a predicate prevents very small jobs (*(pfib n)*, where *n* is less than 10) from being sent to an eval-server for evaluation.

Using a binary tree of depth two the problem of computing the *n*th Fibonacci number using *pfib* is divided up as shown in Figure 2.5.

The total computation time should be dominated by the computation time of (*pfib* (– *n* 2)) as this is the largest job given to the leaf nodes. The following table shows that this is not the case. Times are given in 1/100 sec and *fib* is a typical sequential function for generating Fibonacci numbers.

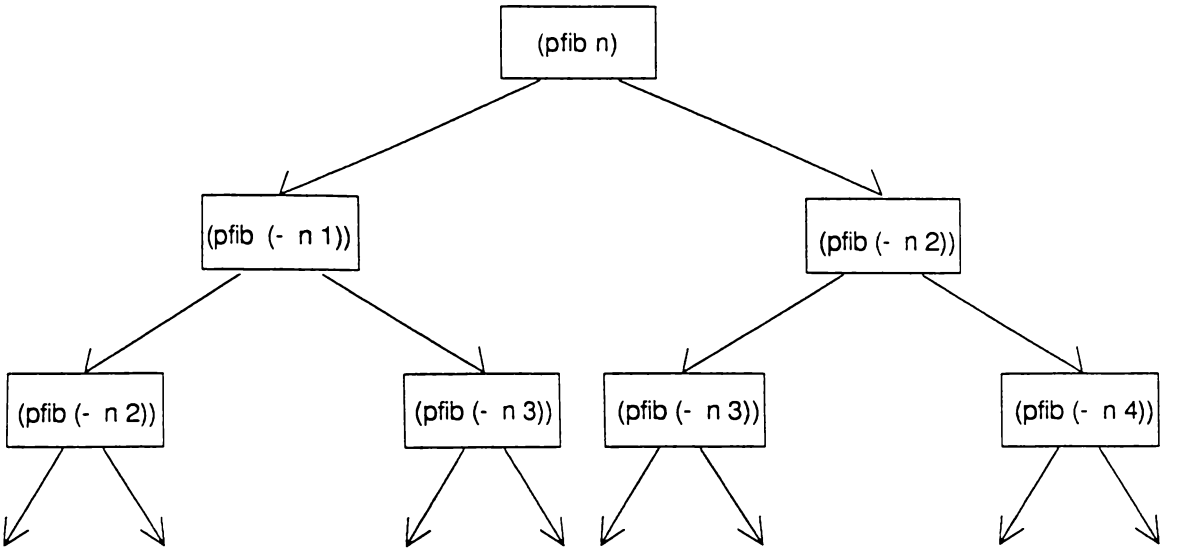

Figure 2.4 Binary Tree Evaluation of (*pfib n*)

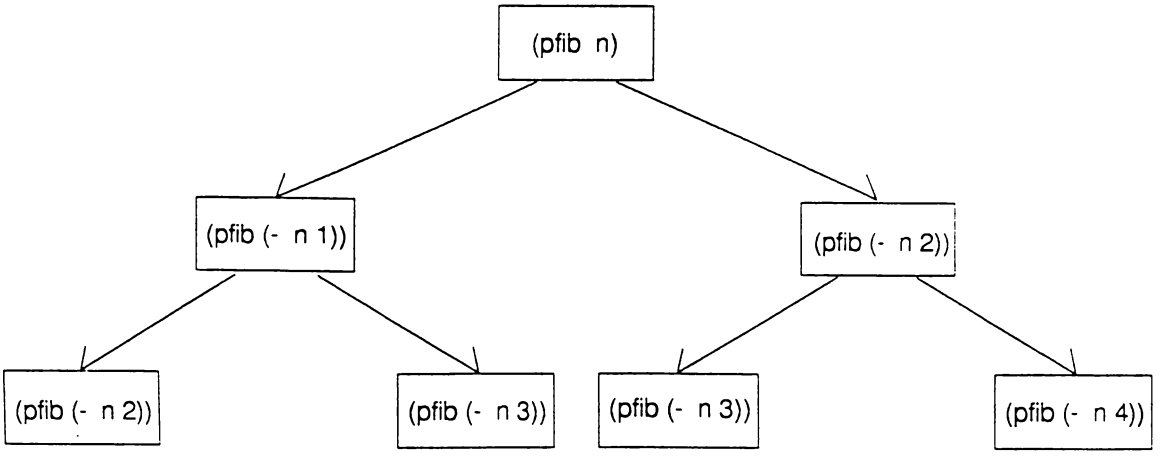

Figure 2.5 Depth 3 Binary Tree Evaluation of (*pfib n*)

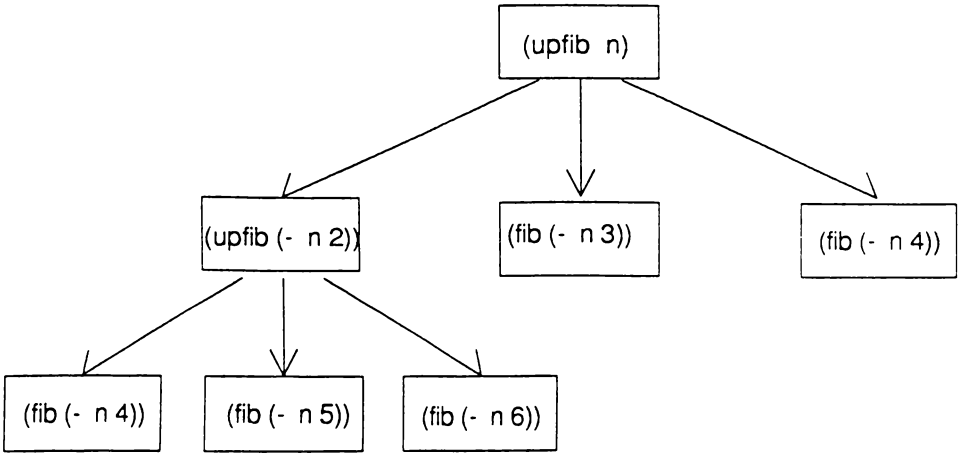

Figure 2.6 One-Sided Binary Tree Evaluation of the Unfolded (*upfib n*) function

n	*(fib n)* sequential time	*(pfib n)* parallel time	*(fib (– n 2))* sequential time
15	116	120	42
20	1296	1253	495
25	14465	13981	5487

The overheads of making redundant remote calls at the leaf nodes have outweighed the gains of concurrently solving several smaller problems. The number of redundant *remote* calls made is bounded below by the formula

$$C(n) > (\frac{3}{2})^{n-1} - 1$$

where $C(n)$ is the number of calls to *pfib* required to calculate the nth Fibonacci number. Thus, when calculating *(pfib 20)* as above, the number of redundant *remote* calls in the dominant computation, *(pfib 18)*, will be more than 984.

A better parallel solution would be to execute a different function at the leaf nodes. The following table gives the times for the same problems as above using the same configuration (Figure 2.5). These times were achieved by having the leaf node execute the sequential function *fib*.

n	*(fib n)* sequential time	*(pfib n)* parallel time	*(fib (– n 2))* sequential time
15	116	55	42
20	1296	512	495
25	14465	5623	5487

This second table of timings shows increased performance from lowering the overheads on the leaf nodes where the majority of the computation takes place. With these overheads reduced it is now evident that the computation time is dominated by *(fib (– n 2))*.

For further performance improvements the problem can be *unfolded* before being divided to increase the amount of work done by the divide stage. Unfolding is a programming technique that is not restricted to parallel programming. In a parallel environment it can better utilise a small number of processing elements. Unfolding *pfib* gives the following definition:

```
; unfolded parallel Fibonacci number generator
(defun upfib (n)
   (prog (x1 x2 x3 x4)
      (return (cond ((or (= n 1) (= n 2)) 1)
                    ((= n 3) 2)
                    ((= n 4) 3)
                    (t (remote (setq x1 (gensym)) (> n 10) '(upfib (- n 2)))
                       (remote (setq x2 (gensym)) (> n 10) '(upfib (- n 3)))
                       (remote (setq x3 (gensym)) (> n 10) '(upfib (- n 3)))
                       (remote (setq x4 (gensym)) (> n 10) '(upfib (- n 4)))
                       (+ (request x4)
                          (request x3)
                          (request x2)
                          (request x1)))))))
```

which can be improved by not duplicating the *(upfib (– n 3))* computation, giving

```
; improved, unfolded parallel Fibonacci number generator
(defun upfib (n)
   (prog (x1 x2 x3)
      (return (cond ((or (= n 1) (= n 2)) 1)
                    ((= n 3) 2)
                    ((= n 4) 3)
                    (t (remote (setq x1 (gensym)) (> n 10) '(upfib (- n 2)))
                       (remote (setq x2 (gensym)) (> n 10) '(upfib (- n 3)))
                       (remote (setq x3 (gensym)) (> n 10) '(upfib (- n 4)))
                       (+ (request x4)
                          (* 2 (request x2))
                          (request x1)))))))
```

Figure 2.6 shows how the problem of computing the nth Fibonacci number may be divided among the same number of processors as before but giving a dominant computation time of that required for a sequential computation of the $(n-3)$-th Fibonacci number. Times given in table below back up this claim.

n	*(fib n)* sequential time	*(upfib n)* parallel time	*(fib (– n 3))* sequential time
15	116	39	26
20	1296	322	308
25	14465	3525	3415

From the example programs given above and from practical experience with Tlisp the

following points to consider when designing a Tlisp program may be given.

(i) Where *remote* calls are made recursively the overheads of redundant *remote* calls at leaf nodes should be considered. Where there is deep nesting of *remote* calls as in the Fibonacci functions, non-parallel function definitions could be used at leaf nodes.

(ii) The most obvious division of the problem and system configuration will not always make efficient use of a given number of processors. *Unfolding* the solution may help to spread the work load.

(iii) For efficiency some attention should be paid to how much work will be done on the non-leaf nodes while the leaf-nodes are busy.

2.5.6. Discussion

Interpreted Tlisp code has a slow computation time. Obviously the solution to this would be to compile the code first – however there is no compiler available for either Tlisp or Xlisp. In obtaining optimal parallelisation of a coarse-grained system such as Tlisp the aim is to generate jobs which have a high computation to communication time ratio. The unnecessarily high computation time of interpreted Tlisp will to some extent distort measures of speedup and efficiency.

Poor performance due to lack of a compiler is a reported problem with other parallel lisps, for example PMLisp (see section 2.4).

Communication time, when defined as the total time to pass an s-expression from one Tlisp process to another (including time for *destructing* the s-expression to a character string, transmitting the string from Tlisp to CCP to Tlisp and *restructing* the character string into an s-expression) is not optimal either. The merits of Buchberger's objective of having each processor able to access its neighbour's memory as easily as its own (see section 2.2) are evident when the overheads of passing messages between processors are taken into account (see section 5.1.3 for a quantitative discussion of these overheads).

As mentioned earlier, s-expressions are passed between processes as print-strings. Although print-strings are one of the shortest representations of s-expressions the overheads of de/re/structing lisp objects into and out of print-strings are high. Another short-coming is the loss of original structure. For example, consider the object *bar* created by

```
> (setq foo '(1 2 3))
> (setq bar (cons foo foo))
```

If *bar* is destructed into its print string and restructed into a lisp object it will no

longer be a cons cell in which both pointers point to the same address. A related and more serious problem arises with infinite structures.

There are alternatives to using print-strings. One is to send structure descriptions (Dourish, 1989). Descriptions preserve the structure and lower the overheads at the *read* end of the communication. PMLisp uses the same idea by way of a simple encoding scheme for lisp objects (Yuasa and Kanawa, 1989). A major drawback of this idea is the potential size of the descriptions which have to travel between processes (although the use of cdr-coding, a technique for compact list representation (Baker, 1978), has been suggested as a solution to this problem).

Another method of communicating lisp objects is that of simulating a "by reference" transfer of data, investigated by Piquer (1989). Lisp objects are destructed into a stream of bytes at the *write* end of the transfer and restructed at the *read* end. If a stream of bytes begins to be too large a remote pointer is sent to the rest of it. At the remote site if the *untransferred* data is accessed an *object-fault* is produced and the data is then copied across. This method of lisp object communication preserves the data structure and is (potentially) very efficient.

The decision to persist with print-strings for s-expression communication in Tlisp was determined by their ready availability and the design objective of simplicity for Tlisp.

Tlisp does have some other short-comings. One of these is the lack of a signal mechanism, once an s-expression has been sent off for remote evaluation, the evaluation cannot be aborted, delayed or suspended. Without a facility for aborting computation and thus freeing up limited computation resources the use of speculative computation (eager evaluation where the result(s) of the evaluation may be unnecessary (Osbourne, 1989)) is not possible. The design of Tlisp has made allowances for the introduction of remote signal facilities by having the CCP keep a record of which s-expressions are being evaluated on which eval-servers. The obstacle to implementing signals for Tlisp is the simple fact that the Xlisp signal mechanisms do not work in the Helios environment and time constraints on this thesis have prevented further work on them.

Chapter 3

Polynomial Greatest Common Divisors

3.1. Basic Algebraic Operations

Rational functions are an important data type for algebraic manipulation systems. Arithmetic operations for rational functions are based on rational polynomial operations and since most rational polynomial problems can be reduced to problems in $\mathbf{Z}[X_0, \ldots, X_v]$ a basic set of algebraic operations on polynomials with integer coefficients is essential for any system which manipulates rational functions.

This basic set clearly must include addition, subtraction, multiplication and division (with remainder). A less obvious but necessary operation to include is evaluation. One further operation that is important enough to include in this set of basic operations for elements of $\mathbf{Z}[X_0, \ldots, X_v]$ is that of finding greatest common divisors. This operation is crucial to rational function simplification which is necessary for an efficient and useful algebraic manipulation system. Polynomial greatest common divisors are also important for polynomial factorisation and squarefree decomposition, symbolic integration, solving diophantine equations, cryptography and many other applications.

A greatest common divisor of two multivariate polynomials F_1 and F_2 may be defined as a multivariate polynomial G such that

(i) $G \mid F_1$ and $G \mid F_2$

and (ii) if H is a multinomial such that $H \mid F_1$ and $H \mid F_2$

then $H \mid G$.

A greatest common divisor is only determined up to multiplication by an arbitrary unit. The *unique preferred greatest common divisor* which can be referred to as *the* greatest common divisor or simply *gcd* may be defined: if a and b are integers then $\gcd(a, b)$ is positive; in $\mathbf{Z}[X_0, \ldots, X_v]$ the leading integer coefficient of $\gcd(F_1(X_0, \ldots, X_v), F_2(X_0, \ldots, X_v))$ is positive and in $\mathbf{F}[X_0, \ldots, X_v]$ where $\mathbf{F}$ is a field, $\gcd(F_1(X_0, \ldots, X_v), F_2(X_0, \ldots, X_v))$ is monic.

The remainder of this chapter will be concerned with the computation of gcds of polynomials with integer coefficients. The following notation, where

$F \in \mathbf{Z}[X_0, \ldots, X_v]$, will be used throughout this and the following chapters:

deg(F) : the degree of the main variable in F,
$\deg_i(F)$: the degree of X_i in F,
lc(F) : the leading (multivariate) coefficient in F,
icoefs(F) : the integer coefficients of the monomials in F,
coefs(F) : the (multivariate) coefficients of the leading variable in F,
icont(F) : the integer content of F, *i.e.* gcd(icoefs(F)),
cont(F) : the content of F, *i.e.* gcd(coefs(F)),
pp(F) : the primitive part of F, *i.e.* F/cont(F).

The notation $\bar{X}$ will be used as an abbreviation for $X_0, \ldots, X_v$, for example $F(X_0, \ldots, X_v)$ will be written as $F(\bar{X})$ and $\mathbf{Z}[X_0, \ldots, X_v]$ will be $\mathbf{Z}[\bar{X}]$.

3.2. Sequential Solutions to the GCD Problem

3.2.1. Classical Algorithms

The problem of finding the gcd of two elements of some domain has been solved for unique factorisation domains at least since Euclid developed his algorithm some 2300 years ago. Since the advent of computer algebra, improved methods for solving the problem have been sought.

Euclid's algorithm is entirely satisfactory as a constructive proof of the existence of a gcd or for hand calculation in simple Euclidean domains, but it is not always practical (in terms of time and space requirements) for the problems that arise in algebraic manipulation systems.

Euclid's algorithm for univariate polynomials over a field may be given as follows.

Euclid's GCD Algorithm

Input: $U(X)$, $V(X)$ non-zero polynomials over a field

Output: gcd($U(X), V(X)$)

Loop until $V(X)=0$
    $R(X) \leftarrow$ remainder $(U(X)/V(X))$
    $U(X) \leftarrow V(X)$
    $V(X) \leftarrow R(X)$
Return $U(X)$

Euclid's algorithm generates a series of remainders (called a polynomial remainder sequence or PRS) as illustrated in the following example due to Knuth (1969) and Geddes *et. al.* (1990).

$$F_1(X) = X^8+X^6-3X^4-3X^3+8X^2+2X-5$$

$$F_2(X) = 3X^6+5X^4-4X^2-9X-2$$

Working in $\mathbf{Z}_{23}[X]$ the following sequence of remainders $R(X)$, is generated by Euclid's algorithm:

$$2X^4-5X^2-8$$

$$-X^2-9X+2$$

$$10X-8$$

$$-4$$

Thus $F_1(X)$ and $F_2(X)$ are relatively prime since -4 is a unit in $\mathbf{Z}_{23}[X]$.

One problem which arises with Euclid's algorithm is illustrated by finding the sequence of remainders for the above example when working in $\mathbf{Z}[X]$. $\mathbf{Z}$ is not a field so the calculations must be done in $\mathbf{Q}[X]$.

The remainder sequence for

$$F_1(X) = X^8+X^6-3X^4-3X^3+8X^2+2X-5$$

$$F_2(X) = 3X^6+5X^4-4X^2-9X+21$$

is

$$-\frac{5}{9}X^4-\frac{1}{9}X^2-\frac{1}{3}$$

$$-\frac{117}{25}X^2-9X+\frac{411}{25}$$

$$\frac{233150}{6591}X-\frac{102500}{2197}$$

$$-\frac{1288744821}{543589225}$$

Again $F_1(X)$ and $F_2(X)$ are relatively prime as the last remainder is a unit in $\mathbf{Q}[X]$.

The PRS in $\mathbf{Q}[X]$ can be simplified by reducing $U(X)$ and $V(X)$ to monic polynomials at each step but rational arithmetic which is far more computationally intense than integer arithmetic, is still required.

The need to work with rational coefficients can be eliminated by using pseudo-remainders instead of remainders. Rather than dividing $U(X)$ by $V(X)$ in $\mathbf{Q}[X]$, $U(X)$ is multiplied by a power of the leading coefficient of $V(X)$ so that $U(X)$ can

then be divided by $V(X)$ in $\mathbf{Z}[X]$.

For $F_1(X)$ and $F_2(X)$ above the PRS in $\mathbf{Z}[X]$ (using pseudo-remainders) is

$$-15X^4+3X^2-9$$

$$15795X^2+30375X-59535$$

$$125452875143750X-1654608338437500$$

$$1259333879550074310093114199218750 0$$

Thus giving a gcd of 1.

The algorithm which generates this last PRS is given by Knuth (1969, section 4.6.1) as the *Generalised Euclidean Algorithm.*

The phenomenon seen in the last PRS is known as intermediate expression swell. The coefficients of $F_1(X)$ and $F_2(X)$ and their gcd are small integers, yet the coefficients of the intermediate polynomials increase exponentially.

Simplifying the previous PRS by taking out common factors of the coefficients leaves a *primitive* PRS. The primitive PRS in $\mathbf{Z}[X]$ for the example polynomials is

$$5X^4-X^2+3$$

$$13X^2+25X-49$$

$$4663X+6150$$

$$1$$

The problem of coefficient growth becomes very apparent when Euclid's algorithm is applied to polynomials in $\mathbf{Z}[X_0, \ldots, X_v]$. The coefficient domain must be a field so the coefficient operations are performed in $\mathbf{Z}(X_0, \ldots, X_{v-1})[X_v]$. To obtain a primitive PRS the gcd algorithm must be recursively applied to the coefficients which are rational functions.

Collins and Brown (Knuth, 1969) independently discovered a polynomial gcd algorithm which is superior to Euclid's, being more efficient and applicable to polynomials in any ring not just in Euclidean domains and which is more efficient for multivariate polynomials than the primitive PRS algorithm. Collin's algorithm avoids the primitive part calculation by instead dividing by an element of the coefficient domain

which is known to be a factor of $R(X)$; this keeps the coefficient growth linear. The resulting PRS is called a *subresultant* PRS.

For $F_1(X)$ and $F_2(X)$ from above the subresultant PRS is

$$15X^4 - 3X^2 + 9$$

$$65X^2 + 125X - 245$$

$$9326X + 12300$$

$$260708$$

This sequence is the simplest PRS of those given here for this example since coefficient growth is kept down but without the high costs of the primitive PRS which removes all common factors of the coefficients at each step.

The construction of a PRS to obtain the gcd of two polynomials is derived from the facts that two polynomials are relatively prime if and only if their *resultant* (the determinant of their Sylvester matrix) is nonzero and that if the Sylvester matrix is triangularised to row echelon form then the last non zero row gives the coefficients of the polynomial gcd. For a detailed explanation of Sylvester matrices and polynomial gcds see Geddes *et. al.*, (1990).

Brown (1971) compares the various PRS algorithms. The Euclidean PRS algorithm "suffers so severely from coefficient growth that it does not merit further consideration". Brown concludes that in most cases the subresultant PRS algorithm is superior to the primitive PRS algorithm but it is possible that there are some cases when the latter has the advantage.

Davenport *et al.*(1988) are more positive with their comment that the subresultant PRS "is the best method known for calculating the gcd, of all those based on Euclid's algorithm applied to polynomials with integer coefficients".

3.2.2. Modular Algorithms

Even when the best one is chosen the PRS algorithms are still not satisfactory for multivariate polynomials because of the recursive gcd calls. An entirely different approach to the problem was taken by Brown (1971) (and previously by Collins for the univariate case). The entire problem may be mapped to a simpler domain via homomorphisms, the solution found in the new domain and used to reconstruct the solution in the original domain. This approach resulted in the modular gcd

algorithms where the simpler domain is $\mathbf{Z}_q[X]$, q being a prime integer. Details of the modular gcd algorithms are discussed in the next chapter.

3.2.3. p-adic Algorithms

The p-adic algorithms for gcd computation (and polynomial factorisation) are based on an algebraic theorem called *Hensel's lemma* which dates back to the early 1900s. Denoting the kth order p-adic approximation to $F(X) \in \mathbf{Z}[X]$ with $F^{(k)}(X)$ and following Geddes *et al.* (1990) Hensel's lemma can be stated as follows.

Hensel's Lemma

Let $p \in \mathbf{Z}$ be prime and $A(X) \in \mathbf{Z}[X]$ be a given polynomial. Let $U^{(1)}(X), W^{(1)}(X) \in \mathbf{Z}_p[X]$ be relatively prime polynomials such that

$$A(X) \equiv U^{(1)}(X)W^{(1)}(X) \pmod{p}$$

then for any integer $k \geq 1$ there exist polynomials $U^{(k)}(X)$ and $W^{(k)}(X)$ such that

$$A(X) \equiv U^{(k)}(X)W^{(k)}(X) \pmod{p^k}$$

and

$$U^{(k)}(X) \equiv U^{(1)}(X) \pmod{p}$$

$$W^{(k)}(X) \equiv W^{(1)}(X) \pmod{p}.$$

Further, if $A(X)$ is monic and $U^{(1)}(X), W^{(1)}(X)$ are chosen to be monic then $U^{(k)}(X)$ and $W^{(k)}(X)$ are unique.

In his 1969 paper Zassenhaus proposed the use of Hensel's lemma to construct, in k iterations, a factorisation of a polynomial modulo p^k from a factorisation modulo p. Moses and Yun (1973) took the idea further and developed a method for finding polynomial gcds. The algorithm is called the Extended Zassenhaus gcd (or EZGCD) algorithm.

The EZGCD algorithm involves a p-adic reconstruction of the gcd and co-factor as two factors of one of the given polynomials. Given a univariate solution in $\mathbf{Z}_p[X]$ the p-adic methods compute (or *lift*) a solution in $\mathbf{Z}_q[X]$ where $q = p^k$ gives a sufficiently large field to contain the coefficients of the true solution. The univariate solution is then lifted from $\mathbf{Z}_q[X]$ to $\mathbf{Z}_q[\overline{X}]$ using ideal-adic methods.

To illustrate the EZGCD algorithm the following an example is due to Geddes *et al.* (1990). Let

$$F_1(X,Y) = Y^2 + 2XY - 3Y + X^2 - 3X - 4$$

and

$$F_2(X,Y) = Y^2+2XY+5Y+X^2-5X+4.$$

The algorithm assumes that the gcd and one of its cofactors will be relatively prime and applies one evaluation homomorphism (section 4.3.2) for each variable and one modular homomorphism (section 4.3.1) for the integer coefficients of F_1 and F_2.

The best evaluation point to start with is $Y=0$ which gives the ideal $I=<Y>$. Now

$$\phi_I(F_1) = X^2-3X-4 \quad \text{and} \quad \phi_I(F_2) = X^2-5X+4$$

and

$$\gcd(\phi_I(F_1),\phi_I(F_2)) = X+1$$

That this evaluation homomorphism has resulted in the correct degree of the gcd is checked by repeating the evaluation at $Y=1$. Now taking F_1 as the product of the gcd and its cofactor

$$\begin{aligned}\phi_I(F_1) &= (X+1)[\phi_I(F_1)/(X+1)] \\ &= (X+1)(X-4)\end{aligned}$$

The gcd and its cofactor are relatively prime so it is valid to use Hensel's construction to lift

$$F_1(X,Y) \equiv (X+1)(X-4) \mod I$$

to

$$F_1(X,Y) \equiv (X+1+Y)(X-4+Y)$$

After checking that $(X+1+Y)$ is also a factor of $F_2(X,Y)$ it can be concluded that

$$\gcd(F_1,F_2) = (X+1+Y)$$

There are four problems associated with the EZGCD algorithm: the leading coefficient problem, the bad zero problem, the unlucky evaluation problem and the common divisor problem, all discussed by Wang (1980). Wang has enhanced the EZGCD algorithm with improvements which mostly overcome all of these problems. Wang's version of the algorithm is known as the EEZ-GCD algorithm and is generally faster and requires less space than the original.

In his 1981 paper, Zippel presents an organisation of the p-adic/Hensel lifting algorithm that takes advantage of the expected sparseness of the polynomial being lifted. The method uses the same probabilistic techniques as the sparse modular interpolation algorithm first presented in (Zippel, 1979) and discussed in Chapter 4 (in fact, the sparse Hensel algorithm was discovered first).

Zippel claims his organisation is more general than the earlier one and makes the relationship between Newton iteration and p-adic techniques clearer. However he does not make any comment about the performance advantages of the new algorithm except to say that the exponential behaviour of the earlier version is not exhibited by his version. Since no indication is given as to whether the new version has practical as well as theoretical advantages over the old version it is not considered further here.

3.2.4. Heuristic Polynomial GCD Algorithm

Geddes *et al.* (1990) have proposed a polynomial gcd algorithm based on a heuristic from an evaluation homomorphism. They define, for $F(X)$ a polynomial over the integers and ξ an integer, the ring homomorphism

$$\phi_{X-\xi} : \mathbf{Z}[X] \rightarrow \mathbf{Z},\ \phi_{X-\xi}(F(X)) = F(\xi)$$

Then if $\phi_{X-\xi}(F_1(X))$ and $\phi_{X-\xi}(F_2(X))$ are relatively prime (for a *suitable* evaluation point, ξ) then it may be assumed that F_1 and F_2 are relatively prime. The algorithm GCDHEU aims to construct the gcd using this heuristic and is based on a few simple steps.

If $G(X) = \gcd(F_1(X), F_2(X))$ and ξ is an integer bounding twice the magnitudes of all coefficients in F_1 and F_2 and in any of their factors, let

$$\alpha = \phi_{X-\xi}(F_1(X))$$

$$\beta = \phi_{X-\xi}(F_2(X))$$

Define $\gamma = \gcd(\alpha, \beta)$ then $\gamma = \phi_{X-\xi}(G(X))$. The aim is to convert the integer γ into its ξ-adic representation:

$$\gamma = c_0 + c_1\xi + c_2\xi^2 + \cdots + c_d\xi^d$$

where d is the smallest integer such that $\xi^{d+1} > 2|\gamma|$ and $-\xi/2 < c_i \leq \xi/2$ for $i = 0, \ldots, d$. Under conditions detailed in their paper Geddes *et al.* (1990) claim that the c_i are exactly the coefficients of $G(X)$ and may be computed using the loop:

$e \leftarrow \gamma$
for i from 0 while $e \neq 0$ do
{ $c_i \leftarrow \phi_{X-\xi}(e)$
$e \leftarrow (e - c_i)/\xi$
}

The method can be generalised to multivariate gcds through recursive applications of evaluations and interpolations. However in both univariate and multivariate cases, the process is not guaranteed to work - the reconstructed gcd may be too small. If the

reconstructed gcd divides both F_1 and F_2 then it is the correct gcd otherwise a new evaluation point is chosen.

Rather than reproducing the algorithm as given by Geddes *et. al.*, a worked example is given here to illustrate how its main concept (the loop given above) works.

Consider the primitive polynomials

$$F_1(X,Y) = X^4+2X^3Y+X^2Y+3XY^2-3X+2Y^3-6Y$$

and

$$F_2(X,Y) = X^4Y-X^3+X^2Y^2+XY^3-4XY-Y^2+3.$$

First ξ, an evaluation point is chosen according to the formula

$$\xi = 2\min(|F_1|\,|F_2|)+2$$

where $|F|$ is the height function (maximum absolute value of the integer coefficients) of polynomial F. Thus, in this case $\xi = 2.\min(6,4)+2 = 10$.

A recursive call to GCDHEU is made to compute

$$\begin{aligned}\gamma &= \gcd(\phi_{Y-\xi}(F_1),\ \phi_{Y-\xi}(F_2))\\ &= \gcd(X^4+20X^3+10X^2+297X+1940,\ 10X^4-X^3+100X^2+960X-97)\end{aligned}$$

A new evaluation point is chosen $\xi' = 1922$ and the following integer gcd, γ', is found using a Euclidean method.

$$\begin{aligned}\gamma' &= \gcd(F_1(\xi',\xi),F_2(\xi',\xi))\\ &= \gcd(F_1(1922,10),F_2(1922,10))\\ &= \gcd(13788294701630,136455837214535)\\ &= 7100048765\end{aligned}$$

The loop given above is used in the following form to construct a polynomial G' from the ξ'-adic expansion of γ'.

$G' \leftarrow 0$
for i from 0 while $\gamma' \neq 0$ do
{ $g_i \leftarrow \phi_{X-\xi'}(\gamma')$
$G' \leftarrow G' + g_i X^i$
$\gamma' \leftarrow (\gamma' - g_i)/\xi'$
}

Executing the loop gives the values tabulated below.

i	g'_i	G'	γ'
0	97	97	3694094
1	10	$10X+97$	1922
2	0	$10X+97$	1
3	1	$X^3+10X+97$	0

At this point G' divides both $\phi_{Y-\xi}(F_1)$ and $\phi_{Y-\xi}(F_2)$ thus G' is their gcd, and $\gamma = X^3+10X+97$.

The main loop is used again to construct G from the ξ-adic expansion of γ. Using

$G \leftarrow 0$
for i from 0 while $\gamma \neq 0$ do
{ $g_i \leftarrow \phi_{Y-\xi}(\gamma)$
$G \leftarrow G + g_i Y^i$
$\gamma \leftarrow (\gamma - g_i)/\xi$
}

generates G as shown in the following table.

i	g_i	G	γ
0	X^3-3	X^3-3	$X+10$
1	X	$XY+X^3-3$	1
2	1	Y^2+XY+X^3-3	0

Since $\gamma=0$, all that remains is to remove the integer content from G and to check if G divides both F_1 and F_2. Thus the algorithm returns

$$\gcd(F_1, F_2) = X^3+XY+Y^2-3.$$

If G did not divide one of F_1 and F_2 then a new evaluation point, the integer part of $(\xi \times 73794)/27011$ (to ensure that the prime decomposition of successive evaluation points have no obvious patterns), would be calculated and the procedure repeated. If more than some nominal number, say six, evaluation points are required or the size of the integers grows too big (say, greater than 4000 digits) the algorithm fails.

Davenport and Padget (1985) (who were responsible for some important improvements to the original GCDHEU algorithm) compare GCDHEU with a PRS algorithm and a p-adic algorithm. They were not able to make definite conclusions as to the better of the three algorithms; the individual gcd problems seem to dictate which algorithm will give best results in terms of time requirements.

3.2.5. Gröbner Bases

Gröbner bases, first developed and formalised by Buchberger (1965) have become very important in computer algebra, at least from a theoretical point of view, and warrant a mention here. As well as solving problems related to systems of polynomial equations Gröbner bases can be used to compute univariate gcds, perform Hensel lifting or reduce multivariate polynomial factorisation to univariate factorisation (Gianni and Trager, 1985).

The method of Gröbner bases consists in two general steps: first a set **F** of multivariate polynomials is transformed (using Buchberger's completion algorithm) into its Gröbner basis, **G**, a canonical form such that **F** and **G** generate the same ideal (or, as systems of algebraic equations, have the same set of zeros); then the problem is solved for **G** rather than for **F**. Many important problems can be solved much more easily for **G** than for **F**.

In special cases Buchberger's completion algorithm for polynomial ideal bases specialises to other known algorithms. For example, in the case of two univariate polynomials it specialises to Euclid's algorithm (Buchberger and Loos, 1983).

There has not yet appeared an application of the Gröbner bases method to finding the gcd of multivariate polynomials.

3.3. Parallel Solutions to the GCD Problem

3.3.1. Current Work

There seems to be relatively little research underway on parallel algebraic algorithms for polynomials in general or the gcd problem in particular. Ponder (1988a) surveys these areas of research reporting two developments relevant to this study.

The first is a "theoretical package for parallel symbolic manipulation", including algorithms for matrix problems, solutions of non-singular systems of linear equations and the gcd of two univariate polynomials (Borodin, von zur Gathen and Hopcroft, 1982). In later papers concerning the same research (for example, von zur Gathen 1984, 1985, 1986) further algorithms are given for matrix problems, basic polynomial arithmetic operations and multivariate polynomial factorisation. This work remains largely theoretical because of the approach to parallelism adopted - the asymptotic approach whereby assuming an effectively unlimited number of processors, the parallel time is minimised. Because of this approach the algorithms are very fine grained and require a synchronous shared memory model of parallel computation.

The other relevant development mentioned by Ponder is the work of Watt (1985a) in parallelising Zippel's sparse modular gcd algorithm (see Chapter 4). Watt's parallel gcd work is again largely theoretical. Although in his thesis he describes the construction of a system for running computer algebra programs on a multiprocessor (as outlined in section 2.2, his system is a version of Maple with facilities to distribute processes over a local area network) he does not give any evidence of having implementing his parallel gcd algorithm.

Further, Watt appears to have misunderstood the application of Zippel's sparse interpolation algorithm to the gcd problem resulting in a gcd algorithm which will not only not work but is easier to parallelise. Watt has in effect parallelised Zippel's sparse interpolation algorithm, but not its application to the gcd problem (or, the sparse modular gcd algorithm) as he claims. As explained in Chapter 4, the sparse modular gcd algorithm finds an image of $\gcd(F_1(X_0, \ldots, X_n), F_2(X_0, \ldots, X_n))$ in $\mathbf{Z}_p[X_0]$ then applies the sparse interpolation algorithm to each non zero coefficient of this image, rather than simply applying the sparse interpolation algorithm to the image as Watt infers.

Watt's parallelisation and analysis of the sparse interpolation algorithm is, however, closely related to the work in the next chapter where further reference to his work will be made.

3.3.2. Decision to Parallelise a GCD Algorithm

A decision was made to parallelise Zippel's sparse modular gcd algorithm rather than any of the others discussed in this chapter. There were several reasons for this, the main one being that Zippel's algorithm is the best suited to the eval-server model of computation to be used (developed in Chapter 2).

The Euclidean based algorithms were discounted on the grounds of their having very little scope for parallelisation. Each of the algorithms is essentially the construction of a sequence of polynomials (a PRS) with each polynomial in the sequence dependent on the previous one. Thus the iterations of the main loop must be done one after another in the order given. Hence the Euclidean algorithms are not amenable to a medium- to coarse-grained model of parallel computation.

Buchberger's completion algorithm for Gröbner bases has been parallelised for a distributed memory MIMD machine (Senechaud, 1991), however the lack of applicability of the Gröbner bases method to multivariate gcds immediately discounts it as a candidate for a parallel gcd algorithm.

There is some potential parallelism present in the GCDHEU algorithm of section 3.2.4 especially when several evaluation points are required before the algorithm either arrives at a solution or fails. Because of its simplicity and speed in determining co-prime polynomials, GCDHEU was implemented and is discussed further in section 5.3 as a companion to the parallel version of Zippel's sparse modular algorithm. It is not suitable as a stand-alone gcd algorithm because of its high failure rate for polynomials of more than about four variables.

The p-adic or Hensel lifting algorithms have gained much popularity for serial computation of polynomial gcds, polynomial factorisation and other applications. As with the Euclidean algorithms the p-adic algorithms involve the inductive construction of a sequence of polynomials, each iteration of the main loop is dependent on the previous one thus preventing coarse-grained parallelisation.

Both the p-adic and modular techniques consist of a series of lifting stages; a sequence of polynomials is constructed. The p-adic method lifts a polynomial to the next stage by applying certain functions to it as a whole. The modular interpolation algorithm however considers the polynomial as a sum of terms in X_0 and lifts each term. It is this feature of the modular algorithms that make them stand out as a candidate for coarse-grained parallelisation.

Since for sparse polynomials (and most polynomials are sparse), Zippel's sparse modular gcd algorithm seems to be the most efficient of the modular gcd algorithms (Zippel, 1988) it was chosen for parallelisation and is the subject of the next chapter.

Chapter 4

Zippel's Probabilistic Sparse Modular GCD Algorithm

4.1. Notation and Preliminaries

The primary focus of this chapter is Zippel's probabilistic sparse polynomial interpolation algorithm, in particular it's application to the calculation of $\gcd(F_1, F_2)$ where F_1 and F_2 are (multivariate) polynomials, as an improvement of Brown's modular gcd algorithm. The algorithm and its subsidiary algorithms are presented and an example worked through. A parallel version of the gcd algorithm is developed followed by a discussion of the probabilistic aspects of the algorithm.

A polynomial $F(X_0, \ldots, X_v)$ consists of the sum of a number of monomials and their corresponding coefficients. The set of exponent vectors, $\{\bar{e}_0, \ldots, \bar{e}_T\}$ of $F(X_0, \ldots, X_v)$ is called the *skeleton* of $F(X_0, \ldots, X_v)$. By $\bar{X}^{\bar{e}_i}$ is meant the monomial $X_0^{e_{i0}} X_2^{e_{i2}} \cdots X_v^{e_{iv}}$.

The gcd algorithm presented in this chapter starts by constructing a univariate skeleton of the gcd and then builds each multivariate polynomial coefficient. It is therefore useful to have an alternative description of a polynomial: a polynomial can be said to consist of a number of *terms* where a term is a power of X_0 and its (non-zero) multivariate polynomial coefficient.

Throughout this chapter the practice of using X_i to refer to an indeterminate and x_i to refer to a value of X_i will be used. During the lifting process discussed later, the notation $P'(X_1, \ldots, X_{k-1}, x_{kj})$ is used for $P(X_1, \ldots, X_{k-1}, x_{kj}, x_{k+1,0}, \ldots, x_{v0})$ since the variables following X_k are not affected by the computations to lift X_k (this is also exploited for the sake of efficiency in the implementation of the algorithms).

Some further notation used is defined at the beginning of Chapter 3.

4.2. The Chinese Remainder Algorithm

The Chinese remainder algorithm (CRA) is derived from the two thousand year old Chinese remainder theorem. In this section the two forms of the CRA which are needed for the gcd algorithm are given. The first CRAI, merges n integers $u_j \bmod m_j$ to a unique integer $u \bmod m$ where m is the product of the m_j. CRAI is applied by CRAmerge (not given here) to construct each of the integer coefficients of a polynomial which has been constructed for several moduli. The second version of the algorithm, CRAII, fits a univariate polynomial to a set of points.

Included in the input data, in this presentation of the CRA algorithms, are certain values, integers q_i and c_i in CRAI and polynomials q_i in CRAII, which may be computed during the iterative step of the algorithms but are precomputed for efficiency since they are often required more than once.

Algorithm CRAI

Input:

Distinct integers $m_1, m_2, \ldots, m_n$ which are pair-wise relatively prime and ordered $m_1 > m_2 > \cdots > m_n$

Integers $u_1, \ldots, u_n$

Integers $q_1, \ldots, q_n$ such that $q_k = \prod_{j=1}^{k-1} m_j$

Integers $c_2, c_3, \ldots, c_n$ such that $c_k = q_k^{-1} \pmod{m_k}$

Output:

The unique integer $u \pmod{\prod_{j=1}^{n} m_j}$ with $u \equiv u_j \pmod{m_j}$ for $j = 1,\ldots,n$ and $0 \le u < \prod_{i}^{n} m_i$

Step CI.1 (Initialise) $u \leftarrow u_1 \bmod m_1$

Step CI.2 (Loop) For $k = 2, 3, \ldots, n$ do

Step CI.3 $v \leftarrow u \bmod m_k$

$a \leftarrow (u_k - v)\, c_k \bmod m_k$

$u \leftarrow u + a q_k$

Step CI.4 Return u

Algorithm CRAII

Input:

Distinct integers $x_1, \ldots, x_n \in \mathbf{Z}_q$ where q is a prime

Integers $y_1, \ldots, y_n \in \mathbf{Z}_q$

Polynomials $q_1, \ldots, q_n \in \mathbf{Z}_q[x]$ such that $q_k = \prod_{j=1}^{k-1}(X - x_j)$

Output:

A polynomial $f(X)$ such that $f(x_i) = y_i$ for $i = 1, \ldots, n$ and $\deg(f(X)) \leq n$

Step CII.1 (Initialise) $f(X) \leftarrow y_1$

Step CII.2 (Loop) For $i = 2, 3, \ldots, n$ do

Step CII.3 $f(X) \leftarrow f(X) + q_i(x_i)^{-1} q_i(X)(y_i - f(x_i))$

Step CII.4 Return $f(X)$

4.3. Modular GCD Algorithms

Modular gcd algorithms map, via homomorphisms, the given polynomials into one or more simpler domains with more algebraic structure in which the gcd of the images can be more easily computed (for example, $\mathbf{Z}_q[\bar{X}]$ has more algebraic structure than $\mathbf{Z}[\bar{X}]$ and the problems of coefficient growth do not arise). The gcd in the original domain is then constructed from its image(s).

Two homomorphism techniques and one basic gcd technique are used by modular gcd algorithms:

- the gcd problem in $\mathbf{Z}[X_0, \ldots, X_v]$ is reduced to a series of gcd problems in $\mathbf{Z}_q[X_0, \ldots, X_v]$ by applying modular homomorphisms,
- the gcd problems in $\mathbf{Z}_q[X_0, \ldots, X_v]$ are reduced to gcd problems in $\mathbf{Z}_q[X_0]$ by applying evaluation homomorphisms,
- the gcd problems in $\mathbf{Z}_q[X_0]$ are solved using Euclid's algorithm.

4.3.1. Modular Homomorphisms

The modular homomorphism $\phi_q : \mathbf{Z}[X_0, \ldots, X_v] \rightarrow \mathbf{Z}_q[X_0, \ldots, X_v]$ gives the advantage of a more structured domain to work in and is suitable to use for gcd calculations since if $C(\bar{X}) = \gcd(F_1(\bar{X}), F_2(\bar{X}))$ then

$$\phi_q(\gcd(F_1(\bar{X}), F_2(\bar{X}))) = c(X_1, \ldots, X_v)\gcd(\phi_q(F_1(\bar{X})), \phi_q(F_2(\bar{X})))$$

for some c unless q is an *unlucky prime* (see below). The problem may be *normalised* by forcing each modular gcd to have a leading coefficient of

$\gcd(\mathrm{lc}(F_1(\bar{X})), \mathrm{lc}(F_2(\bar{X})))$

through multiplying by this value. The interpolation then produces $D(\bar{X}) = c\, C(\bar{X})$ rather than $C(\bar{X})$ which can then be obtained by taking the primitive part of $D(\bar{X})$.

Modular homomorphisms bear the cost of lost information. It can be shown, (Geddes *et al.*, 1990), that

$$\deg(\gcd(\phi_q(F_1(\bar{X})), \phi_q(F_2(\bar{X})))) \geq \deg(\gcd(F_1(\bar{X}), F_2(\bar{X})))$$

(that is, the degree of the homomorphic image is never too small). In particular, if $\phi_q(F_1(\bar{X}))$ and $\phi_q(F_2(\bar{X}))$ are coprime then so are $F_1(\bar{X})$ and $F_2(\bar{X})$. However where a prime, q, results in

$$\deg(\gcd(\phi_q(F_1(\bar{X})), \phi_q(F_2(\bar{X})))) > \deg(\gcd(F_1(\bar{X}), F_2(\bar{X})))$$

then that prime is said to be *unlucky*. Unlucky primes are easily detected by comparing the degrees of successive $\gcd(\phi_q(F_1(\bar{X})), \phi_q(F_2(\bar{X})))$ calculations.

Fortunately, for any given polynomials $F_1(X_0, \ldots, X_v)$ and $F_2(X_0, \ldots, X_v)$ the number of unlucky primes is small. Geddes *et al.* (1990) show that the probability of q being is unlucky is less than $v/2^b$ when q is chosen close to b, the word size of the computer (usually 32). In practice, the author has not yet struck an unlucky prime for any of the many example polynomials with which the modular gcd algorithm of section 4.7.1 has been tested.

The number of modular images required can be determined by finding an upper bound B, on the size of the coefficients of the true gcd. Modular gcds are found for a number of primes q_i and the Chinese remainder algorithm (CRAI) applied to each set of coefficients to find the gcd modulo the product Q, of the q_i. If the q_i are chosen such that $Q > 2B$ then the coefficients of the gcd will have the same numeric value modulo Q as they have in $\mathbf{Z}$.

Following Brown (1971), a simple bound B, on the integer coefficients of $\gcd(F_1, F_2)$ may be computed as the maximum absolute value of the integer coefficients from F_1 multiplied by the maximum absolute value of the integer coefficients from F_2.

If $2B$ is less than *maxint*, the maximum single-word integer for the computer, then only one modular image will be required and q_1 is chosen such that $2B < q_1 < maxint$. It follows that Brown's bound, B, will not affect the efficiency of the algorithm by being too large unless F_1 or F_2 have large enough coefficients to make $q_1 > maxint$. For a computer with wordsize 32 a coefficient of F_1 or F_2 would

have to be at least $\frac{1}{2}\sqrt{2^{32}-1} = 32768$ before $2B > maxint$. For many practical purposes, Brown's assumption that the maximum of the absolute values of the coefficients of the polynomials is small compared to the largest single-word integer justifies his choice of bound.

4.3.2. Evaluation Homomorphisms

An evaluation homomorphism

$$\eta_{\bar{x}}: R[X_1, \ldots, X_v] \rightarrow R[X_1, \ldots, X_k, x_{k+1}, \ldots, x_v]$$

where R is a ring (for the purposes of this research, $\mathbf{Z}_q$) and $\bar{x} = (x_{k+1}, \ldots, x_v)$ is a point of evaluation, reduces a polynomial in v variables to one in k variables.

As with the modular homomorphisms, evaluation homomorphisms produce a loss of information. The vector $\bar{x}$ is said to be an *unlucky* point of evaluation when the gcd in the homomorphic image has degree larger than that of the true gcd (the homomorphic image will never have degree smaller than that of the true gcd). Geddes *et al.* (1990) argue that if q is very large then the probability of a randomly chosen point of evaluation being unlucky is not greater than $\frac{2me(v-1)}{q}$ where $m = \frac{1}{2} \max_{1 \le i \le v-1} (\deg_i(F_1), \deg_i(F_2))$, $e = \max(\deg_v(F_1), \deg_v(F_2))$ and v is the number of variables. Numerous tests with the algorithm by the author have not yet struck an unlucky evaluation point.

4.4. Zippel's Sparse Modular Algorithm

Zippel's algorithm (Zippel, 1979, 1988 and 1990) works in the same way as other modular algorithms, but by making assumptions (explained below) about the sparseness of the goal polynomial less computations are required.

Given that the goal polynomial (the one being computed) is $P(X_1, X_2, \ldots, X_v)$, Zippel's algorithm produces the sequence

$$P_1 = P(X_1, x_{20}, \ldots, x_{v0})$$

$$P_2 = P(X_1, X_2, x_{30}, \ldots, x_{v0})$$

$$P_v = P(X_1, X_2, \ldots, X_v)$$

where $(x_{10}, x_{20}, \ldots, x_{v0})$ is a *good* starting point (see section 4.8). At each stage of the lifting process (*i.e.* computing P_k from P_{k-1}) assumptions are made pertaining to

the sparsity of P_k based on the sparsity of P_{k-1}. The key idea is to assume that if some monomial $X_1^{e_1}X_2^{e_2}\cdots X_{k-1}^{e_{k-1}}$ does not appear in P_{k-1} (that is, it has zero coefficient) then it will not appear in P_k.

Zippel's algorithm for polynomial interpolation can be applied to any problem where a polynomial must be constructed from its values. For finding gcds the procedure begins as for Brown's dense modular algorithm (Brown, 1971). Given operands $F_1(X_0, \ldots, X_v)$ and $F_2(X_0, \ldots, X_v)$ and a starting point $\bar{x}=(x_{10}, \ldots, x_{v0})$, the first step is to make F_1 and F_2 primitive (by using recursive applications of the gcd algorithm to find their contents). Then for a series of primes q, the gcd is constructed using the following broadly outlined algorithm.

Algorithm ZIPPEL-GCD: for Polynomial GCDs

Input: primitive, monic polynomials $F_1(X_0, \ldots, X_v)$ and $F_2(X_0, \ldots, X_v)$
starting point $\bar{x}=(x_{10}, \ldots, x_{v0})$

Output: $\gcd(F_1, F_2)$

Step ZG1 The (sparse) univariate polynomial

$$\gcd(F_1(X_0, x_{10}, \ldots, x_{v0}), F_2(X_0, x_{10}, \ldots, x_{v0})$$

is calculated (and normalised) giving

$$c_0(x_{10}, \ldots, x_{v0})X_0^{e_0} + \cdots + c_n(x_{10}, \ldots, x_{v0})X_0^{e_n}$$

where the c_i are non-zero.

Step ZG2 For each coefficient c_i (lift each c_i using Zippel's sparse interpolation algorithm)

Step ZG3 Use dense interpolation to lift X_1 giving a univariate polynomial in X_1: *i.e.* $c_i(x_{10}, \ldots, x_{v0}) \rightarrow c_i(X_1, x_{20}, \ldots, x_{v0})$

Step ZG4 (Sparse Interpolation) For each variable X_i above X_1

Step ZG5 Use sparse interpolation to generate several polynomials $c_{i_j}(X_1, \ldots, X_{k-1}, x_{kj}, x_{k+1,0}, \ldots, x_{v0})$ for randomly chosen x_{kj}.

Step ZG6 Use dense interpolation (CRAII) to construct each coefficient of the polynomial
$c_i(X_1, \ldots, X_k, x_{k+1,0}, \ldots, x_{v0})$
from the polynomials generated in Step ZG5

Step ZG7 Output $\mathrm{pp}(c_0\bar{X}^{\bar{e}_0} + \cdots + c_n\bar{X}^{\bar{e}_n})$

4.5. A Worked Example

Before presenting Zippel's algorithm formally it is constructive to work through an example. The following example was first given by Zippel (1988). This presentation is based on the revised algorithm of his later paper (1990) which includes the use of a Vandermonde system of equations at Step ZG5 rather than a general system of equations which is less efficient to solve.

Let

$$\begin{aligned} F_1(X_0,X_1,X_2) = X_0^5 &+ (X_1^2 + X_1X_2 + 1)X_0^4 + (2X_1^3 - 7X_1X_2^2 - 2)X_0^3 \\ &+ (2X_1^5 + 2X_1^4X_2 - 7X_1^3X_2^2 + 2X_1^3 - 7X_1^2X_2^3 - 7X_1X_2^2 + 3)X_0^2 \\ &+ (-4X_1^3 + 3X_1^2 + 14X_1X_2^2 + 3X_1X_2 + 3)X_0 - 6 \end{aligned}$$

and

$$\begin{aligned} F_2(X_0,X_1,X_2) = X_0^6 &+ (2X_1^3 - 7X_1X_2^2 + X_1 - 6X_2^3)X_0^4 + 6X_0^3 \\ &+ (2X_1^4 - 12X_1^3X_2^3 - 7X_1^2X_2^2 + 42X_2^5X_1)X_0^2 \\ &+ (6X_1^3 - 21X_1X_2^2 + 3X_1 - 18X_2^3)X_0 + 9. \end{aligned}$$

Both polynomials are primitive and monic.

A simple bound on the coefficients of the gcd is

$$\begin{aligned} B &= \max(\text{coefs}(F_1))\max(\text{coefs}(F_2)) \\ &= 14 \times 42 \end{aligned}$$

thus the gcd will have coefficients in the range $[-588, 588]$.

For this example several small primes, q, are chosen. The q are chosen such that $Q > 2B$ where $Q = \prod q$. The Chinese remainder theorem (Algorithm CRAI) is used to merge the mod q gcds to produce the mod Q gcd. In practice, a single large prime $2B < q < maxint$ would be used for greater efficiency.

Starting with $q = 19$ and a randomly chosen starting point, say $(x_{10}, x_{20}) = (12, 15)$, the univariate gcd at this point is

$$\gcd(F_1(X_0, 12, 15), F_2(X_0, 12, 15)) = X_0^3 + 3X_0 + 3 \mod q.$$

This is Step ZG1. Note that since $\gcd(\text{lc}(F_1), \text{lc}(F_2)) = 1$, no normalisation of the gcd is required.

Assuming $q = 19$ is not an unlucky prime and the starting point is a good one there are now three goal polynomials to compute, $c_1(X_1, X_2)$ the coefficient of X_0^3,

$c_2(X_1,X_2)$ the coefficient of the linear term and $c_3(X_1,X_2)$ the coefficient of X_0^0. X_0^2 is assumed to have zero coefficient.

As discussed later in section 4.7.1, each of the goal polynomials are interpolated using Zippel's algorithm and may be done concurrently. Since (in the final answer) $c_1(X_1,X_2) = 1$ and $c_3(X_1,X_2) = 3$, the linear term is the only interesting one and therefore the lifting process for $c_2(X_1,X_2)$ only is worked through here.

At the first stage of the sparse interpolation procedure (Step ZG3) dense interpolation is used to *lift* the variable X_1 keeping X_2 set at $x_{20}=15$. Random numbers x_{1j} are found and polynomials $c_2(X_0,x_{1j},x_{20})$ computed for $j = 0, \ldots, d_1$ where d_1 is the maximum degree of X_1 in the goal polynomial (in this case $d_1=3$ since maximum degree of X_1 in F_1 and F_2 is 5 and 3 respectively). Using $x_{1j}=0,11,13,5$, the values of $c_2(X_0,x_{1j},x_{20})$ may taken from

$$\gcd(F_1(X_0,\ 0,15),\ F_2(X_0,\ 0,15)) = X_0^3 + 0X_0 + 3$$

$$\gcd(F_1(X_0,11,15),\ F_2(X_0,11,15)) = X_0^3 + 5X_0 + 3$$

$$\gcd(F_1(X_0,13,15),\ F_2(X_0,13,15)) = X_0^3 - 7X_0 + 3$$

$$\gcd(F_1(X_0,\ 5,15),\ F_2(X_0,\ 5,15)) = X_0^3 - 6X_0 + 3$$

Interpolating the coefficients of the linear terms (*viz* 0, 5, –7, –6) against the x_{1j} using the Chinese remainder algorithm (CRAII) results in $c_2(X_1,15) = 2X_1^3 + 2X_1$.

The next stage of the process (Step ZG4) introduces X_2. The goal polynomial is the coefficient c_2, of X_0^1 in the gcd. From the computations so far

$$c_2(X_1,15) = 2X_1^3 + 2X_1,$$

so the goal polynomial is

$$P(X_1,X_2) = p_1(X_2)X_1^3 + p_2(X_2)X_1.$$

Note that since X_1^2 and X_1^0 did not appear in $c_2(X_1,15)$, they are assumed to have zero coefficient in the goal polynomial.

This stage is the $k=2$-th stage of Zippel's algorithm. A value for X_2, x_{20} is chosen along with a random $k-1$ tuple, $\bar{y}$, such that the $\bar{y}^{\bar{e}_i}$ are distinct ($\bar{e}_1=(3)$, $\bar{e}_2=(1)$ are the exponent vectors of $P(X_1,15)$). Suppose $x_{20}=0$ and $\bar{y}=(2)$, then $\bar{y}^{\bar{e}_1}=2^3=8$ and $\bar{y}^{\bar{e}_2}=2^1=2$.

A transposed Vandermonde system is now set up :

$$P(1,x_{2j}) = \text{coef of } X_0^1 \text{ in } \gcd(F_1(X_0,1,x_{2j}),\, F_2(X_0,1,x_{2j}))$$
$$= p_{1j} + p_{2j}$$
$$P(y_1,x_{2j}) = \text{coef of } X_0^1 \text{ in } \gcd(F_1(X_0,y_1,x_{2j}),\, F_2(X_0,y_1,x_{2j}))$$
$$= p_{1j}\bar{y}^{\bar{e}_1} + p_{2j}\bar{y}^{\bar{e}_2}$$

and the system solved for p_{1j} and p_{2j} for $j=0$.

Now

$$\gcd(F_1(X_0,1,x_{20}),F_2(X_0,1,x_{20})) = X_0^3 + 2X_0 + 3 \quad \text{(modulo 19)}$$

and

$$\gcd(F_1(X_0,2,x_{20}),F_2(X_0,2,x_{20})) = X_0^3 - 3X_0 + 3 \quad \text{(modulo 19)}$$

giving the system

$$p_{10} + p_{20} = 2$$
$$8p_{10} + 2p_{20} = -3$$

which is solved: $p_{10}=2$ and $p_{20}=0$ (modulo 19).

Repeating the process of setting up and solving a system of equations d_2+1 times (where $d_2=4$ is the maximum possible degree of X_2 in the gcd) gives the following results:

j	x_{2j}	$\bar{y}$	$\gcd(F_1(X_0,1,x_{2j}),F_2(X_0,1,x_{2j}))$	$\gcd(F_1(X_0,y_1,x_{2j}),F_2(X_0,y_1,x_{2j}))$	p_{1j}	p_{2j}
0	0	(2)	$X_0^3+2X_0+3$	$X_0^3-3X_0+3$	2	0
1	2	(9)	$X_0^3-7X_0+3$	$X_0^3+9X_0+3$	2	-9
2	5	(6)	$X_0^3-2X_0+3$	$X_0^3+9X_0+3$	2	-4
3	-4	(-3)	$X_0^3+4X_0+3$	$X_0^3-3X_0+3$	2	2
4	6	(2)	$X_0^3-3X_0+3$	$X_0^3-6X_0+3$	2	-5

Using Algorithm CRAII to interpolate the x_{2j} values against the p_{1j} and against the p_{2j} gives $p_1(X_2)=2$ and $p_2(X_2)=-7X_2^2$. Thus the linear term in the gcd is $\text{pp}(c_2(X_1,X_2)X_0)=(2X_1^3-7X_1X_2^2)X_0$ (mod 19).

Repeating the entire process with different primes and using CRAI to merge the image polynomials yields that $c_2(X_0,X_1) = 2X_1^3 - 7X_1X_2^2$ is the correct value in **Z**. Thus $\gcd(F_1,F_2)=X_0^3+2X_0X_1^3-7X_0X_1X_2^2+3$.

4.6. Formal Presentation of Zippel's Sparse Interpolation Algorithm

4.6.1. Stage k

As Zippel's algorithm is a repetition of lifting stages, a more formal presentation of stage k where the variable X_k is lifted, is given here. The algorithm is divided into two phases corresponding, respectively, to Steps ZG5 and ZG6 of Algorithm ZIPPEL-GCD.

Algorithm ZIPPEL-INTERPOLATION

Input: $(\bar{e}_1, \ldots, \bar{e}_T)$, skeleton of $P'(X_1, \ldots, X_{k-1}) = p_1\bar{X}^{\bar{e}_1} + \cdots + p_T\bar{X}^{\bar{e}_T}$
d_k, maximum degree of X_k in $P(X_1, \ldots, X_v)$

Output: $P'(X_1, \ldots, X_{k-1}, X_k) = p_1(X_k)\bar{X}^{\bar{e}_1} + \cdots + p_T(X_k)\bar{X}^{\bar{e}_T}$

Phase ZI1 [Compute $P'(X_1, \ldots, X_{k-1}, x_{kj}) = p_{1j}\bar{X}^{\bar{e}_1} + p_{2j}\bar{X}^{\bar{e}_2} + \cdots + p_{Tj}\bar{X}^{\bar{e}_T}$ for $j = 0, \ldots, d_k$]

For $j = 0, \ldots, d_k$ do

Step ZI1.1 choose x_{kj}

Step ZI1.2 choose at random vector $\bar{y} = (y_1, \ldots, y_{k-1})$ such that each of the $\bar{y}^{\bar{e}_i}$ are distinct

Step ZI1.3

Set up the transposed Vandermonde system of linear equations

$$P'(1, \ldots, 1, x_{kj}) = p_{1j} + p_{2j} + \cdots + p_{Tj}$$

$$P'(y_1, \ldots, y_{k-1}, x_{kj}) = p_{1j}\bar{y}^{\bar{e}_1} + p_{2j}\bar{y}^{\bar{e}_2} + \cdots + p_{Tj}\bar{y}^{\bar{e}_T}$$

$$P'(y_1^2, \ldots, y_{k-1}^2, x_{kj}) = p_{1j}\bar{y}^{2\bar{e}_1} + p_{2j}\bar{y}^{2\bar{e}_2} + \cdots + p_{Tj}\bar{y}^{2\bar{e}_T}$$

$$P'(y_1^T, \ldots, y_{k-1}^T, x_{kj}) = p_{1j}\bar{y}^{T\bar{e}_1} + p_{2j}\bar{y}^{T\bar{e}_2} + \cdots + p_{Tj}\bar{y}^{T\bar{e}_T}$$

and solve for $p_{1j}, p_{2j}, \ldots, p_{Tj}$

Phase ZI2 [Independently interpolate the coefficients of each monomial]

For $l = 1, \ldots, T$

Use the Chinese remainder algorithm (CRAII) to interpolate $p_{l1}, p_{l2}, \ldots, p_{ld}$ with $x_{k1}, x_{k2}, \ldots, x_{kd}$ to determine $p_l(X_k)$.

Return

$$P'(X_1, \ldots, X_{k-1}, X_k) = p_1(X_k)\bar{X}^{\bar{e}_1} + \cdots + p_T(X_k)\bar{X}^{\bar{e}_T}$$

i.e.

$$P(X_1, \ldots, X_k, x_{k+1,0}, \ldots, x_{v0}) = p'_1\bar{X}^{\bar{e}'_1} + \cdots + p'_T\bar{X}^{\bar{e}_T'}$$

Step ZI1.3 is performed by using Algorithm VDM-SOLVE (see section 4.6.2) with

$$(v_1, v_2, \ldots, v_T) = (\bar{y}^0, \bar{y}^{\bar{e}_1}, \ldots, \bar{y}^{\bar{e}_{T-1}})$$

and

$$(a_1, a_2, \ldots, a_T) = (P'(1, \ldots, 1, x_{kj}), P'(y_1^2, \ldots, y_{k-i}^2, x_{kj}), \ldots, P'(y_1^T, \ldots, y_{k-i}^T, x_{kj}))$$

as input. The $P'(y_1^i, \ldots, y_{k-1}^i, x_{kj})$ are computed according to the function being lifting. For example if the coefficient of X_0^n in $\gcd(F_1, F_2)$ is being lifted then $P'(y_1^i, \ldots, y_{k-1}^i, x_{kj})$ is the coefficient of X_0^n in

$$\gcd(F_1(X_0, y_1^i, \ldots, y_{k-1}^i, x_{kj}), F_1(X_0, y_1^i, \ldots, y_{k-1}^i, x_{kj}))$$

after normalisation. The algorithm VDM-SOLVE passes the solution vector $(p_{1j}, p_{2j}, \ldots, p_{Tj})$ back to Step ZI1.3.

4.6.2. Solving the Transposed Vandermonde System of Equations

Given a system $A\bar{x} = \bar{a}$ where

$$A = \begin{bmatrix} 1 & 1 & & 1 \\ v_1 & v_2 & & v_n \\ & & & \\ \cdot & & & \\ \cdot & \cdot & & \cdot \\ v_1^{n-1} & v_2^{n-1} & & v_n^{n-1} \end{bmatrix} \quad \bar{x} = \begin{bmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix} \quad \bar{a} = \begin{bmatrix} a_1 \\ a_2 \\ \cdot \\ \cdot \\ \cdot \\ a_n \end{bmatrix}$$

the solution $\bar{x} = (B^{-1})^T\bar{a}$ where $B = A^T$ may be found using the following algorithm due to Kaltofen and Yagati (1988).

Algorithm VDM-SOLVE

Input: Integers $v_1, \ldots, v_n$ from the transposed Vandermonde matrix A
Integer vector $\bar{a} = (a_1, \ldots, a_n)^T$

Output: $\bar{x} = (x_1, \ldots, x_n)^T$ where $\bar{x} = A^{-1}\bar{a}$

Step VS1 Compute $B(z) \leftarrow \prod_{1 \le j \le n} (z - v_j)$

Step VS2 Let $D(z) \leftarrow a_1 z^n + a_2 z^{n-1} + \cdots + a_{n-1} z^2 + a_n z$
and compute $B(z)D(z) = q_{2n} z^{2n} + q_{2n-1} z^{2n-1} + \cdots + q_0$
Set $Q_n(w) = q_{2n} w^{n-1} + q_{2n-1} w^{n-2} + \cdots + q_{n+2} w + q_{n+1}$

Step VS3 For $i = 1, \ldots, n$ set $(\alpha x)_i \leftarrow Q_n(v_i)$

Step VS4 For $i = 1, \ldots, n$ set $\alpha_i \leftarrow B'(v_i)$

Step VS5 Return $((\alpha x)_i / \alpha_i)_{i = 1, \ldots, n}$

Note that when $n = 1$ or $n = 2$ (as is frequently the case in practise) it is more efficient to directly invert the Vandermonde matrix. The implementation of VDM-SOLVE takes this into account.

4.7. Parallelisation of GCD Calculation

4.7.1. Parallelising ZIPPEL-GCD

In order to parallelise ZIPPEL-GCD for Tlisp, the coarse-grained, message-passing parallel lisp system described in Chapter 2, the algorithm needs to be broken down into those sections which contain parallelism and those which do not. In ZIPPEL-GCD there are sufficient instances of large sections of computation which may be executed concurrently to justify the parallelisation of the algorithm.

The main loop of ZIPPEL-GCD, Step ZG2, applies Zippel's sparse interpolation algorithm to each coefficient of X_0 in the skeletal polynomial calculated at Step ZG1. In a serial implementation these applications of the interpolation algorithm would not be computed independently of each other. For efficiency the coefficients should be computed simultaneously as explained in the following paragraph.

Lifting $c_i(X_1, \ldots, X_{k-1}, x_{k0}, \ldots, x_{v0})$ to $c_i(X_1, \ldots, X_k, x_{k+1,0}, \ldots, x_{v0})$ requires $c_i'(X_1, \ldots, X_{k-1}, x_{kj})$ to be calculated for $j = 1, \ldots, d_k$. Each of these involves evaluating $c_i'(y_1^l, \ldots, y_{k-1}^l, x_{kj})$ for $l = 0, \ldots, T-1$ (where T is an upper bound on the number of terms in c_i). This value for c_i' is taken from

$$\gcd(F_1'(X_0, y_1^l, \ldots, y_{k-1}^l, x_{kj}), (F_2'(X_0, y_1^l, \ldots, y_{k-1}^l, x_{kj}))$$

$$= c_1'(y_1^l, \ldots, y_{k-1}^l, x_{kj}) X_0^{e_1} + \cdots + c_n'(y_1^l, \ldots, y_{k-1}^l, x_{kj}) X_0^{e_n}.$$

Thus, provided the vector $(y_1, \ldots, y_{k-1})$ is suitable for each c_i, all the $c_i'(y_1^l, \ldots, y_{k-1}^l, x_{kj})$ are computed simultaneously. In other words, if the coefficients $c_i(X_1, \ldots, X_v)$ for $i = 1, \ldots, n$ were lifted independently of each other in a sequential implementation, then much computation would be replicated, with the number of univariate gcds required being greatly increased.

However, for a coarse-grained parallel implementation of ZIPPEL-GCD each of the n coefficients, $c_i(X_1, \ldots, X_v)$, can be lifted independently, thus allowing the creation of n concurrent tasks. The recomputation of the univariate gcds is more efficient than synchronising the tasks to share the results of a single univariate gcd computation since univariate gcds are computed relatively quickly.

The two steps of the main loop, Steps ZG3 and ZG4, are independent of each other in that one iteration of ZG3 is dependent only on the previous iteration of ZG3 and not on ZG4. Thus the main loop can be split into two separate loops, one performing ZG3 on each of the coefficients c_i, followed by the second which performs ZG4 on each of the coefficients.

With Steps ZG3 and ZG4 thus separated, ZG4 is parallelised by creating n concurrent tasks, one for each term in X_0, as described above. For ZG3, where this approach would lead to very small task sizes, a different approach is needed.

Consider that in Step ZG3, for each coefficient c_i, a univariate polynomial is computed using dense interpolation. First integer coefficients $c_i(x_{1j}, x_{20}, \ldots, x_{v0})$ for $j = 1, \ldots, d_1$ (where d_1 is the maximum degree of X_1 in c_i), are computed from $\gcd(F_1(X_0, x_{1j}, x_{20}, \ldots, x_{v0}), F_2(X_0, x_{1j}, x_{20}, \ldots, x_{v0}))$. Then X_1 is lifted using the Chinese Remainder Algorithm (CRAII). The d_1 univariate gcd computations are independent of each other and may be executed concurrently; the coefficients $c_i'(X_1)$ are not computed separately.

Further parallelism is to be found within each of the n tasks created to perform Step ZG4 on one of the c_i. Step ZG4 is the lifting step of Zippel's interpolation algorithm. It consists of a sparse interpolation (Step ZG5) followed by a dense interpolation (Step ZG6). As mentioned in section 3.3.1, it was essentially this part of the algorithm that Watt (1985a) considered as the entire gcd algorithm in his attempt to parallelise it. Watt effectively parallelised Zippel's sparse interpolation algorithm by introducing parallelism into his equivalent of Steps ZG5 and ZG6. The parallelism which Watt introduced into the algorithm is much the same as that used here for those steps.

Step ZG5 (or Phase ZI1 of ZIPPEL-INTERPOLATION) generates several systems of

linear equations which are solved using VDM-SOLVE. The systems are independent of each other and may therefore each be set up and solved as one complete task.

Step ZG6 (or Phase ZI2 of ZIPPEL-INTERPOLATION) consists of T applications of the Chinese remainder algorithm. These are independent but, as discussed later in section 5.2.1, for the Tlisp implementation each interpolation does not require sufficient computation time to warrant creating a task to solve it.

Below is a parallel version of Algorithm ZIPPEL-GCD for Tlisp, based on the preceding discussion.

Algorithm PARALLEL-GCD

Input: Polynomials $F_1(X_0, X_1, \ldots, X_v)$ and $F_2(X_0, X_1, \ldots, X_v)$
Starting point $\bar{x} = (x_{10}, \ldots, x_{v0})$

Output: $\gcd(F_1, F_2)$

Step PG1 (Initialise)

$Q \leftarrow 1, \quad flag \leftarrow true$

$cont \leftarrow PARALLEL\text{-}GCD\ (\text{cont}(coefs\,(F_1)), \text{cont}(coefs\,(F_2)))$

(contents may be calculated concurrently)

$F_1 \leftarrow \text{pp}(F_1)$ #these steps may be

$F_2 \leftarrow \text{pp}(F_2)$ #done concurrently

$norm \leftarrow PARALLEL\text{-}GCD\ (\text{lc}(F_1), \text{lc}(F_2))$

$B \leftarrow 2 \max(\text{icoefs}(F_1)).\max(\text{icoefs}(F_2))$

Step PG2 (Choose prime and compute univariate skeleton)

Choose q a large prime integer such that q does not divide *norm*

#Compute univariate skeleton

#gcdmodq computes univariate gcd in $\mathbf{Z}_q$

$G_0 \leftarrow \text{gcdmodq}(F_1'(X_0), (F_2'(X_0))*norm$

$d_0 \leftarrow \deg(G_0)$

For $k = 1, \ldots, v$ do in parallel

#find max degrees for each variable

$d_k \leftarrow \deg_k(\text{gcdmodq}(F_1(x_{10}, \ldots, x_{k-1,0}, X_k, x_{k+1,0}, \ldots, x_{v0}),$
$F_2(x_{10}, \ldots, x_{k-1,0}, X_k, x_{k+1,0}, \ldots, x_v 0)) * norm$

#Choose random numbers for dense interpolations

#and pre-compute corresponding q values for CRAII

$d \leftarrow \max(d_0, \ldots, d_v)$

For $j = 0, \ldots, d$ do

$r_j \leftarrow$ random_nr()

For $k = 0, \ldots, d$ do

$$q_k = \prod_{j=1}^{k-1} (X - r_j)$$

Step PG3 (Lift X_1 - dense interpolation)

For $j = 1, \ldots, d_1$ do in parallel

$$G_j(X_0) \leftarrow \text{gcdmodq}(F'_1(X_0, r_j), F'_2(X_0, r_j)) * norm$$
$$= c_{0j} X_0^{e_0} + c_{1j} X_0^{e_1} + \cdots + c_{nj} X_0^{e_n}$$

#Check for bad homomorphisms

If $\deg(G_j) > d_0$ for any $j = 1, \ldots, d_1$ #unlucky prime/evaluation point

Then goto Step PG2 and set $G_0 \leftarrow G_j$

For $i = 0$ to n do in parallel

Interpolate $(r_0, \ldots, r_d)$ with $c_{i0}, \ldots, c_{id_i}$ to get, $c_i(X_1)$, using Algorithm CRAII

#At this point $G'(X_0, X_1) = c_0(X_1) X_0^{e_0} + \cdots + c_n(X_1) X_0^{e_n}$

Step PG4 (Lift variables above X_1 – sparse interpolation)

#Iterate on coefficients of G

For each term $c_i(X_1) = p_1 X_1^{e_1} + \cdots + p_T X_1^{e_T}$ in $G'(X_0, X_1)$ do in parallel

#Lift $c_i(X_1)$ to $c_i(X_1, \ldots, X_v)$ using Algorithm ZIPPEL-GCD

For $k = 2, \ldots, v$ do #Iterate on variables

#Set up and solve $d_k + 1$ systems of equations

For $j = 0, \ldots, d_k$ do in parallel

$x_{kj} \leftarrow r_j$

choose $\bar{y} = (y_1, \ldots, y_{k-1})$ such that the $\bar{y}^{\bar{e}_i}$ are unique

For $i = 0, \ldots, T$ compute

$$p_{1j} \bar{y}^{i\bar{e}_1} + \cdots + p_{Tj} \bar{y}^{i\bar{e}_T} = P'(y_1^i, \ldots, y_{k-1}^i, x_{kj})$$

Solve for $p_{1j}, \ldots, p_{Tj}$ using Algorithm VDM-SOLVE

For $l = 1, \ldots, T$ do in parallel

Interpolate $p_{l1}, \ldots, p_{ld_k}$ against $x_{k1}, \ldots, x_{kd_k}$ to get $p_l(X_k)$ using Algorithm CRAII

#$G(X_0, \ldots, X_v) = c_0(X_1, \ldots, X_v) X_0^{e_0} + \cdots + c_n(X_1, \ldots, X_v) X_0^{e_n}$
#has now been calculated

$G(X_0, \ldots, X_v) \leftarrow \text{pp}(G(X_0, \ldots, X_v))$ #to compensate normalisation

Step PG5 (Check results)

If $flag$ then $GCD \leftarrow G(X_0, \ldots, X_v)$; $flag \leftarrow false$

else $GCD \leftarrow CRAmerge(GCD, G, Q, q)$ # variant of CRAI

$Q \leftarrow Q*q$

If $Q > B$

If ($GCD \mid F_1$ and $GCD \mid F_2$) then return $cont*GCD$

else $Q \leftarrow 1$; $flag \leftarrow true$

Go to Step PG2

4.7.2. Parallelising VDM-SOLVE

The algorithm VDM-SOLVE presents very little scope for medium- to coarse-grained parallelism. Steps VS1 and VS2 are strictly sequential (in a coarse-grained sense). Each of Steps VS3 through VS5 consist of n small but independent operations. Each step could be individually parallelised for a fine-grained system, or if n is sufficiently large k partitions of $\left\lfloor \frac{n}{k} \right\rfloor$ data items can be formed into k tasks each performing all three steps on its data set. Since n is determined by the degree of the multivariate polynomial coefficient of a monomial it is unlikely that n will be large enough to warrant partitioning of the data set to exploit parallelism on a coarse-grained system.

For this reason and because Tlisp does not generally have sufficient processors to exploit parallelism at this level of nested parallelism the Tlisp implementation of PARALLEL_GCD does not make use of any parallelism in VDM-SOLVE.

4.7.3. Removing the Content of a Multivariate Polynomial in Parallel

Since ZIPPEL-GCD is only applicable to primitive polynomials (that is, those with the coefficients of X_0 having no common factor) an efficient algorithm to remove the content (gcd of the coefficients of X_0) from multivariate polynomials is required.

Consider a polynomial

$$F = c_1(X_1, \ldots, X_v)X_0^{e_1} + \cdots + c_n(X_1, \ldots, X_v)X_0^{e_n}.$$

The content of F is defined as

$$\mathrm{cont}(F) = \gcd(c_1, \ldots, c_n).$$

If any $\gcd(c_i, c_{i+1}) = 1$, for $i = 1, \ldots, n-1$, then $\mathrm{cont}(F) = 1$. Thus it would be more efficient to sort the coefficients, c_i, from least to greatest on degree and find the gcds of pairs of smaller coefficients first as these are more likely to have unit gcd. The following parallel algorithm is suggested by these observations.

Algorithm PARALLEL-NGCD

Input: n polynomials $F_1(X_0, \ldots, X_v), \ldots, F_n(X_0, \ldots, X_v)$
Output: gcd $(F_1, \ldots, F_n)$

Step PNG1 (Initialise)
$G \leftarrow (F_1, \ldots, F_n)$

Step PNG2 (Loop)
Do until $length(G) = 1$

Step PNG3 (Sort and compute gcds)
$G \leftarrow sort\_on\_degree(G, <)$
$G \leftarrow spec\_parallel\_apply$ (PARALLEL-GCD, G, 1, (1))

Step PNG4 Output remaining element of G

The iterated step PNG3 calls the function *spec_parallel_apply* which requires further explanation. Consider first a function *parallel_apply* which takes two arguments, a binary function name, f, and a list of data, $(d_1, \ldots, d_n)$. The function *parallel_apply* will return

$$(f(d_1, d_2), \ldots, f(d_{n-1}, d_n)) \quad \text{if } n \text{ is even}$$

or

$$(f(d_1, d_2), \ldots, f(d_{n-2}, d_{n-1}), d_n) \quad \text{if } n \text{ is odd}$$

and the computations $f(d_i, d_{i+1})$ are performed concurrently. Iterations continue until a single value remains in the data list.

The function *spec_parallel_apply* is similar to *parallel_apply* except that if any of the $f(d_i, d_{i+1})$ return a result equal to the third argument then all other $f(d_j, d_{j+1})$ computations are stopped and the fourth argument returned. Thus in PARALLEL-NGCD where

$$spec\_parallel\_apply\text{ (PARALLEL–GCD, } G, 1, (1))$$

is called for $G = (F_1, \ldots, F_n)$, as soon as any gcd(F_i, F_{i+1}) returns 1 then no further gcd computations are required since gcd$(F_1, \ldots, F_n) = 1$.

Speculative parallelism (see *e.g.* Osbourne, 1989) as used by *spec_parallel_apply* and as discussed in section 2.5.6 is not supported by Tlisp mainly due to lack of implementation facilities to abort a computation and reclaim processing resources.

The implementation of PARALLEL-NGCD for Tlisp uses a construct like

parallel_apply rather than *spec_parallel_apply* at Step PNG3. If any of the gcd computations have returned a 1 the function returns 1, otherwise the loop is repeated with the new list of polynomials.

Sorting the polynomials as discussed above is of no advantage when using the Tlisp implementation since the time required for each stage to complete is the maximum time of the $\gcd(F_i, F_{i+1})$ calculations at that stage. Since ordering the F_i so that this maximum is minimised is not possible, the arbitrary ordering that the F_i originally appear in is as good as any other.

4.7.4. Implementation of PARALLEL-GCD

As mentioned earlier the PARALLEL-GCD algorithm is implemented in Tlisp (see Chapter 2). Starting with a (sequential) polynomial arithmetic package written in Franz Lisp by Terry Robb and which included an implementation of the sequential Algorithm ZIPPEL-GCD, modifications and additions were made to enable the package to run under Tlisp. The gcd functions were further modified and then parallelised in accordance with Algorithm PARALLEL-GCD.

The main pieces of Tlisp code for the implementation of Algorithm PARALLEL-GCD are listed in Appendix B. Not included are the polynomial arithmetic functions and the utility functions.

4.8. The Probabilistic Nature of Zippel's Algorithm

Zippel's algorithm is probabilistic since assumptions are made throughout concerning the structure of the polynomial being computed. Thus the gcd algorithm presented here with Zippel's interpolation algorithm at its heart must also be probabilistic. This section looks at why the algorithm might fail.

The algorithm fails when a false assumption has been made. Each sparse interpolation is based on the assumption that coefficients which are zero in one dense interpolation are zero in all cases. If a bad assumption is made, the error will not be detected until the completion of the interpolation. A division check is employed to the test the correctness of the constructed gcd since the gcd of the two input polynomials must divide both those polynomials (see (Willcock, 1987) for an exposition of division checking methods).

The sparse interpolation algorithm (applied to each term of the univariate gcd at Step PG4) depends on a skeletal polynomial as an indication of the structure of the system of linear equations that must be solved. Clearly it is important that this skeletal

polynomial conforms to the true structure of the coefficient being lifted. If it does not, the next iteration will not only be wrong but most probably will be dense (Zippel, 1988).

Success of the interpolation algorithm is entirely dependent on the starting point $(x_{10}, \ldots, x_{v0})$. A bad starting point will lead to an incorrect assumption that a term is zero. To see how this comes about consider that at the end of Step PG3 X_0 and X_1 have been lifted using dense interpolations to give

$$G(X_0, X_1, x_{20}, \ldots, x_{v0}) = c_1(X_1, x_{20}, \ldots, x_{v0})X_0^{e_1} + \cdots + c_s(X_1, x_{20}, \ldots, x_{v0})X_0^{e_s}.$$

In Step PG4 each of the $c_i(X_1, x_{20}, \ldots, x_{v0})$ are lifted. Consider one of these

$$c(X_1, x_{20}, \ldots, x_{v0}) =$$

$$p_1(x_{20}, \ldots, x_{v0})X_1^{f_1} + \cdots + p_t(x_{20}, \ldots, x_{v0})X_1^{f_t}.$$

The algorithm assumes this structure to be correct. If p_i, for some $i = 1, \ldots, t$, vanished at $(x_{20}, \ldots, x_{v0})$ the assumption that the ith term is zero would be incorrect. Similarly after the next iteration,

$$c(X_1, X_2, x_{30}, \ldots, x_{v0}) =$$

$$q_1(x_{30}, \ldots, x_{v0})(X_1, X_2)^{\bar{f}_1} + \cdots + q_t(x_{30}, \ldots, x_{v0})(X_1, X_2)^{\bar{f}_t}$$

and if any of the q_i vanish at $(x_{30}, \ldots, x_{v0})$ bad assumptions will be made.

Continuing this analysis through all the lifting stages gives the generalisation that at the kth stage of the interpolation process there will be at most t polynomials in k variables, the zeroes of which must be avoided. Zippel (1990) shows that the probability of the starting point being a zero of any of these polynomials is

$$\frac{v^2 dt}{q}$$

where d is the maximum degree of the polynomials and the starting values are chosen from $\mathbf{Z}_q$, q a prime. Thus q is generally chosen to be near but not greater than the maximum single-word integer for the computer.

A second potential source of error is the possibility of one of the transposed Vandermonde systems being singular. This happens when the $\bar{y}^{\bar{e}_i}$ are not distinct. Guaranteeing the $\bar{y}^{\bar{e}_i}$ are distinct can be done if the coefficient domain is a unique factorisation domain – simply choose $\bar{y}$ to be distinct primes, for example, if the coefficient domain is $\mathbf{Z}$ then $\bar{y} = (2, 3, 5, \cdots)$.

In the case of the coefficient domain being a finite field, for example, $\mathbf{Z}_q$, then again

primes may be chosen for the components of $\bar{y}$ provided the $\bar{y}^{\bar{e}_i}$ computed in $\mathbf{Z}$ are less than q, for then the $\bar{y}^{\bar{e}_i}$ will be distinct in $\mathbf{Z}_q$.

When the first $k-1$ primes do not make a suitable choice for $\bar{y}$, Zippel (1990) shows that the probability that a randomly chosen $\bar{y}$ will cause two of the $\bar{y}^{\bar{e}_i}$ to have the same value is

$$\frac{d.T.(T-1)}{2(q-1)}$$

where T is an upper bound on the number of terms in the goal polynomial. In other words for the probability of a collision to be less than ε

$$q > \frac{(d+1)^{2\nu+1}}{2\varepsilon}$$

This limitation is not unsatisfactory for problems where the goal polynomial is sparse *i.e.* $T \ll (d+1)^{\nu}$.

The implementation of Algorithm PARALLEL-GCD insures the transposed Vandermonde systems will be non-singular by randomly choosing $\bar{y}$, checking that the $\bar{y}^{\bar{e}_i}$ are distinct and choosing new components for $\bar{y}$ as necessary.

Chapter 5

Performance of the Parallel GCD Algorithm

5.1. Introduction

This chapter looks at the performance of Tlisp with the parallel gcd algorithm by way of a series of experiments using various gcd problems and network configurations. After some discussion about the experiments in general the algorithm is broken down into its parallel and sequential components. The parallel components are investigated individually. Overall performance of the algorithm is then presented graphically for a small range of gcd problems and in less detail for a larger range of problems. Performance of the GCDHEU algorithm is also looked into.

5.1.1. Choice of Data

One of the first problems which arises when planning a strategy to measure the performance of any given algorithm is the choice of input data. This is a particular problem for symbolic computation algorithms where the analysis of the algorithms is often complicated by an inadequate characterisation of inputs and their relationship to running time. For example the running time of Zippel's sparse modular gcd algorithm is largely dependent on the number of terms (non-zero, multivariate polynomial coefficients of the main variable) in the resultant gcd, not on properties of the input polynomials.

Further, a range of terms that would be *typical* for gcds is not easy to determine as this varies between practical applications.

Choosing random polynomials (somehow defined) as input data would be counter-productive as the probability of any two such polynomials having a non-trivial gcd is extremely small. When the gcd is required of two polynomials it is reasonable to assume that they have not been chosen at random and have a reasonable chance of having a non-trivial gcd.

Two sets of polynomial gcd problems for testing the performance of gcd algorithms appear in the literature. One is given by Wang (1980) for his EEZ-GCD algorithm,

the other is given by Zippel (1979). Neither set give any indication as to how they were chosen, but Zippel's set consists of ten problems increasing in number of variables from one to ten. Neither set has much variance in the number of terms in their gcds.

The gcd problems that will be used to test the performance of PARALLEL-ZGCD include the Wang and Zippel problem sets (labelled W1 to W10 and Z1 to Z10 respectively), a polynomial pair (M1) with a larger number of terms in their gcd and a set of polynomial pairs (T1 to T10) with trivial gcds. The gcd problem sets are given in Appendix C.

5.1.2. Model for Performance Statistics

In investigating the performance of a given concurrent processing system with a given problem two metrics are commonly defined – the speedup gained over a sequential solution to the problem and the efficiency, or average utilisation, of the processors (see for example, Flatt and Kennedy, 1989 or Eager *et al,* 1989). Accordingly speedup and efficiency for a given problem will be defined here as follows.

Let T_1' be the time required for execution of the best known sequential algorithm on a single processor. Let T_p be the time required for execution of the parallel algorithm on p processors. Then the *true speedup* is given by

$$S_p' = \frac{T_1'}{T_p}$$

and *true efficiency* is given by

$$E_p' = \frac{1}{S_p'}$$

It is not always feasible to calculate true speedup and efficiency since T_1' may not be readily available. It has become common practice to use *rough speedup* and *rough efficiency* based on T_1, the time required for execution of a sequential version of the parallel algorithm on a single processor. Thus rough speedup is given by

$$S_p = \frac{T_1}{T_p}$$

and rough efficiency by

$$E_p = \frac{1}{S_p}$$

These are the two metrics that will be used in this chapter for investigating the performance of Tlisp and PARALLEL-ZGCD. Further references to speedup or

efficiency will mean rough speedup or rough efficiency respectively.

In an ideal situation, the total amount of processing will be divided among p processors which will all be busy for the duration of the computation, and so speedup will be equal to p and efficiency to one. However, in a real situation there will be overheads associated with setting up the parallel jobs and inter-processor communication and there will be some fraction of the algorithm in question that is inherently sequential or cannot be distributed over all available processors. Thus speedup may be said to be bounded above by p and efficiency by one.

The arguments of the above paragraph are not completely adequate for a parallel lisp system, such as Tlisp, with a distributed memory. If G_s is the number of garbage collections required for sequential execution of a given algorithm and G_p is the total number required for parallel execution on p processors then, provided each of the processors have the same amount of memory, the p processors have p times more memory than a single processor and $G_s \geq G_p$. Thus the total amount of work required for the parallel solution may be less than that for the sequential solution. Consequently, the upper bound on speedup may be given by

$$S_p \leq p + m(p)$$

where $m(p) \geq 0$ and is the contribution to speedup from having more memory but which is distributed over the p processors. The upper bound on efficiency is therefore greater than unity. The magnitude of $m(p)$ will be discussed later in this chapter.

5.1.3. Communication Overheads

Using the gcd problem M1 from Appendix C, eight remote jobs were investigated to give some indication of communication overheads incurred by Tlisp's handling of remote jobs. Execution of a remote job involves first destructing the s-expression into a character string, transmitting the string via the communications coordinator process to an eval-server (which may or may not be a physically neighbouring processor), then restructing the string into an s-expression, evaluating it and repeating the destructure-transmit-restructure procedure with the result of the evaluation.

Timings (in 1/100 sec) were obtained for the destructing and restructing of the s-expressions. Obtaining measurements for interprocessor communication was not straight forward since any client to eval-server message transfer could be routed through one or more intermediate processors (by the Helios operating system).

Data presented in (Perihelion, 1991) indicate that the actual inter-transputer data

transfer time is approximately $1461.42+0.57N$ μs where N is the message size in bytes. Thus a message (character string) of length 1000 characters travelling between two adjacent transputers requires about 0.20 1/100 sec, which is not significant when compared to destructure and restructure times. The contribution to communication times from through-routing by Helios is reported to be very slight (Perihelion, 1991). The contribution from the CCP was not possible to measure but can reasonably be assumed to also be small relative to destructure and restructure times.

Results from the investigation are presented in Table 5.1.1. and the eight jobs are as follows (the named Tlisp functions are given in Appendix B):

1 a typical call to zippel-degrees-gcd,

2 a call to zippel-Fgcd from step PG3 to compute a univariate gcd,

3 a call to zippel-lift-term to lift the coefficient of X_0^1,

4 a call to zippel-lift-term to lift the coefficient of X_0^8,

5 a call to zippel_lift_next_vble to lift X_2 in the coefficient of X_0^0,

6 a call to zippel_lift_next_vble to lift X_2 in the coefficient of X_0^5,

7 a call to vdm_derived_poly during the lifting of X_2 in the coefficient of X_0^2,

8 a call to vdm_derived_poly during the lifting of X_2 in the coefficient of X_0^9.

Remote Job	Client-Evalserv msg			Evaluation time	Evalserv-Client msg		
	nr. char in msg	destruct time	restruct time		nr. char in msg	destruct time	restruct time
1	98	1	1	8	3	0	0
2	1114	5	10	367	116	1	1
3	1988	11	20	7140	13	0	0
4	2122	11	20	14748	21	0	0
5	2082	11	19	7042	13	0	0
6	2113	11	19	7181	13	0	0
7	1144	7	12	1114	12	0	0
8	1131	6	12	530	11	0	0

Table 5.1.1 *Execution times for a selection of remote jobs showing communication overheads*

The size of each remote job in the table is large relative to its communication time. However it is evident that jobs do need to have large execution times to be worthy of sending to a remote processor. Consider, as a base for comparison, that a single *cons* expression requires only 0.02 1/100sec. This experiment has served to confirm statements in Chapter 2 about the coarse granularity of Tlisp.

5.1.4. Accuracy of Timings

All timings given in this chapter were averaged over at least three runs and are given in units of 1/100sec. There is a certain amount of irreproducibility in the measured timings as they are affected to a varying degree by both the use of random numbers in the algorithm and by an undetermined contribution from the Helios operating system which runs processes on the transputers for its own purposes, such as through-routing messages. As an indication of how accurate the timings are, a selection of gcd problems were chosen from the sets in Appendix C and some basic statistical analysis performed on a set of ten timings for each of their solutions on various Tlisp configurations (the configurations are explained in section 5.2).

GCD Problem	Sequential		Tree3		Farm3		Farm7	
	$\bar{x}$	cv	$\bar{x}$	cv	$\bar{x}$	cv	$\bar{x}$	cv
W1	9405.2	0.8	3359.7	1.4	3278.3	1.8	2579.6	2.9
W3	88083.8	1.6	31854.1	1.1	32157.6	3.6	20236.9	5.4
M1	129902.6	0.6	22298.2	1.2	33544.2	0.8	18862.3	1.2

Table 5.1.2 *Statistics for timings of the solution of three gcd problems on various configurations, where $\bar{x}$ is the mean time and the coefficient of variance, $cv = 100s/\bar{x}$ expresses the variance, s, as a percentage of $\bar{x}$.*

In most instances the variance is acceptable (less than 3%). There does, however, exist the possibility of a larger variance (W3, for example, on a farm of seven eval-servers gives a 5.4% variance). In light of these observations (and the time restrictions on this research) it was decided that an average of three timings would be an adequate measure in most cases. Where three timings were seen to give a large variance, more were taken to give a more accurate mean.

5.2. The Parallel GCD Experiments

Except where otherwise stated the experiments discussed in this section were repeated for each of three different configurations; a farm of three eval-servers each with two eval-servers of its own (Tree3), a farm of three eval-servers (Farm3) and a farm of seven eval-servers (Farm7). The Tree3 configuration was chosen as the best tree possible with the limited number of transputers available. The Farm3 configuration compares the performance of Tree3 with a similar configuration without the leaf nodes. The Farm7 configuration compares the farm of three with a similar configuration that uses twice as many transputers and is also the maximum farm size possible with a board of eight transputers. The transputers used were all 25 MHz T800's except for two leaf-nodes on the tree configuration which were slower 20 MHz T800's.

Times are given in 1/100 sec and (as discussed in section 5.1.4) are averaged over at

least three runs.

5.2.1. Examination of the Parallel Steps of the Algorithm

To gain insight into how well PARALLEL-GCD performs with the Tlisp system (and vice-versa) the various parallel steps of the algorithm were examined individually to see which give useful parallelisations and which do not.

Step PG1 requires that the contents of each of two polynomials be computed. Since the content computations are independent of each other they may be done concurrently. However, since the two calculations will not necessarily be of similar size, it is possible that the processing resources would be more efficiently utilised if the contents were calculated sequentially with each able to make maximum use of the available processors using the parallel scheme discussed in section 4.7.3. A set of experiments were performed to investigate this possibility. The results are summarised in the Table 5.2.1 which gives the ratios of the time for taking the contents one after the other over the time for taking them concurrently. A ratio greater than one indicates that, in that instance, taking the contents concurrently is the superior strategy.

Data Set	Tree3	Farm3	Farm7
W1*	0.8	0.8	0.8
W2*	0.5	0.5	0.5
W3*	0.4	0.4	0.4
W4	1.3	1.1	1.2
W5	0.9	1.5	1.4
W6	0.9	1.0	1.0
W7	0.9	1.9	2.0
W8	0.9	2.4	2.5
W9	0.7	1.1	1.1
W10	0.7	2.5	2.7
M1	1.4	1.5	1.6

Table 5.2.1: *Comparison of two possible parallelisations of the content taking step of PARALLEL-GCD with various configurations. Data given is the ratio of times A:B where A is the time for taking the contents one after the other, B is the time for taking them concurrently.*

In Table 5.2.1 an asterisk indicates that the actual time difference is very small (10-20 1/100 sec) since the contents are both immediately trivial (that is, one of the coefficients is 1); the slow down indicates the overhead of creating a redundant remote call.

From the table it is apparent, even with a small data sample, that taking the two contents concurrently gives a worthwhile speedup or makes little difference in most cases.

Step PG4, the sparse interpolation of the gcd to lift the variables above X_1, has two levels of parallelisation. At the first level the terms can be lifted concurrently and at the second level the lifting process itself has parallel two steps – (i) setting up and solving the systems of equations and (ii) interpolating the sets of coefficients using the Chinese remainder algorithm. While it is clear that for speedup the terms should be lifted concurrently (since a relatively large amount of processing is required), it is not immediately clear whether either or both loops in the lifting process are worth parallelising. Consequently the lifting process was the subject of the next set of experiments.

Data for several lifting stages was collated to investigate how best to parallelise Step PG4. Using speedup gained by parallel execution as defined in section 5.1.2, timings were obtained to measure

A. speedup with the first loop parallelised,
B. speedup with the second loop parallelised,
C. speedup with both loops parallelised.

Tree configurations were not considered for these experiments since the parallelism is at only one level (the top level is not being considered) and the leaf nodes would receive no work. The results are summarised in Table 5.2.2 and the terms to be lifted, taken from the data sets in Appendix C are listed below.

1. Lifting X_2 in the quadratic term from M10
   $c_1X_1^3 \rightarrow -10X_1^3X_2^{10}$.
2. Lifting X_3 in the constant term from W1
   $c_1X_1^4 + c_2X_2^4 + c_3 \rightarrow X_1^4X_3^3 + X_2^3 + X_3$.
3. Lifting X_2 in the constant term from W3
   $c_1X_1^5 + c_2 \rightarrow 7557001X_1^5 + 7557001X_2^5 + 196538433$.
4. Lifting X_3 in the constant term from W3
   $c_1X_1^5 + c_2X_2 + c_3 \rightarrow 7557001X_1^5 + 7557001X_2^5 + 7557001X_3^5 + 215142238$.
5. Lifting X_4 in the constant term from W3
   $c_1X_1^5 + c_2X_2 + c_3X_3 + c_4 \rightarrow$
   $7557001X_1^5 + 7557001X_2^5 + 7557001X_3^5 + 7557001X_4^5 + 970006314$.
6. Lifting X_5 in the constant term from W3
   $c_1X_1^5 + c_2X_2 + c_3X_3 + c_4X_4 + c_5 \rightarrow$
   $7557001X_1^5 + 7557001X_2^5 + 7557001X_3^5 + 774195609X_4^5 + 774195609X_5^5$
   $+ 950454437$.
7. Lifting X_6 in the constant term from W3
   $c_1X_1^5 + c_2X_2 + c_3X_3 + c_4X_4 + c_5X_5 + c_6 \rightarrow X_1^5 + X_2^5 + X_3^5 + X_4^5 + X_5^5 + X_6^5$.

Data Set	Sequential Time	Farm3			Farm7		
		A	B	C	A	B	C
1	7540	3.09	0.97	3.07	5.36	0.95	5.25
2	4279	2.64	0.98	2.73	4.07	0.97	4.32
3	1409	2.78	0.97	2.75	4.01	0.96	4.01
4	4382	3.18	0.97	3.13	4.90	0.95	4.91
5	8611	3.26	0.96	3.26	5.17	0.95	5.17
6	16979	3.39	0.97	3.36	5.39	0.95	5.38
7	42601	3.44	0.97	3.44	5.43	0.95	5.50

Table 5.2.2: *Comparison of speedup gained from different parallelisations for Zippel's sparse interpolation algorithm as a part of PARALLEL-GCD* using two farm configurations.

From the columns labelled "A" in Table 5.2.2 it is apparent that the first loop of Step PG4 is worth parallelising. However the columns labelled "B" seem to indicate that the second loop is not worth parallelising (the grain size is too fine). It is interesting to note that the best speedup for each data set is sometimes obtained with both loops parallelised although parallelising only the second gives a slow down. This is possibly a result of the extra computations required in gathering the timings. The difference is only slight and a decision was made not to parallelise the second loop of Step PG4.

Having established the best choices for parallelising the content taking procedure and Step PG4, the algorithm was broken down into its parallel steps, namely

A. content taking and removal,
B. calculation of maximum degrees of the variables in the result,
C. lifting X_1 - Step PG3,
D. lifting the variables above X_1 - Step PG4.

The speedup gained for each step and for the whole algorithm was measured on three different configurations for a selection of data sets (chosen for a variety of values for number of terms and number of variables) from those given in Appendix C. The results are summarised in Table 5.2.3.

GCD Problem		Sequential Time	Tree3 Speedup	Farm3 Speedup	Farm7 Speedup
W1	A	10	1.00	1.00	1.00
	B	767	1.56	1.56	1.56
4 vbles	C	415	2.00	2.01	2.43
	D	8181	3.40	3.50	4.56
2 terms	Tot	9405	2.80	2.87	3.95
W3	A	12	1.00	1.00	1.00
	B	4938	1.69	1.87	1.74
7 vbles	C	478	1.97	1.99	2.29
	D	82504	2.98	3.10	5.09
2 terms	Tot	88084	2.77	2.74	4.35
Z7	A	19268	1.75	1.57	1.60
	B	2759	1.54	1.59	1.57
4 vbles	C	713	2.90	2.90	2.84
	D	57314	5.37	4.51	7.67
4 terms	Tot	84592	2.93	2.29	2.29
M1	A	605	0.49	0.49	0.67
	B	2509	2.16	2.18	2.16
3 vbles	C	6467	3.56	3.71	4.82
	D	111914	6.82	4.53	12.68
10 terms	Tot	129903	5.82	3.87	6.89

Table 5.2.3: *Speedups achieved for parallel steps of PARALLEL-GCD on various configurations.*

The data presented in Table 5.2.3 reveals much about the nature of PARALLEL-GCD. Steps A, B and C give a small to medium amount of speedup but it is clear that step D, the sparse interpolation of each term in the gcd, yields the only really significant speedup. Further, since the farm configurations were not able to exploit the second level parallelism available in step D (*cf.* Table 5.2.2) it is reasonable to assume that if more processors were available the sparse interpolation part of PARALLEL-GCD would experience greater speedup than has been obtained here.

From the speedups of Step D of problem M1, it is evident that the contribution to speedup from the distributed memory can be significant (without the contribution from distributive memory, speedup in the last column, for example, would be bounded above by 8, the number of processors being used). In the case of problem M1, the size of each job created in Step D is large and since the code is interpreted it is reasonable to expect many garbage collections are required for each job. This would account for the large contribution to speedup from the distributed memory in the parallel solution of gcd problem M1. In general, however, it is not possible to make assumptions about the size of this contribution (and neither was it possible to measure it).

5.2.2. Performance of the Algorithm

Four polynomial gcd problems with varying numbers of variables and terms in their gcds, W1 (4 variables, 2 terms), W5 (4 variables, 5 terms), Z6 (6 variables, 2 terms) and M1 (3 variables, 10 terms), were chosen as indicators for the performance of Tlisp and PARALLEL-GCD.

The results of the experiments are displayed in Figures 5.1 to 5.12. Figures 5.1 to 5.4 show the speedup and efficiency statistics for the four gcd problems on various configurations. Each graph represents two different types of configurations - farms and trees. Each farm is simply a client with the indicated number of eval-servers. Each tree is a client with the indicated number of eval-servers each of which acts as a client for its own two eval-servers. There are statistics from only two tree configurations as there were insufficient transputers available for this set of experiments to make larger trees. Figures 5.5 to 5.8 show how the work is distributed over a farm of seven eval-servers and their client for each of the four problems. Figures 5.9 to 5.12 show the same statistics for a tree configuration of three eval-servers each with its own two eval-servers.

Much of what is shown in the twelve figures can be deduced by reasoning based on knowledge of the parallel algorithm and earlier experiments. From Table 5.2.3 it is clear that the sparse interpolation of each of the terms in the gcd generally accounts for most of the sequential execution time. It is also the portion of PARALLEL-GCD which gives the best speedup. Thus how well suited the hardware configuration and the gcd problem are to this part of the parallel algorithm will dictate performance.

For farm configurations the number of terms in the gcd will greatly influence how well the eval-servers are utilised since one remote job is created for each term (except the last which is evaluated locally) and these remote jobs cannot create more remote jobs. The size of each of these jobs is determined by the number of variables in the problem and the maximum degree to which each occurs in the gcd. From these observations it is possible to predict increasing speedup as n, the number of eval-servers, approaches t, the number of terms in the gcd, and a leveling off of speedup once n exceeds t. Particularly good efficiency will be obtained if n is much less than t or if t is near some multiple of n.

These predictions are verified by the graphs for farm configurations in Figures 5.1 to 5.4. Speedup is seen to increase in each graph as n approaches t and the first three graphs show speedup levelling off as n exceeds t. Best performance is gained for problem M1 (Figure 5.4) where t is significantly greater than n.

For the tree configurations the same reasoning as above can be applied for the first

level eval-servers. The results shown in Table 5.2.2 can then be used to reason that presence of leaf nodes will speed up the execution of jobs on their clients. Thus speedup for a farm of n trees should be better than that for a farm of n eval-servers, but the trends as n increases should be the same. These reasonings are supported by the tree configuration graphs, restricted to two values of n, in Figures 5.1 to 5.4.

The reasonings given above are further supported by Figures 5.5 to 5.12. For example Figure 5.6 shows the first 22 time intervals are concerned with initialisation, content taking and Step PG3. The creation of five jobs (one for each term) during Step PG4 is then evident. Approximately ten time intervals are required to execute the fifth job which is done locally and uses the three eval-servers not already occupied. The client then becomes idle until the four remotely executed jobs are complete and the client collates the results and finishes up. For about twelve time intervals (about a quarter of the total execution time) four of the processors are idle.

Figure 5.7 shows similar trends to Figure 5.6. After about 22 time intervals the sparse interpolation of each term begins. In this instance there are only two terms and hence two jobs created. The locally executed job occupies the client and five eval-servers for about ten time intervals, then the client and six eval-servers must wait for the remote job to complete. One eval-server is not utilised at all during solution of the problem.

Figure 5.8 can be interpreted as follows. The first 15 time intervals are concerned with initialisation, content removal and lifting the second variable. The rest of the execution time is taken up with lifting the third variable for each of the 10 terms in the gcd. There are 10 approximately evenly sized jobs created by the client which executes the last one itself, creating more jobs from the second level parallelism. These second level jobs wait until an eval-server becomes available after the first seven of the terms have been lifted.

Figure 5.5 is for a smaller problem. Initialisation and content removal take only a few time intervals. Two small jobs are created for the sparse interpolation, one is done by the client and uses four eval-servers, the other is done remotely. The graph shows that two of the seven eval-servers are not used at all, and the other five are only used effectively for about one third of the execution time.

Utilisation of the processors when configured in a tree of three eval-servers each with two eval-servers of their own is shown in Figures 5.9 to 5.12. Although not as easy to interpret, the graphs show trends similar to those of the farm configuration – much of the work is pushed to the leaf nodes while the clients remain idle and efficient utilisation of the processors is only achieved when the number of terms in the gcd is

near a multiple of the number of eval-servers at the first level.

The performance figures presented in the graphs indicate that the gcd algorithm is worth parallelising for a coarse-grained, message-passing system. Using the Tlisp implementation, performance varies greatly according to each particular gcd being computed and the number and the configuration of the processors being used. Some trade-off between efficiency and speedup is necessary: the tree configurations appear to give greatest speedup but at the cost of poor efficiency. Better use could be made of the processors if a more flexible scheduling system than Tlisp's hierarchical one were used.

As a final performance experiment, speedup was measured for each gcd problem in Appendix C on a farm of seven eval-servers which, as earlier experiments indicate, gives the best speedup (but not necessarily efficiency) for the number of processors available. Results are given in Table 5.2.4, where an asterisk indicates the system ran out of memory space for the parallel execution. Speedups range from 1.00 (none at all) to 6.89 (showing good efficiency at 0.86). Average speedup is 2.43 which gives average efficiency of 0.30.

Data	v	t	Seq. Time	Speedup	Data	v	t	Seq. Time	Speedup
W1	4	2	9405	3.95	Z1	1	3	46	1.00
W2	6	2	53189	4.45	Z2	2	2	1161	1.84
W3	7	2	88084	4.35	Z3	3	3	3671	1.88
W4	3	4	17177	2.92	Z4	4	2	28353	2.00
W5	4	5	60411	3.65	Z5	5	3	55710	1.96
W6	4	2	55821	3.58	Z6	6	2	65672	2.37
W7	4	4	28385	1.78	Z7	7	3	84592	2.29
W8	4	5	45958	2.46	Z8	8	2	156115	1.72
W9	3	4	49178	2.91	Z9	9	3	113625	2.21
W10	3	6	404688	*	Z10	10	3	192719	2.35
T1	1	1	58	1.00	T6	6	1	10992	1.68
T2	2	1	476	1.47	T7	7	1	24255	1.83
T3	3	1	1384	1.28	T8	8	1	43718	1.98
T4	4	1	9285	1.67	T9	9	1	40891	1.76
T5	5	1	15235	1.96	T10	10	1	46092	1.96
M1	3	10	129903	6.89					

Table 5.2.4: *Sequential times and speedups achieved using farm of 7 eval-servers for PARALLEL-GCD,* v *is the number of variables in the problem and* t *is the number of terms in the resultant gcd.*

5.3. Incorporating the GCDHEU Algorithm

As mentioned in section 3.3.2 the GCDHEU algorithm has some merit when used in conjunction with Zippel's algorithm. Recall that the algorithm attempts to find the gcd of any two given polynomials by evaluating them at a point, finding the integer

gcd and using it as a base to construct the polynomial gcd. The algorithm fails if the integer coefficients of intermediate expressions exceed some nominal limit or if the evaluation point gives rise to a constructed polynomial which does not divide both of the input polynomials. Several evaluation points may be used if necessary, until some upper limit is reached or the correct gcd found. Where speculative parallelism is supported the computations with different evaluation points may be started concurrently and when one returns the correct gcd any others still active may be terminated. Since Tlisp does not support speculative parallelism only one evaluation point is used.

Table 5.3.1 compares the speed of GCDHEU and sequential and parallel execution of PARALLEL-GCD. Since performance of GCDHEU is poor for polynomials with more than four variables (cf. Geddes *et al*, 1990), only those gcd problems given in Appendix C which have no more than four variables are investigated. The table shows times for (sequential) execution of GCDHEU and the speedup (or slow down) gained by GCDHEU relative to PARALLEL-GCD executed both sequentially (P-GCD(seq)) and on a farm of seven eval-servers (P-GCD(7)).

GCD Problem	GCDHEU	P-GCD(seq)/ GCDHEU	P-GCD(7)/ GCDHEU
W1	8555	1.10	0.28
W4	1517	11.32	3.38
W6	4038	13.82	3.86
W7	6212*	4.57*	2.56*
W10	2118*	191.07*	-
Z1	107	0.43	0.43
Z2	365	3.18	1.72
Z3	796	4.61	2.42
Z4	1862	15.23	7.63
M1	60181	2.15	0.31
T1	50	1.16	1.16
T2	105	4.53	3.08
T3	273*	5.07*	3.94*
T4	804*	11.55*	6.92*

Table 5.3.1 *Comparison of GCDHEU and PARALLEL-GCD. An asterisk indicates that the gcd was not found after one iteration of GCDHEU, a dash indicates that the parallel execution ran out of memory space.*

The data presented in the table show that while GCDHEU is generally faster than PARALLEL-GCD it does not guarantee a solution. Speculative parallelism must be supported in order to reap the benefits of both algorithms. Then for any gcd problem both algorithms could be applied concurrently. GCDHEU need only occupy one processor (if many are available several can be used, each for a different evaluation point) while the others may be used for PARALLEL-GCD. When either algorithm finds the gcd the other processors may be stopped.

5.4. Summary of Results of Experiments

Although the experiments detailed in this chapter have been limited by a lack of characterisation of input data and a small range of input data, and restricted by the number of transputers available, the trends in the performance of PARALLEL-GCD as implemented for the Tlisp system are evident. Most notable (from Tables 5.2.3. and 5.2.4) is that the algorithm as a whole does not contain enough coarse-grained parallelism to effectively utilise more than about three eval-servers at the first level. Secondly, Zippel's sparse interpolation algorithm is well suited to implementation on the parallel Tlisp system as demonstrated by the results in Table 5.2.2. Consequently, the Tlisp implementation of PARALLEL-GCD gives best results when the sparse interpolation step dominates the execution time; that is, when the problem involves more than three variables (or, more than one to be lifted using the sparse scheme). Further, the greater the number of terms in the gcd, the greater the number of eval-servers that can be utilised at the first level.

What is clear from these experiments is that PARALLEL-GCD would perform better if implemented on a parallel system with a finer grain than Tlisp, with implicit *requests* and with less restrictions on job allocation. The advantages of a finer grain are obvious – any cursory look at the code for PARALLEL-GCD in Appendix B will reveal much unexploited medium-grained parallelism (for example, the second loop of Step PG4 - see Table 5.2.2 above). Implicit requests (where a *request* is done automatically by the system when a placeholder value is required) lead to more elegant code and encourage exploitation of parallelism. A system which allows any job to be scheduled on any available processor will make more efficient use of the processors than Tlisp, which often has both waiting jobs and idle processors but is restricted by its simple hierarchical scheduling scheme.

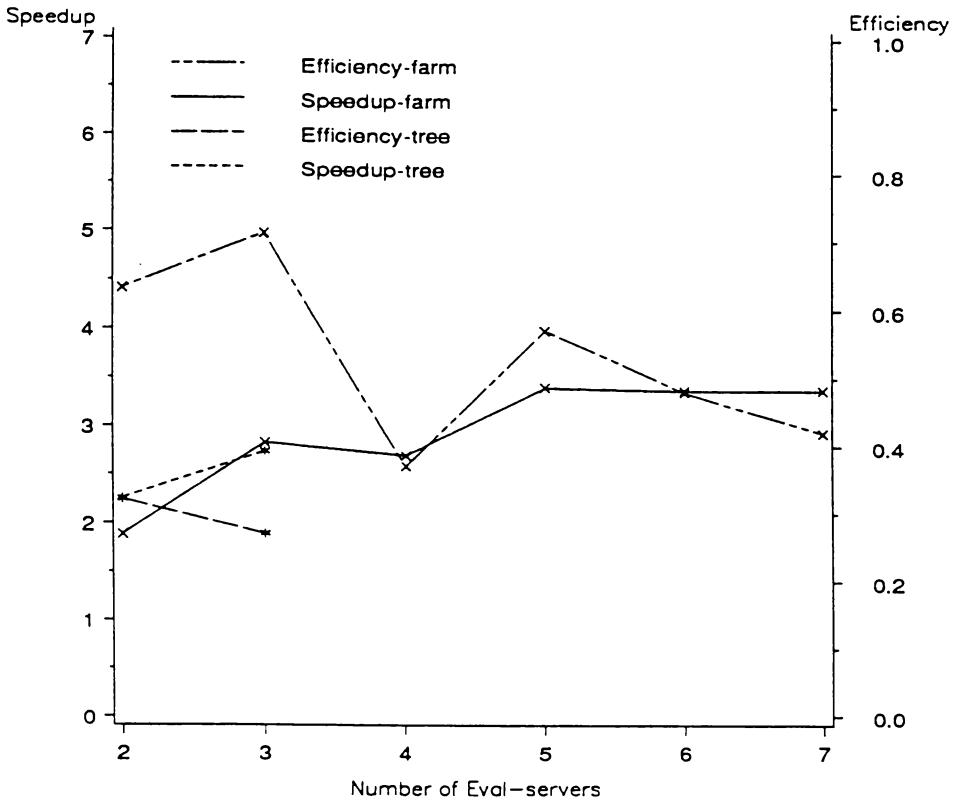

Figure 5.1 *Performance of Problem W1: four variables and two terms in the gcd*

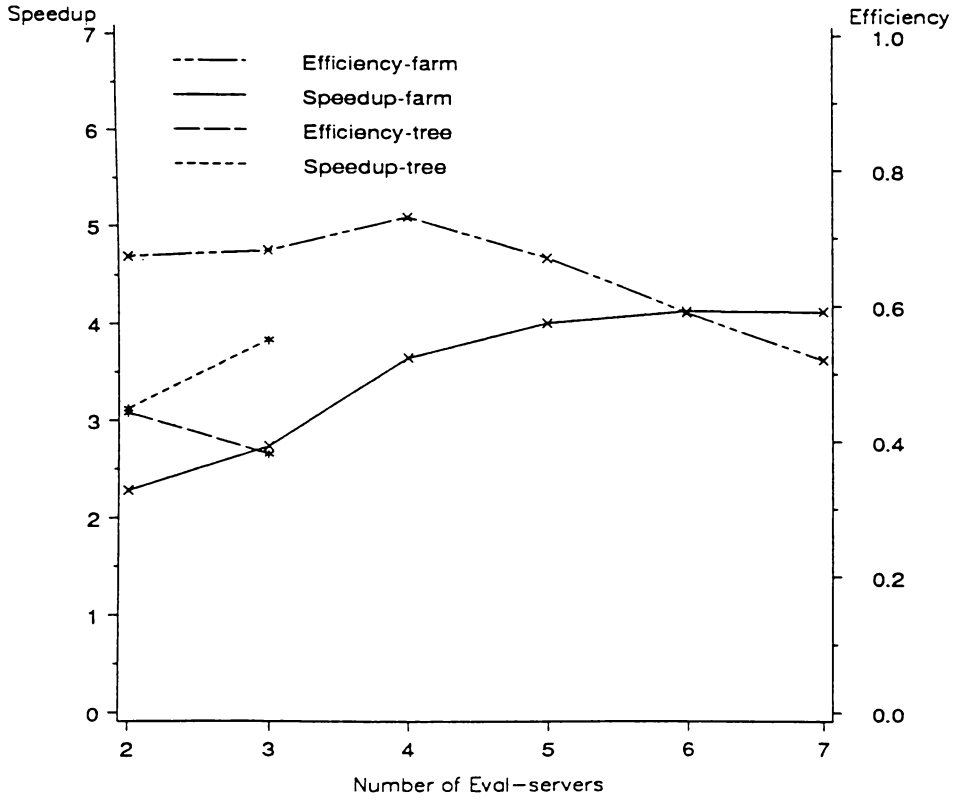

Figure 5.2 *Performance of Problem W5: four variables and five terms in the gcd*

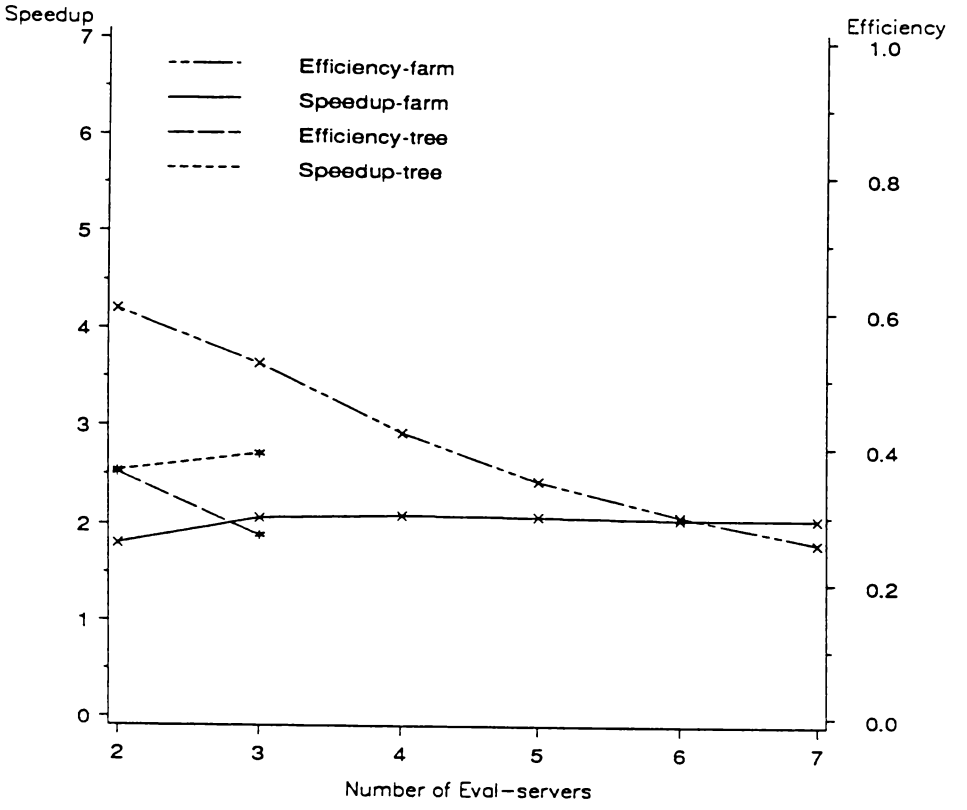

Figure 5.3 *Performance of Problem Z6: six variables and two terms in the gcd*

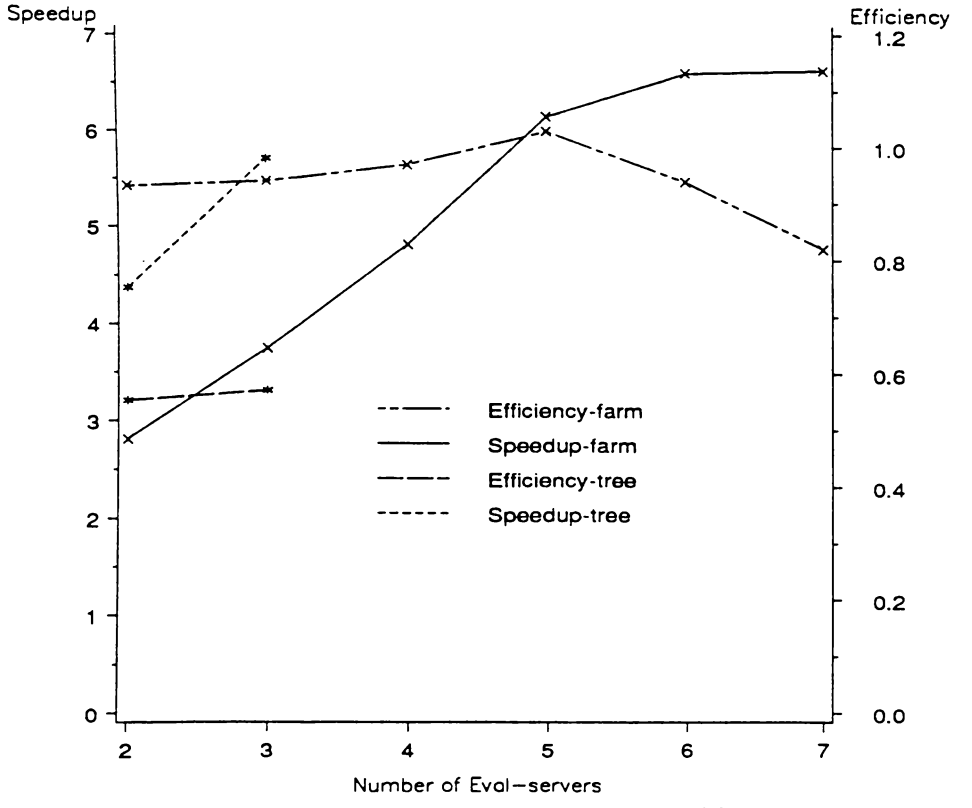

Figure 5.4 *Performance of Problem M1: three variables and ten terms in the gcd*

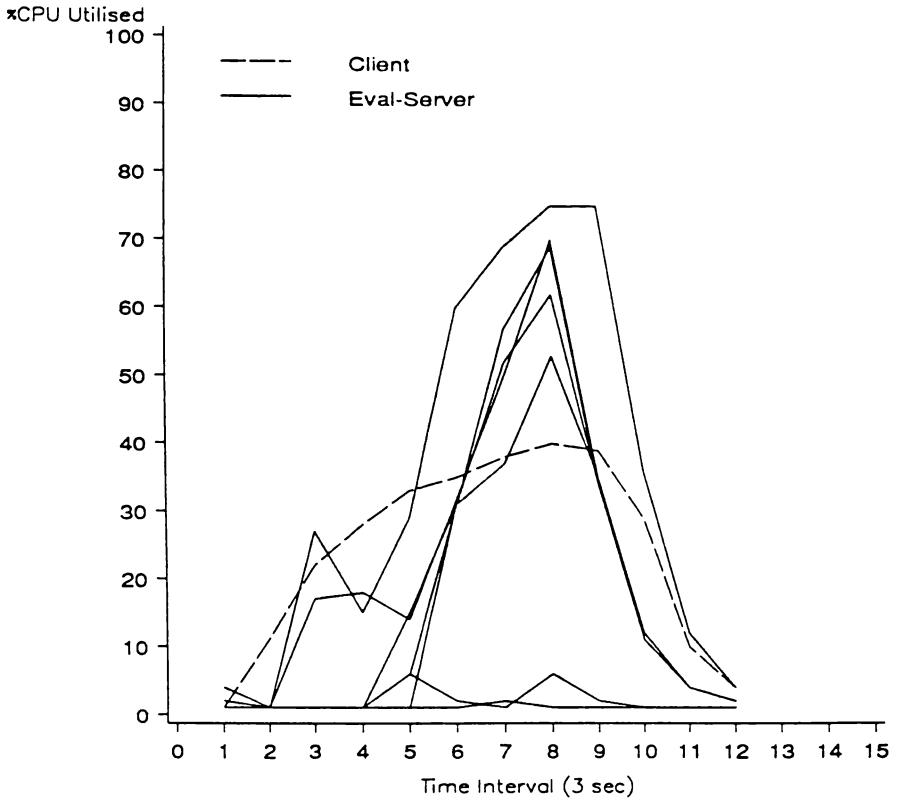

Figure 5.5 *Load at 3 sec intervals, farm configuration. Problem W1: two terms in gcd*

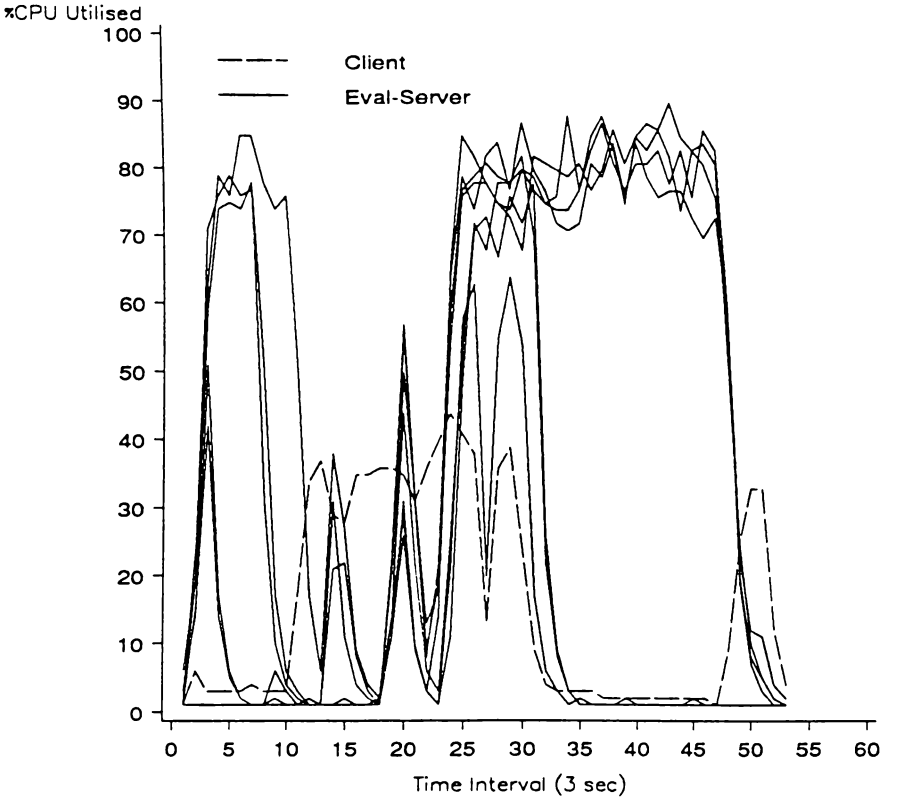

Figure 5.6 *Load at 3 sec intervals, farm configuration. Problem W5: five terms in gcd*

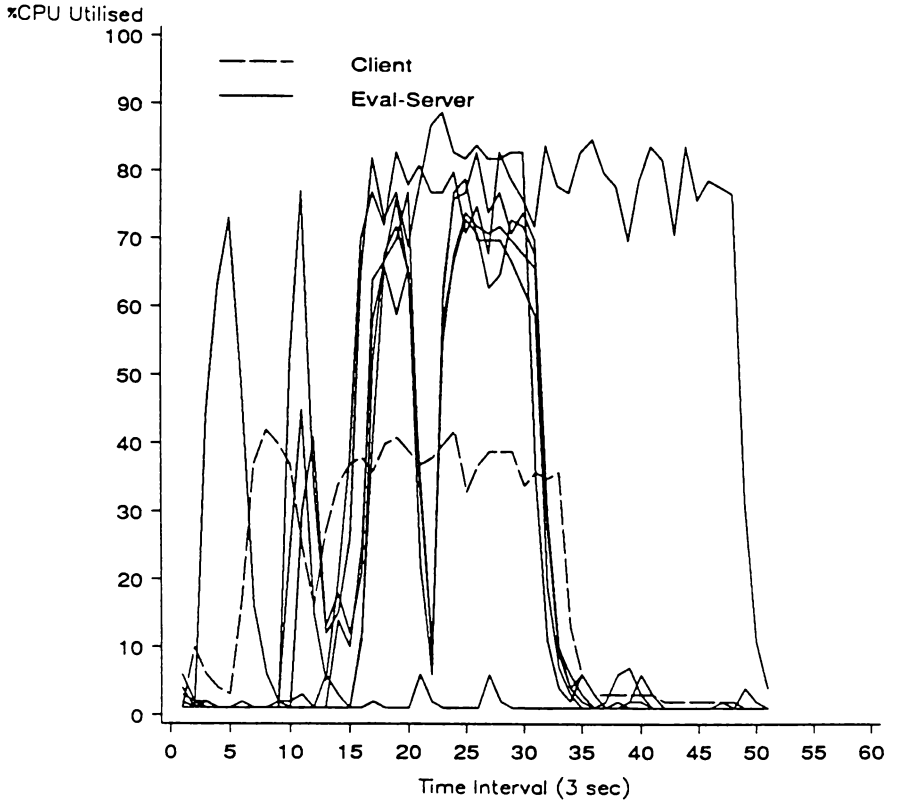

Figure 5.7 *Load at 3 sec intervals, farm configuration. Problem Z6: two terms in gcd*

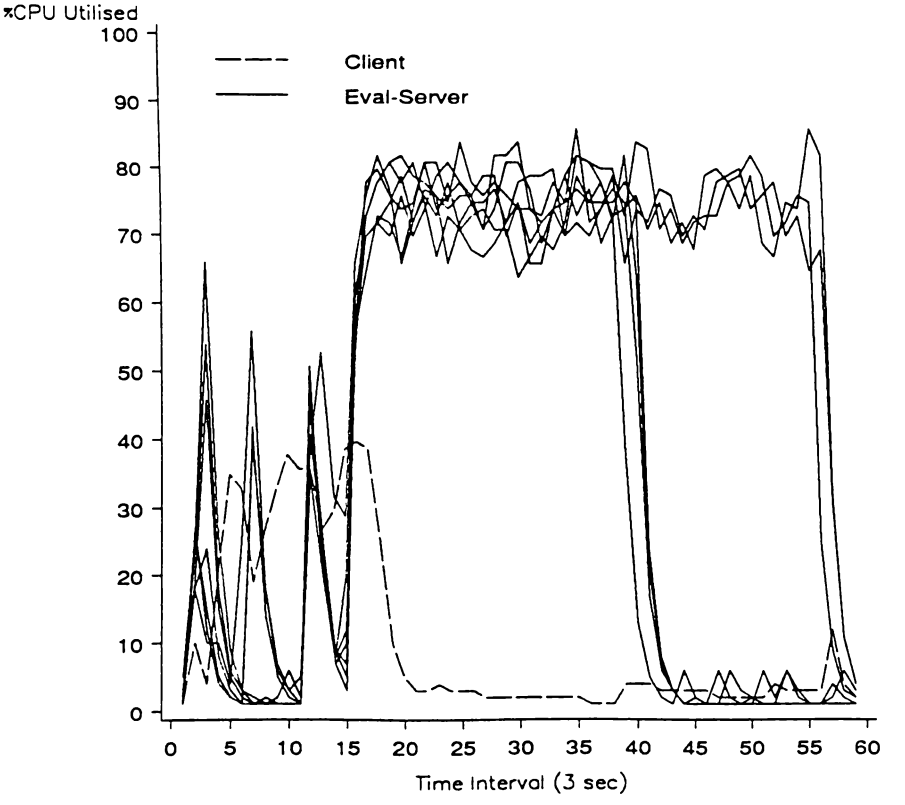

Figure 5.8 *Load at 3 sec intervals, farm configuration. Problem M1: ten terms in gcd*

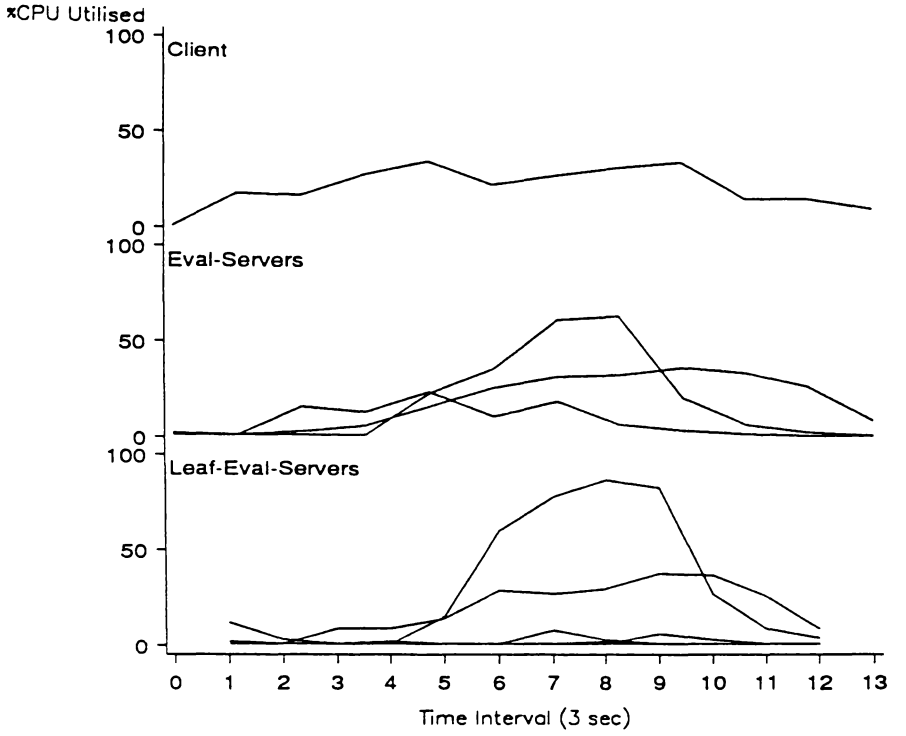

Figure 5.9 *Load at 3 sec intervals, tree configuration. Problem W1: two terms in gcd*

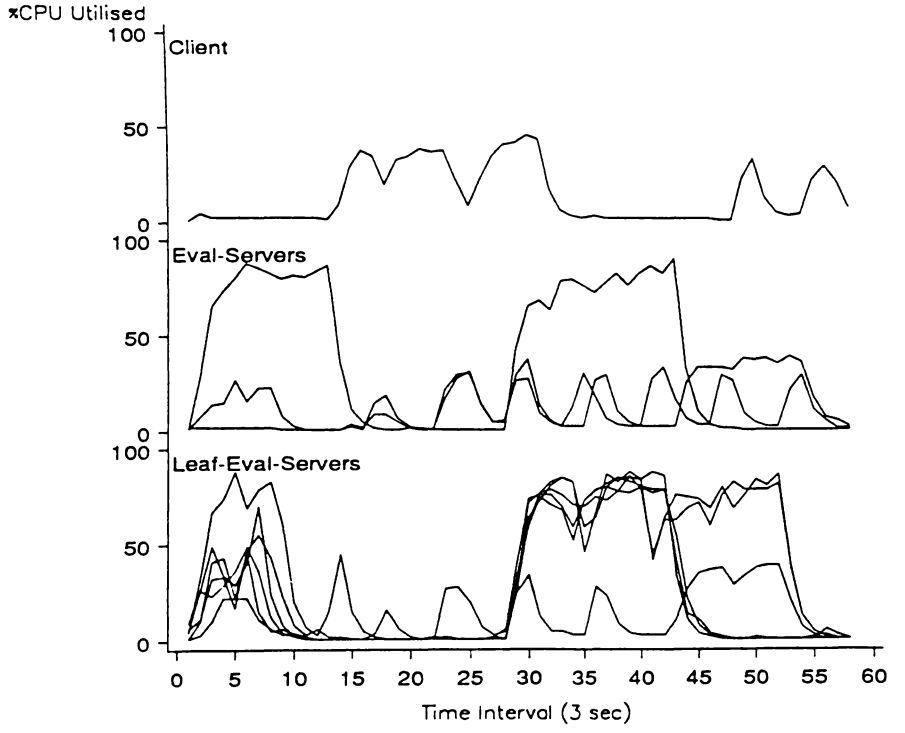

Figure 5.10 *Load at 3 sec intervals, tree configuration. Problem W5: five terms in gcd*

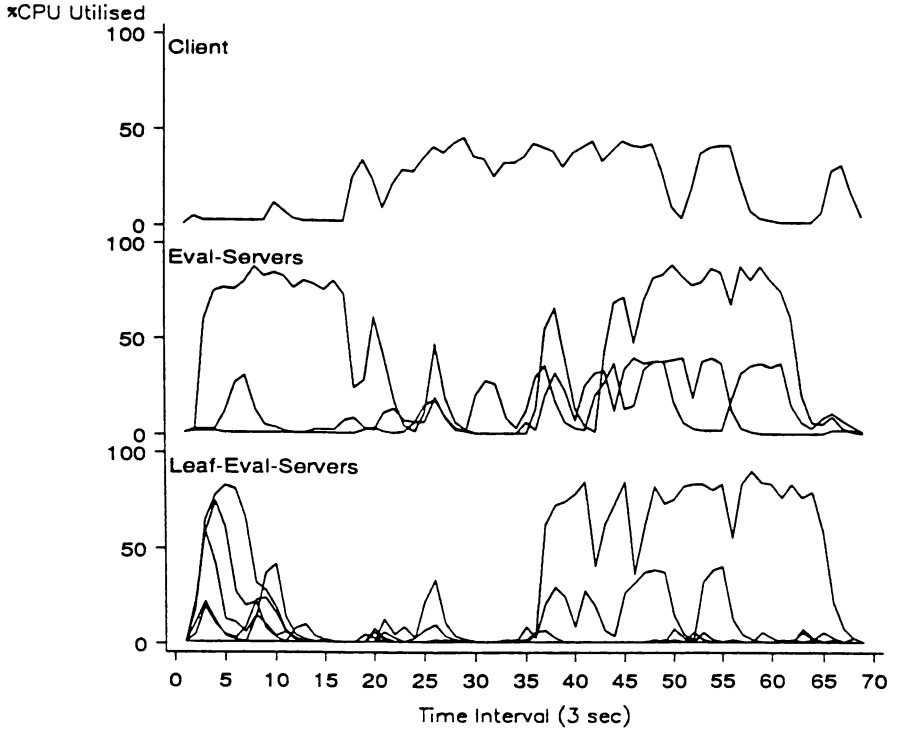

Figure 5.11 *Load at 3 sec intervals, tree configuration. Problem Z6: two terms in gcd*

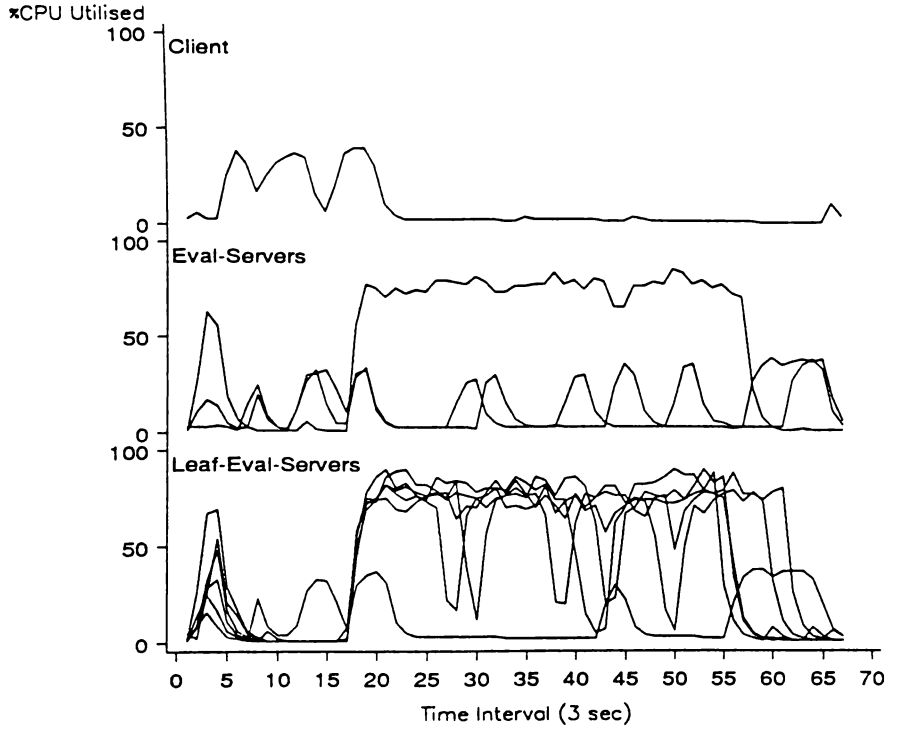

Figure 5.12 *Load at 3 sec intervals, tree configuration. Problem M1: ten terms in gcd*

Chapter 6

Summary

And Concluding Remarks

The research presented in this thesis began with an investigation into the requirements of a parallel computer algebra system with respect to hardware (a parallel computer) and software (a parallel functional language). After matching the hardware requirements as best as possible with available hardware, an array of transputers was chosen as the parallel computer. In the absence of a transputer programming language suitable for computer algebra, the Tlisp version of lisp was developed.

The Tlisp language together with the transputer array form the Tlisp system – a network of asynchronous, message-passing processors programmable in a dialect of lisp which has features to explicitly effect parallelism. The Tlisp system is based on the eval-server model of parallelism where each processor executes a Tlisp process and certain processors (eval-servers) act as evaluators for the other processors (clients). The processors are configured in a tree so that each eval-server has only one client.

The concepts behind Tlisp are simple and this simplicity has been retained at all stages of its development. Although constraints on the design were necessarily imposed by "keeping it simple" the major goal of having a reliable, working system within a reasonable period of time was achieved. Further, Tlisp proved itself to be a useful system and relatively easy to understand from a user's point of view. This was illustrated in Chapter 2 with a presentation of some parallel versions of a Fibonacci number generator.

Tlisp was designed for experimenting with parallel algebraic algorithms, in particular with an algorithm for finding the greatest common divisor of arbitrary, multivariate polynomials. To this end a parallel gcd algorithm was designed. After surveying existing sequential approaches to solving the gcd problem, Zippel's sparse modular gcd algorithm was selected as the best candidate for parallelisation. Detailed examination was made of each step of the sequential algorithm and coarse-grained parallelism extracted. The resulting parallel algorithm was implemented for the Tlisp system.

Extensive analysis of the performance of the parallel polynomial gcd algorithm with the Tlisp system was carried out. Most evident was the observation that the

algorithm contains considerable parallelism several levels deep and that when the Tlisp configuration is tailored to the particular gcd problem being solved, good speedup can be achieved.

The hierarchical scheduling scheme of Tlisp would appear to be its greatest obstacle to utilising all the parallelism present in the gcd algorithm. The overheads of inter-processor communication also present a barrier in that only coarse-grained parallelism is able to be exploited.

The practical experience with a real parallel system (Tlisp) and an important parallel algebraic algorithm (polynomial gcd) and its subsidiary algorithms support the results of previous research into parallel computer algebra systems (much of which has been largely theoretical). In particular, the work of Watt (1985a) which postulated that each problem in computer algebra has some inherent parallelism and coding their solutions for high-level parallelism is not overly difficult, is further strengthened by the Tlisp coding of the algorithm PARALLEL-GCD and its subsidiary algorithms.

The work of Buchberger, which gave as a requirement for a parallel symbolic computation machine that each processor have ready access to its neighbours' memories, (Buchberger, 1985), is to some extent justified by the overheads of message passing observed in Tlisp. Although these overheads dictate the coarse granularity of the exploited parallelism in Tlisp, they could be low enough to make a message-passing computer algebra system a viable possibility. A message-passing system, with a better strategy than Tlisp's for transferring lisp s-expressions from one process to another (as discussed in section 2.5.6), could be quite suitable for a parallel computer algebra system, even without Buchberger's memory requirement.

Future work on parallel computer algebra needs to examine a wider range of algebraic problems to gain a more accurate indication of what grain size a parallel computer algebra system should target. Experimentation with shared memory machines, both full shared memory as suggested by Ponder (1988b) and partial shared memory as suggested by Buchberger (1985), would be useful as a comparison to the work presented here and in (Watt, 1985a) based on message-passing systems.

As a contribution to the research area of parallel computer algebra the research reported in this thesis has produced a useful, experimental parallel lisp system which runs on an inexpensive multiprocessor system. Parallelism is explicitly extracted from applications making code relatively portable. A parallel algorithm has been developed for the important computer algebra problem of finding multivariate polynomial gcds. The algorithm has been implemented and its performance analysed to prove its worth.

Although indications to date suggest that asynchronous, message-passing multiprocessor systems with medium- to coarse-grained parallelism do have much potential for parallel computer algebra systems, there is still much work to be done and the field is very much open to influences of new research and technology.

Appendix A

Code for the

Communications Coordinator Process

This appendix lists the major pieces of C code for the multithreaded communications coordinator process (CCP) as described in section 2.5.4 and Figure 2.3. The organisation is as follows : Thread A (*ccp_in*); Thread B (*remote_server*) comprised of *remote_in*, *remote_out*, *bcast* and *ack_evalserv*; Thread C (*request_server*) comprised of *receive_form*, *request* and *search* and Thread D (*ccp_out*).

Subsidiary functions and structured data types which are used but not listed here have self-explanatory names.

Thread A (*ccp_in*)

```
/* Thread to receive incoming messages from a client and pass them
   to the appropriate thread according to the code number in the first
   byte of the character string */

void ccp_in (from_tlisp, to_remote, to_request, to_search, to_bcast)
soft_link *from_tlisp, *to_remote, *to_request, *to_search, *to_bcast;

{
        char *form = (char *)malloc(STRMAX + 2) ;
        int key ;

        forever {
                internal_read_msg(from_tlisp, form, &key) ;
                switch (*form)
                {
                        case '0' : /* message sent by function remote */
                                   internal_write_msg(to_remote, form+1, key) ;
                                   break ;
                        case '1' : /* message sent by function request */
                                   internal_write_word(to_request, key) ;
                                   break ;
                        case '2' : /* message sent by function broadcast */
                                   internal_write_msg(to_bcast, form+1, key) ;
                                   break ;
                        case '3' : /* message sent by function evalp */
                                   internal_write_word(to_search, key) ;
                                   break ;
                        default  : perror("bad msg code to ccp") ;
                }
        }
}
```

Thread B (*remote_server*)

Thread B is responsible for handling client-to-eval-server messages. The messages are stored on a FIFO queue. Thread B consists of four threads - *remote_in, remote_out, bcast* and *ack_evalserv*. Two semaphores are used within Thread B - *queue_free* controls accesses to the queue and *job_array* controls accesses to the array *job_on_evalserv[]* which stores the key numbers of jobs currently being evaluated and indicates which eval-servers are currently waiting for input.

```
/* Thread to receive forms (with their reference keys) for remote
   evaluation from the client (via ccp_in) and put the forms
   on the queue */

void remote_in (from_client)
soft_link *from_client ;

{
      int key ;
      char *form = (char *)malloc(STRMAX+1) ;

      forever
      {  /* input form from eval server */
            internal_read_msg(from_client, form, &key) ;

            /* enter form on queue */
            Wait(&queue_free) ;
              push(form, key) ;
            Signal(&queue_free) ;
       }
}
```

```
/* Thread to send forms for remote evaluation from the top of the
   queue to waiting eval-servers */

void remote_out(to_evalserv, nr_evalservs)
hard_link *to_evalserv ;
int nr_evalservs;

{       int i, form_sent ;

        forever
        {       form_sent = 0 ;
                Wait(&queue_free) ;
                if (q_top)
                {       Signal(&queue_free) ;
                        Wait(&job_array) ;
                        /* find first available eval-server and send message */
                        for (i = 0; i < nr_evalservs; i++)
                        {       if (!(job_on_evalserv[i]))
                                {       job_on_evalserv[i] = q_top->key ;
                                        Signal(&job_array) ;
                                        external_write_msg(to_evalserv[i], q_top->form) ;
                                        Wait(&queue_free) ;
                                          pop() ;
                                        Signal(&queue_free) ;
                                        form_sent++ ;
                                        break ;
                                }
                        }
                if (!form_sent) Signal(&job_array) ;
                }
        else Signal(&queue_free) ;
        }
}
```

```
/* Thread to receive forms from the client (via ccp_in) to be broadcast
   to all the eval-servers.
   On receipt of a form, the thread waits until the queue is empty,
   then sends the form to each eval-server. */

void bcast(from_client, to_client, to_evalserv, nr_evalservs)
soft_link *from_client, *to_client ;
int *to_eval, nr_evalservs ;
{
        char *form = (char *)malloc(STRMAX+1) ;
        int i, ack=1, key ;

        forever
        {       internal_read_msg(from_client, form, &key) ;
                /* clear the queue */
                Wait(&queue_free) ;
                while(q_top)
                {       Signal(&queue_free) ;
                        Wait(&queue_free) ;
                }

                /* make broadcast */
                for  (i = 0; i < nr_evalservs; i++)
                external_write_msg(to_evalserv[i], form) ;
                Signal(&queue_free) ;
                internal_write_word(to_client, ack);
        }
 }
```

Thread C (*request_server*)

Thread C controls storage of s-expressions in and their retrieval from a store. It consists of three threads - *receive-form, request* and *search.* A semaphore, *store_free*, is used to control accesses to the store.

```
/* Thread to receive forms from an eval-server.
   All forms are placed in the store, each with its key
   which is retrieved from the job_on_evalserv array.
   One of these threads for each eval-server */

void receive_form(STORE, in_port, evalserv_nr)
store *STORE ;
hard_link *in_port ;
int evalserv_nr ;

{       int key ;
        char *form = (char *)malloc(STRMAX+1) ;

        forever
        {       external_read_msg(in_port, form) ;
                /* get key from job array */
                key = job_on_evalserv[evalserv_nr] ;

                Wait(&store_free) ;
                store_entry(&STORE, key, form) ;
                Signal(&store_free) ;

                if (key)
                {       /* update active job array  */
                        Wait(&job_array) ;
                                job_on_evalserv[evalserv_nr] = 0 ;
                        Signal(&job_array) ;
                }
        }
}
```

```
/* Thread to receive requests (in the form of a reference key) from the
   client (via ccp_in) for remotely evaluated forms.
   On receipt of such a request the store is searched repeatedly until
   the required form is found.  The form is sent back to the client
   (via ccp_out). */

void request(STORE, from_client, to_client)
store *STORE ;
soft_link *from_client, *to_client ;

{       int key ;
        char *form ;

        forever
        {       internal_read_word(from_client, &key) ;
                spin:
                Wait(&store_free) ;
                if (retrieve_entry(STORE, key, &form))
                {  Signal(&store_free) ;
                        internal_write_msg(to_client, form, key) ;
                        free(form) ;
                }
                else
                {  Signal(&store_free) ;
                        goto spin ;
                }
        }
}
```

```
/* Thread to receive requests (in the form of a reference key) from the
   client (via ccp_in) for the status of forms sent for remote evaluation.
   On receipt of such a request the store is searched once.
   If the form is found it is sent back (via ccp_out) with a "1" to
   indicate success, otherwise an empty string is sent back with a "0"
   to indicate failure. */

void search (STORE, from_client, to_client)
store *STORE ;
soft_link *from_client, *to_client ;

{       int key ;
        char *form ;

        forever
        {       internal_read_word(from_client, &key) ;
                Wait(&store_free) ;
                if (retrieve_entry(STORE, key, &form))
                {       Signal(&store_free) ;
                        internal_write_msg(to_client, form, 1) ;
                        free(form) ;
                }
                else
                {       internal_write_msg(to_client, "", 0) ;
                        Signal(&store_free) ;
                }
        }
}
```

Thread D (*ccp_out*)

```
/* This thread waits for messages from the other threads to be passed back
   to the client.
   It forks three simple procedures which wait for messages from a
   soft-link and send them back to the client. */

int ccp_out (to_tlisp, from_reqserv, from_search, from_bcast)
soft_link *to_tlisp, *from_reqserv, *from_search, *from_bcast ;
{
        Semaphore eternity ;

        InitSemaphore(&eternity, 0) ;

        if (!Fork (1000, wait_for_msg, 2*sizeof(soft_link *), from_reqserv, to_tlisp))
        return 20 ;

        if (!Fork (1000, wait_for_msg, 2*sizeof(soft_link *), from_search, to_tlisp))
        return 20 ;

        if (!Fork (1000, wait_for_word, 2*sizeof(soft_link *), from_bcast, to_tlisp))
        return 20 ;

        Wait(&eternity) ;
}
```

Appendix B

Code for the

Parallel GCD Algorithms

This appendix contains the major pieces of code for an implementation of the parallel gcd algorithms presented in this thesis. Not included are the polynomial arithmetic and utility functions. The programming language used is Tlisp, a parallel version of Xlisp, as discussed in Chapter 2.

The gcd functions were adapted from those in a sequential polynomial manipulation package written by Terry Robb in Franz Lisp. The original implementation was of Zippel's 1979 version of the algorithm. Once modified for use with Tlisp the implementation was updated to include the use of transposed Vandermonde matrices and then parallelised.

The polynomial data representation chosen for the polynomial arithmetic package is a sparse distributive one, the most suitable for Zippel's sparse interpolation algorithm (see (Stoutemyer, 1984) for a discussion on other types of representation).

A polynomial is represented as a *%poly* which is comprised of a lisp list of a polynomial label, a list of symbols (variables) and a list containing an ordered list of the exponent vectors (one for each term) and coefficient vector *i.e.*

$$(\%poly\ (X_1 \cdots X_n)\ ((\overline{e}_1 \cdots \overline{e}_t)\ (c_1 \cdots c_t)))$$

where each $\overline{e}_i$ is an exponent vector, $\overline{e}_i = (e_{i1} \cdots e_{in})$. Terms with zero coefficient are not listed (thus the representation is sparse).

For example the polynomial

$$3X_1^2X_2 + 2X_1X_2^2X_3 - 4X_3^4 + 6$$

is represented as

$$(\%poly\ (X_1X_2X_3)\ (((2\ 1\ 0)\ (1\ 2\ 1)\ (0\ 0\ 4)\ (0\ 0\ 0))\ (3\ 2\ -4\ 6)))$$

During most calculations involving polynomials much of the information in the *%poly* representation is not explicitly required, for example, when adding two polynomials with the same variables in the same order it is not necessary to continue passing the variable vectors between functions. Two sub data types are used: *poly1* for univariate polynomials and *poly3* for multivariate polynomials. The univariate

polynomial $4X_1^3 + 2X_1 - 3$, for example, has *poly1* representation as ((3 1 0) (4 2 –3)), a list of exponents followed by a list of coefficients. The multivariate polynomial given above has *poly3* representation as (((2 1 0) (1 2 1) (0 0 4) (0 0 0)) (3 2 –4 6)) which is simply the third part of the *%poly* representation.

The two sub data types are used throughout the polynomial package except for at the top-level. Many basic functions are named according to which of the data types are involved, for example *%poly30+* adds a *poly3* to a *poly1*, *%poly3modpevalto3* evaluates a *poly3* in modulo p at a given point returning a *poly3*.

Less general functions are often named according to which algorithm they are associated with, for example the main functions for the implementation of the algorithm for solving a transposed Vandermonde system are prefixed with *vdm.*

Listed on the following pages are the major functions for the implementation of the parallel version of Zippel's sparse gcd algorithm as discussed in section 4.7 of Chapter 4. In function definitions the function name is emboldened for clarity. Calls to *remote* and *request* are also emboldened to highlight where parallelism is achieved.

```
;; given a list of exponent vectors and a coefficient vector for two polynomials
;; (implicitly with the same variables) find their gcd

(defun zippel-gcd (Poly1 Poly2)
  (prog (coef-bound p gcdmodp gcd cont1 cont2 cont F_1 F_2 f1 f2 norm point alist
        degrees dnorm Xpowlist1 Xcoflist1 Xpowlist2 Xcoflist2 %Xveczero
        %initial-prime %baserand)

      ;; Step PG1 - Initialise

      ;; check if one polynomial is an integer
      (if (equal (%poly3deg Poly1) 1)
         (return (%poly1to3 (%poly11gcd (%poly3to1 Poly1) (%poly3to1 Poly2)))))

      ;; calculate and remove contents from polys
      (setq %Xveczero (mapcar #'(lambda (x) 0) (cdaar Poly1)))
      (desetq (Xpowlist1 Xcoflist1) (%poly3toX Poly1))
      (desetq (Xpowlist2 Xcoflist2) (%poly3toX Poly2))
      ;; create a unique variable name for the remote call (since function is recursive)
      (setq cont1 (gensym))
      (remote cont1 t '(%poly3S-parallel-cont Xcoflist1))     ;; remotely get the X content of Poly1
      (setq cont2 (%poly3S-parallel-cont Xcoflist2)           ;; get the X content of Poly2
            cont (%polyXto3 (list (list 0)
                                  (list (%poly33gcd (setq cont1 (request cont1)) cont2))) %Xveczero)
            Xcoflist1 (mapcar #'(lambda (Xcof) (%poly33/ Xcof cont1)) Xcoflist1)
            Xcoflist2 (mapcar #'(lambda (Xcof) (%poly33/ Xcof cont2)) Xcoflist2)
            ;; remove the X content from Poly1
            F_1 (%polyXto3 (list Xpowlist1 Xcoflist1) %Xveczero)
            ;; remove the X content from Poly2
            F_2 (%polyXto3 (list Xpowlist2 Xcoflist2) %Xveczero))
      (if (or (fixp F_1) (fixp F_2)) (return cont))

      ;; normalise the polys
      (setq f1 (car Xcoflist1)              ;; the leadcoef (in X) of F_1
            f2 (car Xcoflist2)              ;; the leadcoef (in X) of F_2
            norm (%poly33gcd f1 f2))
```

```
;; Step PG2 - choose prime, compute unvariate skeleton

;; initialise modulus and base for random numbers
;; (%initial-prime) < (biggest integer) = 2^30
;; %baserand should be a prime <= %initial-prime
(setq %initial-prime (car (%bigprimelist))
      %baserand %initial-prime)

;; calculate maximum degrees in the gcd of each variable
;; (could call zippel-degrees twice to make probability of failure smaller)
(setq point (mapcar #'(lambda (x) (random %baserand)) (caar F_1))) ;; choose point at random
(setq degrees (zippel-degrees F_1 F_2 point %baserand))
(if (equal (car degrees) '(0)) (return cont)) ;; case independent of first variable
(setq dnorm 1)

(if (not (fixp norm))                    ;; if norm~=1, take max of each in (cdr degrees)
   (setq degrees (cons (car degrees)   ;; and add to powers of norm
                       (mapcar #'+ (mapcar #'(lambda (degs) (apply #'max degs)) (cdr degrees))
                               (%apply-mapcar #'max #'max (car norm))))
         dnorm (%maxabs (cadr norm)))) ;; dnorm is used to compensate normalisation in
                                       ;; computation of coefficient bound

;; Steps PG3 and PG4 - lift X1, ..., Xv

;; perform the sparse interpolation
(setq alist (cdr point)    ;; alist has v random numbers.
      coef-bound (* 2 dnorm (max (%maxabs (cadr F_1)) (%maxabs (cadr F_2))))
      p %initial-prime
      gcd (zippel-sparsegcd degrees F_1 F_2 norm alist p %baserand))

;; Step PG5 - Check results

;; generate a new prime and repeat sparse interpolation if necessary
(do ((modp (%nextprime p) (%nextprime modp)))
   ((> p coef-bound))  ;; when p > coef-bound gcdmodp has large enough coefficients
   (setq gcdmodp
         (zippel-sparsegcd degrees F_1 F_2 norm alist modp %baserand)
         gcd (%poly33cra (%invmod p modp) p modp gcd gcdmodp)
         p (* p modp)))

;; convert integer coefficients to balanced representation mod p
;; and return gcd with content restored
(setq gcd (%poly3centre gcd p))
(return (%poly33* gcd cont))))
```

```
;; Given a list of polynomials calculate their gcd as the content of the
;; polynomial whose coefficients of the leading variable make up the list

(defun %poly3S-parallel-cont (polys)
  (prog (loop arglist)
    (setq arglist polys)
    (cond ((member 1 polys) (return 1)) ;; check for immediately trivial gcd
          ((= *nr_evalservs* 0)
           (return (%poly3scont polys))))  ;; if no eval-servers use serial version
    loop
      ;; create remote gcd jobs from each pair of polys
      (cond ((null (cdr arglist)) (return (car arglist)))  ;; check more than one poly in list
            ((null (cddr arglist))                         ;; check more than two polys in list
             (return (%poly33gcd (car arglist) (cadr arglist))))
            ;; arglist = (p1, p2, ... ) replace with (gcd(p1, p2), gcd(p3, p4), ...)
            (t (setq arglist
                 (do ((ret nil
                           (cons (cond ((null (cdr args)) (set sym (car args)) sym)
                                       (t (remote sym t '(%poly33gcd (car args) (cadr args)))
                                          sym))
                                 ret))
                      (sym (gensym) (gensym))
                      (args arglist (cddr args)))
                     ((null args) (mapcar #'request ret))))
             (if (member 1 arglist) (return 1))  ;; check for trivial gcd
             (go loop)))))
```

```
;; The following returns the powers of each variable that will appear in the gcd of F1 and F2.
;; I.e. zippel-degrees returns a degsi, where
;;       degsi = (degs0 degs1 ... degsv) and
;;       degs[i] = set of powers for the i'th variable Xi, i=0,1...v.
;; The Prob of failure of zippel-degrees is (v-1)*(t-1)*D)/B,
;; where B=%baserand is big (~ 10^9), and
;;       v is number of variables in the answer,
;;      t is number of terms in the answer, and
;;       D is total degree of the answer.
;; eg x^7*y^8*z^3 + y^5 + x^2*y + 1 has v=3, t=4, D=7+8+3=18.
;; point =  (a0, ..., av), a list of random integers less than %baserand
;; Works by calculating, for each variable Xi the gcd of F1 and F2 at
;;     (a0, ..., ai-1, Xi, ai+1, ..., av)

(defun zippel-degrees (F1 F2 point p)
  (prog (degrees pows1 pows2 cofs1 cofs2 swop1 swop2 swap1 swap2)
        (desetq (pows1 cofs1) F1)
        (desetq (pows2 cofs2) F2)
        (setq swop1 (%Transpose pows1))     ;; F1 = (list (%Transpose swop1) cofs1)
        (setq swop2 (%Transpose pows2))     ;; F2 = (list (%Transpose swop2) cofs2)
        (setq swap1 (%pointeval point swop1 p))
        (setq swap2 (%pointeval point swop2 p))
       (return
         (do ((ret nil (cons sym ret))
              (sym (gensym) (gensym))
              (swop1s swop1 (cdr swop1s))
              (swop2s swop2 (cdr swop2s))
              (swap1s swap1 (cdr swap1s))
              (swap2s swap2 (cdr swap2s)))
             ((null swop1s) (mapcar #'request (reverse ret)))
             ;; perform the univariate gcd computations concurrently
             (remote sym t '(zippel-degrees-gcd
                                        (car swop1s)
                                        (car swop2s)
                                        (%wapswop cofs1 ',(car swap1s) swap1 p)
                                        (%wapswop cofs2 ',(car swap2s) swap2 p)
                                        p))))))

;; helper function for zippel-degrees
(defun zippel-degrees-gcd (ypows1 ypows2 cofs1 cofs2 p)
  (%poly1pows (%poly11modpgcd (%poly1modpcanon ypows1 cofs1 p)
                              (%poly1modpcanon ypows2 cofs2 p) p)))
```

```
;; Sparse interpolation of a gcd given its univariate skeleton
;; degrees = (degs0 degs1 ... degsv)
;;       where degs0 = list of expected powers for the 0'th X variable
;;      and degs[i] = list of expected powers for the i'th variable Xi.
;; alist = (a1 a2 ... av)    - a random starting point
;; F1, F2 = primitive Poly's (wrt the zero'th variable X) in v variables, plus X.
;; norm = gcd(lc(F1), lc(F2))
;; modp = integer prime base

(defun zippel-sparsegcd (degrees F1 F2 norm a_list modp %baserand)
  (prog (tmp v termlist Xlist clist clists cslist d dvec degsi dmax degsX
        restart expdeg aiv Fxa1 Fxa2 norma rjs GCDmodp  %rjs %qjs)

      (setq v (length a_list))     ;; v, nr vbles in problem, should be 2 or bigger

      (setq degsX (car degrees) degsi (cdr degrees)
          ;; pick out maximum degree of each vble above X0
          dvec (if (fixp (car degsi)) degsi (mapcar #'(lambda (degs) (apply #'max degs))
                                                    degsi))
          dmax (apply #'max dvec))
      restart  ;; here when prime turns out to be "unlucky"
      (setq tmp (%dense-init (car a_list) dmax modp %baserand) ;; %rjs - list of v-1 random vbles
          %rjs (car tmp) %qjs (cadr tmp)                       ;; %qjs - precomputed for CRAII
          Xlist degsX                    ;; expected skeleton for gcd
          expdeg (apply #'max Xlist))    ;; expected degree of any vble in gcd

      ;;--------------------------------------

      ;; Step PG3
      ;; lift X1 using dense interpolation
      (setq d (car dvec) dvec (cdr dvec) aiv (cdr a_list)
          Fxa1  (%poly3modpevalto3 F1   aiv modp)
          Fxa2  (%poly3modpevalto3 F2   aiv modp)
          norma (%poly3modpevalto3 norm aiv modp))
      (setq clists  ;; list of values of coefs ((c1(r0), ..., cT(r0)), (c1(r1), ..., cT(r1)), ... )
          (do ((j 0 (1+ j))
              (rjs %rjs (cdr rjs))
              (sym (gensym) (gensym))
              (Fxs nil))
             ((> j d) (mapcar #'(lambda (Fx) (zippel-clist Xlist Fx))
                              (mapcar #'request (reverse Fxs))))
             (setq Fxs  (cons sym Fxs))
             ;; create remote job for each univariate gcd computation (last one local)
             (remote sym (/= j d)
                     '(zippel-Fgcd Fxa1 Fxa2 norma (list (car rjs)) modp))))

      ;; check for bad homomorphisms
      (cond ((zippel-badhomomorhism clists)
            ;; unlucky prime - start over
            (setq modp (%nextprime modp)) (goto restart)))
```

```
(setq ;; cslist is ((c1(r0), c1(r1), ..., c1(rd)), (c2(r0), c2(r1), ..., c2(rd)), ... )
      cslist (%Transpose clists)
      ;; for each ci use CRAII to interpolate the d+1 values against %rjs to get ci(X1)
      termlist (mapcar #'(lambda (cs) (%poly1to3x (zippel-dense cs modp %rjs %qjs))) cslist))

;;-------------------------------------

;; Step PG4
;; taking each term separately lift X2, X3, ..., Xv using sparse interpolation
(if (> *nr_evalservs* 0)
    ;; do parallel version
    (setq termlist
      (do ((terms termlist (cdr terms))  ;; iterate on terms
           (exps xlist (cdr exps))
           (sym (gensym) (gensym))
           (ret nil ))
          ((null terms) (reverse ret))
          (setq ret (cons sym ret))
          ;; create a remote job to lift each term (last one local)
          (remote sym (cdr terms)
                  '(zippel-lift-term v (car terms) (car exps) dvec aiv norm F1 F2
                                     %rjs %qjs modp %baserand))))

    ; else do serial version (less repeated computation than parallel version)
    (do ((k 2 (1+ k)))   ;; iterate on variables
        ((> k v) )
        (setq d (car dvec) dvec (cdr dvec) aiv (cdr aiv))
        (cond ((nilorzerop d)                          ;; ie gcd is indep of the k'th variable
               (setq termlist (%termuponevar termlist)) ;; tack on a 0 to each skeleton
               (go nextvariable)))
       (setq Fxa1  (%poly3modpevalto3 F1   aiv modp)
             Fxa2  (%poly3modpevalto3 F2   aiv modp)
             norma (%poly3modpevalto3 norm aiv modp))

        (setq termlist  ;; lift next variable in each term
          (do ((ret nil (cons (zippel_lift_nxt_vble
                                (car skeletonlist) Fxa1 Fxa2 norma (car x) (1- k) d
                                %rjs %qjs modp %baserand)
                              ret))
               (x xlist (cdr x))
               (skeletonlist (mapcar #'car termlist) (cdr skeletonlist)))
              ((null skeletonlist) (reverse ret))))
    nextvariable
    ))

;; make lifted terms primitive (to compensate normalisation)
(setq GCDmodp (zippel-XPoly Xlist (%termlistprim (mapcar #'request termlist))))
(return GCDmodp)))
```

```
;; lift X2,...,Xv in the polynomial coefficient of X0^exp
;; term is the skeletal polynomial

(defun zippel-lift-term (v term exp dvec aiv norm F1 F2 %rjs %qjs modp %baserand)
 (do ((k 2 (1+ k))   ;; iterate on variables
      (Fax1 nil) (Fax2 nil) (norma nil) (d nil))
     ((> k v) term)
     (setq d (car dvec) dvec (cdr dvec) aiv (cdr aiv))
     (cond ((nilorzerop d)                                  ;; ie gcd is indep of the k'th variable
            (setq term (car (%termuponevar (list term))))) ;; tack on a 0 to the skeleton
            (go nextvariable)))
     (setq Fxa1  (%poly3modpevalto3 F1   aiv modp)
           Fxa2  (%poly3modpevalto3 F2   aiv modp)
           norma (%poly3modpevalto3 norm aiv modp))
     (setq term
           (zippel_lift_nxt_vble (car term) Fxa1 Fxa2 norma exp (1- k) d %rjs %qjs modp %baserand))
     nextvariable))


;; Using Fxa1, Fxa2 and norma to generate the goal polynomial (the multivariate coefficient
;; of X0^n) at various points (y1, y2, ..., yk-1, xks) the variable Xk is lifted.
;; S is the skeleton of the goal polynomial
;; d is the maximum degree of Xk in the gcd

(defun zippel_lift_nxt_vble (S Fxa1 Fxa2 norma n k d xks qjs modp %baserand)
  (prog (y_vec v_vec a_vec T_)
    (setq y_vec (vdm_pick_y k S %baserand modp)   ;; choose k-1 suitable integers y
          v_vec (cadr y_vec)                      ;; v_vec is those powers of y which
          y_vec (car y_vec))                      ;; which make up transposed vdm matrix

    (setq sjs     ;; set of d+1 function heights on each of skeletal terms
          (do ((j 0 (1+ j))
               (xkj xks (cdr xkj))
               (sym (gensym) (gensym))
               (sjs nil (cons sym sjs)))
              ((> j d) (mapcar #'request (reverse sjs)))

              ;; create d+1 jobs to set up and solve a system of equations
              (remote sym (/= j d)
                      '(vdm_derived_poly n Fxa1 Fxa2 norma y_vec v_vec S ',(car xkj) k
                                         modp %baserand))))

    (return (zippel-next sjs S modp xks qjs))))  ;; use CRAII to fit a polynomial in the next variable
                                                 ;; through each set of function heights in sjs
                                                 ;; (sets may be interpolated concurrently)
```

```
;; Setup and solve a transposed system of Vandermonde equations
;; Given  v_vec, the powers of which make up A (the transposed Vandermonde matrix),
;; this function computes a_vec, a list of values of the goal polynomial (y_vec^i, xkj),
;; and solves Ax = a_vec for x.
;; The goal polynomial is the coefficient of X0^n in the gcd

(defun vdm_derived_poly (n Fxa1 Fxa2 norma y_vec v_vec skeleton xkj k modp baserand)
  (prog (a_vec T_)
    (setq T_ (length skeleton))  ;; T_ is the max nr. of terms in goal polynomial

    (setq a_vec
        (do ((ret nil
                (cons (%coef-nterm
                        n
                        (zippel-Fgcd Fxa1
                                Fxa2
                                norma
                                (append (mapcar #'(lambda (x) (%modp~^ x i modp))
                                                y_vec)
                                        (list xkj))
                                modp))
                      ret))
             (i 0 (1+ i)))
            ((= i T_) (reverse ret))))
    (return (vdm_solve v_vec a_vec modp))))   ;; solve the system and return solution
```

Appendix C

Data for Testing the Parallel GCD Algorithm

This appendix consists of four sets of test data used in Chapter 5 for testing the parallel gcd algorithm. Each set consists of several triples of polynomials (D_i, F_i, G_i) where D_i is the gcd and F_i and G_i are cofactors.

Set W : given by Wang (1980)

W1

$$D_1 = z^4+(y^4+x^4)z^3+a^4$$

$$F_1 = (z^4-2z+y^4+x^4+a^4)$$

$$G_1 = (z^4-y^4+2y+x^4+a^4)$$

W2

$$D_2 = z^5+(y^5+x^5)z^3+(c^5+b^5)z^2+a^5$$

$$F_2 = (z^5-2z+y^5+x^5+c^5+b^5+a^5)$$

$$G_2 = (z^5-y^5+2y+x^5+c^5+b^5+a^5)$$

W3

$$D_3 = z^5+(y^5+x^5)z^3+(p^5+c^5+b^5)z^2+a^5$$

$$F_3 = (z^5-2z+y^5+x^5+p^5+c^5+b^5+a^5)$$

$$G_3 = (z^5-y^5+2y+x^5+p^5+c^5+b^5+a^5)$$

W4

$$D_4 = x^3y^3z^3+x^2y^2z^2+xyz+1$$

$$F_4 = x^3y^2z^3+x^2y^3z^2+4xyz+2$$

$$G_4 = x^3y^2z^3+x^2y^4z^2+x^2y^6z+3$$

W5

$$D_5 = u^4x^4y^4z^4 + u^3x^3y^3z^3 + u^2x^2y^2z^2 + uxyz + 1$$

$$F_5 = u^3x^3y^2z^3 + x^2y^3z^2 + 4xyz + u^2y^4 + 2$$

$$G_5 = u^2x^3y^2z^3 + ux^2y^4z^2 + x^2y^6z + u^3x^3 + 3$$

W6

$$D_6 = u^2xy(y^2 + x^2)z^4 + x^2y^2z^2 + x^3y^3z + x^4y^4 + u^3 + 1$$

$$F_6 = (x - y^2)z^3 + (y - z)^2 + y + x + u$$

$$G_6 = (z + y)^2 + (y^2 + x)z^3 - y - x - u$$

W7

$$D_7 = u^3(x^2 - y)z^2 + (u - 3u^2x)yz - u^4xy + 3$$

$$F_7 = (y^2 + x)z^2 + u^5(xy + x^2)z - y + 5$$

$$G_7 = (y^2 - x)z^2 + u^5(xy - x^2)z + y + 9$$

W8

$$D_8 = u^3v^3(x^2 - y)z^2 + v(u - 3u^2x)yz - u^4xy + 3$$

$$F_8 = v^3(y^2 + x)z^2 + u^5(xy + x^2)z - y + 5$$

$$G_8 = v^3(y^2 - x)z^2 + u^5(xy - x^2)z + y + 9$$

W9

$$A = z - xy + 1$$

$$B = z - y + 3x$$

$$C = xyz + z + y + x$$

$$F_9 = AB^2C^2$$

$$G_9 = A^2BC$$

W10

$$F_{10} = A^2B^4C^2$$

$$G_{10} = A^4B^2C$$

Set Z: given by Zippel (1979)

Z1

$$D_1 = x_1^2 + x_1 + 3$$

$$F_1 = 2x_1^2 + 2x_1 + 1$$

$$G_1 = x_1^2 + 2x_1 + 2$$

Z2

$$D_2 = 2x_1^2 x_2^2 + x_1 x_2 + 2x_1$$

$$F_2 = x_2^2 + 2x_1^2 x_2 + x_1^2 + 1$$

$$G_2 = x_1^2 x_2^2 + x_1^2 x_2 + x_1 x_2 + x_1^2 + x_1$$

Z3

$$D_3 = x_2^2 x_3^2 + x_2^2 x_3 + 2x_1^2 x_2 x_3 + x_1 x_3$$

$$F_3 = x_3^2 + x_2^2 x_3 + x_1^2 x_2 x_3 + x_1 x_3 + x_1^2 x_2^2$$

$$G_3 = x_2 x_3 + 2x_1 x_3 + x_3 + x_1$$

Z4

$$D_4 = x_1^2 x_4^2 + x_2^2 x_3 x_4 + x_1^2 x_2 x_4 + x_2 x_4 + x_1^2 x_2 x_3$$

$$F_4 = x_1 x_2 x_3^2 x_4^2 + x_1 x_3^2 x_4^2 + x_1 x_4^2 + x_4^2 + x_1 x_3 x_4$$

$$G_4 = x_1 x_3^2 x_4^2 + x_3^2 x_4^2 + x_4^2 + x_1 x_2^2 x_3 x_4 + x_1 x_2^2$$

Z5

$$D_5 = x_1^3 x_2^2 x_3^2 x_4 x_5^2 + x_1 x_2^2 x_5^2 + x_1^3 x_3 x_4^2 x_5 + x_1^3 x_2 x_3^2 x_4 x_5 + x_1^2 x_2 x_3^2 x_4^2$$

$$F_5 = x_1 x_2^2 x_5^2 + x_1 x_2 x_3^2 x_4 x_5 + x_1 x_2 x_3^2 x_4^2 + x_1 x_2^2 x_4^2 + 1$$

$$G_5 = x_1 x_3^2 x_4 x_5^2 + x_2 x_5^2 + x_1 x_2 x_4 x_5 + x_2 x_5 + x_1 x_2 x_3 x_4^2$$

Z6

$$D_6 = x_1x_2x_4^2x_5^2x_6^2 + x_1x_2^2x_3^2x_4x_5^2x_6^2 + x_1^2x_3x_6^2 + x_1^2x_2x_3^2x_4x_5^2x_6 + x_1^2x_3x_5x_6$$

$$F_6 = x_1^2x_2x_4x_5^2x_6^2 + x_1x_3x_5^2x_6^2 + x_1x_2^2x_6^2 + x_1^2x_2^2x_3^2x_5x_6 + x_1x_3^2x_4x_5$$

$$G_6 = x_2^2x_3^2x_4x_5^2x_6 + x_1x_4^2x_5x_6 + x_2^2x_3^2x_4x_5x_6 + x_1x_2^2x_3x_4^2x_6 + x_1^2x_3x_5^2$$

Z7

$$D_7 = x_1x_2^2x_4^2x_6^2x_7^2 + x_1^2x_3x_4x_6^2x_7^2 + x_3^2x_4^2x_7^2 + x_1^2x_2x_4^2x_6 + x_3x_4x_5^2$$

$$F_7 = x_1^2x_2x_4^2x_5x_6^2x_7^2 + x_1x_2x_3x_6x_7 + x_3x_4^2x_5^2x_7 + x_1x_4^2x_5^2x_7 + x_1^2x_2x_3x_4^2x_5x_6$$

$$G_7 = x_1x_3x_5x_6^2x_7^2 + x_2^2x_3^2x_4^2x_5x_6x_7^2 + x_4x_6x_7^2 + x_1^2x_2x_3x_5x_6x_7 + x_1^2x_3^2x_4x_5^2$$

Z8

$$D_8 = x_2^2x_4x_5x_6x_7x_8^2 + x_1^2x_2x_3^2x_4^2x_6^2x_7^2x_8 + x_1^2x_3x_4^2x_6^2x_7^2 + x_1^2x_2^2x_3^2x_4x_5^2x_6x_7^2 + x_2^2x_4x_6$$

$$F_8 = x_1^2x_2^2x_3x_4^2x_5x_6^2x_8^2 + x_2x_5x_6^2x_8^2 + x_1^2x_2^2x_3^2x_4^2x_6^2x_7^2x_8 + x_1^2x_3^2x_4x_5^2x_7^2x_8 + x_1x_2^2x_3^2x_5^2x_7$$

$$G_8 = x_1x_4^2x_5x_6x_7x_8^2 + x_1x_2^2x_4^2x_5^2x_6^2x_8 + x_1^2x_2x_3x_4^2x_6^2x_8 + x_1^2x_2^2x_3^2x_4x_5^2x_8 + x_1x_2x_4^2x_5^2$$

Z9

$$D_9 = x_1^2x_3^3x_4x_6x_8x_9^2 + x_1x_2x_3x_4^2x_5^2x_8x_9 + x_2x_3x_4x_5^2x_8x_9 + x_1x_3^3x_4^2x_5^2x_6^2x_7x_8^2 + x_2x_3x_4x_5^2x_6x_7x_8^2$$

$$F_9 = x_1^2x_2^2x_3x_7^2x_8x_9 + x_2^2x_9 + x_1^2x_3x_4^2x_5^2x_6x_7^2 + x_4^2x_5^2x_7^2 + x_3x_4^2x_6x_7$$

$$G_9 = x_1^2x_2x_4x_5x_6x_7^2x_8^2x_9^2 + x_1^2x_2x_3x_5x_6^2x_7^2x_8x_9^2 + x_1^2x_3x_4x_6x_7^2x_8x_9 + x_1^2x_2^2x_6x_8^2 + x_2^2x_4x_5x_6^2x_7$$

Z10

$$D_{10} = x_1x_2^2x_4^2x_8x_9^2x_{10}^2 + x_2^2x_4x_5^2x_6x_7x_9x_{10}^2 + x_1^2x_2x_3x_5^2x_7^2x_9^2 + x_1x_3^2x_4^2x_7^2x_9^2 + x_1^2x_3x_4x_7^2x_8^2$$

$$F_{10} = x_1x_2x_3^2x_4x_6x_7x_8x_9^2x_{10}^2 + x_2^2x_3^2x_4^2x_6^2x_9x_{10}^2 + x_1x_2^2x_3^2x_4x_5x_6x_7x_8^2x_9^2x_{10} + x_1^2x_2x_4^2x_5^2x_8^2x_9^2x_{10} + x_3x_4^2x_5x_6x_7^2x_9x_{10}$$

$$G_{10} = x_1x_2^2x_3^2x_5^2x_6^2x_7x_8x_9^2x_{10}^2 + x_3x_8x_9^2x_{10}^2 + x_1x_2^2x_3x_4x_5^2x_6^2x_8^2x_9x_{10} + x_1x_3x_6x_7x_8x_{10} + x_4^2x_5^2x_6^2x_7x_9^2$$

Set M: example with many terms in the gcd

M1

$$D = x^{10}y^3z + 6x^9y + x^8y^2 + 7x^8z - x^7 - 3x^6z^3 + x^5y^5z^5 + 2x^3z^2 - 10x^2y^3z^{10} + x - 20$$

$$F = x^3 + 2yz - x$$

$$G = 4x^4(y^2 + z) - 2x^2y + z - 3$$

Set T: examples with trivial gcds, based on set Z.

T1

$$F_1 = ZF_1 - 1$$

$$G_1 = ZG_1$$

T2

$$F_2 = ZF_2 - (2x_1 - 2)$$

$$G_2 = ZG_2 - x_1^2$$

T3

$$F_3 = ZF_3 - (x_2^2x_3^3 - x_2^2x_3^2)$$

$$G_3 = ZG_3 - (2x_1^3x_2x_3 - 2x_1^3x_2 - 8x_2x_3^2)$$

T4

$$F_4 = ZF_4 - (-1)x_1^2x_2x_3x_4^2$$

$$G_4 = ZG_4$$

T5

$$F_5 = ZF_5 - (x_1x_2^2x_5^2 - x_2^2x_5^5)$$

$$G_5 = ZG_5 - 2x_1^2x_2^2x_3^2x_4^2x_5$$

T6

$$F_6 = ZF_6 - (x_1^4 x_2^3 x_3^4 x_4 x_5^3 x_6^2 + x_1^2 x_2 x_3 x_4^2 x_5^4 x_6^4 - x_1^2 x_2 x_3 x_4 x_5^4 x_6^4)$$

$$G_6 = ZG_6 - (x_1^4 x_2 x_3^3 x_4 x_5^4 x_6 + x_1 x_2^3 x_3^2 x_4^3 x_5^4 x_6^3 + x_1 x_2^3 x_3^2 x_4^3 x_5^3 x_6^3 - x_2^3 x_3^2 x_4^3 x_4^4)$$

T7

$$F_7 = ZF_7$$

$$G_7 = ZG_7 - (x_2^2 x_3^3 x_4^3 x_5^3 x_6 x_7^2 - x_2^2 x_3^3 x_5^3 x_6 x_7^2 + x_3^2 x_4^3 x_6 x_7^4)$$

T8

$$F_8 = ZF_8 - x_1^4 x_2^4 x_3^4 x_4^3 x_5^2 x_6^3 x_7^4 x_8$$

$$G_8 = ZG_8 - (x_1 x_2^2 x_4^3 x_5^2 x_6^2 x_7^2 x_8^4 - x_1 x_2^2 x_4^3 x_5^2 x_7^2 x_8^4 + x_1 x_2^2 x_4^3 x_5 x_6^2 x_7 x_8^2 - x_1 x_2^2 x_5 x_6^2 x_7 x_8^2)$$

T9

$$F_9 = ZF_9$$

$$G_9 = ZG_9 - (x_1^3 x_2 x_3^4 x_4^2 x_5^3 x_6^4 x_7^3 x_8^3 x_9^2 + x_2^3 x_3 x_4^2 x_5^3 x_6^2 x_7 x_8 x_9 - x_2^3 x_5^5 x_6^2 x_7 x_9)$$

T10

$$F_{10} = ZF_{10}$$

$$G_{10} = ZG_{10} - x_1^3 x_{10} x_2^2 x_3^2 x_4^2 x_5^5 x_6^2 x_7^2 x_8^4 x_9$$

References

Baker, H. (1978), List Processing in Real Time on a Serial Computer. *Comm. of the ACM* **21** 280-294.

Borodin, A., J. von zur Gathen and J. Hopcroft (1982), Fast Parallel Matrix and GCD Computations. *Info. and Control* **52** 241-256.

Brown, W.S. (1971), On Euclid's Algorithm and the Computation of Polynomial Greatest Divisors. *J. ACM* **18** 478-504.

Buchberger, B. (1965), *An Algorithm for Finding a Basis for the Residue Class Ring of a Zero-Dimensional Polynomial Ideal (German).* Ph.D. thesis presented to Math. Inst., Univ. of Innsbruck, Austria and *Aequationes Math* **4** 374-383 (1970).

Buchberger, B. (1985), The L-Machine: An Attempt at Parallel Hardware for Symbolic Computation. *Proc. Third Int. Conf. AAECC-3 Grenoble, France (July 15-19, 1985),* LNCS 229, 333-347.

Buchberger, B. and R. Loos (1983), Algebraic Simplification. In *Computer Algebra, Symbolic and Algebraic Computation,* Ed. by B. Buchberger *et. al.*, Springer-Verlag, Wien.

Burton, F. (1987), Functional Programming for Concurrent and Distributed Computing. *The Computer J.* **30** 437-450.

Calmet, J. and J.A. van Hulzen (1983) Computer Algebra Applications. In *Computer Algebra, Symbolic and Algebraic Computation,* Ed. by B. Buchberger *et. al.*, Springer-Verlag, Wien.

Davenport, J.H. and J. Padget (1985), HEUGCD: How Elementary Upper Bounds Generate Cheaper Data. *Proc. EUROCAL '85, Linz, Austria (April 1-3, 1985),* LNCS 204, 18-28.

Davenport, J.H., Y. Siret and E. Tournier (1988), *Computer Algebra Systems and Algorithms for Algebraic Computation.* Academic Press, London.

Dourish, P (1989), *Implementing a Parallel Lisp system on a Transputer Array.* Progress Report, Edinburgh University (private communication).

Eager, D.L., J. Zahorjan and E.D. Lazowska (1989), Speedup Versus Efficiency in Parallel Systems. *IEEE Trans. Comp.* **38** 408-423.

Flatt, H.P. and K. Kennedy (1989), Performance of Parallel Processors. *Parallel Computing* **12** 1-20.

Geddes, K., S. Czapor and G. Labahn (1990), *Algorithms for Computer Algebra.* (Draft manuscript), Aug 1990.

Gelernter, D. (1985), Generative Communications in Linda. *ACM Trans. Programming Languages and Systems* **7** 80-112.

Gianni, P. and B. Trager (1985), GCD's and Factoring Multivariate Polynomials using Grobner Bases. *Proc. EUROCAL '85, Linz, Austria (April 1-3, 1985)*, LNCS 204, 409-410.

Goldman, R. and R. Gabriel (1988), Qlisp: Experience and New Directions. *SIGPLAN Notices* **23** 111-123.

Graham I.D. and T. King (1990), *The Transputer Book*. Prentice Hall, London.

Halstead, R. Jnr. (1985), Multilisp: A Language for Concurrent Symbolic Computation. *ACM Trans. Programming Languages & Systems* **7** 501-538.

Iwasaki, H. (1989), mUtilisp: A Lisp Dialect for Parallel Processing. In *Parallel Lisp: Language and Systems. Proc. US/Japan Workshop on Parallel Lisp, Sendai, Japan, (June 5-8, 1989)*, LNCS 441, 316-321.

Kaltofen, E. and L. Yagati (1988), Improved Sparse Multivariate Polynomial Interpolation Algorithms. *Proc. ISSAC '88, Rome, Italy (July 4-8, 1988)*, LNCS 358, 467-474.

Kessler, R. and M. Swanson (1989), Concurrent Scheme. In *Parallel Lisp: Language and Systems. Proc. US/Japan Workshop on Parallel Lisp, Sendai, Japan, (June 5-8, 1989)*, LNCS 441, 200-234.

Knuth, D.E. (1969), *The Art of Computer Programming: vol 2 Seminumerical Algorithms*. Addison-Wesley, Reading, Mass.

Kranz, D.A. (1989), Mul-T: A High Performance Parallel Lisp (extended abstract). In *Parallel Lisp: Language and Systems. Proc. US/Japan Workshop on Parallel Lisp, Sendai, Japan, (June 5-8, 1989)*, LNCS 441, 306-311.

Krishnamurthy, E.V. (1989) *Parallel Processing: Principles and Practice*. Addison-Wesley, Sydney.

Loos, R. (1983), Introduction to *Computer Algebra, Symbolic and Algebraic Computation*. Ed. by B. Buchberger *et. al.*, Springer-Verlag, Wien.

Mayes, P. and S. Hammarling (1987), Parallel Computers and Algorithms. *NAG Newsletter* 1/87 3-15.

Moses, J. and D. Yun (1973), The EZGCD Algorithm. *Proc. 1973 ACM Nat. Conf. New York, (August, 1973)*, 159-166.

Osbourne, R. (1989), Speculative Computation in Multilisp. In *Parallel Lisp: Language and Systems. Proc. US/Japan Workshop on Parallel Lisp, Sendai, Japan, (June 5-8, 1989)*, LNCS 441, 322-349.

Perihelion Software, (1991), *The Helios Parallel Operating System*. Prentice Hall, London.

Piquer, J. (1989), Private communication.

Ponder, C. (1988a), Parallelism and Algorithms for Algebraic Manipulation: Current Work. *SIGSAM Bulletin* **22** 7-14.

Ponder, C. (1988b), Parallel Processors and Systems for Algebraic Manipulation: Current Work. *SIGSAM Bulletin* **22** 15-21.

Senechaud, P. (1991), A MIMD Implementation of the Buchberger Algorithm. *Parallel Computing* **17** 9-37.

Stamos, J. and D. Gifford (1990), Remote Evaluation. *ACM Trans. Programming Languages and Systems* **12** 537-565.

Stoutemyer, D.R. (1984), Which Polynomial Representation is Best? *Proc. 1984 MACSYMA Users Conference, Schenectady, New York, (July 23-25, 1984),* 221-243.

von zur Gathen, J. (1984), Parallel Algorithms for Algebraic Problems. *SIAM J. Comput.* **13** 802-824.

von zur Gathen, J. (1985), Factoring Sparse Multivariate Polynomials. *J. Computer & System Sciences* **31** 265-287.

von zur Gathen, J. (1986), Parallel Arithmetic Computations: A Survey. *Proc. Mathematical Foundations of Computer Science 1986: Proc. 12th Symposium, Bratislavia, Czechoslovakia, (Aug 25-29),* LNCS 233 91-112.

Wang, P.S. (1980), The EEZ-GCD Algorithm *SIGSAM Bulletin* **14** 50-60.

Watt, S. (1985a), *Bounded Parallelism in Computer Algebra.* Ph.D. thesis presented to Univ. of Waterloo, Canada.

Watt, S. (1985b), A System for Parallel Computer Algebra Programs. *Proc. EUROCAL '85, Linz, Austria, (April 1-3, 1985),* LNCS 204, 537-538.

Willcock, D.M.K. (1987), *A Polynomial Factorisation Package.* M.Sc. Dissertation, University of Waikato.

Yuasa, T. and T. Kanawa (1989), PM1 and PMLisp: An Experimental Machine and its Lisp System for Research on MIMD Massively Parallel Computation. *Parallel Lisp: Language and Systems. Proc. US/Japan Workshop on Parallel Lisp, Sendai, Japan, (June 5-8, 1989),* LNCS 441, 322-349.

Zassenhaus, H. (1969), On Hensel Factorisation I. *J. Number Theory* **1** 291-311.

Zippel, R. (1979), Probabilistic Algorithms for Sparse Polynomials. *Proc. EUROSAM '79, Marseille, France (June, 1979),* LNCS 72, 216-226.

Zippel, R. (1981), Newton's Iteration and the Sparse Hensel Algorithm. *Proc. 1981 ACM Symposium on Symbolic and Algebraic Computation, Snowbird, Utah, (Aug 5-7)* 68-72.

Zippel, R. (1988) *Computer Algebra Part 1: Algebraic Techniques.* Course notes, M.I.T, (draft manuscript).

Zippel, R. (1990), Interpolating Polynomials from their Values. *J. Symbolic Computation* **9** 375-403.

Zorn, B., P. Hilfinger, K. Ho, J. Larus and L. Semenzato (1988), Features for Multiprocessing in Spur Lisp. *Report No. UBC/CSD 88/406 1988 Computer Science Division (EECS) Uni. California Berkeley.*

ERRATA

Page 34

Line 8

This sequence is the simplest PRS of those $\cdots$

should read

This PRS is the simplest to compute of those $\cdots$

Page 39

Line 15

$(\xi \times 73794)/27011$ (to ensure that $\cdots$

should read

$(\xi \times 73794)/27011$ (where 73794/27011 was choosen at random such that $\cdots$

Page 49

Line 17

$$B = \max(\text{coefs}(F_1))\max(\text{coefs}(F_2))$$

should read

$$B = \max(\text{icoefs}(F_1))\max(\text{icoefs}(F_2))$$

Page 50

Line 11

Using $x_{1j} = 0, 11, 13, 5$, the values of $c_2(X_0, x_{1j}, x_{20})$ may taken from

$$\gcd(F_1(X_0,\ 0, 15),\ F_2(X_0,\ 0, 15)) = X_0^3 + 0X_0 + 3$$

should read

Using $x_{1j} = 12, 11, 13, 5$, the values of $c_2(X_0, x_{1j}, x_{20})$ may be taken from

$$\gcd(F_1(X_0,\ 12, 15),\ F_2(X_0,\ 12, 15)) = X_0^3 + 3X_0 + 3$$

Line 17

Interpolating $\cdots$ (*viz* 0, 5, –7, –6) $\cdots$

should read

Interpolating $\cdots$ (*viz* 12, 5, –7, –6) $\cdots$

Line 28

Suppose $x_{20}=0$ and $\bar{y}=(2)$ then $\bar{y}^{\bar{e}_1}=2^3=8$ and $\bar{y}^{\bar{e}_2}=2^1=2$.

should read

Suppose $\bar{y}=(4)$ then $\bar{y}^{\bar{e}_1}=4^3=7$ and $\bar{y}^{\bar{e}_2}=4^1=4$.

Page 51

Lines 7 through 14 should read

Now

$$\gcd(F_1(X_0,1,x_{20}),F_2(X_0,1,x_{20}))=X_0^3+4X_0+3 \quad \text{(modulo 19)}$$

and

$$\gcd(F_1(X_0,2,x_{20}),F_2(X_0,2,x_{20}))=X_0^3+3X_0+3 \quad \text{(modulo 19)}$$

giving the system

$$p_{10}+p_{20}=4$$

$$8p_{10}+2p_{20}=3$$

which is solved: $p_{10}=2$ and $p_{20}=2$ (modulo 19).

First row of table should have entries

0	15	(4)	$X_0^3+4X_0+3$	$X_0^3+3X_0+3$	2	2

Page 52

The transposed Vandermonde system should have T equations, i.e. the last equation (line 25) should read

$$P'(y_1^{T-1},\ldots,y_{k-1}^{T-1},x_{kj})=p_{1j}\bar{y}^{(T-1)\bar{e}_1}+p_{2j}\bar{y}^{(T-1)\bar{e}_2}+\cdots+p_{Tj}\bar{y}^{(T-1)\bar{e}_T}$$

Page 53

Lines 11 through 14

$$(v_1,v_2,\ldots,v_T)=(\bar{y}^0,\bar{y}^{\bar{e}_1},\ldots,\bar{y}^{\bar{e}_{T-1}})$$

and

$$(a_1,a_2,\ldots,a_T)=(P'(1,\ldots,1,x_{kj}),P'(y_1^2,\ldots,y_{k-i}^2,x_{kj}),\ldots,P'(y_1^T,\ldots,y_{k-i}^T,x_{kj}))$$

should read

$$(v_1, v_2, \ldots, v_T) = (\overline{y}^{\overline{z}_1}, \ldots, \overline{y}^{\overline{z}_T})$$

and

$$(a_1, a_2, \ldots, a_T) = (P'(1, \ldots, 1, x_{kj}), P'(y_1^2, \ldots, y_{k-i}^2, x_{kj}), \ldots, P'(y_1^{T-1}, \ldots, y_{k-i}^{T-1}, x_{kj}))$$

Line 15

$\cdots$ according to the function being lifting

should read

$\cdots$ according to the function being lifted

Page 55

Line 12

$\cdots$ one iteration of ZG3 is dependent only on the previous iteration of ZG3 and not on ZG4

should read

$\cdots$ one iteration of ZG3 is dependent on neither the previous iteration of ZG3 nor on ZG4

Page 56

Line 15

cont $\leftarrow$ *PARALLEL–GCD* (cont(*coefs* (F_1)), cont(*coefs* (F_2)))

should read

cont $\leftarrow$ *PARALLEL–GCD* (cont(F_1), cont(F_2))

Line 25

$G_0 \leftarrow \cdots$ **norm*

should read

$G_0 \leftarrow \cdots$ **norm'*(X_0)

Line 29

$d_k \leftarrow \cdots$ **norm*

should read

$d_k \leftarrow \cdots$ **norm*($x_{10}, \ldots, x_{k-1,0}, X_k, x_{k+1,0}, \ldots, x_{v0}$))

Page 57

Line 7

$G_j(X_0) \leftarrow \cdots * norm$

should read

$G_j(X_0) \leftarrow \cdots * norm'(X_0, r_j)$

Line 25

For $i = 0, \ldots, T$ compute

should read

For $i = 0, \ldots, T-1$ compute

Page 64

Line 21 $E_p' = \frac{1}{S_p'}$ *should read* $E_p' = \frac{S_p'}{p}$

Line 28 $E_p = \frac{1}{S_p}$ *should read* $E_p = \frac{S_p}{p}$

Functions named but not listed in Appendix A

internal_write_msg*(link, string, key) writes string and key to soft_link, link*

internal_read_msg(*link, string, key) reads string and key from soft_link, link*

external_write_msg*(file_des, msg) writes msg to file descriptor file_des*

external_read_msg*(file_des, msg) reads msg from file descriptor file_des*

push *(form, key) puts job consisting of form and key on back of job queue*

pop ()*takes job consisting of form and key from front of job queue*

wait_for_msg *(link-in, link-out) repeatedly read in msg and write to link-out*

wait_for_word *(link-in, link-out) repeatedly read in word and write to link-out*

retrieve_entry *(store, key, form) retrieves, from store, form with key as index*

store_entry *(store, key form) stores, in store, form with key as index*

Functions called but not listed in Appendix B

zippel-Fgcd *(F1 F2 norm x-vec p) returns (gcd(F1(xvec),F2(x-vec)))*norm(x-vec) mod p*

zippel-next *(sjs S p %rjs %qjs) where S is a skeleton poly, uses CRA and function heights in sjs to lift next variable*

vdm-solve *(v-vec a-vec p) returns x-vec=* A^{-1}*a-vec(mod p) where A is transposed vdm matrix from v-vec*

%poly3modpevalto3 *(poly consts p) evaluates (mod p) poly with last m variables set to the m values in consts, returns a poly3*

%termuponevar *(clist) tack on a zero power to each skeleton coef in clist*

zippel-Xpoly *(Xlist clist) returns a poly3 constructed from powers in Xlist and coefs in clist*

%polyXto3 *(Xpoly %xveczero) Xpoly is a poly1 except coefs are poly3s - singles out* X_0*;*
%poly3tox *(Poly) these functions do conversions between Xpoly and poly3 data types*

%termlistprim *(clist) returns primitive part of each coef in clist*

%Transpose *(A) returns transpose of A which is a list of lists*

%nextprime *(p) gets next prime from a list of primes*

zipple-clist *(xlist Fx) splices a 0 into xlist to mark missing terms*

%dense-init *(r dmax modp %baserand) computes dmax random nos* $r_0,\cdots,r_{d\max}$ *(where* $r_0 = r$*) and corresponding* d_j *values for CRA*

zippel-badhomomorphism *(clists) checks for a poly (being one of the lists of coefs in clists) with degree greater than that of the other polys*

%coef-nterm *(n poly1) returns coef of* x^n *in poly1*

%maxabs *(consts) returns max absolute value of constants in consts*

%**apply-mapcar** *(op op2 args) similiar to (apply #'mapcar (cons op args)).*

%**invmod** *(m p) returns m1 such that m1*m=1 mod p (positive inverse)*

%**pointeval** *obscure special purpose functions used by zippel-degrees*
%**wapswap** *for evaluaiton of polys with changing main var*

%**poly1pows** *poly1 →power list*

%**poly11modpgcd** *poly1* x *poly1* x *p* → *poly1*

%**poly11gcd** *poly1* x *poly1* →*poly1*

%**polymodpcanon** *(powers coefs p) makes a poly1 (mod p) from non-canonical powerlist and coefs*

%**poly1to3** *poly31→ poly3 only applicable*
%**poly3to1** *poly3 → poly 1 to univariate polys*

%**poly3deg** *(poly3) returns degree of poly3*

%**poly3scont** *list of poly3 → poly3*

%**poly33gcd** *poly3* x *poly3 → poly3*

%**poly33cra** *(c ml m2 p1 p2) uses CRA to return poly3, p, such that p ≡p mod m1 and p ≡p2 mod m2*

%**poly3centre** *poly3* x *p → poly3 converts integer coefs to balanced representation*

%**poly33*** *poly3* x *poly3 → poly3*