Technical Report

UCAM-CL-TR-537
ISSN 1476-2986

Number 537

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Parallel systems in symbolic and algebraic computation

Mantsika Matooane

June 2002

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted August 2001 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Trinity College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

Series editor: Markus Kuhn

ISSN 1476-2986

Abstract

This thesis describes techniques that exploit the distributed memory in massively parallel processors to satisfy the peak memory requirements of some very large computer algebra problems. Our aim is to achieve balanced memory use, which differentiates this work from other parallel systems whose focus is on gaining speedup. It is widely observed that failures in computer algebra systems are mostly due to memory overload: for several problems in computer algebra, some of the best available algorithms suffer from *intermediate expression swell* where the result is of reasonable size, but the intermediate calculation encounters severe memory limitations. This observation motivates our memory-centric approach to parallelizing computer algebra algorithms.

The memory balancing is based on a randomized hashing algorithm for dynamic distribution of data. Dynamic distribution means that the intermediate data is allocated storage space at the time that it is created and therefore the system can avoid overloading some processing elements.

Large scale computer algebra problems with peak memory demands of more than 10 gigabytes are considered. Distributed memory can scale to satisfy these requirements. For example, the Hitachi SR2201 which is the target architecture in this research provides up to 56 gigabytes of memory.

The system has *fine granularity*: tasks sizes are small and data is partitioned in small blocks. The fine granularity provides flexibility in controlling memory balance but incurs higher communication costs. The communication overhead is reduced by an intelligent scheduler which performs asynchronous overlap of communication and computation.

The implementation provides a polynomial algebra system with operations on multivariate polynomials and matrices with polynomial entries. Within this framework it is possible to find computations with large memory demands, for example, solving large sparse systems of linear equations and Gröbner base computations.

The parallel algorithms that have been implemented are based on the standard algorithms for polynomial algebra. This demonstrates that careful attention to memory management aids solution of very large problems even without the benefit of advanced algorithms. The parallel implementation can be used to solve larger problems than have previously been possible.

Contents

List of Figures

Chapter 1

Introduction

Computations with *symbolic* data items form a significant part of scientific computation methods. These differ from the more widely used systems for numerical data manipulation in that the main goal of symbolic computation is *exactness*. Symbolic manipulation may be considered to *defer* numerical computation; computing with, and storing symbolic data delays the introduction of numerical constants, therefore reduces numerical error propagation. Within the classification of symbolic computation, this work focuses on symbolic algebraic computation (SAC) or computer algebra.[1]

This dissertation describes work on the development of a parallel computer algebra system capable of performing algebraic manipulation on very large data sets. The algebraic domains that are considered have arbitrary precision integers, polynomials in several variables, and matrices with polynomial entries as basic data objects.

Computer algebra systems are widely used in science and industry to perform large calculations. For some users, they are powerful calculators providing higher mathematical calculations that would be tedious and error-prone when performed by hand. For these users, advanced commercial systems on single processor personal computers are adequate.

On the other hand, some scientists making use of available systems are able to stretch the resources, and often encounter the time or space limitations of current systems. For these *power users*, computer algebra is an invaluable tool but they require more from it. To address this need for powerful systems with more resources, researchers in computer algebra have encouraged urgent attention to parallelism [23].

A report by Boyle and Caviness [14, pages 65–66], warns that

> "...symbolic computation software development is lagging behind new hardware technology...in the use of new architectures, including supercomputers."

The authors recommend more research on parallel symbolic algorithms. This challenge is the starting point for the work reported here. The design and implementa-

[1] Other areas of symbolic computation include symbolic logic computation and automated theorem proving.

tion of a parallel computer algebra system was undertaken with particular emphasis on memory management.

1.1 Motivation

The pursuit of a parallel system for computer algebra leads to several theoretical, system and application questions. At the theoretical level, the fundamental models of parallel computation apply, and many competing models and architectures are available. Practical system development requires decisions on issues such as granularity. The identification of computer algebra algorithms that have inherent concurrency is a critical factor in application selection.

The main research question concerns the relative importance of the three key resources in a parallel system: time, space and communication. This thesis emphasises the critical nature of memory space for computer algebra problems.

Many parallel implementations (for computer algebra or other problems) aim to make a computation faster, attaining *speedup*. This thesis suggests that, for parallel computer algebra systems, the critical resource is *memory space*. The justification for this approach relies on the following observations about computer algebra systems:

1. Intermediate expression swell,
2. Global memory overflow and
3. Node memory overflow

These are considered in turn and the issues arising in each case are highlighted.

1.1.1 Intermediate expression swell

Computation with algebraic terms has often been found to depend critically on the maximum amount of memory required by the algorithm at any stage of the computation [14]: the input size of a problem may be small, but its memory use in intermediate stages of the calculation may grow very large. This phenomenon is described by Tobey [118] as *intermediate expression swell* and has been observed since the earliest computer algebra systems [19]. Consider the following example from Char *et al.* [27]:

$$\begin{aligned} p1 = \; & 91x^{99} + 6x^{89} - 34x^{80} - 20x^{76} + 56x^{54} + 25x^{52} + 86x^{44} \\ & + 17x^{33} - 70x^{31} + 17 \end{aligned}$$

$$\begin{aligned} p2 = \; & 16x^{95} + 46x^{84} + 38x^{76} + 88x^{74} + 81x^{72} + 21x^{52} - 91x^{49} \\ & - 96x^{25} - 64x^{10} + 20 \end{aligned}$$

These polynomials are relatively prime therefore $gcd(p1, p2) = 1$, but the computation of recursive polynomial remainder sequences shows growth in the number of

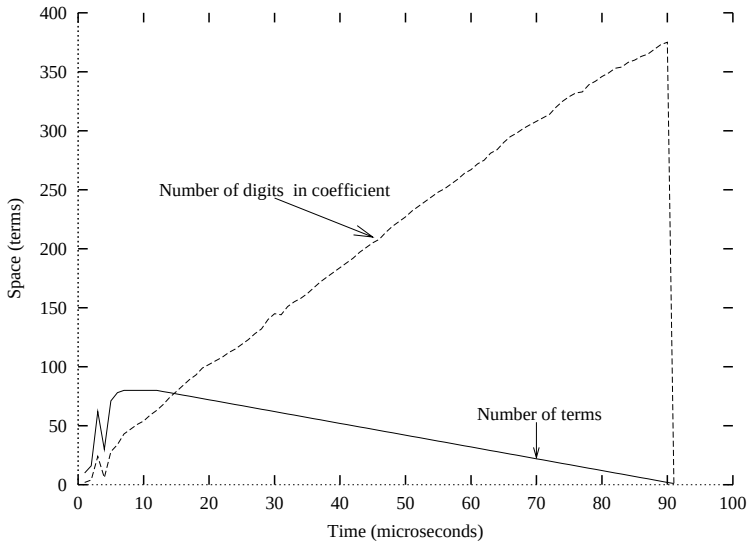


Figure 1.1: Growing memory requirements for a GCD calculation

terms in the intermediate results and coefficient size before collapsing to the final result. The growth in resource requirements is shown in figure 1.1.

The example is small by today's standards and a naive GCD algorithm was deliberately used so that the growth could be illustrated. For larger problems of the same kind, state-of-the-art algorithms are often available, but many will still require very large memory.

The example serves to demonstrate that the *peak* memory demands for larger problems of this class may exceed available resources. Therefore memory resource management becomes a key requirement for computer algebra systems.

Algorithmic improvements are the best way to manage expression size. The subresultant algorithm [28, 18, 20] avoids intermediate expression swell in sequential computation of polynomial gcd's. Further algorithmic improvements have been developed by Corless *et al.* [32] where hierarchical representation tools are employed to reduce expression swell in perturbation problems. Such algorithmic improvements are as yet available for a small number of algorithms. Parallel systems combined with careful attention to the efficient use of memory [112] can provide an alternative approach that satisfies the large memory requirements.

This research concentrates on memory allocation and space complexity, in contrast with some parallel computer algebra systems where the parallelism is used to speed up computations that can take too much time on a workstation of typical size. In this work, speedup is not the major goal, thus placing this work on a comparatively different footing from related studies in parallel computer algebra. The critical role of intermediate expression swell demands a focus on memory systems that cope with *peak* memory demand during a calculation, even when the final result is a solution of modest size.

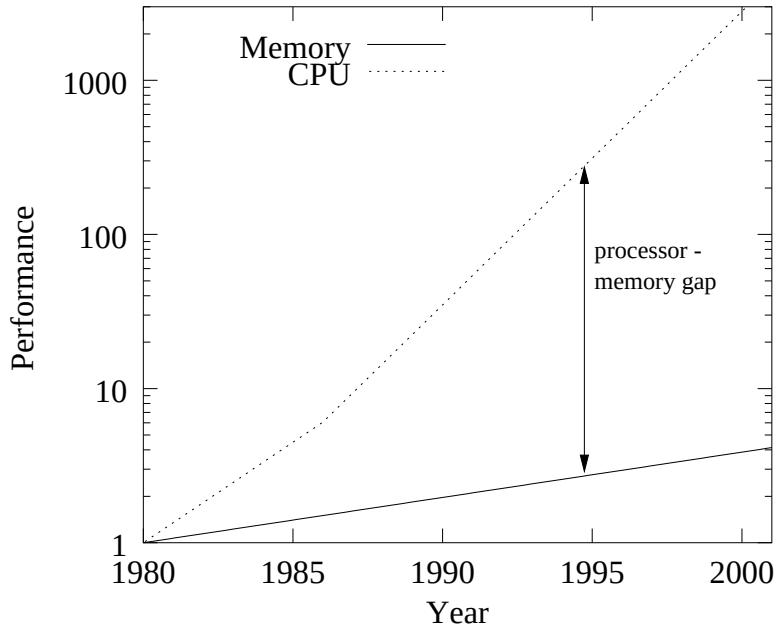


Figure 1.2: The comparative performance of memory and CPU

1.1.2 Global memory overflow

Theoretical models of computation including Turing machines, register machines, recursive functions and many more, assume unlimited memory [116]. However, implementation in hardware gives finite memory, placing a physical limitation on computation. Computations that require more memory than available encounter memory overload and terminate.

In addition to the *size* of memory available, we have to address the *performance* of memory. While processor speed continues to grow according to Moore's Law, the performance of memory has consistently been over 50% slower. The widening processor-memory gap [61] shown in figure 1.2 places further constraints on systems.[2] Fast caching principles and the memory hierarchy are successfully used to keep the fast processor busy. However, 32-bit processors have a 4GB address space therefore to provide over $10GB$ demands a parallel approach. The most scalable memory architecture is the distributed memory machine since each additional node increases the total storage by the amount of its local memory.

This research focuses on calculations that require a large amount of *real memory*. While the use of virtual memory is a successful architectural feature for sequential systems, it is worth noting that:

- Virtual memory provides large memory for problems with known locality of data access. Applications targeted in this thesis display little data locality and it is quite difficult to identify locality accesses.

- Virtual memory may become less effective for applications with irregular ac-

[2] As an example of the existing gap, an average workstation now has a $1GHz$ processor but only 128MB memory.

cess to large amounts of data, due to thrashing. This thesis is concerned with irregular computer algebra algorithms such as sparse systems of linear equations, and these suffer from inefficiency in the use of the cache.

Algebraic computations, unlike purely numeric computation on a massively parallel processor, tend to have unpredictable data dependencies, irregular data access patterns and varying computation time dependent on parameters such as order. All these make it difficult to predict patterns of memory use, and therefore render some of the current techniques ineffective.

1.1.3 Node memory overflow

The distributed memory parallel architecture distinguishes between local and global memory. Each processor can directly access its local storage, but access to data in other processing elements can only be accessed through communication messages.

Since each processor has a local address space, it is possible for one processor to exceed its available memory, while there is still space available globally [103]. This may be referred to as *node memory overflow*. This would terminate the entire computation, although the total available resources could have been enough for the computation.

Avoiding these failures requires careful attention to memory demands. Memory overload in a parallel system is difficult to anticipate since the global availability does not prevent excess requirements in a single node.

1.2 Randomized dynamic distribution

A feature of our memory management solution is the use of dynamic relocation of data during a computation, thus bounding the memory use per node.

The relocation is achieved through a *randomized algorithm*. The algorithm leads to a good distribution of the data across all processing elements of the system.[3] The algorithm is based on distribution at fine granularity, where each buffer sent across the network for storage at some processing element is of a small uniform size. Such fine granularity leads to increased total number of messages and therefore higher communication costs.

Dynamic distribution of data is a useful strategy in this case since static analysis before execution rarely gives much guidance about locality or the probability of a high number of communication messages. However, random data placement would be unsuitable for tasks where it is possible to make reliable predictions about communication patterns and hence keep most communication local.

1.2.1 Static data distribution

The most widely used data distribution algorithms, such as block distribution, cyclic distribution, and variations on these, are *static*. At the start of a computation, data

[3]The use of the word *distribution* here has a statistical meaning. In different contexts distribution refers to storage allocation. We will use the word *relocate* for operations that move data.

are divided between all available processors. The distribution algorithm is selected to increase data locality giving each processor quick access to the data it requires.

In systems with static data distribution, the threat of node memory overflow is increased: high memory demands on a single node, with no redistribution, can quickly exhaust the local memory leading to failure.

On the other hand, a successful static distribution is cheap. The multiple instruction stream multiple data stream (MIMD) programming model implicitly requires some initial data partitioning for the start of parallel computation (see Section 2.1.5). For these reasons, a combined distribution strategy incorporating early static distribution and then dynamic relocation may be used.

1.2.2 Granularity

The granularity of a problem is a measure of the size of a block of computation between communication steps. Problems may fall into one of two granularity classes: *coarse grain* or *fine grain* problems. This classification is clearly very broad and the boundary may vary. Fitch [44] defines coarse granularity as *functional* parallelism at the level of function invocation, while fine granularity is *algorithmic* parallelism.

Granularity is inversely proportional to the communication time, therefore many systems implement only coarse grain parallelism in order to reduce communication time. Let τ be a lower bound on the communication time for parallel implementation of a problem. Problems that can be implemented with minimal communication τ will have coarse granularity.

Fine granularity incurs communication cost but gives greater flexibility in memory allocation,[4] by allowing each small computation block to carry with it the memory requirements. Coarse granularity will often create fewer communication instances, and each message may involve movement of a larger block of data, thus reducing the accuracy of any approximation of the memory size in use.

1.2.3 Load balancing

Load balancing measures the variance in the CPU time (execution time) for each processor. The execution time for a computation is the maximum CPU time of all processors. Consider a computation distributed across several processing elements PE_i, and let the execution time for a computation on each processor be E_i. Then the total parallel execution time T_p for complete computation of is given by:

$$T_p = \max(E_i) \text{ where } 1 \leq i \leq p \tag{1.1}$$

Then we may determine the idle time I_i for each processing element (PE) as the difference between the its local execution time and the time to complete the computation:

$$I_i = T_p - E_i$$

[4] A recurring point in this research is that where several options are weighed, the one that simplifies the memory model is often adopted.

The completion of execution is limited by the slowest running time of all PEs, $T_p = E_i + I_i$, at some processor i. Therefore reduction of idle time through good load balance is crucial.

Load balancing techniques minimize idle time by attempting to assign each PE a task requiring an equal amount of CPU time. A good load balancing procedure improves the speedup of a computation. The challenge is that our memory balancing procedure should provide enough data locality so that the CPU load on each processor is maintained.

1.3 Algebraic structures

The implementation is restricted to a few algebraic structures, where suitable problems with intermediate expression swell can be found.

Definition 1.3.1 (Lipson [86]) *An Ω-algebra (also called an algebraic system) is a pair [A;Ω] where*

- *A is a set called the* carrier *of the algebra. A can be a finite or infinite set. The number of elements of A, card(A), is called the* cardinality *of A.*
- *Ω is a collection of operations defined on A. An operation $\omega \in \Omega$ of* arity *n takes operands $a_1, \ldots, a_n$ into $\omega(a_1, \ldots, a_n)$.*

The scope of this thesis is restricted to a few clearly defined algebras suitable for practical implementation and demonstration of the parallelization ideas presented. The main algebras in our computer algebra system are:

1. The *ring* of integers with the usual operations for addition, subtraction and multiplication $[\mathrm{Z};+,-,\times]$.
2. The ring of multivariate polynomials with integer coefficients and the usual operations of polynomial addition, subtraction and convolution $[\mathrm{Z}[x_1, \ldots, x_s]; +,-,*]$.
3. The ring of matrices over a ring R of integers or multivariate polynomials $[M(R);+,-,*]$. The operations $+,-,*$ are the usual matrix addition, subtraction and matrix multiplication.
4. Multivariate polynomial division by extending the integers to the field of rationals $[Q[x_1, \ldots, x_n]; +,-,*,/]$.

The parallel operations in the above algebras form the *kernel* of the computer algebra system, including support for arbitrary size integer arithmetic in parallel.

A basic linear algebra solver is an essential part of a computer algebra system. This involves matrices whose entries integers or polynomials. In order to continue to work in integral domains, the algorithms that are selected are division-free or *integer-preserving.*

Polynomial division is considered expensive as well as requiring an extension of the base carrier, therefore the last of the above algebras is introduced late in this work and mainly required for the section on Gröbner base computations.

The framework in figure 1.3 guides the development of a parallel computer algebra system that incorporates the algebraic structures discussed here.

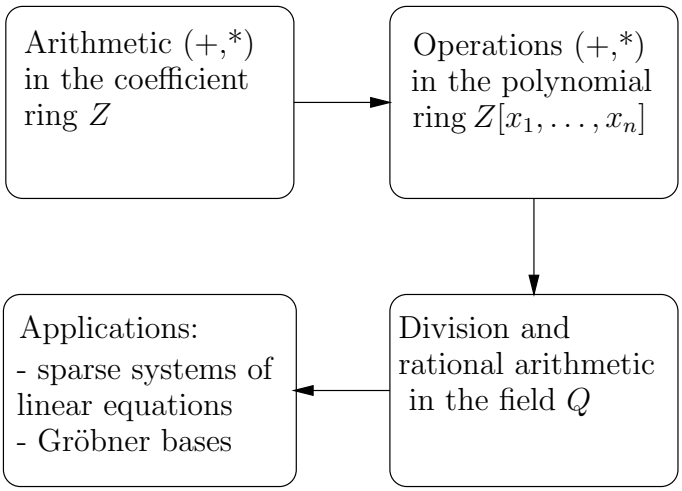


Figure 1.3: Development of a parallel computer algebra system

1.4 Outline of the dissertation

Two main objectives emerge, and these are addressed in the rest of the dissertation:

1. To devise suitable data structures for parallelizing the data from the sets of carrier ring A of the algebra.

2. To develop parallel algorithms for the algebraic operations Ω within each algebra.

The first problem is addressed in chapter 3. Distributed data structures that support efficient memory allocation are presented. These are based on a randomized hashing model which distributes data with fine granularity. A disadvantage of fine grained algorithms is that communication is increased. This is countered by a scheduler that overlaps communication and computation thus hiding the high cost of communication.

The second problem is addressed in two stages. The first stage is the parallelization of the basic arithmetic operations over arbitrary sized integers and multivariate polynomials. This is discussed in chapter 4.

Applications that build on the polynomial and integer arithmetic are then discussed in chapter 5. Firstly, algorithms for solution of *sparse* systems of linear equations are discussed. The second application addresses the Gröbner base algorithm, which is useful for solution of some systems of polynomial equations. The algorithm requires a parallel division operation thus extending the arithmetic operations.

Implementation of the system on the Hitachi SR2201 parallel computer enabled us to evaluate the feasibility of randomized global storage allocation. The conclusion in chapter 6 is that the system is suitable for solving large problems in polynomial algebra.

Chapter 2

Literature review

This chapter reviews parallel architectures and parallel algorithm design. The breadth of parallel platforms has to be understood, to place this research in context. There have been significant advances in applying parallelism to computer algebra, and the algorithms and systems for parallel computer algebra are discussed.

While several systems have been developed for shared memory parallel architectures, there are few implementations of parallel computer algebra on distribed memory massively parallel processors. Furthermore, many systems have been developed with the aim of gaining speedup, and do not fully address the memory problems that may arise when some processing elements are overloaded with data. In order to build on some of these previous results, some areas that continue to provide unique challenges for parallel computer algebra are highlighted.

2.1 Target architecture

The target architecture is the class of massively parallel processors (MPPs), with the following features:

1. Distributed memory (tightly coupled to each processor) with no global memory pool.

2. Separate address spaces so that inter-process communication is solely through sending and receiving messages.

3. Processing elements (PEs), consisting of processor and memory, are arranged in a fixed physical topology. Each edge connecting a pair of PEs consists of a fast interconnection channel with low latency.

A Hitachi SR2201 massively parallel processor was used in this research. The particular installation has 256 processors at 150Mhz and 256MB of memory per processor. The system is capable of a peak performance of 76GFLOPS and a total memory availability of 56GB.

Resource	Metric
CPU	time
Memory	space
Network	communication

Figure 2.1: Resources in a parallel system

2.1.1 Benefits of parallel implementation

Advances in processor technology governed by Moore's Law have result in the doubling of computing power every eighteen months. Workstations with a sequential architecture now have powerful processors capable of high performance on large problems.

The benefits of developing parallel implementations should therefore outweigh the simple solution of waiting for a new powerful processor and, secondly, present a strong case for climbing the steep learning curve required as programming parallel systems remains an intricate process.

Parallel solutions are justified in two main classes of problems:

1. Problems whose solution on current sequential computers takes a long time (often any algorithm with higher than quadratic complexity). For these problems the parallel methods aim to speed up the computation time by partitioning the problem and computing with the sub-problems on several processors concurrently. This class of problems is referred to as *processor bound*.

2. Problems that cannot be solved given the resources of a single sequential computer. The main culprit here is often the size of memory resources available. Advanced virtual memory techniques have been effective for some problems in capturing locality in data access. However, large irregular data access does not benefit from virtual memory. This class of problems is called *memory bound*.

Much of the research in parallel systems today focuses on the first of these problems. Frameworks similar to that developed in processor design [61] are used mainly to pursue *speedup* in the overall execution time of a system.

Attacking problems in the second category requires a carefully balanced approach to all three parallel resources shown in figure 2.1. In particular, problems in computer algebra require careful analysis of the memory requirements. Such memory bound problems are interesting in that their solution may add to the body of knowledge enabling us to solve problems that have previously been too large.

2.1.2 Challenges of parallel implementation

The design and implementation of parallel algorithms continues to be a difficult task dependent on the architecture and algorithms:

- Firstly, there is a wide and varied selection of parallel architectures. Systems developed for one architecture are not portable to another.

- Secondly, there is an inter-dependence between architecture and algorithms. Parallel systems are highly dependent on the underlying hardware and the resulting complexity in managing both hardware and software is often placed firmly on the programmer.

There are two approaches to parallel system development: bottom-up design of algorithms to suit architecture or top-down discovery of algorithms and building a machine to suit them. In the later case, hardware implementations of some parallel algorithms have been developed. For the bottom-up designs, many systems have been developed for shared memory or distributed memory machines, and some have been optimized for particular architectures such as the Cray T3D and the Hitachi SR2201.

Some great advances in parallel processing have been made despite these problems.[1] Architectural ideas have been filtered to bring about some form of convergence to two classes of shared memory and distributed memory machines. Furthermore, there are now some emerging standards in parallel programming such as the message passing interface (MPI) for distributed memory computers.

The advances in processor speed and low prices have also had a major impact on parallel computers: the emergence of *Beowulf clusters* of commodity components to form very powerful parallel machines [115] will possibly widen availability and use of parallel systems due to their substantial price/performance advantage.

2.1.3 Comparison of architectures

The development of a parallel system requires consideration of concurrent algorithms, parallel programming, and competing models for parallel systems. Parallel architectures generally fall into two classes:

- *Shared memory machines*, in which several processors share one address space. The shared memory may consist of one large common pool of memory. Hybrid systems have a large common pool in addition to some locally attached memory at each processor.

  In the case of hybrid systems, each processor has access to every other processors local memory, but there is a difference in access times due to fetching from a remote location. These are referred to as *non-uniform memory access (NUMA)* systems.

- *Distributed memory machines*, in which each processor has its own local memory and communication is only through message passing. These are also called *massively parallel processors* (MPPs).

The number of processors in a shared memory architecture is limited by the frequency of collisions when more than one processor requires access to a memory word. A high level of collisions may lead to large wait delays or deadlock, therefore any size of memory has a maximum number of processors that can be supported.

[1] It is possible that parallel systems have not delivered on the great expectation and huge promises as envisaged during the 1980's heyday.

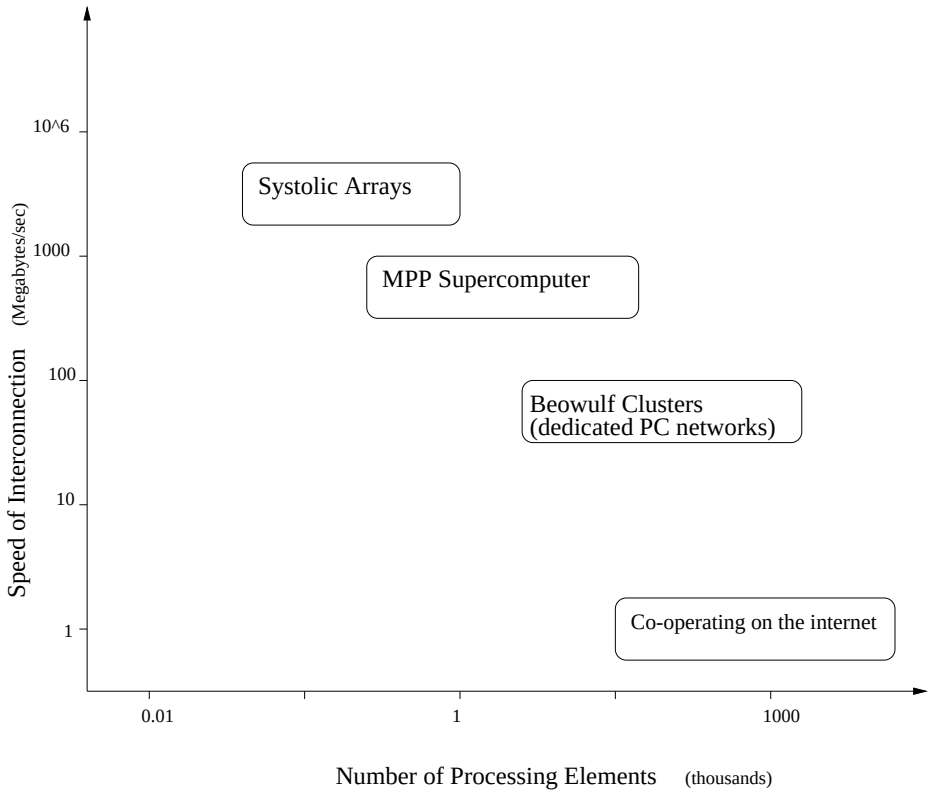


Figure 2.2: Classes of distributed memory parallel machines

In contrast, massively parallel processors may scale to a higher number of processors and larger memory due to the fact that each will have its own memory. The scale may be limited by the underlying communication network, as faster network connections can support a larger number of processors.

Several different physical realizations depending on the number of processors and the speed of the interconnection are shown in figure 2.2. The figure highlights the challenge of *portability* of systems for any distributed memory parallel system; it would be useful to have a system developed on one platform in figure 2.2 to be executable on another platform without severe performance penalty.

In this study, portability is achieved between a MPP and a development platform consisting of networked workstations. Cross-compilation between a Pentium PC and a high performance Hitachi SR2201 successfully transfers algorithms between the two platforms. In both cases the MPI message passing interface adds a level of portability.

2.1.4 Parallel system components

Parallel systems can be viewed as several layers shown in figure 2.3, with the application level software still very highly dependent on the underlying architecture. This dependence points to a property of the intermediate layers that may be called *thin*; the application layer is not completely shielded by any of the interim layers.

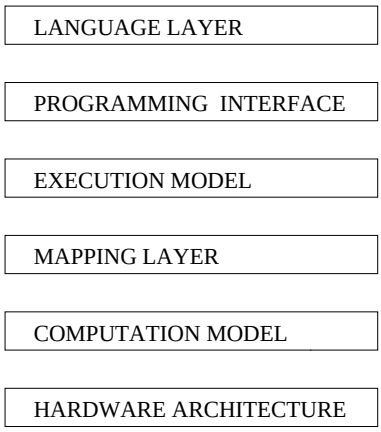


Figure 2.3: Layering of parallel components

Contrast this with sequential processing where large systems with layers provide *abstraction*, that is, the higher layers are shielded from the complexities of the underlying system.

The concept of *layer* organisation for large systems has been closely associated with *modularization* as tools for viewing large complex systems. These concepts have been successfully applied in areas such as digital communications to create the layered TCP/IP architecture. The layers are software modules that provide greater abstraction to the higher layers, shielding users of higher layer services from the details of any lower layers.

Parallel systems are different in that many algorithms are highly dependent on the particular architecture. In addition, services such as communication between two processors (and in some cases, whether these two processors can communicate at all) are dependent on the underlying topology. Thus robust parallel systems will rarely benefit from hidden layers given a top-down view of the system hierarchy.

A different approach is to provide parallel algorithms with an additional view: *slicing*. A slice is defined as a system view that reveals all the layers at some time t, and gives the option of selecting one to see in more detail. Therefore parallel system developers are given two views of the parallel system allowing them to adapt algorithms accordingly.

A useful tool is that the communication middleware message passing interface (MPI) [41] can be installed on supercomputers and PC clusters therefore providing the desired level of portability. In this thesis algorithms developed for supercomputing environment (Hitachi SR2201) have also been executed on a cluster of personal computers. This is effective for algorithms that are sufficiently general and not sensitive to network latency. However, we will show that for many of the computer algebra data structures that are discussed in this thesis, the fine granularity is highly sensitive to the communication latency therefore the performance penalty on a cluster with a slow communication network can be high.

2.1.5 Parallel programming models

The message passing model defines primitives that allow the programmer to describe the parallel computation and determine the communication and computation schedule. The complexity of parallel system design lies in the requirement for the developer to manage all interactions within a system with potentially thousands of processors and many millions of communications.

Some areas of concurrency may be identified and exploited with little additional complexity for the developer. For example, Revesz [102] demonstrates the power of implicit parallelism within functional programming, by reducing lambda terms in parallel with different processors searching different branches of an expression graph. Termination is determined when one of the processors reaches a normal form of the expression.

The programming model determines an overall picture that a programmer has about the system. The Flynn taxonomy [45] is widely accepted for describing the different modes of computation:

SISD Single Instruction Stream Single Data Stream.

SIMD Single Instruction Stream Multiple Data Stream.

MISD Multiple Instruction Stream Single Data Stream.

MIMD Multiple Instruction Stream Multiple Data Stream.

The Hitachi SR2201 target architecture presents a programming model based on MIMD in which each PE executes instructions taking a different path through the program based on its unique identifier. The same program is distributed initially to all participating PEs.

2.1.6 Parallel complexity models

Abstract models of computation are useful aids in understanding of algorithm design and assist comparison of algorithms [2]. Several models for sequential machines (Turing machines, register machines, lambda calculus) have been proven to be equally powerful.

The large number of different parallel machines make it difficult to obtain a universally accepted model for parallel systems. This section briefly discusses the parallel random access machine (PRAM) model.

The PRAM consists of a combination of random access machines (RAMs). In a model used successfully in [49] each of the component RAMs is an arithmetic processor and there is a synchronization primitive for communication between them. Computation is initiated by a single processor called the *root*, and data is distributed through task splitting with synchronous communication.

The standard PRAM requires shared memory between processors. A modified model is the LPRAM model [1] which describes a non-uniform memory access (NUMA) computer consisting of:

- Unlimited number of processors

- A global memory pool accessible to all processors
- Each processor has local, separately addressable memory
- A processor can have at most one communication request outstanding at anytime
- A global synchronizing clock where each time step is either a computation or communication step.

Although the LPRAM has local memory similar to MPPs, the model also has a pool of shared address space therefore it cannot be used directly to model a MPP.

At the programming language level the LPRAM a language may differentiate between local (private) variables and global (shared) variables. The distinct address spaces in a distributed memory massively parallel processor are not encompassed by LPRAM. A step towards providing separate address spaces is found in the work by Roch and Villard [104]. They LPRAM at the language level by introducing a language for asynchronous task handling (ATH) that has a greater restrictions between local and global access. Since the change is at the language level, the underlying architecture still has the LPRAM global memory pool. However, ATH provides local and global memory access with the following features:

- Global memory access has cumulative concurrent read, concurrent write (CRCW) semantics.
- Size of elementary instruction is enlarged to a *block* sequence of instructions. The block is executed sequentially with local access.

The extended model becomes very close to distributed memory as the language semantics CRCW can be mapped to a distributed memory architecture; the concurrent writes may be considered as providing each processor with local memory. The important step of increasing unit grain to several instructions becomes important in compensating for delays introduced by communication. However, the changes at the semantic level are a step towards distributed memory, but the model still allows shared address space therefore it does not fully describe our system.

2.2 Performance metrics

This section describes the performance metrics for evaluating performance of parallel systems. It poses a slight modification of the question presented by Aho *et al.* [2]: how can we compare a parallel algorithm with another that solves the same problem and how do we judge the performance of systems that incorporate these algorithms? Two levels of system evaluation are considered:

1. Empirical evaluation where tests are conducted on the Hitachi SR2201 to provide some empirical measurements of the system.
2. Analytical evaluation which involves analysis of algorithms within a parallel model and comparison based on asymptotic complexity of the algorithm.

2.2.1 Empirical analysis of parallel systems

Performance analysis is based on the principle that the key resources in any system are valuable. It may seem counter-intuitive to consider a parallel system as having *limited* resources. However, this serves to emphasise that a parallel system increases the actual available processing time or memory, but inefficient use of any of these has a negative impact on the performance of the system.

Let n be the size of a problem (for example, the number of terms in a polynomial, or the degree of a polynomial). In a parallel system, the data for any problem is distributed across several processing elements, therefore n is the overall size of the problem, and we let N be the size of the partitioned problem on each processor. For simplicity let $n = N \times p$. Then we may define functions T, S by:

$$T(n) = \text{time}$$

$$S(n) = \text{space}$$

where the time $T(n)$ is a measurement of the elapsed time for computation with data of size n and $S(n)$ measures the amount of memory during execution of the program. In analysis of parallel systems, a third resource is measured: *communication bandwidth*. Communication between processors may be through shared memory, in which case the bandwidth measures how many processors may access the data.

$$C(n) = \text{communication time}$$

In general, message-passing systems have a slow channel between processors leading to a bottleneck. Moreover, the network is not fully connected therefore communicating between processors will involve intermediate forwarding nodes.

2.2.2 Communication

Communication cost has to account for both latency and bandwidth. Communication latency may depend on the synchronization. In synchronous algorithms, latency includes the transfer time for message from one processor to another, while in asynchronous communication the latency is equal to local buffer setup time. In synchronized algorithms, the communication is in phase, therefore the wait time for any processor that has to wait for another to synchronize is high. Processor requiring data from an asynchronous connection may have perform other functions while the data is not available, and only wait when the scheduler determines that there are no other local computations to be performed.

The communication patterns of algorithms differ and may be classified [39] as uniform, bursty or periodic. Such a classification is a useful tool enabling allocation of bandwidth depending on the the projected communication power spectrum. This may be useful in lowering communication cost.

Communication within a message passing system is between *source* and *destination* nodes. Several patterns emerge depending on the numbers of source or destination nodes involved in a communication:

- The *one-to-one* communication between one source and one destination is the basic form of communication. Let the time to send a message from a source to a destination be given by C_{1-1}.

$$C_{1-1} = T_{transfer} + T_{delay}$$

where the *transfer time* is determined by the number of edges (hops) from the source to the destination on a given route, and by the time to send a message between two directly connected nodes with an edge between them.

$$T_{transfer} = \text{No. of hops } * \text{ Latency}$$

The delay time incorporates time spent at the source arranging message packets and initiating transfer, and also the time for assembling at the destination receiving messages.

$$T_{delay} = T_{setup} + T_{recv}$$

In most systems the delay time can be kept to a small constant and the latency is determined by the speed of the interconnection network selected. Therefore the communication time depends critically on the number of hops between the source and destination. The topology of the interconnection network and the routing algorithm determine the number of hops.

- A *multicast* involves one source and the destination nodes are a subset of all nodes.

- The *broadcast* of a message requires that every node receive the message. A broadcast may originate at one source *(one-to-all)* or may have all nodes as a source and destination *(all-to-all)*. The broadcast time C_{br} is the time elapsed from the beginning of a send until the last node has received a copy of the message. Randomized broadcast algorithms have been developed for radio networks [84] and poly-logarithmic algorithms for the number of hops required are available.

The total communication time within a system will be accumulated from the time taken for several broadcast or point-to-point messages within an algorithm.

Reduction of communication time in a parallel application leads to efficient use of CPU time and therefore improved speedup for parallel implementations. The goal in this section is to consider communication time at the application level, which can be broken down into several constituents. The constituents are placed in a layer as shown in figure 2.4. Each layer influences the communication patterns available and contributes to the overall communication time. In most systems the mapping of application communication graph to the underlying network, the message passing and the assignment of unique identifiers to the processors is performed by one subsystem.

Our model for communication is based on specification of the communication graph for an application and the ensuing network traffic. The model interacts with

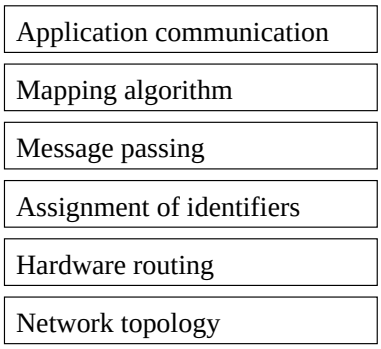


Figure 2.4: Communication layers

MPI which provides the facilities for the lower layers of mapping and message passing. If the communication pattern closely matches one of the known topologies it can be efficiently mapped onto the underlying network, otherwise communication proceeds in a partitioning provided by the message passing interface. For example one of the widely used patterns in our system is a master-slave pattern which can be closely mapped to a star network topology. However the target Hitachi SR2201 is a three dimensional torus therefore the message passing interface has to provide a suitable close match.

The communication time is related to the following parameters:

- The granularity of the problem.
- The message size.
- Average network traffic.

The granularity of the problem determines the level of detail for sequential time analysis. This means that either the computation on all local data in the partition size is used before any communication occurs, or the time is subdivided into smaller grain size computations. Fine granularity has more communication steps and therefore larger communication overhead than a coarse grained computation.

Let N_{comm} be the count on the number of communication messages sent (this is the number of messages for the particular granularity of the problem). Let C_{comm} be the time for each communication. The model for communication time is given by:

$$C(n) = N_{comm} \times C_{comm}$$

where

$$C_{comm} = C_{setup} + C_{transfer} + C_{recv}$$

The setup time C_{setup} is the time for initiating communication at the sender side. This includes packing data, preparing a message envelope and initiating communication protocol.

The transfer time in sending data of size N_j from a processor j to k is determined by the channel characteristics and the topology of the network:

$$C_{transfer} = \max(N_j \times \text{bandwidth}_j) + (\text{delay}_j \times \text{Number of hops}) \quad (2.1)$$

Network topology

In a distributed memory system, there is great flexibility in how the network is organized. The differences in *topology* affect the communication times between any two processors. Some widely used topologies are ring, mesh and cube [119, 99].

Hardware routing refers to the selection of a path for sending a message between a pair of processors in a parallel system. This is highly dependent on the topology of the system. The Hitachi SR2201 provides some hardware based routing mechanisms for the most common typed of communication: one-to-one and broadcast. In addition to the hardware based routing, a message passing library provides other routing algorithms which may be deterministic or adaptive.

Deterministic algorithms select a single path for any communication. In communication networks such as the mesh and crossbar, the most widely implemented deterministic algorithm is XY routing which sends messages to the right set in another dimension. The Hitachi SR2201 enhances XY routing with failure modes and deadlock prevention making it a simple and effective communication routing.

Adaptive algorithms may take into account such factors as congestion on a particular channel to select an alternative route with better throughput. However, these are more complex and prone to implementation error.

2.2.3 Space

The memory model determines the use of local memory on each PE and the dynamic relocation with global memory use. Clearly the total amount of memory available for an application is the sum of all local memory

$$S(n) = \sum_{i=1}^{p} S_i \quad (2.2)$$

where the size of memory per node, S_i, is expected to be a reasonable size for current commercial systems ($\approx$ 64 to 128MB, and continuing to rise). For 32bit CPU architectures a ceiling of addressable space by a single processor is approximately 4GB. Thus any computation that requires more than 4GB of main memory cannot be performed on a single workstation or a shared memory computer. Given these considerations, the distributed memory model is the most suitable for our computer algebra computations with large memory demands. The contribution of local memory use to the total system load is determined by the global data distribution algorithm.

2.2.4 Time

The CPU time in a parallel algorithm is the time that each PE is performing calculations. This may be delayed if some of the operands have not been received. The

total execution time for a problem T, is given by the execution time on the slowest processor:

$$T(n) = \max_i(T_i)\ 0 \leq i \leq p$$

This leads directly to the need for load balancing. A low standard deviation in T_i indicates good load balancing and efficient processor use.

The effect of optimal load balancing is often to spread tasks among all available processors, thus reducing memory locality. Similarly, optimizing for memory locality will often starve some processors by clustering work on a few processors. Thus achieving good time and space balance is a challenge [60].

Several researchers in parallel algorithm analysis consider the time and communication complexity of algorithms [1, 95] where the memory model is fixed.

2.2.5 Parameters for parallel algorithms

Parallel applications incur overheads adding to the total computation time. Several parameters may be used to classify and compare parallel implementations.

Granularity The algorithms implemented in this research are mainly fine grain algorithms. This has an impact on the other areas of the system such as communication time.

Synchronization The communication system provides both synchronous and asynchronous interfaces. Synchronization may be expensive in a system with some load imbalance, therefore the choice for each communication is important.

Scheduling The scheduler determines the overlap of communication and computation within an application. An efficient schedule overlaps the computation and communication as far as possible to hide the communication delay. This is difficult to achieve therefore several schedules may be available for selection.

Load balancing To improve performance of the parallel system, it is essential to minimize the fraction of time that the CPU is idle. The load on each processor is determined by the data partitioning algorithm that provides local work on each PE.

Speedup Speedup measures the comparative improvement of the parallel implementation compared to a good sequential system.

2.2.6 Granularity

Granularity refers to the level of size of each computation step within a concurrent computation. This corresponds to the abstract definition in general system design, and often depends on the way in which the problem naturally partitions itself into distinct operations. Two levels are defined: *fine grain* and *coarse grain* parallel processes. These give an imprecise but useful classification of problems and their solution.

Coarse Coarse grained parallelism results in a system whose characteristics are similar to those of a distributed system. The emphasis is placed on *minimizing communication* as communication cost for a large message is high. This requires finding a *data distribution* algorithm that identifies a good level for process size. A good coarse grained algorithm achieves high speedup.

Fine A fine grained parallel algorithm has a small task size. Sending a small message between two PEs can be achieved with low latency. However the communications are more frequent therefore the overall system often has a large number of communications leading to high overhead. The fine granularity also necessitates *good synchronization* between the processes. Speedup is adversely affected but completion of the problem may be the main aim.

We will often quantify granularity in terms of the instructions executed in each process, for example a task that executes in 10 ms could be considered 'more fine grained' than division into tasks of 100 ms. A granularity measurements tells us how sensitive the system is to the impact of a single data item. Thus a single term in a fine grain system will invoke a distribution function, while such a term in a coarse grain system is captured within some block without much overhead.

The quantitative measurement of granularity computes the average *time to reach an essential communication* with another process. This gives a unit amount of useful work. Then the granularity $g(n)$ of an algorithm is given by the ratio of CPU time $T(n)$ and communication times $C(n)$ [104]:

$$g(n) = \frac{T(n)}{C(n)}$$

Figure 2.5 shows the effect of granularity with block distribution of data where $n/p = 4$. In a fine grain computation, there is shorter computation time before a necessary communication while a coarse grain algorithm will seek to minimize communication and distribute data accordingly.

Fine grained systems place an emphasis on *collaboration* and *coordination* whereas in coarse grained parallelism the emphasis is on *division of labour*. Some algorithms have inherent coarse parallelism[2] and can be implemented in parallel on a loosely coupled distributed network of workstations. A collating process *farms out* the computation to numerous *worker* processors and collects the results. This method of parallel implementation has been successfully used for problems such as partial evaluation [114] and large integer factorization [15] where the successful factorization of a large integer was done over the Internet by volunteers running some parts of the computation on their PCs when not in use. Coarse granularity gives more stability to the system since we consider an entire block of instructions and data items. Fine granularity has the advantage of greater flexibility.

Granularity has an inverse relation to communication cost therefore, reducing grain size increases communication [105]. The algorithms discussed here have very fine grained parallelism hence substantial communication. Two methods may be used to contain the effects of communication: The first analyses the communication

[2]These are sometimes referred to as *embarrassingly parallel* algorithms [15, page 10].

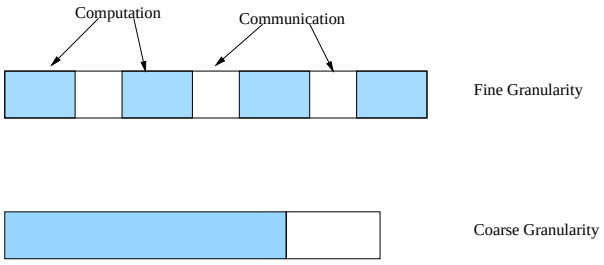


Figure 2.5: Partitioning of data with different granularity

patterns of the algorithms and uses this information to inform decisions about the details of the algorithms to reduce bottlenecks. The second option is to design algorithms that overlap communication and computation thus masking the effects of communication.

2.2.7 Synchronization

Specifying the communication pattern of a system with several processes executing concurrently is quite difficult. We require minimal communication time, deadlock prevention, and fault tolerant communication. Some widely used communication patterns are:

Master-Slave In this case one processor is designated a master process and distributes work to all slave processes.

Linear Array Processes may have local communication with a subset of the available nodes in a *nearest neighbour* pattern.

Tree Communication takes place in synchronized steps with each parent process communicating only with its child processes in a communication tree.

These communication patterns are supported by synchronous or asynchronous structure. Further performance optimization in synchronous communication involves resolution of resource contentions using mutual exclusion models [122]. These optimizations may be extended to distributed memory architectures to give a performance model that incorporates synchronization.

2.2.8 Scheduling

Scheduling for heavyweight processes in our parallel computer algebra system has some interesting differences to other scheduling procedures. Firstly the system has probabilistic distribution which creates some uncertainty regarding loads at any time t. The inherent uncertainty affects the scheduling policy since most tasks will be conditional on the current state.

One possibility for scheduling in such a system is to allocate lightweight threads for each possible branch. This is the approach taken in [25, 4]. It increases the level of parallelism as well as simplifying the scheduling since the scheduler would consist

of 'fork' and 'join' commands for thread creation. However, the Hitachi SR2201 does not provide lightweight threads and imposes a one-to-one relation between process and processor. Each process executes on a single physical processor (this allows us to use *process* and *processor* interchangeably). The scheduler for our parallel computer algebra system has the following properties:

- The scheduler does not make use of threads therefore there are a fixed number of processes and any dynamic increase in computation requirements cannot be assigned to a newly created thread. It therefore relies on redistribution and manipulation of granularity to balance the computation, communication and storage requirements of the system.
- The scheduling is based on a model for the underlying architecture. Schedules based on abstract models [100] have been discussed and shown to be adaptable to real architectures. Adaptive scheduling is a key technique in some applications [22], however it has not been widely used in parallel computer algebra.
- The scheduling of computation blocks has to minimize the overall completion time for the algorithm. A schedule is dependent on several precedence constraints on the execution of blocks. Some operation p_i cannot proceed before p_j completes. The scheduling problem can therefore be considered as an optimization problem to minimize the execution time given certain constraints on the order of block computation. A branching procedure [34] that generates a search tree with minimal footprint makes efficient use of memory.

The probabilistic function for determining target communication means that the communication pattern cannot be determined at compile time. The irregularity in the communication patterns requires dynamic scheduling of communication and is more expensive. The system generates good approximations to the optimal solution and in practice these methods are very effective.

2.2.9 Load balancing

Load balancing refers to measurement of the efficiency of the computation distribution across all processors. There are two main strategies for load balancing: *static* and *dynamic*.

Static load balancing

Static load balancing predicts the computation and communication patterns of an application and determines the *data distribution* strategy before computation begins. The most widely used data distribution strategies are:

1. **Block distribution** The data is divided into blocks of the same size m. If the problem have data size n, and is to be distributed on a system with p processors, then
$$m = \frac{n}{p}$$

The block distribution has the advantage of simplicity and for problems that have regular data access patterns and good spatial locality, this is an effective distribution scheme.

2. **Cyclic distribution** For applications that have weaker locality constraints and access most of the data sets, a cyclic distribution can create smaller blocks and assign them to processors in a cycle of period p.

3. **Block cyclic** Combines the advantages of block and cyclic distributions by dividing data into blocks and having cyclic assignment within each block.

Application will often be tested with all the different distribution strategies to determine their performance and the best strategy.

Dynamic load balancing

Static load balancing requires some knowledge of the behaviour of the application and is inflexible once the computation begins. For irregular parallel applications where prediction is difficult, we would like to make run-time decisions relieving any processor that has fewer resources. There are three approaches:

- Redistribute the data.
- Migrate activities across available nodes.
- Random relocation.

2.2.10 Speedup

Speedup arises from improved performance due to parallel implementation of an algorithm. It is defined by Hennessy and Patterson [61] as follows:

$$\text{speedup} = \frac{\text{total execution time without enhancement}}{\text{total execution time using enhancement when possible}}$$

Now let T_p be the time to perform the task in parallel. This is expected to be much less than the time T_s to perform the same task on a single processor sequentially. Then

$$\text{speedup} = \frac{T_s}{T_p}$$

Efficiency ξ measures the overhead incurred in moving the computation from sequential to parallel:

$$\xi = \frac{\text{speedup}}{\text{no. of processors}} = \frac{T_s}{T_p \times p}$$

A computation that can be parallelized perfectly has efficiency 1, but in practice $0 \leq \xi \leq 1$.

2.3 Algorithms for data partitioning

This section briefly examines how to make use of the conventional partitioning technique of 'blocking' to enhance the new dynamic randomized partitioning of this chapter.

The partitioning algorithm distributes data to PEs where space is available. The data allocated to each PE forms local data, and the expectation is that each retrieval will encounter data local to that processor.

The most widely used data partitioning algorithms are block algorithms that divide the data into several blocks of size b. The block size is based on the granularity of the problem, with more coarse grain algorithms having larger block sizes of several terms or polynomials, and fine grain algorithms requiring small block size of a few terms. Several options are available when distributing polynomials:

- The obvious distribution of several polynomials is to keep all the terms of each polynomial on the same PE. Data is therefore partitioned along polynomial lines: given a set of polynomials $(p_1, \ldots, p_k)$, each polynomial is assigned to a processor.

  In this way each polynomial is easily identified and polynomial operations can proceed by sending one complete polynomial to another PE for computation.

  This scheme clearly has its benefits but it also leads to inefficient use of the PEs since some processors will have to be idle while others compute (we cannot duplicate computations on all processors). In addition, if the number of polynomials is less than the number of processors, then some processors will not have any data at all!

- To alleviate the possibility of some processors not having any data at all, we may partition the data at a finer granularity by creating blocks of individual terms within a polynomial.

  For example, in figure 2.6 the first block is stored on PE_0 and the second block on PE_1. This method will work well for dense polynomials or matrices. For sparse polynomials as shown in this case, the polynomials have terms with the same exponent vector but the terms $2x^2y^2$ and $5x^2y^2$ are kept on different processors.

  In general, a block distribution partitions data into p blocks of equal size n/p where p is the number of processors and n is the number of terms. As the block partition is based on the order of terms, the mismatch of alignment is a frequent occurrence and the resulting duplication of exponent vectors on different PEs leads to poor performance.

- In order to keep related terms on the same processor we shall have to look at the degree of each term and create blocks based on the degree. However the block structure cannot be given strict ranges assigned to different processors because then it will be possible to have some processors with no terms in their degree range. This has a negative impact on the performance of polynomial addition which is the most frequent operation.

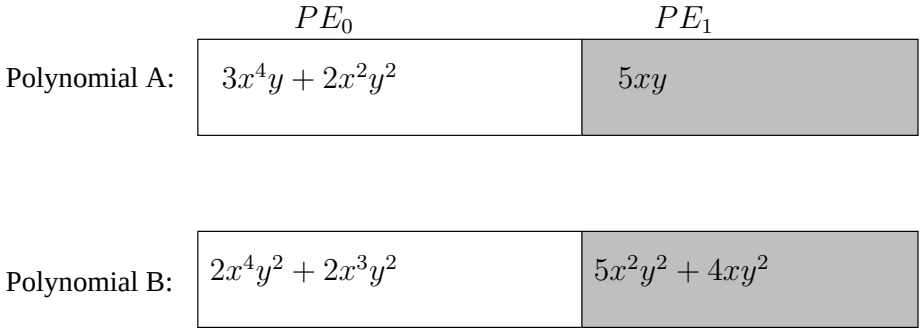


Figure 2.6: Block distribution

The restrictions on block structure and size motivate an approach based on random selection to provide good load balancing while maintaining some data locality. This relies on a distributed hash table data structure to provide global support for data storage. Two levels of hashing are introduced: the first hash function determines the processor and the second hash function determines the placement within the local processor hash table. The algorithms for randomized and probabilistic distribution are described in chapter 3.

2.4 Computer algebra systems

The approach that is adopted in implementing a parallel computer algebra system is of a bottom-up design, starting with the internal representation and data structures that incorporate an efficient memory model. While the interaction between modules in the design support good memory efficiency, it appears that the interfaces are similar to other parallel computer algebra systems such as the extended Maple implemented by Bernadin [12]. In this section the similarities between our approach and other parallel computer algebra platforms are explored, and the differences are highlighted.

2.4.1 Approaches to parallelization

Several algorithms and systems for parallel computer algebra have been developed with different requirements and for competing parallel architectures:

- Some systems add parallel primitives for communication and cooperation to existing computer algebra systems. This has the benefit of lowering development costs and reusing robust implementations. As Fitch [44] notes, algebra systems "represent a large investment of intellectual effort which we can ill afford to lose".

- A large number of parallel computer algebra systems have been developed for shared memory architectures.

- Computer algebra hardware based on very large scale integration (VLSI) technology has been developed. The use of pipelining systolic architectures has also received some attention.

- Distributed systems for loosely coupled PC networks or parallelism across the Internet have been developed.

- Distributed memory systems based on standard communication middleware.

This work is based on distributed memory with standard message passing middleware. It addresses a range of parallel architectures from MPPs to Beowulf clusters.

2.4.2 System design

In many ways the design and implementation of a robust, reliable, efficient and scalable computer algebra system relies on the same principles as other large systems; computer algebra systems are large and highly sophisticated systems incorporating a language, graphical user interface and other components. System design methodologies such as modularity and abstraction as used in other complex systems such as operating systems [8], become equally useful and valid in the design of a computer algebra system. On the other hand, some development problems are more specific to the design of computer algebra systems:

1. The representation of basic objects in a computer algebra systems as integer or polynomial data structures has direct impact on the algorithms. Whether data is kept ordered or unordered also affects the system.

2. Domain specification. Computer algebra systems have to manage a highly developed algebraic theory with a well defined structure.

3. Intermediate expression swell. Many computer algebra algorithms require large memory working space even when the result is of modest size.

4. Irregular data [59]. Symbolic computation algorithms are highly dependent on irregular data, therefore making it difficult to find a useful partitioning of data at compile time.

5. The complexity of some algebraic calculations limits the estimation of resource requirements, making the task of parallel resource allocation more difficult.

2.4.3 PACLIB

Systems that extend existing sequential uniprocessor CAS have been developed by providing a supporting interface for multiprocessor systems. PACLIB [64, 65] is a parallel computer algebra system that extends the general purpose algebra system *Saclib* with lightweight processes resulting in a general parallel CAS. The main features of the parallel interface include:

- *Concurrent tasks* at multiple levels and each program function can be executed as a separate task.
- *Shared memory communication* through access to shared data. Task arguments and results are passed by reference therefore avoiding expensive copying of data structures. However, this requires careful mutual exclusion using semaphores, and tasks waiting for data will block.
- *Non-determinism* and *task termination*. A task may wait for the results from several other computing tasks. The waiting task will resume computation when the first result is delivered. If delivery of one result renders some computing tasks superfluous, they may be explicitly terminated.
- *Streams* and *task pipelines*. Tasks may be connected by streams of data forming a conceptual pipeline. These provide an additional communication mechanism through a stream of data $\langle s_1, \ldots, s_n \rangle$. Each stream is a pair (f, rs) where f is the data element to work on and rs is another stream for use in sending to the next stage.
- *Parallel garbage collection* based on the 'mark-and-scan' method is also used in Saclib. PACLIB provides a non-uniform memory system comprised of a local allocation list (LAVAIL) and a global allocation list (GAVAIL). Memory is allocated first from LAVAIL and if there are no free cells then data is placed in a cell on the GAVAIL list. This combines the advantages of shared memory with the availability of of distributed memory. The garbage collector is triggered if no cell is found on either LAVAIL or GAVAIL.

The PACLIB memory system is an interesting hybrid pointing to gains in combining different systems. However, the limitation of shared memory in terms of scalability for very large problems will persist.

2.4.4 PARSAC

Kuechlin [81] introduced symbolic threads called *s-threads* for parallel symbolic computation on a shared-memory processor. These special lightweight processes are used in parallelizing the SAC-2 computer algebra system [29] resulting in the PARSAC-2 parallel system [80]. The s-thread environment makes efficient use of system resources. It has been used to gain significant performance improvements for application such as Gröbner base computations [4].

A disadvantage of s-threads is that when a thread terminates, its working space cannot automatically be deallocated without checking all references to it. This system also uses shared memory.

2.4.5 Other parallel computer algebra systems

One view of developing parallel computer algebra systems is to exploit instruction level parallelism. This approach relates the pipelining principles for hardware design [61] to the development of highly optimized *computer algebra processors*. The

physical implementation in VLSI technology is the subject of studies by Smit [112] where the model for a MIMD computer with large memory and limited number of processors is used. The system was considered well-suited for memory intensive computer algebra problems. This is encouraging for our present work targeted at a MPP; when considered in the context of the overlapping architecture sets in figure 2.2, this raises the prospect of portability of algorithms across VLSI implementations and MPP implementation provided the memory model is sufficiently robust.

A different approach for hardware development was pursued in the work of Buchberger [21] with the design for the L-machine which was implemented in TTL logic rather than VLSI. The system consists of a reconfigurable assembly of processor, memory, an open/close bus switch, and a 'sensor bit'. The sensor bit is used for access rights to the shared memory and for synchronization. The processors can be configured and extended for different purposes. A special L-language is provided for parallel programming of computer algebra algorithms. The L-machine is more suited to shared memory architectures, which makes it different from our system.

Systolic systems

Jebelean [67, 68] has developed systolic algorithms for symbolic integer arithmetic. Multiprecision integers are represented with the 'least significant digit first' positional order. This leads to algorithms that exploit the data flow network of the systolic architecture and give significant improvements in performance. In considering portability of these systolic algorithms to MPPs, it is clear that such an embedding would significantly increase the number of algorithms available on massively parallel processors. However, enforcing a data flow network would be quite expensive therefore our work considers implementations based on the classical algorithms no restrictions on flow of data between any two processors.

Maple/Linda extensions

In developing completely new systems, only a few algorithms are tested, making the systems too small for most practical uses. In addition, users will not have to learn a new system. An alternative approach is to add parallelism to an established system with a large number of implemented algorithms. The addition of parallel primitives to an established computer algebra system such as Maple and has the advantage of having access to a huge wealth of algorithms already implemented.

Watt [123] presents a parallel version of the Maple computer algebra system on a distributed architecture network of workstations. This is a message-passing system with primitives *spawn* and *kill* for creating and terminating processes consecutively. The communication is provided by procedures for *send*, *receive* and *reply*. The system is transparent to the user program therefore easier to program. Further work in parallelising Maple has been performed and algorithms implemented on the systems [96] for indefinite summation on rational functions. Another system based on this approach is the Sugarbush system [26].

Bernadin [12] has implemented message passing interface to port Maple to the Intel Paragon MPP, with gains in speedup. A master-slave approach to distributed

scheduling is used to maintain a single node access in interactive use of Maple. A similar approach is implemented in the STAR/MPI system [30] where the interactive use is given special attention. STAR/MPI is available for at least two symbolic algebra systems (GNU Common LISP and GAP) with these can be replaced by other interactive languages.

The master/slave approach is limited by having only one node as the master throughout a computation. In distributing fine grain computations, this often leads to accumulation of *wait time* at the slave nodes. We will propose *dynamic priorities* as a way of revolving the *master* label where any PE (but only one at a time) can become master. This enables the system to control the allocation of tasks dynamically.

There are some similarities between the `nxcall()` feature for integrating the Maple library with the communication library in Bernadin's system, and our MPI conversions to integrate timing and synchronization data.

LISP extensions

Marti and Fitch [87] extended the LISP language with a tool for automatic identification of concurrency. The system accepts a LISP program for a computer algebra algorithm, analyses it for available concurrency and generates a program for parallel execution on a multiprocessor comprising a network of workstations. The power of this system lies in the data flow analyser which can be used to analyze a complete symbolic algebra system such as REDUCE [44] and identify areas that can be parallelized.

A different LISP extension is given by Halstead [59] where explicit concurrency primitives *future* and *delay* are provided. The *future* primitive gives data-flow parallelism with demand-driven [108] evaluation.[3] This creates a new concurrent task and every parent task waits for a *future* to resolve. The concept of a *future* is carried to a distributed network of workstations [107] with implementation of algebraic algorithms.

DSC and FoxBox

The DSC system [37] for distributed symbolic computation is designed for symbolic computation on heterogeneous network of workstations and across the Internet. The system has a master scheduler to distribute tasks to workstations based on the available resources on each and determined by some threshold conditions. DSC has successfully been used in algebraic computing with 'titanic integers' (having more than 10^3 decimal digits) and for solution of sparse systems of linear equations [36]. DSC can support computations with 'black box' representations at coarse granularity.

The FoxBox system is designed for computation with objects in black box representation [38]. The system provides for distributed computing with a client/server

[3]Demand-driven evaluation is also called *lazy evaluation* in functional programming terminology, and refers to a scheme where an expression is not evaluated until it is required by some parent node within the graph.

interface to other computer algebra systems. Parallel computation utilizes the functions *distribute, wait, kill* which are provided and compliant with the MPI standard.

2.4.6 Automatic parallelization

The generic attribute *concurrency* describes algorithms whose execution overlaps in time. Concurrency can be exploited in different forms, however, not all algorithms benefit from parallel implementation. Some algorithms are inherently sequential, while others incur large overheads when parallelized therefore cancelling out the benefits. It is necessary to avoid wasting resources in pursuing parallelization of inherently sequential process, and one approach is to use automatic tools to detect concurrency.

Instruction level parallelism as used in processor design [61] is identified automatically by many compilers. Extension of the LISP language to a parallel implementation [87] based on an automatic tool has been discussed. Other work in this area includes the emerging High Performance Fortran (HPF) compilers for parallelization of numerical algorithms.

An interesting tool by Fahringer and Scholz [43] uses symbolic evaluation to determine concurrency. This seems to point to the interesting situation of a computer algebra system used to identify concurrency in another computer algebra algorithm.

Such tools are not yet available for large systems, therefore in building systems we make specification decisions about which algorithms to pursue. In particular, the following sources of concurrency are identified:

- Determine mutual exclusion between procedures [87].
- Identify recursion on data structures such as trees, lists and sets [59].

In terms of computer algebra problems, Smit [112] identified two categories of algorithms that benefit from a *memory intensive partition* for parallel processing:

1. Algorithms that depend on identification of like terms, for example polynomial addition.
2. Knowledge based algorithms such as symbolic differentiation and Gröbner base computations.

This work addresses problems in the first category. Polynomial addition and multiplication are parallelized. Other applications are then based on the underlying parallel polynomial arithmetic.

2.5 Parallel arithmetic

Polynomial and integer arithmetic accounts for a large fraction of the computation time. In Gröbner base computations, Jebelean [67] found that multiprecision integer multiplication accounts for 62% to 99% of the CPU time. Therefore efficient implementation of these algorithms is essential.

2.5.1 Multiprecision integer arithmetic

Parallel multiprecision integer arithmetic is a significant part of large calculations such as factorization [15] and Gröbner base computation.

Fine grain arithmetic with integer coefficients can be inefficient for univariate polynomials with small integer coefficients [12]. However our system for multiprecision coefficients in large multivariate polynomials the need for performance improvements in integer arithmetic becomes more acute. Therefore special attention is paid to integer arithmetic in this work.

Many parallel algorithms are developed for parallel arithmetic in finite fields and there is an implementation on a MPP [109]. Modular integer multiplication [35] and exponentiation [91] on parallel architectures have been implemented. These are quite different from the basic integer arithmetic in our system.

Systems for multivariate integer arithmetic on distributed memory machines [103] and distributed computation on the Internet [15] share the essential characteristics that are needed for our implementation.

The greatest common divisor is an important operation on both integers and polynomials. Parallel integer GCD algorithms based on the Euclidean algorithm [74] have been developed and they have sub-linear time complexity. Other algorithms [90] use Fast Fourier Transform (FFT) methods.

Standard representation

The standard representation of a radix β integer takes the form of a linked list of coefficients a_i. An A has the following representation:

$$A = \sum_{i=0}^{n} a_i \beta^i$$

This standard representation has been effectively used in multiplication of unsigned numbers on shared-memory architectures [82]. The PARSAC-2 system gains parallel speedup through creating multiple threads in implementing Karatsuba multiplication and FFT-based methods.

For systolic architecture, the standard representation has been used in a 'least significant digit first (LSD)' representation suitable for systolic parallel arithmetic [67]. Related work by Brent and Kung [16] describes a parallel adder with a lookahead. This is based on generate-propagate adders, and can be extended to multiple precision through pipelining of the single adders.

Signed digit representation

The signed digit representation by Avizienis [5] is suitable for distributed memory implemenation. Each digit is signed therefore a number can be distributed across several processors without loss of sign information.

The signed digit representation is used in the CALYPSO system [24]. CALYPSO implements three multiplication algorithms: Karatsuba [77], 3-primes FFT and floating point FFT. The selection of which algorithm to use is based based on

performance thresholds in asymptotic analysis: on smaller inputs, Karatsuba is best but for larger problems the FFT methods are better.

2.5.2 Parallel polynomial arithmetic

The operations on polynomial data influence the representation and data structures. Interesting polynomial representations have been presented for systems where 'shift' operations are most frequent [128]. The vectorized monomial representations in [6] are mainly targeted towards speeding up operations such as test for divisibility of two monomials, computation of the degree, and ordering of monomials in a polynomial. In our system polynomials are stored in an indexed hash table on each PE. The global hashing distributes each term of a polynomial to a PE bucket. A second level hashing procedure determines the position of each term in the local hash table.

The level 2 hash function places all terms with a similar exponent in one chain, therefore addition of any two polynomials is very fast; simply walk down each chain where the polynomials have terms. This 'make addition fast' is a significant feature of the hash method used, since addition is the most common operation in polynomial arithmetic.

2.6 Sparse systems of linear equations

Consider a linear system of equations

$$A\mathbf{x} = \mathbf{b} \tag{2.3}$$

where A is a square matrix of size $n \times n$, $\mathbf{x}$ and $\mathbf{b}$ are vectos of size $n \times 1$. The solution to the system is a vector $\mathbf{x}$ satisfying the equation (2.3).

The *density* function quantifies the relative occupation of the matrix:

$$density = \frac{\text{no. of non-zero elements}}{\text{total no. of elements}}$$

A matrix A is called *sparse* if a large fraction of the entries of A are zero ($density < 0.5$). It is called *dense* otherwise. The methods of solving linear systems of equations differ according to the density of the matrix and the coefficient field in which the matrix entries lie.

This work considers algebraic methods for solution of sparse systems of linear equations. Sparse systems can have a high level of parallelism making coarse grained algorithms attractive [105]. However, these sparsity also leads to irregular data access and huge memory demands, and the fine grain approach can be used. Consider two main methods for solution of linear systems:

1. Cramer's rule.

2. Gaussian elimination with back substitution.

Cramer's rule relies on computation of determinants of the matrix. Let $det(A)$ be the determinant of A and $det(A_i)$ be the determinant of the matrix formed by

replacing column i in A by the vector $\mathbf{b}$. Then Cramer's rule is given in equation (2.4).

$$x_i = \frac{det(A_i)}{det(A)} \tag{2.4}$$

Cramer's rule is impractical for small workstations because of the large determinant calculations. Consider that the determinant of a matrix $A(n \times n)$ is a polynomial with n^2 variables and $n!$ terms [77], therefore determinant calculations require a large amount of memory. Parallel systems with distributed memory can provide the memory required for large determinant calculations.

Let A be a square matrix whose entries a_{ij} are multivariate polynomials. The determinant of A is shown in equation (2.5):

$$det(A) = \begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ a_{31} & a_{32} & \dots & a_{3n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{vmatrix} \tag{2.5}$$

Efficient computation of determinants is the core step in Cramer's rule, therefore the focus is on parallelization of determinant calculations. In fact, these determinant algorithms have wider applications in computing the characteristic polynomial of a matrix, solving the eigenproblem by finding eigenvalues and eigenvectors, in inverting matrices and other linear algebra problems.

Several methods for calculation of the determinant are considered: the Gaussian elimination method, the Bareiss recurrence formula, and the classical minor expansion along a row or column.

2.6.1 Determinant by Gaussian elimination

The Gaussian elimination method is based on the property that the determinant of a triangular matrix is equal to the product of the diagonal entries. Therefore if L is a lower triangular matrix $l_{ij} = 0 \ \forall j > i$ then

$$det(L) = l_{11} l_{22} \cdots l_{nn}$$

and similarly for an upper triangular matrix W where $w_{ij} = 0 \ \forall j < i$ then

$$det(W) = w_{11} w_{22} \cdots w_{nn}$$

The Gaussian elimination algorithm proceeds by eliminating entries below (or above) the diagonal of the matrix until a triangular form is reached.

Matrix method

If matrix multiplication is efficient then elimination can be accomplished with multiplication by suitable coefficient matrices [17, 11]. Consider the matrix A. If $a_{11} \neq 0$, then there exists a matrix $U(n \times n)$ such that:

$$UA = \left(\begin{array}{c|ccc} a_{11} & a_{12} & \dots & a_{1n} \\ \hline 0 & & & \\ \vdots & & A_1 & \\ 0 & & & \end{array}\right)$$

The matrix U is a lower triangular matrix given by:

$$U = \begin{pmatrix} 1 & & & \\ u_{21} & 1 & & 0 \\ u_{31} & & \ddots & \\ \vdots & & 0 & \\ u_{n1} & & & 1 \end{pmatrix}$$

where

$$u_{j1} = \frac{-a_{j1}}{a_{11}} \; j = 2, 3, \dots, n$$

Since $a_{11} \neq 0$ then $u_2, \dots, u_n$ exist. All the entries on the main diagonal of U are 1, hence $det(U) = 1$. Therefore

$$det(UA) = det(U)det(A) = det(A)$$

Recursively, a new multiplier matrix U_1 is found such that

$$U = \begin{pmatrix} 1 & & & \\ 0 & 1 & & 0 \\ 0 & u_{32} & \ddots & \\ \vdots & u_{42} & 0 & \\ & \vdots & & \\ 0 & u_{n2} & & 1 \end{pmatrix}$$

Repeated application of the procedure will lead to an upper triangular matrix

$$X = U_{n-1}U_{n-2} \dots UA$$

whose determinant is then calculated as the product of the diagonal entries in X. Since each new application of U does not change the determinant, $det(X) = det(A)$. This algorithm may be used where there is fast matrix multiplication available [17]. A parallel algorithm that requires matrix multiplication can be found in [48].

Bareiss method

An elimination algorithm that uses field arithmetic and determinants of principal (topmost-leftmost) minor is developed by Bareiss [10]. This is analysed by Horowitz and Sahni [66] who present the algorithm in figure 2.7.

1. $a_{00}^0 = 1$
2. **for** $(k = 1;\ k \leq n-1; k++)$
   - **(a) for** $(i = k+1; i \leq n;\ i++)$
     - **i. for** $(j = k+1; j \leq n\ ;\ j++)$
     - **ii.** $a_{ij}^{k+1} = \frac{(a_{kk}^k a_{ij}^k - a_{ik}^k a_{kj}^k)}{a_{k-1,k-1}^{k-1}}$
3. **return** $det(A) = a_{nn}^n$

Figure 2.7: The Bareiss determinant algorithm

Let a_{ij}^k $(k < i, j \leq n)$ denote the entry at $[i, j]$ after k iterations of Gaussian elimination:

$$a_{ij}^k = \begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1k} & a_{1j} \\ a_{21} & a_{22} & \cdots & a_{2k} & a_{2j} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{k1} & a_{k2} & \cdots & a_{kk} & a_{kj} \\ a_{i1} & a_{i2} & \cdots & a_{ik} & a_{ij} \end{vmatrix} \tag{2.6}$$

Then the Bareiss method for the determinant of the matrix A proceeds through $n-1$ iterations and the result is that

$$det(A) = a_{nn}^n$$

.

The division in step $2a(ii)$ is proven to be exact [10], therefore exact division algorithms [68] may be used to reduce the cost of division. However if the entries are multivariate polynomials then each multiplication $a_{kk}^k a_{ij}^k$ is also quite expensive. An improvement due by Sasaki and Murao [106] reduces multiplication cost by avoiding some extraneous polynomial multiplications.

2.6.2 Determinant by minor expansion

The determinant of a matrix can be calculated through recursive minor expansion along a row or a column. The classical minor expansion [79, page 373] along a row j where $j \in \{1, 2, \ldots, n\}$ is shown in equation (2.7).

$$det(A) = \sum_{k=1}^{n} a_{jk} C_{jk} \tag{2.7}$$

C_{jk} is called the *cofactor* and is defined by equation (2.8).

$$C_{jk} = (-1)^{j+k} M_{jk} \tag{2.8}$$

where M_{jk} is called the *minor* of a_{jk}. The minor is a determinant of order $n-1$ obtained from the sub-matrix of A by deleting the j-th row and the k-th column.

With reference to Berkowitz [11], we may also define the *adjoint* of A by transposing the cofactor definition:

$$Adj_{jk}(A) = (-1)^{j+k} M_{kj} \tag{2.9}$$

The practicality of the minor expansion algorithm tends to be limited on sequential workstations as it quickly exceeds the memory available on many machines. The analysis by Horowitz and Sahni [66] shows that the Bareiss determinant algorithm is practical for large systems while the minor expansion algorithm quickly exceeds available memory. However, in asymptotic time for minor expansion is less than Gaussian elimination. In fact, minor expansion is optimal to within a constant factor under certain conditions on the sparsity matrix.

Smit [111] has developed an algorithm that augments minor expansion with a suitable factorization and 'memo' facility for identifying previously computed minors. The scheme effectively reduces the storage requirements and improves performance, since minors are not re-computed and each computed minor is assigned a key and stored in a hash table. A parallelization of this method would improve the storage requirements and exploit the asymptotically fast time for minor expansion.

As our polynomial arithmetic already maintains a local hash table, parallelization of Smit's method can essentially proceed if the factorization that eliminates common cancellations can be achieved in parallel. In chapter 5, the observation by Smit [110, 111] are incorporated in a parallelization of the recursive minor expansion algorithm for the determinant.

Related work by Sasaki and Kanada [105] parallelizes the minor expansion algorithm by computing each minor in block form by finding determinants of size $n/2$. The algorithm uses re-ordering of rows and columns in a sparse matrix to move nonzero entries close to the matrix diagonal. Recall that when two rows (or columns) of a determinant are interchanged then the determinant changes sign. Thus an even number of interchanges will not change the value of the determinant but may result in a simpler calculation for some band matrices.

2.6.3 Other sparse methods

Wiedemann's algorithm [124] is a randomized method for the solution of large sparse systems over a finite field GF(2). The block Wiedemann algorithm [31] relies on matrix-vector multiplication. An efficient parallelization of this method is achieved by Kaltofen and Pan [73]. An abstraction of matrix multiplication leads to a matrix-free algorithm where a *black box* representation of a matrix is used. A black box representation of a matrix requires a procedure for polynomial evaluation at a vector v. Distributed implementations of the algorithm with a black box representation have been developed [69, 36].

The methods for finite fields are not applied to the case of integer or polynomial coefficients. However, Kaltofen's division-free method [71] works in arbitrary commutative rings therefore widens the scope potentially to fields as well. It avoids divisions in Wiedemann's method. One may consider this as doing for Wiedemann's

algorithm what Smit's [111] 'cancellation-free' approach achieves for the minor expansion algorithm.

A modular method for solving linear systems of equations with polynomial entries has been developed by McClellan [89]. The system relies on Gaussian elimination in finite fields and matrix multiplication for evaluation of polynomials at a vector.

2.7 Gröbner bases

This section reviews the sequential Gröbner base algorithm by Buchberger which has application in solving systems of non-linear equations. Gröbner base computations are large irregular applications [25]. The size of the computation and the irregular data structures required make parallelization an attractive option for improving performance of the algorithm. However, parallelization based on static partitioning is not suited to the Gröbner base algorithm because there are no well-marked computation and communication phases that can be identified.

Let K be a field and $R = K[x_1, \ldots, x_n]$ be a ring of multivariate polynomials with coefficients in the field K. Consider a finite set of polynomials $F = \{f_1, \ldots, f_m\}$ where each $f_i \in R$ $(1 \leq i \leq m)$.

Definition 2.7.1 (Term) *For any non-zero $f_i \in F$, a term of the polynomial may be written*

$$c\mathbf{x}^{\alpha} = cx_1^{\alpha_1}x_2^{\alpha_2}\cdots x_n^{\alpha_n}$$

where c is a coefficient, $\mathbf{x}^{\alpha} \in R$ is called a monomial. The vector $\alpha = (\alpha_1, \ldots, \alpha_n) \in \mathbb{N}^n$ is called an exponent vector.

Definition 2.7.2 (Monomial order) *A monomial order in R is a relation $>$ on $\mathbb{N}^n$ such that*

1. *$\alpha > 0$, $\forall \alpha \in \mathbb{N}^n$*
2. *For all $\alpha, \beta \in \mathbb{N}^n$, $\alpha = \beta$, or $\alpha > \beta$, or $\beta > \alpha$.*
3. *$\alpha > \beta \Rightarrow \alpha + \gamma > \beta + \gamma$, for all $\alpha, \beta, \gamma \in \mathbb{N}^n$.*
4. *$>$ is a well-order.*

The standard monomial orderings are lexicographical order, graded lexicographical order and graded reverse lexicographical order. Given a monomial order $>$ then any $f \in R$ with its terms ordered, will have a leading term $c_1\mathbf{x}^{\alpha_1}$ so that it can be written

$$f = c_1\mathbf{x}^{\alpha_1} + f'$$

where

- the leading term of f under the monomial order $>$ is denoted $lt(f) = c_1\mathbf{x}^{\alpha_1}$
- the leading coefficient of f, $lc(f) = c_1$
- the degree of f, $deg(f) = \alpha_1$

Definition 2.7.3 (Gröbner basis) *A Gröbner basis for the ideal I generated by the set F, with respect to an admissible order $>$ is a set of polynomials $G = \{g_1, \ldots, g_r\}$ such that $lt(I)$ is the ideal generated by the set $\{lt(g_i) : g_i \in G\}$.*

From the definition, the following hold:

1. $g_i \in I$, for $i = 1, \ldots, r$.
2. $\forall h \in I, \exists g \in G$ such that $lt(g)$ divides $lt(h)$.

Buchberger's algorithm

Buchberger's algorithm for computing the Gröbner base of a polynomial ideal is shown in figure 2.8.

Input $f_1, \ldots, f_s \in F[x_1, \ldots, x_n]$, and order $>$.

1. Initialize $G = \{f_1, \ldots, f_s\}$
2. Form s-pairs $P = \{(g_i, g_j)\}$ $\forall i, j \in G$
3. While $P \neq \emptyset$
   (a) Select next s-pair (g_i, g_j) and form s-polynomial
   $$sp = \text{spoly}(g_i, g_j)$$
   (b) Reduce sp with respect to the current basis G,
   $$r = sp \text{ rem}(g_1, \ldots, g_r)$$
   (c) If $r \neq 0$ then add r to the basis, $G = G \cup \{r\}$.
   Form new s-pairs induced by the new basis element r, $P = P \cup (g_k, r)$.

Figure 2.8: Buchberger's algorithm for Gröbner basis

Several parallel implementations of the Gröbner base algorithm have been developed. The parallelization by Amrhein *et al.* [4] consists of parameterized work distribution on a shared memory architecture. Related work [25] uses application level threads on a distributed memory system. Reeves [101] implements a homogeneous Gröbner base algorithm on a distributed memory system.

The main challenge for our parallelization within our distributed representation is the question of maintaining order where the polynomial terms are stored on different PEs. As we will see, the data structures and operations in chapter 3 maintain a robust memory model and efficient memory balance. However, these representations are inherently unordered. Therefore strategies for provision of order information in Gröbner base computations based on this representation is discussed.

S-polynomials

The first step in a Gröbner base computation is the creation of s-polynomials from s-pairs $(f_1, f_2) \in G$. The s-polynomial h is defined by

$$h = uf_1 + vf_2 \tag{2.10}$$

where u and v are monomials. Let $\mathbf{x}^\alpha = x_1^{\alpha_1} x_2^{\alpha_2} \cdots x_n^{\alpha_n}$ be the leading monomial of f_1 and $\mathbf{x}^\beta = x_1^{\beta_1} x_2^{\beta_2} \cdots x_n^{\beta_n}$ be the leading monomial of f_2. Also, let $m = lcm(x^\alpha, x^\beta)$, then

$$\begin{aligned} u &= \frac{m}{x^\alpha} \\ v &= \frac{m}{x^\beta} \end{aligned}$$

and the leading monomials of uf_1 and vf_2 are both equal to m.

The s-polynomials are generated for each pair. To gain an estimate of the number of s-polynomials and pairs generated, we follow the analysis by Chakrabarti and Yelick [25]. Suppose the input basis consists of s polynomials $G = \{f_1, \ldots, f_s\}$ and the result basis after applying the algorithm has $n \geq s$ polynomials. Then the initial number of pairs is given by:

$$\|P\| = \left(\begin{array}{c} s \\ 2 \end{array} \right).$$

Consider each new addition to the basis polynomial. The i-th addition creates $i-1$ new pairs. The range of i is $s \leq i \leq n-1$ therefore the number of pairs created is given by equation (2.11).

$$\left(\begin{array}{c} s \\ 2 \end{array} \right) + \sum_{i=s}^{n-1} i = \left(\begin{array}{c} s \\ 2 \end{array} \right) + (n-s)\frac{n+s-1}{2} \tag{2.11}$$

For many large problems the result basis is much larger than the input basis ($n >> s$). Hence the right hand side of equation (2.11) gives $O(n^2 + ns)$ pairs created. Now the basis only contains $O(n)$ therefore there are about $O(n^2)$ number pairs that reduce to zero.

Pair selection criteria

There is a large number of possible s-pairs that reduce to zero $O(n^2)$. This has lead to several optimizations that use heuristic methods to identify the 'critical pairs' that do not reduce to zero. We briefly discuss the approaches taken in some parallel Gröbner base implementations:

1. The parallelization due to Ponder [97] distributes the pairs f_1, f_2 across available processors and computes $h_1, \ldots, h_p$ in parallel.

   The main shortfall of such a procedure is that it is overly dependent on initial pair distribution, while the basis set is a dynamic set that is expanding during execution. Therefore the initial distribution is incomplete and additional calls to distribute pairs are needed whenever a new polynomial is added to the basis.

2. Reeves [101] works with homogeneous polynomials whose terms all have the same degree. A scheme where all the s-polynomials for all pairs are generated and then ordered by the degree is implemented. Since all terms have the same degree, each processor can keep track of the correct order.

   The main drawback of this approach from the point of view of fine grained distribution is that s-polynomials are computed first and then ordered. Recomputing s-polynomials that are not selected would tend to be expensive. In addition, inhomogeneous polynomials have to be homogenized and this can become a bottleneck. The cost of homogenization may be reduced by a 'sugar' strategy introduced by Giovini *et al.* [54] which keeps a close approximation (sugar) of the degree of a polynomial without full homogenization.

3. The system by Amrhein *et al.* [4] does not particularly order the pairs, but creates *blocks* of pairs to fit a window of concurrent s-polynomials. The s-polynomials in a block are distributed and reduced concurrently with the expectation that in each window at least one s-polynomial will reduce to a non-zero additional basis polynomial.

   This concurrent block reduction is attractive in that it delays the introduction of new pairs until a few of the old pairs have been considered. However, this scheme is still not fully dynamic and it operates at more coarse granularity by distributing full polynomials.

Our parallel implementation in section 5.1.1 simply reduces all s-pairs. This is based entirely on the s-pairs therefore does generate the s-polynomials first. We rely on an enumeration along the rows leading to a deterministic algorithm. An alternative column major argument leads to a more heuristic algorithm which gives the correct basis up to some probability.

2.8 Summary

The target parallel environment for this research is a distributed memory architecture with a high speed interconnection network. A conceptual view of our parallel environment is developed in section 2.1 and comprises of of a message passing model with MIMD semantics. At the language level the C programming language and the MPI communication library are used. The metrics used to determine the relative merits of parallel systems were discussed in section 2.2. The main interest is in how parameters such as speedup, communication, locality are affected by fine grained randomized term distribution in the system.

Many of the parallel computer algebra systems that have been described extend existing general purpose systems with new parallel interfaces resulting in improved parallel functionality. The advantage of reusing the intellectual investment in the current systems can be weighed against the flexibility of a new implementation. We argue that a robust memory model is best incorporated in a suitable design and new implementation rather than attempting to fit this to an existing system. Therefore a small parallel computer algebra system with the memory-centric features will be developed.

The sequential algorithms for solution of sparse systems of linear equations and for Gröbner base computations require large memory and long computation time. Parallelization of these algorithms has mainly focused on speeding up the computation rather than balanced memory load. The challenges in developing parallel implementation of determinant algorithms and Buchberger's algorithm were discussed.

Chapter 3

Data structures and memory allocation

In a distributed memory system, data has to be partitioned among the available processors. The data structures supporting parallel computation therefore become more complex. The issues that present a challenge for data structure design are the following:

- Locality of data. Ideally data would be available on the processor that is most likely to require it, and communication between PEs avoided. Such ideal cases are rare, and for most computer algebra problems locality is difficult to achieve.

- Granularity of distribution. The granularity of data determines how much computation each PE can perform before essential communication. The decision to implement fine granularity has an impact on data structures at the level of integer arithmetic or polynomial operations.

- Memory balance. The system requires data placement that balances the memory load per PE, thus avoiding node memory overflow. Data structures supporting block distribution, randomized hashing distribution or other methods are possible. Our selection is based on the operations that will be performed on the data.

- CPU balance. Efficient use of CPU time requires that idle time be minimal. The CPU is idle mainly during communication or when data is not available therefore the size of data allocated to each processor is a major consideration.

In this chapter techniques for global storage management with *dynamic partitioning* are described. The techniques introduced by Norman and Fitch [93] based on random generators for selecting the storage location of data at relatively fine granularity (polynomial terms) are extended. The main supporting data structure is a *distributed hash table.* Here hashing is considered as a dynamic mapping of data to PEs and has wide implications for locality, granularity and load balancing.

3.1 Data locality

Distributed memory parallel systems are built on the principle of spatial locality: data that is likely to be needed soon should be kept local to the processor which requires it. To achieve data locality, it is necessary to develop sophisticated partitioning of data, often at the start of any computation, that can be shown to minimize communication between processors. Data partitioning at the start of a computation is referred to as *static partitioning*. The static partitioning techniques aim to keep as much data as possible local to each PE, and therefore allocate data in large blocks resulting in *coarse granularity*.

In the case of the polynomial data, the algorithms for polynomial multiplication, division, or other operations do not have uniform access patterns that permit a clean partition for locality. The polynomial matrices that considered are sparse and irregular therefore they do not display any of the properties for smooth locality constraints. The sparse systems of linear equations and the Gröbner base computations that are considered in this work have the interesting properties:

- The memory and CPU demands cannot be determined statically. In the case of sparse matrices, we encounter *fill in* at execution time, which undermines any initial static memory allocation. Gröbner base computations also generate new sets of tasks at execution time which are induced by new basis polynomials.

- There are numerous possible execution paths making it difficult to select one that minimizes execution time. In the case of finding the determinant of a sparse matrix, the choice of row or column along which expansion proceeds has an impact on the size of the computation. For Gröbner base computations, the selection criteria for next spair that is to be reduced is crucial in reducing execution time.

For these reasons, implementations based on the usual static data partitioning algorithms (block, cyclic, block-cyclic) cannot achieve suitable efficiency. For the problems under consideration, communication is an inherent necessity. In this work, dynamic rather than static data partitioning is used. The dynamic data partitioning means that as new polynomials 'fill-in' a matrix, or as new basis elements are generated, their storage allocation is determined at the same time.

The locality can be measured by the inverse waiting time: a process that spends a lot of time waiting for data has poor data locality. Figure 3.1 shows that the total waiting time is dominated by the bottleneck in communicating with the master. The randomized storage is fast in terms of the waiting time at the sending PE and the waiting time at the receiving PE, which contribute less than 10% of the waiting time.

The communication with the master to get new data reveals inefficiencies in the model. This problem is addressed in two ways: a scheduler to overlap some of the wait time with local computation, and block communication to pre-fetch some data. The effects of block communication are discussed in section 3.2.1 while the scheduler is described in section 3.9.

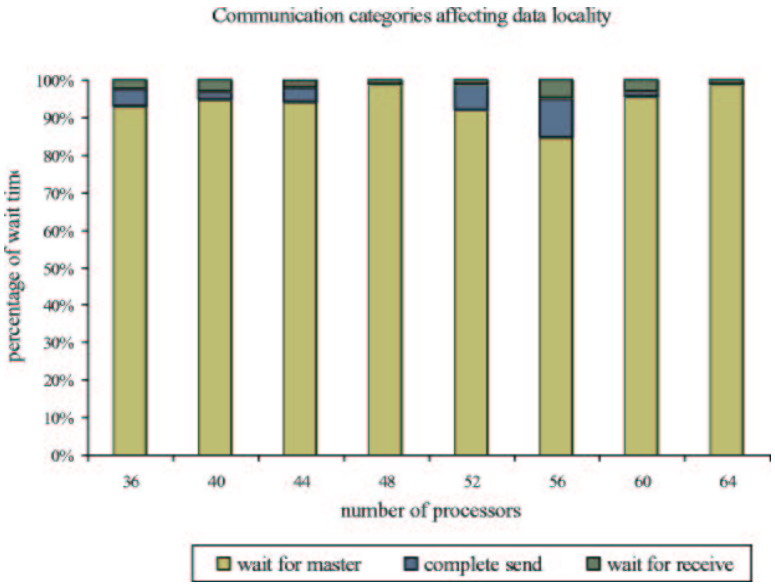


Figure 3.1: Communication categories affecting data locality

3.2 Granularity

Granularity may be addressed at levels of large integer arithmetic, representation of polynomials or basis reduction operations. At each of these levels, fine granularity is supported by suitable data representation and algorithms.

3.2.1 Block integer representation

Multiprecision integer arithmetic represents m-digit integers in a radix β. In many systems β is chosen to fit in one computer word as this is also convenient for computations with small integers. Integers in this representation have variable length of n words.

Communication with variable length messages is complex, therefore it is preferable to transfer a fixed size integer. Clearly communicating a single word digit per message leads to expensive n communications per integer.

To balance these requirements, a 'block' data structure is used. First select the block size t where $1 < t < \sqrt{n}$. Then each integer can be represented with n/t words, and the radix $\beta < 2^{32t}$. The choice of block size affects the use of memory: if t is large then there is inefficiency in representing small integers, but if t is small then a large number will induce several communications to transfer to another PE. The main advantage of a block strategy is that the message size can be kept uniform.

3.2.2 Representation of polynomials

Polynomials are mainly represented in distributed representation or recursive representation. The recursive representation is used systems have many small polynomials, and leads to coarse granularity at the polynomial level.

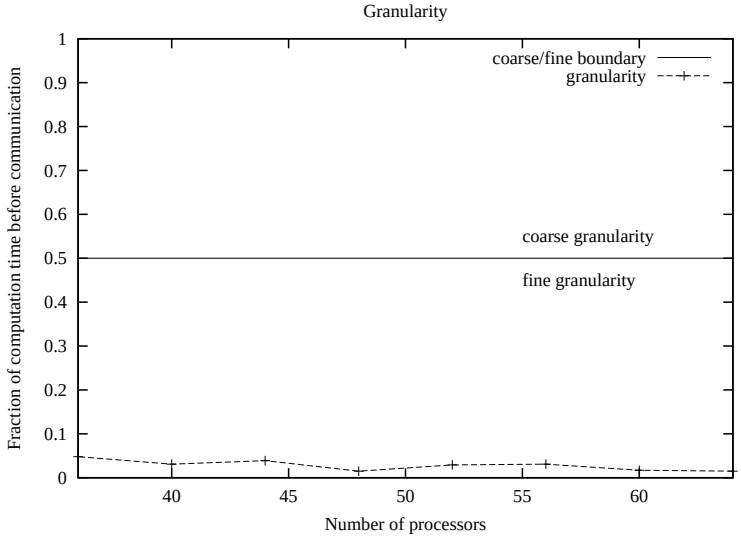


Figure 3.2: Fine granularity during multiplication

In our study, not only can the system have many hundreds of polynomials but each of the polynomials is very big: a multivariate polynomial in s variables with degree t in each variable may have up to $(t+1)^s$ terms. When computing with such large polynomials, if each polynomial is stored on one processor, then any binary operation on two polynomial operands on different processors would incur the significant communication cost of transferring a very big polynomial to another processor.

A distributed representation of polynomials provides the fine grain solution of distributing the *terms* of each polynomial across different processors. In this way, the common case which is polynomial addition can be made very fast. The fine granularity during polynomial multiplication is shown in figure 3.2.

3.3 Randomized memory balancing

It is suggested in [127] that a distributed memory parallel system requires two data structures:

1. a *solution* structure to hold partial solution of the final result, and
2. a *scheduling* structure to hold information about the tasks that have to be executed.

This section describes the main solution data structure while the scheduling structures are discussed in section 3.8. The main data structure for balancing memory use per PE is a global hash table as shown in figure 3.3. This global structure supports distributed representation of polynomials. A hash function determines the target location for storage of each *term* of the polynomial. The data partitioning is *dynamic* since a hash function is computed for each newly created term at execution

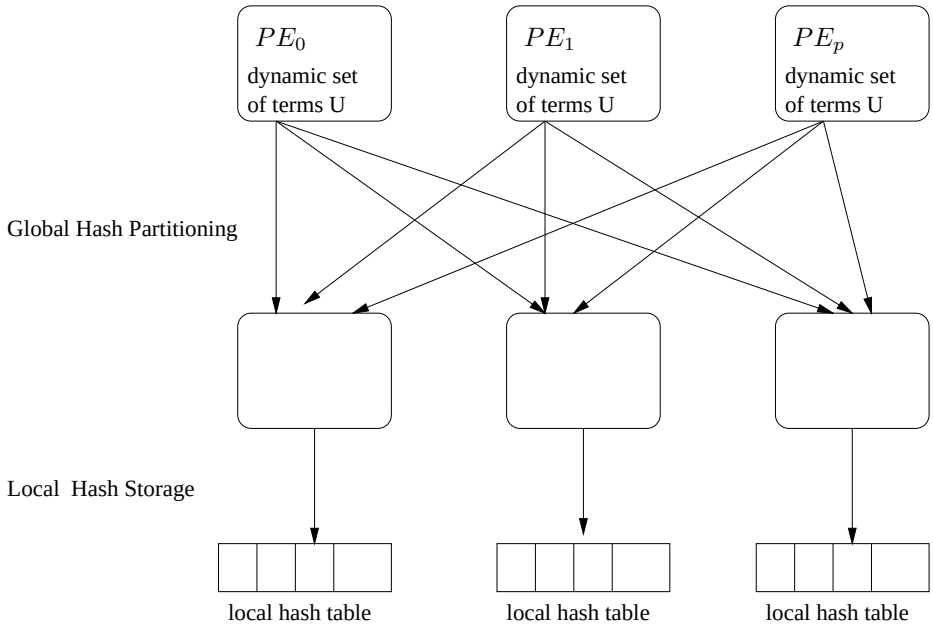


Figure 3.3: Randomized memory allocation

time. The data structure is *global* since the hash function is distributed to all participating processing elements, and the storage allocation of each term is determined separately and in parallel.

Consider each processing element as a storage *bucket* in a global *hash table*. Hashing allocates objects from a dynamic set of multivariate polynomial terms U to each bucket. The term $k \in U$ is a *key* to the hash table. Then a *hash function* $h(k)$ maps each of the term keys to a storage location on a particular PE bucket.

The above procedure gives an effective and simple dynamic data mapping of data to processors. The data is distributed at the term level rather than as full polynomials, therefore we have

- Fine granularity and
- Increased communication.

The hash table has fixed size which is a count on the number of available processors p. Recall from section 2.1 that the our parallel model has fixed topology and also the implementation of MPI used here does not permit variable number of processors.[1] In normal hashing the size of the table is chosen to be a prime number [78], but this would be too restrictive on the limited number of processors available. Therefore the table size p can be statically chosen for any number of available processing elements.

As the number of processors is small compared to the number of terms in a system of polynomial equations, collision of terms k on the same processor results in some data locality being maintained.

[1]The new standard MPI 1.2 does have threads and new processes can be created and terminated at execution time.

In the worst case, all terms map to the same processor leading to poor load balance. However, the average case evenly distributes the data across all processors giving acceptable levels of load balancing.

3.3.1 Duplicate terms

Keys are used to uniquely identify their records. In the case where the keys are formed from the exponent vector of a polynomial created during a computation, it is quite easy to see that the same key can be created more than once by a computation such as multiplication. If a term is generated on PE_i and the same term generated at PE_j with $i \neq j$, do these hash to the same bucket? As these may be generated separately and concurrently, this places some demands on the function and execution. Two cases arise:

1. Equal exponents, single polynomial: If a term with exponent vector $(1, 4, 3, 2)$ is created then this is hashed to some value and stored on a PE. If at some later stage a new term, with possibly different coefficient, but the same exponent vector $(1, 4, 3, 2)$ is created, then we would like to combine the two terms and form one term.

2. Equal exponents, different polynomials: If a polynomial p_1 has a term $5x^1y^4z^2$ and polynomial p_2 also has a term $2x^1y^4z^2$ the two keys are different but addition would be much simpler if they were on the same processor. Therefore the hash function should place these on the same PE.

The main requirement is that duplicate keys, that may be generated on different processors, should hash to the same processor. This will enable addition and evaluation operations on each PE to be conducted concurrently. To start with, we will consider only the case where once an item has been placed in a bucket, it is not relocated. In section 3.6 the issue of relocating some data to achieve a target balance is considered.

3.4 Local memory allocation

The operation of inserting a new term into a distributed hash table is dependent on the hash function. In a parallel system where each bucket is a processing element as shown in figure 3.4, the cost of insertion includes the cost of communication:

$$T_{insert} = T_{hash} + T_{comm} + T_{update}$$

The selection of local data representation depends on the operations on the data and also on whether the algorithms required ordered data:

- Additive operations. Each PE is essentially an additive processor, and the level 1 hashing ensures that like terms hash to the same PE. Therefore a data structure that simplifies the most common operations of addition and evaluation are essential.

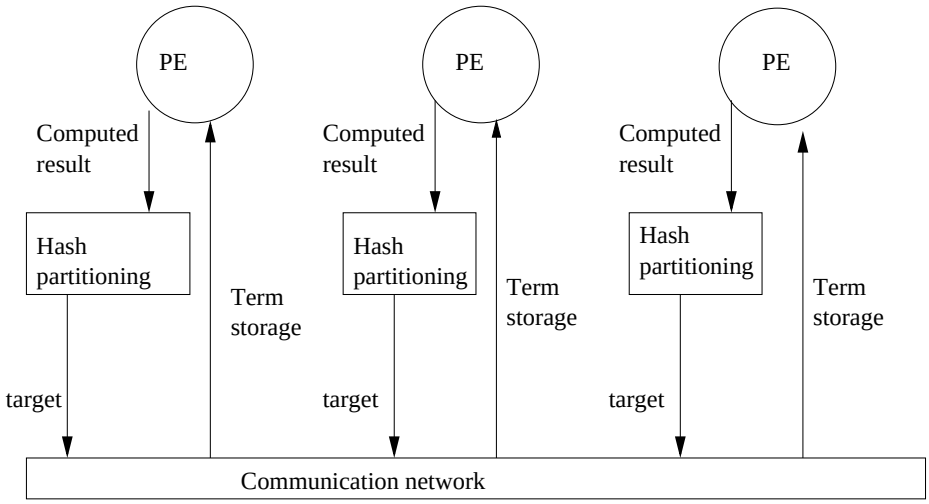


Figure 3.4: Dynamic partitioning at execution time

- Order. An unordered storage is less costly. However, if the system is to be used primarily for applications such as Gröbner base computations, then order is required, and the best way of achieving this is with a data structure that supports sorting of terms.

The main options for local memory management are to keep terms unordered in a hash table (a local one), or ordered in a list or heap. Local representations that store data in sorted order are useful for the Gröbner base computation where finding the 'leading term' of a polynomial is a key operation. Local storage in a heap data structure results in a partial ordering of the terms of a polynomial where locating the leading term is an $O(1)$ operation. Maintaining a heap costs $O(n \log n)$ which has to be offset against the other operations in the basis reduction algorithm.

Another sorted order approach maintains local term storage in a binary search tree [76]. High performance polynomial multiplication algorithms based on this representation can be performed in time $O(n \log n)$.

The implementation provides unordered storage. Polynomial terms are allocated to a processor location through a randomized hash procedure. Local terms undergo a second level of hashing to determine their placement in a local hash table H_2. The hash table storage is unordered, however the hash table has the capability of aligning like terms from different polynomials. The alignment means that addition of two or more polynomials is a fast sweep through one bucket of the hash table, which makes hashing an attractive data structure.

The local hash insertion places one exponent vector (key) per bucket. Terms from different polynomials that have the same exponent vector are chained to the same bucket. Thus collision only occurs when two different exponent vectors hash to the same bucket $h_2(k_i) = h_2(k_j)$. In this case an *open addressing* [33] scheme takes effect. A probe sequence is generated:

$$h_2^1(k), h_2^2(k), \ldots, h_2^n(k)$$

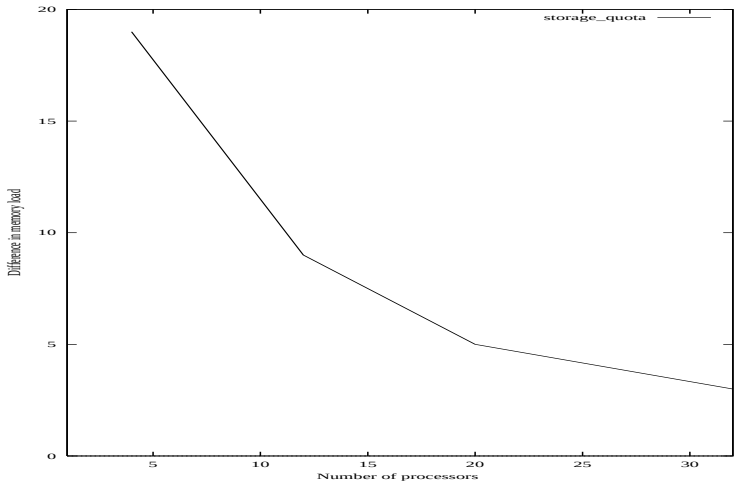


Figure 3.5: Margin of imbalance in memory load (κ)

The probe sequence is a permutation of the set of buckets and this determines the order of checking buckets for an empty space.

3.5 Load factor

The *load factor* indicates the current number of terms on each PE. The relative sizes of the load factor inform the distribution algorithm in determining the next target for a term.

Let n be the number of keys that have to be stored, and m is the number of buckets in a hash table, where each bucket can hold a maximum b keys. Then the *load factor* ζ is defined by

$$\zeta = \frac{n}{mb}$$

The load factor gives the average number of key data in a particular bucket. Now consider the case where each bucket is a processing element. Then m is the total number of processors, and b is the size of available memory for storing polynomial data on that processor.

Let p be the number of processors and ζ_i $(0 \leq i \leq p)$ be the load factor for each PE in the system. Then one can determine the imbalance in the memory load by determining the variance

$$\kappa = \max(\|\zeta_i - \zeta_j\|) \text{ where } 0 \leq i, j \leq p$$

Figure 3.5 shows that as the number of PEs grows the distribution of data becomes more even as κ decreases. Thus the system achieves a good memory distribution.

3.6 Relocation

The two levels of hashing described are an effective memory balancing system, which lead to uniform distribution of the data. However, if the load factor is high, then

this will lead to cases where the entire probe sequence is checked without a free slot, and the PE encounters *local memory overflow*. This triggers a trap to global *relocation* procedure. The following methods of dealing with such global relocation are available:

1. Modified linear probing.
2. Cascading probe.
3. Larsen and Kajla method.

Modified linear probing

The sequential linear probing method searches through adjacent buckets until an empty space is located. A simple parallel scheme will search the PEs in a sequence

$$h(k) - 1, h(k) - 2, \ldots, 0, p - 1, p - 2, \ldots, h(k) + 1 \tag{3.1}$$

At each probe, the following protocol is engaged:

1. First, send a *probe* message requesting bucket status from the processor indicated by i in the probe sequence.
2. The target replies with a *yes* or *no* depending on whether it can accept the data.
3. If the target response is *yes*, then send data. Otherwise move to the next probe in the sequence and begin again at step 1.

Each probe will require at least three messages making this protocol expensive if the search is short (for example if space is available immediately on the first PE $h(k) - 1$). However, the advantage of this method is that the data is sent only once from the current processor to the target processor that has space available.

Cascading probe

To limit the number of point-to-point messages between a pair of PEs in a probe, a cascading probe is useful. Cascading probe assumes it is cheaper to send one data message than the three protocol messages. With fixed message size, a proper bound on the size of data is available and therefore this assumption can hold. The simple idea for the cascading probe is to generate the probe sequence 'on the fly' at each PE. This exploits the fact that each of our buckets have processing power, which is not the case when probing on a single processor.

In a cascading probe, only one message containing the full data is transmitted per probe. The main requirement is that each processor generates the correct next probe as if the probe were generated by a single processor. In this case the following protocol is in effect:

1. Receive data.

2. If there is no space on current PE whose identifier is i, then compute the next PE in the sequence.

3. Let the target PE be k, then send data to PE_k.

The method of Larson and Kajla

The cascading probe does not give a bound on the number of hops and may easily send $O(p)$ messages where p is the number of processors. Is it possible to determine at the origin processor, where the correct empty bucket is without the communication overhead of the global linear addressing?

Larson and Kajla [85] describe a modification to open addressing adapted for file storage. The use of objects called *separators* to identify the boundaries of storage buckets in a probe sequence leads to highly efficient method of identifying the location of a key.

The method of Larson and Kajla makes use of a small table to aid searching through the a probe sequence, and can uniquely determine the correct bucket. Therefore only one message is sent for each term storage.

The method selects the keys to be relocated from an overflowing bucket, based on special identifiers called *separators*. The separator shows the maximum key that can be held in a bucket and in the case of overflow, the separator is lowered and any keys above the limit have to be hashed to a different location.

This method relies on dynamic relocation for data that is already inserted into the storage. This presents a problem for our parallel system since the relocation would need additional setup time and new buffers. For large number of processors (> 512), this method can be used in conjunction with cascading where a suitable cross over point is available.

3.7 Analysis of the randomized global hashing

How does the randomized memory distribution model balance memory requirements and satisfy *peak memory demand* of algebraic algorithms? The algorithm dynamically distributes additive terms from computations. Every PE within the system uses the same algorithm, therefore the protocol is *uniform* [84] across all participating processors. The main item being distributed is a *term*, therefore the algorithm may be classified as fine grain.

Consider the parallel system as a graph $G = (V, E)$, where the vertices V are the processors and the edges E are the connections based on the system topology. If there are p processors then $V = (\mathrm{PE}_1, \ldots, \mathrm{PE}_p)$. The memory balance algorithm is essentially a mapping

$$h_1^{ij}(k) : \mathrm{PE}_i \rightarrow \mathrm{PE}_j (1 \leq i, j \leq p)$$

where PE_i is the *source* processor where the term k is generated, and PE_j is the destination processor where storage for the term is allocated. This definition clearly permits $i = j$ where storage is allocated on the same processor.

Two random seeds s_1, s_2 are generated at the source processor PE_i. Let v be the length of the exponent vector. Then v keys are generated for by a random key generator using the seeds:

$$key_d = rand(s_1, s_2)\ 0 \leq d \leq v.$$

The hash function makes use of the randomized keys to determine the target location for each term:

$$h_1^{ij}(k) = \left(\sum_{d=0}^{v} key_d k_d \right) \mod p \tag{3.2}$$

.

For each new term computed during an arithmetic operation, storage space is allocated on a *target* PE. The number of PEs gives a count on the number of buckets into which the term can be placed. This places a bound on cycle length for a random placement algorithm.

The randomised algorithm is expected to select a destination processor PE_j with equal probability. Thus at each iteration a processor has expectation of $O(p)$ incoming communications. This means that some $p-1$ buffers have to be allocated for all possible messages. In a system that is optimising memory, this is too much space. The next section discusses introduction of a probabilistic weight that improves the buffer allocation.

3.8 Weighted memory balancing

The global hashing gives a uniform probability distribution and the destination processor PE_j is selected with equal probability. The function in equation (3.2) is independent of i, j. However, the communication channel is vulnerable to multiple messages arising from concurrent execution of the function. At each iteration a processor has anticipates $p-1$ incoming communications, therefore some $p-1$ buffers are allocated for receiving messages from any other processor. A traffic analysis can determine actual number of communications in a time slice t. Therefore a general weight can be assigned to each new communication to improve the selection of the least recently used channel for the PE to send data to and prediction of where to receive data from. Two additional data structures in the form of matrices are used by the scheduler to support the weight calculations.

The main scheduling data structures are a *history matrix* and a *traffic matrix* on each PE. The history matrix allocates *temporal locality* to each PE, showing the communication patterns in recent iterations. The traffic matrix shows total communication traffic in order to identify potential 'hot spots' in communication and avoids communication in that path even when the hash function gives that destination.

Traffic model

To assess the message traffic within the system, a communication log is kept in two matrices: A receive matrix $R(l \times p)$ and a send matrix $S(l \times 3)$, where l is a variable

size of the history to be saved. The receive matrix is a binary matrix where the cell (i, j) is set if during the $i - th$ iteration data was received from PE_j. The send matrix can be contracted in size by enumerating the *message tags* that indicate the type of data. The matrices give temporal locality information so that the following questions may be answered:

- Which is the least recently used receive/send port?
- What is the weight of communication on a link c?

Consider the receive matrix R. Let the first column indicate the previous send. The rest of the $p - 1$ columns are for the previous receive. A schedule based on the communication matrices introduces a probabilistic element to the computation itself, as well as the selection of storage location. The weight can be based on three options [98]: pairwise scheduling, balanced scheduling and greedy scheduling.

Traffic generated by the system can be analysed to evaluate the probability of communication clash. These clashes will not usually generate deadlock, since alternative routing through the network is possible, but such alternative routing results in delays in transmission of messages. There are two potential hazards in the communication generated:

1. The funnel hazard: Several processors sending to one processor at the same time. This has the potential to delay all the messages in a queue which the receiving processor handles each in turn.
2. The succession hazard: Successive send to the same processor. In this case the second message may be delayed while the previous message is sent.

The randomized dynamic distribution algorithm is quite efficient in maintaining data locality and local storage is preferred as shown in figure 3.6. The algorithm has to ensure that when PEs begin to send messages to each other in quick succession, the channel delay is not increased drastically. In figure 3.6, consider the time slice 30 where PE_0 sends a message to PE_1, and PE_1 itself sends a message to PE_0. This concurrent send is likely to cause a delay to the next send in the next stage (time 35). This 'succession hazard' is flagged by weights w_{ij} at each PE, which are used in the next time slice 35.

time	5	10	15	20	25	30	35	40	45
PE_0	0	0	0	0	0	1	1	1	1
PE_1	1	1	0	1		0		0	

Figure 3.6: Traffic during multiplication with 2 PEs

The communication 'hot spots' resulting in succession hazards may be remedied with *block communication*. Given a non-zero weight w_{ij} on a channel from source i to destination j , a block size $b = f(w_{ij})$ $b > 1$ is calculated. Messages are then sent when the block is full. In this way any successive sends to the same processor are caught locally. However, any communication which involves just one message will

be delayed until the final sweep of all buffers. If all messages are storage messages with the data not being reused, then such delays are acceptable.

For a large number of PEs there is a second potential hazard for the communication channel: a 'funnel hazard' where tens of PEs are sending to the same PE in one time slice. Note that this has no effect on the uniformity of the memory allocation. However it is a hazard because it requires that $p-1$ buffers be allocated memory, and also potential delay as each message is received one at at time. The example in figure 3.7 shows the spread of communication in terms of pairs involved. Note also that in each of the columns, there is no occurrence of the same destination number indicating that all processors are sending to the same PE. Thus the system has efficient handling of the funnel hazard. Indeed, for this example, a blocking factor of $b = 2$ fully controls the succession hazard as well.

time	5	10	15	20	25	30	35	40	45	50	55	60	65
PE_0	2	2			0		0			0	0	1	1
PE_1	0		0	2	2			2	2	1	1	2	2
PE_2	1			0			0			0			1

Figure 3.7: Traffic during multiplication with 3 PEs

Channel selection

The traffic matrices can be used in a 'least recently used' protocol on the communication channels to say that if PE_j has recently sent a term to PE_k during iteration i then it should not send another data item to the same processor at the next iteration $i+1$.

The probabilistic information has two important consequences:

1. The number of input buffers can be reduced by placing them all in one pool. Incoming message are allocated the first buffer in the cyclic pool. This is because the expectation of incoming messages equal to the number of processors is reduced, and the number of incoming messages far less than the normal $p-1$.

2. A cyclic pool is efficient in this case rather than a full array of $p-1$ buffers. This means that the size can be adjusted depending on the application, with better tuning to the requirements of the system.

The probabilistic communication function to determine whether to send a message μ_j should be sent to a destination PE or whether to perform a silent local summation where the destination is the current PE. The processor therefore moves into a new state depending on whether a non-local communication is being initiated. At the end of these steps, each processor sends and receives all its messages.

Two important questions arise:

1. What is the probability that at time j processor PE_k will receive $p-1$ messages?

2. What is the weight of a previous communication μ_{j-1} in determining the next communication?

The first question impacts on the number of receive buffers that each processor needs to allocate for incoming communications. Memory space is expensive therefore we wish to limit the number of buffers without incurring the possibility of losing messages. The second consideration relates to the communication patterns of the algorithm. In a history based algorithm, the selection of appropriate target communication so that each PE does not communicate heavily with another will have an impact on the communication.

From these two considerations, a successful time unit j is one where:

- number of messages received at v_k is less than or equal to the number of buffers, and
- all messages are guaranteed to be received.

3.9 The scheduler

A master/slave assignment of processes will often speed up distribution of data and is useful as a funnel for operations such as Input/Output where concurrent access to a shared stream is not provided.

The master/slave assignment can be relaxed to provide *equivalence*; each PE may become a master process based on a floating token assignment. This generates a more flexible cyclic processor alignment, where the processor that holds the 'master' token can broadcast data to all other processors.

The overhead of the dynamic scheme occurs when a change of status occurs. The challenge is that the processor that is currently master must willingly relinquish its master rights, and a new master selected from the slave processes. This process occurs in several steps:

1. The master process informs all other processes that it is relinquishing the master token. This can be achieved as the master process will have broadcast rights. Therefore given a unique code, a message indicating the end of current master rights is transmitted.
2. Each of the slave processes computes its elevation level depending on the identity of the exiting master. One and only one processor should hold the master token at a time.
3. All processors re-align their receive buffers to receive from the new master process.

The simplest cyclic hand over of the master status between PEs is a round-robin protocol. A fully distributed and orthogonal implementation of round-robin relieves much of the inefficiency of a statically assigned master process which idles throughout the computation after the system initialization.

Secondly, this is an important part of delivering fine grain distribution of data. Since data is distributed at term level a 'just-in-time' transport of terms from the master PE to all slave PEs avoids over crowding within the network.

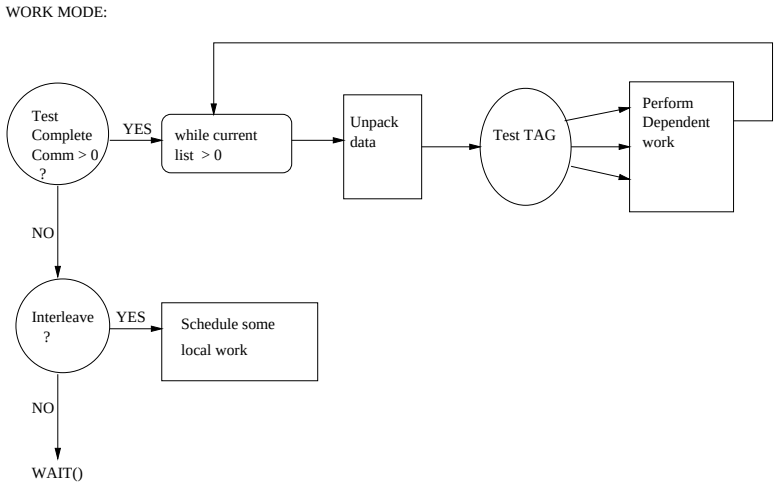


Figure 3.8: Local process scheduling

3.9.1 Process mode

In addition to a dynamic status, a process attains a *mode*. The classification of process modes provided by the lowest scheduling level and essentially divides a process time into *work mode*, *communication mode*, and *resource management*.

A process in work mode could perform *local work* where all the required data is available locally. *Dependent work* relies on remote data on other processors and may incur some wait time before the data is available locally. This allows us to perform second level scheduling where dependent work has higher priority in order to clear communication buffers.

The algorithm shown in figure 3.8 tests for incoming data in buffers and proceeds to computations that depend on the data. If there is no incoming data, some local work is performed until test window has expired.

The interleave function makes use of the resource manager to determine a heuristic measure for the amount of local work that can be performed before a new test. It is found to lead to acceptable waiting times.

An alternative strategy of performing local work until dependent data becomes available requires a more robust interrupt service routine; it should be capable of handling the worst case possibility of $p-1$ processors sent to the same processor in one cycle (even though there is low probability of this happening) .

3.9.2 System organization

The architecture of our parallel computer algebra system is shown in figure 3.9. The kernel of a parallel computer algebra system determines the services offered to application level programs and the functionality of the system. We have focused our investigation on the representation and data structures on a MPP for Multiprecision Integers and Multidimensional Polynomials. The implementation and support of basic arithmetic operations based on the data structures are then clearly different.

We are in fact creating, bottom-up, a language and programming system capa-

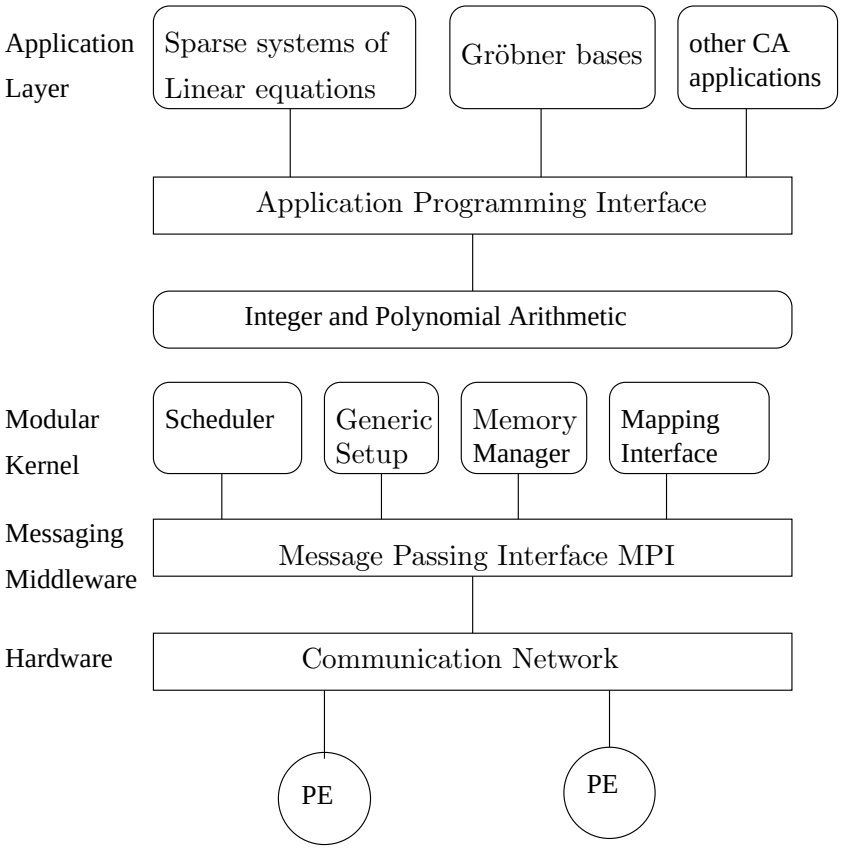


Figure 3.9: Architecture of a parallel computer algebra system

ble of supporting very large symbolic and algebraic computations. To achieve this requires a careful integration of the parallel paradigm within the known structures for computing environments.

3.10 Summary

In this chapter a randomized memory distribution model to balance memory requirements and satisfy *peak memory demand* of algebraic algorithms was described. The algorithm uses a hash function to dynamically distribute additive terms from computations. Every PE within the system uses the same algorithm therefore the protocol is *uniform* [84] across all participating processors. The main item being distributed is a *term*, therefore the algorithm may be classified as fine grain.

An integrated approach to the partitioning problem combines an initial static partitioning with dynamic storage allocation at run time. Two methods of dynamic storage allocation have been described: a random allocation method and a probabilistic allocation method. Both these methods provide system-wide fine grain control over memory allocation, and permit flexible load balancing at the cost of increased communication. The dynamic distribution is supported by a distributed hash table data structure which forms an important part of the system allowing fast insert and internal conflict resolution. Relocation of data is not implemented, although the method of Larson and Kajla shows a way of introducing relocation while avoiding deadlock.

Chapter 4

Parallel arithmetic

This chapter describes the parallelization of basic arithmetic on very large multiprecision integers and on multivariate polynomials. The classical algorithms are found to be flexible in terms of exploiting the fine grain data distribution from the previous chapter. Faster algorithms for both integer and polynomial arithmetic have been developed [70, 40, 13]. However the parallelization of classical algorithms enables us to show that a robust memory management strategy leads to significant performance improvement even with the very simplest algorithms.

4.1 Multiprecision integer arithmetic

The arithmetic kernel is comprised of suitable data representations and the main operations of addition, multiplication and division as shown in figure 4.1.

The multiprecision integers are used as coefficients of polynomials therefore integer and polynomial operations are discussed as related entities and not separate packages, which has the following effects:

- Polynomial representation. Integer coefficients have to be included in the representation of a polynomial.
- Term distribution. The distribution of integers is placed at the same level as the distribution of terms of a polynomial.
- Message length. To keep message sizes uniform, large integers of variable size will have to be represented in suitable sized blocks.

	Multiprecision Integers	Multidimensional Polynomials
Representation and Data Structures	- Redundant Block Representation - Classical methods	- Distributed Hashing - Randomized Placement - Non-replication
Arithmetic Operations	- Addition and Carry propagation - Multiplication - Coefficient within polynomial	- Multiplication - Division

Figure 4.1: Kernel components

We first discuss the representation of multiprecision integers, and then in later sections examine the consequences for polynomial representation, term distribution and message length.

4.1.1 Redundant signed digit representation

Any representation balances two considerations: the sign data for negative integers and carry propagation.

Consider an integer D written as a polynomial in a radix β as follows:

$$D = s \sum_{k=0}^{n} d_k \beta^k$$

The sign s presents a difficulty for distributed representation of integers; if the straight forward sign-magnitude representation were to be used, then the sign information would only be available on one PE making it necessary to search for the sign in every arithmetic operation.

Since each digit d_k is to be distributed, an appropriate representation is the signed digit representation [5], which has the following benefits:

- Each digit encapsulates sign information, therefore arithmetic can be performed on the digits concurrently.
- Signed digit representation limits carry propagation to adjacent digits. This simplifies communication and leads to almost fully parallel arithmetic.

For other parallel environments such as systolic processors, redundant signed digit representation may not be very attractive [67]. However, in a distributed memory system where data is spread across several processors, there is a real saving from avoiding communication for carry propagation or sign transmission.

The properties of a redundant signed digit representation with a radix $\beta > 2$ are described below.

1. Each digit d_i is allowed to assume one of q values in the range

   $$\beta + 2 \leq q \leq 2\beta - 1$$

   This extends the range of allowed values of each digit beyond the β values of the standard representation. The minimum redundancy is $\beta + 2$ which means that there are at least 2 extra digits within the representation of each digit.

2. Subtraction is performed by adding the minuend to the additive inverse of the subtrahend. Thus placing a requirement for the additive inverse of every representable digit. To satisfy this condition, the absolute value of each digit is bounded:

   $$\|d_i\| \leq \beta - 1$$

3. The intermediate sum w_i is bounded so that if w_{max} is the maximum representable integer per word, and w_{min} is the minimum. Then we have:

$$w_{max} - w_{min} \geq \beta - 1$$

Therefore if the radix β is *odd* we get possible values completely symmetric about zero and works with digits in the region

$$-\frac{1}{2}(\beta - 1) \leq w_i \leq \frac{1}{2}(\beta - 1)$$

For example, for a radix 10000 the representable range would be -5000 to 4999.

Choice of radix

In most implementations the radix β is often chosen as a power of 2 with minimum requirement is that $\beta - 1$ can be represented within one computer word. Therefore the widely used radices are in the range $\beta = 2^a$, $(29 \leq a \leq 31)$. Additional considerations regarding the radix include the following:

- An odd exponent a is useful in a redundant representation since it assures that we have a unique representation for zero.
- An even exponent a makes binary splitting symmetric therefore if the main operation is multiplication then an even radix may be preferred.
- Addition requires that an overflow check be performed. To check intermediate overflow only once, we must have $w_{max} > 3\beta$. This needs 2 overflow bits. On a 32 bit computer, this means $\beta = 2^{29}$. However, if we are willing to check overflow twice then we can have $w_{max} > 2\beta$ giving a minimum $\beta = 2^{30}$.

Block representation

To enforce uniform message size, variable length integers cannot be sent all in one message. Therefore a fixed block size is selected, and blocks of digits transmitted at a time. The block representation extends the radix beyond a representable integer. This is similar to the PARSAC implementation [82] which makes use of blocks of size $m \leq 8$. In the current implementation with fine granularity, a size $m \leq 4$ is used. In fact, once the length of digits is fixed at 4 we may now consider the single digit having length 4 words and adjust the radix to work in radix $\beta \leq 2^{128}$. A convenience of this adjustment is that it is quite cheap to select an integer power of 10, which simplifies decimal conversion [86]. Consider $\beta = 10^{37}$, which meets the criteria for $\beta \leq 2^{128}$ and simplifies conversion between decimal and internal representations.

Each digit d_k is held in four computer words. The exponent of the radix is needed to identify each block and in a coefficient representation, we can actually attach this radix exponent to the exponent vector for the term and simplify memory allocation.

$carry = 0$
for $(i = 0;\ i \leq l - 1;\ i++)$
    $carry = carry + a_i + b_i - n_i$
    $c_i = carry \mod \beta$
    $carry = \lfloor carry/\beta \rfloor$
end

Figure 4.2: Multiprecision addition

Message size in communication

The distribution of the representation of integers of arbitrary size is dependent on the communication. The most obvious representation creates a variable length object to hold an integer so that arithmetic could be performed on a single processor and has the advantage of clarity. However this imposes a heavy penalty on communication; the messages to send an integer coefficient of a term would also have to be of variable length. Variable sized data communication requires that the send and receive buffers be at least large enough to hold the largest possible coefficient, therefore creating space for only a few buffers and in most cases the integers will be of much smaller size than the allocated space. The irregularity in communication and the need for very large buffers with internal fragmentation suggests that the block representation is more suitable for our purposes.

4.1.2 Operations on signed digit integers

The complexity in the data representation shown above, points to several ways of implementing the basic arithmetic operations. The classical algorithms are preferred, in order to display the performance of the underlying representation.

The signed digit representation limits the range to represent positive and negative numbers. Arithmetic proceeds in two stages: first the operations per processor give a result which may be outside the representable range. The second step approximates carry propagation.

Addition

The classical addition algorithm in figure 4.2 is implemented on a MPP with signed-digit representation and a carry-lookahead. We extend the parallel prefix algorithms [42] for multiple bits per processor, to create an implementation with multiple bits per digit. This gives the multiprecision integer and modification to the *propagate* and *generate* bits in parallel.

The addition and subtraction of large integers is relatively cheap $O(1)$. Carry propagation with the signed digit representation is bounded, and addition can be carried out as a sweep through the local hash table to a depth s, where s is the number of integers being added.

Multiplication

The parallel multiplication is based on the classical multiplication algorithms. As each polynomial has a distributed representation, each processor can independently compute a term multiplication, and form part of the result.

Division

The classical long division algorithm gives good performance for inputs up to 8192 bytes [24]. The key performance measures in a distributed division procedure are *efficiency* and *load balance*.

The simplest implementation collects all the terms of the operands onto one master process which performs an optimized division. This breaks the distribution and leads to a sequential computation time. The other PEs will be idle while the master computes therefore very poor efficiency is achieved. However, for short numbers (and all those with 1 B-digit), this is a reasonably simple procedure.

For the special cases where the divisor is much shorter than the dividend, it is possible to distribute all the terms of the divisor to each PE. This enables the local computation of partial results in an efficient way.

Multiplication is a high speed operation in CABAL. This can be used in a division algorithm that approximates the reciprocal of the divisor (using Newton-Raphson iteration) before multiplying all the terms of the dividend.

In applications such as the Gröbner base computations in section 4.1.2 the fraction can be kept as a numerator and denominator before a normalization step is taken. Since a common operation in Gröbner base computations is $\frac{\mathrm{lcm}(a,b)}{a}$ the representation is efficient for the frequent case where a, b are relatively prime.

Rational arithmetic

Much of our arithmetic is with integer coefficients. However inexact division can lead to rational numbers, therefore we briefly discuss the rational arithmetic.

A simple approximation to rational arithmetic that retains the integer representation is used. A common denominator Q is kept. The representation is then

$$\frac{c_1\mathbf{x}^{\alpha_1} + c_2\mathbf{x}^{\alpha_2} + \ldots + c_n\mathbf{x}^{\alpha_n}}{Q}$$

This relies on the low expectation that the coefficients $c_i, 1 \leq i \leq n$ are coprime. If a large number of coefficients do happen to be coprime, then Q becomes a very large integer with the worst case

$$Q = \prod_{i=1}^{n} c_i.$$

Input/Output

A large fraction of the work in implementing a multiprecision arithmetic package is in the conversion between different radices.

The usual external representation of numbers is sign-magnitude form. The output (printing) subsystem for the distributed redundant representation format has to convert the internal form to the sign-magnitude format. It performs the following steps:

1. First normalize the integer in radix β with a full second stage carry propagation. This requires that each slave PE identifies its leading digit and send the $(sign, position)$ to the master processor. This information is used by the master in determining the order for terms, and the overall sign of the multi-precision integer.

   In the worst case the recursion may be as deep as the number of digits $O(n)$. However the average case propagates to only two stages therefore this becomes an $O(1)$ operation.

2. When all the digits on all PEs are normalized, the master may merge the ordered digits and send them to the output stream.

The conversions in I/O are expensive and in addition the streaming buffers for the master require significant memory allocation in the final stage. Two heuristic approaches to improving the final merge stage:

1. For integers with number of digits less than m each slave PE can send its terms to the master, and let the master perform full sorting.

2. It may be necessary to cap the size of integers that may be printed so that only digits up to some upper limit may be printed.

4.1.3 The impact of pseudo-vector processing

Few architectures provide integer arithmetic operations, therefore integer operations and especially big number arithmetic is performed in the floating point unit. Integer arithmetic is then performed in double precision and converted back to integers without loss of accuracy.

In this section, a special feature of the floating point processor on the Hitachi SR2201 is used for integer arithmetic. The Hitachi SR2201 architecture provides a pseudo-vector processor (PVP) that is highly optimized for numerical computations. The PVP operates on 128 registers with a fixed 32 register address space. This provides an opportunity for *prefetch* and *poststore* operations that hide the memory latency and speed up computations.

Each β-digit in the representation of integers is a vector of length at least 4. The vector processor reduce the cost of memory latency pre-fetching all 4 words of a digit in this representation (it can fetch up to 32 therefore 4 is a relatively small size). Full use of the PVP window of 32 words could therefore compute with any integer of up to 10^{296} decimal digits in working storage at register level. This number is large enough for the usual purposes, and any larger integers can have some of the digits stored in main memory.

This facility can be used to speed up integer arithmetic, with a few modifications. A general algorithm for full vector processors is used [92], therefore the performance

enhancement is useful beyond the SR2201 target architecture. However, full portability of the code cannot be guaranteed.

4.2 Parallel polynomial arithmetic

A multivariate polynomial with integer coefficients is represented as in distributed form as a list of terms. Each term is made up of a *coefficient* and an *exponent vector*. Let $\mathbf{x} = (x_1, x_2, \ldots, x_k)$ and $\alpha_i = a_{i1}, a_{i2}, \ldots, a_{ik}$. Then $\mathbf{x}^{\alpha_i} = x_1^{a_{i1}} x_2^{a_{i2}} \cdots x_k^{a_{ik}}$. A polynomial C is represented in distributed form:

$$C = \sum_{i=0}^{n} c_i \mathbf{x}^{\alpha_i}$$

where each coefficient c_i may be of arbitrary length:

$$c_i = \sum_{k=0}^{m} d_k \beta^k$$

Space and uniformity considerations lead to two equivalent views:

- The coefficient may be represented in the uniform polynomial representation of a list with exponent vectors having zero entries for all except one variable.
- The coefficient may be represented with an implicit univariate representation of an array of 'coefficients' in the radix B.

Recall from the previous chapter that the block representation uses up the first exponent in the vector α_i so that in an array representation $\alpha_{i0} = k$ indicating the position of the digit in the coefficient. The pairs (d_k, α_i) uniquely describe a block within a term and they are distributed among the available PEs through the randomized hashing algorithm (see chapter 3).

4.2.1 Polynomial addition

Addition of t polynomials proceeds by iteration on addition of 2 polynomials, therefore we consider addition of two multivariate polynomials $p + q = r$. Let p have n terms partitioned such that there are N terms per PE. Also, q is of size m with M terms per PE. The addition algorithm in figure 4.3 is executed by each PE separately and in parallel.

Common exponent vectors will hash to the same bucket, therefore the first step of the addition proceeds as a sweep down the linked list of terms in the bucket to find the coefficients corresponding to p and q. This linked list has depth equal to the number of polynomials in the system so the sweep is an $O(t \times \max(N, M))$ operation where t is the number of polynomials.

After identifying the coefficients, each processor calls the algorithm in figure 4.2 to add two multiprecision integers. At this stage carries are not propagated beyond the current term. Note that each of the terms holds just one positional digit in the

1. **for** $(i = 0;\ \ i < \max(N, M);\ \ i++)$
2. add $p[i]$ to r
   (a) **if** $(\text{hash}(p[i]) = \text{hash}(q[j]))$
      i. sweep down a shared bucket
      ii. add coefficients over Z
      iii. create block terms
   (b) **else** add $q[j]$ to r

Figure 4.3: Parallel polynomial addition

1. Increment the positional exponent a_{i0}
2. Find PE location: $\text{target}(a_{i0}, a_{i1}, \ldots, a_{ik})$
3. Find bucket: $\text{hash2}(a_{i0}, a_{i1}, \ldots, a_{ik})$

Figure 4.4: Identifying blocks of the same term

coefficient therefore if there is a carry it will not affect other terms but it does affect other blocks of the same coefficient.

The next step checks if there are higher positional blocks using the algorithm in figure 4.4. Recall that the first index in the exponent vector indicates the positional place for a block integer. The algorithm increments this and finds a new bucket. If the higher positional integer block exists then the carry from the lower position is added to it. The blocks are updated for each carry, and we have seen from section 4.1.1 that the signed digit representation bounds such cascade of such carries to 2 levels on average. Therefore the cost of this operation is about $2 \times C_{comm}$.

From this analysis, the parallel addition has cost

$$O((t + 2C_{comm}) \times \max(N, M))$$

and if the system achieves good memory balance then $N = n/p$ and $M = m/p$. The size of the problem can will be the maximum of n, m therefore the cost parallel polynomial addition becomes:

$$O(\frac{nt + 2nC_{comm}}{p})$$

The algorithm does not use a lot of auxiliary storage and the depth t in many cases will be shallow (if the system is used only to find the sum of two polyno-

mials then $t = 2$, but in a Gröbner base calculation t is equal to number of basis polynomials).

4.2.2 Polynomial multiplication

Given the sparse representation of polynomials, multiplication using the classical long multiplication algorithm gives acceptable performance: The distribution of data results in speedup by a factor of at least 2. However, the computation incurs high communication cost for storage allocation. To reduce the cost of communication and overlap computation and communication, the algorithm is placed under the management of a scheduler. Each step of the computation assumes a 'state' requiring some auxiliary working memory.

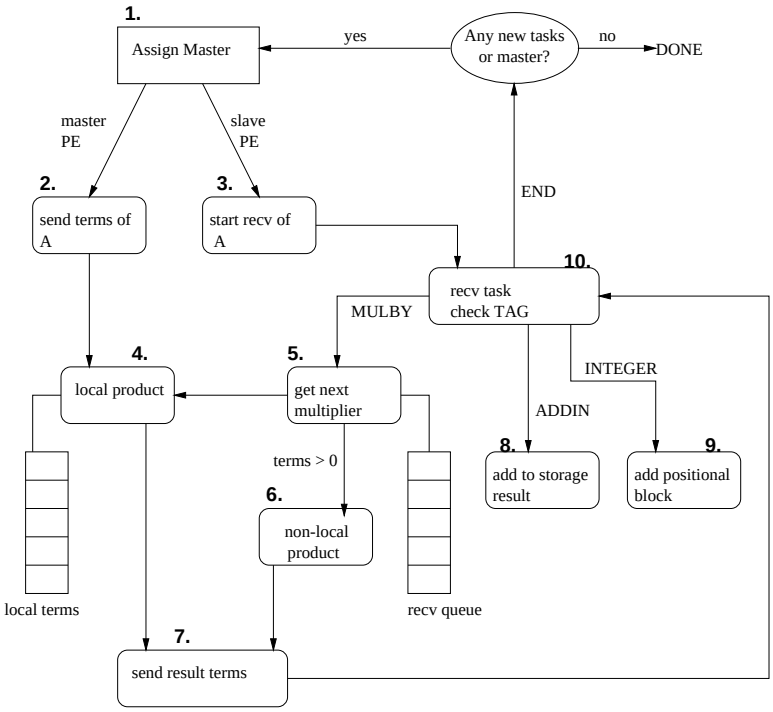


Figure 4.5: Parallel polynomial multiplication

The parallel multiplication tasks are shown in figure 4.5. The master token is assigned at state 1. The simplest assignment is round robin, however this may be changed. A master broadcasts data for new tasks in 2, and can then proceed to performing local work in state 4.

Slave processes have to allocate persistent receive buffers at state 3. This should ensure that when a master broadcasts new data, the slave is ready to receive. At the start of the computation, many of the messages received by the slave at state 10 are from the master, and have a special tag MULT. However, the main receiving task at 10 can also receive messages with several other tags to instruct the scheduler on which task to start up next.

The scheduler for a slave process can select a new term to multiply with at state 5. Data from the master has higher priority and a product is formed at 6. If

there is no data from the master then a local product (from terms already on the current PE) may be scheduled. New product terms are allocated storage space by the randomized hash partition algorithm at state 7.

The algorithm terminates when all the master tasks have been distributed and there are no outstanding messages in the communication pipeline.

The scheduler has some freedom in the above process to try to overlap communication and computation as far as possible. Thus there may be a wait associated with receiving task data from a master process, another task may be scheduled in the intervening time.

The scheduler can also take advantage of buffered communication to combine data to destined for a particular PE so that instead of sending several messages, only one message is sent. This is effective if the data is memory bound but not processor bound, so that the recipient does not suffer from the delay in the data. The impact of increased buffer size as shown in figure 4.6. This performance improvement may be attributed to the fact that an increase in the size of blocks reduces communication time for each multiplication.

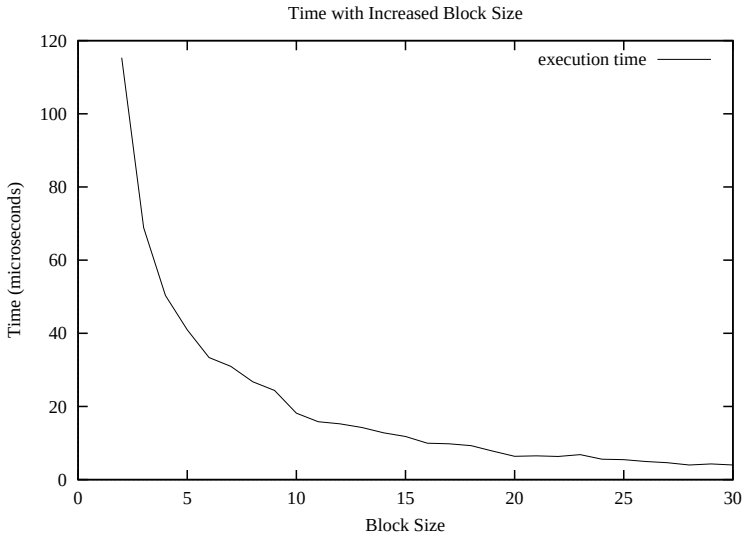


Figure 4.6: Execution time when increasing buffer size

4.3 Summary

Implementation of arithmetic for arbitrary size integers on a massively parallel processor has been described. The redundant signed digit representation is combined with randomized term distribution to support highly parallel forms of the classical algorithms. Carry propagation during addition is bounded therefore communication can be delayed until a final normalization step when the data is used. The polynomial arithmetic is dominated by multiplication. In this case the scheduler is used to overlap communication and computation. At potential delay points in

the algorithm, the scheduler can run some local tasks that are independent of any communication.

The integer and polynomial arithmetic algorithms described in this chapter make effective use of the distributed data structures. The global and local memory allocation procedures of the system give good memory balance. The use of the scheduler to overlap communication and computation is a useful tool for reducing the impact of the increase in number of communications due to fine granularity.

Chapter 5

Systems of polynomial equations

In this chapter, parallel algorithms for solving systems of polynomials equations are discussed. These problems require large amounts of memory and exhibit intermediate expression swell. Two classes of equations are considered:

- Sparse systems of linear equations. The main memory consuming function is the computation of determinant. In addition, sparse systems encounter 'fill in' of new entries which add to the demand for memory during a computation.
- Solution of non-linear systems makes use of the Gröbner base algorithm which requires large memory resources.

The above problems are capable of producing truly huge computations with significant amount of intermediate data. The aim is to parallelize familiar algorithms for solving sparse linear equations. In addition, the underlying polynomial arithmetic adds a level of fine grained parallelism to the Gröbner base algorithm.

5.1 Parallel Gröbner base algorithm

Consider a set of multivariate polynomials $F = \{f_1, \ldots, f_m\}$ where $F \subset K[x_1, \ldots, x_n]$. Let the ideal generated by the polynomials be

$$I = \langle f_1, \ldots, f_m \rangle = \sum_{i=1}^{m} q_i f_i \; q_i \in R$$

A homogeneous system formed by these polynomials may be written as

$$\begin{aligned} f_1(v) &= 0 \\ f_2(v) &= 0 \\ \vdots &\quad \vdots \\ f_m(v) &= 0 \end{aligned}$$

The *variety* of I denoted by $V(I)$ is the set of vectors v that satisfy equation (5.1). Thus

$$V(I) = \{v \in R : f_1(v) = f_2(v) = \cdots = f_m(v) = 0\}$$

The Gröbner base algorithm is often used to answer questions about the algebraic variety of a set of polynomials and also has wider applications in computer algebra. The sequential algorithm for computing the basis is given in section 2.7, and several parallel implementations have been discussed.

The parallelization of the sequential Gröbner base algorithm may occur in several stages:

- Parallelize the monomial ordering.
- Parallelize the selection of s-pairs.
- Parallelize the coefficient arithmetic.
- Parallelize the basis reduction.

The expensive operation in the algorithm is the reduction step where each s-polynomial is reduced by the entire intermediate basis. The multivariate reduction algorithm is dependent on the monomial order of terms which are distributed randomly and unordered. Therefore an intermediate procedure is required to transform our underlying unordered storage to some partial ordering of the monomial terms.

5.1.1 Partial ordering of pairs

The main loop in the Gröbner base algorithm is dependent on the selection of the next s-pair for the reduction. The simplest strategy of reducing all pairs is easy to parallelize. It can also be used with finer granularity. We do not incur the cost of finding an s-polynomial first, which is the case in the homogeneous methods [101].

The number of distinct pairs for a basis of size n is about $O(n^2/2)$ therefore we can make use of diagonalization to produce a natural ordering of the set of s-pairs with two important features:

1. The size n is used in a 'step test' to determine whether to move on to the next line. Therefore n can change dynamically between tests to accommodate growth in the basis.
2. Each processor can independently find the next pair without expensive communication.

Row algorithm

Let each new addition to the basis be added to the last row. Then the selecting a new pair consists of a walk below the diagonal for each row as shown in figure 5.1.

A new addition to the basis will be the last row and it will not have been visited yet (otherwise it would have been generated previously). Consider an example basis with four polynomials: the list of pairs in the order they are visited is $(1,0),(2,0),(2,1),(3,0),(3,1),(3,2)$.

The full algorithm is shown in figure 5.2. This will generate all possible s-pairs for the Gröbner basis. It has the following properties:

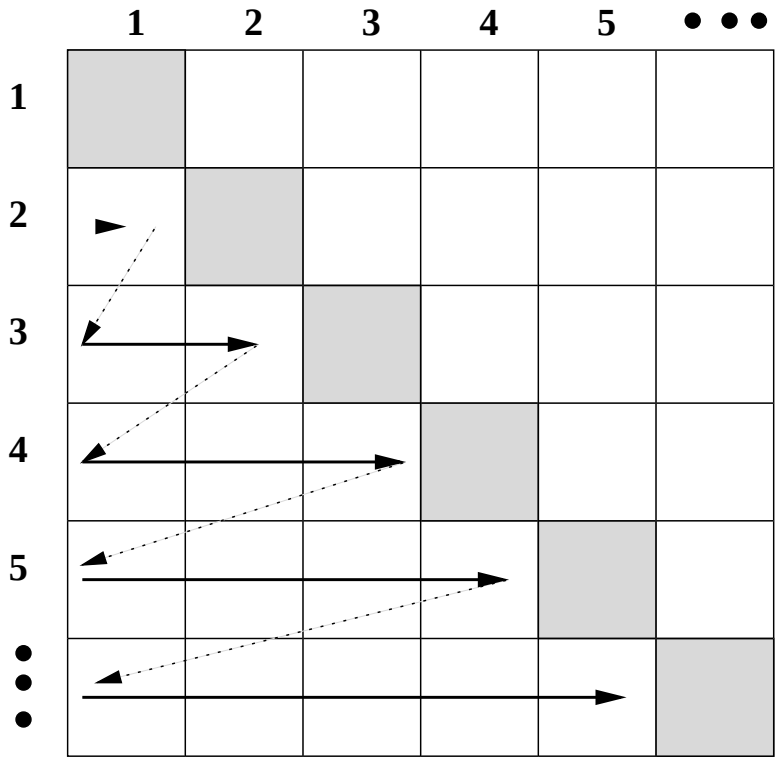


Figure 5.1: Ordering s-pairs below the diagonal

$i = 1;$

while $(i < n)$
    for $(j = 0;\ j < i;\ j++)$
        spair(i, j)
    $i++;$

Figure 5.2: Row algorithm for selecting critical s-pairs

- The number of polynomials in the current basis n, can grow dynamically before the next test for i at the *while* loop.
- Each row is bounded by the diagonal ($j < i$) therefore the algorithm will always proceed to the next row if it exists. This means that if any new basis polynomial is generated, we can be sure that the s-pairs induced by the new polynomial will eventually be considered.

This partial ordering can be computed concurrently on each processor. Therefore each PE can proceed independently if the data it requires is available.

Deletion of superflous pairs

The ordering of pairs that has been described does not take into account pairs that do not give meaningful new basis elements. The process of identifying pairs that can be deleted before the expensive reduction operation is performed can significantly decrease the execution time of the algorithm.

In the case of homogeneous polynomials, a partial ordering induced by the degree of the s-polynomial is available. Let $in(f)$ be the leading monomial of a polynomial f, and let the least common multiple of two numbers a, b be given by $lcm(a, b)$. Then the following deletion criteria may be formulated [101] where $0 \leq i, j, k \leq p$:

1. chain criteria: For $i, j < k$, if

   $$lcm(in(f_i), in(f_k))$$

   divides

   $$lcm(in(f_j), in(f_k))$$

   then delete (f_j, f_k).

2. product criteria: If

   $$lcm(in(f_i), in(f_j)) = in(f_i) \cdot in(f_j)$$

   then delete (f_i, f_j).

5.1.2 Finding the leading term

The need for a monomial order poses a particular problem within our distributed representation; terms from a single polynomial may be stored on different processors. Therefore a total order of the terms would incur high communication cost in transferring storage to a single PE.

However, observing that Gröbner base calculations and other algorithms make use of order information to determine the leading term only, the order may be relaxed to the weaker condition of finding the maximal term.

An algorithm based on the merge sort is a natural choice since the PEs have no shared address space. Figure 5.3 shows an online merge and broadcast procedure for identifying the leading term. At the end of the algorithm, each PE has a local copy of the leading term that is not permanently stored.

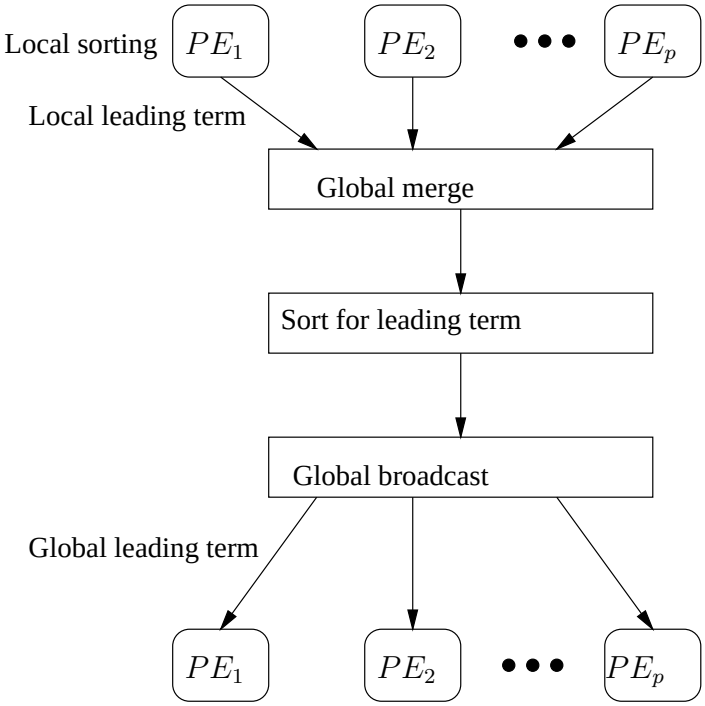


Figure 5.3: Selecting leading term of a polynomial

5.1.3 S-polynomial computation

In our algorithm, the terms of f_1, f_2 are distributed across several processors. This provides a way of computing the result of equation (2.10) in just 1 instruction instead of the $(mult, mult, add)$ sequence of the obvious scheme. Recall from section 2.7 that an s-polynomial h of the two polynomials f_1, f_2 is defined by

$$h = uf_1 + vf_2$$

where u and v are monomials which are formed from the least common multiple of the leading terms of the two polynomials.

The computation of s-polynomials only requires that the *leading monomials* be known. The order of the terms in the tail end of the polynomial has little impact on the computation. This observation enables us to relax the order requirement to just finding the maximum within a given order.

The distributed algorithm therefore has to first compute the multipliers u and v. Each PE finds a local leading term, and sends it to the master process. The master process then sequentially computes u and v. These can then be broadcast to all PEs.

The level of concurrency here is determined by the number of terms in f_1 and f_2. Since these are large polynomials, distributing within each s-polynomial rather than letting each processor compute a different s-polynomial leads to significant time efficiency in computing the products uf_1 and vf_1 since u, v have only one term. The multiplication can therefore be done in $O(n/p)$ time rather than $O(n)$ where n/p is the average size of each data partition and n is the overall problem size.

broadcast (u,v)
    the *MULBY* tag indicates multiplication by f_1 or f_2
    add into h
start next s-polynomial

Figure 5.4: Distributed s-polynomial computation

1. Is f reducible? The reducibility condition is used to check if $lt(f) \in L(G)$.
2. If f is NOT reducible, then return $f' = f$.
3. Otherwise f is reducible, and $\exists t_i$ such that $lt(f) = \sum t_i lt(g_i)$.
4. Let $f_1 = f - \sum t_i lt(g_i)$. Since $\sum t_i lt(g_i) = lt(f)$, the leading term of f is cancelled, so $\deg(f_1) < \deg(f_2)$.
5. Repeat until a reduced $f' = f_i \mod (g_1, \ldots, g_r)$, or a zero.

Figure 5.5: Reduction algorithm

5.1.4 Reduction algorithm

Consider a ring of multivariate polynomials $R = K[x_1, \ldots, x_n]$ with coefficients in the field K. Let $f \in R$ and $G = \{g_1, \ldots, g_r\}$ where each $g_i \in R$. Let $L(G)$ be the leading term ideal of G, that is, the ideal generated by $\{lt(g_i)$ such that $g_i \in G\}$.

Definition 5.1.1 (Reducible) *f is reducible modulo G if f is non-zero and $lt(f) \in L(G)$.*

The reduction algorithm then consists of constructing an f' such that

$$f = f' \mod (g_1, \ldots, g_r)$$

where f' is reduced modulo G. The reduction algorithm as in Gianni *et al.* [53] is given in figure 5.5.

The parallelization of the reduction algorithm in figure 5.6 makes use of the polynomial multiplication and addition algorithms. Some computation time may be saved in finding the leading term ideal by delaying the sequential merge operation.

5.1.5 Order

The hashing method for distributing data across the PEs creates an unordered storage. However, the Gröbner base algorithm makes use of the leading monomial hence

1. Use the algorithm in figure 5.3 to find the leading term of f.
2. Find a pseudo-leading term ideal of $pL(G)$ by only finding the local leading terms of the set of polynomials G (this makes use of only the first step in figure 5.3, without the merge).
3. Check in parallel if $lt(f) \in pL(G)$. If this condition is not satisfied then return $f' = f$.
4. If $lt(f)$ is in the pseudo ideal then use the leading term algorithm in figure 5.3 to find $L(G)$.
5. Perform the multiplication and summation $f_1 = f - \sum t_i lt(g_i)$ using the parallel algorithms from chapter 4.
6. If f' is non-zero then distribute its terms.

Figure 5.6: Parallel reduction algorithm

requires some ordering of the monomials. The computation is largely dependent on the ordering of the monomials. We have two options on how to proceed:

1. Keep the unordered data structure and provide an additional facility for maintaining order.
2. For Gröbner base computations, select a different data structure that simplifies order decisions.

Ordered representations

An alternative method of maintaining order is to select a different polynomial representation for computations that require order information. Two methods are provided: heap of terms, and a binary search tree (BST).

A term heap keeps the leading monomial (largest in a given order) at the root of the heap. For Gröbner base computations, this means an $O(1)$ retrieval of the leading monomials when forming an s-polynomial. The *heapify()* procedure is $O(n \log n)$ hence the high priority leading term can be found efficiently.

A BST sorts the terms of a polynomial and the leading monomials can be retrieved in $O(\log n)$ time. This is quite efficient for very large polynomials.

Ordered representations lose the relationships between terms within polynomials since each is kept in a separate data structure. Therefore the performance enhancement of fast addition in unordered storage is lost. However, this may be recovered in the efficiency of the retrieval operations and faster algorithms for addition or multiplication.

5.2 Parallel sparse linear systems

Sparse systems of linear equations provide several challenges for memory management:

- Managing 'fill in' of cells during computation. This will often result in significant increases in the memory requirements of a system.
- Partitioning data across several processors.

In this section, we consider parallel algorithms for matrices with polynomial entries. Basic linear algebra operations include algorithms for computing the determinant of large sparse polynomial matrices and finding the characteristic polynomial of a matrix. The parallel multiplication from section 4.2 is utilized.

5.2.1 Polynomial matrices

Sparse matrices have many zero entries therefore a representation that preserves memory by not storing empty cells is useful. The polynomial entries for the matrices are stored in a polynomial table as described in chapter 3. The matrix representation therefore keeps references into this polynomial table for every non-zero entry. This creates some flexibility in how the matrix is stored, since it does not directly hold the data. Some options have to be eliminated first: a block distribution of the matrix could clearly allocate an empty block (whose entries are all zero) to a PE, therefore an alternative is required.

The simplest solution is to let each processor hold the full matrix as references to the non-empty entries. Since the polynomials are all distributed, it is possible for a PE not to have any terms from a particular polynomial, but if it holds the full matrix indicating that such a polynomial entry exists, then operations on that entry are not affected.

Recall from chapter 3 the representation for polynomials. The sparse matrix is then represented by a list of handles to data for the non-zero matrix entries. An example is shown in figure 5.7.

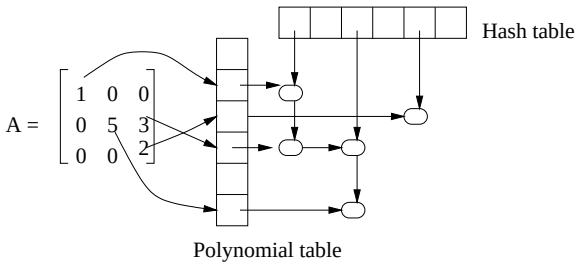


Figure 5.7: Sparse matrix representation

5.2.2 Parallel Bareiss algorithm

To parallelize the algorithm by Bareiss [10, 66], we would like to loosen the data dependency between successive iterations in the algorithm 2.7 so that concurrent execution becomes possible. Recall that Bareiss algorithm is based on sub-determinants of the form in equation (5.1).

$$a_{ij}^{k} = \begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1k} & a_{1j} \\ a_{21} & a_{22} & \cdots & a_{2k} & a_{2j} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{k1} & a_{k2} & \cdots & a_{kk} & a_{kj} \\ a_{i1} & a_{i2} & \cdots & a_{ik} & a_{ij} \end{vmatrix} \tag{5.1}$$

Also note that this method is for Gaussian elimination, therefore at the k-th iteration some of the entries below the diagonal are already zero. A modified derivation of Bareiss method emphasises independent 2×2 matrix determinants embedded in the algorithm. Using additional observations from Smit [111] the exact division can also be performed internally. Therefore the parallel implementation consists of:

- The 2×2 determinant as the main independent operation which can be performed in parallel.
- Parallel polynomial arithmetic for computing the 2×2 determinant in two parallel multiplications and one parallel subtraction.

The derivation manipulates Gaussian elimination where U is a lower triangular matrix

$$U = \begin{pmatrix} 1 & & & \\ u_{21} & 1 & & 0 \\ u_{31} & & \ddots & \\ \vdots & & 0 & \\ u_{n1} & & & 1 \end{pmatrix}$$

and the entries of U are

$$u_{j1} = \frac{-a_{j1}}{a_{11}} \; j = 2, 3, \ldots, n.$$

Expanding the product UA leads to a determinant form of Knuth [77]:

$$UA = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ 0 & a_{22} - a_{12}(a_{21}/a_{11}) & \ldots & a_{2n} - a_{1n}(a_{21}/a_{11}) \\ 0 & a_{32} - a_{12}(a_{31}/a_{11}) & \ldots & a_{3n} - a_{1n}(a_{31}/a_{11}) \\ \vdots & \vdots & & \vdots \\ 0 & a_{n2} - a_{12}(a_{n1}/a_{11}) & \ldots & a_{nn} - a_{1n}(a_{n1}/a_{11}) \end{pmatrix} \tag{5.2}$$

The product matrix can be partitioned in the following way

$$UA = \left(\begin{array}{c|c} a_{11} & a_{12} \ldots a_{1n} \\ \hline \mathbf{0} & A_1 \end{array} \right) \tag{5.3}$$

Recall the usual formula for the determinant of a 2×2 matrix:

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{12}a_{21} \tag{5.4}$$

Now consider the entries in the submatrix A_1 in equation (5.2). The entry $[2, 2]$ is

$$a_{22} - a_{12}\frac{a_{21}}{a_{11}} \tag{5.5}$$

If we could multiply equation (5.5) by a_{11}, then this would give the formula in equation (5.4). To achieve this, we recall the determinant property that if one row or column is multiplied by a factor α then the value of the determinant is also multiplied by α. Hence multiplying the $n-1$ rows of a matrix gives

$$det(\alpha A) = \alpha^{n-1}\ det(A) \tag{5.6}$$

So let $\alpha = a_{11}$ and multiply rows $2, \ldots, n$ of the matrix by a_{11} to get

$$det(UA) = \frac{1}{\alpha^{n-1}} det(\alpha UA)$$

Rearranging the terms leads to the following alternative determinant presentation by Smit [111]:

$$det(UA) = \frac{1}{a_{11}^{n-1}} \begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} & \cdots & \begin{vmatrix} a_{11} & a_{1n} \\ a_{31} & a_{3n} \end{vmatrix} \\ \vdots & \ddots & & \vdots \\ 0 & \begin{vmatrix} a_{11} & a_{12} \\ a_{n1} & a_{n2} \end{vmatrix} & \cdots & \begin{vmatrix} a_{11} & a_{1n} \\ a_{n1} & a_{nn} \end{vmatrix} \end{vmatrix} \tag{5.7}$$

Now the calculation of a 2×2 determinant in equation (5.4) can be achieved relatively cheaply with two polynomial multiplications and one polynomial addition in parallel using our implementation in chapter 4.

In the Bareiss notation, let a_{ij}^k be the element $[i, j]$ after k iterations. After successive iterations of the form in equation (5.7), we can see that the sub-determinants are based on calculation of 2×2 determinants. For example the first three entries on the main diagonal are:

$$a_{11}^1 = a_{11} \tag{5.8}$$

$$\begin{aligned} a_{22}^2 &= \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} && (5.9) \\ &= a_{11}a_{22} - a_{21}a_{12} && (5.10) \\ &= a_{kk}^k a_{ij}^k - a_{ik}^k a_{kj}^k && (5.11) \end{aligned}$$

$$
\begin{aligned}
a_{33}^3 &= \begin{vmatrix} \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{13} \\ a_{21} & a_{23} \end{vmatrix} \\ \\ \begin{vmatrix} a_{11} & a_{12} \\ a_{31} & a_{32} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{13} \\ a_{31} & a_{33} \end{vmatrix} \end{vmatrix} && (5.12) \\
&= \begin{vmatrix} a_{22}^2 & a_{23}^2 \\ a_{32}^2 & a_{33}^2 \end{vmatrix} && (5.13) \\
&= a_{22}^2 a_{33}^2 - a_{32}^2 a_{23}^2 && (5.14) \\
&= a_{kk}^k a_{ij}^k - a_{ik}^k a_{kj}^k && (5.15)
\end{aligned}
$$

Equations (5.11) and (5.15) derive the main iteration in Bareiss algorithm:

$$a_{ij}^{k+1} = a_{kk}^k a_{ij}^k - a_{ik}^k a_{kj}^k \tag{5.16}$$

In this case the exact division is delayed until the main diagonal has been completed and cancellation by the fraction part

$$\frac{1}{a_{11}^{n-1} \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix}^{n-2} \quad \cdots} \tag{5.17}$$

eliminates entries and the Bareiss algorithm returns the determinant

$$det(A) = \frac{1}{a_{11}}\, a_{nn}^n \tag{5.18}$$

where a_{nn}^n is the entry at (n, n) after the n-th iteration.

5.2.3 Parallelization of recursive minor expansion

Our approach to parallelizing the recursive minor expansion algorithm is to maintain high level sequential operation, but parallelize the underlying arithmetic. We assume that the multivariate polynomials within the matrix are very large, so that the cost of arithmetic is the overriding concern.

The recursive minor expansion algorithm for the determinant of an $(n \times n)$ matrix A is given by

$$
\begin{aligned}
|A(1 \times 1)| &= a_{11} \\
|A(n \times n)| &= \textstyle\sum_{k=1}^{n} a_{jk} C_{jk}
\end{aligned}
$$

where the cofactor C_{jk} is given in terms of the $(n-1) \times (n-1)$ minor M_{jk}:

$$C_{jk} = (-1)^{j+k} M_{jk}.$$

To parallelize the recursive minor expansion algorithm, we first observe the recursion tree for the algorithm. Figure 5.8 shows an example of the minor expansion tree for a matrix of dimension 3. The leaf nodes are 1×1 determinants and a parent node is multiplied by each of its child nodes.

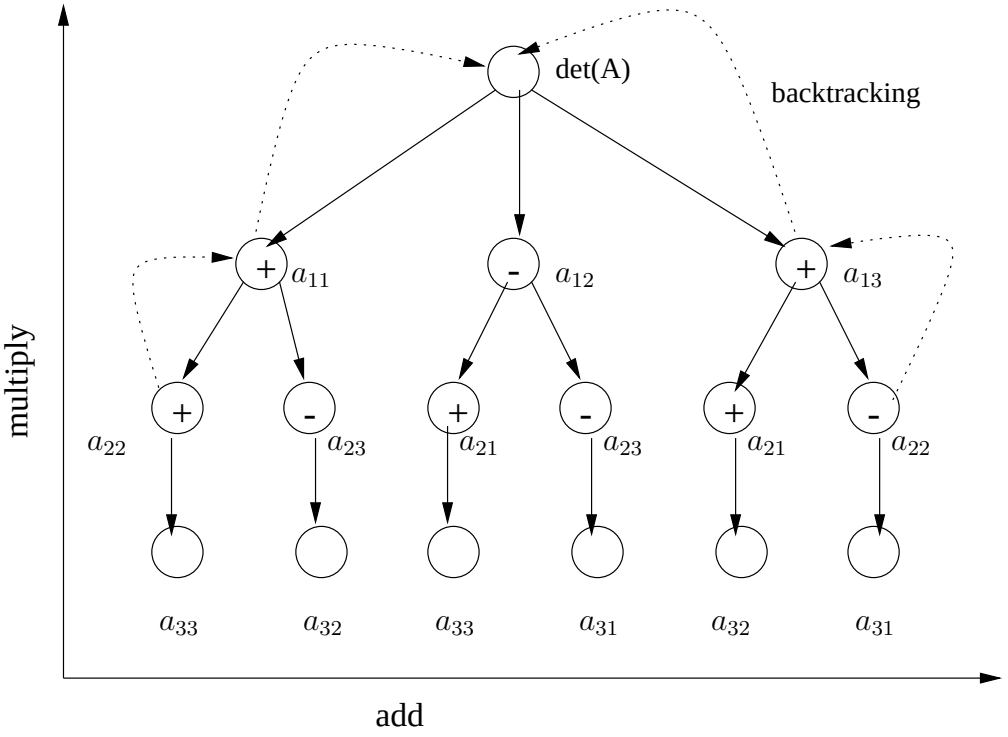


Figure 5.8: Recursion tree for minor expansion

depth	cost
0	$\lvert A(n,n)\rvert$
1	$(n-1)\,\lvert A(n-1,n-1)\rvert$
2	$(n-1)(n-2)\,\lvert A(n-2,n-2)\rvert$
$\vdots$	$\vdots$
$n-1$	$(n-1)(n-2)\cdots\lvert A(1,1)\rvert$

Figure 5.9: Costs at different levels of recursion tree

The tree has depth $O(n)$ since each minor is computed from a sub-matrix in which one row and one column are deleted. A parent node at depth d where $0 \leq d \leq n-1$, has $n-1-d$ children. The cost of each level of the tree is shown in figure 5.9.

The sequential cost of the algorithm is $O(n!)$. The recursion tree shows two clear options for parallelization aligned with different methods for traversal of the tree:

1. A *depth-first* traversal of the tree follows the leftmost path to a leaf node. This gives an algorithm where the minors are calculated sequentially. The opportunity here is to parallelize the backtracking step.

2. A *breadth-first* traversal of the tree visits each node at depth d before moving on to the next level at depth $d+1$. Parallelization assigns each branch of the tree to a different processor and calculates the minors in parallel.

5.2.4 Parallelization by breadth-first traversal

A parallel implementation of breadth-first traversal calculates all the minors at level d concurrently to satisfy the breadth-first condition. Each branch of the recursion tree may be assigned to a different processor to increase the parallelism and allow asynchronous computation of minors. Once a minor has been computed, the result is communicated to the parent node with a tree synchronization step (see section 2.2.7). Then the parallel cost can be calculated as

$$T_p = \max\left(\sum_{d=0}^{n-1}(n-d)T_{d-1}\right).$$

where T_{d-1} is the time to traverse a branch at depth $d-1$ and there are $n-d$ multiplications for each completed branch.

The main advantage of a breadth-first algorithm is that given enough processors the data distribution can be done dynamically and efficiently. The mapping of a matrix entry to a PE for multiplication is performed by the scheduler. For example $a_{11} \mapsto PE_1, a_{12} \mapsto PE_2$, and so on. The mapping performs a dynamic allocation of data therefore large data sets can be used.

However, breadth-first suffers from a requirement for a large number of processors ($O(n^2)$ in the worst case) and inefficiency due to the idle processor time until the depth for that particular PE is reached.

In addition, breadth-first traversal commits to evaluating every minor. This has implications in a sparse system, since sparse systems display numerous equal minors. In a breadth-first system this sparsity is not fully exploited and nested minors can be recomputed.

5.2.5 Parallelization by depth-first traversal

A depth-first traversal has two components:

- A forward step to discover the nodes in the leftmost path until the maximum depth.

- A backtracking step to complete a visit to a node when all nodes on its adjacency list have been examined.

The main objection to a depth-first traversal is that it has the forward sequential component For the effect of the sequential part to be reduced, the individual minor computations have to be very large and therefore offset the cost of committing all parallel resources to computing one minor at a time. This condition is satisfied by our target problems which are sparse matrices where each polynomial entry in the matrix is very large. Hence applying depth-first with parallel polynomial operations is justified.

We can therefore parallelize the polynomial arithmetic in the backtracking stage. The backtracking stage of this algorithm performs $n - d - 1$ multiplications and at depth d. These operations can be parallelized using the algorithms in section 4.2.

The scheduler for the depth first recursive minor expansion is shown in figure 5.10. This controls the forward step for visiting minors. This sequential component essentially synchronizes all PEs to participate in finding the next minor. The parallel stage performs polynomial operations, updates supporting data structures and invokes a procedure to mark the completed minor. The scheduler can then synchronize the processes to select the next minor.

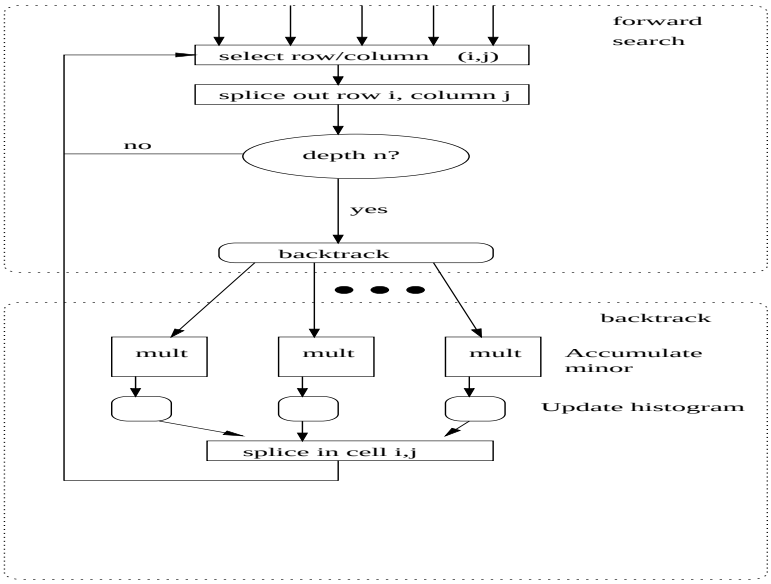


Figure 5.10: Parallelizing recursive minor expansion

The main advantage of the depth-first algorithm is that computed minors will be known at the time that a minor M_{jk} is computed. Therefore we can make use of Smit's *histogram* to avoid recomputing minors.

5.2.6 Zero minors

Large sparse systems have many hundreds of zero entries. A minor can have a zero row or column due to deletion of the non-zero elements. Efficient minor expansion

relies on support data structures for identifying zero minors and selecting the best row (or column) to expand. A minor that contains a zero row (or column) is equal to zero (by expanding along the zero row). If such cases are identified then the system can save time by returning zero immediately.

The difficulty here is that one PE may not hold any terms of a particular matrix entry, but a complete sweep of the hash table to find empty cells is too expensive.

An *occupation list* keeps Boolean indicators that show whether a polynomial exists on other PEs for that particular row. This is performed by capturing the terminating messages with the 'end' tag. If such a message corresponds to an empty entry in the occupation list, then the occupation list is updated.

5.3 Summary

Solving sparse systems of linear equations with entries from a polynomial field is a problem that incurs irregular data access has the potential for matrix 'fill in'. The randomized distribution of data and parallel field arithmetic have been applied to the parallelization of the standard determinant algorithms. Calculation of the determinant is the main step in solving linear systems. The implementation also has wider applicability to finding the characteristic polynomial of a system of linear equations.

We have also applied the same randomized memory allocation algorithm in a parallel implementation of the Gröbner base algorithm. Our interest in this algorithm is due to its resource requirements. Again we have focused on the design implications when extreme fine tuning of memory allocation is employed. Our basic implementation is interesting for its handling of order within a distributed representation, and ability to cope with increase in run time memory demands.

Our derivation of parallel algorithms is *operational* in that we rely on parallel operations such as polynomial multiplication or a 2×2 determinants rather than a higher level structural parallelism. The polynomials in our target applications are large enough to make this operational parallelization of low level operations viable.

Chapter 6

Conclusions

In this thesis we have advanced the case for deeper understanding of the memory requirements of algebraic algorithms and how to meet these. We have started from the perspective that the pursuit of speedup of computation time has been widely investigated. Therefore the key motivation in this work has been balanced and efficient use of distributed memory. An architecture to meet the peak memory demands of symbolic algebraic algorithms has been proposed. We have argued that an efficient and robust memory management scheme incorporates the following features:

- Dynamic memory allocation, to catch any peaks in intermediate stages.
- Distributed representations and data structures, so that a polynomial is not kept all in one processor, and also not replicated on all processors.
- Fine granularity, to give greater flexibility in tuning the allocation.
- Unordered storage, therefore dynamic partial ordering algorithms are required for problems such as the Gröbner base algorithm.

6.1 Contributions

The phenomenon of intermediate expression swell has been observed from the very earliest computer algebra systems. The main theme throughout this thesis has been the requirement to manage intermediate expression swell and accommodate the peak memory demands of computer algebra computations. This thesis has attacked the problem through dynamic memory distribution with fine granularity. The following contributions are highlighted:

1. A randomized algorithm for dynamic memory distribution. This method works by determining storage location of data at the time the data is generated. A key differentiator has been the fine granularity which greater flexibility and fine-tuning of the memory allocation.
2. Asynchronous scheduling to reduce the impact of high latency communication. In addition, a traffic model for the communication pattern has been

described for computation of probabilistic weights for channel selection during communication. The system guarantees deadlock-free communication and small requirements for send and receive buffer space.

3. Implementation of parallel polynomial and integer arithmetic. The system relies on this operational level parallelism for higher applications. This approach is justified by the huge size of polynomials generated within the system, therefore parallelising the field operations leads to an efficient implementation.

4. Application to two computer algebra problems with huge memory requirements and irregular data access patterns. Implementations for solving large sparse systems of linear equations and for parallel Gröbner base computations have been described.

6.2 Future directions

A framework for integrated memory management in parallel algebraic computation has been advanced through this research. The interesting questions now relate to how far such a memory-centric approach can be applied. We have seen that this approach is particularly suited to some computer algebra problems because of their vulnerability to intermediate expression swell. The next step is to identify further application areas, and more than that, to identify potential concurrency within such applications. Further investigation is required in the following key areas:

1. Garbage collection. The current work has focused on memory allocation and not deallocation. This is because we have mainly considered the case where the large storage requirement is for persistent data. However a good garbage collector for deallocating unused memory can free up significant intermediate storage. This presents a technical challenge for global garbage collection when data may be referenced by distinct processors.

2. Partial order within a distributed representation. An early design choice for this system was unordered storage. Therefore it was necessary to implement partial ordering for the Gröbner base algorithm where the leading term of a polynomial is needed. Maintaining order information in a distributed system where the constant communication results in low data stability is a challenge. The first steps have been made within our Gröbner base computation in section 2.7 and a plug-and-play facility permits a replacement with an ordered representation such as a heap if necessary. For further examination of this problem, we propose to pursue partitioning algorithms that link the topological mapping to the order of data.

3. Advanced parallel algorithms. We have implemented practical standard algorithms on a massively parallel processor. From this we have gained insight into the memory requirements of parallel systems. How much more can be gained with the parallelization of more advanced algorithms but with a similar memory scheme? Consider for example, how FFT multiplication would benefit given our dynamic memory allocation.

4. Portability to Beowulf clusters. This research has been targeted at supercomputers, however, there has been recent growth in the use of Beowulf clusters of workstations. Cluster technology promises better price/performance for parallel machines hence we are confident that parallel systems for problems such as computer algebra will be in high demand for these emerging architectures. Portability has been achieved with the use of the standard MPI library, however we would have to re-examine the impact of fine granularity to account for the higher latency for communication within Beowulf clusters.

The motivation for this work was drawn from definitive computer algebra challenges set in 1990 [14]. A new compilation of urgent research challenges for computer algebra set out in the year 2000 [72] for a new decade of research, does not include parallel systems. In the intervening years there has been much progress in parallel computer algebra systems, but many interesting problems remain. As we have seen, the parallel memory balancing problem is rich in subtle problems and further work along the lines suggested here has the potential to improve our parallel systems.

6.3 Conclusions

Efficient memory balance in a distributed memory system cannot be achieved with static data partitioning algorithms. This thesis makes use of dynamic memory allocation procedure based on randomized hashing.

The aim has been to keep all the processors in a parallel system balanced in terms of the amount of data stored on each. Balancing the memory load takes into consideration the principles of data locality in order to achieve load balancing of the CPU time as well. The randomized memory allocation is fine grain, allowing finer control over the distribution and load metrics.

The implementation on a Hitachi SR2201 has been used to support the *peak* memory demands of some computer algebra applications such as Gröbner base calculations and large sparse systems of linear equations.

Memory availability and performance is particularly important for computer algebra problems as they frequently suffer intermediate expression swell. The techniques for distributed memory management that have been applied in this thesis show potential for wide application in computations that have huge memory demands.

Appendix A

CABAL: a short manual

This appendix describes the implementation of CABAL version 2.0 (Cambridge and Bath algebra system). It gives an overview of the system packages and the specification of the main interfaces.

The first version of CABAL by Norman and Fitch [93] was written as a small system for polynomial algebra. It used the PVM (parallel virtual machine) communication library [51, 50].

This report is based on our new development of the system [88] based on the message passing interface (MPI). In addition, the memory model has been extended. New packages for multiprecision integers, matrix algebra and Gröbner base computations have been added. As CABAL 2.0 is a much larger system, a restructuring to move some of the facilities into reusable application libraries has been introduced.

The system is written in the C programming language. The development platform consists of a network of workstations and a dual processor Pentium PC running Linux. The MPI implementation installed here is the *mpich* [57, 56]. The production testing is carried out on a Hitachi SR2201 which runs a high performance version of Unix (HI-UX/MPP) and vendor implementation of MPI for massively parallel processors.

The firmly held view in this study has been that the key issues in parallelism and computer algebra are best discussed in the context of a real implementation. The fundamental concepts and significant aspects of the design rationale have already been described in the main body of the dissertation. This manual is useful in that it presents the finer implementation details that can help clarify the points made earlier, as well as helping programmers who use the system or do parallel programming in general.

The implementation is organized in a few packages and a small number of interfaces. The main part of the system (the kernel) consists of the randomized hashing distribution subsystem which directly interacts with MPI in allocating data to different processors, and at the higher level implements the algorithms for computing the target storage location at very fine granularity for each term. MPI is described further in appendix D, however we may reiterate that the implementation needs the following capabilities from the communication library MPI:

- Assignment of unique identifiers to a fixed number of processing elements in a set topology.

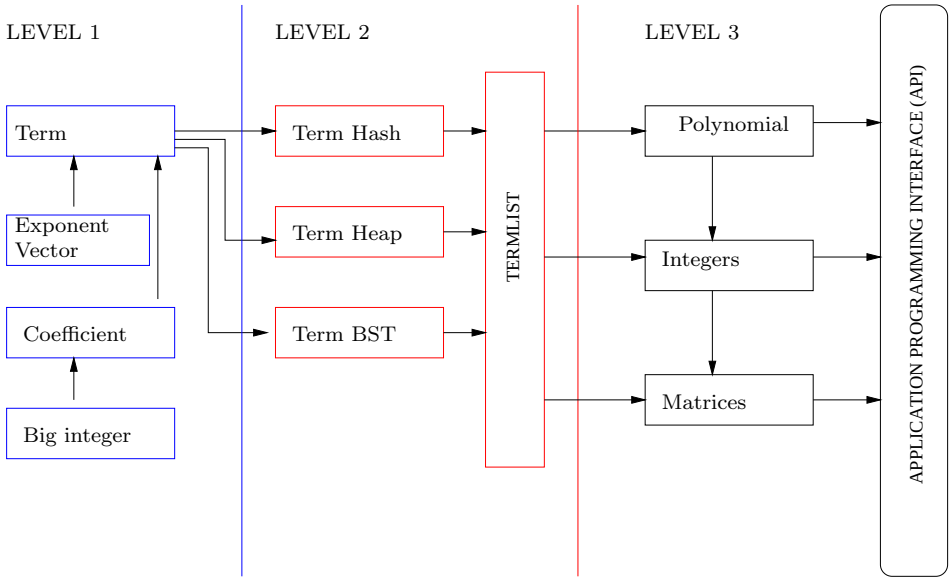


Figure A.1: Application programming interface

- Point-to-point and collective communication enabling send and receive of data between any two processing elements.
- Persistent communication capability which is essential to the efforts in overlapping communication and computation.
- A facility for packing data into one message.

CABAL can provide an application programming interface (API) for parallel computer algebra programming in C. The simplest use of the system is as an operational system with state. This means that CABAL will handle initialization and communication and global facilities such as polynomial and hash tables.

A.0.1 Incorporating the Library

The CABAL library may be used in any C program in an environment that includes the MPI message passing library. The main CABAL interface can be included through the header file `cabal.h` , therefore at the beginning of a C program the include directives will be `stdio.h, mpi.h, cabal.h` .

The main interface *cabal* is supported by several packages for integer, polynomial, matrix and Gröbner base computations. The application programming interface is shown in figure A.1.

A.1 CommSystem interface

The *CommSystem* interface interacts with MPI initially to set up the system by calling `MPI_Init()` to initialize MPI, and other initialization procedures such as

`MPI_Comm_rank()` to assign each processing element a unique identifier. It also sets up the log files and basic CABAL options. A user may run CABAL with some random polynomials and specify the command line options **-t** for the number of terms in a polynomial, **-r** to specify a random seed, **-d** for the maximum degree of each polynomial. Other options include **-n** to control how many runs of the program to execute when collecting data, **-b** to increase or decrease the number of buffers for block communication which may improve communication time.

This package is kept very small so that many of the algebraic facilities are in the application layer and can be reused. The main error handling facility is provided by *CommSystem*. The procedures in the system may raise an error with a given code depending on whether it is an algebraic error (such as division by zero), a terminal system error (such as running out of memory), or several input/output errors which may be encountered when the user makes a data entry error.

A.2 Polynomial interface

The polynomial interface provides a portable interface for polynomial operations. The interface works with *handles* for polynomials which are indices into a global polynomial table. This provides a 'plug-and-play' module where the underlying storage allocation of a polynomial may change. The interface provides the following procedures for polynomial operations:

- `int polyNew()` Get a new handle for a polynomial.
- `polyAdd(int r,int a, int b)` Add polynomials a and b and return the result r.
- `polySubtract(int r, int a, int b)` Subtract b from a and the result in r.
- `polyMult(int c, int a, int b)` Multiply a and b and put the result in c.
- `polyGetLeadingTerm(int p,Term *t)` Get the leading term of a polynomial in a set standard order: lexicographic, graded lexicographic or reverse graded lexicographic order.
- `polyPrint(int p, FILE *printfile)` Print a polynomial p to a file.

The main implementation of the polynomial interface is the *TermHash* implementation which provides a level 2 hash table for local term storage. Terms are linked in both the polynomial and hash tables giving flexibility and speeding up operations such as polynomial addition.

Although we have barely made use of the plug-in facility, it is useful to take advantage of it to provide a different polynomial local storage facility. For example, while our work in the main thesis body uses a level 2 hash table for local storage, it may be faster to keep the terms of a polynomial in a heap so that the operation of finding a leading term is faster as this is essential in Gröbner base calculations. To

do this, one plugs in the *TermHeap* implementation of the polynomial interface to replace the hash representation.

Other plug-ins such as a *TermBST* may be developed for suitable cases where terms in a binary search tree provide fast total ordering. Some computer algebra systems use such ordered representations for special operations.

A.3 Bignum interface

The main type for a multiprecision integer is a 4-word array forming a digit. In a polynomial, a term with a long coefficient of more than one digit has a special place in the exponent vector to indicate the position of the digit in a 'bignum'.

- `add37(unsigned int *a, unsigned int *b)` This computes $a = a + b$ where a and b are single digits (4 words) of a multiprecision integer in the range $-\beta \leq a, b \leq \beta$. The carry is added to $a[4]$ until a normalization step.
- `subtract37(unsigned int *c,unsigned int *a, unsigned int *b)` Subtract $c = a - b$. This will first negate b and use the addition code.
- `mult37(unsigned int *ap1, unsigned int *bp1,unsigned int *c0)` This multiplies two 4-word digits $ap1, bp1$ and returns $c0$ of double length, with a high digit $c0[4], \ldots, c0[7]$ and a low digit $c0[0], \ldots, c0[3]$.
- `divrem37(unsigned int *ap,unsigned int *bp, unsigned int *c)` Division with remainder.
- `exactdiv37(unsigned int *ap,unsigned int *bp, unsigned int *c)` Exact division of ap by bp.
- `print37(unsigned int *ap, int lz)` Print a multiprecision integer. If lz is set, then print exactly 37 decimal digits using padded zeros if necessary. The 128-bit representation can hold 37 decimal digits therefore this is evident here.

A.4 Grobner interface

The *Grobner* interface provides facilities for Gröbner base computations. A basis table is kept, which indicates which of the polynomials in the normal polynomial working table are part of the basis (at the start, all the polynomials may be part of the input basis). The input polynomials should be placed in the basis table at initialization and any further changes to the basis table is made only through the `grobner()` procedure. The main procedures are:

- `grobner(int order)` Find the grobner basis induced by the initial polynomials in the basis table.
- `grobSpair(int *f,int *g)` Selects the next s-pair by walking below the main diagonal of the matrix formed by the current basis elements.

- `grobSpoly(int spoly)` Creates a new s-polynomial. It calls *grobSpair* to select the next s-pair from the current basis elements.

- `grobReduce(int p, int r)` Reduces the polynomial indicated by the handle p, with respect to the current basis. The polynomial p is an s-polynomial for a Gröbner base computation. The reduced polynomial is returned in r. Note that in the case of just 2 polynomials in the input, the reduction can be used to find the *gcd* of the two polynomials.

The result is that on exit from `grobner()`, the basis table holds the reduced Gröbner basis. The Gröbner base package makes use of the parallel polynomial arithmetic in the polynomial interface. Therefore there are at least two levels of parallelism: parallelizing the polynomial arithmetic, and parallelizing the reduction operation.

Appendix B

The Hitachi SR2201

In any discussion of parallel systems and especially their performance, the target machine has a large impact as the architectural specification may result in special (non-portable) instructions. This leads to some difficulty comparing performance and determining progress.[1]

In this chapter we will describe the architectural features of the Hitachi SR2201, and the configuration as installed at the University of Cambridge high performance computing facility (HPCF). The SR2201 is a massively parallel processor (MPP) with scalable configuration capable of supporting 8 to 2048 processors. The Cambridge system has 256 processors. This discussion relies mainly on the work of Fujii *et al.* [47] concentrating on the hardware and the operating system interface. Additional information is drawn from lecture notes and user manuals for the Cambridge HPCF [120].

B.1 The processor

Each processor on the SR2201 is a 64bit RISC processor with a clock speed of 150MHz. The architecture is enhanced with two floating point pipelines and one load/store pipeline. The peak performance is over 76MFLOPS.[2]

The pipeline

Consider two large vectors $A(1,n)$ and $B(1,n)$ for n very large. The code for the vector inner product of A and B is shown in figure B.1.

[1] Performance comparison between parallel implementations is notoriously prone to error [9] due to difficulty of matching the many parallel architectures.

[2] An estimated factor of 2 faster than the Cray T3D.

```
for k = 1 to n step 1 do
    S = S + A[k] * B[k];
```

Figure B.1: Vector inner product

$PreloadA[1] \rightarrow FPR8$	
$PreloadB[1] \rightarrow FPR9$	
$PreloadA[2] \rightarrow FPR10$	
$PreloadB[2] \rightarrow FPR11$	
$\vdots$	
$PreloadA[40] \rightarrow FPR86$	
$PreloadB[40] \rightarrow FPR87$	
Label A:	
$FMPYFPR8, FPR9 \rightarrow FPR9$	A[k] * B[k]
$PreloadA[k+40] \rightarrow FPR88$	
$FADDFPR9, FPR7 \rightarrow FPR7$	S = S + A[k]*B[k]
$PreloadB[k+40] \rightarrow FPR89$	
$FWSTPSET$	window switch
$COM1B$	branch to label A

Figure B.2: Object code for vector inner product

An optimized code scheduler for the inner product generates the object code [62] shown in figure B.2.

The loading of $A[k]$ and $B[k]$ takes 2 clock cycles and the multiplication takes 1 clock cycle therefore this loop has a theoretical peak performance of 150 MFLOPS.

B.2 Pseudo vector processing

The SR2201 provides an innovative performance enhancing feature called the pseudo-vector processor (PVP). This is a key performance driver especially for numerical calculations as it delivers levels of performance comparable with vector processors on some vector code. Pseudo-vector processing creates cache bypass for pre-loading and post-storing floating-point data for array sizes far beyond secondary cache size. For long vectors, this reduces the cache miss rate and contributes to significant performance improvement for large-scale numerical computation [47, 75].

The ALU addresses 32 registers, but the PVP function requires many target registers for pre-loading, therefore the SR2201 has 128 physical floating point registers (FPR) per PE. The pseudo-vector processing function works by pre-loading (analogously post-storing) data into all 128 registers.[3] This is possible since there are different floating point and load/store pipelines, which means a *load* instruction can be issued in the same clock cycle as a floating-point operation.

Once the data is loaded, the PVP operates a *sliding window* over the registers to select the logical 32 processors for current operations [47]. This is shown in figure B.3

Having a sliding window means that only a few changes are made to the conventional instruction set architecture for the PA-RISC architecture on which the processor is based, since the CPU still addresses 32 registers. Additional instruc-

[3]Effectively creating a 128 stride access compared to unit stride on normal uniprocessor CPUs and 2 stride cache pre-fetch on the CRAY T3D.

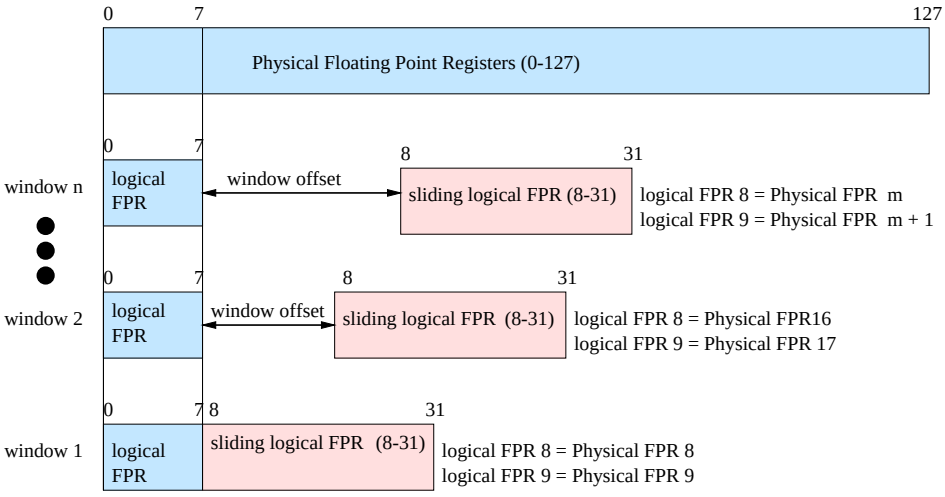


Figure B.3: Sliding windows for pseudo-vector processing

Instruction Usage	**Meaning**
Preload A[40] →FPR122	load A[40] into register 122
Poststore FPR88 →C[10]	store contents of register 88
FWSTPSET	*Window switch* with offset

Figure B.4: Some additional instructions in SR2201

tions are needed for a few operations such as shown in figure B.4.

B.3 Interconnection network

The processors are arranged in a three dimensional switching crossbar communication link (hyper-link) capable of peak bandwidth of 300MB/s.

Each node connects to the crossbar through 3 dual communication ports: X,Y,Z as shown in figure B.5. The X-crossbar switch can switch up to 8×8 connections. The Y-crossbar and the Z-crossbar can switch up to 16×16 connections.[4]

The crossbar can switch to connect each node directly thus giving a virtual totally connected network, but at a fraction of the cost. A communication path between any two processors within the network has length less than 3.

The 3D crossbar facilitates hardware-based global communication. The *barrier synchronization* and *one-to-all broadcast* are both performed in hardware taking advantage of the network topology and therefore achieve very low latency.

However, the crossbar cost does not scale well for large number of processors therefore the cost of the machine is dominated by the cost of the intercommunication network [83]. An intermediate network configuration that has much of the functionality of the 3D Crossbar at less cost is the idea of a *multistage interconnec-*

[4]This gives a total number of processors up to $8 \times 16 \times 16 = 2048$.

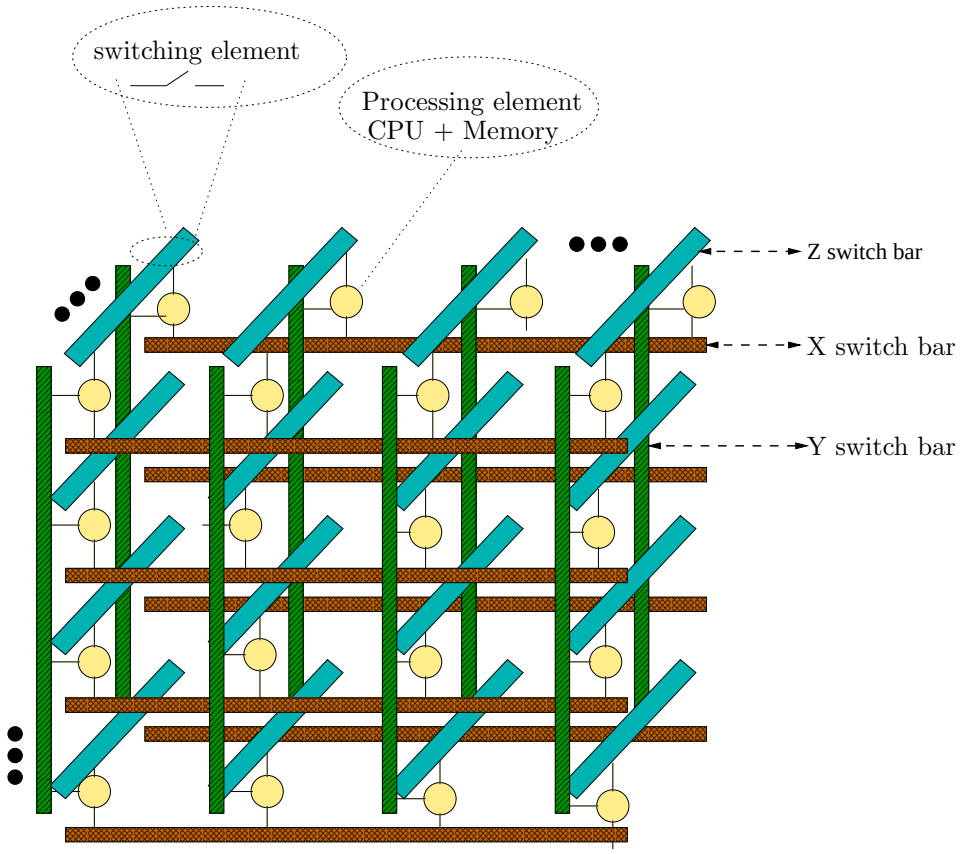


Figure B.5: 3D crossbar switching network

tion network [125].[5] Reconfigurable networks [58] based on crossbar switching are alternative designs that give good flexibility and mapping of several topologies.

B.4 Mapping of processors to topology

The 3D crossbar network is supported by robust routing through the switching network [126]. This makes the Hitachi SR2201 very flexible and conducive to mapping algorithms for different network topologies. Programs requiring ring, mesh, hypercube or tree-connected networks can be mapped onto the 3D crossbar efficiently [126]. The MPI library provides a mapping interface that consists of a fully connected graph with each PE capable of sending a message to another.

B.5 Inter-process communication

Processes on the Hitachi SR2201 execute on physical processor and there is no thread library to support lightweight processes. Thus inter-process communication (IPC) involves a transfer across the network, making the IPC protocol an important consideration.

The Hitachi SR2201 implements a *remote direct memory access* (rDMA) facility for direct transfer of data between processors with delayed interrupts and more importantly without any Operating System memory copy.

To clarify the strong features of rDMA, consider the conventional distributed IPC based on message passing. These systems create OS support for complex buffer management and a detailed protocol for *send* and *receive* acknowledgement and authentication [8, 117, 94]. The system overview is shown in figure B.6.

The Operating System first copies the message into allocated buffers within the system space which is then transmitted across the network to the recipient's address space.[6] The overhead for context switching and memory copying adds considerably to the communication setup time.

In the SR2201 rDMA system, the OS reserves some buffers for user-space communication (combuf). These are directly addressable through each PE publishing its *combuf ID* and other information required for access. The DMA system uses these to completely bypass the OS during a process communication by effecting a direct transfer from the address space of the sender PE to the explicit address of the receiver PE. This is shown in figure B.7. Since the cache is write-through, the data sent in the message has also been written to the cache, while for the receiving process, when new data arrives through DMA, the cache data is immediately invalidated and may be purged.

The remote DMA system clearly reduces communication overhead and simplifies the transmission protocol. However, it does require more complex access control mechanisms for start and completion indicators.

[5] The modular network in the SGI Origin 2000 [55] is an example of a multistage network.

[6] In a conventional distributed system the transmission protocol such as TCP/IP is layered (packet, frame, physical etc.) and more copies between layers are introduced.

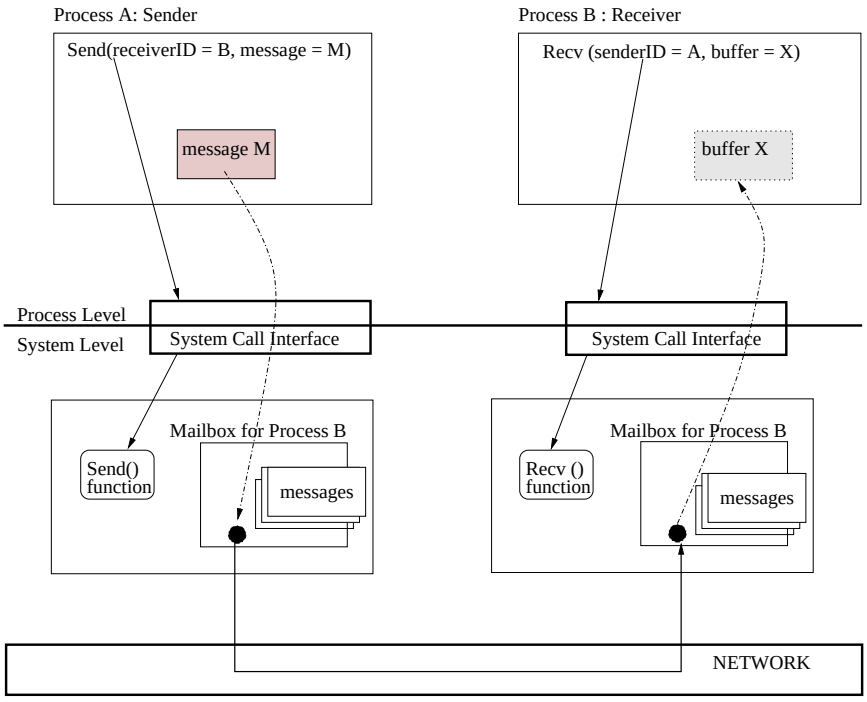


Figure B.6: Message passing with system copy

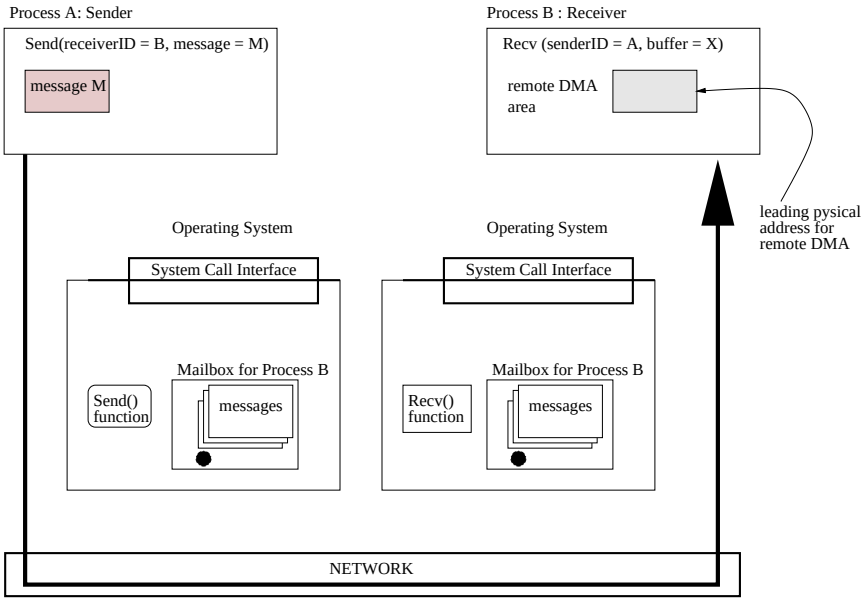


Figure B.7: Remote direct memory access (rDMA) system

Protocol	latency (usec)	bandwidth(MB/sec)
MPI	31	280
Express	26	104
PVM	77	89

Figure B.8: Comparison of some message passing systems

B.6 Memory hierarchy

The 128 registers per processor with sliding window form the first level of the memory hierarchy in the SR2201.

The next level provides 16KB primary instruction cache and 16KB primary data cache. Level 2 caches for instructions and data are 512KB each. Both caches are direct mapped and write through:

- A direct mapped cache places a block that is fetched from memory only in one place overwriting any previous data in that place:

  cache address = memory block address mod number of blocks in cache.

  Direct mapping simplifies replacement policy and speeds up translation.

- In a write through cache, on a *store* instruction, the data is written to both the block in the cache and through to lower-level memory.

  This gives simple cache coherency as the cache and memory are kept in step, however it incurs a high write penalty due to the increased memory traffic.

Each processor has block translation lookaside buffers (bTLB) resulting in up to 32MB addressing on each TLB. Each PE has 258MB of RAM locally. Thus with 256 processors, we have 64GB global memory. However, 8GB of this is on the I/O processors therefore programs have a global addressable space of 56GB. A RAID (redundant array of independent disks) system provides 350GB of local disk space, however in this work virtual memory is not used therefore disk storage is only for persistent data from completed computations.

B.7 Programming environment

The SR2201 runs Hitachi HI-UX/MPP for massively parallel processors. This is a micro-kernel based OS. The languages supported are Fortran90 and C/C++. The Message Passing Interface (MPI) is the main messaging middleware. MPI implements the Single Program Multiple Data (SPMD) programming model, although other distributions are possible and message-passing can be emulated.

The comparison in figure B.8 shows the communication latencies and bandwidth on the Cambridge SR2201 [120].

Appendix C

Network topologies

We briefly define several possible connections of processing nodes into a network. Practical architectures will have a static communication network connecting multiple processors. Ideally, each node in the network should have a direct link with another, however this requires $O(p^2)$ communication links and this quickly becomes too expensive for large p. This discussion is based on that in Kumar *et al.* [83]. The goal is to show alternative network architectures to support comparison with the Hitachi SR2201 network architecture.

We compare networks according to several criteria:

Connectivity. The connectivity of a network is the number of possible paths between any two processors. A high connectivity leads to lower contention for resources therefore is more desirable.

Diameter. The distance between any two processors within a network is the shortest number of links between them (*hops*). The diameter of a network is *maximum* distance within the network.

Cost. The cost of a network may be measured in terms of the *total* number of links required to connect p processors with the topology.

The different network topologies have better measures for different metrics and therefore there is no clear determination of the *best* network, and the selection is based on the application requirements and the cost of each. A useful feature is *embedding* in which a topology may be simulated on a different physical network. The flexibility of a network is determined by the ease of embedding other topologies efficiently and therefore meeting the demands of different applications.

C.1 Fully connected network

Figure C.1 shows a network which is a *complete graph* of 4 processors. Each node has a direct communication link with every other node in the network. This is the ideal network topology with number of hops = 1. Therefore communication latency between any two nodes is equal to the transfer time.

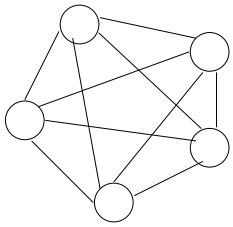
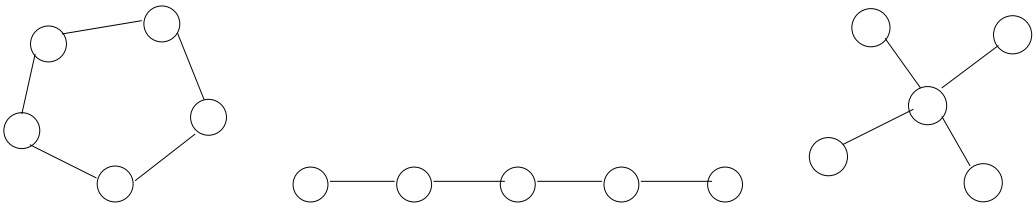


Figure C.1: A fully connected network of 5 processors



(a) Ring (b) Linear (c) Star

Figure C.2: Ring, linear and star topologies

C.2 Ring network

A ring network connects each node to two *neighbours*. If the link is one-way then minimum latency can be achieved if the source and destination nodes are adjacent. However the ring has worst case maximum communication time requiring traversal of the entire ring. On a duplex (two-way) link then a simple communication mechanism is to send a message either to the left or right depending on which yields the shortest path to the destination. In figure C.2, a ring network of 5 processors is shown.

Removing one link from the ring gives a flat linear array topology as shown in figure C.2.

C.3 Star network

A star network has a central (master) processor to which all processors are connected. Therefore communication between any non-central processors must pass through the central processor. This means communication between any two processors will require at most 2 hops, however it creates a bottleneck at the central processor if the number of messages is high.

C.4 Tree network

A tree network arranges processors into (*parent,child*) pairs such that there is only one path from parent to child. Thus there is only one path between any pair of

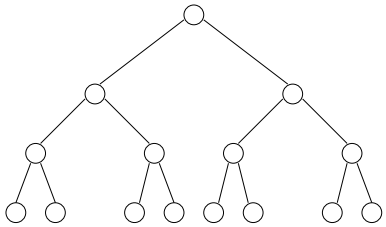


Figure C.3: A binary tree

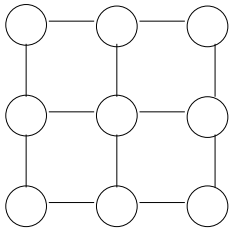


Figure C.4: A 2D mesh network

processors within the network. A balanced binary tree is shown in figure C.3 where each processor also has a forwarding switch to the child processor.

An alternative arrangement would be to have all processors at the leaves and the internal (parent) nodes act as forwarding switches. This leads to *fat trees* such as the network used for the Connection Machine CM-5.

C.5 Mesh network

A two-dimensional mesh is shown in figure C.4. It extends a linear array to a matrix with four-way connections for all internal nodes. A 2D mesh with $M = n \times m$ processors may be square ($n = m$) or rectangular ($n \neq m$).

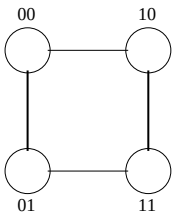
Each node in a mesh may be labelled with an integer coordinate pair (x, y). Then two nodes (x_i, y_i) and (x_j, y_j) have a channel between them if

$$|x_i - x_j| + |y_i - y_j| = 1$$

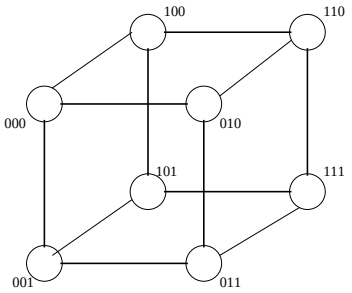
A 3D mesh connects nodes in a further dimension. Adding a wraparound channel between nodes at the ends creates a different topology called the *torus*.

C.6 Hypercube

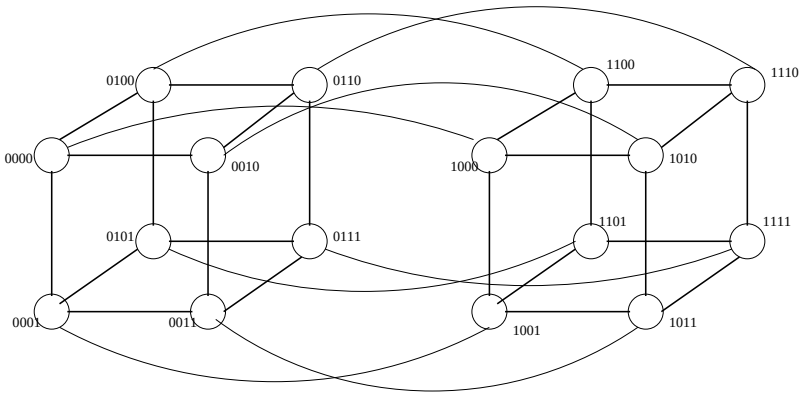
A hypercube is a mesh with only 2 nodes in each dimension. A hypercube is d-dimensional and can therefore be defined for any dimension d from zero. Hypercubes of different dimensions are shown in figure C.5.



(a) 2D Hypercube



(b) 3D Hypercube



(c) 4D Hypercube

Figure C.5: Hypercubes in 2,3,4 dimensions

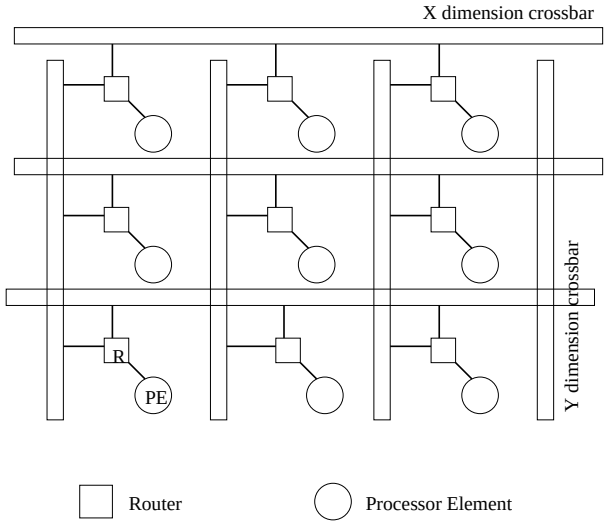


Figure C.6: A 2D crossbar network

C.7 The crossbar

Crossbar networks extend the mesh and torus topologies by creating removing the dependence on distance in any one dimension. This is achieved by creating a crossbar layer in each dimension, to which all processors that communicate in the row are connected. An *XY wormhole* routing algorithm enables any two processors connected to the same crossbar to communicate directly in one hop. If there is no *channel contention* then the crossbar effectively bounds the number of hops for any message to the number of dimensions in the network.

Crossbar networks give the widest flexibility in embedding other topologies and are referred to in the literature as *adaptable* or *reconfigurable* topologies. A two dimensional crossbar switching network is shown in figure C.6. The three dimensional crossbar used in the Hitachi SR2201 is in figure B.5.

Appendix D

Message passing interface

The Message Passing Interface (MPI) [41, 46] is a standard providing communication facilities for parallel applications. Several implementations of MPI are freely available. We have used the mpich [57, 56] implementation on a network of workstations for initial development and testing before cross-compiling for the Hitachi SR2201. This brings down development costs as time on the supercomputer is expensive.

Major vendors of massively parallel processors also support MPI as the standard communication interface. The Hitachi SR2201 provides MPI with cross compiling facilities to maintain portability when developing on a different platform. There is therefore strong motivation for the use of MPI in developing any parallel application.

D.1 Features of MPI

The MPI library is designed for high performance on large massively parallel processors. An MPI message packet has the structure shown in figure D.1. The main features of the library are the following [52, 113]:

- A rich set of *point-to-point communication* capabilities. Processes may communicate by sending messages to a named destination nodes provided that each *send* be matched by a corresponding *receive* at the destination node. Communication can be *blocking* or *non-blocking* giving different completion semantics.

- A large set of *collective operations*, where all processes in a group participate in the communication. The *broadcast* messages to all other processes or synchronize through a *barrier* call are basic collective communications, forming the basis for a larger set of communication operations.

- MPI introduces *process groups* for safe communication among a subset of executing processes. Often, only one group including all the processes is used.

- The ability to specify process topologies is useful for arranging processes in logical interconnection groups. The mapping to the underlying physical network can often be assisted by the topology specification.

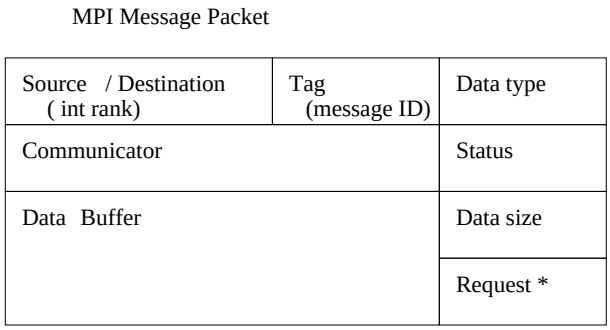


Figure D.1: A message passing interface (MPI) packet

- Derived datatypes add to the MPI internal types and can describe non-contiguous data. This enables creation of *structures* for types with different data elements.

An MPI packet is shown in figure D.1. A message specifies a destination while the receive call will specify a source. A unique tag identifies the message. If any message from a particular source is to be received, then a wild card tag may be specified. Persistent communications place a *request* for communication which may be repeatedly activated. A communicator is required for every communication to give a context for the communication, ensuring that any processes outside the communicator do not affect the message.

D.2 Communicators

A key concept within MPI is the *communicator*. The communicator binds a *communication context* to a *process group*. Every MPI message has communicator specified. This means that only those processes that are within a particular communicator may handle a message marked with that message. This forms a vital part of the MPI message safety, providing that a message sent within a particular communicator cannot be received in a different context.

The library provides a global communicator at startup, which is a constant

`MPI_COMM_WORLD`

The global communicator is the largest group formed by the number of processors indicated at initialization of MPI. The global communicator is often adequate for most applications and is used extensively in this work.

All other communicators are derived from the global communicator. Processes gain new capabilities only within their communicators.

This communicator enables communication between all active processes within the parallel environment. It creates a virtual fully connected graph so that each process can send a message to every other process.

Effective use of communicators requires that there are clear means of creating and freeing communicators, revoking capabilities and restructuring locality. The communication patterns in *Cabal* are varied and dynamic and therefore would require

several dynamic creation and freeing of communicators. It is often easier to utilize the global communicator only, and have point-to-point communication determined at each stage by a robust selection of the *target* processor.

D.3 Point-to-point communication

Point-to-point communication in MPI is enabled by *send* and *recv* functions between two processors. MPI makes two requirements for successful communication:

- The ability to send from the source must be matched by a receive call at the destination.
- The two communicating processes must be within the same communicator. If the communicator is the global communicator, then any process may communicate with any other.

The communication latency is measured as the product of the transfer time between processors with a physical channel connection, and the number of channel *hops* selected by the distributed routing algorithm. The channel characteristics are preset, therefore latency is directly dependent on the hops within the routing path. The visibility of the total latency is determined by the communication protocol selected; a communication may be *blocking* or *non-blocking*.

D.3.1 Blocking communication

A blocking communication executes a protocol that completes all changes of state required by the communication (all the buffer data is removed in a send, or all data is received for a receive call), before returning. Therefore after a blocking send, all the buffer space for the sent data may be reused. Blocking send and receives are quite slow since one of the pair of processes executing the call may have to wait for the other to rendezvous before the communication can begin.

D.3.2 Non-blocking communication

Non-blocking communication is a useful feature of MPI, allowing completion of a *send* or *recv* call before the transfer protocol actually finishes. Re-use of the buffer resources for a non-blocking communication generates an error as the data may not have been transferred. Non-blocking communications are a key feature in overlapping communication and computation to hide latency. They enable the user application to return from a send communication, proceed with other computations while transfer occurs, and check the completion of the communication at a later time.

D.3.3 Persistent communication

A very useful feature of the MPI standard is the provision for persistent communication requests. These provide a naming feature (comparable to port) for parts of

```
MPI_Recv_init(&buffer_pool[i], MESSAGE_SIZE,
 MPI_PACKED, MPI_ANY_SOURCE, CAB_ADDTERM,
MPI_COMM_WORLD, &request_pool[i])
```

Figure D.2: A persistent receive message

```
 MPI_INTEGER, MPI_REAL, MPI_BYTE, MPI_PACKED
```

Figure D.3: MPI types

the message envelope so that the information can be reused on subsequent communications. The structure of a persistent receive is shown in figure D.2.

D.4 Type matching

All MPI communications require that the sender and receiver specify a *datatype* for the message. Type matching is performed at each stage of the communication protocol to ensure that the receiver has the right buffer for a requested message. The MPI library specifies some MPI types from the list in figure D.3. These types match the types for the user language either C or Fortran. In addition a type *packed* is defined to enable different data types to be transmitted in one message. The receiver will *unpack* a message to match the data of given type.

New types can be defined in MPI with the utility of *derived datatypes* to allow messages whose data is comprised of different types. The core services of middleware such as the MPI library is in providing a communication abstraction. An important part of this is in enforcing a send/receive protocol. The MPI library performs type matching at each phase of the communication protocol to ensure that messages are received by the right PE.

D.5 Collective communications

Collective communication extends the point-to-point primitives to permit transfer of data to all processes within a specific communicator. The two requirements for point-to-point communication are modified only slightly:

- Any collective operation must be matched by a similar call on all processes that take part in the communication.
- A collective communication occurs for all processes within a communicator.

Unlike point-to-point communication, collective communication is *always blocking*. Therefore, while a matching collective operation may occur anywhere within the

code executed by the other processes, the fact that the operation is blocking places a scheduling requirement to avoid excessive waiting by any process. A notable omission is that collective communication does not include a provision for synchronizing all calling processes. Different implementations may have synchronization as a side effect, but the MPI standard merely allows each call to complete as soon as it is free to reuse all the call resources.

Several collective functions are provided [41]:

- Barrier synchronization of all processes. This is the only call with explicit synchronization semantics.
- Broadcast from one process to all. In some implementations, broadcast may have a synchronizing side effect, but this is not required.
- Gather data from all processes to one.
- Scatter data from one process to all. This is different from broadcast in that processes get different items of data, while in broadcast all processes receive the same data. Scatter may be considered the inverse of gather.
- Global reduction operations such as summation, maximum, minimum where a function takes data from all processes and broadcasts the result to all of them.

D.6 Process groups and topologies

The communicator is a convenient object for specifying a group of processes to participate in a communication. It also enables a hierarchy of process groups. Processes are identified relative to their group. Therefore several processes may have ID = 1, but being in different communicators, these are interpreted differently. In any way that this logical interpretation is created, there remains a requirement for how a process is assigned a physical processor.

D.7 Comparison of MPI and PVM

The MPI specification is designed to be the standard for all parallel applications. However, by its very generality, it is not the best available solution in all situations.

The Parallel Virtual Machine (PVM) was the first widely available established communication middleware. The PVM model provides a *virtual machine* on a heterogeneous, loosely coupled networks of workstations. The virtual machine is an abstraction that presents the different machines as a single parallel computer [50], and communication is achieved through message passing. Several systems have been built on the PVM model, and first versions of CABAL [93] were based on this model.

The virtual machine concept in PVM places stronger requirements on process management facilities, making PVM stronger than MPI in several areas [52]:

- Fault tolerance. Since PVM is based on heterogeneous networks, the requirement for surviving node failures within the system are included. For example,

when one workstation reboots, PVM notifies all the other processors that may wish to communicate with it.

- Resource manager. PVM provides a resource manager for dynamically adding or removing processors to the machine. The *spawn* function creates a named new process and may be tracked throughout its lifetime.

The richer resource management facilities give greater flexibility to update resource allocation, and may be used give interactive use.

The concept of a virtual machine is not part of MPI specification, therefore MPI provides explicit message passing facilities. The advantage of not having a virtual machine is that MPI can take advantage of any facilities in the underlying hardware to speed up communication. For example, on the SR2201, the remote DMA facility is available to the MPI implementation, leading to less portability but better performance on the particular hardware. MPI also provides *topology* with some of the functionality of a virtual machine.

Resource management was not part of the original MPI specification, but will be introduced in MPI-2 [46]. Implementations of the standard often provide static process creation at the beginning of each computation with no options to change this dynamically. The static process creation improves performance while reducing the overhead of resource management, however it is inflexible and provides limited information.

Another key design specification in MPI is that quality of service is assured, that is, fault tolerance is provided outside the standard. If any task fails, then the entire application fails through a *FINALIZE* command.

D.8 Other communication libraries

Some applications where the PVM library may be a better choice for communication have been shown. In addition, much effort has been directed at communication infrastructure specifically supporting symbolic algebra.

The distributed symbolic computation (DSC) system [36, 37] is designed for distributing symbolic computations on a network. It greatly improve the performance for symbolic computation. The FOXBOX system [38] is based on DSC and the system has been successfully used in large primality testing problems [121].

The multi protocol (MP) system [7] connects established computer algebra systems, providing communication layer for polynomial computations. The exchange of polynomial objects is facilitated by a *dictionary* of standard representations that are interpreted by the different end nodes for system specific type and representation information.

Bibliography

[1] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, 71(1):3–28, March 1990.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison Wesley, 1974.

[3] Patrick Amestoy, Philippe Berger, Michel Daydé, Iain Duff, Valérie Frayssé, Luc Giraud, and Daniel Ruiz, editors. *Euro-Par'99 Parallel Processing*, volume 1685 of *Lecture Notes in Computer Science*. Springer, 1999.

[4] Beatrice Amrhein, Oliver Gloor, and Wolfgang Kuchlin. A case study of multithreaded grobner basis completion. In *ISSAC96*, 1996.

[5] Algirdas Avizienis. Signed digit number representations for fast parallel arithmetic. *IRE Transactions on Electronic Computers*, September 1961.

[6] Olaf Bachmann and Hans Schönemann. Monomial representations for Gröbner bases computations. In *Proceedings of the 1998 International Symposium on Symbolic and Algebraic Computation*, pages 309–316. ACM Press, 1998.

[7] Olaf Bachmann, Hans Schönemann, and Simon Gray. MPP: A framework for distributed polynomial computations. In Y.N. Lakshman, editor, *Proceedings of the 1996 International Symposium on Symbolic and Algebraic Computation*, pages 103 – 112. ACM Press, 1996.

[8] Jean Bacon. *Concurrent Systems*. Addison-Wesley, second edition, 1998.

[9] David H. Bailey. Twelve ways to fool the masses when giving performance results on parallel computers. Technical Report RNR Technical Report RNR-91-020, NASA Ames Research Center, June 1991.

[10] Erwin H. Bareiss. Sylvester's identity and multistep integer-preserving gaussian elimination. *Mathematics of Computation*, 22(103), July 1968.

[11] Stuart J. Berkowitz. On computing the determinant in small parallel time using a small number of processors. *Information Processing Letters*, 18:147–150, 1984.

[12] Laurent Bernadin. Maple on massively parallel, distributed memory machine. In Markus Hitz and Erich Kaltofen, editors, *Parallel Symbolic Computation PASCO '97*, pages 217–222. ACM Press, 1997.

[13] Laurent Bernadin. On bivariate hensel lifting and its parallelization. In Oliver Gloor, editor, *Proceedings of the 1998 International Symposium on Symbolic and Algebraic Computation*, pages 96–100. ACM Press, 1998.

[14] Ann Boyle and B.F. Caviness, editors. *Future Directions for Research in Symbolic Computation*. SIAM Reports on Issues in the Mathematical Sciences. Society for Industrial and Applied Mathematics, 1990.

[15] Richard P. Brent. Some parallel algorithms for integer factorisation. In Patrick Amestoy, Philippe Berger, Michel Daydé, Iain Duff, Valérie Frayssé, Luc Giraud, and Daniel Ruiz, editors, *Euro-Par'99 Parallel Processing*, number 1685 in Lecture Notes in Computer Science. Springer, 1999.

[16] Richard P. Brent and H.T. Kung. A regular layout for parallel adders. *IEEE Transactions on Computers*, C-31(3), March 1982.

[17] Richard P. Brent and Brendan D. McKay. On determinants of random symmetric matrices over z_m. Technical Report TR-CS-88-03, Oxford University, 1998.

[18] W.S. Brown. On euclid's algorithm and the computation of polynomial greatest common divisors. *Journal of the ACM*, 18(4):478–504, October 1971.

[19] W.S. Brown, J.P. Hyde, and B.A. Tague. The ALPAK system for nonnumerical algebra on a digital computer. *Bell System Technical Journal*, 43(2):785–804, 1964.

[20] W.S. Brown and J.F. Traub. On euclid's algorithm and the theory of subresultants. *Journal of the ACM*, 18(4):505–514, October 1971.

[21] Bruno Buchberger. The parallel l-machine for symbolic computation. In Bob F. Caviness, editor, *Eurocal85: Proceedings of European Conference on Computer Algebra*, number 204 in Lecture Notes in Computer Science, pages 537–538. Springer-Verlag, 1985.

[22] Henri Casanova, MyungHo Kim, James S. Plank, and Jack J. Dongarra. Adaptive scheduling for task farming with grid middleware. In Patrick Amestoy, Philippe Berger, Michel Daydé, Iain Duff, Valérie Frayssé, Luc Giraud, and Daniel Ruiz, editors, *Euro-Par'99 Parallel Processing*, number 1685 in Lecture Notes in Computer Science, pages 31–43. Springer, 1999.

[23] B.F. Caviness. Computer Algebra: Past and future. *Journal of Symbolic Computation*, 2(3):217–236, September 1986.

[24] Giovanni Cesari. CALYPSO: A computer algebra library for parallel symbolic computation. In Markus Hitz and Erich Kaltofen, editors, *Parallel Symbolic Computation PASCO '97*, pages 204–216. ACM Press, 1997.

[25] Soumen Chakrabarti and Katherine Yelick. Implementing an irregular application on distributed memory multiprocessor. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (POPP'93)*, pages 169–178. ACM Press, May 1993.

[26] B.W. Char. Progress report on a system for general-purpose parallel symbolic algebraic computation. In S.M. Watt, editor, *Proceedings of the 1991 International Symposium on Symbolic and Algebraic Computation*, pages 96–103. ACM Press, 1991.

[27] B.W. Char, Geddes K.O., and Gonnet G.H. Gcdheu: Heuristic polynomial gcd algorithm based on integer gcd computation. In *Proceedings of EUROSAMM 84*, volume 174 of *Lecture Notes in Computer Science*, pages 285–296, Berlin, 1984. Springer-Verlag.

[28] George E. Collins. Subresultants and reduced polynomial remainder sequences. *Journal of the ACM*, 14(1):128–142, January 1967.

[29] George E. Collins. The SAC-2 computer algebra system. In Bob F. Caviness, editor, *Eurocal85: Proceedings of European Conference on Computer Algebra*, number 204 in Lecture Notes in Computer Science, pages 34–35. Springer-Verlag, 1985.

[30] Gene Cooperman. STAR/MPI: binding a parallel library to interactive symbolic algebra systems. In *Proceedings of the 1995 International Symposium on Symbolic and Algebraic Computation*, pages 126–132. ACM Press, July 1995.

[31] Don Coppersmith. Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm. *Mathematics of Computation*, 62(205):333–350, January 1994.

[32] R.M. Corless, D.J. Jeffrey, M.B. Monagan, and Pratibha. Two perturbation calculations in fluid mechanics using large expression management. *Journal of Symbolic Computation*, 23(4):427–443, April 1997.

[33] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[34] Ricardo C. Correa and Afonso Ferreira. A polynomial-time branching procedure for the multiprocessor scheduling problem. In Patrick Amestoy, Philippe Berger, Michel Daydé, Iain Duff, Valérie Frayssé, Luc Giraud, and Daniel Ruiz, editors, *Euro-Par'99 Parallel Processing*, number 1685 in Lecture Notes in Computer Science, pages 272–279. Springer, 1999.

[35] Menouer Diab. Systolic architectures for multiplication over finite fields $GF2^m$. In S. Sakata, editor, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes: 8th International Conference*, number 508 in Lecture Notes in Computer Science, pages 329–340. Springer-Verlag, 1990.

[36] A. Diaz, M. Hitz, E. Kaltofen, A. Lobo, and T. Valente. Process scheduling in DSC and the large sparse linear systems challenge. *Journal of Symbolic Computation*, 19(1-3):269–282, 1995. also in Proc. DISCO'93 Springer LNCS 722, pp.66-80.

[37] A. Diaz, E. Kaltofen, K. Schmitz, and T. Valente. DSC: A system for distributed symbolic computation. In S.M. Watt, editor, *Proceedings of 1991 International Symposium on Symbolic and Algebraic Computation*, pages 323–332. ACM Press, July 1991.

[38] Angel Diaz and Erich Kaltofen. FOXBOX: A system for manipulating symbolic objects in black box representation. In Oliver Gloor, editor, *Proceedings of 1998 International Symposium on Symbolic and Algebraic Computation*, pages 30–37. ACM Press, 1998.

[39] Peter A. Dinda, Brad M. Garcia, and Dwok-Shing Leung. The measured network traffic of compiler-parallelized programs. Technical report, Carnegie Mellon University School of Computer Science, 1998. Techincal Report CMU-CS-98-144.

[40] Andreas Dolzmann, Oliver Gloor, and Thomas Sturm. Approaches to parallel quantifier elimination. In Oliver Gloor, editor, *Proceedings of the 1998 International Symposium on Symbolic and Algebraic Computation*, pages 88–95. ACM Press, 1998.

[41] Jack J. Dongarra, Steve W. Otto, Marc Snir, and David Walker. A message passing standard for MPP and workstations. *Communications of the ACM*, 39(7):84–90, 1996.

[42] Barry S. Fagin. Fast addition of large integers. *IEEE Transactions on Computers*, 41(9):1069–1077, 1992.

[43] Thomas Fahringer and Bernhard Scholz. A unified symbolic evaluation framework for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 11(11):1105–1125, November 2000.

[44] J. Fitch. Can REDUCE be run in parallel? In *Proceedings of 1989 International Symposium on Symbolic and Algebraic Computation*, pages 155–162. ACM Press, 1989.

[45] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9), 1972.

[46] Message Passing Interface Forum. *MPI-2: Extensions to the Message Passing Interface*, July 1997.

[47] Hiroaki Fujii, Yoshiko Yasuda, Hideya Akashi, et al. Architecture and performance of the Hitachi SR2201 massively parallel processor system. In *Proceedings of the 11th International Parallel Processing Symposium IPPS'97*, 1997.

[48] Zvi Galil and Victor Pan. Parallel evaluation of the determinant and of the inverse of a matrix. *Information Processing Letters*, 30:41–45, January 1989.

[49] Thierry Gautier and Jean-Louis Roch. Fast parallel algebraic numbers computations. In Hitz and Kaltofen [63], pages 31–37.

[50] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, Mass., 1994.

[51] G.A. Geist. PVM 3: Beyond network computing. In J. Volkert, editor, *Parallel Computing*, volume 734 of *LNCS*. Springer-Verlag, October 1993.

[52] G.A. Geist, J.A. Kohl, and P.M. Papadopoulos. PVM vs MPI: A comparison of features. In *Calculateurs Paralleles*, 1996.

[53] Patrizia Gianni, Barry Trager, and Gail Zacharias. Gröbner bases and the primary decomposition of polynomial ideals. *Journal of Symbolic Computation*, 6:149–167, 1988.

[54] Alessandro Giovini, Teo Mora, Gianfranco Niesi, Lorenzo Robbiano, and Carlo Traverso. "One sugar cube, please" or selection strategies in Buchberger algorithm. In *Proceedings of ISSAC'91*, pages 49–54. ACM Press, July 1991.

[55] Andrew Grant. *Origin2000 Architecture*. Silicon Graphics Inc., 1997-99.

[56] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high performance, portable implementation of the MPI message passing interface. *Parallel Computing*, 22(6):789–828, September 1996.

[57] William D. Gropp and Ewing Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI*. Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

[58] Vipul Gupta and Eugen Schenfeld. Performance analysis of a synchronous, circuit-switched interconnection cached network. In *Proceedings of 1994 International Conference on Supercomputing*, pages 246–255. ACM Press, 1994.

[59] Robert H Halstead, Jr. Parallel symbolic computing. *IEEE Computer*, 9(8):35–43, August 1986.

[60] Babak Hamidzadeh, Lau Ying Kit, and David J. Lilja. Dynamic task scheduling using online optimization. *IEEE Transactions on Parallel and Distributed Systems*, 11(11), November 2000.

[61] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, second edition, 1996.

[62] Hitachi Corporation. *Hitachi Online Introduction to SR2201*, 2000.

[63] Markus Hitz and Erich Kaltofen, editors. *Second International Symposium on Parallel Symbolic Computation PASCO '97*. ACM PRESS, 1997.

[64] Hoon Hong, Andreas Neubacher, and Wolfgang Schreiner. The design of the SACLIB/PACLIB kernels. *Journal of Symbolic Computation*, 19(1-3):111–132, 1995.

[65] Hoon Hong, Wolfgang Schreiner, Andreas Neubacher, et al. PACLIB user manual version 1.17. Technical report, RISC-Linz Austria, 1992. Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler University, Linz, Austria.

[66] E. Horowitz and S. Sahni. On computing the exact determinant of matrices with polynomial entries. *Journal of the Association for Computing Machinery*, 22(1):38–50, January 1975.

[67] Tudor Jebelean. *Systolic Multiprecision Arithmetic*. PhD thesis, RISC-Linz, Research Institute for Symbolic Computation, 1994.

[68] Tudor Jebelean. Integer and rational arithmetic on maspar. In Jacques Calmet and Carla Limongelli, editors, *Design and Implementation of Symbolic Computation Systems*, Lecture Notes in Computer Science, pages 162–173. Springer, 1996.

[69] E. Kaltofen and A. Lobo. Distributed matrix-free solution of large sparse linear systems over finite fields. In *High Performance Computing '96*, 1997.

[70] Erich Kaltofen. Fast parallel absolute irreducibility testing. *Journal of Symbolic Computation*, 1(1):57–67, 1985.

[71] Erich Kaltofen. On computing determinants of matrices without divisions. In P.S. Wang, editor, *Proceedings of the 1992 International Symposium on Symbolic and Algebraic Computation*, pages 342–349. ACM Press, 1992.

[72] Erich Kaltofen. Challenges of symbolic computation: My favourite open problems. *Journal of Symbolic Computation*, 29(6):891–919, 2000.

[73] Erich Kaltofen and Victor Pan. Processor efficient parallel solution of linear systems over an abstract field. In *Proceedings of 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 180–191. ACM Press, JULY 1991.

[74] Ravindran Kannan, Gary Miller, and Larry Rudolph. Sublinear parallel algorithm for computing the greatest common divisor of two integers. *SIAM.J.Comp*, 16(1):7–16, February 1987.

[75] Garry Kelieff and Nilesh Raj. *Single Node Optimization Techniques on the Hitachi SR2201*.

[76] Dorothea A. Klip. New algorithms for polynomial multiplication. *SIAM Journal of Computing*, 8(3), 1979.

[77] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, third edition, 1969.

[78] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, second edition, 1969.

[79] Erwin Kreyszig. *Advanced Engineering Mathematics*. John Wiley and Sons, seventh edition edition, 1993.

[80] Wolfgang Küchlin. PARSAC-2: A parallel SAC-2 based on threads. In S. Sakata, editor, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes: 8th International Conference*, number 508 in Lecture Notes in Computer Science, pages 341–353. Springer-Verlag, 1990.

[81] Wolfgang Küchlin. The S-threads environment for parallel symbolic computation. In R.E. Zippel, editor, *Computer Algebra and Parallelism: Second International Workshop*, number 584 in Lecture Notes in Computer Science, pages 1–18. Springer-Verlag, 1990.

[82] Wolfgang Küchlin, David Lutz, and Nicholas Nevin. Integer multiplication in PARSAC-2 on stock microprocessors. In H.F. Mattson, T. Mora, and T.R.N. Rao, editors, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes: 9th International Conference*, number 539 in Lecture Notes in Computer Science, pages 206–217. Springer-Verlag, 1991.

[83] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1994.

[84] Eyal Kushilevitz and Yishay Mansour. An $\omega(d \log(n/d))$ lower bound for broadcast in radio networks. *SIAM Journal on Computing*, 27(3):702–712, 1998.

[85] Per-Ake Larson and Ajay Kajla. File organization: Implementation of a method guaranteeing retrieval in one access. *Communications of the ACM*, 27(7):670–677, 1984.

[86] John D. Lipson. *Elements of Algebra and Algebraic Computing*. Addison-Wesley Publishing Company, 1981.

[87] Jed Marti and John Fitch. The bath concurrent LISP machine. In J.A van Hulzen, editor, *EUROCAL83: European Computer Algebra Conference*, number 162 in Lecture Notes in Computer Science. Springer-Verlag, 1983.

[88] 'Mantŝika Matooane and Arthur Norman. A parallel symbolic computation environment: Structures and mechanics. In Amestoy et al. [3], pages 1492–1495.

[89] Michael T. Mcclellan. The exact solution of systems of linear equations with polynomial coefficients. *Journal of the Association for Computing Machinery*, 20(4):563–588, October 1973.

[90] Peter L. Montgomery. *An FFT Extension of the Elliptic Curve Method of Factorization*. PhD thesis, University of California Los Angeles, 1992.

[91] Masakatu Morii and Yuzo Takamatsu. Exponentiation in finite fields using dual basis multiplier. In S. Sakata, editor, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes: 8th International Conference*, number 508 in Lecture Notes in Computer Science, pages 354–366. Springer-Verlag, 1990.

[92] Winfried Neum and Herbet Melenk. Implementation of the LISP arbitrary precision arithmetic for a vector processor. In J. Della Dora and J Fitch, editors, *Computer Algebra and Parallelism*, Computational Mathematics and Applications, pages 75–89. Academic Press Limited, 1989.

[93] Arthur Norman and John Fitch. Cabal: Polynomial and power series algebra on a parallel computer. In Hitz and Kaltofen [63], pages 196–203.

[94] Gary J. Nutt. *Operating Systems: A Modern Perspective*. Addison-Wesley, second edition, 2000.

[95] Christos H. Papadimitriou and Jeffrey D. Ullman. A communication/time tradeoff. *SIAM Journal on Computing*, 16(4):639 – 646, August 1987.

[96] Roberto Pirastu and Kurt Siegl. Parallel computation and indefinite summation: A ‖ maple ‖ application for the rational case. *Journal of Symbolic Computation*, 20(5-6):603–616, 1995.

[97] Carl G. Ponder. Evaluation of 'performance enhancements' in algebraic manipulation systems. In J. Della Dora and J. Fitch, editors, *Computer Algebra and Parallelism*, Computational Mathematics and Applications, pages 51–73, 24-28 Oval Road, LONDON NW1, 1989. Academic Press Limited.

[98] Ravi Ponnusamy, Rajeev Thakur, Alok Choudhary, and Geoffrey Fox. Scheduling regular and irregular communication patterns on the CM-5. In *Proceedings of Supercomputing '92*, pages 394–402. ACM Press, November 1992.

[99] Swaminathan Ramany and Derek Eager. The interaction between virtual channel flow control and adaptive routing in wormhole networks. In *Proceedings of 1997 International Conference on Supercomputing*, pages 136–145. ACM Press, 1997.

[100] Xavier Redon and Paul Feautrier. Scheduling reductions. In *Proceedings of 1997 International Conference on Supercomputing*, pages 117–123. ACM Press, 1997.

[101] Alyson A. Reeves. A parallel implementation of buchberger's algorithm over $z_p for p \leq 31991$. *Journal of Symbolic Computation*, 26(2), 1998.

[102] G.E. Revesz. *Lambda-Calculus, Combinators, and Functional Programming*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1988.

[103] Jean-Louis Roch. An environment for parallel algebraic computation. In R.E. Zippel, editor, *Computer Algebra and Parallelism: Second International Workshop*, number 584 in Lecture Notes in Computer Science, pages 1–18. Springer-Verlag, 1990.

[104] Jean-Louis Roch and Gilles Villard. Parallel computer algebra. Tutorial Notes, July 1997.

[105] Tateaki Sasaki and Yasumasa Kanada. Parallelism in algebraic computaion and parallel algorithms for symbolic linear systems. In Paul S. Wang, editor, *Proceedings of 1981 ACM Symposium on Symbolic and Algebraic Computation*, pages 160–167, 1981.

[106] Tateaki Sasaki and Hirokazu Murao. Efficient gaussian elimination method for symbolic determinants and linear systems. In Paul S. Wang, editor, *Proceedings of 1981 ACMSymposium on Symbolic and Algebraic Computation*, pages 155–159, 1981.

[107] Steffen Seitz. Algebraic computing on a local net. In R.E. Zippel, editor, *Computer Algebra and Parallelism: Second International Workshop*, number 584 in Lecture Notes in Computer Science, pages 1–18. Springer-Verlag, 1990.

[108] John A. Sharp. A brief introduction to data flow. In John A. Sharp, editor, *Data Flow Computing: Theory and Practice*, pages 1–15. Ablex Publishing Corporation, 1992.

[109] Ernest Sibert, Harold F. Mattson, and Paul Jackson. Finite field arithmetic using the connection machine. In R.E. Zippel, editor, *Computer Algebra and Parallelism: Second International Workshop*, number 584 in Lecture Notes in Computer Science, pages 51–61. Springer-Verlag, 1990.

[110] J. Smit. New recursive minor expansion algorithms: A presentation in a comparative context. In Edward W. Ng, editor, *Proceedings of International Symposium on Symbolic and Algebraic Manipulation*, Lecture Notes in Computer Science 72, pages 74–87. Springer-Verlag, 1979.

[111] J. Smit. A cancellation free algorithm, with factoring capabilities, for the efficient solution of large sets of equations. In Paul S. Wang, editor, *Proceedings of the 1981 ACM Symposium on Symbolic and Algebraic Computation*, 1981.

[112] J. Smit. Computer algebra and VLSI: prospects for fertilization. In J.A. van Hulzen, editor, *EUROCAL'83: European Computer Algebra Conference*, number 162 in Lecture Notes in Computer Science. Springer-Verlag, 1983.

[113] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference*. The MIT Press, Cambridge, Massachusetts, 1996.

[114] Michael Sperber, Herbert Klaeren, and Peter Thiemann. Distributed partial evaluation. In Markus Hitz and Erich Kaltofen, editors, *Parallel Symbolic Computation: PASCO '97*, pages 80–87. ACM Press, 1997.

[115] Thomas Sterling and Daniel Savarese. A coming of age for beowulf-class computing. In Amestoy et al. [3], pages 78–88.

[116] Thomas A. Sudkamp. *Languages and Machines: An Introducation to the Theory of Computer Science*. Addision-Wesley, second edition, 1997.

[117] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems: Design and Implementation*. Prentice Hall, second edition, 1997.

[118] R.G. Tobey. Experience with FORMAC algorithm design. In *Communications of the ACM*, pages 589–597, 1966.

[119] Yih-jia Tsai and Philip K. McKinley. A dominating set model for broadcast in all-port wormhole-routed 2d mesh networks. In *Proceedings of 1997 International Conference on Supercomputing*, pages 126–135. ACM Press, 1997.

[120] University of Cambridge. *Cambridge High Performance Computing Facility*, 1998. http://www.hpcf.cam.ac.uk/bench.html.

[121] Thomas Valente. *A Distributed Approach to proving large numbers prime*. PhD thesis, Rensselaer Polytechnic Institute, Troy, New York, 1992.

[122] Arjan J.C. van Gemund. The importance of synchronization strucutre in parallel program optimization. In *Proceedings of the 1997 International Conference on Supercomputing*, pages 134–148. ACM Press, 1997.

[123] Stephen M. Watt. A system for parallel computer algebra programs. In Bob F. Caviness, editor, *Eurocal 85: Proceedings of European Conference on Computer Algebra*, number 204 in Lecture Notes in Computer Science, pages 537–538. Springer-Verlag, 1985.

[124] Douglas H. Wiedemann. Solving sparse linear equations over finite fields. *IEEE Transactions on Information Theory*, IT-32(1), January 1986.

[125] Chuan-Lin Wu and Tse-Yun Feng. On a class of multistage interconnection networks. *IEEE Transactions on Computers*, c-29(8):694–702, August 1980.

[126] Yoshiko Yasuda, Hiroaki Fujii, Hideya Akashi, et al. Deadlock free, fault tolerant routing in the multi dimensional crossbar network and its implementation for the Hitachi SR2201. In *IPPS 97: Proceedings of the 11th International Parallel Processing Symposium*. IEEE Computer Society, April 1997.

[127] K. Yelick, C.-P Wen, S. Chakrabarti, et al. Portable parallel irregular applications. In *Workshop on Parallel Symbolic Languages and Systems*, Beaune, France, October 1995.

[128] Eugene V. Zima. Mixed representations of polynomials oriented towards fast parallel shift. In Markus Hitz and Erich Kaltofen, editors, *Parallel Symbolic Computation: PASCO '97*, pages 150–155. ACM Press, 1997.