A Lazy, Self-Optimising Parallel Matrix Library

by

Olav Beckmann

Submitted in partial fulfilment of the requirements for the MSc Degree in Engineering of the University of London and for the Diploma of Imperial College of Science, Technology and Medicine.

September 1996
# Contents

Abstract 1

Acknowledgements 1

1 Introduction and Motivation 2
   1.1 Approach of this Project 3
   1.2 Key Issues and Work of this Project 3
   1.3 Relevance to Other Fields 4
   1.4 Structure of this Report 4

2 Background 5
   2.1 Outline of Last Year's Design 5
   2.2 Literature Review 9
      2.2.1 Automatic Parallelisation and Restructuring Compilers 9
      2.2.2 Data-Parallel Programming 10
      2.2.3 Higher-Order and Declarative Parallel Programming 11
      2.2.4 Runtime Parallelisation 13
      2.2.5 Data Distributions 14
      2.2.6 Optimisation-Related 16
      2.2.7 Matrix-Vector Primitives 17
      2.2.8 Other Relevant References 18
      2.2.9 Summary 19
   2.3 The Message-Passing Paradigm and MPI 20
      2.3.1 Some important communication concepts 20
      2.3.2 What is MPI? 21
      2.3.3 Why use MPI for this project? 21
   2.4 Technical Information about the AP1000 24

3 Mathematical Groundwork 25
   3.1 Vectorspaces, Bases, Dimensions 25
   3.2 Linear Functions, Isomorphisms 26
   3.3 Quotient Spaces and Canonical Functions 26
   3.4 Kernel and Image of a Linear Function 27
   3.5 The Homomorphism Theorem 27
   3.6 “Inverting the uninvertible” 27
   3.7 Theorem: Linear Completion 29
   3.8 Linear Functions and Matrices 29
      3.8.1 Dual Vectorspace and Linear Forms 30
| Section | Description | Page |
|---------|-------------|------|
| A       | Runtime System With Host | 92   |
| A.1     | Location of the Runtime System | 92   |
| A.1.1   | Option 1 — Host | 92   |
| A.1.2   | Option 2 — Host plus One Cell | 93   |
| A.1.3   | Option 3 — Host plus All Cells | 93   |
| A.1.4   | Evaluation | 93   |
| B       | Lazy Library Include File | 95   |
| C       | Example User Program | 99   |
|         | Bibliography  | 102  |
Abstract

This project describes a methodology for parallelising numerical programs by means of calls to a library of parallel implementations for a number of common computationally expensive operations. Such a library can provide an attractive alternative to a full-scale parallelising compiler, since users can keep both the structure of their program and the compiler they are using unchanged, and instead merely have to link with the parallel library. A naive implementation of such a library would have a number of significant problems; since we cannot, from the library implementor’s point of view, analyse the user program, we would normally for each library call have to distribute the operands and collect the result, with no scope for optimising data distributions over a sequence of operations. This project uses lazy evaluation to address these problems. Calls to library functions are initially suspended and a recipe for the result is stored instead. When a value is forced, e.g. because of output or a conditional test, we can optimise the execution of the DAG which has been built up.

Work has been done on this topic in a previous project [30]. The main work of this project has been in the development of a methodology for representing data distributions which has the following key characteristics:

- A wide range of regular data distributions, including all those available in HPF, can be represented.

- Redistributions have essentially the same format as distributions, we can calculate the required redistribution from any two distributions, and the composition of a distribution with any redistribution is always a valid new distribution.

- A method is presented for deriving the code for a contention-free implementation of distributions and redistributions from their mathematical representations.

The design of the ‘lazy’ runtime machine is described, and an algorithm for optimising data placements is given.

An experimental implementation demonstrates the practical working of the data distributions method described and shows the performance benefit of lazy data movement.
Acknowledgements

I am very grateful to my project supervisor Dr. Paul Kelly for suggesting the project, for lots of encouragement and enthusiasm, for always asking more questions, and also lots of patience and good humour.

I would like to acknowledge the use of the Fujitsu AP1000 belonging to the Fujitsu / Imperial College Parallel Computing Research Centre.

Further, I am grateful to the people from Edinburgh Parallel Computing Centre who ran a very helpful course on MPI here at Imperial College, to Hing Wing To for pointing me to some very useful literature, to my parents for their encouragement and practical support, and to my brother Roland for his help and cheering me up with Wilhelm Busch quotes.

I lift up my eyes to the hills —
where does my help come from?
My help comes from the Lord,
the Maker of heaven and earth.

Psalm 121:1–2
Chapter 1

Introduction and Motivation

Programming parallel computers is hard and writing parallel versions of existing serial programs seems even harder since serial languages hide rather than expose the inherent parallelism of a problem. At the same time, it is clear that ‘going parallel’ is the only feasible way of getting the kind of speedups that are required for allowing a number of important systems to tackle problems of realistic size. Therefore, a lot of work has been done on the problem of how parallel programs can be built, and more importantly, of how parallel versions of existing programs can be built.

Writing from scratch
There are a number of programming languages, or language extensions, that allow writing parallel programs from scratch. In some ways, it could be argued that this is the way that parallel programs should be developed, since the programmer is forced to think about parallelism form the very start. However, it is very unlikely that application programmers will do this. Rather, most developers will start with a working serial version of a program, which they try to speed up by parallelisation.

Vectorising compilers
These extract parallelism from the innermost level of a loop nest and aim to execute such a loop in a pipelined fashion on a vector processor. This is one of the earlier methods. The cost, though, is not small. For a start, the system does not only require a vector processor, but also a memory system that is capable of supplying one array element per clock cycle of the VP, both of which mean that vector machines are expensive. Then, users have to not only rely on a special, expensive machine, but also a special, expensive, and less widely used compiler.

Parallelising compilers
As an alternative to vector machines, multiprocessors have been developed that use a number of conventional RISC chips, each with a performance that is a significant fraction of that offered by a single vector processor. Compilers for such machines have to perform automatic parallelisation of serial programs. Their task is to extract parallelism from in the outermost level of a program and to coordinate the parallel execution of these components on different processors, setting up any communications that may be required. Clearly, writing such a compiler is not easy, and expecting it to come up with a really good parallel version of an algorithm for which it has received the serial code is asking a lot. From a user’s point of
view, they are again being asked to rely on a heavily specialised, and probably somewhat experimental compiler.

**Skeletons**

Another approach that has been suggested is to capture a number of common parallel programming paradigms in so-called skeletons, which are a higher-order, functional-style language, and which can be parameterised with user code in order to achieve the particular parallel algorithm the user wishes to implement. This does expose parallelism explicitly, and at the top-most level of a program. However, from a user’s point of view, a program has to be altered at its higher level, which is far from easy, because the pieces of code that remain unchanged are just those routines that happen serially, so the user ends up designing a parallel algorithm. Skeletons do provide help for this activity, but it is not clear that users will be prepared to embark on it. Further concerns are that it is not possible to make just part of a program parallel, the entire program has to become a skeletons-program, and again, there is the need to have a replacement compiler.

1.1 Approach of this Project

This project describes a methodology that aims to provide a way for users to incrementally have certain computationally expensive parts of their program execute in parallel. The basic idea is to have a library of parallel implementations for a number of common numerical problems, which users can simply plug into their program to execute parts of it in parallel. A naive implementation for such a library would, though, have a number of problems:

Since, from the library implementor’s point of view, we cannot analyse the user’s code, each operation would have to distribute the operands and collect the result, with no scope for optimisation of data distributions over a sequence of library calls. Lazy evaluation is proposed as a way round this problem. Rather than executing each library operation immediately, we return and store a recipe for the result that it defines. In that way, we build up a DAG representing some sequence of operations. If then we actually need to know a value (either because of output or a conditional test on a value), we have the full DAG available for devising an optimised execution plan. The fact that we are performing optimisations at runtime, in the ‘lazy’ runtime system for the library, means that users can use whatever compiler they want. Further, runtime optimisation may in many situations allow us to do more than at compile time, because we have all possible information available. This applies in particular to computations involving sparse data structures. In order to avoid redundant work, a “cache” of previous optimisation results can be kept.

1.2 Key Issues and Work of this Project

There are a number of interesting issues in the design of such a library.

1. **Data Distributions**

   The main thing we are trying to optimise is data distributions. Because of that, we require a good method for representing distributions, that should ideally allow us (a) to derive any redistributions that are required between two given distributions, (b) to calculate the resulting new distribution if we apply a redistribution to a given old
distribution, and (c) to derive the code for implementing the distributions from their formal representations.

2. Optimisation
The optimisation problem involved is a complex one. Work needs to be done on how we can derive good distributions without having to do an exhaustive search.

Work was done on this topic in a previous project [30] (see also Section 2.1). The work of this project has mainly been in the following areas:

- A methodology for representing data distributions that satisfies the requirements set out above. Section 4.10 gives a summary of the technique developed.
- The design of a ‘lazy runtime machine’ is described that ensures honouring of data dependencies, keeps track of data structures, stores the DAG, and provides a garbage collection mechanism.
- An optimisation algorithm is suggested.
- An experimental implementation shows the practical working of the data distributions methodology developed, in particular of the algorithms for deriving the code for implementing distributions from their representations, and of the runtime system, including the handling of the DAG, its execution, and of the garbage collection mechanism.

The implementation for this project is written in MPI and was run on the Fujitsu AP1000 belonging to the Fujitsu / Imperial College Parallel Computing Research Centre. Implementation in MPI gives portability and means that the system could also be run on workstation clusters.

1.3 Relevance to Other Fields
The topic of representing and optimising data distributions has relevance to a lot of other fields in High Performance Computing. Data locality is key to the performance of most systems, whether they are message-passing or shared-memory systems. In the case of the latter, performance relies on caches and keeping the data that are locally required on each processor in the cache for that processor. At the same time, most data distribution optimisation algorithms rely on a mathematical model for representing distributions.

1.4 Structure of this Report
This report is structured as follows. Chapter 2 gives an overview of background information, including a literature review in Section 2.2. Chapter 3 is the mathematical groundwork for the data distributions methodology developed, which itself is described in Chapter 4. Chapter 5 outlines the design of the ‘lazy’ runtime system. Then, the proposed optimisation algorithm is described in Chapter 6. Performance results are presented in Chapter 7, and Chapter 8 concludes. Finally, Appendices contain some design considerations that were not used in the current implementation but are useful for future work, and listings of both the header file for the library functions and an example user program.
Chapter 2

Background

This chapter gives an overview of background information for the project. Most of the topics discussed here are included either because they are useful for understanding the following chapters, or because they motivated design decisions that were taken later on.

The first section outlines the approach of last year’s project. This will be referred to frequently in the chapters dealing with design and data distributions, and it will be contrasted with the approach taken in this report. The next section is an extensive review of related literature. Finally, since the project will be implemented in MPI on the Fujitsu AP1000 supercomputer, there is a section that briefly outlines the message-passing approach to parallel programming and the MPI Standard, and a section that gives some relevant technical information about the Fujitsu AP1000.

2.1 Outline of Last Year’s Design

The design of last year’s project is heavily based on abstract data types, implemented as C++ classes and pretty much the full power and scope of C++ is used somewhere. Relevant for the design are especially the following points.

- Matrix operations are defined by operator-overloading, i.e. the user can write statements like $A = B \times C$;

- The ‘template’ mechanism is used to parameterise matrices with respect to their element type, so we could have matrices of $\text{ints}$, $\text{doubles}$, or even other abstract data types, such as complex numbers. Templates do not have an efficiency penalty [76, page 257].

Matrix Identifiers

The use of lazy evaluation means that matrices cannot just be referred to by their ‘name’ since this would cause problems with honouring data dependencies [64], instead, each matrix and each computation-result receives a unique identifier (an integer). This eliminates any possible problems with anti- and output-dependencies in a manner analogous to the Tomasulo algorithm [80]. However, there is a significant penalty — this approach means that data cannot be overwritten, and hence each result requires a fresh piece of memory. That in turn leads to firstly possible problems with cells running out of memory, and secondly, a need for a garbage collection mechanism.
fits(var(C,DistC),add_mm,var(A,DistA),var(B,DistB),Implementation) :-
   Implementation = add_mm_aligned,
   DistC = DistA = DistB.

fits(var(C,DistC),mul_mm,var(A,DistA),var(B,DistB),Implementation) :-
   Implementation = mul_mm_colwise,
   DistC = colwise,
   DistA = rowwise,
   DistB = colwise.

fits(var(C,DistC),mul_mm,var(A,DistA),var(B,DistB),Implementation) :-
   Implementation = mul_mm_rowwise,
   DistC = rowwise,
   DistA = rowwise,
   DistB = colwise.

Figure 2.1: Example Prolog constraints for describing data distributions in parallel operations [31].

Data Distributions
Data distributions are taken care of by a Prolog prototype that seeks to satisfy constraints of the form shown in Figure 2.1 [31]. As can be seen from that, data distributions are represented by a ‘name’, such as colwise. The Prolog system will find a ‘compatible’ set of data distributions for a DAG if one exists, it does not cater for redistributions. The report [30] states that duplication of data in different distributions is not prevented, but also not implemented. This will have to be looked at later.

Optimisations
Optimisation is the responsibility of the so-called ‘matrix handler’, which will be described in more detail below. No optimisations are actually implemented, but a number of possibilities are described.

For each operation, there are different implementations, each with different resulting distributions of the operands, and (probably) also with different costs. If no ‘compatible’ set of implementations for a sequence of operations is found, there may be the need for redistribution of data. The proposal for the overall optimisation plan is to build a DAG, where operations are nodes, and data transfer is represented by edges. Both nodes and edges carry weights, depending on the cost of either the implementation of an operation, or the data redistribution which may take place. The idea then is to somehow search this DAG for a ‘minimum path’.

Apart from this data distribution optimisation, Common Subexpression Elimination is proposed to be implemented by searching the DAG for any identical expression whenever a new result is declared and assigned an identifier. This could be quite costly — if it was possible to use hash-functions instead of a search, that would be preferable. Note that Partial Dead Code Elimination is automatically taken care of by lazy evaluation, results that are never used will not be evaluated. CSE and PDC elimination are very useful for
optimising bad programs! One would hope that even a programmer who mainly writes sequential programs would not actually calculate the same $500 \times 500$ matrix twice.

It is also suggested that algebraic transformations, such as those described in the following three equations might be used. This requires performance models for the different operations.

\[
A \cdot B + A \cdot C = A \cdot (B + C) \quad (2.1) \\
A \cdot I = A \quad (2.2) \\
A \cdot A^{-1} = I \quad (2.3)
\]

Of the three algebraic transformations mentioned, only (2.1) will be investigated further, since it could be argued that it is not immediately obvious which of the two implementations is cheaper. For the other two equations, the view will be taken that it is unlikely that a programmer would not be aware of there being redundant work here.

Software Implementation

The interface between the user’s program and the library is through a conventional include file and linking with the matrix library. The system is planned for implementation using the AP1000 native communication functions. This means that separate host and cell programs have to be written and compiled by the user (albeit the cell program is very small). The following is a description of the main features of the software design.

1. The Matrix Handler
   All matrix operations are translated to calls to one single object, the matrix handler. The matrix handler is responsible for assigning matrix identifiers and for storing the DAG. It is possible for the user to make direct calls to the matrix handler, avoiding the front-end matrix class. This approach will almost certainly not be repeated, since it results in all sorts of consistency problems.

   Another responsibility of the matrix handler is to detect a ‘force’ and then set up the resulting calculations. Status information about matrices, i.e. whether or not they are evaluated, and how they are distributed, is also stored.

   A stated problem with the matrix handler is that the operator used to access elements of a matrix returns a reference to an element, which means that it is impossible to check whether the user’s program has changed that element. A different approach here is required.

2. The Matrix Class
   The matrix class represents matrices, and allows user code to use overloaded operators on matrices. All instances of this class share one matrix handler. The matrix class does not store data, that function is performed by the matrix handler.

3. Matrix Store
   This is a separate class, responsible for managing the storage of matrix data. Data is

\footnote{That is why a design option will be considered later on that loses the benefit of automatic PDC elimination.}

\footnote{Let $I$ denote the identity matrix and $A^{-1}$ the inverse of $A$.}

\footnote{A completely separate issue is whether it might be worth checking unknown matrices for whether they are either the identity matrix or an inverse matrix before carrying out multiplications.}
stored as an associative array, indexed by the matrix identifiers. Underlying the array is a doubly-linked list.

One of the issues that will have to be looked at afresh in this year’s project is that matrix identifiers are never re-used. If a matrix is updated, and there are no outstanding references to its old value, it should be overwritten, rather than a new one generated, and the old one (hopefully) garbage-collected.

4. Operations Provided

The actual operations provided to the user are as follows: declare/delete a matrix, input/output a matrix to cin/cout, transpose, overall number of elements, number of rows, number of columns, return reference to a specific element, multiplication, addition, subtraction and “solve”.

The “Data Cache”

Finally, last year’s project uses the idea of a “data cache” that is distributed over the cells and ensures that all data required for computations is always available. The “data cache” described is indeed very cache-like, in that it is based on a local view of the data items required by or held on each processor, rather than a global view. This results in ‘fetch-style’, receiver-driven communication. As will be described in Chapter 4, the method for representing data distributions that was adopted for this project allows for sender-driven communication, which is better. The ‘cache’ tries to hold on to data for as long as possible, with a replacement policy for when a cell runs out of memory.

The data items dealt with by this cache are submatrices, identified by a tuple of the form

\[(\text{Matrix Id}, \text{Col Offset}, \text{Row Offset}, \text{Columns}, \text{Rows})\].

Naturally, with this approach, there is no conceptual problem with having submatrices that overlap.

Data are fetched into the ‘cache’ on each processor by operations pre-declaring the data they require. It is not clear how this would work in, say, Cannon’s algorithm, where data are shifted as part of the operation — the point to note here is that since cell-to-cell interrupts are not possible, data-fetching cannot happen while an operation is executing, unless cells constantly poll for messages. In order to ensure consistency, all ‘cache’-data is read-only.

The approach outlined above allows individual cells to know what the data is that they have. A key question is how cells know where the data they do not have is located. Two possibilities are outlined to solve this — (1) the host calculates this information in advance and directs cells to exchange data, and (2) the information on which data is held on each cell is replicated on all cells. That information is to be described in a data distribution mapping which should “allow for replication but [ ... ] also [ ... ] for an arbitrary distribution of data”. The nature of this mapping is not described.
2.2 Literature Review

This section reviews literature related to this project. Topics discussed have been included mainly because they are either key aspects of the general problem of "how to program a parallel computer", or because they are specifically relevant to the techniques employed in this project.

In areas where the main body of work related to an area of literature is elsewhere in this thesis, the main, detailed survey on that topic has been included as part of the Chapter concerned. In particular, Chapters 4 and 6 on data distributions and optimisation contain their own sections on related work and how it relates to the work of this project. In those cases, this section only contains a shorter collection of pointers to the relevant sources.

The basic structure of the discussions that follow is in each case a brief outline of the work that has been done and then some points about how that relates to the topic of this project.

2.2.1 Automatic Parallelisation and Restructuring Compilers

Probably one of the largest bodies of work in the area of parallel computing is on the topic of automatic parallelisation. Because of that, it isn't really possible to give a true survey of that field here. Instead, three example papers will be discussed briefly that give some idea of the type of problems that are typically dealt with in automatic parallelisation. These papers all come from the area of restructuring compilers, that is, compilers that do not just try to extract maximum parallelism from a given program, but that also restructure a program into a semantically equivalent one that has been optimised with respect to some criterion. There are three very common objectives that restructuring compilers might aim for, as follows.

- **Parallelism in innermost loop level**
  Vector processors can be used to speed up the execution of the innermost level of a loop nest by pipelining array computations.

- **Parallelism in outermost level**
  On multiprocessor architectures, a loop that contains parallelism in its outermost levels can be parallelised by assigned different iterations of the outer loops to different processors.

- **Data Locality**
  It is often crucial to the performance of a computation that the number of data accesses in the innermost level of a loop nest is minimised. This is due to the different access time parameters for different levels of memory hierarchy and is known as the data locality optimisation problem.

Banerjee [6] shows how double loop nests can be transformed for extracting parallelism. All transformations described are represented as unimodular matrices acting on the loop indices. Results are given for such unimodular transformations in general, so that operations like reversal, interchange and skewing can be handled as special instances. The paper gives theorems on the validity of transformations, computation of new loop limits and the existence of matrices that allow parallel execution. The idea for the affine data transformations that are described in Section 4.5 originally came from trying to apply the theory of this paper to data structures rather than iteration spaces. When dealing with data,
there are, however, additional operations that wouldn't make sense for loop nests, such as translations.

Wolf and Lam [83] extend the work of Banerjee by dealing with general loop nests, rather than just double nests. The main difference here is that data dependencies cannot just be represented by integer distance vectors (these can represent data dependencies in loop nests that satisfy the property that an \( n \)-deep nest has at least \( n-1 \) levels of parallelism), but are now represented by the more general direction vectors (for loop nests where two or more loops must execute sequentially). The algorithm presented by Wolf and Lam can maximise for both coarse-grain parallelism (for multiprocessors) and fine-grain parallelism (for vector processors), and even works for machines that can exploit both types of parallelism.

Finally, considerably more complicated, but probably the most relevant to this project is the paper by Wolf and Lam on optimising data locality [84]. This deals with single loop nests and combines unimodular operations with loop tiling in deriving an algorithm that is exponential in the loop nest depth. Models for different types of data re-use are given and used in the algorithm. The idea that links this paper to the current project is that when we have accumulated a DAG of matrix operations and are looking for an optimal execution plan, what we are trying to do is minimise data movement, in other words maximise locality. However, the big difference is that a sequence of operations will not have the form of one perfect loop nest (i.e. with all computation in the innermost loop). Also, by dealing with library calls to what are already fully parallel algorithms, we have reduced the complexity of the problem considerably, because extracting parallelism as such is no longer a task. In that situation, it would appear a bad idea to start introducing more complexity by again looking at loop nests.

Summary

- There are quite a lot of similarities in some of the optimisation problems dealt with in automatic parallelisation and the task of optimising data placement and algorithm implementation of this project. One such example is data locality maximisation, albeit that automatic parallelisation does this on a different level (e.g. cache).

- The prevailing difference between automatic parallelisation and the problems addressed in this project is that there is usually an additional level of complexity in the former — that of computation placement and the issue of extracting parallelism in the first place.

2.2.2 Data-Parallel Programming

As outlined in the last section, one approach to achieving portable parallelism are parallelism extraction systems that start with a serial program and produce a parallel one. Other ideas are based on virtual machine emulation [13]. In this section, we will briefly look at a third approach that aims to avoid some of the drawbacks of both the previous ones. The view has been expressed that serial programming languages really hide parallelism [12], and are therefore not well suited for expressing parallel problems. On the other hand, full-scale parallel languages land the user with having to organise processes, load-balancing, communication synchronisation, etc., which is very difficult and unlikely to find widespread use. The idea behind data-parallel programming is to provide a style of programming where parallelism is explicitly expressed in the form of operations that are applied to entire arrays or other large data structures.
Data-parallel languages seem to be naturally best suited to implementation on SIMD parallel machines. Chatterjee [13] discusses their implementation on MIMD computers. The paper describes an optimising compiler for a (nested) data-parallel language targeted onto a shared memory multiprocessor. Some of the problems that would be encountered by a naive ‘port’ of a data-parallel program from SIMD to MIMD are that the parallelism is often too fine-grained (this affects performance for large problems), the serial overhead of load-balancing (affecting the small-problem performance), and the cost of emulating the lock-step-style synchronisation that happens automatically with SIMD in MIMD. Chatterjee shows how programs can be transformed in order to overcome these problems by increasing the granularity, relaxing the implicit lock-step synchronisation and other transformations to do with data storage and locality.

Bozkus et al. [12] describe the implementation of a compiler for Fortran 90D, a data-parallel extension to Fortran 90. Since the design of the HPF (High Performance Fortran) language was based on Fortran D, the techniques employed in writing this compiler are claimed to be a suitable framework for HPF compiler design. The output of the compiler described is an SPMD program for distributed memory MIMD computers.

Finally, data-parallel languages in their raw form are very well suited to dense matrix and other linear algebra algorithms, but they are not a natural way of expressing sparse algorithms. Nested data-parallel languages [8] aim to avoid this problem by providing a combination of data-parallelism and control parallelism. This is best shown by example. A sparse matrix can be represented as a sparse sequence of rows, each of which is itself a sparse sequence of elements. If a parallel function is then applied to the entire structure, an efficient implementation can be derived by a technique referred to as flattening nested parallelism [8]. The compiler described in this paper works on a nested data-parallel language called Nesl.

Compilation of another data-parallel language, Fortran 8x, for the Connection Machine is described by Albert et al. in [3]. Finally, APL [28, 1] is a programming language that provides constructs for manipulating large aggregates of data in the form of arrays. The concept of nested data-parallelism is mirrored in the notion of nested arrays in APL.

2.2.3 Higher-Order and Declarative Parallel Programming

The aim of achieving portability for parallel programs has motivated what is variously called either the Skeletons, Structured or Higher-Order approach to parallel programming. There are large number of references on this topic.

Darlington et al. [19] describe the problems of parallel programming as being with performance prediction and portability, two issues that are a lot easier to address in the sequential von-Neumann model. Two possible approaches are outlined to try and address these problems. The first is the design of a general parallel machine model or an abstract parallel machine, such as the PRAM. The other is the development of a new methodology for programming parallel machines.

The central idea with the latter is to capture so-called skeletons a set of common patterns for parallel algorithms, which can be parameterised with user code. Performance models are supplied for each skeleton for a range of different parallel architectures which allow the user to make decisions in an interactive way about program design. Lastly, program transformations are studied. These can be used for either translations of higher-level specifications onto a lower level, or for transformations of programs into semantically equivalent ones that may execute more efficiently on other architectures. In [20], the skeletons approach is studied with the example of raytracing. In practical applications, many skele-
tons are combined. The issue of how to optimise such combinations is the topic of [79]. The skeleton language in [19, 20, 79] is SCL.

A different structured parallel programming methodology is $P^3L$ [17, 18]. Writing on the same topic, [65] suggests that a parallel programming methodology should have three main features — a language, a support system for that language, and a strategy for the programmer to choose ‘good’ algorithms. It is interesting to see what these issues correspond to in the parallel libraries approach of this project. The choice of language is easy: the sequential language which the user program is written in. More complicated is the support system and the strategy for choosing good implementations for algorithms. These are here taken over by the runtime system, which will obviously have to take some fairly complex decisions. However, the lazy evaluation strategy means that maximum information is available, something that is not the case for the developers of a general (compile-time) methodology. For example, [65] states that most problems of optimising the combination of parallel algorithm implementation and exploitation of machine features are intractable unless restrictions are made, e.g. on the topology of the architecture. In the lazy libraries approach, we know the architecture!

Using purely functional language, the idea in [33] is to identify a small set of higher-order functions that can be implemented efficiently in parallel on some target architecture, and parallel programming then happens by expressing algorithms in these higher-order constructs which are transformed into an implementation.

A slightly different angle on programming with coordination structures is given in [57]. Here, parallel programs are classified by their communication pattern into synchronous and asynchronous ones. Synchronous programs are ones whose communication pattern has both deterministic connectivity and uniqueness, the latter meaning that any given pair of subcomputations communicates only once and in one direction. Such programs can be represented by so-called execution graphs, that is, data-flow graphs where each node is a stateless subcomputation. Asynchronous programs, on the other hand, are represented by message graphs, which differ from execution graphs in that nodes have a local state, and that more than one data item may travel along edges. The main message of [57] is that the best methodologies for writing synchronous programs are irreconcilably different from those for writing asynchronous programs. This paper also states the view that the task of a parallel programming environment should be the coordination between sequential subcomputations. The problem with that is that users are being asked to rewrite the top-most level of a program, something which is far from easy and that many users are also reluctant to do because it means that they have to change the compiler they are using and cease to have control over the basic structure of their program.

Hill [36] addresses “lazy data-parallel functional programming”, as follows. The basic idea of data-parallel programming is a map of some function over a large collection of data, executed synchronously. In non-strict functional programming, lazy evaluation means that a function is only executed on those elements of a list where the result is forced in a subsequent computation. This causes the loss of the parallelism which existed in the data-parallel model. [36] proposes a method for reconciling these differing views of map.

A cost calculus for functional parallel programming that focusses on the issue of how to handle compositions is described by Skillicorn and Cai in [74]. Finally, [14] is a book on skeletons.
How does this relate to lazy libraries?

Some of the problems that need to be solved for implementing skeletons are quite similar to the problems that need to be addressed here. For example, when a force happens in the lazy libraries approach, what has to be done is an optimisation of the sequence of parallel operations that has been called from the user program. In some ways, the lazy library approach could be in a better position to solve such problems, since a lot more information is available. So, a library that has been implemented for some architecture only has to consider that particular architecture model in any optimisations. More generally, lazy evaluation means that we know what operations the user has actually called, and with what data, so we have in a way the maximum possible information available.

2.2.4 Runtime Parallelisation

There are situations where a parallelising compiler cannot construct a valid parallel schedule at compile-time. This arises when the inter-iteration dependency pattern of a loop cannot be determined statically. An example are operations on sparse matrices where efficiency dictates that only non-zero elements are used in computations, but where the location of those non-zero elements within the matrices is only known at runtime. In these cases, runtime parallelisation is the only alternative. Joel Saltz et al. have done a lot of work on this topic [70, 71, 69]. Building on that is work by Leung and Zahorjan [52, 53], which is what most of this section is based on. These papers also contains pointers to further literature. The basic idea of the Saltz et al. approach to runtime parallelisation is that the compiler produces two pieces of code for each source loop. The first is called the inspector; it calculates the inter-iteration dependencies and then constructs a parallel schedule for the iterations of the loop. The second piece of code is called the executor. It uses the schedule to achieve a parallel execution of the loop. The parallel schedule is based on partitioning the set of iterations of the loop into wavefronts — subsets that contain no dependencies and can hence be executed concurrently. The number of wavefronts in a schedule is called its depth. As a broad rule, schedules with minimal depth are most likely to achieve the shortest parallel runtime.

The straightforward way to write an inspector is to run through the loop and calculate the values of the (statically unknown) index functions of the arrays involved for each iteration of the loop, and calculate the data-dependencies from that. The problem with this approach is that the inspector then has the same order of magnitude complexity as the source loop. This has led to work on an alternative runtime parallelisation approach, known as ‘doacross parallelisation’ [70]. Iterations are assigned to processors in a wraparound manner and each instruction ‘waits’ until all operands necessary for its execution have been produced. Doacross parallelisation does not require an inspector step, but its practical performance is poor, due to a very bad, ‘random’ schedule. However, since there are no data-dependencies as such in the inspector, this can be parallelised usefully by ‘Doacross’ [71].

Leung and Zahorjan [52] describe two alternative ways of parallelising the inspector. First, sectioning works by dividing the entire range of iterations into subranges or sections of appropriate size. The sections are then concurrently inspected and separate ‘mini-schedules’ derived. The overall schedule then results from concatenating the ‘mini-schedules’. This approach results in maximum parallel speedup (i.e. $T_{\text{parallel}} \approx T_{\text{serial}} / P$) for the inspector, but gives a parallel schedule with suboptimal depth. Second, bootstrapping works by first finding a valid schedule for the inspector using sectioning, but then running the (global)
serial inspector loop in a wavefront-by-wavefront (i.e. parallel) manner. This results in
the same optimal minimum-depth schedule that the serial inspector would have produced,
but the inspector itself may run in suboptimal time. Experimental results show that for
loops where the number of iterations greatly exceeds the number of available processors,
sectioning will produce better results more often than not.

How does this relate to lazy libraries?

The similarity here is that since we cannot analyse the library-user’s source code, no depen-
dence information is available statically, so this information is obtained by lazy evaluation —
the user’s code runs while the runtime system builds the DAG, i.e. collects the data-
dependence information. This step is exactly analogous to a serial inspector loop — in a
way, we could think of the inspector-loop as a compiled-in ‘lazy’ run through the loop. Next,
the lazy library’s runtime system devises an execution plan, and effects the calculation of
the result. Again, this is remarkably similar to the ‘executor’ concept. However, there are
a few significant differences:

- The approach discussed above deals with automatic parallelisation, i.e. the items be-
tween which dependencies etc. have to be calculated are instances of instructions, and
in a sizeable algorithm, the number of such instructions is at least $O(N)$. Hence the
associated problems with having to parallelise the executor. The items in the DAG of
this project are parallel algorithms, and the number of nodes in the DAG can probably
be bounded above by a small-ish constant independently of $N$.

- In this project, we are not constructing schedules, but optimised data distributions.
However, the two problems are quite related, e.g. the general scheduling problem is
NP-hard, and by considering data-dependencies and data re-use without redistribution
as the analogous concepts in the two problems, it can probably be shown that finding
optimal data layouts in the general case is also NP-hard.

It is interesting to see how the same problem (i.e. lack of compile-time information) has led
to very similar solutions.

2.2.5 Data Distributions

There is an extensive review of literature on data distributions at the end of Chapter 4,
which also looks at how the approach of this project relates to the published work reviewed.
For that reason, this section here will only briefly point to some relevant literature on data
distributions.

The main point of interest for this project is how data distributions are represented in
the literature, what type of distributions are used, and how redistributions are handled.
Lawrie [51] describes the design of a memory system for an array processor, i.e. a machine
with $N$ ALUs that process vectors of up to $N$ elements in one step. One key issue discussed
is how a memory system for such a processor can store data (e.g. a matrix) in such a
way that the vectors of data on which the array processor should operate can be accessed
concurrently. Various types of skewing of data dimensions over memory units are described
and it is shown how these can be used to extract parallelism in different data dimensions.
Li and Chen [56] discuss data alignment. The point here is to find a set of functions
that will provide a mapping from one uniform index domain to the various access patterns
of arrays involved in a computation. Data partitioning then happens over the single index
domain. Li and Chen show that the problem of finding a set of alignment functions, in the very general form they discuss, is NP-complete, but they give a heuristic algorithm.

A lot of parallel programming languages, or input languages to parallelising compilers, such as HPF, leave the task of determining data partitionings to the user. Gupta and Banerjee [32] discuss automatic partitioning. Their approach works by determining parallelisation and communication constraints on data distributions for data structures. A whole program is considered, not just a single loop nest, and the various constraints are combined to obtain an overall distribution scheme. As described in [32], the scheme does not provide for re-organisation of data if there are conflicting constraints, rather, constraints are given weights and conflicts are resolved in favour of the ‘heaviest’.

Anderson and Lam [5] discuss global optimisations for parallelism and locality in a program. This involves both data- and computation-decomposition, which means that the problem is considerably more complex (in fact it is shown to be NP-hard) than finding optimal data distributions as we are trying to do in this project. Decompositions are represented by a not necessarily affine function, mapping (in the case of data) array indices onto processors. Blocked distributions are represented as vector spaces, or as the span of a canonical basis for the number of loop dimensions. This allows calculating all the iterations that fall into one tile, but it is not a mapping. A related paper is [4] by Amarasinghe and Lam. This starts with a description of how a computation is to be partitioned and then gives an algorithm for deriving an SPMD to implement that computation. In order to improve data-flow efficiency, a value-centric analysis, rather than a location-centric analysis is used. It is shown that conventional data-dependence analyses are over-cautious.

Kaushik et al. [44] describe a method for communication-efficient array redistribution. This is applicable to redistributions from one type of blocked-cyclic distribution to another, and the basic idea is to trade off sending fewer messages in total for sending larger messages. Since communication startup times are usually about two orders of magnitude higher than the per-byte message transfer time, this makes good sense up to a certain message size. From this project’s point of view, the interesting thing in this paper is their method for representing block-cyclic data distributions as tensor basis factorisations (see Section 3.13). Thakur et al. [77] also discuss array redistributions, giving efficient algorithms for implementing redistributions between different HPF distributions. This will be referred to in Section 4.8.

Feautrier [23] proposes a method for automatic distribution. This is based on a data-flow analysis of a loop nest which is to be parallelised. A placement function for the computation is derived and optimised, which then determines data placement by the owner-computes rule. This paper mainly relates to Chapter 6, however, it suggests some useful assumptions that will be used in demarking the scope of the work in Chapter 4, in particular, to leave ‘changing the geometry’ as a separate issue.

Tatsuya Shindo et al. [72] describe Twisted Data Layout as a technique for extracting parallelism from both dimensions of a 2-dimensional array that is distributed over a ring. It will be shown in Chapter 4 how the data distribution method described in this project can be used to represent twisted layout.

Finally, Megson [60, 61] has published work on the mapping of so-called regular algorithms, and has also done work on non-linear scheduling techniques for such algorithms.
2.2.6 Optimisation-Related

Probably the most difficult and widest area of literature that needs to be considered is that of optimisations. Feautrier’s work on ‘automatic distribution’ [23] will be used in Chapter 6, and is reviewed in depth there. The remainder of this Section will therefore just give a quick survey of other relevant publications in this area.

Nicol, Saltz and Townsend [63] discuss the scheduling of delay points in irregular parallel computations. Delay points are points in a computation where some restructuring or remapping has to take place, either semantically, or for improving subsequent performance. Data redistributions would be one such example. They show that there is often a choice for the scheduling of such delay points, and describe a polynomial-time algorithm that optimises their placement in an execution plan with respect to overall execution time. Though not with simple examples, this could in future be very relevant for the lazy libraries approach, not only with respect to scheduling redistributions, but also for the problem of how much time to spend optimising an execution plan and when.

Li and Chen [55] describe a method for compiling shared-memory parallel programs (where the issue of data distribution is implicitly left to the hardware) into communication-efficient programs for massively parallel distributed memory machines. They describe their method as being a software-approach to bridging the gap between a shared-address-space programming paradigm and a physically distributed memory. The paper focuses on efficient communication more than optimising data layout as such.

Bokhari [10] discusses the problem of optimal assignments of tasks in a computation to the processors of an inhomogeneous architecture. This is relevant in that total execution time, rather than data movement or communication is what is minimised. This is also what we want to do in this project, and some ideas from [10] about how to combine different cost parameters to form a model for total execution time might be very useful. However, it also seems that dealing with inhomogeneous architectures introduces a level of complexity which, for the moment at least, should not be part of this project.

In a related paper by Bokhari [11], a number of partitioning problems are solved for the more specific case of a single-host multiple-satellite architecture. The type of problems solved are all to do with computation-partitioning, rather than data-partitioning and distribution, so they have some considerably different characteristics to what is addressed in this project. However, since the assumption that we are dealing with a uniform architecture (see Chapter 4) should, at least for the AP1000, at some point be changed to include the host, the model of this paper might prove relevant.

In Section 2.2.2, the data-parallel programming language Fortran D was mentioned. A paper that discusses optimisations for Fortran D compilers by Hiranandani et al. is [37]. The optimisations discussed are evaluated for stencil computations, something that has not been dealt with so far in this project, but should be addressed at a future date. The idea of overlapping communication and computation can easily be done in MPI by means of non-blocking communication, and will be used for this project already. Another point discussed in [37] is the effectiveness of communication optimisations as a function of the ratio of computation to communication in a program. This looks like being related to the question of how much time should be spent optimising data distributions for this project.

Automatic array layouts in SIMD machines, such as the Connection Machine, are discussed by Knobe et al. in [48]. Optimisations are made primarily for minimising the cost of communication when arrays are moved between processors, but the reduction of total memory usage is also discussed. They point out that changing data layout may allow to
use different communication operations, at a lower cost to the communication demanded by a previous layout.

One of the optimisations that will be discussed in Chapter 6 is the redistribution of product-sums. A paper that presents an algorithm to determine the equivalence of expressions in so-called random polynomial time is [29]. This uses a random number generator to test expressions. A 'not equivalent' result is always correct, while the probability for an 'equivalent' result being correct can be determined by the user via parameter-setting.

Finally, a somewhat speculative idea is whether the very efficient constraint-solving algorithm invented by Henglein [34] for the purpose of binding-time analysis in partial evaluation could somehow be adapted for solving data distribution constraints. This should certainly be looked at, since that algorithm reduces the complexity of a previously \(O(N^3)\) problem to almost linear time.

### 2.2.7 Matrix-Vector Primitives

The idea behind linear algebra *primitives* [26, 2] is to provide ready implementations of certain linear algebra routines of varying degrees of complexity that can then be combined to build algorithms or programs of a higher complexity. This approach has a lot of similarities with the 'parallel library' being developed in this project. Gallivan et al. [26] point out that there are several advantages in investigating linear algebra algorithms in terms of primitives:

1. Investigating the performance of lower-level primitives will often give a very good estimation of the likely performance of algorithms that are made up by combining them.
2. Primitives can help to identify directions for future work in developing parallel languages and restructuring compilers.
3. A good understanding of the mapping of primitives onto parallel architectures is an essential framework for designing more complex parallel algorithms.
4. If certain parallel architectures are shown to have weaknesses in executing primitives, that can serve either as a useful pointer to their further development, or as an indication for the type of problem they are best suited to solve.

One very common collection of primitives is BLAS (Basic Linear Algebra Subroutines). This is described in [26]. There are three levels of BLAS, depending on the complexity of the problems they solve:

1. **BLAS 1**
   These are \(O(N)\) routines that operate on \(O(N)\) data. Examples are vector-vector dot products and SAXPY (Single\(^4\) Precision ax Plus y) loops.

2. **BLAS 2**
   BLAS 2 are routines of complexity \(O(N^2)\) working on \(O(N^2)\) data, such as vector-matrix products, matrix updates etc.

3. **BLAS 3**
   Third-level BLAS primitives perform \(O(N^3)\) algorithms on \(O(N^2)\) data, with parallelism in an additional dimension to BLAS 2. They will normally consist of several independent BLAS 2 primitives. The key example is matrix multiplication.

---
\(^4\)Or Double, presumably.
Slightly different primitives are discussed by Agrawal et al. [2]. Their “vector-matrix primitives” are *extracting* a vector from a row or a column of a matrix, *inserting* a vector into a matrix, *distributing* a vector across the rows or columns of a matrix, and *reducing* the rows or columns of a matrix over some binary operator to form a vector.

The key difference between the approach in this project and the work outlined above is that there is not such a concept of different levels of routines in this project. It seems that this should be investigated for further work, especially because a lot of sparse linear algebra algorithms seem to decompose into routines of different levels. If this was done, it would seem crucial to have a way of representing the type of primitives discussed by Agrawal et al. in terms of what they do to data distributions. This will be addressed again in the conclusion of this report.

### 2.2.8 Other Relevant References

This section finally gives pointers to other useful references on high performance computing that are relevant to this project. Firstly, books. [35] is an excellent book on computer architecture and includes material on vector processors and the Tomasulo algorithm, referred to in the introduction and the design of the ‘lazy’ system respectively. [49] contains a large number of parallel algorithms. Performance and efficiency issues are discussed extensively. This was the source for the performance models for the parallel algorithms used in this project. Some of the mathematical proofs require a ‘handle with care’ warning. [54] discusses a wide range of different parallel computing models, such as shared- versus distributed memory programming, object-oriented, data-parallel and functional parallel programming. It is also a good source on issues like the complexity of scheduling problems and on how to build DAGs. Finally, [24] deals with declarative parallel programming.

In a lot of publications, the Connection Machine is referred to (see also Chapter 4). [43] is a paper on the CM-5. [50] discusses the blocking of algorithms for improving their use of caches. Since it is vital that high-performance parallel programs at least run efficiently in their serial sections, this should be referred to for the implementation of the local matrix-multiples etc. in parallel algorithms. Data dependencies, and a wide range of loop transformations, such as stripmining, merging, etc. are described in very readable form by Padua and Wolfe in [64].

This project has so far not dealt with stencil-style computations. When that is done, the data distribution approach of Chapter 4 will have to be extended. Reed et al. [66] address the problem of how the data partition of a stencil-style computation should be handled.

Finally, [45] is a review paper on parallel machine models and parallel algorithms that are relevant to combinatorial optimisation problems. Particular emphasis is placed on the classification of parallel computations according to their complexity.

The problem of mapping parallelism onto a given architecture, consisting of task scheduling and code generation for the separate tasks, is discussed by Gerasoulis et al. [27]. Although, they are dealing with *computation*—rather than data-based problems, it seems that their model of a task graph (a DAG) could be used in designing the DAG that is required for this project. Similarly, Mace [59], in a discussion of memory storage patterns for parallel processing, introduces program graphs with properties very much like the DAGs of this project. The problem discussed is the selection of ‘shapes’, or memory storage patterns, to achieve optimal execution time for such a graph. Several versions of the problem are shown to be NP-complete, but for dynamic programming solutions are described for more restricted variants of the problem. These two publications will be referred to again in
Chapter 5.

2.2.9 Summary

The key issues from High Performance Computing literature that relate to this project are these:

- Data locality — this is crucial to performance in most systems.

- Optimisation algorithms — whether their subject is scheduling or data distributions, a lot of the algorithms described deal with problems of essentially very similar structure and tractability.

- A key issue in compile-time systems is always whether data-dependencies can be fully detected. The runtime approach of this project means that we have all data-flow information available.

- A key reason for the complexity of many of the problems that were described in this section is the existence of the twin objectives of maximising both parallelism and data locality. In this project, library functions are already parallel algorithms, and this significantly reduces the complexity of the problem to be addressed.
main(...)
{
    if(process is controller)
        Controller(arguments);
    else if(process is worker number i)
        Worker(i, arguments);
}

Figure 2.2: Emulating general message-passing with SPMD, C syntax.

2.3 The Message-Passing Paradigm and MPI

The basic concept behind the message-passing paradigm for parallel programming is that programs are written for a parallel platform consisting of several processors, each with its own memory, and each of these processors executing its own task [24]. The processors have no direct access to each other’s memory, and therefore all communication happens by exchange of messages between the processors. This concept has a number of parallels with object-oriented programming, which means that C++ is in many ways a natural language for implementing message-passing programs.

Message-passing programs can be run on distributed- (or even shared-) memory multiprocessors, clusters of workstations, or uniprocessor systems, but it is clear that the multiprocessor and MIMD control models are best suited to the message-passing programming model. However, a lot of systems do not actually allow tasks to be created or destroyed during execution, or multiple tasks to be run on processors (both of which are possible under the multiprocessor control model), but instead create a fixed number of identical tasks on program startup. This restriction can be justified by the observation that for most practical problems, different tasks follow the same pattern anyway. The resulting model is referred to as SPMD, or Single Program Multiple Data. The MPI standard is written for the SPMD model, that is, when using MPI, we write one program. It is possible, though, to emulate the general message-passing model in SPMD, as shown in Figure 2.2 [58]. This obviously results in large executables if there are many different tasks, but that is no longer a big problem on most current platforms.

It is very useful, especially when programming with MPI, to think of a message-passing system as being broadly analogous to more conventional communication systems, e.g. the phone network.

2.3.1 Some important communication concepts

- **Point to Point and Collective Communication.**
  
  Point to point communication simply means that a message is sent from one sending processor (the source) to one receiving processor (the destination), and only these two processors are involved in the communication. Collective communication involves a larger number of processors. Examples are barrier synchronisation, broadcast, reduction and scan. The meaning of the first two should be clear. A reduce uses a binary
operator to recursively calculate one value from a large, distributed set of values. A
scan is similar, except that partial results are calculated and preserved.

- **Synchronous and Asynchronous Communication.**
  Synchronous communication means that the sender receives some acknowledgement
  from the receiver once a message has been received. In asynchronous communication,
  the sender only knows that the message has been sent.

- **Blocking and Non-Blocking Communication.**
  Blocking communication means that a process that participates in an exchange of
  messages (whether as sender or receiver) will block until the communication has been
  completed. In non-blocking communication, a process can proceed to do other work
  while the communication is still going on. This is useful for latency-hiding, but care
  must be taken not to alter any send-data before the send has been completed, and
  similarly, not to use receive-data before the receive has been completed. For that
  purpose, there are usually routines that check for the completion of a communication.

### 2.3.2 What is MPI?

MPI (*Message Passing Interface*) is the first standard for a message passing interface across
the whole parallel processing community. It was developed with backing from the US and
Europe, and from vendors and users. MPI is defined in a document known as the MPI
Standard [62]. In practice, MPI takes the form of a library of C or Fortran routines.

MPI provides *source-code portability* across different parallel platforms for message-passing programs written in C or Fortran. This also extends to heterogeneous architectures. Source-code portability is particularly useful when a program is to be developed and tested on a system other than the one where it is eventually used. Another important feature is that the use of MPI routines allows for efficient underlying implementations that are adapted to the particular architecture in use. MPI is huge, but a large number of things can be done with a reasonably small core. The most important tasks that are outside the scope of MPI are the loading of processes onto processors, the spawning of processes during execution (i.e. SPMD is assumed), and parallel I/O.

*Every* MPI program takes the format shown in Figure 2.3. This has some important
implications for the design of the library in this project. Any calls to MPI will obviously be
hidden in the library, but, by using the library functions, that in turn call MPI functions,
the user’s program does in fact become an MPI program. This means that it is essentially
an SPMD program, i.e. there will have to be some routines that allow the user to mark out
statements that are to be executed *only* by the controller process (e.g. standard I/O on the
host). I will return to this in Chapter 5. Furthermore, the executable will have to be loaded
like any other MPI program executable.

### 2.3.3 Why use MPI for this project?

MPI has been chosen as the implementation method for this project. What follows is a
run-through of the reasons that motivated this decision and a description of some features
of MPI that will be used.

- **Portability**
  The source-code portability over different architectures provided by MPI means that
#include <mpi.h>

main(...) {
    MPI_Init(argc, argv);

    if (controller) {
        Controller(argc, argv);
    } else if (worker i) {
        Worker(i, argc, argv);
    }

    MPI_Finalize();
}

Figure 2.3: Format of MPI programs in C.

development can be carried out on a workstation cluster, rather than the (single-user!) AP1000.

- **Efficient Collective Communications**
  Collective communication routines provided by MPI are already efficient implementations. This is very useful when deriving the code for data distributions and redistributions from their mathematical model — there is no need to ensure efficiency, MPI is merely told what to do, and the actual implementation should be efficient on the particular platform used.

- **Interleaving Communication and Computation**
  MPI makes it easy to use different types of communication, e.g. blocking and non-blocking. The latter can be used to interleave communication with computation and thus hide some of the communication latency.

- **Sender-Driven Communication**
  Communication in MPI is very much sender-driven. This is better than receiver-driven communication (there is no need for a ‘fetch’), and, furthermore, one of the advantages of this project’s mathematical model for data distributions that will be described later is that it allows sender-driven communication. Each processor can work out which of its data it needs to send where.

- **Well-Behaved Messages**
  MPI gives some guarantees about the implementation of its operations, e.g. no overtaking of messages.

- **Timing Information During Development**
  MPI, as well as the AP1000 native communication functions allow for timing, however, the AP1000 simulator, Casim, does not provide timing information. Although there is no MPI simulator available at the moment\(^6\), development can be done on a workstation.

\(^6\)Somebody write one!
cluster, and that does provide timing information.

- **No need for making consecutive `send-packets`**.
  Derived datatypes in MPI make it possible to send compound information *without copying the data into a contiguous packet*. Instead, a ‘template’ is superimposed on existing data, stating which parts to send away (or into which locations to read new data). This is ideally suited for generating messages that consist of individual rows and/or columns of a matrix. Clearly, this will be very useful for this matrix library.

- **Relative Independence from Machine Topology**
  The Virtual Topologies feature of MPI makes it very easy to address processors within the context that is most suitable, e.g. as a mesh with coordinates, or as rings that form columns or rows of the mesh. This means that a design decision to assume a mesh of processors does not necessarily cause any problems if the actual architecture is not a mesh.

- **Doesn’t necessarily need ‘host’**.
  MPI programs can be run in SPMD style purely on the parallel architecture, with no distinct ‘host’. This is a very interesting point to look at — there probably isn’t any advantage to running the sequential part of the program (except I/O) just on one processor. To the contrary, a lot of communication could possibly be avoided by having no host. This needs to be looked at more closely.

- **Modularity**
  MPI Communicators allow for modularity in communication — i.e. communications that happen in the context of two different communicators are insulated from each other.

- **High-level routines make for relatively easy implementation**
  It is quite clear that by combining some of the more high-level MPI features, such as Reductions and Persistent Communication, several parallel algorithms can be implemented in a considerably less complex way than when the AP1000 native routines are used.

- **Language Issues**
  MPI can be used with C++ bindings, but not with any of the features of C++ that exceed C. This needs to be looked at, since last year’s project uses pretty much the full available power of C++.
  MPI uses `gcc` by default, whereas the AP1000 native routines run with `cc`.

- **Free MPI implementations are available.**

Some of the above points will be referred to in the design, and later on when implementation is discussed.

---

7Whether hardware-contention actually results in a slow-down is something that needs to be investigated.
2.4 Technical Information about the AP1000

The main platform for the implementation of this project will be the Fujitsu AP1000 supercomputer owned by the Imperial College / Fujitsu Parallel Computing Research Centre. This section briefly outlines some of the key features of this machine that are relevant to this project.

The AP1000 is a large-scale distributed-memory parallel computer, which can be configured for between 16 and 1024 nodes or ‘cells’, the machine here at Imperial College has 128 cells. The processors can operate independently and thus allow for MIMD parallel programming, however, the operating system software provides direct support for SPMD programming (see Section 2.3 and Figure 2.2), and that is the model that will be used for this project. The AP is connected to and controlled by a front-end host machine, a SPARC-Server 4/360 with 128MB of RAM.

Each cell consists of a SPARC processor running at 25MHz with 16MB RAM (four-way interleaved, with ECC), and with a 128KB, i.e. 32K word or 16K double-word, direct-mapped cache. This information about the cache will be used in determining the right blocksize for the implementation of algorithms such as matrix multiplication.

The cells are connected by three different networks, the B-Net, the T-Net, and the S-Net. The B-Net is a 32-bit, 50MB/s broadcast network that connects all cells and the host, and it is mainly used for host-cell communications. It supports scatter/gather operations. The T-Net provides cell-cell communications. It is arranged as a 2-dimensional (wraparound) torus. The bandwidth is 25MB/s on each link. This network uses cut-through or ‘wormhole’ routing. The presence of a routing controller (RTC) on each cell means that messages can be forwarded without requiring any action by the CPU of the cell concerned. The S-Net supports fast synchronisation and status-checking.

In addition to the basic architecture outlined above, several cells have additional units. The AP1000 at IFPC has 32 DDV (Distributed Disk and Video) units, and 16 Numerical Computation Accelerators. These consist of on-board micro-VP vector units, each with 16MB static RAM for fast memory access. These units have a peak performance of 100 MFLOPs.
Chapter 3

Mathematical Groundwork

The mathematical representation of data distributions which will be described in Chapter 4 requires a fair amount of linear algebra, particularly in the form of justifications for design decisions. Also, some published references make extensive use of linear algebra in dealing with data distributions, e.g. [44]. For that reason, this chapter will review some of the underlying mathematics involved, both in published work and in this report.

However, this is not purely a review, some sections are actually the mathematical groundwork for what is done in Chapter 4. In the form presented here, these sections are my work, however, I’m sure that there are published references. General references for claims made in this chapter are [67, 81, 85]. It will be indicated what is “my work” (with the proviso made).

Important sections in this chapter are Section 3.2 with its point about the structure of isomorphisms between additive vector spaces and Section 3.6 about “inverting the uninvertible”. Section 3.8 states the relationship between linear functions and matrices, and Section 3.10 looks at invertible matrices. These then lead to the definition of affine functions in Section 3.11, which is very extensively used in Chapter 4. Finally, Section 3.13 looks at tensors and also leads into some thoughts on how this material might be taken further in Section 3.14.

3.1 Vectorspaces, Bases, Dimensions

Let \((V, +)\) be a commutative group, i.e. + is associative in \(V\), there exist both an identity element and inverse elements in \(V\) with respect to +, and + is commutative in \(V\). Let \(F\) be a field. Then, \((V, +, F, \cdot)\), for short \(V\), is a vectorspace, if for all \(a, b \in V\) and all \(\alpha, \beta \in F\),

\[
\begin{align*}
\alpha \cdot (a + b) &= \alpha \cdot a + \alpha \cdot b \\
(\alpha + \beta) \cdot a &= \alpha \cdot a + \beta \cdot a \\
(\alpha \cdot \beta) \cdot a &= \alpha \cdot (\beta \cdot a) \\
1_F \cdot a &= a.
\end{align*}
\]

(3.1) (3.2) (3.3) (3.4)

\(V\) is also referred to as an \(F\)-vectorspace or a vectorspace over \(F^1\).

\(^1\)Notation. We will normally denote elements of vector spaces by small Roman letters \((a, b, \ldots)\), and elements of a field \(F\), i.e. scalars, by small Greek letters \((\alpha, \beta, \ldots)\). The notation \(1_F\) indicates that we mean the unit element from \(F\), similarly for \(0_V\).
A subset $B = \{b_1, \ldots, b_n\}$ of $V$ is called a basis of $V$ if and only if

$$\forall a \in V \left( \exists \alpha_1, \ldots, \alpha_n \in \mathcal{F} \left( a = \sum_{1 \leq i \leq n} \alpha_i b_i \right) \right)$$

(3.5)

and

$$\forall \alpha_1, \ldots, \alpha_n \in \mathcal{F} \left( \sum_{1 \leq i \leq n} \alpha_i b_i = 0_V \Rightarrow \alpha_1 = \ldots = \alpha_n = 0_\mathcal{F} \right)$$

(3.6)

(3.5) means that all elements of $V$ can be represented as linear combinations over $B$, and (3.6) means that $B$ is linearly independent. Let $V$ be a vectorspace with basis $B = \{b_1, \ldots, b_n\}$. Then, $n$ is called the dimension of $V$. It can be shown that all bases of a vectorspace have the same number of elements and that all vectorspaces have at least one basis [81].

The coordinates of an $N$-vector can be viewed as the vector space $(\mathbb{Z}_N, +_N, \mathcal{F}, \cdot)$, where $\mathbb{Z}_N$ is the set of all integers $\{0 \ldots N\}$, $+_N$ is addition modulo $N$, and $\mathcal{F}$ is a suitable field. Similarly, we can represent the coordinates of an $N \times N$-matrix as elements of $\mathbb{Z}_N^2$, and the same vector spaces can be used for a ring of $P$ processors, or a mesh. Operations in these vector spaces are multiplication by scalars (e.g. $-1$ for reversals) and translations, that correspond to addition of an offset vector $\left( t_i \right)$.

### 3.2 Linear Functions, Isomorphisms

Let $V$ and $\mathcal{V}$ be vectorspaces over some field $\mathcal{F}$ (e.g. $\mathbb{R}$). Then, a function $f : V \to \mathcal{V}$ is called a linear function, if for all $a, b \in V$, $\alpha \in \mathcal{F}$,

$$f(a + b) = f(a) + f(b)$$

(3.7)

$$f(\alpha a) = \alpha f(a).$$

(3.8)

Functions that conform to this pattern are also called homomorphisms. Invertible homomorphisms are called isomorphisms, and finally, isomorphisms that map $V$ onto itself are called automorphisms over $V$. If there exists an isomorphism between two vectorspaces $V$ and $\mathcal{V}$, then $V$ and $\mathcal{V}$ have identical properties in vectorspace theory, they are called isomorphic, and we write $V \cong \mathcal{V}$.

It can be shown that all isomorphisms $f : (\mathbb{R}, +) \to (\mathbb{R}, +)$ have the form $f(x) = \alpha x$, i.e. they are linear functions through the origin\(^2\). This may seem bizarre, but is very useful, because it means that if we are looking for a structure-preserving invertible function (a homomorphism) mapping an additive vectorspace onto itself, that function has to be a line through the origin. This will be used in Section 4.2.

### 3.3 Quotient Spaces and Canonical Functions

Let $U$ be a proper subspace of a vectorspace $V$, i.e. $U \subsetneq V$ and $U$ is a vectorspace. Let $x \in V$ and $x \notin U$, and define $x + U = \{x + u \mid u \in U\}$. $x + U$ is called the (left-) associated class of $x$ with respect to $U$.

It can be shown that two such classes are either identical or disjoint. The set of all $x + U$ is

\(^2\)Similarly, it can be shown that all isomorphisms $f : (\mathbb{R}^+, \cdot) \to (\mathbb{R}, +)$ are logarithms.
denoted by \( V/U \), \(|V/U| \) is called the index of \( V \) with respect to \( U \), and written \( \text{ind}(U, V) \).
We have [67]

\[
|V| = \text{ind}(U, V) \cdot |U|.
\]

(3.9)

We can define an equivalence relation \( \sim \) over \( V \) by \( (x \sim y) \iff (y \in x + U) \). Then, \( V/U \) can be viewed as the quotient-set \( V/\sim \), and \( x + U \) is the equivalence class \([x]\) and there is a mapping \( k : v \to V/U \), defined by \( k(x) = [x] \). It makes sense to define operations \( \oplus \) and \( \odot \) on \( V/U \) by \([x] \oplus [y] = [x + y]\), and \( \alpha \odot [x] = [\alpha \cdot x] \). \( V/U \) then is a vectorspace with respect to \( \oplus \) and \( \odot \), and \( k \) is a linear function. We call \( V/U \) the quotient-space of \( V \) over \( U \), and \( k \) the canonical function.

### 3.4 Kernel and Image of a Linear Function

Let \( V, \tilde{V} \) be vector spaces and \( f : V \to \tilde{V} \) a linear function. Then, we define

\[
\text{Kern} \, f = \{ x \in V \mid f(x) = 0_{\tilde{V}} \} \quad (3.10)
\]
\[
f[V] = \{ f(x) \mid x \in V \}. \quad (3.11)
\]

\( \text{Kern} f \) is called the kernel of \( f \) and \( f[V] \) the image of \( f \). It can be shown that \( \text{Kern} f \) is a subspace of \( V \) and that \( f[V] \) is a subspace of \( \tilde{V} \).

### 3.5 The Homomorphism Theorem

Let \( V, \tilde{V} \) be vector spaces and \( f : V \to \tilde{V} \) a linear function. Then, since \( \text{Kern} f \) is a subspace of \( V \), \( V/\text{Kern} f \) is a quotient-space, and we have the canonical function \( k : V \to V/\text{Kern} f \). Note that

\[
k(x) = [x] = \{ y \mid y \in x + \text{Kern} f \} = x + \text{Kern} f \quad (3.12)
\]

Then, \( f[V] \) is isomorphic to \( V/\text{Kern} f \), by means of the mapping \( \text{nat} : V/\text{Kern} f \to f[V], \text{nat}([x]) = f(x) \), and we have \( f = \text{nat} \circ k \). \( \text{nat} \) is known as the natural function and is invertible. If the original function \( f \) is surjective (‘onto’), then we have \( f[V] = \tilde{V} \), and therefore \( V/\text{Kern} f \cong \tilde{V} \). Figure 3.1 illustrates this relationship.

### 3.6 “Inverting the uninvertible”

This section illustrates the significance of Sections 3.4 and 3.5. In the form presented here, this is my work, however, there should be published references.

Suppose we have a non-invertible homomorphism \( f : V \to \tilde{V} \), but what we are really interested in is an inverse for that function, “\( f^{-1} \)”, which obviously doesn’t really exist. An example for this, that will come up in Chapter 4 is a mapping that maps all locations on a mesh of processors that hold a copy of some piece of data onto the ‘owner’ location

---

3This uses some properties of \( + \) and \( \cdot \), such as commutativity.
for that piece of data. In order to describe the copying operation that underlies this, we would like to have an inverse for that function. Using the homomorphism theorem, we can do something.

1. We restrict the range of \( f \) to the image of \( f \). This is fairly straightforward, we only consider those elements of \( V \) that are actually taken as values of \( f \). So, we re-write

\[
    f : V \rightarrow f[V] .
\]

\( f \) is now surjective (‘onto’), but not necessarily injective (one-to-one).

2. Suppose \( x_1, x_2 \) are both mapped onto \( y \) by \( f \), i.e. \( f(x_1) = f(x_2) = y \). Then, we claim that \([x_1] = [x_2] \).

   \textbf{Proof:}

   \[
   f(x_1) = f(x_2) \quad (3.14)
   \]

   \[
   \Leftrightarrow (\text{nat} \circ k)(x_1) = (\text{nat} \circ k)(x_2) \quad (3.15)
   \]

   \[
   \Leftrightarrow \text{nat}(k(x_1)) = \text{nat}(k(x_2)) \quad (3.16)
   \]

   \[
   \Leftrightarrow \text{nat}([x_1]) = \text{nat}([x_2]) \quad (3.17)
   \]

   Since \text{nat} is invertible by the Homomorphism theorem, it is injective, and hence

   \[
   \text{nat}([x_1]) = \text{nat}([x_2]) \Rightarrow [x_1] = [x_2] . \quad (3.18)
   \]

3. We consider the mapping

\[
    \text{nat}^{-1} : f[V] \rightarrow V/\text{Kern } f . \quad (3.19)
\]

Since \( g([x]) = f(x) \), we can write in a simple way \( \text{nat}^{-1}(f(x)) = [x] \). However, that is not very practical since our starting point are elements of \( f[V] \), that don’t carry the information which elements of \( V \) are mapped onto them.

But, suppose we have a function \( \hat{f}^{-1} \) that maps elements \( y \in f[V] \) to any one of the \( x \in V \) that are mapped onto \( y \) by \( f \). Then, by the above proof, we have

\[
    \text{nat}^{-1}(y) = \left[ \hat{f}^{-1}(y) \right] . \quad (3.20)
\]

\(^4\)The reasoning behind calling the function \( \hat{f}^{-1} \) is that it has the direction that we want for “\( f^{-1} \)”, but it need not be identical to \( f \) in its “forward” direction.
4. Recall the definition of quotient spaces above. We have:

$$
\left[ f^{-1}(y) \right] = f^{-1}(y) + \text{Kern } f
$$

(3.21)

What this shows is that we can now calculate the set of all \( x \in V \) that are mapped onto \( y \in f[V] \), provided we know \( \text{Kern } f \), i.e. the set of all \( x \in V \) that are mapped to \( 0_{f[V]} \).

Summary
Given a linear non-injective function \( f : V \to f[V] \), we have simplified the general problem of solving for all \( y \in f[V] \) equations of the form “find \( x \) such that \( f(x) = y \)” to the much simpler problem of solving once the equation \( f(x) = 0 \) and then having a function that will map \( y \in f[V] \) onto any one \( x \) such that \( f(x) = y \).

3.7 Theorem: Linear Completion

Let \( V \) be a vectorspace with basis \( B \), let \( \bar{V} \) be a vectorspace, and let \( h : B \to \bar{V} \) be any function from \( B \) to \( \bar{V} \). Then, there exists exactly one linear function \( h^* : V \to \bar{V} \) satisfying \( h^*(b) = h(b) \forall b \in B \). Recall that we can write all elements \( x \in V \) as \( x = \sum_{b \in B} \alpha_b b \). Because \( B \) is a basis linearly independent, the \( \alpha_b \) are unique. Then, \( h^* \) is defined by

$$
\begin{align*}
    x &= \sum_{b \in B} \alpha_b b \\
    &\rightarrow \sum_{b \in B} \alpha_b h(b)
\end{align*}
$$

(3.22)

the crucial point being that the \( \alpha_b \) are the same on both sides. We call \( h^* \) the linear completion of \( h \) on \( V \).

Since all vectorspaces have a basis, this theorem means that linear functions between vectorspaces are already uniquely defined by “what they do to” the elements of a basis of the vectorspace concerned.

3.8 Linear Functions and Matrices

Let \( V, \bar{V} \) be vectorspaces over some field \( \mathcal{F} \), and let \( B = \{ b_1, \ldots, b_n \} \) and \( \bar{B} = \{ \bar{b}_1, \ldots, \bar{b}_m \} \) be bases of \( V \) and \( \bar{V} \) respectively. Then the previous theorem states that for elements \( \bar{x}_1, \ldots, \bar{x}_n \in \bar{V} \), there exists a unique linear function \( f : V \to \bar{V} \) such that

$$
f(b_i) = \bar{x}_i, \quad 1 \leq i \leq n .
$$

(3.23)

Since the \( \bar{x}_i \) have a unique representation \( \bar{x}_i = \sum_{1 \leq k \leq m} \alpha_{ik} \bar{b}_k \) in \( \bar{V} \) with respect to \( \bar{B} \), we can write

$$
f(b_i) = \sum_{1 \leq k \leq m} \alpha_{ik} \bar{b}_k, \quad 1 \leq i \leq n .
$$

(3.24)

Hence, each linear function \( f : V \to \bar{V} \) is uniquely described with respect to the bases \( B \) and \( \bar{B} \) by an \((n, m)\)-matrix \((\alpha_{ik})\) of elements of the field \( \mathcal{F} \).

We denote the set of all linear functions \( f : V \to \bar{V} \) by \( \mathcal{L}(V, \bar{V}) \) and the set of all \((n, m)\)-matrices over \( \mathcal{F} \) by \( \text{Mat}(n, m, \mathcal{F}) \). It can be shown that there is a bijective mapping \( l : \mathcal{L}(V, \bar{V}) \to \text{Mat}(n, m, \mathcal{F}) \). The addition of linear functions, \( f + g : V \to \bar{V} \), defined by
\[(f + g)(x) = f(x) + g(x),\] corresponds to the addition of matrices, defined by
\[(\alpha_{ik} + \beta_{ik}) = (\alpha_{ik} + \beta_{ik}),\]
and similarly for scalar products. As might be expected, both \(\mathcal{L}(V, \mathcal{V}), +, \mathcal{F}, \cdot\) and \((\text{Mat}(n,m,\mathcal{F}), +, \mathcal{F}, \cdot)\) are vectorspaces, and we have
\[
\mathcal{L}(V_{(n)}, \mathcal{V}_{(m)}) \cong \text{Mat}(n,m,\mathcal{F}) .
\] (3.25)

Note that this is a natural extension of the statement in Section 3.2 that all linear functions mapping additive vectorspaces onto themselves take the form of a line through the origin. To see this: if \(V\) and \(\mathcal{V}\) are both 1-dimensional, then Equation 3.24 would simplify to \(f(b) = \alpha b\).

### 3.8.1 Dual Vectorspace and Linear Forms
Given a vectorspace \(V\) of dimension \(n\) over \(\mathcal{F}\), we can define the so-called dual vectorspace to \(V, \mathcal{L}(V, \mathcal{F})\). Elements of \(\mathcal{L}(V, \mathcal{F})\) are linear functions \(f : V \to \mathcal{F}\), and are called linear forms in \(V\). The significance of linear forms is that we can describe subspaces of \(V\) (e.g. lines, planes) by sets of linear forms.

### 3.9 Function Composition and Multiplication of Matrices
It was previously stated that the respective definitions of addition and scalar product correspond in \(\mathcal{L}(V, \mathcal{V})\) and \(\text{Mat}(n,m,\mathcal{F})\). Given two linear functions \(f : V \to \mathcal{V}\) and \(g : \mathcal{V} \to \mathcal{V}\), with associated matrices \((\alpha_{ik})\) and \((\beta_{ik})\), we now find that the composition \(h = f \circ g : V \to \mathcal{V}\) of these functions corresponds to matrix multiplication of the associated matrices, defined by
\[
(\gamma_{ij}) = (\alpha_{ik})(\beta_{kj}) = \sum_{1 \leq k \leq m} \alpha_{ik} \beta_{kj} .
\] (3.26)

Given this, it is clear why we can only multiply matrices with dimensions such as \((n, m)\) and \((m, p)\) — there is no such thing as the composition of functions where the range of the first is not the same as the domain of the second. Similarly, it is clear that matrix multiplication [composition of linear functions] is associative, but not normally commutative. The matrix corresponding to the identity-function is \(I = (u_{kj})\), with \(u_{kj} = 1\) for \(k = j\) and 0 otherwise.

### 3.10 Invertible Matrices
Recall that invertible linear functions are called automorphisms. The set of all automorphisms \(f : V \to V\) is written \(\text{Aut}(V)\). The matrices that correspond to automorphisms are a subset of \(\text{Mat}(n,n,\mathcal{F})\), and are called invertible, or regular matrices. The set of all regular \((n,n)\)-matrices is written \(\text{GL}_n(\mathcal{F})\). The question of exactly how we can see whether or not a matrix is regular will be answered shortly.

### 3.10.1 Determinants
Let \(P = \left( \begin{array}{c} 1 \ldots \n \\ i_1 \ldots i_n \end{array} \right)\) denote a permutation of the numbers \(1, \ldots, n\), and let \(S_n\) denote the set of all such permutations. Suppose that \(P\) can be achieved by a sequence of \(p\) individual

---

\(^5\)GL stands for ‘groupe linéaire’. 

exchanges of two neighbouring elements. Then, the sign of \( P \), \( \text{sign}(P) \), is defined as \((-1)^p\).

It can be shown that given any permutation \( P \), \( \text{sign}(P) \) is uniquely determined.

Let \((a_{ik})\) be an \((n, n)\)-matrix over \( F \). Then, the determinant of \((a_{ik})\), \( \det(a_{ik}) \), is defined as

\[
\det(a_{ik}) = \sum_{P \in S_n} \left( \text{sign}(P) \prod_{1 \leq j \leq n} a_{ij} \right) .
\] (3.27)

That means that \( \det(a_{ik}) \) is a sum of \( n! \) terms. For \( 2 \times 2 \) matrices, we have

\[
\det \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = a_{11}a_{22} - a_{12}a_{21} .
\] (3.28)

3.10.2 Rules about Determinants

1. Transposition of a matrix does not alter the determinant. That means that in the following, we can always read ‘row’ for ‘column’.

2. The determinant of a matrix whose columns are linearly dependent is zero.

3. Exchanging two columns of a matrix changes the sign of its determinant.

4. Adding a scalar multiple of one column of a matrix to another column does not change the value of its determinant.

5. \( \det(\beta(a_{ik})) = \beta \cdot \det(a_{ik}) \).

6. For triangular matrices,

\[
\det(a_{ik}) = \prod_{1 \leq i \leq n} a_{ii} .
\] (3.29)

Hence, \( \det(I) = 1 \).

7. Given a matrix \((a_{ik})\) and some column index \( s \) and row index \( r \), define \( A_{rs} \) to be the determinant of the matrix resulting from \((a_{ik})\) by replacing the \( s\)th column and the \( r\)th row with zeros, except for one 1 at position \((r, s)^6\). Then, with \( s \) remaining fixed,

\[
\det(a_{ik}) = \sum_{1 \leq r \leq n} a_{rs} A_{rs} .
\] (3.30)

\[
A_{rs} = \det \begin{pmatrix} a_{11} & \cdots & a_{1(s-1)} & 0 & a_{1(s+1)} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{(r-1)1} & \cdots & a_{(r-1)(s-1)} & 0 & a_{(r-1)(s+1)} & \cdots & a_{(r-1)n} \\ 0 & \cdots & 0 & 1 & 0 & \cdots & 0 \\ a_{(r+1)1} & \cdots & a_{(r+1)(s-1)} & 0 & a_{(r+1)(s+1)} & \cdots & a_{(r+1)n} \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{n(s-1)} & 0 & a_{n(s+1)} & \cdots & a_{nn} \end{pmatrix}
\]
3.10.3 Theorem: Regular Matrices

Using the definition of $A_{rs}$, define $\text{ad}(\alpha_{ik})$, the ‘adjunct matrix’ to $(\alpha_{ik})$, by $\text{ad}(\alpha_{ik}) = (A_{ik})$. Then, an $(n,n)$-matrix $(\alpha_{ik})$ is regular (invertible) if and only if $\det(\alpha_{ik}) \neq 0$, and the inverse of $(\alpha_{ik})$ is

$$(\alpha_{ik})^{-1} = \frac{1}{\det(\alpha_{ik})} \cdot \text{ad}(\alpha_{ik}) \quad .$$

Matrices with $\det(\alpha_{ik}) = \pm 1$ are called unimodular. Recall that the set of all regular matrices is written $\text{GL}_n(F)$. Then, we have for $n$-dimensional $V$,

$$\text{GL}_n(F) \cong \text{Aut}(V) \quad .$$

3.11 Affine Functions

Having defined regular matrices, we can now state the following definition. Affine functions $f : V \to V$ are functions of the form

$$f(\vec{x}) = A\vec{x} + \vec{t} \quad ,$$

where $A$ is a regular matrix. When used as geometric mappings, affine functions scale areas by $\det(A)$.

The inverse of the above affine function is

$$f^{-1}(\vec{y}) = A^{-1}\vec{y} - A^{-1}\vec{t} \quad .$$

Given an affine function $f(\vec{x}) = A\vec{x} + \vec{t}$ and a second affine function $g(\vec{x}) + B\vec{x} + \vec{s}$, we can define an ‘in between’ function $h$ such that $h(f(\vec{x})) = g(\vec{x})$. $h$ is affine\(^7\) and we have

$$h(\vec{y}) = (BA^{-1})\vec{y} + (\vec{s} - (BA^{-1})\vec{t}) \quad .$$

This will be used in Chapter 4 for defining redistributions between two affine distribution functions, and we will use the fact that $h$ is again an affine function.

3.12 Eigenvectors and Eigenvalues

Let $f$ be an affine function. Then, vectors $\vec{v}$, for which we have $f(\vec{v}) = \lambda \vec{v}$ with $\lambda$ a scalar are called eigenvectors of $f$. We can easily see that affine functions with $\vec{t} \neq \vec{0}$ do not have eigenvectors.

Eigenvectors $\vec{v}$ have to satisfy the equation $A\vec{v} = \lambda \vec{v}$, i.e. $(A - \lambda I)\vec{v} = \vec{0}$. This equation has solutions that are distinct from $\vec{0}$ if and only if the polynomial equation $\det(A - \lambda I) = 0$ has real solutions for $\lambda$. Such solutions are called eigenvalues of $f$.

A line $l$ that is parallel to an eigenvector of $f$ is mapped to a line that is parallel to $l$ by $f$. Furthermore, if some point $P \in l$ is mapped onto $l$ by $f$, then the whole line is mapped onto itself.

Assuming that $\lambda$ is an eigenvalue, then the equation $(A - \lambda I)\vec{v} = \vec{0}$ has solutions $\vec{v}$ that are distinct from $\vec{0}$, and these are known as the eigenvectors associated with $\lambda$. It is clear that if $\vec{v}$ is an eigenvector associated with $\lambda$, then so is any scalar multiple of $\vec{v}$. Hence, we can always state a normalised eigenvector $\vec{v}_0$. We have:

\(^7\)The product of two regular matrices is a regular matrix.
Eigenvectors associated to different eigenvalues are orthogonal.

If \( \lambda \) is an \( n \)-times solution to the above polynomial equation, then there are \( n \) linearly independent eigenvectors associated with \( \lambda \). Each linear combination of these is then also an eigenvector.

It follows from the two previous points that for an \( n \times n \)-matrix \( A \), or the corresponding function, we can always find \( n \) pairwise orthogonal eigenvectors.

### 3.13 Tensors

As the last section of this broad-sweep of mathematical background, we briefly look at tensors, since they are used in some publications on data distributions [44], [67] is not very comprehensive on this topic; better references are [78, 73, 68], and [38, 39], the latter of which is what [44] refers to.

Let \( V \) be an \( n \)-dimensional vector space with basis \( A = \{a_1, \ldots, a_n\} \). Then, each \( x \in V \) has a unique representation

\[
x = \sum_{1 \leq i \leq n} \alpha_i a_i.
\]  

(3.36)

If we think of the \( a_i \) as a coordinate system for \( V \), then the \( \alpha_i \) are the coordinates of \( x \) in that system.

Suppose we want to change to a different coordinate system, or algebraically, to a different basis, \( B = \{b_1, \ldots, b_n\} \). The question arises, how we find the new coordinates \( \beta_i \) such that

\[
x = \sum_{1 \leq i \leq n} \beta_i b_i.
\]  

(3.37)

We can uniquely represent the \( b_i \) in terms of the \( a_i \), as follows

\[
b_i = \sum_{1 \leq k \leq n} \tau_{ik} a_k.
\]  

(3.38)

The matrix \( (\tau_{ik}) \) is invertible. (All linear functions that map bases onto bases are automorphisms [67, 81].) Then, we can see how the coordinates are transformed:

\[
x = \sum_{1 \leq i \leq n} \beta_i b_i = \sum_{1 \leq i \leq n} \alpha_i a_i
\]  

(3.39)

\[
\Rightarrow \sum_{1 \leq i \leq n} \beta_i \left( \sum_{1 \leq k \leq n} \tau_{ik} a_k \right) = \sum_{1 \leq i \leq n} \alpha_i a_i
\]  

(3.40)

\[
\Rightarrow \sum_{1 \leq i \leq n} \beta_i \tau_{ik} a_k = \sum_{1 \leq i \leq n} \alpha_i a_i
\]  

(3.41)

\[
= \sum_{1 \leq k \leq n} \alpha_k a_k
\]  

(3.42)

\[
\Rightarrow \sum_{1 \leq i \leq n} \beta_i \tau_{ik} = \alpha_k.
\]  

(3.43)
We could also represent the old basis \( A \) with respect to the new one:

\[
a_i = \sum_{1 \leq i \leq n} \sigma_{ik} b_k
d\tag{3.44}
\]

and it can be shown that

\[
(\sigma_{ik}) = \left[(\tau_{ik})^T\right]^{-1}
d\tag{3.45}
\]

The old coordinates can then be transformed into the new ones as follows:

\[
\beta_k = \sum_{1 \leq i \leq n} \sigma_{ik} \alpha_i
\tag{3.46}
\]

Looking at equations (3.38) and (3.45), and keeping (3.46) firmly in mind, we can see that change of bases and change of coordinates are 'opposite' actions with respect to the matrix \((\tau_{ik})\). The coordinates \(\alpha_i\) are therefore said to behave in a 'kontra-variant' way under \((\tau_{ik})\). We say that matrices from the set \(\text{GL}_n(\mathcal{F})\), i.e. of type \((\tau_{ik})\), assign a first-degree tensor \((\alpha_i)\) to a vector \(x\).

**Explanation.**

We are now going backwards — start with a matrix \((\tau_{ik}) \in \text{GL}_n(\mathcal{F})\). This defines a basis transformation \(A\) to \(B\) on \(V\), where \(A\) and \(B\) are uniquely determined. The coordinates \((\alpha_i)\) of a point \(x\) with respect to the first basis, \(A\), are then our first-degree tensor for \(x\).

### 3.14 Conclusion and Further Work

Sections 3.1 to 3.12 have dealt with vector space algebra. This can be used to represent and understand linear transformations, and will be used extensively in Chapter 4. In all of that, we are implicitly staying within one coordinate system, vectors are represented with respect to their (Cartesian) unit coordinates. If we think of matrices and vectors as objects consisting of a collection (or grid) of points in Cartesian space, then the material presented in those sections is well suited to describing transformations of those objects within that space, such as skewings.

Section 3.13 has briefly introduced the idea of going from one coordinate system to another of the same dimension. What really happens when we distribute, say, a matrix from a host processor onto a mesh of processors is that we represent this matrix with respect to a higher-dimensional coordinate system. The inverse of this operation can be thought of as the vector dot product — we go from a higher-dimensional coordinate system and a set of higher-dimensional coordinates to a lower-dimensional value. This operation seems intrinsically uninvertible. What follows are a few fairly speculative ideas on how tensors might be used to tackle some of these problems.

- \(\{1, \ldots, N\}\) is the set of all (integer!) coordinates of an \(N\)-vector. Assuming that this is somehow a vectorspace and that we have a field of integer scalars to go with it (see

---

8 This uses again the theorem that vectors have a unique representation as linear combinations of bases.
below), we could define the way our vector is distributed by giving an “integer basis” with respect to which we express the coordinates $i$. This is best shown by example.

\begin{align*}
i &= 0 \cdot N + i \cdot 1 \\
i &= \left( i \div \frac{N}{P} \right) \cdot \frac{N}{P} + \left( i \mod \frac{N}{P} \right) \cdot 1 \\
i &= (i \mod P) \cdot 1 + (i \div P) \cdot P
\end{align*}

\begin{align*}
{\{N, 1\}} \\
{\left\{ \frac{N}{P}, 1 \right\}} \\
{\{1, P\}}
\end{align*}

The sets on the right could be thought of as integer bases, or coordinate systems for $i$. We could then ask the question how we get from one of these coordinate systems to another, which would lead onto tensors and the material hinted at in Section 3.13.

- A problem with the above approach is that sets of integers \{1, \ldots, N\} are not normally vectorspaces, never mind fields. We can view the set of the first $N$ integers as the quotient space $\mathbb{Z}/N\mathbb{Z}$, but this is only a field (which is what we need for the scalar multiples) if $N$ is prime.

It seems that exploring these ideas further along the lines indicated could be highly beneficial. However, that would require a more in-depth study of tensor analysis, and, possibly, of integer theory.
Chapter 4

Data Distributions

This chapter deals with the representation of data distributions. This is one of the key ‘interesting issues’ stated in Section 1.2, and in many ways represents the main work of this thesis. The chapter is structured as follows. First, requirements for a method for representing data distributions are discussed. In the next Section (4.2), different possible approaches for solving that problem are looked at, and a train of reasoning that leads to the approach chosen for this project is outlined. Section 4.3 gives simplifying assumptions that were made.

This is followed by an extended description of the method used in this project. That starts with four examples in Section 4.4, and is followed by the formal definition of firstly the affine transformations (4.5) and secondly the folding function (4.6), which together constitute the method for representing data distributions that has been developed in this project. Section 4.7 describes a neat technique that uses the homomorphism theorem to represent the copying of data. Section 4.8 the outlines how the mathematical models from the previous sections can be used to derive the code for implementing data distributions. Section 4.9 reviews related work.

A summary of the technique developed can be found in Section 4.10.

4.1 Requirements for Representing Data Distributions

A common approach to handling data distributions, partly in order to simplify the problem of optimising both for parallelism and data locality, is that the programmer directs a parallelising compiler how to align data [5]. In HPF for example, data distributions are described by special directives using keywords such as BLOCK or CYCLIC. This means that different possible distributions are essentially enumerated. That is also the case for the method used in last year’s project (see Section 2.1 and Figure 2.1), where distributions are ‘named’. The aim in this project was to find a more general method for describing data distributions, which should satisfy the following requirements.

- No Enumerations or ‘Naming’
  Different possible distributions should not be enumerated or ‘named’, but rather, there should be one uniform representation, capable of describing a sufficiently large number of distributions. This is not only a point of elegance, but also of efficiency. Using one uniform representation means that there is no increased complexity when a large number of different distributions are used.
Matrix- and not Element-based
The method should describe distributions with respect to whole matrices and vectors, rather than individual matrix elements. So ideas like having a set of element-processor pairs would not be acceptable. This means that distributions will be restricted to ‘regular’ ones in the sense that the same method is used for calculating the locations of all elements of a matrix or vector. Last year’s report stated that ‘arbitrary’ distributions have to be allowed. This will not be possible.

A Functional Mapping
The basic task of a representation for data distributions should be to map the location of a matrix or vector element onto a pair \((p,l)\) of a processor number and the local index on that processor. Hence, the representation of data distributions should have the basic form of a function acting on the coordinates of the data structure to be distributed.

Invertibility
The distribution function has to be invertible. This is crucial, since it allows calculating both where a certain matrix element is (which is what would be required for receiver-initiated communication), but also which global element any local element on a processor corresponds to. Having that information allows for sender-initiated communication, which is a lot better than receiver-initiated. Why? Sender-initiated communication avoids ‘request’ messages, but more important still, the fact that cell-to-cell interrupts are not generally possible on parallel architectures (and certainly not on the AP1000 or workstation clusters), the only way to implement receiver-initiated communication is for all cells to regularly poll for request messages. This is not necessary with sender-initiated communication. With invertible distributions, we can even know at the receiving end what data is to be expected.

Calculating Redistributions and Closure
It should be possible to calculate a redistribution function from two different data distributions. This is essential for the lazy libraries approach in situations where no implementations with compatible distributions for a sequence of operations exist. Similarly, given a distribution function and a redistribution, that should always define a new valid distribution function, and, it should be possible to calculate that new distribution from the given original distribution and redistribution.

A Cost Model
For the optimisation phase, it is essential that we have a model for the cost of redistributions.

Deriving the Implementation Code
Finally, it should be possible to calculate the code for carrying out distributions and redistributions from the functions. This seems quite tricky, but it is essential for having a truly general way of describing data distributions. Furthermore, if we can calculate the code for implementing a redistribution, we can also calculate its cost!

Range of distributions to be represented
The range of distributions to be represented by our model should be a large set of regular distributions. What is meant by that is that all elements of a data object
should obey the same distribution rule. That should also mean that we can determine
the global distribution function from what happens to a small set of ‘basis’ elements.

In summary, we are looking for a general, invertible functional model for representing data
distributions and redistributions that allows calculating both the implementation code and
the associated cost, and that satisfies closure.

4.2 Discussion of Different Possible Approaches

This section will discuss different possible approaches to representing data distributions.
Consideration of the points made here led in the end to the model described in this chapter.

Two possibilities that were considered are ‘naming’, and the concept of a data ‘cache’.
However, ‘naming’ of distributions does not meet the requirements set out above.

In last year’s project, that approach was combined with the notion of a data ‘cache’,
as detailed in Section 2.1. The cache-approach does explicitly allow for duplication of
data on different processors. This immediately raises a problem — a mapping that maps
one location onto a number of different processors would not be a function, and unless
there is some very regular pattern to this (as will be detailed in Section 4.7). That causes
problems, and would most likely result in having to enumerate processors that hold data,
which is not acceptable. Another point is that this is essentially a local, submatrix-based
approach to describing the location of data. If the distribution is regular, that results in
a lot of redundant information, e.g. in a blocked distribution, the submatrix held on the
first processor would allow calculating what submatrices are held on all other processors.
If on the other hand distributions are not regular, i.e. submatrices can migrate round the
architecture independently and with no regular pattern, it is very hard to see how global
optimisation of data distributions can be possible.

The idea leading up to the technique adopted for this project was to use a method
similar to that used by Banerjee [6] for describing transformations of double loop nest
iteration spaces. Briefly, Banerjee uses unimodular $2 \times 2$-matrices to map the iteration
vectors of the old loop nest onto the iteration vectors of the transformed nest.

★ So, what we want is a function that maps the indices of the undistributed array
onto the processor- and local indices of the distributed array. To illustrate this, assume we
want to distribute a matrix onto a mesh of processors in a blocked fashion. Then, we need
the following type of mapping, where $i, j$ are the matrix indices, $p_x, p_y$ are the processor
coordinates, and $l_x, l_y$ are the local coordinates on each processor.

$$
\begin{pmatrix}
  i \\
  j
\end{pmatrix}
\rightarrow
\begin{pmatrix}
  p_x \\
  p_y \\
  l_x \\
  l_y
\end{pmatrix}
$$

This basic point leads to the following considerations.

The Problem

There are two separate problems that need to be addressed. Firstly, distribution in a
more narrow sense — the problem of how we distribute the elements of a data object over
the parallel architecture. The two most common examples that we need to account for
are blocking and cyclic blocking (see for example [49]). Secondly, we have to be able to
represent any regular transformation of the distributed data object. Examples for this would be matrix transposes, skewings [72, 51] and translations. Bearing in mind that what we require is a functional mapping that acts on the indices of data objects, these two aspects somehow need to be integrated into one general distribution function.

An Affine Transformation

The function for describing transformations of data objects, such as matrix transposition, has to fulfill the following two requirements.

1. It has to be linear. This arises mainly from the point in Sections 3.2 and 3.8 that all homomorphisms (i.e. structure-preserving functions) between additive vector spaces take the form of linear functions through $0$. It would clearly be very desirable that transformation functions have the property of being structure-preserving. Adding a constant offset is not a homomorphism, but is a linear operation and very easy to understand. There is no reason why at some point non-linear transformations should not be considered, however, this would clearly complicate things considerably. Two examples for problems with non-linear transformations would be firstly, that the range of transformation functions always has to be the integers, since the values taken by the function represent coordinates, and secondly, that non-linear transformations might result in either an ‘expansion’ or, worse, a ‘compression’ of a data object, where multiple locations would end up being mapped onto one.

2. Furthermore, the function has to be invertible. This is stated in the requirements, but apart from that, it is also clear that there would be big problems with having non-invertible transformations, since this would almost certainly mean a loss of information.

Given these two constraints, it follows that our transformation function has to be an affine function on the coordinate space, and therefore has to take the form (see Section 3.11)

$$ f(x) = Ax + b $$

where $x$ and $b$ are vectors, and $A$ is an invertible matrix. Section 4.5 discusses the exact role and significance of this affine function.

A Folding Function

The above affine transformation unfortunately cannot be used to represent the distribution of a data structure (in the more narrow sense) onto the parallel architecture. This can be seen by the fact that in, say, a blocked distribution, location $i$ in a vector is mapped onto processor $(i \text{ div } N_P)$ with local index $(i \text{ mod } N_P)$, in other words, we need div and mod. But:

- div and mod are not affine [23].

A common solution to this type of problem is to represent the distribution function as the composition of two functions:

$$ d = \phi \circ f $$

where $f$ is affine, and $\phi$ is a non-affine ‘folding function’. Note that the composition of affine functions is always affine, but that the composition of a number of functions where any one is not affine cannot be affine. This will be addressed in detail in Section 4.6.
Virtual Processors
The idea of using a composition of an affine and a non-affine function is otherwise more commonly viewed as mapping data first onto virtual processors, and then mapping the virtual processors onto physical processors by means of a folding function, and that is also the terminology that we will use henceforth. Virtual processors are a very common device for solving problems like this; see for example [23, 72].

Redistributions
As mentioned in the requirements, we are looking for a representation for data redistributions that can be calculated from two given distributions between which we wish to redistribute. An advantage of using affine functions is that given two affine functions \( f(\vec{x}) = A\vec{x} + \vec{l} \) and \( g(\vec{x}) = B\vec{x} + \vec{s} \), we have an affine redistribution function \( h \), such that \( h(f(\vec{x})) = g(\vec{x}) \), as follows:

\[
h(\vec{y}) = (BA^{-1})\vec{y} + (\vec{s} - BA^{-1}\vec{l}).
\] (4.4)

Since in turn the composition of any two affine functions is again affine, we can always calculate the new resulting affine distribution function from the original function and a redistribution. This will be dealt with in Section 4.5.5.

Copying of Data
The final preliminary point to mention is the copying of data onto more than one processor. The problem here is that mapping one location onto many is not a function. However, with certain fairly tight restrictions, a method has been developed that uses the mathematical background of Section 3.6 to describe a number of very useful copy-distributions, such as the duplication of scalars on all processors and of vectors that are blocked across the columns of a mesh on all rows, or vice versa. The basic idea is to give the inverse of the copy function, i.e. a mapping from all the processors that have a copy of some data to the ‘owner’. Provided that these functions are linear, we can then construct an ‘inverse’ that allows us to calculate all processors that hold a copy of a piece of data without having to enumerate them. This will be discussed in Section 4.7.

4.3 Notation and Assumptions
This section firstly introduces the notation used in the following sections, and then describes assumptions that were made for simplification.

Notation
We will use \( N \) to denote the number of elements of a vector, and \( N_x, N_y \) to denote the number of rows and columns of a matrix respectively. As indices for vectors, \( i \), and for matrices \((i, j)\). For the parallel architecture, \( P \) will be used to represent the total number of processors, and when using a mesh, \( P_x \) and \( P_y \) will be used to denote the number of processors along the two dimensions. The rank and coordinates of a specific processor will be denoted by lower-case letters, \( p, p_x, p_y \). Virtual processor coordinates will be written \( v_x \) and \( v_y \), and as previously indicated, affine functions will normally be denoted by \( f \), the folding function by \( \phi \), and the entire distribution function by \( d \).
Assume P|N
We now discuss assumptions. Firstly, we will always assume that $P_x$ and $P_y$ (or $P$ in a ring) divide $N$ for vectors and both $N_x$ and $N_y$ for matrices. This is a common simplification (see e.g. [44]) and not a conceptual restriction, dealing with cases that do not conform to this is merely quite ‘messy’. Furthermore, for most matrix operations, we could easily pad matrices with zeros up to the required size, without influencing the result. Let $A$, $B$ be matrices that haven’t got ‘enough’ columns and rows. We have

$$\begin{pmatrix} A & 0 \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} B & 0 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} A \cdot B & 0 \\ 0 & 0 \end{pmatrix}$$

(4.5)

for suitable numbers of all-zero rows and columns. For the situation where this assumption does not hold, care has to be taken that this is implemented in such a way that one processor has less work than all the others, rather than one processor having more work.

Architectures and “Changing the Geometry”
Secondly, the architectures dealt with will be rings and meshes. As it turns out, uniprocessors are a trivial case of these as far as the distribution method described here is concerned, and it seems that hypercubes could also be dealt with as a very natural extension. However, we will not consider redistributions that involve what in Connection Machine [43] terms is called changing the geometry, i.e. changes from one type of architecture to another. In fact, mapping processor numbers of a mesh onto a ring can be done quite easily:

$$\begin{pmatrix} p_x \\ p_y \end{pmatrix} \cdot \begin{pmatrix} P_x \\ 1 \end{pmatrix} = (p)$$

(4.6)

However, this mapping is obviously not affine, and hence trying to incorporate this would raise new problems. This is also a simplification that is found in the literature [23], but unlike the previous point, the reasoning behind it is not that it could be easily filled in, but rather that it represents a whole new set of problems. A different angle onto this is that the MPI concept of virtual topologies (see Section 2.3) allows addressing the same architecture in differing ways, and hence might also allow addressing different architectures in the same way, but this has to remain for future work.

No Host
It would simplify matters considerably if the architecture dealt with was uniform, i.e. all processors have the same amount of memory, and communication between processors has the same latency and bandwidth throughout the architecture. On a workstation cluster, this would normally be broadly correct. However, when using the AP1000, the host processor is different to all others, and has a considerably higher communication latency than that for inter-cell communications. Because of that, the controller process for the lazy libraries system will be Cell 0, and not the front-end SPARC machine. MPI makes it quite easy to do this by means of the “nohost” option. This is an approach that is also otherwise used: when part of a program is to be executed on a supercomputer, we can either make a call to the supercomputer from the processor running the original code, or, we can transfer control to the supercomputer altogether. The second option is what will be used here.
**Wormhole Routing**

Finally, where that is relevant, we will always assume that networks use wormhole, or cut-through routing. This has become the most efficient flow-control technique for multicomputers [22], with various either deterministic or adaptive routing mechanisms being used. The absence of cyclic dependencies is a necessary and sufficient condition for avoiding deadlock with deterministic routing. There are weaker conditions for adaptive routing algorithms, but that will not be needed. However, it should be noted that contention, even if it does not result in deadlock, will cause higher node delays. Therefore, care has to be taken to make communication contention-free. Under that assumption, the message-time for networks with wormhole routing is more or less independent of the distance across the network between the communicating processors.

### 4.4 Representing Data Distributions — Four Examples

Before formally defining the data distribution method adopted, we will first give four specific examples: a vector and a matrix distributed over both a ring and a mesh. This will then in the next section lead to a formal definition of data distributions that incorporates all four examples from this section.

#### 4.4.1 Vector onto Ring

This is the simplest case. The coordinate space of both the vector and the ring is 1-dimensional. Therefore, the virtual processor space will also be 1-dimensional.

**Identity Distributions**

At first, we will look at an “identity mapping”, i.e. a mapping that represents the fact that the entire vector is located on one processor. This is an important case, since after a ‘force’, and for the purpose of user-input, or input from file, the entire vector needs to be located on the controller processor. A suitable folding function for identity mappings of vectors onto rings is

\[
\phi(v_i) = \begin{pmatrix} (v_i \div N) \mod P \\ v_i \mod N \end{pmatrix} \tag{4.7}
\]

where \(v_i\) represents the virtual processor coordinates, \((v_i \div N) \mod P\) is the processor index and \(v_i \mod N\) is the local index on that processor. Given this, we can now give the affine functions, and for clarity, the complete distribution functions for identity distributions of vectors onto rings.

**Entire vector on processor 0**

\[
(i) \xrightarrow{f} (1)(i) + (0) = (i) \xrightarrow{\phi} \begin{pmatrix} 0 \\ i \end{pmatrix}
\]

**Entire vector on processor \(p\)**

\[
(i) \xrightarrow{f} (1)(i) + (pN) = (i + pN) \xrightarrow{\phi} \begin{pmatrix} p \\ i \end{pmatrix}
\]
Entire vector in reverse order on processor 0

\( (i) \xrightarrow{f} (-1)(i) + (N - 1) = (N - 1 - i) \xrightarrow{\phi} \left( \begin{array}{c} 0 \\ N - 1 - i \end{array} \right) \)

Notice that all the affine mappings are \textit{unimodular} in their multiplicative element. This means that we have a ‘dense’ mapping, i.e. if for the distributed vector, locations \( i \) and \( j \) with \( i - j > 1 \) contain some element of the vector, then so do all elements in between, and similarly for processors.

The given mappings are invertible. Let \( (p, l) \) be any pair of processor and local index, then we can calculate the original array index from this by left-multiplying with \( \left( \begin{array}{c} N \\ 1 \end{array} \right) \) to obtain the virtual processor index and then applying the inverse affine function. As an example, consider the case where the entire vector resides on processor \( p \).

\[
\left( \begin{array}{c} p \\ l \end{array} \right)^T \left( \begin{array}{c} N \\ 1 \end{array} \right) = pN + l \xrightarrow{f^{-1}} (i)
\]

As stated in Section 3.11, the inverse of an affine function \( f(x) = ax + t \) is \( f^{-1}(y) = \frac{1}{a}y - \frac{1}{a}t \), so in the above example, \( f^{-1}(v_i) = (1)(v_i) - (pN) \).

\textbf{Blocked Distributions}

For blocked distributions, the folding function becomes

\[
\left( \begin{array}{c} p \\ l \end{array} \right) = \phi(v_i) = \left( \begin{array}{c} v_i \text{ div } \frac{N}{P} \\ v_i \text{ mod } \frac{N}{P} \end{array} \right).
\]

Using this, we can now represent any number of blocked distributions for vectors onto rings.

Standard blocking of vector over ring

\( (i) \xrightarrow{f} (1)(i) + (0) = (i) \xrightarrow{\phi} \left( i \text{ div } \frac{N}{P} \right) \)

Standard blocking, but rotated by \( p \) processors

\( (i) \xrightarrow{f} (1)(i) + \left( p \frac{N}{P} \right) = (i + p \frac{N}{P}) \xrightarrow{\phi} \left( i \text{ div } \frac{N}{P} + p \right) \)

Reversed vector blocked over ring

\( (i) \xrightarrow{f} (-1)(i) + (0) = (-i) \xrightarrow{\phi} \left( \frac{N}{P} - 1 - (i \text{ mod } \frac{N}{P}) \right) \)

Blocked vector shifted right by \( r \) elements

\( (i) \xrightarrow{f} (1)(i) + (r) = (i + r) \xrightarrow{\phi} \left( \frac{(i + r) \text{ div } N}{P} \right) \)

These functions again are invertible. We left-multiply a \( (p, l) \) pair by \( \left( \begin{array}{c} N \\ 1 \end{array} \right) \) in order to obtain the virtual processor coordinate, and then apply the inverse affine function:

\[
\left( \begin{array}{c} p \\ l \end{array} \right)^T \left( \begin{array}{c} N \\ 1 \end{array} \right) = \left( p \frac{N}{P} + l \right) \xrightarrow{f^{-1}} (i).
\]
Cyclic Distributions

Finally, we will look at one example for a cyclic distribution. Here, the folding function is

\[
\begin{pmatrix} p \\ l \end{pmatrix} = \phi(v_i) = \begin{pmatrix} v_i \mod P \\ v_i \div P \end{pmatrix}.
\]

Then, the whole distribution function for a cyclic blocking of a vector onto a ring is

\[
(i) \rightarrow (1)(i) + (0) = (i) \rightarrow \begin{pmatrix} i \mod P \\ i \div P \end{pmatrix},
\]

and this mapping can be inverted by left-multiplying \((p, l)\) by \((1/P)\) and applying the inverse affine function:

\[
\begin{pmatrix} p \\ l \end{pmatrix}^T \begin{pmatrix} 1 \\ P \end{pmatrix} = (p + lP) \xrightarrow{\phi^{-1}} (i).
\]

4.4.2 Vector onto Mesh

Our next example shows how we can represent the distribution of a vector over a mesh of processors. We will focus on those points that are different to the previous example. Shifts, rotations, reversals etc. can be included if required. With vectors, we would normally have a one-dimensional coordinate space for data, however, since we are wanting to map onto a (2-dimensional) mesh, our virtual processor space needs to be 2-dimensional. Since affine functions only exist between vector spaces of the same dimension (3.11), this means that we also need to represent the original vector space in two coordinates. That can be done quite easily by viewing a vector as a two-dimensional data structure where the number of elements in the second dimension is \(N^2\).

For the moment, we will only consider distributions of the vector over one row or column, the issue of copying will be addressed in Section 4.7.

Folding Functions

First, we will give the folding functions for identity, blocked and cyclic distributions. It should be noted that these are ‘special cases’, a general and much more comprehensive folding function that covers vectors, matrices, rings and meshes will be given in Section 4.6.

We have a two-dimensional virtual-processor tuple \((v_i, v_j)\) that has to be mapped onto a three-dimensional tuple \((v_i', v_j', l)\) in which the first two dimensions hold the processor coordinates and the third holds the local index.

\[
\begin{pmatrix} v_i \\ v_j \end{pmatrix} \rightarrow \begin{pmatrix} (v_i \div N) \mod P_x \\ (v_j \div N) \mod P_y \\ \max(v_i \mod N, v_j \mod N) \end{pmatrix}
\]

Identity (4.13)

\[
\begin{pmatrix} v_i \\ v_j \end{pmatrix} \rightarrow \begin{pmatrix} (v_i \div N/P_x) \\ (v_j \div N/P_y) \\ \max(v_i \mod N/P_x, v_j \mod N/P_y) \end{pmatrix}
\]

Blocked (4.14)

\[
\begin{pmatrix} v_i \\ v_j \end{pmatrix} \rightarrow \begin{pmatrix} v_i \mod P_x \\ v_j \mod P_y \\ \max(v_i \div P_x, v_j \div P_y) \end{pmatrix}
\]

Cyclic (4.15)

\footnote{This does not simply mean regarding a vector as a matrix with one row — for vectors, we will allow data copying along the column-dimension, where for matrices we demand that \(P_x | N_x\) and \(P_y | N_y\).}
Affine Functions

With this, we can now complete vector-onto-mesh distribution by stating the affine functions.

Vector distributed over Row 0

\[
\begin{pmatrix} i \\ 0 \end{pmatrix} \overset{f}{\mapsto} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} i \\ 0 \end{pmatrix}
\]

(4.16)

Vector distributed over Column 0

\[
\begin{pmatrix} i \\ 0 \end{pmatrix} \overset{f}{\mapsto} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} i \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ i \end{pmatrix}
\]

(4.17)

Vector distributed along diagonal

\[
\begin{pmatrix} i \\ 0 \end{pmatrix} \overset{f}{\mapsto} \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} i \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} i \\ i \end{pmatrix}
\]

(4.18)

The last function (along diagonal) is probably of limited use, since we will normally copy vectors along the dimension in which they are not distributed. Notice that the folding function would collapse the local address into one dimension, which is what we would want. Rotations and other additional transformations could be applied in a way analogous to the ring example.

4.4.3 Matrix onto Ring

We again begin by giving the folding functions, where we now have a two-dimensional data structure that is being mapped onto a one-dimensional parallel architecture. This will result in a two-dimensional virtual processor space with coordinates \((v_i, v_j)\) which has to be mapped to a three-dimensional tuple \((p, l_x, l_y)\) where \(p\) is the processor index, and \(l_x, l_y\) are the local indices\(^2\).

\[
\begin{pmatrix} v_i \\ v_j \end{pmatrix} \overset{\phi}{\mapsto} \begin{pmatrix} (v_i \div N_x) \mod P \\ v_i \mod N_x \\ v_j \mod N_y \end{pmatrix}
\]

Identity \hspace{1cm} (4.19)

\[
\begin{pmatrix} v_i \\ v_j \end{pmatrix} \overset{\phi}{\mapsto} \begin{pmatrix} (v_i \div N_x/P) \\ v_i \mod N_y \\ v_j \mod N_y/P \end{pmatrix}
\]

Blocked \hspace{1cm} (4.20)

\[
\begin{pmatrix} v_i \\ v_j \end{pmatrix} \overset{\phi}{\mapsto} \begin{pmatrix} (v_i \mod P) \\ v_i \div P \\ v_j \mod N_y \end{pmatrix}
\]

Cyclic \hspace{1cm} (4.21)

We can now again complete the example by stating a number of different affine functions for matrix-onto-ring distributions.

\(^2\)Care needs to be taken here that if in the affine function \(\det(A) = -1\), the dimensions \(N_x\) and \(N_y\) are exchanged. This will be addressed more comprehensively in Section 4.5.
Standard Matrix Distribution
\[
\begin{pmatrix} i \\ j \end{pmatrix} \mapsto \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix}
\]
(4.22)

Transposed Matrix Distribution
\[
\begin{pmatrix} i \\ j \end{pmatrix} \mapsto \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} j \\ i \end{pmatrix}
\]
(4.23)

Skew Columns by Rows
\[
\begin{pmatrix} i \\ j \end{pmatrix} \mapsto \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} i \\ i + j \end{pmatrix}
\]
(4.24)

Skew Rows by Columns
\[
\begin{pmatrix} i \\ j \end{pmatrix} \mapsto \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} i + j \\ j \end{pmatrix}
\]
(4.25)

The last two examples (skew columns by rows and skew rows by columns) show why the folding function needs to have the component \(v_j \mod N\) for the second local index \(l_y\) — this will ensure a wraparound effect for skewings or rotations along that dimension.

### 4.4.4 Matrix onto Mesh

The affine functions here are of the same kind as for the matrix-onto-ring distributions, i.e. we can represent transposes, skewings, plus obviously rotations and other transformations that were shown in the very first example. So, we will here only give the folding functions. The virtual processor space is again two-dimensional, but we now need to map onto a four-dimensional tuple, with the first two dimensions representing the processor coordinates and the last two the local indices. As already hinted at in the previous example, we have to be careful that if \(\det(A) = -1\) in the affine function, the dimensions \(N_x\) and \(N_y\) are exchanged. This will be addressed in more detail in the next section, but for now note that the processor dimensions do not change.

\[
\begin{pmatrix} v_i \\ v_j \end{pmatrix} \mapsto \begin{cases} \left( v_i \div N_x \right) \mod P_x \\ v_i \mod N_x \\ v_j \mod N_y \end{cases}
\]
(Identity) \hspace{1cm} (4.26)

\[
\begin{pmatrix} v_i \\ v_j \end{pmatrix} \mapsto \begin{cases} \left( v_i \div N_x / P_x \right) \\ v_i \div N_y / P_y \\ v_j \div N_x / P_x \\ v_j \div N_y / P_y \end{cases}
\]
(Blocked) \hspace{1cm} (4.27)

\[
\begin{pmatrix} v_i \\ v_j \end{pmatrix} \mapsto \begin{cases} \left( v_i \mod P_x \right) \\ v_i \div P_x \\ v_j \div P_y \end{cases}
\]
(Cyclic) \hspace{1cm} (4.28)

The above currently assumes that the same type of distribution is used in each dimension. That is not necessary — we could, for example have a cyclic distribution of the rows, while the columns remain blocked. This will be addressed further in the formal definition for folding functions in Section 4.6.
4.5 The Affine Transformation Function

Having given four extensive examples, we can now proceed to define formally the affine part of the data distribution scheme presented in this chapter.

4.5.1 Data and Architecture Dimensions

Let \( d_d \) denote the number of dimensions of the data structure we wish to distribute, and let \( d_a \) denote the number of dimensions of the parallel architecture. So, for instance for a vector we have \( d_d = 1 \), for a matrix \( d_d = 2 \), for a ring \( d_d = 1 \), for a mesh \( d_d = 2 \) and for a hypercube \( d_d = \log_2 P \). Two special cases that need to be noted are \( d_d = 0 \), which represents a scalar, and \( d_a = 0 \), which represents a uniprocessor.

Then, let formally \( N \) be a \( d_d \)-dimensional array that contains the number of data elements in each data dimension. We can define an augmentation \( \tilde{N} \) of a \( d_d \)-dimensional data structure \( N \) onto a higher dimension \( d_d \) as follows.

\[
\tilde{N}_i = \begin{cases} 
N_i, & 0 \leq i \leq (d_d - 1) \\
1, & d_d \leq i \leq (d_d - 1)
\end{cases}
\]  

It turns out to be convenient to augment all data structures to the highest number of data dimensions that is used within a problem. The data indices in each dimension \( i \) then run from 0 to \( N_i - 1 \). For higher dimensions, we will use \( i_0, \ldots, i_{d_d-1} \) as data indices, however, for vectors, we will simply use \( i \), and for matrices \( i_0 = i \) and \( i_1 = j \).

Note that if we are dealing with just one data structure, we can always ignore any augmentation — the “augmented part” of \( N \) has been instantiated to the appropriate identity elements (1 for multiplication and 0 for addition) for any operations that are carried out.

Similarly, let \( P \) be a \( d_a \)-dimensional array that contains the number of processors in each architecture dimension. We can again define an augmentation \( \tilde{P} \) to a higher dimension. There may seem to be little use for that since we are unlikely to want to deal with distributions onto different architectures at the same time. However, if we are trying to determine (lazily, at runtime) what the best configuration for a certain algorithm is, we might have to look at performance models for different possible architecture configurations. In this case, we will augment all onto the highest number of dimensions we are interested in. Again, indices run from 0 to \( P_i - 1 \). We will for the general case use \( p_0, \ldots, p_{d_a-1} \) to denote processor indices, but for the special cases of ring and mesh, we will use simply \( p \) and \( p_x, p_y \) respectively.

Examples

To illustrate this, we will give a few examples. For scalars, \( d_d = 0 \), and if we use scalars with other data structures, we have \( N_i = 1 \ \forall i \) and all indices are 0. Note that it doesn’t matter how many 0 offsets we have — we still refer to the same, correct location. If we use a vector with matrices, we would augment the vector to two dimensions, so that \( N_0 \) is the number of elements of the vector and \( N_1 = 1 \). Note that this means that \( i_0 \) is always 0, and we are effectively representing element \( i \) of the vector by the tuple \( (i_0) \). For a hypercube of \( P \) processors, we would have \( d_a = \log_2 P \), and \( P_i = 2 \ \forall i \).
4.5.2 The Virtual Processor Space

The affine function maps data elements onto virtual processors. From those, the folding function then maps them onto the physical processors. An important point to realise is that the affine function determines which data dimension is distributed over which architecture dimension by its mapping of data dimensions onto virtual processor dimensions.

Notation: Let \( d_v \) denote the number of dimensions of the virtual processor space, and let \( V \) be a \( d_v \times d_v \)-dimensional array analogous to the arrays \( N \) and \( P \).

If \( d_d = d_a \) in some problem, then things are relatively simple. We set \( d_v = d_d = d_a \), and that allows us to use the affine mapping to determine which data dimension is to be distributed over which architecture dimension.

An issue that requires special attention is what happens when the number of data dimensions and the number of architecture dimensions are not the same. In that situation, there is either a choice of along which architecture dimension the data are to be distributed (if \( d_d > d_a \)), or there is a choice of which of the data dimensions are to be distributed and which not (if \( d_d > d_a \)). There are two possible ways of addressing this problem in terms of the dimensionality of the virtual processor space.

1. Let \( d_v = (d_d + d_a) \). This will be referred to as the “full-dimensional” approach. It results in quite large coordinate vectors, and especially, quite a large regular matrix \( A \). Furthermore, the virtual processor space is then “sparse”, so there seems at first sight to be a lot of redundancy, so for example a distribution of a matrix onto a ring would result in 3-dimensional virtual processor space with one index always zero. These considerations led to this idea being put to one side for the purpose of this project, however, it became clear that there is some additional flexibility offered by this approach\(^3\), so it should be reconsidered for future work.

2. Let \( d_v = \max(d_d,d_a) \). This will be referred to as the “compressed” approach. It means that we only use the minimum number of dimensions necessary to avoid loss of information.

So, if \( d_d < d_a \), we will have \( d_d < d_v \), and we can determine over which physical dimensions of the architecture we distribute which data dimensions by the dimensions of \( V \) that the dimensions of \( N \) are mapped to. If we augment \( N \) to \( d_v \) before distribution, then we will have 0 coordinates in all architecture dimensions along which we are not distributing the data. For example, when distributing a vector onto a mesh, we augment the vector coordinates to \((v,0)\), and we can have virtual processor coordinates either of the form \((v,0)\), which means that we distribute the vector along row 0, or of the form \((0,0)\), which means that we distribute along column 0.

Conversely, if \( d_d > d_a \) and therefore \( d_v > d_a \), we make the convention that the last \( d_v - d_a \) dimensions are not distributed. So, if we distribute a matrix over a ring, we get \( d_v = 2 \), and if we map the first matrix dimension (i.e. the rows) onto the first virtual processor dimension, we distribute by rows, with each processor holding complete columns, or, if we map the second matrix dimension onto the first virtual processor dimension, we distribute by columns.

Some care needs to be taken with this approach, because we use the same number of virtual processor dimensions, and, more importantly, the same virtual processor

\(^3\)“Twisted Data Layout”, where the tiles of a matrix are skewed, but where there is no skewing within the tiles, is one example of a distribution which seems easier to represent with the “full-dimensional” approach.
coordinates with different implicit semantics, depending on which of \( d_a \) and \( d_d \) is greater. However, since we always know \( d_d \) and \( d_a \), this is not really a source of ambiguity, and it is a very convenient way of reducing the complexity of applying the affine distribution functions. That is why this was the approach adopted for this project.

So, in summary, we will have a virtual processor space of \( d_v = \max(d_a, d_d) \) dimensions.

### 4.5.3 The Function

The affine function maps data coordinates onto virtual processor coordinates. Since affine functions always have an identical domain and range (see Section 3.11), we have to make sure that we augment, if required, \( N \) to \( d_v \). As previously explained, that does not influence the calculation, since all coordinates in the augmented dimensions will be 0. Then, our affine function is

\[
f \left( \begin{array}{c} i_0 \\ \vdots \\ i_{d_v-1} \end{array} \right) = \left( \begin{array}{cccc} a_{00} & \cdots & a_{0(d_v-1)} \\ \vdots & \ddots & \vdots \\ a_{(d_v-1)0} & \cdots & a_{(d_v-1)(d_v-1)} \end{array} \right) \left( \begin{array}{c} i_0 \\ \vdots \\ i_{d_v-1} \end{array} \right) + \left( \begin{array}{c} t_0 \\ \vdots \\ t_{d_v-1} \end{array} \right),
\]

(4.30)

**Remarks**

- This function determines which data dimensions are going to be distributed over what architecture dimensions. It may seem at first sight that this function could map a lower-dimensional data structure into a higher-dimensional one, or vice-versa, however, that is not the case. If \( d_a \) and \( d_d \) do not match, some dimensions will be ‘sparse’, and this function allows us to determine which ones.

- This allows us to describe all invertible regular transformations of a data structure within the same number of dimensions.

- If we thought of all sparse dimensions as being collapsed, this function would allow us to represent all invertible transformations of data structures within the same coordinate system. But — the sparse dimensions do actually add some meaning because they determine the embedding of the data dimensions onto the architecture dimensions.
• The semantics of the matrix $A$ are that it affects fix-point transformations, the fix-
point being 0. Some of the most common are given below for 2 dimensions.

\[
\begin{pmatrix}
1 & 0 \\
0 & 1 \\
\end{pmatrix} \quad \text{Identity} \quad (4.31)
\]

\[
\begin{pmatrix}
0 & 1 \\
1 & 0 \\
\end{pmatrix} \quad \text{Transpose} \quad (4.32)
\]

\[
\begin{pmatrix}
1 & 0 \\
1 & 1 \\
\end{pmatrix} \quad \text{Skew Rows by Columns} \quad (4.33)
\]

\[
\begin{pmatrix}
1 & 1 \\
0 & 1 \\
\end{pmatrix} \quad \text{Skew Columns by Rows} \quad (4.34)
\]

\[
\begin{pmatrix}
-1 & 0 \\
0 & 1 \\
\end{pmatrix} \quad \text{Reflection over x-Axis} \quad (4.35)
\]

\[
\begin{pmatrix}
1 & 0 \\
0 & -1 \\
\end{pmatrix} \quad \text{Reflection over y-Axis} \quad (4.36)
\]

• The semantics of the vector $t$ is rotations. These can happen along any of the dimen-
sions.

4.5.4 Unimodularity

All of the transformations in the examples of the previous section have been unimodular,
i.e. $\det(A) = \pm 1$ (see Section 3.10). This means that the area of objects is preserved. For
data distributions, the semantics of non-unimodular transformations could be one of the
following.

• A data structure is mapped to another, larger, structure that is ‘sparse’\(^4\), i.e. there
  are locations that do not contain any data. Lawrie [51] gives one such transformation,
  and suggests that another data object could be stored in the ‘holes’. However, that
  seems very messy, if it is merely for the purpose of recovering the space! There is a
  role for such transformations, though, if two data structures are to be inter-woven in
  some regular fashion. An example for this would be if we have two $4 \times 4$-matrices $A,
  B$ that we want to distribute in a pattern like

\[
\begin{bmatrix}
a_{00} & a_{01} & b_{00} & b_{01} & a_{02} & b_{02} & a_{03} & b_{03} \\
b_{13} & a_{10} & b_{10} & a_{12} & b_{11} & a_{11} & b_{12} & a_{13} \\
\vdots
\end{bmatrix}
\]

This could be done with the following distribution functions:

\[
f_A(i, j) = \begin{pmatrix} 1 & 0 \\ 1 & 2 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix}
\]

\[
f_B(i, j) = \begin{pmatrix} 1 & 0 \\ 1 & 2 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix}
\]

\(^4\)\text{This is a bit loose: the distinction is not between zero and non-zero elements, but between locations that contain defined data and ones that don’t.}
The other possibility is that several locations of the original data structure are mapped onto one. This clearly has undefined semantics, and will therefore not be allowed, and can also very easily be ruled out by insisting that the coordinates of \( A \) are integer.

For the scope of this project, all transformations have been unimodular.

### 4.5.5 Affine Redistributions

The aim of the lazy libraries approach is to resolve as many data layout conflicts as possible. However, where that is not possible, we need to redistribute data from one distribution to another. Assume that for some data structure, the affine distribution function at some point in the program is \( f(i) = A \vec{r} + \vec{t} \), and that at a later point in the program, the affine distribution with which that same data structure is required is \( g(i) = B \vec{r} + \vec{s} \), then, we call the redistribution function an affine function \( h \) such that \( h(f(x)) = g(x) \). Section 3.11 shows that \( h \) then has the form

\[
h(i) = (BA^{-1}) \vec{r} + (\vec{s} - (BA^{-1}) \vec{t})
\]

So, the redistribution function is actually itself an affine function with the same domain and range as the distribution functions. Furthermore, given any affine distribution function and any affine redistribution function, their composition will always result in a valid affine function, so we have a closure property. This is also important for the derivation of the code to implement distributions from their functions — the problem of distributing data in its initial layout and the problem of redistributing between two distributions are the same.

### 4.5.6 Inverse of Affine Transformations

All affine functions are by definition invertible. As shown in Section 3.11, the inverse \( f^{-1} \) of an affine function \( f \) is

\[
f^{-1}(\vec{v}) = A^{-1} \vec{v} - A^{-1} \vec{t}
\]

The argument is \( \vec{v} \) because what the inverse affine function conceptually does is to map virtual processor coordinates back onto data coordinates.

### 4.5.7 Summary — Properties of Affine Transformations

A comprehensive method has been presented for describing the affine part of regular data distributions. In summary, this method has the following properties.

- Any number of data and architecture dimensions can be handled. Trivial cases are scalars and uniprocessors, respectively. Multidimensional arrays and hypercubes can be handled.

- All invertible linear transformations of data structures can be described. They consist of a fix-point operation and a rotation in any number of directions.

- It can be determined what data dimensions are to be distributed along what architecture dimensions. If the two dimensions are not the same, the affine function determines either along which architecture dimensions we do not distribute data (if the architecture dimensions are greater)\(^5\) or which of the data dimensions are not distributed (if the data dimensions are greater).

\(^5\)In this case, we can copy along those dimensions, see Section 4.7.
- For the purpose of this project, all transformation have been unimodular, but an example has been given to show how non-unimodular distributions can be represented using the same method.
- All transformation functions are invertible.
- Affine redistribution functions have the same form as the transformation functions.
- The composition of any affine transformation function and an affine redistribution function is again a valid affine transformation function, so we have a closure property.

### 4.6 The Folding Function

This section formally defines the folding function that constitutes the second part of the overall data distribution scheme presented in this chapter. In the examples in Section 4.4, the folding functions all took a slightly different form. That was done for the sake of clarity. In this section, we will define one overall folding function that is applicable in all cases.

The task of the folding function formally is to map the $d_v$-dimensional set of virtual processors onto tuples of size $d_a + d_d$, where the first $d_a$ coordinates represent the processor coordinates on the parallel architecture, and the remaining $d_d$ coordinates represent the local data coordinates on the processors. So, a $d_d$-dimensional data structure that is partitioned in some way over a parallel architecture will always be partitioned into $d_d$-dimensional substructures on each processor. Some of the dimensions might obviously be "fully distributed" in the sense that there is only one element of that dimension (such as one row of a matrix) per processor.

#### 4.6.1 Modularity Considerations — The Index Function

As stated in Section 4.5.1, we have the array $N$ that contains the number of data elements along each of the $d_d$ dimensions. If the transformation function "exchanges" some of the dimensions in their mapping onto the virtual processor dimensions, then we have to make sure that the folding function matches up the right element numbers from the array $N$ with the coordinates of the virtual processor space. In other words, we need to access the array $N$ via an indexing function that takes account of any exchanges of dimensions. This has to take the form of a permutation from the set $S_{d_v}$, see Section 3.10. As stated there, each permutation can be uniquely described as a sequence of pairwise exchanges of neighbouring elements. This leads to the algorithm shown in Figure 4.1 for determining the permutation

$$
\text{index} = \left( \begin{array}{c} 1, \ldots, d_v \\ i_1, \ldots, i_{d_v} \end{array} \right),
$$

and we can from now on use $i$ rather than $i$ for correctly accessing the array $N$.

#### 4.6.2 Definition of the Function

Kaushik et al. state in [44] that for a cyclic(b) distribution of a vector over a ring of processors, element $i$ of the vector is mapped to processor $((i \div b) \mod P)$ with local index $((i \div Pb)b + i \mod b)$. This has been used and extended to define the folding functions of
Let \( \text{index} = (i_1, \ldots, i_{d_v}) \) with \( i_i = i \).
for \( i = 0 \) to \( d_v \)
for \( j = 0 \) to \( d_v \)
if \( \det \begin{pmatrix} A_{ii} & A_{ij} \\ A_{ji} & A_{jj} \end{pmatrix} < 0 \)
then swap \( i_i \) and \( i_j \) in \( \text{index} \)

**Figure 4.1:** Algorithm for calculating \( \text{index} \) permutation for correctly accessing the data dimensions array \( N \).

this section. Let \( v \) be some virtual processor coordinate, then we define

\[
\phi_1(v, N, P, b) = ((v \mod N) \div b) \mod P \quad (4.40)
\]

\[
\phi_2(v, N, P, b) = ((v \mod N) \div Pb) + (v \mod N) \mod b \quad (4.41)
\]

\[
\phi_3(v, N) = v \mod N \quad (4.42)
\]

The reason why we have to take the virtual processor coordinates modulo \( N \) first is that unlike Kaushik et al. we cannot assume an “identity alignment”, i.e. in terms of this project, an identity affine function. In a skewing for example, the skewed dimensions need to ‘wrap round’.

Using the above definitions, we can now define our overall folding function \( \phi : V \to P \times N \):

\[
\phi(\bar{b}, \bar{v}) = \begin{pmatrix} p_0 \\ \vdots \\ p_{d_v-1} \\ l_0 \\ \vdots \\ l_{d_a-1} \end{pmatrix}, \quad (4.43)
\]

where

\[
p_i = \phi_1(v_i, N_i, P_i, b_i) \quad \forall i \quad (4.44)
\]

\[
l_i = \begin{cases} \phi_2(v_i, N_i, P_i, b_i), & 0 \leq i \leq (d_d - 1) \\ \phi_3(v_i, N_i, P_i, b_i), & d_d \leq i \leq (d_v - 1) \end{cases} \quad (4.45)
\]

**Remarks**

- Note that we have \( d_d \leq d_v \) and \( d_a \leq d_v \). If \( d_d < d_v \) some of the virtual processor dimensions will be ‘sparse’, so all coordinates in those dimensions will be zero, which means that the physical processor coordinates will also be zero. If \( d_a < d_v \), some virtual processor dimensions will not be distributed, and for those dimensions, \( \phi_3 \) is applied to guarantee wrap-around modulo \( N \).

- The array \( \bar{b} \) is of the same size as \( V \), i.e. \( d_v \), the number of dimensions of the virtual processor space. It contains the blocksizes for the block-cyclic distribution along the
different virtual processor dimensions. The next section (4.6.3) will show how \( \tilde{b} \) can be used to affect identity, blocked, and cyclic distributions.

### 4.6.3 The Blocksize \( \tilde{b} \)

The meaning of the blocksize \( b_i \) along any one dimension \( i \) is that in that dimension, \( b_i \) consecutive elements of the data structure will be allocated to a processor. Setting this parameter allows us to define any block-cyclic distribution in general, and in particular the special cases of identity, blocked, and cyclic(1) distributions. We will now complete the definition of data distributions by showing how we can use \( \tilde{b} \) to affect these three main distribution patterns.

\[
\begin{align*}
    b_i &= N_{i_i} & \text{Identity} & (4.46) \\
    b_i &= \frac{N_{i_i}}{P_i} & \text{Blocked} & (4.47) \\
    b_i &= 1 & \text{Cyclic} & (4.48)
\end{align*}
\]

Notice that we can obviously set different values for \( b_i \) in different dimensions, so as an example, for a Gaussian Elimination algorithm on a mesh, where we want to distribute the matrix rows cyclically and the columns in a blocked fashion, we would set \( b_0 = 1 \) and \( b_1 = \frac{N}{P} \).

### 4.6.4 ‘Inverting’ the Folding Function

The folding function presented above is obviously not directly invertible. However, by using a dot-product with an appropriate vector, we can invert the function “modulo \( N \)” in the sense that we can give an inverse to within modulo \( N \). For unimodular affine functions, that is also sufficient, since for, say, an \( N \)-vector, the virtual-processor space will also have \( N \) ‘dense’ locations. The question of what happens with non-unimodular affine distributions needs further thought.

We give the ‘modulo \( N \)’ inverse functions for the three main distributions, identity, blocked and cyclic. For corresponding pairs \((p_i, l_i)\) from the tuple \( \phi(\tilde{b}, \tilde{v}) \), we have:

For \( b_i = N_{i_i} \) (Identity):
\[
    v_i \equiv \left( \frac{p_i}{l_i} \right)^T \left( \begin{array}{c} N_{i_i} \\ 1 \end{array} \right) \pmod{N_{i_i}}
\]

For \( b_i = \frac{N_{i_i}}{P_i} \) (Blocked):
\[
    v_i \equiv \left( \frac{p_i}{l_i} \right)^T \left( \begin{array}{c} N_{i_i} \\ P_i \\ 1 \end{array} \right) \pmod{N_{i_i}}
\]

For \( b_i = 1 \) (Cyclic):
\[
    v_i \equiv \left( \frac{p_i}{l_i} \right)^T \left( \begin{array}{c} 1 \\ P \end{array} \right) \pmod{N_{i_i}}
\]

### 4.6.5 Summary — Properties of the Folding Function

This section has presented one general folding function that together with the affine transformations from Section 4.5 can be used to describe a very wide range of data distributions. In summary, this function has the following properties.

- Any general block-cyclic distribution can be handled. That includes all the distribution patterns that are available in HPF [77], the common patterns identity, blocked, and cyclic become special instances of the general case.
The possible presence of a negative modulus in the affine function, i.e. an “exchange” of dimensions is taken into account.

The function is invertible ‘modulo N’, which is sufficient when we have unimodular affine transformations.

### 4.7 Copying Data

This section looks at the problem of how to represent the copying of data to more than one location. The type of copying we are interested in here is not the replication of some data structure in several different arbitrary distributions, but rather the somewhat more specific problem of replicating lower-dimensional data structures along all dimensions of a higher-dimensional architecture. Examples would be the replication of a scalar on each processor in some architecture, or the replication of a vector that is blocked over the columns of a mesh on each row, or vice-versa.

A copying of one location to many is not a functional mapping, so rather than trying to use that, one possible functional way of describing a copying operation is to give the inverse of the copy function, i.e. a function that maps all locations that hold a copy of some piece of data onto its owner location. The problem with this is that it isn’t invertible (obviously), and therefore doesn’t tell us how to perform the copy operation.

Section 3.6 has described a method for “inverting the uninvertible”, where, provided that the uninvertible function \( f \) is a homomorphism, we can by means of the homomorphism theorem (3.5) simplify the general problem of finding the set of all elements of the domain that are mapped to each element of the range to the much simpler problem of finding \( \text{Kern } f \) (the set of all elements that are mapped to \( \tilde{0} \)), and then finding a function that will map each element of the range of \( f \) to any one of the elements of the domain that are mapped onto it by \( f \). Before going on to the specific copy function, also recall that Equation 3.25 showed that all homomorphisms \( f : V_n \to V_m \) between additive vector spaces of \( n \) and \( m \) dimensions, respectively, take the form of a multiplication by an \( n \times m \)-matrix.

### 4.7.1 Definition of the Problem

Suppose we have a \( d_d \)-dimensional data structure that is being distributed over a \( d_a \)-dimensional parallel architecture. Then, the type of copying described in this section arises if \( d_d < d_a \). The affine function determines over which of the \( d_a \) dimensions of the architecture the \( d_d \) data dimensions are being distributed. For the remaining \( d_a - d_d \) dimensions, we will have only one element of the data structure in that dimension, i.e. all coordinates will be zero. Recall that the array \( N \) holds the number of elements of the data structure in each dimension, and that \( V \) holds the number of virtual processors in each dimension. If \( d_d < d_a \), we have \( d_v = d_a \), so \( V \) is ‘sparse’ and we have \( d_v - d_d \) elements \( V_i = 1 \). If for those dimensions, \( P_i > 1 \), we want to be able to describe in a rigorous way whether and how data is to be copied along those dimensions.

### 4.7.2 The Inverse Copy Function

We can now give the ‘inverse copy function’, i.e. the function that maps all locations that hold a copy of some piece of data onto its owner location. This function has to go from the processor space to the virtual processor space. These have the same dimension, but
the virtual processor space is ‘sparse’. Let $\vec{0}_{d_a}$ denote an all-zero column vector of length $d_a$, and let $e^i_{d_a}$ denote a basis column vector of length $d_a$ with a 1 at position $i$ and zeros elsewhere. Then, the copy function has the form

$$c^{-1}(\vec{p}) = (c_1|\ldots|c_{d_a})\vec{p},$$

(4.52)

where $(c_i)$ is a $d_a \times d_a$ matrix of column vectors, and

$$c_i = \begin{cases} 
\vec{0}_{d_a} & \text{if we want to copy along dimension } i \\
e^i_{d_a} & \text{otherwise}
\end{cases}.$$  

(4.53)

So, if for example we wanted to copy a vector that is blocked across the columns of a mesh on each row, we would have $(c_i) = \left( \begin{array}{c} 0 \\ 0 \\ 0 \\ 1 \end{array} \right)$.

### 4.7.3 Calculating the Forward Copy Operation

Having stated the inverse copy function, we will now look at how Section 3.6 can be used to calculate the set of all processors we need to send data to in order to implement the copy operation. The first thing we need to do is calculate $\text{Kern } c^{-1}$, the set of all processors that are mapped to the virtual processor $\vec{0}$. This is fairly easy:

$$\text{Kern } c^{-1} = \{ \vec{p} \mid p_i = 0 \text{ if } c_i = e^i_{d_a} \}.$$  

(4.54)

So, for the example of $(c_i) = \left( \begin{array}{c} 0 \\ 0 \\ 0 \\ 1 \end{array} \right)$, we would have $\text{Kern } c^{-1} = \{ \vec{p} \mid p_0 = 0 \}$, i.e. all processors on column 0.

The only other thing we need now is to have a function that will map the processor we are copying from to any one of the processors we want to copy to. The easiest solution for that is simply the identity function $c(\vec{p}) = I\vec{p} = \vec{p}$, since the processor we are copying from obviously keeps its copy of the data!

Now, for each processor $\vec{p}$, the set of processors to copy data to simply is $\vec{p} + \text{Kern } c^{-1}$. Note that this can be calculated simply by addition. For the above example of copying the vector which is blocked across the columns of Row 0 onto every row, the processors we need to send data to from each processor $\vec{p}$ on row 0 is $(\vec{p} + \text{Column0})$, i.e. the column of $\vec{p}$, which is what we would expect.

### 4.7.4 Summary

- We have a very neat method for representing the copying of data over unused dimensions of a higher-dimensional architecture, that gives a homomorphic inverse copying function and then allows us to calculate the set of all processors that data need to be sent to in order to implement the copy operation by solving just one equation and then performing an addition for all processors we are copying from.

- This looks conceptually quite similar to wildcard matching, but is more rigorous, and simpler to solve, since by insisting on the homomorphism property, we reduce the complexity of calculating the set of processors to send data to to solving just one equation.

- The reason why this works is the homomorphism theorem (Section 3.5).
4.8 Deriving the Code for Data Distributions

Part of the aim of defining data distributions as invertible functional mappings, and also part of the requirements for a data distribution method, was that it should be possible to derive the implementation code from the functional representation.

Recall that Section 4.5.5 showed that the affine redistributions and distributions do in fact have the same format. Because of that, the problem of implementing distributions and redistributions is essentially the same. What we do to achieve a distribution of some data that is being held on the controller processor is to carry out a redistribution from an identity distribution to the distribution we want.

There are then two problems that we need to solve: (1) creating a data structure with a certain distribution. That could either initially be an identity distribution on the controller process, or if it is to be used for some distributed result, it could be a partitioned distribution straightaway. (2) redistributing a data structure from one distribution to another. These will be addressed in the following two subsections.

4.8.1 Creating a Data Structure with a Specific Distribution

This is the easier of the two problems, and it has been fully implemented. What we have to do is to calculate how many elements of the data structure each cell holds. We can then allocate distributed space for the structure, and can hold on each cell a pointer to the structure and the information about how many elements that cell holds in each dimension. The pseudocode algorithm is given in Figure 4.2. The result of that algorithm is an array $E$ of dimension $d_d$ for each processor that contains the number of elements in all dimensions.

4.8.2 Creating the Code for Redistributions

This section looks at how we can derive the code for implementing data distributions from the distribution functions. Since redistributions and distributions have essentially the same form, we only need to implement redistributions. Data structures that are created with an identity distribution on the controller process (e.g. for input from file) can then be partitioned to their desired distribution by redistributing from an identity distribution. Section 4.5.5 shows how to calculate the affine redistribution function from two given affine functions.

The method for implementing redistributions presented here is not claimed to be optimal, however, it is \textit{contention-free}, and, because of that, we can accurately calculate its cost, rather than having to rely on prediction by some estimate. The idea is to implement redistribution as a sequence of operations, each of which is known to be contention-free, and is implemented as some well-known communication primitive for which we can give accurate cost models for a given architecture.

For two quite complicated distributions, the corresponding redistribution may be quite a long series of steps, however, it is not very likely that we would in practice carry out such redistributions. It is far more likely that any redistributions that occur in practice actually correspond to some ‘atomic’ step.

\textbf{The Algorithm}

For ease of notation, we will omit vector arrows.
On processor $p_i$  
Let $E$ be a $d_d$-dimensional array  
\( \forall i, \; 0 \leq i \leq d_d \)  
do  
\[ \text{if } b_i = N_{i_i} \]  
\[ \text{let } c_0 = f(\vec{0}) \]  
\[ \text{let } c_l = f(\vec{0} \uparrow i \rightarrow N_{i_i}) \]  
\[ \text{if } c_0 \geq c_l \]  
\[ \text{if } p_i = \phi(c_0) \]  
\[ E_i = N_{i_i} - (t_{i_i} \mod N_{i_i}) \]  
\[ \text{else if } p_i = \phi(c_l) \]  
\[ E_i = t_{i_i} \mod N_{i_i} \]  
\[ \text{else} \]  
\[ E_i = 0 \]  
\[ \text{else } (c_0 < c_l) \]  
\[ \text{if } p_i = \phi(c_0) \]  
\[ E_i = N_{i_i} - (t_{i_i} \mod (N_{i_i} - 1)) \]  
\[ \text{else if } p_i = \phi(c_l) \]  
\[ E_i = t_{i_i} \mod (N_{i_i} - 1) \]  
\[ \text{else} \]  
\[ E_i = 0 \]  
\[ \text{else } (b_i = \frac{N_{i_i}}{P_i} \text{ or } b_i = 1) \]  
\[ E_i = \frac{N_{i_i}}{P_i} \]  

Where $\vec{0} \uparrow i \rightarrow x$ denotes the vector $\vec{0}$ with element $i$ changed to $x$.

Figure 4.2: Algorithm for calculating the number of elements of a data structure in each dimension that are held on each processor under an affine distribution $f(\vec{i}) = A\vec{i} + \vec{l}$ with parameter $\vec{b}$ for the folding function $\phi$. 
1. Let \( f(i) = Ai + t \) with block-cyclic parameter \( b \) be the starting distribution and \( \tilde{f}(i) = \tilde{A}i + \tilde{t} \) with block-cyclic parameter \( \tilde{b} \) be the target distribution.

2. If \( b \neq \tilde{b} \), then:
   - if \( f \) is not an identity-affine function, i.e. \( A = I \) and \( t = 0 \), then we first redistribute to an identity affine mapping. Call algorithm recursively with target distribution \( f(i) = Ii + 0 \) and block-cyclic parameter \( b \). Now assume that \( A = I \) and \( t = 0 \).
     - If \( b = N_i \) and \( \tilde{b} = \frac{N_i}{P_i} \),
       do a scatter with block-size \( \frac{N_i}{P_i} \) and stride \( \frac{N_i}{P_i} \).
     - If \( b = Ni \) and \( \tilde{b} = 1 \),
       do a scatter with block-size 1 and stride \( P \).
     - If \( (b = \frac{N_i}{P_i} \text{ or } b = 1) \) and \( \tilde{b} = N_i \),
       do a gather with same parameters as above.
     - If \( b = \frac{N_i}{P_i} \) and \( \tilde{b} = 1 \) or vice-versa,
       do an all-to-all broadcast.

3. Else if \( b = \tilde{b} \), calculate the affine redistribution function. Let \( R \) be the matrix and \( r \) the offset vector of that function. Then perform the following steps for all dimensions \( i \) of the data structure.
   - If for all \( j, 0 \leq j \leq d_d \) and \( j \neq i, R_{ij} \neq 0 \), perform a skewing by \( R_{ij} \) of dimension \( i \) by \( j \). Notice that a skewing is a set of concurrent rotations.
   - Rotate dimension \( i \) by \( r_i \).

Remarks
The above is an outline algorithm that looks quite complex. Once in practice some parameters become fixed, it simplifies considerably. So, if we know that we are dealing with, say, a mesh, the skewing step simplifies to inspecting elements \( R_{01} \) and \( R_{10} \). Notice that a matrix transpose would be implemented as a sequence of two skewings. All the operations mentioned can be implemented as single-step communication operations in MPI. The advantage of this is that we can rely on the efficient and consistent implementations of these operations in MPI, and it allows us to make accurate cost calculations.

4.8.3 Summary
A method for deriving the code to implement data distributions has been presented. This requires two algorithms, one for creating data structures with a given distribution, and one for carrying out redistributions.

- The algorithm for creating new data structures is efficient.

- The redistribution algorithm is not necessarily optimal, but is contention-free and therefore allows accurate cost calculation. It directly translates to implementation in MPI.

- There is published work on efficient implementations of runtime array redistributions [77, 44]. This means that there is a good case that what should be done here is to derive from the mathematical models what redistributions have to be done, and not
how to do them efficiently. Thakur et al. [77] for example provide efficient primitive redistribution routines that can either be compiled-in by parallelising compilers or that can be used independently. Routines like this could be plugged into the redistribution algorithm stated above to make it efficient.

4.9 Related Work

This section discusses related work. Section 2.2 gave a general review of literature on data distributions. Here, we will focus in a bit more detail on how data distributions are represented, whether the approach of this project can be used to represent the same distributions, and whether the representations used fulfill the requirements for representing data distributions as outlined in the introduction to this chapter.

In HPF, data distribution is done by means of explicit alignment and distribution directives given by the user. This two-phase approach corresponds to the concept of a composition of an affine and a folding function in this project, and all distribution patterns available to HPF users can be represented by the method developed in this chapter. There has been a lot of interest of how the HPF user-directives can be automated, since by having to determine the data distributions, the user really takes the main decisions in determining how to parallelise a program.

4.9.1 Automatic Alignment

Li and Chen [56] discuss how alignment can be automated. Their approach is based on finding a set of alignment functions for all arrays used in a computation that embed the arrays into a single index domain, i.e. their access pattern from loop counters etc. This domain is then used to determine a partitioning of all data structures that minimises data movement.

The paper allows for literally all possible alignment patterns, in that the alignment functions include shifts, reflections, embeddings, and permutations. A permutation of an index domain 1...N of some array can be any re-ordering of those numbers. For N elements, there are N! different possible permutations. The general problem of index domain alignment as formulated in this paper is therefore not very surprisingly shown to be NP-complete. However, Li and Chen give a heuristic algorithm which is shown to give good results.

The approach of this project is much more restrictive in that only affine transformations are allowed. However, since the library algorithms that are to be implemented will normally have a regular access pattern, it seems that the benefits of having a much more tractable problem, and an invertible transformation function (permutations can only be inverted by enumeration) should outweigh the disadvantage of having a more narrowly defined problem. However, the problem of finding a uniform index domain and using that for optimising partitionings is essentially the same as the problem of minimising data movement by choosing compatible data distributions, which is the aim of this project.

4.9.2 Automatic Partitioning

Gupta and Banerjee [32] deal with the related problem of automatic partitioning. This paper is based on determining partitioning constraints for data structures, from which a consistent picture of the data distribution scheme required is built. The first major difference between
this and other works is that constraints are accumulated over a whole program and not just over single loop nests.

There are two types of constraints, parallelisation constraints and communication constraints. In that terminology, this project only deals with communication constraints, since the library functions represent parallel algorithms, not programs from which we need to extract parallelism.

Gupta and Banerjee resolve the issue of conflicting constraints by attaching them with weights and deciding in favour of the ‘heaviest’. There are no redistributions. That is in contrast to the approach taken here, where it may not always be possible to find compatible distributions, in which case we have to redistribute.

Two other relevant points are these. Firstly, optimisation is done with respect to execution time, not just data movement. That is an important issue that will have to be included in the discussion of optimisations in Chapter 6, since the parallel library algorithms may perform differently with different layouts. Secondly, data copying is, in line with the approach of this chapter, allowed only in architecture dimensions where a wildcard matching can be given, see Section 4.7.

4.9.3 Runtime Redistribution Techniques

Thakur et al. [77] and Kaushik et al. [44] are two papers that discuss efficient techniques for runtime redistribution of arrays. The main relevance to this chapter here is that the existence of publications like this means that there is a good case that in deriving the code for implementing data distributions (Section 4.8), what should be done is to determine what redistributions have to be performed, and not how they can be implemented efficiently.

Thakur et al. describe efficient algorithms for redistributing between different distributions available to the user in HPF. That includes redistributions from cyclic(1) to blocked, which has so far here been represented somewhat crudely as an all-to-all broadcast. For future work, it should be investigated how the routines from this paper could be used to make the redistribution algorithm efficient.

Kaushik et al. is relevant in two ways. Firstly, they optimise redistributions between different types of block-cyclic distributions by trading off sending fewer messages for sending larger messages. Since the communication startup time in wormhole routing is typically two orders of magnitude larger than the per-byte message transfer time, this makes good sense for messages up to a certain size. Secondly, Kaushik et al. represent data distributions as factorisations of tensor bases. This should definitely be looked at in more detail, as suggested in Section 3.13. However, it seem that this method defines distributions, but does not give a simple way of calculating the index mappings in the inverse direction.

4.9.4 Array Alignment in an Array Processor

Lawrie [51] describes the design of a memory system for an array processor, i.e., a collection of $N$ ALUs that can operate concurrently on an $N$-vector. For this to work efficiently, the memory system needs to be able to supply concurrently $N$ data elements. Lawrie shows that except for trivial cases, that can only be done if there are more than $N$ independent memory units.

The other issue that is relevant to this project is that if different ‘slices’ of a multidimensional array are to be accessed (i.e., if there is parallelism in more than one dimension), the array needs to be distributed over the memory units in such a way that all slices go
`across’ the memory units, rather than residing on just one. Various types of skewings are suggested for resolving these conflicts. All the skewings given could be represented with the method described in this chapter. However, Lawrie’s skewings are not unimodular. To avoid inefficient memory use, it is suggested that other data could be stored in the ‘holes’. That seems a bit messy ... In some situations, however, we might want to deliberately inter-weave data structures. An example of this is given in Section 4.5.4.

### 4.9.5 “Twisted Data Layout”

Tatsuya Shindo et al. [72] describe “Twisted Data Layout” as a data distribution technique for resolving conflicts among optimal layouts between different parts of a program. Coming from automatic parallelisation, their approach is a two-stage process, as follows. The first phase determines the optimal data layout for maximising parallelism in each individual loop nest. A global optimisation phase then attempts to resolve conflicts among the distributions selected in the first phase. In the form presented in [72], twisted data layout is brought about by mapping a two-dimensional array firstly in a conventional blocked or cyclic way onto a mesh of virtual processors. Those virtual processor tiles are then mapped onto a ring of physical processors in a skewed fashion so that parallelism in each dimension of the array can be exploited. The tiles themselves are not skewed.

Shindo et al. declare twisted data layout by HPF directives, as shown in Figure 4.3. They formally define it by a set of pairs of virtual processors \( \bar{v} \) and physical processors \( p \), where \( \bar{v} \) is mapped onto \( p \), \( V \) is the set of virtual and \( P \) the set of physical processors, and \( m \) is the rank of a virtual processor and \( n \) the number of physical processors, as follows.

\[
T = \left\{ (\bar{v}, p) \mid \bar{v} \in V, p \in P, \left( \sum_{1 \leq k \leq m} v_k \right) \mod n = p \right\}
\]  

(4.55)

A ‘subscript translation method’ is given that translates the subscripts of the original array into processor number and local subscript of the distributed array. An inverse of that mapping is not given. Since the background of this paper is automatic parallelisation, a computation-mapping is also given. This works by stripmining the original loops into loops iterating over virtual processors and local indices. Bounds and stride of these loops are calculated in differing ways depending on whether the original data layout was blocked or cyclic, which shows that there is no general method for doing this.

### 4.9.6 Critique and comparison with this project’s approach

There is no description of an algorithm that spots conflicts in optimal data layout and makes decisions about whether they should be resolved by twisted data layout. It seems
that in practice the user has to direct the compiler to use twisted layout. Also, in the form presented in the paper, the application of twisted layout is limited to situations where parallelism is to be extracted from both dimensions of a two-dimensional array mapped onto a ring. If a mesh was used, this issue would not arise. There should still be good reasons for using this layout on a mesh, but that is not discussed. Performance figures are given from the ELMHES subroutine from the Eispack library [75, 1972]. These look at speedup for different data layout strategies (without base-line performance figures!) with increasing number of processors. What would be interesting is to see the performance of TDL with increasing array sizes, i.e. — as the cost of computation begins to outweigh the cost of possible redistributions, is there a point where TDL becomes suboptimal (e.g. because data are no longer consecutive)? Furthermore, the ELMHES (calculate upper Hessenberg form) subroutine is mainly used in conjunction with other routines, e.g. when preceded by one and followed by four other Eispack routines, it can be used to calculate eigenvalues of a general real matrix. It would be interesting look at performance figures for the full sequence of computations.

The data distribution method described in this chapter can be used to describe twisted data layout. For simplicity, assume a square $N \times N$ matrix, and let the architecture be a ring of $P$ processors, as in the paper. Then, the affine function for describing twisted data layout is

$$f \left( \begin{array}{c} i \\ j \end{array} \right) = \begin{pmatrix} \frac{N}{P} & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

and the folding function is

$$\phi \left( \begin{array}{c} v_x \\ v_y \end{array} \right) = \begin{pmatrix} v_x \text{ div } \left( \frac{N}{P} \right)^2 \\ v_x \text{ mod } N \\ v_x - v_y \frac{N}{P} \text{ div } \frac{N}{P} \end{pmatrix}.$$  

If, as discussed earlier, we used $4 \times 4$ matrices and 4-vectors in the affine functions, this would be a bit simpler still, because the affine function itself could differentiate between skewing just the tiles, or the tiles and the data within the tiles, which is here done by the rather more complicated folding function. Note that the unimodular matrix is a ‘skewed inverse’.

4.9.7 A Decomposition Method

Anderson and Lam [5] deal with global optimisations for parallelism and locality in a parallelising compiler. The point of interest for this project from that paper is the representation of data decompositions. The first point is that since Anderson and Lam come from automatic parallelisation, there are always two decompositions here — data decomposition and computation decomposition. That raises the complexity of the so-called dynamic decomposition problem considerably, and it is in fact shown to be NP-hard. Heuristics are, however, suggested.
Data decompositions are described by a function that maps array locations to processors, not to processors and local indices, as in this project. The partition is then defined as the kernel of that mapping, i.e. the set of all elements that are mapped to the same processor (strictly speaking, the kernel only defines the set of all elements mapped to 0 ...). The orientation of the distribution is defined as the way in which the axes of the array are mapped onto the processor axes. Finally, there is a displacement.

These notions are all contained in the method described in this chapter. However, the affine functions used here are invertible and therefore do not involve sets (=enumerations).

Finally, Anderson and Lam describe blocked decompositions as the span of a canonical basis for a vector space of the same dimension as the depth of a blocked loop nest. This allows calculating all the iterations that fall into one tile, but is not a mapping, and certainly not an invertible one.

4.9.8 Code Generation

In a related paper to [5], Amarasinghe and Lam [4] describe an algorithm that given a partition for a computation derives an optimised SPMD program to implement that computation.

The optimisation in this paper is based on an accurate data-flow analysis which works by determining not just whether statements refer to the same location, but also whether they refer to the same value. The traditional data dependence approach, which is value-based, is shown to be over-cautious.

Communications are optimised by aggregating messages in various ways. This has an interesting relationship to the approach of this project. Since [4] talks about parallelising compilation, a section of code can be passed and messages then ‘vectorised’. This would not normally work in the libraries approach, since we have no knowledge of the user’s source code. Lazy evaluation is the only way that allows us to consider optimisations like message aggregation.

4.10 Conclusion and Future Work

This chapter has described a comprehensive technique for representing and handling regular data distribution. To my knowledge, there is no published work using this technique as a whole, however, there are a lot of interesting parallels in published work to different separate components of this method.

Summary of Technique

- Data structures and parallel architectures of any dimension can be handled.
- Data distributions are described as a composition of an affine transformation function and a folding function.
- The affine transformation can be used to describe all invertible linear transformations of data structures within the same coordinate system.
- It can be determined which data dimensions are mapped onto and distributed over which dimensions of the parallel architecture. That includes cases where the number of dimensions doesn’t match.
• So far, all transformations have been unimodular, however, an example for a non-unimodular transformation is given.

• The affine transformation functions are invertible.

• Affine redistributions have the same form as distributions, which greatly simplifies their implementation.

• The composition of an affine transformation function and a redistribution is always again a valid affine transformation function, so we have a closure property.

• The folding function allows us to represent any block-cyclic distribution. Identity, blocked and cyclic distributions become instances of the general case. All distribution patterns available in HPF can thus be represented.

• The ‘exchange’ of dimensions by a possible negative modulus for the matrix in the affine function is handled correctly by the folding function.

• The folding function is invertible to within modulo $N$, which is sufficient for at least all unimodular transformations.

• A very neat method for describing data copying has been presented that uses the homomorphism theorem to simplify the problem of calculating the set of all processors to which data have to be sent to solving just one equation and then performing an addition for each sending processor. This is more rigorous and more efficient than wildcard matching. The type of copying covered is the copying of a lower-dimensional data structure along those dimensions of a higher-dimensional parallel architecture across which it is not being distributed.

• Two algorithms are presented for implementing data distributions from their mathematical description, one for creating new structures with a given distribution, and one for performing a redistribution. The former is efficient. The latter is not guaranteed to be optimal, but it does provide contention-free communication, and thus allows to accurately calculate the cost of redistribution. The existence of published work on efficient implementations for redistribution primitives suggests that an algorithm that determines what primitives have to be performed, rather than how to perform them efficiently can be implemented efficiently.

Chapter 7 demonstrates the practical feasibility of these algorithms in an implementation.

Future Work
Various suggestions have already been made along the way how this work could be extended and improved.

• Representation of Distributions
  As suggested in Section 3.14, it may be possible to use tensors to describe the ‘change of coordinate system’ that takes place when data are distributed over a parallel architecture. This would be preferable to the folding function approach, since having a tensor would allow us to calculate the set of coordinates (i.e. indices) in one system from another, in both directions. At the moment, the folding function is only invertible modulo $N$, which is sufficient for unimodular distributions, but not in general.
There are a number of problems, though. The coordinate systems we are dealing with are of different dimensions, which would seem to rule out at least first-order tensors. Also, we really only want to use integers as our scalar multiples in all of this, but sets of $N$ integers can normally only be vector spaces if $N$ is prime.

- **Copying of Data in Distinct Distributions**
  The method for representing copying described here is only applicable to situations like the copying of a vector over all rows of a mesh. It should be investigated whether there can be some efficient, non-enumerating method for describing more general copying, such as the duplication of a matrix in two distinct distributions on a mesh.

- **Implementation of Distributions**
  It should be investigated how the published work on efficient redistribution primitives can be integrated into the redistribution algorithm presented here in order to optimise it.

- **A “Join”**
  A number of important algorithms work on separate rows or columns of a matrix at different times. It would considerably enlarge the number of problems this approach could be applied to if a method was found for deriving one distribution function for the matrix from a “join” of the vector-distribution functions of its rows or columns.

- **Stencil-style distributions**
  The method described in this chapter has so far not included the type of data copying that would happen in stencil computations. It could be argued that this should be done by local communication, and that is indeed an approach found in the literature. However, the data distribution scheme presented in this chapter should be extended to handle stencils. There is published work on how stencil shapes should influence data partitions [66].

I feel that with further work, the method described here could provide a framework for a comprehensive method for handling data distributions.
Chapter 5

Software Design

This chapter describes the design of the software system that lies behind the front-end library functions which are called from user programs. A key aspect of this system, the representation of data distributions, has already been treated in Chapter 4, and there is also a lot of interaction between the material of this chapter and the topic of optimisation, which will be dealt with in Chapter 6.

This chapter is structured as follows. Firstly, Section 5.1 summarises the tasks of the runtime system. Then, Section 5.2 discusses the choice of parallel algorithms and their implementation. This includes performance models for the algorithms. Section 5.3 describes design decisions about the types of parallel architectures and connection topologies to be handled. Section 5.4 explains how the library in practice interacts with user programs. Following that are a number of sections that contain the technical detail of the runtime system: (5.5) data identifiers and the renaming mechanism for honouring anti-dependencies, (5.6) the core 'lazy' machine of the runtime system, (5.7) the format of the DAG, (5.8) a description of what happens when a value is forced, and (5.9) the garbage collection mechanism. Finally, Section 5.10 gives a brief overview of the software structure of the implementation for this project.

5.1 Tasks of the Runtime System

The runtime system is responsible for implementing both the library's "lazy" and its "self-optimising" attributes. As such, the key tasks the runtime system has to perform are as follows.

- To assign data identifiers by which matrices and other data are referred to, and to handle their renaming in a Tomasulo-style algorithm [80] that removes anti- and output-dependencies

- To keep track of the status of data. The status information that is stored about data is whether a distribution has been assigned and if so, the distribution; whether the data are evaluated, and either a recipe for calculating the data, or a pointer to its starting location. Further, as part of the distribution function, we also store the number of dimensions of a data item and its size. Finally, for reasons to do with garbage collection and renaming, we keep a count of the number of references that recipes for other data structures have made to a piece of data.
• To store the DAG representing the sequence of operations that are called from a user program.

• To allocate space for data structures as they are moved around the parallel architecture, and to garbage-collect that space when no longer required.

• When a value is forced, to produce an execution plan for the evaluation of the DAG, and to call the computations and communications required. This includes optimisation for data distribution and for choice of parallel implementation of each operation.

• To keep a "cache" of previous optimisation results.

5.2 Parallel Algorithms and Performance Models

An important issue in the design of the runtime system is the choice of parallel algorithms that are used to implement the library functions. Also, for the purpose of optimisation, we need performance models for the different possible implementations of the algorithms. Therefore, this section will briefly outline different algorithms (full descriptions can be found in, e.g. [49]) and, based on performance-models, give decisions about what algorithms are to be used.

The performance models will be based on the assumption that communication is by wormhole routing, as already outlined in Section 4.3. The parameters in the models are given below, together with approximations for the AP1000\(^1\).

- \(t_h\) is the time it takes to communicate the message header in wormhole routing. Approximate figure for the AP1000 is 0.7\(\mu\)s, but this has been ignored in the models below, as is fairly common practice [49].

- \(t_s\) is the communication startup time. Approximate figure for the AP1000: 240\(\mu\)s.

- \(t_w\) is the per-word message transfer time once the communication has been set up. Approximate figure for the AP1000: 2.8\(\mu\)s.

- \(t_b\) is the time taken for the computation in the innermost loop of an algorithm. In all the algorithms discussed below, all computation is contained in one statement in the innermost loop, but this is obviously not generally true. This parameter is also influenced by the cell cache, and as such depends on blocking, data size, etc. As a very rough guide for the order of magnitude of this parameter, the following are figures for matrix multiplication\(^2\) on the AP1000, where \(K\) is the size of the cache in double words (16) and \(S\) is the size of the matrix blocks held on each cell: 0.5\(\mu\)s if \(S \leq K\), 0.75\(\mu\)s if \(K < S \leq 2K\), and 1\(\mu\)s if \(S > 2K\).

All the models given in this section are combined or derived from models given by Kumar et al. in [49]. Throughout, let \(T_p\) denote parallel execution time, \(P\) the number of processors, and \(N\) the "problem size", so for a vector, the number of elements would be \(N\), and for an \(N \times N\)-matrix, \(N\) would be the problem size (rather than \(N^2\)).

\(^1\)These figures come from a course handout by Dr. Tony Field.

\(^2\)With a very naive ijk-loop, no blocking.
5.2.1 Vector-Vector Dot Product

For vector-vector dot products, we need the two vectors to be distributed in the same way. Any type of blocking (or even cyclic blocking) is acceptable, provided each processor holds the same elements from each vector. Then we have:

For a Ring, with all layouts

\[ T_p \approx \frac{N}{P} t_b + (t_s + t_w) (\log P) \]  \hspace{1cm} (5.1)

For a Mesh, with all layouts

\[ T_p \approx \frac{N}{\sqrt{P}} t_b + (t_s + t_w) \left( \log \sqrt{P} \right) \]  \hspace{1cm} (5.2)

The model for meshes assumes that the vectors are distributed over just one column or row of the mesh, or all columns and all rows. This results in lower performance than for rings. However, since vector dot products often happen in conjunction with matrix operations with much higher complexity, it is often worthwhile to perform dot products on all rows or columns.

5.2.2 Vector-Matrix Product

Here we find that different algorithms have to be used depending on whether the architecture is a ring or a mesh. On a ring, we replicate the vector on each processor, and assign a block of matrix rows (if it is a right-multiply) or columns (for left-multiply) to each processor, which then carries out local dot products. The resulting vector is blocked over the ring.

On a mesh, the straightforward algorithm is to have the matrix blocked over all processors, and to have a copy of the vector blocked over each row (or column, as before) of the mesh. All processors then perform partial local dot-products, and this is followed by a ‘to-all’ reduce (with +) along the rows of the mesh. The result vector is then blocked over the columns, i.e. in transposed distribution to the input vector. If, as very often occurs, we require the result vector with the same distribution as the input vector, we can alter the algorithm as follows. Rather than doing a ‘to all’ reduce, we perform concurrent but separate reduce operations of the rows so that the result vector ends up distributed along the diagonal. From there, we can use a one-to-all broadcast on each column to achieve a blocked distribution over the rows, the same as the input vector. This second algorithm has a higher cost, but slightly less still than performing the normal matrix-vector multiply followed by a transpose.

For a Ring, Layout as described

\[ T_p \approx \frac{N^2}{P} t_b + 2 t_s \sqrt{P} + t_w N \]  \hspace{1cm} (5.3)

For a mesh, with the result vector in transposed distribution

\[ T_p \approx \frac{N^2}{P} t_b + t_s \left( \log \sqrt{P} \right) + t_w \frac{N}{\sqrt{P}} \left( \log \sqrt{P} \right) \]  \hspace{1cm} (5.4)
For a mesh, result vector in same distribution as input vector

\[ T_p \approx \frac{N^2}{P} t_b + t_s (\log P) + t_w \frac{N}{\sqrt{P}} (\log P) \]  \hspace{1cm} (5.5)

So, as can be seen above, the main difference between the two mesh algorithms are the factors for the communication parameters, that contain \( \log P \) rather than \( \log \sqrt{P} \) in the second algorithm.

It can be seen very clearly that the ring algorithm is worse, and progressively so for larger problem sizes. Some calculations for \( N = 500 \) show that the benefit of using a mesh for one vector-matrix multiply far outweighs the disadvantage of performing several dot products with redundant work on just the rows of the mesh.

### 5.2.3 Matrix-Matrix Product

The ring algorithm for multiplying two matrices works by first giving each processor a block of rows of one matrix and a block of columns of the other. Each processor then repeatedly carries out local dot-products, followed by a rotation of one of the two matrices, until it has seen all rows/columns of the matrix being rotated. Depending on which of the matrices we rotate, the result is distributed in normal, or in transposed distribution.

For a mesh, the best algorithm is Cannon’s algorithm (see [49]), which uses a skewed initial layout of both matrices, and then rotates the tiles in their respective rows for one and columns for the other operand. Another mesh algorithm is Fox’s algorithm, but the best that can be said about this is that it asymptotically achieves the same performance as Cannon’s algorithm!

On a Ring

\[ T_p \approx \frac{N^3}{P} t_b + Pt_s + 2N^2 t_w \]  \hspace{1cm} (5.6)

On a Mesh (Cannon’s Algorithm)

\[ T_p \approx \frac{N^3}{P} t_b + 2\sqrt{P} t_s + 4 \frac{N^2}{\sqrt{P}} t_w \]  \hspace{1cm} (5.7)

Clearly, the mesh algorithm is a lot more efficient, and therefore, a mesh should really be used whenever possible if a computation involves matrix multiplication.

### 5.2.4 Additions and Multiplication by Scalars

Additions always require the two operands to be distributed in the same way, and always have the complexity of the number of data elements, i.e. \( O(N) \) for vectors and \( O(N^2) \) for matrices. What the distributions is doesn’t matter. No communication is involved. Likewise, multiplication by scalars can be done in any distribution and without communication, and has the same complexity as addition.
5.3 Parallel Architecture Issues

It was already stated in Section 4.3 that for the purpose of data distributions, it would be assumed that architectures dealt with are either meshes or rings. In fact, the distribution technique easily extends to uniprocessors and hypercubes. In this section, we now have to make practical decisions. The AP1000 is physically a torus mesh, but can be configured as either a mesh or a ring. There are a number of issues in the design of the runtime system to do with the type of architecture used that need to be considered.

5.3.1 How to decide what topology to use

As stated in Section 4.3 the issue of "changing the geometry" will not be considered. Therefore, we can only decide once during a program run what topology to use.

One possible way of doing this would be to wait for the first force, and then inspect the operations that have been called and use the above performance models for taking a decision as to what architecture to use. Care needs to be taken here that this doesn’t become too complex — data distributions will not yet have been optimised, and they cannot be optimised unless the architecture is known, so it would seem that some heuristic should be used. From the above performance models, it is clear that meshes are preferable wherever matrices are involved, and that the complexity of matrix operations means that their contribution to the total cost is usually dominant, so that any vector-operations which would be more efficient on a ring can be ignored. Also, for the AP1000, it seems intuitively a bad idea to disregard the higher connectivity that is physically present in favour of a ring configuration. Therefore, the following heuristic is suggested:

- Use a mesh whenever matrices are involved.

We could then as part of the "optimisations cache" keep a record of whether that initial decision turned out to be a good idea.

The second possibility is to provide a library function that instructs the system what architecture to use. It could be argued that this doesn’t really base too much responsibility for “parallel” programming on the user, since most decisions about the architecture should be quite intuitive, and furthermore, “performance debugging” in this situation is very easy, one simply has to change the instruction.

The approach for this project has been that such a function has been written, and that the library initialisation automatically calls this function for a mesh. This could be over-ridden by the user.

5.3.2 MPI Virtual Topologies

Since the implementation for this project uses MPI, the MPI concept of “virtual topologies” (see Section 2.3) allows using a Cartesian mesh in the runtime system, whatever the physical architecture. This obviously does not create non-existing links, but it does mean that the implementation of the runtime system that was written for this project can be used on any platform that supports the MPI standard.

5.3.3 “No Host”

As already stated in Section 4.3 for the purpose of data distributions, a uniform architecture will be assumed. This means that in AP1000 terms, we are not using the host, which can
be done in MPI by means of the “-noshost” option. As will be detailed in Section 7 there are some problems with this approach, mainly the amount of memory available to the controller processor, which in this case is simply one cell. However, since MPI would also allow using this library on a workstation cluster, which is a uniform architecture, the issue of using the AP1000 front-end machine is really a separate and special issue that will have to be addressed as part of future work.

5.3.4 Design of Runtime System with Host

Before the decision was taken to assume a uniform architecture (because of the data distribution models), different options for the design of the runtime system with host were considered. This involves careful consideration of what information has to be broadcast etc. Although this was not actually used in the implementation of this project, it is felt that the design decision taken there would be usable in an implementation that does use the host, and hence, they are documented for future reference in Appendix A.

5.4 Interaction with User Programs

This section describes how the library is used from a user program in practice. Since the library implementation is based on MPI, this means that the entire program has to be an MPI program, i.e. it has to be compiled and executed like an MPI program. However, the underlying compiler for MPI is *gcc*, and all MPI calls are hidden in the runtime system, so other than the compilation- and execution commands, the user doesn’t have to deal with MPI.

- The interface with user programs is through a conventional include file (*lazy_lib.h*). This is documented in Appendix B.

- All functions start with the prefix *L_* for “lazy”. Before any other function is called, the library needs to be initialised by calling *L_InitialiseLibrary*, and similarly, as the last call, the library needs to be tidied up by calling *L_FinaliseLibrary*.

- Since MPI programs are SPMD (i.e. single) programs, any I/O in the user program needs to be included in conditional statements that check whether the program is running on the controller process. A library function *L_ControllerProcess()* has been provided for that purpose.

- The data structures representing matrices, vectors and lazy scalars are part of the include-file interface, so the user can access their members, in particular, a pointer to the data if they are evaluated. This means that data can be read without performance penalty once evaluated.

- All data structures need to be initialised to inform the runtime system about their size.

- An example user program is given in Appendix C.
5.5 Data Identifiers and Renaming

The runtime system refers to all data structures by identifiers. These are integers that are at the moment simply assigned in ascending order, starting from 0. This means that identifiers for data structures that have been overwritten or deleted are not reused. It is unlikely that this will cause problems, since the number of positive integers available is very large ($2^{15}$). However, if re-use was to become necessary, we could keep a stack of redundant identifiers. This would probably also not be too large, since a program with a lot of re-use would pop the stack frequently.

5.5.1 Overwriting Data

Lazy evaluation means that we cannot simply overwrite data, since we do not know whether a previous call to the old value of a data structure exists that has not yet been forced, i.e. we have to worry about honouring anti- and output-dependencies. In order to solve this, we use a simple renaming algorithm that resembles the Tomasulo method for register-renaming [80]. This was already part of last year's project.

The difference in the implementation of this year's project is that whenever a suspended call to a library function is made, we increase a reference counter in the status information held on both the operands. This allows us to actually overwrite data if that is legal with respect to anti- and output-dependencies.

The use program in Appendix C has been annotated with the identifiers that are actually used. Figure 5.2, which shows the DAG for the same program, has also shows identifiers.

5.5.2 Data Handles in the User Program

In the user program, data are referred to by a handle. This is a structure that mainly just contains the identifier, but also the size of the data object, a flag for whether it is “valid”, i.e. evaluated and located on the controller processor, and a pointer to the start of the actual data, which should only be used if valid is true. The point of this pointer (sorry) is that the user program can access data without performance penalty once evaluated.

The semantics of assigning one handle to another in the user program are that the reference is assigned, not the actual data. The reasoning behind this is that this corresponds to parameter-passing by reference in function calls. Actual replication of data would have to be done by an explicit library function.

- The include file (Appendix B) shows the definition of handles.

The current implementation provides handles for lazy matrices, vectors and scalars. Lazy scalars are necessary in order to allow expressions involving vector dot-products to be suspended.

5.6 The Runtime System

The runtime system is really the core of the “lazy machine” that supports the library. This section will briefly describe that system, mainly with respect to the information that is stored, and in what form. The next two sections then give more detail about the DAG and how a computation is actually performed.
5.6.1 Recipes

When a suspended call to a library function is made, we store a recipe for calculating the data item that is defined by that function call. This recipe is a struct that consists of the operation code (these are defined in a header file containing global definitions), the identifiers of the two operands (where there is only one, we don't assign the second, and it is clear from the opcode that only the first one is of interest), and the distributions in which the two operands are required.

Those distributions are assigned by a call to the actual library routine for the operation which is flagged for assigning distributions, rather than performing a calculation. These routines will on the first, suspended, call assign some standard distribution, which may then later be changed by the optimiser.

5.6.2 Data Record

The information stored for each data item is identical, whether it is a matrix, vector or scalar. This is because the data distributions methodology described in Chapter 4 is generally applicable to data structures of any dimension. Each data record is a struct, containing the following information: the data identifier, the number of dimensions of the structure, the array $N$ (see Section 4.5.1) containing the size of the structure, an array that contains the number of elements held locally on each processor from that data structure, a flag for whether a distribution has yet been assigned, the distribution, a count for the number of references in the DAG to the structure (see Section 5.9), a flag for whether or not the data is evaluated, and depending on that flag, either a recipe, or a pointer to the starting location.

5.6.3 Global Data stored in the Runtime System

The global runtime system data is also held in a struct, but this is mainly for the purpose of clearly identifying data that is part of the runtime system’s ‘lazy machine’. The data stored are: the number of architecture dimensions, the array $P$ (see Section 4.5.1), the overall number of processors, the (integer) rank of the local processor, the coordinates of the local processor within the parallel topology, MPI communicators for the Cartesian topology, and each processor’s row and column in a mesh. Then, the data handler is an indexed array of data records, as outlined above, together with an integer that stores the next identifier to be assigned. As will be described more closely in the next section, this data handler already contains all the DAG information that needs to be stored, and is highly efficient to access because it is an array. The user should have a good idea from his program what a likely maximum number of data structures and results is, and if it exceeds the default, it can be increased easily.

The fact that the data handler is an array indexed by the data identifier is a good argument for implementing a re-use of garbage-collected identifiers, since unused identifiers effectively correspond to empty locations in the data handler. As previously described, this could be done very easily with a (small) stack.

5.7 The DAG

This section now looks at the design of the DAG. As an example, we will look at the program given in Appendix C. A pseudocode form for that program is given in Figure 5.1. The key
Figure 5.1: Pseudocode for the example discussed in this chapter. The DAG for this program is shown in Figure 5.2, and the full code in Appendix C.

points in the design of the DAG are the following.

• **What are nodes and how are they identified?**
  Nodes are either evaluated data structures, or recipes for unevaluated results. They are identified by the identifier for the data structure or result. This view of nodes corresponds to what is found in the literature in several places [27, 59]. A point to note here is that this means that we can only have functional-style *single-output routines*, i.e. operations like LU decomposition would have to be represented by returning one matrix, which is understood to contain the ‘L’ and ‘U’ matrices in the respective parts of the result matrix. This also corresponds to the way such algorithms are commonly implemented in practice.

• **Is there a natural way for defining the weight of a node?**
  Yes, the cost of the computation associated with evaluating the recipe. This is zero if the data is evaluated, and is calculated by means of the performance models given in Section 5.2 if it is not.

• **What are edges and how are they represented?**
  The meaning of edges in this DAG is that the data item represented by the node at the source of an edge is used in the recipe for the result represented by its destination. Since both the source-data-item and the recipe contain a distribution function, we have a possible redistribution associated with each edge.

• **Is there a natural way for defining the weight of an edge?**
  Yes, the cost of the possible redistribution associated with it. As described in Section 4.8, we can derive the code for redistributions from their representation, and this means that we can *calculate* their cost. It might be worth investigating whether there is a quicker method that would give a good enough approximation to the cost, e.g. by eigenvalues.

• **How do we store the DAG?**
  The data handler described above *is* the DAG. Since each value or result has an entry in the data handler, this corresponds to the nodes of the DAG. Then, since each recipe for an unevaluated result stores both the identifiers of the operands and their required distributions, this is all the information we need to store about edges. We
can calculate the redistributions required from the two distributions at either end (see Section 4.5.5).

Figure 5.2 shows the DAG for the example given in Figure 5.1. Note that this DAG is evaluated in a left-branch-first manner, and hence, as indicated, some results will already have been redistributed to the distribution required by the time that another recipe refers to them.

5.8 What happens on a force — Execution

One possible view of what should happen on a force is that the runtime system has to produce an execution plan, consisting of a sequence of global computations and communications. The method adopted for this project is that the DAG is the execution plan, since it contains all required information. It is evaluated recursively, by calling the algorithm shown in Figure 5.3 with the identifier to be forced and an identity distribution that locates the resulting data structure on the controller processor. Note that the optimiser in Chapter 6 will then simply have to alter the DAG in order to implement its optimisations.

5.8.1 Special Situations in Loops

There is a situation that needs special consideration where loops are involved. If a data structure is updated in each iteration of a loop, but not forced until loop exit, we could potentially build up a huge unevaluated expression. The real problem with this is that we need more space for evaluating such an expression than for just storing it, so a simple mechanism that triggers a force if cells threaten to run out of memory is not sufficient, unless the threshold is set at a very low proportion of the total memory available.

So, we would ideally like to detect this sort of pattern by means other than running out of memory. At the same time, we want to wait for sufficient iteration to have full information available for how results from one iteration are used in the next. The following is a suggested method for solving this problem.

- Each time we update a data structure, i.e. where the old identifier of the structure is one of the operands in a recipe, we note this by keeping a count for the number of self-updates for this structure that there have been.

- It has to be considered whether indirect ‘self-updates’ should be looked for, and to what depth.

- If we update a data structure by a recipe containing itself for the third time, we force the value. We have then got a picture of how the results are used in the next iteration by inspecting the ‘middle’ iteration of the three.

An alternative would be to evaluate all suspended expressions on any one force and at the same time always keep an evaluated DAG for another two iterations. This would then eventually provide the same result by means of ‘refining optimisation’.

5.9 Garbage Collection

The renaming mechanism that is used in order to guarantee that output dependencies are honoured has a drawback — data is not normally overwritten, and this means that we need
Figure 5.2: DAG for the example program shown in pseudocode in Figure 5.1 and in full in Appendix C. This shows both the structure of the DAG (nodes are identified by data structures or results of computations, and the use of identifiers and renaming. Note that all operations are functional, i.e. have one result.
force( identifier id distribution d )
if id is evaluated
  if the current distribution of id is d
    return
  else
    redistribute id to d
    return
else
  inspect the operands (op1, d1) and (op2, d2) in the recipe for id
  force(op1, d1)
  force(op2, d2)
  perform calculation in recipe
  redistribute id to d
  return

Figure 5.3: Algorithm for evaluating a DAG.

some kind of garbage collection mechanism.

- **What data can we garbage-collect?**
  Data that has been overwritten from the user program’s point of view, and that is no longer required in any suspended expressions, and, secondly, data the handles of which have left scope in the user program and can therefore no longer be referred to.

- **Mechanism implemented in this project**
  The mechanism implemented in this project deals with the first type of garbage collection mentioned above, i.e. overwritten data. We store with each data record a count for the number of references to it that have been made by recipes for other results. When a recipe is evaluated, we decrease that reference count accordingly. If a data structure is overwritten, and its reference count is zero, we deallocate the previous data for that structure.

  This mechanism will handle all garbage collection that can be done as long as the user program has one level of scope for data handles.

- **Possible extensions**
  The C++ constructor / destructor mechanism could be used to detect handles leaving scope, and the destructor could be defined in such a way that it clears up all those structures that have no outstanding references to them. However, this wouldn’t be quite sufficient, since it is possible that there are suspended calls to data the handles for which have left scope, and we would need an additional mechanism to detect this.

The garbage collection mechanism described above has been implemented and tested on the example program shown in Appendix C.

## 5.10 Implementation Overview

This section briefly outlines the implementation for this project. The structure of the software system is shown in Figure 5.4. What follows is a description of the various components.
- **glob_def.h**
  This contains global definitions of constants, including the operation codes used in recipes, constants for the number of dimensions of specific architectures and data structures, communication tags, and error codes.

- **matutil**
  This module implements mathematical utility functions, such as calculating the determinant of a $2 \times 2$-matrix.

- **distrib**
  This module contains the implementation for the data distributions method described in Chapter 4.

- **runtimes**
  The main runtime system, as described in this chapter.

- **lazy_lib**
  The actual lazy library. Calls from this get translated into calls to the runtime system, there are typically about one to three runtime system calls per library function.

---

Figure 5.4: Structure of the software implementation.
5.11 Summary

This chapter has given a description of the design of the runtime system for the lazy library. Points dealt with were the tasks of the system, the choice of parallel algorithms by performance models, the parallel topologies handled, the interaction of the library with user programs, data identifiers and the renaming mechanism, the information stored in the runtime system, the design of the DAG and of how it is executed, a garbage collection mechanism, and a description of the structure of the implementation of this project.
Chapter 6

Optimisation

This chapter deals with optimisations that we can perform once a DAG has been accumu-
lated and a value is forced. There is a large body of published work on related topics in this
field (see Section 2.2). A very common source of restrictions on the type of problems han-
dled in such publications is the lack of full data-dependence information at compile-time.
In certain situations, e.g. where sparse data structures are involved, it may also be the case
that even full data-dependence information is not all that is required to guarantee good op-
timisation. Lazy evaluation means that this project could provide a very good background
for implementing optimisations, since there are no limitations on the information available.

In this chapter, two types of optimisation will be discussed. Firstly, having a full DAG
of operations available may allow certain algebraic optimisations. Secondly, we can optimise
data distributions. This will be the main focus of attention in this chapter. The algorithm
discussed here is quite closely based on an algorithm for automatic distribution suggested
by Feautrier [23] in the context of automatic parallelisation.

6.1 Algebraic Optimisations

Last year’s project [30] proposed a number of algebraic optimisations, see Section 2.1. Of
these, only redistribution of product sums has been considered further. What we need
to do here is to inspect the DAG for a pattern that could be redistributed. There is
published work on how the equivalence of expressions can be tested for efficiently (e.g. [29]).
However, in order to implement a reasonably simple algorithm, all we really have to do is
check the operands of sums, and see whether they contain the same identifier in the same
position. Since matrix multiplication is not commutative, we don’t normally have to check
for equivalent expressions with respect to commutativity.

6.2 Description of a published algorithm

The main work of this chapter is a suggested algorithm for optimising data distributions.
This has to a large extent been adapted by a paper entitled “Towards Automatic Distribu-
tion” by Paul Feautrier [23]. This section describes the method of that paper.

It should be noted that the paper deals with automatic parallelisation, so is very much
based on individual loop statements, iterations, and array elements. A lot of introductory
points need to be considered in order to understand the algorithm described.
1. "Ether" communication model.
This model has been assumed throughout. It is based on uniform access times over
the network. The cost for accessing a data element on the local processor is assumed
to be zero, and the cost of access to a remote element is assumed to be a high constant.

2. Restrictions on source programs
The method described by Feautrier only works if a detailed data-flow-analysis of the
source program is possible. This is guaranteed by demanding that program statements
are restricted to assignments and "do" loops, data are restricted to scalars and arrays,
and all array subscripts and loop bounds are affine functions of surrounding loop
counters. Feautrier suggests that programs that do not conform to this pattern might
be analysable at runtime.

3. The Data Flow Graph
A data flow graph is built up the nodes of which are operations (pairs of program
statements and iteration vectors). Quite a large amount of work then involves com-
puting for each scalar or array element referenced in an operation its source (the
operation where it was computed).

4. Edges in the DFG
Edges in the data flow graph occur where a value computed in one operation is used
in another. Each edge has a source, a sink (destination), a governing predicate, which
is a conditional on the iteration vector for whether the data flow actually occurs, and
a sink-to-source-transformation which computes the source-iteration-vector from the
sink.

5. The Schedule
From the DFG described above, we can then construct a schedule. This is a function
that assigns each operation in the DFG to the earliest possible time when it can be
executed. This has to honour the so-called 'causality condition', i.e. all its operands
have to have been computed earlier. Finding causal schedules has received much
attention.

6. Fronts
From the schedule, we can then calculate the set of all operations scheduled at some
time \( t \), known as the front \( F(t) \) at time \( t \). There cannot be any data dependencies
or data transfers within \( F(t) \). Therefore, the operations within one front can be
distributed arbitrarily over all available processors.

7. Placement Functions
A placement function assigns to all operations the processor on which they are exe-
cuted. If two operations are connected in the DFG, a communication will be necessary
unless the placement function assigns them to the same processor. Most of the time,
demanding this as a condition for a placement function, i.e., that all operations with
edges between them are assigned to the same processor, will result in the so-called
trivial placement function, where all operations are assigned to the same processor.

8. The Optimisation Problem
In order to avoid a trivial placement function, what we would like to do is calculate
for possible placement functions the set of all operations that do satisfy the condition
that if they are connected by an edge, they are assigned to the same processor, and then to select the placement function that maximises this set, subject to the constraint that it is not trivial.

The problem mentioned above is, however, a very complex one. So, instead, Feautrier proposes to solve a very similar problem in an algorithm that has the complexity of Gaussian elimination. This works by classifying edges in the DFG into cut and uncut edges. Cut edges are the ones that do satisfy the placement requirement (for no communication) outlined above. We then try to maximise the set of cut edges.

The next problem to be addressed is one very similar to a problem that had to be faced in this project in Chapter 4. Namely, we would like to express the placement functions for operations $x$ is equational form, and that is only easy if the placement functions are affine. However, Feautrier shows that an affine placement function would normally have a greater range than there are processors, hence, the placement function $\Pi$ is expressed as the composition of two functions: $\Pi = \chi \circ \pi$, where $\pi$ is affine. This is exactly analogous the the affine-plus-folding-function idea of Chapter 4. Feautrier calls the affine function also the virtual function, because it maps onto virtual processors. The problem solved by the algorithm below addresses virtual placement.

The edge-cutting condition for each edge can be expressed as an equation (the difference between the placement of the source and sink of the edge is zero) for each iteration vector where data travel along the edge. These placement equations can be unified into a system of equations over all iteration vectors for each edge, and in turn over all edges for the entire DFG. This results in a system of linear, homogeneous equations.

This is likely to be quite a large system of equations! Hence, it is proposed not to use an ‘exhaustive search’ but a greedy algorithm. The equations are ordered in decreasing importance, i.e. with decreasing weight for the edges they represent. Then, a process very similar to Gaussian elimination is used: we start with some initial substitution, apply this to the next-most-important equation. If that equation becomes true (meaning that there is no communication in the corresponding edge), restart (there’s nothing to do for that edge). Else, solve the particular equation at hand, update the provisional substitution with the solution to that equation, and then check whether this new substitution results in trivial placement equations. If so, abandon it and proceed to the next edge. Else, restart with the new substitution.

The pruning in this algorithm happens where a possible solution is abandoned if it causes placement functions to become trivial. In an exhaustive search, we would instead have to try whether other parameters can be changed so as not to get a trivial placement function with the substitution we are checking.

Much of the complication in this algorithm comes from having to deal with iteration vectors, which means that each node in the DFG represents several actual computations. As will be shown in the next section, this becomes a lot more simple in the context of this project.

### 6.3 An optimisation algorithm

This section now proposes an optimisation algorithm for data distributions in this project. The idea for this algorithm came from the paper just described, however, there are significant differences. We will proceed much along the same lines as the previous section.
1. **Communication model**

   We cannot assume uniform access times, since we are not dealing with the sending of individual array elements, but with data structures that may be of hugely varying size. However, we will assume that the *location in the network* with which a communication happens doesn’t matter. For wormhole routing systems, this is approximately correct.

2. **The DAG**

   None of the restrictions that were outlined in the last section need apply here since they were mostly for the purpose of deriving an exact DFG. We have *got* the DAG!

3. **Weights in the DAG**

   We have to assign weights to both the edges and nodes of the DAG. As outlined in Section 5.7, the weight for edges is the cost of redistribution that may have to take place. Since we have the code for redistribution, we can calculate that cost. More difficult is the cost for nodes, since this depends on parameters like the cache miss-rate, and that is not straightforward to estimate [50]. However, the performance models given in Section 5.2 are suggested as a starting point.

4. **Scheduling**

   Likewise, fronts, scheduling, etc., need not be considered here, because the nodes in our DAG are parallel algorithms, not sequential operations that need to be scheduled in parallel.

5. **The Problem**

   So, the optimisation problem we have to solve is to obtain a DAG with a minimum ‘spanning walk’, i.e. where the sum of all edges and nodes, when we traverse in the way stated in Section 5.8.

   Similarly to the optimisation stated for last section, this is quite complex. Therefore, the following algorithm is proposed for solving a partial, but quite similar problem.

   1. When the DAG is first built up, we select the distributions that will minimise the computational cost of nodes.
   2. Express each edge in the form of an equation representing the difference between the affine distribution functions.
   3. Order the equations by weight of the corresponding redistribution.
   4. Starting from the ‘heaviest’ edge, request the next possible distribution of operands. Propagate this through the system of equations.
   5. There is no triviality test, instead, we check the weight of the DAG. If the substitution has led to a higher weight, we abandon it, else, proceed to the next edge.

**Summary**

The above algorithm deals with the affine distribution functions, rather than the folding functions, in order to enable us to have a system of linear equations in the above algorithm. This means that what we can hope for from this algorithm is that it will resolve conflicts in the area of affine layout, not the folding functions. However, the types of conflicts that we might hope to resolve are more likely to be transposes etc. than block-versus-cyclic conflicts.
in the first place. The above algorithm should therefore be able to resolve the larger part of conflicts that arise in most linear algebra problems.
Chapter 7

Performance Evaluation

The lazy library system described in this report is supported by an implementation. This chapter describes preliminary experimental results that were obtained with this implementation. All plots are collated in a section at the end of this chapter in two-column format. This chapter has two main sections. Section 7.1 describes the experiments that were carried out and discusses the results and Section 7.2 briefly discusses another example (BiCGSTAB) that was studied in a pencil-and-paper fashion. Finally, Section 7.3 contains the graphs.

7.1 Experiments and Discussion

The example program that was studied is the example for which the DAG was shown in Figure 5.2, and the code for which can be seen in Appendix C. The implementation that is available and working so far includes the lazy runtime machine supporting the library, and the full area of data distributions, including the algorithms for deriving the code from the mathematical representation of distributions, but it does not implement the optimisation algorithm shown in Chapter 6. Nevertheless, even without optimisation, performance benefits are shown to exist, just by means of the lazy data movement mechanism. To demonstrate this, all the experiments compare two programs:

- The “lazy” program shown in Appendix C, where the DAG is built up and then forced when the final scalar result is printed.

- A “manual force” version where each result is immediately forced by inserting a specific force-statement.

A problem that became apparent when doing the experiments is that with not using the host (in order to maintain a uniform architecture), the memory available on the controller is somewhat limited. With hindsight, it would have been better to take an example where more vectors and maybe only one matrix is involved. Because of this memory-limitation, some of the experiments carried out where done in such a form that the input (random) data are generated in distributed form, rather than on the controller. This can be done very easily (by changing one line that determines the distribution in the random-data-generating-procedure), thanks to the general nature of the algorithm for creating new data structures. Three types of experiments, then, were carried out:

- Generate input data on controller. Vary the problem size with the number of processors constant at 1, 4, 16 and 64.
• The same as above, but generate data in distributed form. This allows us to go up to a higher problem size (4096 with 64 processors).

• Generate data in distributed form, and vary the number of processors with constant problem size. Normally, this wouldn’t allow for very big problem sizes since fewer numbers of processors wouldn’t have enough memory, so for bigger problem sizes, only upwards of 4, or 16, processors have been considered.

**What these experiments show**

1. That the lazy runtime system works.

2. That the algorithms for deriving the code for implementing distributions from their mathematical representation (see Section 4.8) work.

3. That the Tomasulo-style renaming mechanism works.

4. That the garbage collection mechanism (see Section 5.9) works.

5. That the ‘lazy’ overhead is small where there are no benefits from lazy evaluation, i.e. on one processor. See Figure 7.1.

6. That there are already very real performance benefits from ‘lazy data movement’, i.e. the fact that we only return those intermediate results to the host that are required there, and don’t redistribute anything that isn’t further referenced. See in particular Figures 7.2, 7.5 and 7.6.

**What these experiments don’t show**

1. Optimisations.

2. Caching of previous optimisation results, and other more exotic ideas.

### 7.2 BiCGSTAB

Another example that was considered in the BiCGSTAB iterative solver method [7, 82]. This is used by members of the Department of Aeronautics at Imperial College for analyses of how soundwaves behave when hitting an underwater “plate” (possibly even of elliptic shape) [42]. What immediately prevented comparing performance figures for their solver with one implemented via this library is that they use complex numbers, and that hasn’t been implemented for this library yet. However, it can be argued from the plots shown in the following section that once this is done, there should be real performance benefits, even just by virtue of lazy data movement, since their current system distributed data and returns them to the host for each call.

There is a layout conflict in BiCGSTAB [7], so once the optimiser has been implemented on top of the current lazy system, more performance benefits should be expected.

Studying iterative methods in general, it became clear that not all of them decompose into matrix-vector operations. Rather, in some cases, a number of steps are expressed as vector-vector operations where the vectors are individual rows or columns of the matrix involved. This will be addressed in Chapter 8.
7.3 Plots

Figure 7.1: Data generated in distributed form. This shows the (small) overhead of lazy evaluation.

Figure 7.2: Data generated in distributed form. Lazy versus non-lazy evaluation over different problem sizes on four processors.

Figure 7.3: Data generated in distributed form. With no cost for initial distribution of data, this is very much dominated by computation, and hence shows fairly small benefit from lazy evaluation.

Figure 7.4: Same as previous figure, but data generated on controller. The freak measurement for the borderline case of having one element per processor is persistent, i.e. can be reproduced. It is probably caused by the lazy overhead dominating computation with a problem size of one element per processor.
Figure 7.5: Data generated in distributed form, varying problem sizes (up to 4096!) on 64 processors. This shows quite nice and constant benefit for using lazy data movement.

Figure 7.6: Same as previous figure, but data generated on controller. This allows only smaller problem sizes, but gives better speedup, and is also the more realistic situation.

Figure 7.7: Note the logarithmic scale! Data in problem size 512 generated in distributed form, over varying numbers of processors. As the number of elements per processor decreases, communication overhead begins to dominate.

Figure 7.8: Problem size 1024, data generated in distributed form. This cannot be run on one processor, because there is not enough memory. Again, note the logarithmic scale!

Figure 7.9: Problem size 2048. This only works on 16 processors or more. Data generated in distributed form.
Chapter 8

Conclusion

This thesis has described a way of incrementally adding parallelism to an existing serial program by executing certain computationally expensive operations in parallel. The main contributions of this project to that overall methodology have been as follows:

- The development of a technique for representing data distributions that allows representing a wide range of regular distributions, including all those available in HPF.

- Key characteristics of this method are that we can calculate redistributions between two given distributions, and new distributions that are generated by the composition of an old distribution and a redistribution. Further, algorithms have been presented that derive the code for implementing the distributions from their representations.

- A very neat and efficient way of describing the copying of data along the dimensions of an architecture where it is not being distributed is described.

- A summary of the data distributions method developed can be found in Section 4.10.

- The design of the lazy runtime system, including storage of the DAG, its execution, and a garbage collection mechanism are described.

- An optimisation algorithm is suggested.

- A supporting implementation demonstrates the practical working of the data distributions method and the runtime system, and shows some performance benefits from lazy data movement.

It is felt that the work presented here is a useful framework for a system that has potential to give very real performance benefits to users by enabling them to use parallelism in a simple and incremental way.

8.1 Further Work

There is a lot of scope for expanding the work described here. Each of the suggestions made below would probably require quite a lot of further thought, but would increase both the performance benefit offered by this library and its potential relevance to other fields.
• **Representation of Distributions**

The possibility of finding a more mathematically rigorous way of representing the ‘change of coordinate system’ that underlies the distribution of a data structure from one onto many processors should be looked at. Suggestions were made in Section 3.14 for how tensors might be used here.

• **Concept of a ‘Partition’ and ‘Join’**

There are a number of difficulties with the current implementation that point towards the need for representing not just distributions of matrices and vectors separately, but also for having a theory that combines vector-distribution functions for separate rows/columns of a matrix into a distribution for the whole matrix: (1) This corresponds to the skeletons concept of ‘partition’. (2) It seems that a lot of iterative methods are based on operations that are performed on the rows of the matrix involved. (3) The BLAS library provides a three-level system of routines, where BLAS-2 routines can be applied to separate rows/columns of a matrix.

So, it would seem that for making this library usable for a wider range of applications, it should maybe have the same type of three-level structure as BLAS, but with the difference that there is a rigorous mathematical way of representing the interaction of routines from different levels. This is probably the most challenging of the suggestions made here.

• **Comparison with a known benchmark**

Implementing the previous suggestion would allow the performance of the library to be measured against a benchmark suite for iterative methods [21].

• **Optimisations**

The current optimisation algorithm should be implemented on top of the existing runtime system, its performance evaluated, and other algorithms studied.

• **Lazy Evaluation in Loops**

The issue of what happens when a data structure is updated in each iteration of a loop, but not forced till loop exit needs to be looked at further, in particular, the problem of how to trade off the memory for storage of a DAG (and intermediate results that cannot be garbage-collected) against the memory required for evaluating it.

• **Caching of Optimisation Results**

The area of how we could ‘cache’ previous optimisation results has not yet been studied.

• **Partial Evaluation**

Consol [15, 16] suggests ‘parameterised partial evaluation’ as a technique for applying partial evaluation not just where a program has two input parameters, and we partially evaluate with respect to one of them being static, but also for other ‘things’, rather than just input parameters, being static. In the context of this library, the structure of the user program would be a ‘static parameter’ with respect to which we could try to partially apply the library, so that we end up doing optimisations once per program, rather than each time it is executed.

This would be a very interesting way of moving the optimisation stage ‘back to compile-time’, with the partial evaluator acting as an unusually aggressive compiler.
Appendix A

Runtime System With Host

Before the design decision was taken that a uniform architecture was to be assumed, which means that in AP1000 terms, the host is not used and the controller processor is Cell 0, the issue of the design of the runtime system with host was considered. This appendix documents that design for future reference.

A.1 Location of the Runtime System

This section will outline three possible options for where the runtime system should reside and evaluate these options with respect to the following criteria.

1. What computations have to be done and where? How advantageous is it to do that? Note that the ‘computations’ we’re concerned with here are purely the tasks of the runtime system, e.g., optimisation, not the actual numerical computations implemented by the library.

2. What information has to be communicated from where to where? How expensive is that?

A.1.1 Option 1 — Host

The entire runtime system resides on the host. Only the actual numerical library operations are executed on the parallel architecture. This implies the following.

- The DAG is built up and stored on the host.
- On a force, the optimised execution plan is derived on the host.
- The following information is then broadcast from the host to the parallel architecture — the initial data with some information on the initial alignment (if not already stored on the parallel architecture) and the execution plan.
- The host then waits for the result to be sent back. Note that since the data handler resides on the host, it will know in what alignment the result will arrive.
A.1.2 Option 2 — Host plus One Cell

Do most of the work of the runtime system on the parallel architecture. The idea is to avoid superfluous host-cell communications. But does it work? The ‘one cell’ will be called Cell 0.

- The DAG is built up on the host, as in Option 1. Alternatively, we could broadcast information to Cell 0 on each call of a library function. Since the latency of host-cell communications is very high, even though the bandwidth is higher than for cell-cell communications, this seems a bad idea and won’t be discussed any further.

- On a *force*, the DAG is sent to Cell 0. Cell 0 then derives the optimised execution plan.

- If all the initial data already resides on the parallel architecture, the execution plan is then simply broadcast from Cell 0 to all other cells.

If all the initial data is not already on the parallel architecture, Cell 0 has to first send a request for the required data, together with the required initial alignment to the host, then broadcast the execution plan plus information on what initial data to receive from the host and in what alignment to all other cells, and finally, all cells then receive their initial data from the host.

- When the execution is finished, a signal to receive the result, together with information on how it will be aligned, has to be sent to the host.

A.1.3 Option 3 — Host plus All Cells

Use not just one, but all cells. The idea here is to avoid the broadcast of the execution plan from Cell 0 to the other cells, and also to possibly exploit the full power of the parallel architecture for optimisation. Only the differences to Option 2 are listed.

- The DAG is broadcast to *all* cells from the host.

- Optimisation and derivation of the execution plan is done *in parallel*. This would naturally involve some inter-cell communication for the parallel optimisation algorithm. An alternative would be to simply let the optimisation algorithm from Option 2 run on all cells, which should all find the same solution in the same time.

- There is no need to broadcast the execution plan to all cells — they all have it. However, if data is required from the host, Cell 0 would still have to send the same request as in Option 2.

A.1.4 Evaluation

The DAG should be a smaller packet of information than the execution plan. This means that under Option 2, compared to Option 1, less information is sent from the host to Cell 0. However, the latency of host-cell communications is very high. Therefore, the additional host-cell communication required in Option 2 if some data still needs to be fetched from the host would certainly nullify this advantage. Also, Option 2 requires an additional cell-host communication to inform the host about the result. Option 3 is by far the most complicated if a parallel optimisation algorithm is used, but might offer a considerable speedup on the optimisation phase. These considerations lead to the following conclusions.
- Option 1 is better than Option 2 if some of the initial data needs to be fetched from the host.

- Option 2 is marginally better than Option 1 if all the initial data already resides on the parallel architecture.

- Option 3 is the most complex, but might be worth considering for large optimisation problems.

★ Design Decision.
Because of the relative ease of implementing Option 1 (everything on the host), this will be the initial choice for the purpose of developing the rest of the runtime system, especially the data storage handler and the optimisation algorithm. Figure A.1 shows the adopted solution, Option 1, in a diagram. It should be relatively easy to step from Option 1 to Option 2, and once an implementation is available, this could be done, and some comparisons made. Option 3 could be investigated later — but not before the data storage handler and the optimiser have been designed!

![Diagram of the runtime system](image)

Figure A.1: Design of the runtime system — adopted solution (Option 1).
Appendix B

Lazy Library Include File

/* Header file for Lazy self-optimising parallel matrix library. */
/* Lazy_Lib.h */

#include <glob_def.h>

/* Type definitions */

typedef struct /* Matrix */
{
    int id;     /* Matrix identifier */
    int n_x;    /* Dimensions */
    int n_y;
    int valid;  /* True if evaluated */
    ELEMENT_TYPE *data; /* Pointer to data IF evaluated */
} L_Matrix;

typedef struct /* Vector */
{
    int id;
    int n;
    int valid;
    ELEMENT_TYPE *data;
} L_Vector;

typedef struct /* Lazy Scalar */
{
    int id;
    int valid;
    ELEMENT_TYPE *data;
}
L_Scalar;

/*******************************************************************************************
/* Function headers */
/*******************************************************************************************/

int L_Initialise_Library(int argc, char **argv);
/* Pass arguments to main, don’t worry what they contain. */

int L_Initialise_Matrix(L_Matrix *mat, int n_x, int n_y);
/* Initialise a lazy matrix. 
   Arguments: mat - reference to L_Matrix to be initialised 
   n_x - size in x-dimension (number of rows) 
   n_y - size in y-dimension (number of columns) */

int L_Initialise_Vector(L_Vector *vec, int n);
/* Initialise a lazy vector. 
   Arguments: vec - reference to L_Vector to be initialised 
   n - length of vector */

int L_Initialise_Scalar(L_Scalar *alpha);
/* Initialise a lazy scalar. 
   Argument: alpha - reference to L_Scalar to be initialised. */

void L_Error_Analysis(int err);
/* Prints an error analysis for an error code from the library. 
   Doesn’t print anything if error code suggests no problem. */

int L_Controller_Process(void);
/* Returns 1 if the calling process id the controller process, 0 otherwise. */

int L_Finalise_Library(void);
/* Finalises and clears up lazy library. */

Service Input / Output Functions ****************************

int L_Input_Vector(L_Vector *vec);
/* Reads in a vector from stdin. 
   Argument: vec - reference to L_Vector to be read in. */
int L_Output_Scalar(L_Scalar alpha);
    /* Output a lazy scalar to stdout.
       Argument: alpha - L_Scalar to be output.
    */

void L_Print_Processor_Info(void);
    /* Prints available information about parallel architecture.
    */

void L_Print_Vector(L_Vector *vec);
    /* Prints a vector, no matter what distribution.
    */

void L_Print_Matrix(L_Matrix *mat);
    /* Prints a matrix, no matter what distribution.
    */

void L_Print_Scalar(L_Scalar *alpha);
    /* Prints a vector, no matter what distribution.
    */

void L_Print_Data_Info(void);
    /* Prints Information held by the data handler.
    */

/************ Utility Functions ************/

int L_Random_Matrix(L_Matrix *mat);
    /* Generates a random matrix in the dimensions of mat.
    */

int L_Identity_Matrix(L_Matrix *mat);
    /* Generates the identity matrix in the dimensions of mat.
    */

int L_PositiveDefinite_Matrix(L_Matrix *mat);
    /* Generates a positive definite matrix.
    */

int L_Random_Vector(L_Vector *vec);
    /* Generates a random vector in the dimensions of vec.
    */

int L_Test_Matrix(L_Matrix *mat);
    /* Fills matrix with successive numbers.
    */

int L_Test_Vector(L_Vector *vec);
/* Fills vector with successive numbers. */

/****** Parallel Calculation Functions *************/

int L_VV_Add(L_Vector u, L_Vector v, L_Vector *result);
/* Adds two vectors (a and v).
   Postcondition: result contains reference to u + v. */

int L_VV_Dot(L_Vector u, L_Vector v, L_Scalar *result);
/* Calculates dot product of vectors u and v. Reference to result is put in lazy scalar result. */

int L_MV_Multiply(L_Matrix A, L_Vector v, L_Vector *result);
/* Calculates the matrix-vector product A * v, and places reference to result in result. */

int L_MM_Add(L_Matrix A, L_Matrix B, L_Matrix *Result);
/* Adds matrices A and B. */

int L_MM_Multiply(L_Matrix A, L_Matrix B, L_Matrix *Result);
/* Calculates the matrix product of A and B. */

/****** Special 'Lazy' Functions ***********/

int L_M_Force(L_Matrix A);
int L_V_Force(L_Vector v);
int L_S_Force(L_Scalar alpha);
/* These functions do a manual 'force' of a result. */

/****** Timing Functions *******************/

void L_Start_Timer(void);
double L_Return_Time(void);

/******
Appendix C

Example User Program

The % signs on the first two lines are somehow required to pacify \LaTeX, they are not part of the program!

```c
#include <stdio.h>

/* You can #define ELEMENT_TYPE here. The default is double. */
#include <lazy_lib.h> /* ‘Lazy Library’ */

main(int argc, char **argv)
{
  L_Matrix A, B, C; /* Lazy Matrix */
  L_Vector u, v, w, x, y, z; /* Lazy Vector */
  L_Scalar alpha; /* Lazy Scalar */
  
  int err, SZ;
  double time;

  if(argc < 2) {
    printf("Usage: lazy <problemsize>\n");
  } else {
    SZ = atoi(*++argv);
  }

  err = L Initialise_Library(argc, argv);
  L_Error_Analysis(err);

  if(L_Controller_Process())
    L_Print_Processor_Info();

  L_Initialise_Matrix(&A, SZ, SZ); /* A has id 0 */
  L_Initialise_Matrix(&B, SZ, SZ); /* B has id 1 */
  L_Initialise_Matrix(&C, SZ, SZ); /* C has id 2 */
```
L_Initialise_Vector(&u, SZ); /* u has id 3 */
L_Initialise_Vector(&v, SZ); /* v has id 4 */
L_Initialise_Vector(&w, SZ); /* w has id 5 */
L_Initialise_Vector(&x, SZ); /* x has id 6 */
L_Initialise_Vector(&y, SZ); /* y has id 7 */
L_Initialise_Vector(&z, SZ); /* z has id 8 */

L_Initialise_Scalar(&alpha); /* alpha has id 9 */

if(L_Controller_Process())
{
  printf("Generating matrices and vectors on controller.\n");
};

L_Random_Matrix(&A); /* instantiate 0 */
L_Random_Matrix(&B); /* instantiate 1 */
L_Random_Matrix(&C); /* instantiate 2 */
L_Random_Vector(&u); /* instantiate 3 */
L_Random_Vector(&w); /* instantiate 5 */

if(L_Controller_Process())
{
  printf("Doing calculations.\n\n");
};

L_Start_Timer();
L_MV_Multiply(A, u, &v); /* 4 = 0 * 3 */
L_MV_Multiply(B, w, &x); /* 6 = 1 * 5 */
L_MV_Multiply(A, x, &y); /* 7 = 0 * 6 */
L_MV_Multiply(C, x, &z); /* 8 = 2 * 6 */
L_VV_Add(z, y, &w); /* rename 5 to 10. 10 = 8 + 7 */
L_VV_Add(v, x, &u); /* rename 3 to 11. 11 = 4 + 6 */
L_VV_Dot(w, u, &alpha); /* 9 = 10 + 11 */

if(L_Controller_Process())
{
  printf("Dot product alpha is ");
};

L_Output_Scalar(alpha);

if(L_Controller_Process())
{
  printf(".\n\n");
};

time = L_Return_Time();
if(L_Controller_Process())
  printf("\nTime taken: %f seconds.\n", time);

L_Finalise_Library();
return;
}
}
Bibliography

[1] ACM. *APL 90 Conference Proceedings*. ACM, June 1990.

[2] Ajit Agrawal, Guy E. Blelloch, Robert L. Krawitz, and Cynthia A. Phillips. Four vector-matrix primitives. In *Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 292–302, June 1989.

[3] Eugene Albert, Kathleen Knobe, Joan D. Lukas, and Guy L. Steele Jr. Compiling Fortran 8x array features for the connection machine computer system. *ACM SIGPLAN Notices*, 23(9):42–56, September 1988.

[4] Saman P. Amarasinghe and Monica S. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation: Albuquerque, New Mexico, June 23–25, 1993*, volume 28(6) of *ACM SIGPLAN Notices*, pages 126–138, New York, NY, USA, June 1993. ACM Press.

[5] Jennifer M. Anderson and Monica S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation: Albuquerque, New Mexico, June 23–25, 1993*, volume 28(6), pages 112–125, New York, June 1993. ACM Press.

[6] Uptal Banerjee. Unimodular transformations of double loops. Technical Report TR-1036, Center for Supercomputing Research and Development (CSRD), University of Illinois at Urbana-Champaign, 1990.

[7] Richard Barrett, Mike Berry, Tony Chan, Jim Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Chuck Romine, and Henk van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1994.

[8] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. Technical Report CMU/CS-93-112, Carnegie Mellon University, School of Computer Science, February 1993.

[9] Arndt Boede, Mike Reeve, and Gottfried Wolf, editors. *PARLE '93, Parallel Architectures and Languages Europe: Proceedings of the 5th International PARLE Conference, Munich, Germany, June 14–17, 1993*, Berlin, 1993. Springer-Verlag. LNCS 694.
[10] Shahid H. Bokhari. A shortest tree algorithm for optimal assignments across space and time in a distributed processor system. *IEEE Transactions On Software Engineering*, SE-7(6):583–589, November 1981.

[11] Shahid H. Bokhari. Partitioning problems in parallel, pipelined and distributed computing. *IEEE Transactions on Computers*, C-37(1):48–57, January 1988.

[12] Zeki Bozkus, Alok Choudhary, Geoffrey Fox, Tomasz Haupt, and Sanjay Ranka. Fortran 90D/HPF compiler for distributed memory MIMD computers: Design, implementation, and performance results. In *Supercomputing 1993*. IEEE Computer Society Press, 1993.

[13] Siddhartha Chatterjee. Compiling nested data-parallel programs for shared-memory multiprocessors. *ACM Trans. Programming Languages and Systems*, 15(3):400–462, July 1993.

[14] Murray I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. Pitman / MIT Press, 1989.

[15] Charles Consel and Siau Cheng Khoo. Parameterized partial evaluation. Research Report 865, Yale University, New Haven, Connecticut, 1991. Extended version.

[16] Charles Consel and Siau Cheng Khoo. Parameterized partial evaluation. *ACM Transactions on Programming Languages and Systems*, 15(3):463–493, 1993.

[17] Marco Danelutto, Roberto di Meglio, Salvatore Orlando, Susanna Pelagatti, and Marco Vanneschi. A methodology for the development and the support of massively parallel programs. *Future Generation Computer Systems*, 1992.

[18] Marco Danelutto, Roberto di Meglio, Salvatore Orlando, Susanna Pelagatti, and Marco Vanneschi. A methodology for the development and the support of massively parallel programs. In D. B. Skillicorn and D. Talia, editors, *Programming Languages for Parallel Processing*. IEEE Computer Society Press, 1994.

[19] John Darlington, Anthony J. Field, Peter G. Harrison, Paul H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While. Parallel programming using skeleton functions. In Bode et al. [9]. LNCS 694.

[20] John Darlington, Moustafa M. Ghanem, and Hing Wing To. Structured parallel programming. In *Programming Models for Massively Parallel Computers*, pages 160–169, Berlin, September 1993. IEEE Computer Society Press.

[21] Jack J. Dongarra and Henk A. Van der Vorst. Performance of various computers using standard sparse linear equations solving techniques. In Jack J. Dongarra and Wolfgang Gentzsch, editors, *Computer Benchmarks*, pages 177–188, Amsterdam, The Netherlands, 1993. North-Holland.

[22] José Duato and Pedro López. Highly adaptive wormhole routing algorithms for N-dimensional torus. In Hsu et al. [40].

[23] Paul Feautrier. Towards automatic distribution. *Parallel Processing Letters*, 4(3):233–244, 1991.
[24] Ian Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.

[25] K. A. Gallivan, Michael T. Heath, Esmond Ng, James M. Ortega, Barry W. Peyton, R. J. Plemmons, Charles H. Romine, A. H. Sameh, and Robert G. Voigt. *Parallel Algorithms for Matrix Computations*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1990.

[26] K. A. Gallivan, R. J. Plemmons, and A. H. Sameh. Parallel algorithms for dense linear algebra computations. In *Parallel Algorithms for Matrix Computations* [25], pages 1–82.

[27] Apostolos Gerasoulis, Jia Jiao, and Tao Yang. Scheduling of structured and unstructured computation. In Hsu et al. [40].

[28] Leonard Gilman and Allen J. Rose. *APL - an Interactive Approach*. John Wiley & Sons, New York, 3 edition, 1984.

[29] G. Gonnet. Determining equivalence of expressions in random polynomial time. In *Proceedings of the 16th ACM Symposium on the Theory of Computing*, pages 334–341, 1984.

[30] Simon Gouver. Parallel optimising matrix library, July 1995. MEng4 Individual Project No. 402. Department of Computing, Imperial College of Science, Technology and Medicine, London, UK.

[31] Simon Gouver and Paul H. J. Kelly. A lazy, self-optimising parallel matrix library. In *Glasgow Functional Programming Workshop*, Ullapool, July 1995.

[32] Manish Gupta and Prithviraj Banerjee. Demonstration of automatic data partitioning techniques for parallelising compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[33] Peter G. Harrison. A higher-order approach to parallel algorithms. *The Computer Journal*, 35(6):555–566, 1992.

[34] Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991* (Lecture Notes in Computer Science, vol. 523), pages 448–472. ACM, Springer-Verlag, 1991.

[35] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantative Approach*. Morgan Kaufman, San Mateo, California, 1st edition, 1990. Contains hundreds of references.

[36] Jonathan M. D. Hill. *Data-parallel lazy functional programming*. PhD thesis, Queen Mary and Westfield College, Department of Computer Science, Mile End Road, London, E1 4NS, UK, September 1994.

[37] Seema Hiranandani, Ken Kennedy, and Chun-Wen Tseng. Evaluating compiler optimizations for Fortran D. *Journal of Parallel and Distributed Computing*, 21(1):27–45, April 1994.
[38] Roger A. Horn and Charles A. Johnson. *Matrix Analysis*. Cambridge University press, Cambridge, 1985.

[39] Roger A. Horn and Charles A. Johnson. *Topics in Matrix Analysis*. Cambridge University press, Cambridge, 1991.

[40] D. Frank Hsu, Arnold L. Rosenberg, and Dominique Sotteau, editors. *Interconnection Networks and Mapping and Scheduling Parallel Computations*, volume 21 of DIMACS — Series in Discrete Mathematics and Theoretical Computer Science, February 1994.

[41] ICS ’94, *International Conference on Supercomputing, Manchester, England, United Kingdom*, New York, July 11–15 1994. ACM Press.

[42] Richard Jeans. C-SCAT: a robust package for large-scale underwater acoustic analysis. Internal Document of the Department of Aeronautics, Imperial College, London, United Kingdom, 1994.

[43] S. Lennart Johnsson. The connection machine systems CM-5. Technical Report TR-09-93, Harvard University, Computer Science Department, 1993.

[44] S. D. Kaushik, C.-H. Huang, R.W. Johnson, and P. Sadaayappan. An approach to communication-efficient data redistribution. In ICS ’94 [41], pages 364–373.

[45] G. A. P. Kindervater and J. K. Lenstra. Parallel computing in combinatorial optimisation. Review paper.

[46] Kathleen Knobe, Joan D. Lukas, and Guy L. Steele, Jr. Massively parallel data optimization. In *Second Symposium on the Frontiers of Massively Parallel Computation*, pages 551–558, Fairfax, Va., October 1988.

[47] Kathleen Knobe, Joan D. Lukas, and Guy L. Steele, Jr. Massively parallel data optimization. Technical Report CA-8810-1401, Compass, 1988. Also appeared in 2nd Symp. on Frontiers of Massively Parallel Computation, October, 1988.

[48] Kathleen Knobe, Joan D. Lukas, and Guy L. Steele, Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, February 1990.

[49] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing*. Benjamin/Cummings, 1993.

[50] Monica S. Lam, Edward E. Rothenberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Santa Clara, CA, April 1991. Available as ACM SIGPLAN notices volume 29, number 4.

[51] Duncan H. Lawrie. Access and alignment of data in an array processor. *IEEE Trans. on Computers*, C-24(12):1145–1155, December 1975.

[52] Shun-Tak Leung and John Zahorjan. Improving the performance of runtime parallelization. Technical Report TR 92-10-05, Department of Computer Science and Engineering, University of Washington, February 1993. Revised version. Available by ftp from ftp://ftp.cs.dartmouth.edu/pub/CS-techreports/TR94-235.ps.Z.
[53] Shun-Tak Leung and John Zahorjan. Improving the performance of runtime parallelization. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 83–91, May 1993. Also available as SIGPLAN Notices 28(7) July 1993.

[54] Ted G. Lewis and Hesham El-Reewi. *Introduction to Parallel Computing*. Prentice-Hall International Editions, Englewood Cliffs, NJ, 1992.

[55] Jinke Li and Marina Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, July 1991.

[56] Jinke Li and Marina Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, 13(2):213–221, October 1991.

[57] Steven Lucco and Oliver Sharp. Parallel programming with coordination structures. In *Proceedings of the 18th ACM Symposium on Principles of Programming Languages, Orlando, Florida*, pages 197–208, New York, January 1991. ACM Press.

[58] Neil MacDonald, Elspeth Minty, Tim Harding, and Simon Brown. Writing Message-Passing Parallel Programs with MPI. Edinburgh Parallel Computing Centre, The University of Edinburgh. (Course notes for EPCC course on MPI).

[59] Mary E. Mace. *Storage Patterns in Parallel Processing*. Kluwer Academic Press, 1987.

[60] G. M. Megson. Systolic rank updating and the solution of non-linear equations. In V. P. Kumar, editor, *Proceedings of the 5th International Parallel Processing Symposium*, pages 344–351. IEEE Computer Society Press, April-May 1991.

[61] G. M. Megson. Mapping a class of run-time dependencies onto regular arrays. In V. Prasanna, editor, *Proceedings of the 7th International Parallel Processing Symposium*, pages 97–104. IEEE Computer Society Press, April 1993.

[62] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*. University of Tennesee, Knoxville, Tennesee, June 12 1995. Version 1.1.

[63] David M. Nicol, Joel H. Saltz, and James C. Townsend. Delay point schedules for irregular parallel computations. *IEEE Transactions on Computers*, 18(3):69–90, March 1989.

[64] David A. Padua and Michael J. Wolfe. Advanced compiler optimization for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[65] Susanna Pelagatti. *A Methodology for the Development and Support of Massively Parallel Programs*. PhD thesis, Università Delgi Studi Di Pisa, 1993.

[66] Daniel A. Reed, Loyce M. Adams, and Merrell L. Patrick. Stencils and problem partitionings: Their influence on the performance of multiple processor systems. *IEEE Transactions on Computers*, C-36(7):845–858, July 1987.

[67] Fritz Reinhardt and Heinrich Soeder. *dtv-Atlas zur Mathematik, Tafeln und Texte*, volume 1 & 2. Deutscher Taschenbuch Verlag, München, 7th edition, 1987. (In German).
[68] Hermann Rothe. *Einführung in die Tensorrechnung*. Verlag von L.W. Seidel & Sohn, Vienna, Austria, 1924. (In German.)

[69] Joel H. Saltz, H. Berryman, and J. Wu. Multiprocessors and runtime compilation. *Concurrency: Practice and Experience*, 3(6):573–592, 1991.

[70] Joel H. Saltz and Ravi Mirchandaney. The preprocessed Doacross loop. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume II, Software, pages II–174–II–179, Boca Raton, FL, August 1991. CRC Press.

[71] Joel H. Saltz, Ravi Mirchandaney, and D. Baxter. Runtime parallelization and scheduling of loops. In *Proceedings of the 1st ACM Symposium on Parallel Algorithms and Architectures*, Santa Fe, NM, pages 303–312, June 1989.

[72] Tatsuya Shindo, Hidetoshi Iwashita, Shaun Kaneshiro, Tsunehisa Doi, and Junichi Hagiwara. Twisted data layout. In ICS ’94 [41], pages 374–381.

[73] James G. Simmonds. *A Brief on Tensor Analysis*. Undergraduate Texts in Mathematics. Springer-Verlag, 1982.

[74] D. B. Skillicorn and W. Cai. A cost calculus for parallel functional programming. Technical Report ISSN-0836-0227-93-348, Department of Computing and Information Science, Queen’s University, Kingston, Canada, March 1993.

[75] B. T. Smith, J. J. Boyle, J. M. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. *Matrix Eigensystems Routines-EISPACK Guide, Lecture Notes in Computer Science*, volume 6. Springer-Verlag, Berlin, second edition, 1976.

[76] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, USA, second edition, 1991.

[77] Rajeev Thakur, Alok Choudhary, and Geoffrey Fox. Runtime array redistribution in HPF programs. Technical Report SCCS–601, Northeast Parallel Architectures Center, Syracuse University, 1994.

[78] E. C. Thompson. Notes on tensor and exterior products in vector spaces. Mathematical Institute, Oxford, 1979. Revised Edition.

[79] Hing Wing To. *Optimising the Parallel Behaviour of Combinations of Program Components*. PhD thesis, Department of Computing, Imperial College, London, U.K., September 1995.

[80] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal Research and Development*, 11(1):25–33, January 1967.

[81] Lothar Tschampel. Materialien für die Leistungskurse in Mathematik. Berlin, Erste Auflage, 1983–1990.

[82] Henk A. Van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of non-symmetric linear systems. *SIAM Journal of Science and Statistical Computing*, 13:631–644, 1992.
[83] Michael E. Wolf and Monica S. Lam. A loop transformation theory and an algorithm to maximize parallelism. In *IEEE Transactions on Parallel and Distributed Systems*. The Stanford SUIF Compiler Group, October 1991.

[84] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. *ACM SIGPLAN Notices*, 26(6):30–44, June 1991.

[85] H. Wunderling and H. Heise, editors. *Schülkes Tafeln. Funktionswerte, Zahlenwerte, Formeln*. B.G. Teubner, Stuttgart, Germany, 55th edition, 1980. (In German).