ISSN 1361 - 6161

THE UNIVERSITY of MANCHESTER

Computer Science
University of Manchester

On the Quest for Perfect Load Balance in Loop-Based Parallel Computations

Rizos Sakellariou

Department of Computer Science
University of Manchester
Technical Report Series
UMCS-98-2-1

On the Quest for Perfect Load Balance in Loop-Based Parallel Computations[1]

Rizos Sakellariou

Department of Computer Science
University of Manchester
Oxford Road, Manchester, UK.
rizos@cs.man.ac.uk

25/03/1998

[1]Awarded the Best PhD Thesis Prize in the Department for 1996-97.

ON THE QUEST FOR PERFECT LOAD BALANCE IN LOOP-BASED PARALLEL COMPUTATIONS

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Science and Engineering

October 1996

By
Rizos Sakellariou
Department of Computer Science

Contents

List of Figures

List of Tables

Abstract

Loop structures are a potentially rich source of parallelism in programs written in a high-level programming language, such as FORTRAN. Parallelisation of loop structures, by assigning and executing different loop iterations to and on each processor of a parallel computer, may lead to dramatic improvements in performance.

Parallelising compilers aim to exploit this potential by converting a sequential program into a semantically equivalent parallel form, by means of a sequence of appropriately selected transformations. In order to achieve this, one necessity is *mapping* schemes which distribute the computational work, embodied in the parallel loop, across the multiple processors as evenly as possible. Ideally, each processor is assigned exactly the same amount of computational work, in which case *perfect load balance* is achieved; otherwise, some *load imbalance* is said to exist.

This thesis investigates the extent to which perfect load balance can be attained when parallelising members of the class of loop nests which contain bounds that are either constant or linear expressions involving the indices of the surrounding loops.

First, an algorithm for counting the number of iterations of a given loop nest is developed. This is capable of handling *symbolic variables*; that is, variables whose value is not known at compile-time. The resulting, possibly symbolic, count can be used to provide estimates for the execution time of the loop nest.

Using this algorithm as a basis for the quantitative evaluation of load imbalance, the main body of the thesis develops a compile-time load balancing strategy for mapping members of this class of loop nests. This strategy associates an appropriate mapping scheme with each loop nest depending on the amount of computational work contained within it. At the heart of the strategy, a connection with an old problem of Number Theory, the Prouhet-Tarry-Escott problem, is established.

Finally, a comparative analysis of related mapping schemes is conducted. Experimental results on a virtual shared memory parallel computer, the KSR1, show that, in many circumstances, the strategy proposed in this thesis achieves better performance.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Στους γονείς μου, Ιωάννη και Ελένη

To my parents, Ioannis and Eleni

Acknowledgements

This thesis could not have been completed without the help of a number of people who made it possible; it is a pleasure to acknowledge them.

My supervisor, Professor John Gurd, has been an invaluable source of continuous support, advice and encouragement. I am particularly grateful to him for his patience and his rigorous attention when writing this thesis.

This work owes much to all past and present members of the Centre for Novel Computing, who were always being prepared to discuss and provide answers to my questions. In particular, I would like to thank my office mates in room 2.126 of the Department of Computer Science. Mike O'Boyle, Gholam Hedayat, Zbigniew Chamski, and the numerous discussions I had with them in the last four years, contributed significantly to the ideas expressed in this thesis; I am grateful to Mike especially for his help, in various ways, during the last months this thesis was being written. Henry Okora Okoyo and Armando Fortuna contributed to the creation of a stimulating environment in which it was a pleasure to work. Special thanks are also due to Elena Stöhr for constructive comments on an earlier draft of this thesis.

During the years that research for this thesis was being undertaken, as well as in the years that led to this stage, there were many people who helped, in their own way, and to whom I am grateful. However, there are two persons whose sacrifices have been by far unparalleled; these are my parents, who, alongside my sister, have always been an inexhaustible source of support. This thesis is dedicated to them.

Finally, I am indebted to the State Scholarships Foundation of Greece (Ίδρυμα Κρατικών Υποτροφιών – I.K.Y.) for providing financial support.

This thesis was set using the LaTeXdocument preparation system. The use of *Mathematica* for carrying out several of the computations presented in the text is also acknowledged.

πολλὰ τὰ δεινὰ κοὐδὲν ἀνθρώπου δεινότερον πέλει·
Σοφοκλῆς, Ἀντιγόνη

many wonders there be, but naught more wondrous than man.
Sophocles, Antigone

Notation

The notation used in the thesis is rather standard; for easy reference, the symbols used are listed below, along with a short explanation:

$\lfloor x \rfloor$	the greatest integer less than or equal to x.
$\lceil x \rceil$	the least integer greater than or equal to x.
$m \mid n$	m divides n, i.e. there exists an integer k such that $n = mk$, m, n integers.
$m \nmid n$	m does not divide n, i.e. there exists no integer k such that $n = mk$, m, n integers.
$\gcd(m, n)$	The Greatest Common Divisor of m and n.
$[a, b]$	All the values of x (real or integer) such that $a \leq x \leq b$.
$\max\limits_{l \leq i \leq u}(x_i)$	The maximum value of x_i for all integer values of i in $[l, u]$.
$sign(x)$	It returns -1, 0, 1, depending on whether x is negative, zero, or positive, respectively.
$\wedge$	Logical *and*.
$\vee$	Logical *or*.
$A \cap B$	Set intersection.
$A \cup B$	Set union.

A number of loop-related terms dominate much of the thesis. Although these have been established in the literature, the reader who is unfamiliar may consult Section 1.4.2.2. Particular mention is made to the notion of a canonical loop nest, introduced in this thesis, which is explained by Definitions 4.1 and 4.2.

Finally, throughout the thesis, the end of the proofs of theorems and lemmata is marked by **QED**, an abbreviation of the Latin phrase *Quod Erat Demonstrandum* (i.e. which was to be proved). The symbol $\square$ is used to mark the end of examples.

Chapter 1

Introduction

1.1 The Advent of Parallel Computing

The concept of parallelism in computing has been around for long time; it even appears that Charles Babbage was aware of the basic advantages parallelism can offer.[1] The first general-purpose electronic digital computer, ENIAC, was built with a highly parallel and highly decentralised architecture. However, it operated as a serial computer since it was realised that this would make it much easier to put problems on it. Contrary to ENIAC's architecture, the first stored-program computers were based on the sequential *von Neumann model* which was destined to dominate the development of computer technology.

In the years that followed, the main efforts to introduce parallelism were directed towards the low level of the machine architectures. At the same time, trying to understand parallel computation, theoretical principles for parallel computing were laid down alongside various approaches to the architectural design of computers capable of supporting high-level parallel programming. However, the starting point for the explosion of interest in parallel computing can be placed around the mid-1970's. On the one hand, the development of VLSI technologies enabled the construction of parallel computers on a wider scale than before; on the other hand, given that these same technologies were approaching physical limitations, it was realised that parallel computing provided an alternative, safe way of continuing to enhance computer performance. Increased performance has been the main driving force for research in parallel computing since then.

Nowadays, parallel computing has proved successful in tackling a number of computationally expensive problems [77]. Its potential has rendered it one of the main vehicles in the current quest for *high performance computing*, which promises to open exciting new possibilities in science and engineering. However, despite its undoubtedly large impact in several scientific disciplines, there is comparatively little use of parallelism in the mainstream software production industry. The most common reason attributed to this slow takeup of parallelism is the lack of adequate software support. What constitutes a good parallel programming methodology may vary from machine to machine and, not infrequently, programmers need to spend much time porting programs from one machine to another.

A solution, which has been put forward in recent years, has been the development of *parallelising compilers*. In a broad sense, the term refers to compilers whose objective is to transform sequential programs to semantically equivalent parallel ones and tailor them to a specific architecture.[2] The process of this transformation, known as *automatic parallelisation*, has been the subject of extensive research. The motivation is obvious: compilers capable of performing this transformation by themselves will emancipate programmers from this time-consuming task, thus permitting faster production of parallel software. Problems drawn from this general field of research are examined in this thesis.

[1] Evidence that the idea was clear at that time stems from parts of an 1842 publication describing the Analytical Engine. The relevant text has been quoted in several places; see [107, p. 8] or [136, p. 46] for instance.

[2] The terms *high performance compilers* [229] and *supercompilers* [225, 242] have also been used. To the extent that these refer to compilers that are well beyond the capabilities of most existing ones, these terms are equally acceptable. However, neither definition necessarily implies a *parallel* computing platform.

1.2 The Problem

Given the current state of the art in parallelising compiler technology, constructing a compiler capable of performing equally well on a wide class of problems and a wide class of architectures appears to be an intractable task. The lack of a satisfactory model of parallel computation, as well as the divergent parallel programming paradigms currently used, constitute major obstacles. Thus, existing parallelising compilers usually target specific architectural paradigms while their internal parallelisation strategy may differ; in some cases, critical decisions may still be left to the programmer.[3]

However, at a high level, it is useful to consider two phases which are at the core of automatic parallelisation. The first phase deals with the *detection* of available parallelism and applies any program transformations which may enable this; the second phase deals with the *mapping* of the detected parallelism onto a specific parallel architecture. In both phases, the compiler has to choose among several available options; for instance, in the first phase, application of a number of different sequences of transformations to exploit parallelism may be possible, while, in the second phase, various ways of mapping may exist. The option that the compiler chooses must be that which leads to higher performance; this implies that a parallelising compiler must be armed with a performance model, and that any optimisation techniques should be applied subject to their predicted impact on performance.

Traditionally, these two phases have been examined independently of one another. The reasons can be attributed mainly to historical factors. Mapping, being a phase directly connected with the exploitation of parallelism at the machine level, has received particular attention since the first parallel computers were built. On the other hand, automatic parallelisation was first centred around detection of parallelism. However, both phases are interdependent and they should be examined under a common framework, namely that of achieving high performance or, in the context of a parallelising compiler, optimising the model that describes performance.

This thesis addresses the issue of mapping *loop structures*. Since the early years of computing, it has been recognised that most of the execution time of a program is spent on loop structures [127]; thus, their parallelisation is expected to lead to high performance gains. In fact, parallel loops account for the greatest percentage of parallelism in scientific programs [180, p. 4]. As a result, significant research effort has concentrated on how to map loop structures onto a parallel computer; this problem is known as *loop scheduling* and a number of different schemes have been proposed to deal with it. Most of the these schemes defer the final decisions from compile-time to run-time in an attempt to profit from the use of information which is not available, or hard to determine, at compile-time; the advantage is a potentially better workload balance amongst processors. However, in the context of automatic parallelisation, not knowing the final mapping decisions (i.e. which processor executes what) at compile-time, may limit the applicability of any performance model and hinder the decision-making process regarding an optimum sequence of transformations.

Among the reasons which traditionally lead to the choice of run-time loop mapping schemes has been that, often, loops do not have a uniform structure; that is, not every loop iteration performs the same amount of work. However, even in this case, the amount of work executed by each iteration may follow a regular pattern, for example it may increase in a linear fashion. The question is whether it is feasible to derive efficient, compile-time mapping techniques for this class of loops. The main bulk of this thesis is devoted to answering this question.

1.3 Solution and Contributions

This thesis advocates the use of symbolic analysis techniques to handle compile-time unknown variables, thus improving the compiler's 'knowledge' of the program. The derived information can be used to provide estimates for program execution performance and, thereby, evaluate the impact of compile-time loop mapping schemes.

An overheads-based model, described in Section 1.4.3.2, is adopted for modelling performance. From

[3] The term *interactive parallelisation* has been used to characterise this process [157].

the classified overheads, *load imbalance*, that is, the overhead caused by poor distribution of the computational work among processors, is the focus of our attention; thus, the ultimate goal of the loop mapping schemes discussed in this thesis is to minimise load imbalance, ideally achieving *perfect load balance*, where the computational work is distributed exactly equally among the parallel processors. At the same time, care is taken to avoid options which may increase other sources of overhead.

The loop mapping schemes described in this thesis are based on generalised loop nests, whose loop bounds are variables whose values are unknown at compile-time. This formulation permits the extraction of widely applicable rules. Based on these rules, it may be possible to add the appropriate statements to transform a source program so that the way that loop iterations are mapped onto processors is known at compile-time. Throughout the thesis, this is illustrated using program fragments written in FORTRAN. However, there is nothing inherent in our approach which obliges us to use FORTRAN: the methods described in the thesis can be applied to any imperative language.

In particular, the thesis makes the following contributions:

- A *methodology* is described for computing symbolically the number of times that specific statements inside the body of a loop nest are executed; this number, expressed as a function of the loop bounds, can be used to provide an estimate for the amount of work present in a loop nest.
- A *strategy* is developed and analysed for mapping, at compile-time, a special class of loop nests defined as *canonical*; these consist of loops whose bounds satisfy certain constraints, and where each iteration may perform a different amount of work. This strategy is based on the minimisation of the overhead due to load imbalance.
- A *methodology* is presented for splitting certain non-canonical loop nests into multiple loop nests each of which can be mapped according to the strategy previously described.
- Finally, an experimental study of the performance of different loop mapping schemes is conducted.

1.4 Basic Concepts

Serving as a preamble to the main core of the thesis, this section provides a brief description of the fundamental concepts of parallel computing, examined especially in the context of the ideas presented in later chapters.

1.4.1 Parallel Computer Architectures

Traditionally, the most widely cited approach of classifying computer architectures has been Flynn's taxonomy, based on the concepts of *instruction stream* and *data stream* [75]. According to this, parallel computers fall into one of two categories: *Single Instruction stream Multiple Data stream* (SIMD) computers, which exploit parallelism by having a number of processors performing simultaneously the same instruction but on different sets of data; and *Multiple Instruction stream Multiple Data stream* (MIMD) computers, in which each processor is capable of performing its own instruction. The former class consists of computers built on the basis of a common control unit for all processors, while, in the latter class, each processor has its own control unit.[4]

However, Flynn's taxonomy cannot be considered adequate for today's parallel computers. The current trend in developing parallel computers seems to be in favour of MIMD rather than SIMD,[5]

[4] For completeness, it must be noted that the remaining two classes of computers according to Flynn's taxonomy are Single Instruction stream Single Data stream (SISD), the class which includes the conventional serial computers, and Multiple Instruction stream Single Data stream (MISD); the latter, potentially also a parallel architecture, is only a hypothetical possibility since it is considered impractical for real applications.

[5] Consider, for instance, Thinking Machines Corporation, one of the leading parallel computer vendors, who, after a tradition of successful SIMD computers, in their last product, CM-5, supported both the SIMD and MIMD modes of operation [104].

but, most important, the key feature in the architectural design is the structural relationship between processing elements and memory. Based on this approach, two major categories of parallel computers result: *distributed memory computers* are parallel computers in which each processing element has its own, locally addressable memory, while *shared memory computers* are parallel computers in which the same, global memory is addressable from any of the processing elements. A third class, known as *virtual shared memory computers*, sits between the previous two: this class contains physically distributed memory computers which embody mechanisms to make the memory appear globally addressable from the user's point of view.

Distributed memory computers are characterised by a high cost for communication between processors. Hence, minimising the amount of communication between them turns out to be a key issue; however, this goal sometimes conflicts with exploitation of parallelism, and a trade-off has to be found. Conversely, in shared memory computers, memory contention becomes a significant source of overhead as the number of processors increases. Thus, distributed memory computers have been considered as more likely to achieve *scalability*, i.e. increasing performance proportionally to any increases in the problem size or the number of processors.

Although it could be argued that the techniques described in this thesis do not target any specific architecture, it is fairer to admit that they are orientated towards shared memory or virtual shared memory computers. A virtual shared memory parallel computer, the KSR1, has also been used for the experimental evaluation; details of its architecture can be found in [123, 174, 191, 193, 239].

1.4.2 Programming Parallel Computers

1.4.2.1 Parallel Programming Languages

Two basic approaches have been considered for programming parallel computers: the first is based on parallel languages which support parallelism as an integral part of their design, while the second is based on the parallelisation of programs written in conventional sequential languages. In the former, a number of languages, based on paradigms which facilitate the exploitation of parallelism, have been developed in recent years; these include single assignment languages, such as Sisal [69], several functional languages [207], and process-based languages, such as occam [111]. However, the scientific user community is unwilling to accept new languages [172, 173], although it has been argued that the latter can satisfy the performance requirements of the users [34]. Thus, most of today's parallel programming relies on enhanced versions of conventional sequential languages, such as FORTRAN, which dominates the field, C [98, 175] or C++ [212].

Two basic types of parallelism can be identified in programs written in conventional languages: *loop parallelism* refers to the parallelism resulting from the execution of different loop iterations on different processors; *task parallelism* refers to the parallelism resulting from executing different parts of the program in parallel. Another type of parallelism, *data parallelism* [1], is targeted primarily at distributed memory architectures; this has received extensive attention recently, mainly because of its potential for scalability. Here, arrays are divided into independent subsets with each subset mapped onto a different processor; parallelism is exploited by having different processors act on different subsets of data. Given that arrays are usually manipulated by loops, in many cases data parallelism turns out to be equivalent to loop parallelism.

In parallel versions of FORTRAN, task and/or loop parallelism can be represented by means of either compiler directives or additional parallel constructs; these are usually added as extensions to FORTRAN 77 [115]. Standardisation efforts have led to Parallel Computing Forum (PCF) FORTRAN, a parallel extension of FORTRAN 77 aimed at shared memory systems [140], and High Performance FORTRAN (HPF), a language for exploiting data parallelism, on both shared and distributed memory computers, which has gained widespread support [103, 131]. HPF is based on FORTRAN 90[6] and FORTRAN D;[7]

[6] FORTRAN 90, which supersedes FORTRAN 77, is now the international standard for FORTRAN [112]; it includes a number of new features, such as operations on arrays, recursion and dynamic space allocation [164]. Apart from HPF, it has formed the basis for other new parallel languages, such as FORTRAN 90 Plus [165].

[7] An extension of FORTRAN 77 to support data parallelism via directives for array distribution, FORTRAN D is the result of the continuing effort of a research group at Rice University, under the leadership of Ken Kennedy, and includes directives

```
DO I=L1,U1
   DO J=L2,U2
      DO K=L3,U3
         (statements.1)
      ENDDO
      DO K=L4,U4
         (statements.2)
      ENDDO
   ENDDO
ENDDO
```

Figure 1.1: A rectangular loop nest of depth 3.

ongoing research aims to make HPF capable of exploiting task parallelism [89].

Throughout this thesis, FORTRAN is used for illustration of the examples presented. Attention is focused on *loop parallelism*, which is denoted by means of the commonly used `DOALL` construct; this indicates that the iterations of a specified `DO` ... `ENDDO` loop can be executed concurrently on different processors.[8] When presenting a transformed code, ready to be compiled onto a parallel machine, directives based on KSR FORTRAN [122] are used; these directives are introduced in Chapter 4.

1.4.2.2 Loops

The syntax of a `DO` ... `ENDDO` loop (henceforth usually referred to as a **single loop** or, simply, a **loop**) is illustrated below:

```
DO loop_index=lower_bound, upper_bound[, stride]
   loop_body
ENDDO
```

This permits the repeated execution of the statements found between the `DO` and `ENDDO` (referred to as the **loop body**) a number of times, or **iterations**, depending on the value of three parameters: the **lower bound**, the **upper bound** (these are collectively termed the **loop bounds**) and the **stride**; if the stride is omitted its value is assumed to be 1. These parameters may be constants, variables whose value is unknown at compile-time, or general arithmetic expressions; unless explicitly mentioned, this thesis considers all three parameters as integer valued.

For each iteration, the **loop index** (typically, an integer variable) has a different value. For the first iteration, its value is equal to the value of the lower bound; for subsequent iterations, the value of the stride is added to the loop index. Whenever the value of the loop index exceeds the upper bound (if the stride is positive), or the upper bound exceeds the value of the loop index (if the stride is negative), the loop terminates. Clearly, if, in the former case, the lower bound exceeds the upper bound, or, in the latter case, the upper bound exceeds the lower bound, the loop body is never executed; it is common to refer to such a loop as an **empty loop**. Although allowed in FORTRAN (albeit a bad programming practice), this thesis assumes that the loop index is not assigned a value in the loop body.

The general term **loop nest** is used to denote a loop whose body contains other loops; a typical loop nest is shown in Figure 1.1. Individual loops in the loop nest are usually specified by their index; **loop** `I` in the loop nest of Figure 1.1, for instance, denotes the **outermost** loop (that is, the loop whose body contains all the remaining loops). However, two loops in the loop nest may make use of the same variable as a loop index when they are at the same **level**, that is, they are surrounded by exactly the same number of loops; this is the case with the two `K` loops shown in Figure 1.1. The level of a loop

to support data distribution [105]. Similar ideas have also been developed in Vienna FORTRAN [40].

[8] Loops of the form `DO WHILE` ... `ENDDO` (a construct, non-existent in standard FORTRAN 77, but which has been included in the specification of FORTRAN 90) are not examined in this thesis; however, the parallelisation of such loops is usually achieved by converting them to a `DO` ... `ENDDO` form [45, 190, 233].

is denoted by the number of its surrounding loops plus one; thus, in the loop nest shown in Figure 1.1, the `I` loop is located at level 1, the `J` loop is located at level 2, and the two `K` loops are both located at level 3. The loop having the maximum level value determines the **depth** of the loop nest. A loop nest of depth m, $m \geq 2$, for which there is only one loop at each level and the body of the loop at level i, $1 \leq i < m$, contains only the loop at level $i+1$ is called a **perfect loop nest**; for example, the loop nest shown in Figure 1.1 would be perfect if one of the `K` loops did not exist. Perfect loop nests of depth two are usually described as **double loops**.

For any particular execution of a statement in a loop nest, the values of the indices of the surrounding loops define an **iteration vector** or **iteration point**; this corresponds to one iteration of the loop nest. The set of all the iteration points defines the **iteration space** of the loop nest; the latter can be thought of as an n-dimensional cartesian space, where n is the number of surrounding loops. Whenever the bounds of all surrounding loops do not depend on a loop index, the iteration space may be regarded as an n-dimensional rectangle. As a result, loop nests whose bounds are independent of the indices are commonly referred to as **rectangular**; for example, the loop nest shown in Figure 1.1 is rectangular.

1.4.3 Performance

1.4.3.1 Performance Measurement

As already mentioned, the main reason for building parallel computers is to achieve higher performance. Evaluating the extent to which this target has been met by a specific program and computer requires measurement of its performance. Over the years, a number of performance metrics have been used, the most common being *elapsed time*, *speed-up*, and *efficiency*.

Elapsed time (also known as *wall clock time*, since it refers to the total running time of the program, as opposed to 'pure' CPU time) is the most obvious way of describing the performance of parallel programs. However, the presentation of performance results has been mostly based on metrics derived from elapsed time.

The speed-up, S_p, of a parallel program running on p processors is defined as the ratio t_s/t_p, where t_s is the time taken to execute the best known sequential version of the program, and t_p is the time taken to execute the program on p processors.[9] When running a program on p processors, an achieved speed-up equal to p is termed *linear*. At one stage, linear speed-up was considered to be an upper bound; thus, when *superlinear speed-ups* (i.e. speed-ups greater than p on p processors) were reported, they triggered a debate on the issue [63, 73, 203]. Superlinear speed-ups are a possible outcome, for instance as a result of gains due to the collectively larger size of cache memory when using multiple processors.

Finally, the efficiency, E_p, of a parallel program running on p processors is defined as the ratio S_p/p, where S_p is the respective speed-up; efficiency is a measure of the cost-effectiveness of the computation.

Several variations of speed-up and efficiency have been proposed [86, 116, 205, 240]; usually, the motivation for their development is clearer presentation of a parallel program's properties, such as scalability [206]. However, all these metrics may be misleading if the way that they have been derived is unclear [51]. To overcome this, a simple approach to illustrate scalability trends is to consider the inverse of the elapsed time, $1/t_p$. However, in this thesis, the main purpose of any experiments conducted is to compare performance; thus, elapsed time is adopted for the presentation of all results.

1.4.3.2 Performance Prediction

A necessary factor for the fast and efficient development of software is to *predict* performance in advance of a complete implementation. In sequential computation, the von Neumann model provides an easy means of abstraction: the execution time of an application is the sum of the execution times of its parts. By decomposing the application into elementary parts (for example, machine instructions), it is possible

[9] This definition of speed-up is the most widely accepted. Its advantage is that it eliminates the possibility of showing performance gains as a result of parallelising a highly parallel but comparatively slow algorithm; its disadvantage is that it may demand extensive practical investigation which may not be feasible. Thus, in practice, t_s may represent the time taken to execute a sequential version of the parallel program (that is, a version which excludes at least all code required for handling the cases where $p > 1$), or even t_1, that is, the time taken to execute the parallel program on a single processor.

to derive an estimate for its performance; for instance, using the 'big O' notation, it is possible to obtain an estimate to within a constant factor [129]. However, this form of abstraction does not apply readily to parallel machines. The execution time is determined by the application's *critical path*, i.e. the longest sequence of instructions executed by the machine. Although critical path analysis can be performed [11], the critical path may be radically affected by machine characteristics, such as the way processors communicate, synchronise or use their memory hierarchies.

Significant research effort has focused on the PRAM (Parallel Random Access Machine) model [117], which is an abstraction model for shared memory multiprocessors. However, the model has been considered unrealistic in terms of physical implementation and inadequate for describing the behaviour of most real parallel computers [97]. Other models have also been suggested [52, 217], but there is not enough evidence to argue whether or not they are more realistic than the PRAM model.

Another common approach to performance modelling centres around prediction of the overheads associated with parallel computing. Assuming that t_o is the time spent on overheads and t_s is the time required to execute the sequential version of a program, then the running time, t_p, of a parallel program on p processors is given by the equation $t_p = t_o + t_s/p$. Historically, this approach goes back to Gene Amdahl, who, in 1967, questioned the viability of parallel computing by computing an upper bound for the speed-up based on the time spent on the execution of the inherently sequential parts of a program as a percentage of the total running time [6]; improved models, also based on the serial and the parallel fractions of an algorithm, are considered in [35, 57, 191]. Worlton has studied a model which incorporates synchronisation overheads [231] (also discussed in [162, pp. 35–36]) and sources of performance loss are classified in [32].

A more elaborate classification of overheads has been described recently by Crovella [49, 50]. Adopting this approach, the following classes of parallel overhead are defined for use in this thesis:

- *Unparallelised code*: this refers to the overhead caused by sections of the program which are (or have to be) executed sequentially.
- *Load imbalance*: this refers to the overhead caused by poor distribution of the parallelised computational work among processors.
- *Communication*: this refers to the overhead which occurs when a processor is waiting for data to be moved from memory; it may be further subdivided into classes depending on whether data are moved from another processor's memory (if such a memory exists), or from a local memory. The communication overhead may eventually decrease, as more processors are used, thus explaining the phenomenon of superlinear speed-up.
- *Synchronisation*: this refers to the overhead caused when a processor is waiting to acquire a lock or at a barrier.
- *Parallel start-up*: this refers to the overhead caused because of any additional computation required for the exploitation of parallelism.

Roughly speaking, the above classes of overhead are orthogonal and additive; thus, the total time spent on overheads, t_o, is given by the sum of the times spent on each class. The definition of a performance model based on this approach requires a cost function, say F, for each source of overhead, which represents an estimate for the time spent as a result of the occurrence of the particular source of overhead. Therefore, assuming the cost functions F_{UC}, F_{LI}, F_C, F_S, F_{PS} for each of the overhead classes described, an estimate for the total overhead, t_o, can be computed using

$$t_o = F_{UC} + F_{LI} + F_C + F_S + F_{PS}. \tag{1.1}$$

Each of the functions F is highly machine-specific. However, there is also an implicit, machine-independent part which represents a quantitative metric for each individual source of overhead; such a metric is expected to be a function of the program size and the number of processors used. These latter metrics, alongside the machine-dependent parameters, may provide a performance estimator; for example, a way of using them for predicting performance on the KSR1 is described in [49].

The ultimate goal of parallelisation techniques is to minimise t_o. However, in practice, it may not be feasible to compute analytically all the quantities involved in (1.1). In some cases, performance estimates can be obtained by progressively evaluating the most dominant source of overhead [167]; for instance, apart from unparallelised code, this may be load imbalance for shared memory machines, or communication for distributed memory machines [234]. However, trade-off situations may arise; reducing one source of overhead may increase another and *vice versa*. Dealing with all the overheads at once is usually intractable; thus, an attempt is made to reduce one source of overhead at a time, avoiding actions for which there is evidence that another source of overhead may increase.

Throughout this thesis, the described techniques aim to reduce the overheads as expressed in (1.1); thus, frequent reference is made to this equation.

1.5 Thesis Overview

This chapter has described the motivation for the study presented in this thesis; it has also provided a brief introduction to some parallel computing concepts. The remainder of the thesis is structured as follows:

Chapter 2 elaborates on the details of the problem examined. It provides an up-to-date review of the transformation techniques used for automatic parallelisation, focusing on loop parallelisation. It shows the effect that some of these transformations may have on loop nests, and emphasises the importance of symbolic techniques for improving the amount of compile-time information.

Chapter 3 presents a framework for computing the number of times that statements inside the body of loop nests are executed. A geometric analogy of this problem is briefly described, and an algebraic approach is chosen for the development of our framework. The latter is based on the description of rules for the evaluation of algebraic summations of polynomials or integer functions.

Chapter 4 considers the issue of mapping loop nests, at compile-time, in such a way that the overhead due to load imbalance is minimised; this is evaluated using the techniques developed in Chapter 3. Emphasis is placed on mapping non-rectangular loop nests; a special class of the latter, termed the canonical loop nests, forms the basis for developing a mapping scheme which, if certain criteria are satisfied, results in perfect load balance.

Chapter 5 reports on experiments that aim to compare the performance of the loop mapping schemes developed in Chapter 4 with others, as well as to establish the appropriateness of the quantitative approaches followed in the previous chapters as a means of justifying the selection of a loop mapping scheme.

Finally, Chapter 6 summarises the work of this thesis, adding a critique, as well as directions for further research.

Chapter 2

Automatic Parallelisation

2.1 Introduction

As explained in the previous chapter, *automatic parallelisation* has been promoted as a means of emancipating programmers from the time-consuming task of transforming existing sequential programs into semantically equivalent parallel ones. No standard strategy can be followed to accomplish this transformation; however, as also suggested in [201], given a sequential program, the programmer (or an automatic parallelisation tool) can parallelise it by adopting one of the following approaches:

- *Loop parallelisation* is based on the exploitation of loop parallelism; single loops or loop nests are inspected and arranged for their parallel execution in such a way that program semantics remain unchanged (that, is the state of the parallel program on exit from the loop or loop nest corresponds to the state of the sequential program). This approach requires an analysis of the program semantics at the loop level to guarantee that parallelisation will not alter them.

- *Algorithm parallelisation* refers to identifying parts of a program fragment (say an algorithm), which can be executed in parallel. Although algorithms may be implemented by loops, this term is reserved for those situations which require an analysis of program semantics beyond the loop level; it may also cover cases where the exploitation of task parallelism is feasible.

- *Program parallelisation* refers to finding parts of the whole program that can be executed in parallel; this approach may coincide with one of the previous two approaches, or both of them, but it may also involve finding parallelism across parts parallelised by either of these two approaches. For the latter, an analysis of the program semantics for the whole program may be required.

- *Problem parallelisation* refers to discovering alternative methods of writing the program which are more amenable to parallelisation. This approach requires knowledge beyond that which can be extracted from the sequential program.[1]

From the programmer's point of view, the choice of an approach depends on his/her knowledge about the program and the requirements of the architecture. Thus, the programmer selects the approach which he/she considers to be the most suitable to the parameters of the problem; this decision may be based on estimates for minimising the overheads, as described in Section 1.4.3.2. The next step is the transformation of the sequential program to a semantically equivalent parallel form, its compilation, execution, and the measurement of its execution time; this procedure may be repeated, for different parallelising strategies, to determine the one that performs best. The last approach, which can be characterised as a *measure-modify* paradigm [142], albeit time-consuming, has been a common practice for programmers in applied parallel computing [167].

[1] Parallel computing has revived interest in techniques which had been considered somewhat inefficient for sequential computers but are more amenable to parallel execution. A typical example is the Jacobi method for the eigenvalue problem [159, Ch. 5].

Most existing parallelising compilers attempt to exploit parallelism by loop parallelisation. The primary reason for this is the low cost/effectiveness ratio of this approach. The other approaches require more sophisticated methods of program analysis; indeed, in the case of problem parallelisation, the compiler is even required to know alternative methods for solving a problem, something infeasible for the current state-of-the-art in compiler technology. Furthermore, since the early years of parallel computing, loop parallelisation has been tried for many systems and results in significant performance gains, whereas other approaches have proved difficult without yielding much benefit.

Applying a strategy based on loop parallelisation, the task of a parallelising compiler, with respect to automatic parallelisation, can be divided into two major phases:

- The *detection of parallelism* phase attempts to discover which loops can potentially be executed in parallel; this process may also involve the use of program restructuring transformations, in order to minimise overheads.
- The *mapping* phase maps the potentially parallel tasks onto the processors of a given parallel architecture.

The mapping phase presupposes the existence of a detection phase. However, the two phases should not be regarded as necessarily independent. Knowledge of the way that the parallelism is going to be mapped can be crucial for the compiler to decide which parallelisation transformations should be applied. On the other hand, the growing popularity of High Performance FORTRAN and the data parallel paradigm, which give priority to the mapping of data onto processors, may change our view for these two phases in the future.

An overview of the techniques used in each phase is given in the following sections.

2.2 Detecting Parallelism

2.2.1 Dependence Analysis

A typical program consists of a sequence of statements. In order to determine whether any two statements can potentially be executed in parallel, a parallelising compiler must check whether there exists any relationship between the two statements which places a constraint on their execution order. The identification of these constraints is known as *dependence analysis.*

Two types of dependence exist:

• *Control dependences* arise as a result of the control structure of the code. Thus, there is a control dependence between two statements S_1 and S_2 when statement S_1 determines whether or not S_2 will be executed. Typically, S_1 is an `IF` statement.

• *Data dependences* arise as a result of the appearance of the same variable in different statements of the program. Depending on which side of assignment statements the variable in question appears, there are three major types of data dependence. Assuming that statement S_1 precedes statement S_2 in the program sequence then:

a) There exists a *data flow dependence* between S_1 and S_2, usually denoted by $S_1 \delta^f S_2$, if a variable which is computed in statement S_1 is read by S_2.

b) There exists a *data anti-dependence* between S_1 and S_2, usually denoted by $S_1 \delta^a S_2$, if a variable which is read by statement S_1 is computed by S_2.

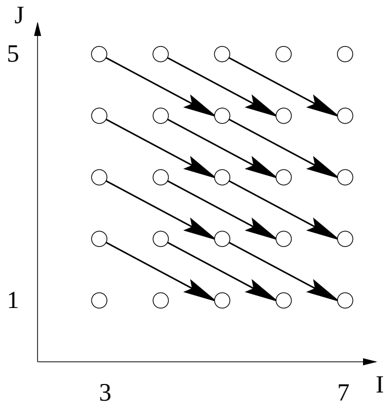
c) There exists a *data output dependence* between S_1 and S_2, usually denoted by $S_1 \delta^o S_2$, if the same variable is computed by both S_1 and S_2.[2]

Data anti-dependence and data output dependence relations are usually caused by program coding practices and are mainly due to storage reuse; it has been shown that they can be removed automatically [5]. Therefore, the only types of dependence that place a constraint on executing two statements in

[2] For completeness, it must be noted that the fourth possible relationship, i.e. when a variable is read by both statements, is referred to as *data input dependence.* Although this does not impose any constraints on the order of execution of the statements, it may be a useful concept for memory management.

```
DO I=3,7
   DO J=1,5
      A(I,J)=A(I-2,J+1)+B(I,J)
   ENDDO
ENDDO
```

a) A loop nest.



b) The iteration dependence graph.

Figure 2.1: An example of a loop and its iteration dependence graph.

parallel are the data flow dependence and the control dependence which, consequently, are referred to as *true dependences*.

Data dependence information is represented by means of a *statement dependence graph*; the nodes of the graph represent statements while an arc between two nodes indicates a dependence. Control dependences can also be represented by transforming them to data dependences. Assuming an `IF` statement at the source of the control dependence, this is achieved by replacing the `IF` with an assignment statement to a boolean variable; then, the latter is added as an operand to all the statements that are control dependent on the `IF`, thus creating a data-flow dependence. This technique is described in [4]. More recently there have been proposals to represent both control and data dependences in a consistent manner using an intermediate language [81, 82]. Another school of research has investigated the benefits resulting from use of the *control flow graph*, i.e. the graph representing control dependences [54, 71].

The identification of data dependences is generally trivial for scalar variables; the real problems arise with the use of array variables. For the latter, the compiler has to determine whether two array references refer to the same element; in other words, whether the values of the subscripts (indices) are the same for both references. In the case where the array references form part of statements which are executed multiple times as a result of a loop structure surrounding them, dependences may exist between different loop iterations; these are called *loop-carried dependences*. The detection of loop-carried dependences determines whether a loop can be executed in parallel by running different iterations on different processors. Therefore, in the context of loops, it is necessary to find any possible dependences between iterations; this information can be represented by means of an *iteration dependence graph*.

To illustrate this, consider the loop structure depicted in Figure 2.1.a. The corresponding iteration dependence graph is given in Figure 2.1.b, and represents the dependence relations between iterations; thus, assuming that the vector (i, j), $3 \leq i \leq 7$, $1 \leq j \leq 5$, denotes the iteration for which $\mathtt{I} = i$ and $\mathtt{J} = j$, iteration $(7, 2)$ depends on iteration $(5, 3)$ which depends on iteration $(3, 4)$. This information is essential to decide which iterations can be executed in parallel; in this example, iterations $(7, 2)$ and $(5, 3)$, for instance, cannot be executed in parallel, but iterations $(7, 2)$ and $(7, 1)$ can.

It is helpful to describe some additional concepts related to data dependence using this example. From the iteration dependence graph it can be observed that all dependent iterations are equidistant from the iteration that creates the dependence; thus, there is a *uniform* data dependence among loop iterations, represented by the *distance vector* D, $D = (2, -1)$ [228]. The vector representing the sign of the elements of the distance vector, i.e. $sign(D) = (1, -1)$, is called the *direction vector*.[3] Direction vectors are useful for computing the *level* of the loop-carried dependences, i.e. the outermost loop which

[3] The function $sign(x)$ returns only three values: 1 for positive x, 0 for $x = 0$, and -1 for negative x. Historically, for direction vectors, the symbols $<$, $=$, $>$ have also been used in place of $1, 0, -1$, respectively [225].

causes, or carries, a data dependence; this is defined by the order of the first nonzero element of the direction vector. Thus, in our example, this value is 1, and therefore dependences are carried by the `I` loop.

In order to find which iterations carry dependences, in this example, one approach would be to remove the `DO` ... `ENDDO` statements by replicating the assignment statement of the loop body a number of times equal to the total number of loop iterations and changing the values of `I` and `J` appropriately;[4] then, based on the definitions of data dependence already presented, we can determine those instances of the loop statement showing a data dependence. However, this approach may be tedious for large loops. To shorten this procedure, it can be observed that a data dependence exists between any two iterations which refer to the same element of the array variable `A`; these iterations, say (i_1, j_1) and (i_2, j_2), can be computed by solving the equations $i_1 = i_2 - 2$ and $j_1 = j_2 + 1$, where $2 \leq i_1, i_2 \leq 6$ and $1 \leq j_1, j_2 \leq 5$.

Assuming that the subscript expressions are linear functions of the loop index variables (the case most commonly encountered in real programs), the general form of this problem is equivalent to finding the solutions of a system of linear diophantine equations[5] under certain constraints; this is known to be an NP-complete problem [83]. As a result, for practical purposes, approximate methods have been developed to check whether there is no solution and, therefore, non-existence of dependence. These methods are usually referred to as *inexact dependence tests*, since they may detect dependences which do not exist (false dependences); this is in contrast to *exact dependence tests*, which detect dependences only if they actually exist.

Historically, the first inexact dependence test was the GCD test, which is based on an elementary theorem of number theory first stated by Gauss: a linear diophantine equation, written in the form

$$\sum_{j=1}^{n} c_j i_j = c_0,$$

where the c_i, $0 \leq i \leq n$, are integer constants and the i_j are unknown integers, has a solution if the greatest common divisor (GCD) of the coefficients c_j of the left-hand side is a divisor of c_0. If the GCD of c_j is not a divisor of c_0 then there is no dependence, however the opposite is not necessarily true. The so-called Banerjee's inequalities [13] provide another means of checking dependence by looking for real solutions. The I-test [132] combines both the previous methods. Other tests, such as the the generalised GCD test [16, pp. 121–122], the λ-test [147] and the power test [226], consider the simultaneous solution of a system of linear diophantine equations. Practical experiences using these tests are described in [83, 155].

An exact method for solving linear diophantine equations with two unknowns has been described by Banerjee in [16, pp. 152–156]. In the more general case, exact methods can be based on integer linear programming techniques [197]. This approach has been followed by Wallace [219], Eisenbeis and Sogno [62] and Pugh [184]. Pugh's method, known as the Omega test, is based on an extension to integer programming of the Fourier-Motzkin elimination method for solving linear inequalities [197, pp. 155–157] (for an implementation of the method in the context of parallelising compilers see [22]). Although the Fourier-Motzkin elimination method has exponential time complexity, Pugh presents evidence that, for the problems encountered in data dependence analysis, the Omega test is fast enough to be suitable for use in parallelising compilers.

Thus far, it has been assumed that the subscript expressions are linear functions of the loop indices. However, nonlinear functions may arise when either some coefficients of the loop index variables in subscript expressions are not known at compile-time, or the subscript expressions are nonlinear functions of the loop index variables. Haghighat and Polychronopoulos [92] provide evidence that the former case may be encountered frequently in certain numerical packages; to deal with it, they propose a symbolic version of the Banerjee's inequalities test. Observations drawn from attempts to parallelise loops from

[4] This transformation is known as loop unrolling [55].

[5] A diophantine equation is a polynomial equation with integer coefficients where integer or rational solutions are sought [163]; a first degree, in terms of the unknown variables, polynomial equation is a linear diophantine equation [126]. The name diophantine stems from Diophantos of Alexandria (circa 3rd or 4th century AD) whose treatise *Arithmetica* is the earliest known work which studied this type of equation in a systematic way [100].

the Perfect Benchmarks[6] [27] led Blume and Eigenmann to the development of a similar dependence test capable of handling symbolic expressions [26, 28]; whenever the latter fails to provide an answer, run-time techniques for data dependence analysis are also considered [24]. A different approach is taken by Maslov [154]; under certain conditions, his algorithm breaks a nonlinear expression into several linear expressions which can be analysed using conventional data dependence tests. Recently, Pugh and Wonnacott have described how to extend the Omega test to handle these situations [186, 187]. In all four cases, the main characteristic is that the compiler is required to perform symbolic analysis, i.e. to handle variables whose values are unknown at compile-time; this issue is discussed again later in this chapter.

More demanding is the development of data dependence tests in the presence of dynamic, pointer-based data structures; for recent results the reader is referred to [109, 178]. Difficulties in data dependence analysis may also arise when the loop body invokes functions or subroutines; this issue, an instance of the more general problem of *interprocedural analysis*, is addressed in [95, 99, 156, 160].

2.2.2 Loop Transformations

Since the early days of compiler technology, the use of program restructuring transformations has been considered a valuable approach for improving the execution time of the compiled program [3, 150]. The main constraint on the application of these transformations is preservation of the program semantics. In the context of parallelising compilers, the primary goal is to reduce the overheads associated with the computations, as described in Equation (1.1). It is beyond the scope of this thesis to present an exhaustive list of program transformations; for comprehensive surveys the reader is referred to [10, 12, 171, 180]. In this section, our purpose is to review transformations applicable in the context of loops (loop transformations), and to show the effects that these have on the loop structure.

One approach to classifying loop transformations is based on the part of the loop structure that they affect; for example, order of iterations, order of statements in the loop body, etc. [68]. In our case, it would be desirable to have a classification based on the class of overhead that is being reduced: for example, transformations for communication, transformations for synchronisation, etc. While there have been studies of transformations in the context of reducing particular overheads [37, 87, 158], the same transformations can be applied to reduce different sources of overheads; therefore, a strictly overhead-based classification would not result in an orthogonal classification scheme. A more general approach has been followed in [12]; based on this, we identify three major classes of transformations:

- *Transformations to enable parallelism* attempt to uncover parallelism by changing the order in which the iterations are executed.
- *Transformations for locality enhancement* are useful in the context of machines with complex memory hierarchy and aim to minimise the cumulative cost of memory accesses.
- *Transformations for breaking data dependences* provide the means to eliminate loop-carried dependences, usually by changing the loop structure.

Using this classification, loop transformations are presented next.

2.2.2.1 Transformations to Enable Parallelism

Loop interchange exchanges the position of two loops in a perfect loop nest [225, Ch. 6]. In many cases it can be used to increase the number of unit stride array references[7] in a loop but it can also be used to enable parallelism or to increase the number of loop iterations that are executed in parallel, thus

[6] The Perfect Benchmarks are a set of 13 FORTRAN programs, having a total of over 60000 lines of source code, which include applications from different fields of engineering and scientific computing; they have been suggested (and, in fact, used extensively) as a means for evaluating supercomputer performance [21].

[7] A unit stride array reference is a reference accessing contiguous array elements in consecutive loop iterations. Increasing the number of such references will usually result in faster execution in the case of computers with cache memories. The latter move blocks of consecutive words from the main memory to the cache; therefore, unit stride references minimise the number of transfers from the main memory to the cache.

```
 DO I=2,N-1
    DO J=2,M-1
       A(I,J)=(A(I-1,J)+A(I,J-1)
&             +A(I+1,J)+A(I,J+1))/4
    ENDDO
 ENDDO
```

a) Original code.

```
 DO I=2,N-1
    DO J=I+2,I+M-1
       A(I,J-I)=(A(I-1,J-I)+A(I,J-I-1)
&             +A(I+1,J-I)+A(I,J-I+1))/4
    ENDDO
 ENDDO
```

b) After applying skewing.

```
 DO J=4,M+N-2
    DOALL I=MAX(2,J-M+1),MIN(N-1,J-2)
          A(I,J)=(A(I-1,J-I)+A(I,J-I-1)+A(I+1,J-I)+A(I,J-I+1))/4
    ENDDO
 ENDDO
```

c) After applying skewing and interchange.

Figure 2.2: Enabling parallelism using loop skewing and loop interchange.

improving parallel performance [180, pp. 23–24]. Care must be taken when interchanging two loops to satisfy the dependence relationships of the original loop. In general, two loops may be interchanged if there is no data dependence with a direction vector of the form $(1, -1)$ [14].

Loop skewing changes the bounds of a loop by adding a value to both the upper and the lower bounds, while at the same time subtracting the same quantity from every appearance of the loop index in the body; as a result, skewing does not change the meaning of the program and, therefore, it is expected to be always a legal transformation.[8] Skewing may enable parallelism in a class of loops when used in combination with loop interchange. This is illustrated in the example shown in Figure 2.2; while in the original code, in Figure 2.2.a, none of the loops can be executed in parallel, after applying loop skewing and loop interchange, a loop which can be executed in parallel (see Figure 2.2.c) results. It is interesting to notice that, although the bounds of the original loops are linear expressions, the bounds of the resulting parallel loop are non-linear expressions.

Loop reversal inverts the order in which the loop iterations are executed (e.g. running from `N` to `1`, rather than `1` to `N`). It may be useful to increase locality, for instance between adjacent loops, but it may also be used to uncover parallelism in conjunction with the previous two transformations; however, this has not been analysed to the extent that the two previous transformations have [15].

The three transformations presented so far are also known as *unimodular transformations*. This is because any combination of them can be formally represented by an $n \times n$ unimodular matrix.[9] This representation is discussed further in Section 2.2.3.

Loop distribution, also called *loop fission* or *loop splitting*, partitions the statements in the loop body of a single loop nest into several loop nests; each of the new loop nests is identical with the original one except that it executes a subset of the statements of the original loop. Thus, it may be possible to execute some of the new loops in parallel, but it may also be possible to create perfect loop nests to which unimodular transformations can be applied. A necessary condition for loop distribution is that statements belonging to a cycle in the statement dependence graph must be placed in the same loop [242, pp. 197–205]. Further techniques for the distribution of loops containing conditional statements are described in [124, 157].

Cycle shrinking can be used in loops where there is a dependence distance, d, greater than one. Then, a serial loop is transformed to an outer serial loop, which scans the original iteration space with step d, and an inner parallel loop where d iterations can be executed in parallel [192, 198]. Cycle shrinking is

[8] Although it has been claimed that loop skewing is always legal [225, p. 137], it is not mentioned that this requires the loop bounds' expressions to be of the same type as the loop index variable. Failure to satisfy this condition may result in a difference in the number of iterations executed between the original and the skewed loop; this phenomenon is illustrated with an example in the next chapter.

[9] A unimodular matrix is a square, integer matrix whose determinant is either 1 or -1.

```
DOALL J=1,N
   DO K=1,N
      DOALL I=1,N
        A(I,J)=A(I,J)+B(I,K)*C(K,J)
      ENDDO
   ENDDO
ENDDO
```

a) Original code.

```
      DOALL JJ=1,N,SJ
         DOALL II=1,N,SI
            DO J=JJ,MIN(JJ+SJ-1,N)
               DO K=1,N
                  DO I=II,MIN(II+SI-1,N)
                     A(I,J)=A(I,J)+
&                           B(I,K)*C(K,J)
                  ENDDO
               ENDDO
            ENDDO
         ENDDO
      ENDDO
```

b) After applying tiling.

Figure 2.3: Using loop tiling for matrix multiplication.

a special case of *strip mining*, which transforms a single loop into a double loop; strip mining is useful primarily in locality enhancement, as discussed below.

2.2.2.2 Transformations for Locality Enhancement

Loop tiling, also known, especially among numerical analysts [78, 84], as *blocking*, generalises strip mining for an n-deep loop nest, transforming it into a $2n$-deep loop nest. Applying loop interchange, assuming that the loops can be interchanged, the iterations are performed over *tiles* (i.e. blocks) of the original iteration space which are of a size that can be fully accommodated in the cache memory. Tiling improves considerably the performance of certain matrix operations, such as matrix multiplication or LU decomposition, on machines with a memory hierarchy [78]; as a result, it has received extensive coverage in the literature [37, 188, 224]. To illustrate its benefits, we consider the well-known code for matrix multiplication, as presented in Figure 2.3.a, and a tiled version, as presented in Figure 2.3.b. Assuming a suitable choice for the tile size, the tiled code is expected to execute significantly faster.[10] Nevertheless, finding an optimum tile size has proved a hard problem, and often its choice depends on experimental determination [194, 218, 223].

Loop fusion is the inverse of loop distribution since it merges two adjacent loops into one. The main requirements for applying loop fusion are that: the loops must have identical loop bounds; and there must be no data dependence between statements belonging to different loops [225, pp. 89–94]. Originally, loop fusion was suggested as a means of reducing loop overhead (as early as the mid-1960's [237]); however, the main benefit for today's computers is to improve data reuse [36, 125, 151].

2.2.2.3 Dependence Breaking Techniques

Scalar expansion is a transformation which enables the parallelisation of a loop whose loop-carried dependences are only data anti-dependences or output dependences [242, pp. 225–229]. By definition, these dependences imply the existence of variables which are computed at an iteration following one

[10] Indicatively, the execution time in seconds of both the untiled and tiled version on the KSR1 for `N` $= 400$, `SJ` $=$ `SI` $= 50$ and a varying number of processors is as presented in the following table:

Processors	1	4	8	16
Untiled Code	114.98	36.82	18.84	10.14
Tiled Code	25.75	6.53	3.33	1.73

The tiled version is always over 4 times faster than the untiled version. Similar behaviour is observed in [223], albeit using a slightly different tiled version. The choice of the version may depend on the architecture as well as the structure of the language used. Recall that, in our case, FORTRAN stores arrays in a columnwise order. Therefore, the order `JKI` of loops in the untiled version maximises the number of unit stride accesses, giving optimum performance, a result that corroborates findings in [36] and [37, Section 4.3].

```
DO K=1,NZ
   DO J=1,NY
      DO I=1,NX
         L=L+1
         A(L)=F(B(K),C(L))
      ENDDO
   ENDDO
ENDDO
```

a) Original code.

```
      DOALL K=1,NZ
         DOALL J=1,NY
            DOALL I=1,NX
               A(L+(K-1)*NY*NX+(J-1)*NX+I)=F(B(K)
&                   ,C(L+(K-1)*NY*NX+(J-1)*NX+I))
            ENDDO
         ENDDO
      ENDDO
```

b) After eliminating the induction variable.

Figure 2.4: Eliminating induction variables to break dependences.

which also made use of them, whether read or write; this may arise, for instance, where temporary scalar variables are used by the programmer. Scalar expansion transforms the latter to arrays, thus creating a copy of the variable for each iteration of the loop and enabling loop parallelisation. If the variable causing the dependences is an array, as, for instance, in the loop nest illustrated earlier in Figure 2.1, then the array will be expanded to a higher dimension; the name *array expansion* has been used to describe this transformation [66, 67]. To reduce the amount of memory required by scalar expansion, an alternative method is to create one copy per processor of any variable causing an anti- or output dependence; this technique is known as *privatisation* and, in experimental studies, it has been shown to be crucial for loop parallelisation [24, 61]. Given that parallelising compilers are not always successful in privatising (or expanding) arrays, substantial recent research has been directed towards array privatisation [148, 213, 214].

Induction variable elimination can enable the parallelisation of loop nests whose only loop-carried dependences are due to statements involving redundant induction variables.[11] For example, consider the program fragment in Figure 2.4.a, which shows a simplified version of a loop nest found in program ADM, an implementation of a hydrodynamic model, from the Perfect Benchmarks. The statement `L=L+1`, which causes the dependences, can be removed since the value of `L` can be computed directly from the loop indices; this results in the loop nest of Figure 2.4.b, for which all loops can be executed in parallel.[12] Attempts to parallelise other programs from the Perfect Benchmarks have found that induction variable elimination is not often feasible; however, where applicable, it can be critical for parallelisation [61].

Index set splitting transforms a single loop nest into multiple adjacent loop nests where each loop performs a subset of the original iterations. Assuming that the order of iterations is kept the same, index set splitting always preserves the loop semantics. Although it is a useful transformation in breaking data dependences [225, p. 65], as well as enabling the use of tiling [37, p. 62], it has been rather neglected in the literature and is not even mentioned in recent surveys of loop transformations [10, 12].

2.2.2.4 Miscellaneous

Two transformations that are more general, but of rather limited practical use, and which are mentioned later, in Chapter 4, are considered below.

Loop coalescing combines multiple loops of a perfect loop nest into a single loop. This has been suggested as a means of improving loop scheduling, as well as reducing loop overhead [180, pp. 49–59]. Consider, for instance, the loop nest in Figure 2.4.b; by combining the three loops into one having an index variable `M` and bounds `1` and `NZ*NY*NX`, the subscripts for the array variables `A` and `C` become `L+M`, while for `B` the subscript becomes `FLOOR((M-1)/(NY*NX))+1`, where the function `FLOOR()` returns the greatest integer less than or equal to the quantity passed as its argument.

Loop normalisation changes the sequence of values of the loop index variable, usually by making the lower bound 0 or 1 and the stride 1, adjusting all references to the particular loop index in the loop

[11] A variable is called an induction variable if, every time it changes value in a loop, it is incremented or decremented by some constant. If there is more than one induction variable in a loop, it may be possible to eliminate them [227].

[12] If `L` is used after the loop terminates, then the statement `L=L+NZ*NY*NX` must follow immediately after the end of the loop. In the example, it is assumed that the value of `L` is not known before the beginning of the loop.

```
 DO J1=10,500
    DO J2=10,500
       A(J1,J2)=A(J1-2,J2-4)
&                +A(J1-1,J2-2)
    ENDDO
 ENDDO
```

a) Original code.

```
 DOALL K1=-990,480
    DO K2=MAX(10,5-FLOOR(K1/2)),
&         MIN(500,250-CEILING(K1/2))
          A(K2,K1+2*K2)=A(K2-2,K1+2*K2-4)
&                       +A(K2-1,K1+2*K2-2)
    ENDDO
 ENDDO
```

b) Transformed code.

Figure 2.5: An example of using unimodular transformations.

body accordingly. Although this was initially suggested to simplify data dependence tests, it has been recently argued that it can distort the properties of data dependences, thus preventing the application of important transformations, such as loop interchange [227]. Furthermore, it tends to render the source code incomprehensible to the user, which may be a disadvantage in interactive environments. However, it may be useful in shortening program analysis, as remarked in Chapter 4.

2.2.2.5 Summary

Two conclusions can be drawn from the transformations described briefly in this section. Firstly, each transformation is subject to its own rules and legality checks, which are usually associated with dependence information. Secondly, and more important in the context of this thesis, some transformations may substantially alter the structure of the loops, as illustrated by the examples presented.

2.2.3 Unified Frameworks

Given the plethora of available loop transformations and the different effects that each of them may have on program structure, a problem that arises for the compiler is how to choose the *sequence* of transformations needed for optimising the program, i.e. keeping the overheads as small as possible. This, in turn, implies a *unified* means for representing transformations, to enable the formulation of rules for deciding which ones should be applied when. The latter remains one of the major issues in automatic parallelisation, even though a significant amount of research effort has been spent already on development of unified frameworks for transformations.

Unimodular transformations, mentioned earlier, provide the means for describing an instance of this problem in linear algebraic terms. This formulation, first analysed by Banerjee [14, 15], permits a more concrete representation compared with earlier approaches, such as those presented by Lamport [138] and Irigoin and Triolet [113], which are commonly cited. A short description follows; for a detailed presentation of the theory, the reader is referred to [16].

Let J denote a vector whose elements are the n loop index variables of a perfect loop nest and K a vector whose elements are the n loop index variables of the transformed loop nest. Then, a unimodular transformation is defined by an $n \times n$ matrix, say U, and the mapping of the original loop nest to the transformed one is defined by the matrix equation $K = JU$, which can be rewritten equivalently as $J = KU^{-1}$. Clearly, in the case where U is the identity matrix I no transformation is performed. In terms of I it is possible to describe three classes of matrices representing respective loop transformations: matrices for loop interchange are obtained by interchanging two columns or rows of I; matrices for loop skewing are obtained by replacing a zero element in I by a nonzero integer; matrices for loop reversal are obtained by negating a diagonal element of I. In the presence of data dependences in the original loop, represented by a dependence distance matrix D, the elements of the matrix DU must be non-negative integers; if a column of DU consists of zero elements, then the respective loop in the transformed code can be executed in parallel. Having determined U, the loop bounds of the transformed loop can be computed by applying Fourier-Motzkin elimination.

In order to illustrate this method, we consider the following example.

Example 2.1 Consider the loop nest illustrated in Figure 2.5.a. It has two distance vectors, $(2,4)$ and $(1,2)$. The unimodular matrix

$$U = \begin{pmatrix} -2 & 1 \\ 1 & 0 \end{pmatrix}$$

defines a transformation which makes the outer loop parallel, since

$$DU = \begin{pmatrix} 2 & 4 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} -2 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 2 \\ 0 & 1 \end{pmatrix}.$$

The transformation from the index variables of the original loop nest, `J1` and `J2`, to the index variables of the transformed loop, `K1` and `K2`, is defined by the equation $(j_1, j_2) = (k_1, k_2)U^{-1}$; therefore, $j_1 = k_2$ and $j_2 = k_1 + 2k_2$. Applying the constraints on j_1 and j_2 to k_1 and k_2 respectively, we get $10 \leq k_2 \leq 500$ and $10 \leq k_1 + 2k_2 \leq 500$; using Fourier-Motzkin elimination, the loop bounds for the new index variables are computed. The resulting code is illustrated in Figure 2.5.b; the outer loop can now be executed in parallel.[13] It is interesting to note the difference in the structure of the transformed loop, compared with the original one. The transformed loop is no longer rectangular, the bounds expressions are more complicated, and the array is not accessed with unit stride; the effect of such changes is of particular interest in this thesis. □

Unimodular transformations can be applied only to perfectly nested loops, and not all the above transformations can be represented by unimodular matrices. To overcome these problems, there has been research on how to cope with non-unimodular transformations [18, 19, 70, 145, 189, 235]. Most notable among these efforts is the unified framework proposed by Kelly and Pugh [118, 119, 120]. Their approach is based on the idea that a transformation can be represented as a schedule or a mapping that maps the original iteration space to a new iteration space. Unimodular transformations can be considered as a special case of schedules; however, the latter remove several of the constraints associated with the former. In particular, schedules allow different ways of mapping each statement of the loop body, while the number of loops of the transformed iteration space can be different from the number of loops of the original one. Thus, the framework can represent most of the transformations described in Section 2.2.2. A further extension, which provides a means for deciding which transformations should be applied to a program, based on a performance estimator, has been described in [121]. However, this framework is still under development and its capabilities have not been fully evaluated.

2.3 Mapping

The outcome of the phase of detecting parallelism is a loop which can be executed in parallel, i.e. a `DOALL` loop. In the mapping phase, the compiler must devise a method to map the parallelism available onto a given parallel architecture in such a way that the overheads, as identified in Equation (1.1), are minimised.

From a parallelising compiler's (or, for that matter, an applications programmer's) point of view, mapping can be regarded as a two-stage process: *partitioning* is the stage at which sequential tasks are formed; *scheduling* is the stage where these tasks are assigned to processors. By definition, partitioning precedes scheduling and, therefore, following the approach used in [196, p. 19], there are three possible options for a compiler: a) to perform both partitioning and scheduling at compile-time, or b) to perform partitioning at compile-time and postpone scheduling until run-time, or c) to postpone both partitioning and scheduling until run-time. The main reason for postponing mapping decisions until run-time is that

[13] The functions `CEILING(K/2)` and `FLOOR(K/2)`, in the transformed code, return the least integer greater than or equal to $k/2$ and the greatest integer less than or equal to $k/2$ respectively; they are both defined in FORTRAN 90 but not in earlier versions of the language. The corresponding mathematical notation of the two functions would be $\lceil k/2 \rceil$, $\lfloor k/2 \rfloor$; a more in-depth discussion is given in Chapter 3. Function `FLOOR()` should not be confused with the built-in function `INT()` of FORTRAN 77, which returns the integer part of the quantity passed as its argument.

additional information, not available at compile-time, may lead to a better workload balance for each processor; however, this is traded for extra run-time overhead.[14]

Mapping has received extensive coverage in the literature. Much research has been devoted to the development of general-purpose techniques for mapping parallel modules onto a given parallel architecture. These techniques are usually based on graph-theoretic approaches;[15] for a survey of related work we refer to [166], while, for an example of a software tool using such a representation to generate code for message passing architectures, the reader is referred to PYRROS, designed by Yang and Gerasoulis [236]. However, these techniques are meant to solve the problem of mapping in the general case of a program involving parallel modules, and the assumptions they make are too simplifying to render them applicable in the context of loops.

Mapping, being a process directly related to the characteristics of the parallel architecture, can be classified according to the targeted architecture. Thus, techniques for mapping loops can be divided into: a) those applicable mostly for shared-memory computers; and b) those applicable mostly for distributed-memory computers. An analytical description of the techniques used in each case is presented next.

2.3.1 Mapping Loops for Shared Memory Computers

Mapping `DOALL` loops for shared memory computers can be achieved by forming tasks comprising a number of iterations and assigning the resulting tasks to processors. Care must be taken that each processor performs the same, or nearly the same, amount of work (i.e. there is a balanced workload), and that any associated overheads are minimised. The process of finding such a mapping scheme has generally been called in the literature *loop scheduling* [76, 149, 152, 180, 215, 238]. As discussed earlier, this term tends to over-emphasise the importance of scheduling the loop iterations.

A number of loop mapping schemes are described below. Based on whether the scheduling decisions are taken before or during run-time, we distinguish between *static loop mapping schemes* (where, usually, partitioning is the most critical aspect) and *dynamic loop mapping schemes* (where, usually, scheduling is the most important aspect), respectively.

2.3.1.1 Static Loop Mapping Schemes

A simple way to map a loop is to distribute its iterations among processors as evenly as possible, with each processor being assigned a fixed number of iterations. For a given loop, there may exist a huge number of distinct distributions; however, assuming that simplicity is preferred, there are two main approaches. The first is to map the iterations onto processors in a round robin fashion; thus, given p processors and assuming that the total number of iterations is n, processor 0 executes iterations $1, p+1, 2p+1, \ldots$, processor 1 executes iterations $2, p+2, 2p+2, \ldots$, in general, processor i, $i = 0, 1, \ldots, p-1$, executes iterations $i+1+kp, k = 0, 1, 2, \ldots, (n/p-1)$. We refer to this scheme as *cyclic partitioning*. The second approach is to map contiguous iterations onto processors in a consecutive manner; thus, processor 0 executes iterations 1 through n/p, processor 1 executes iterations $n/p+1$ through $2n/p$, in general processor i, $i = 0, 1, \ldots, p-1$, executes iterations $in/p+1$ through $(i+1)n/p$. We refer to this scheme as *block partitioning*.[16]

---

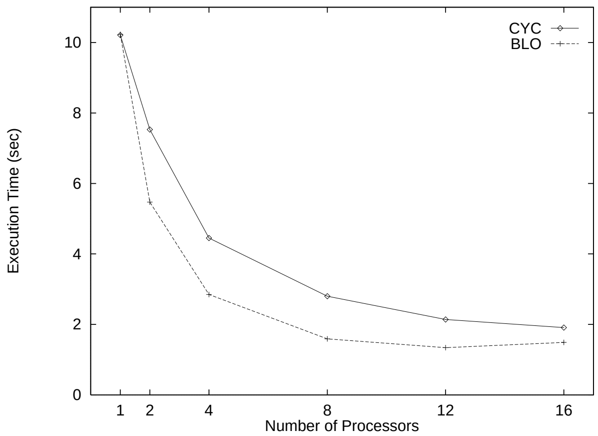
[14] It must be noted that, in many cases, the mapping literature fails to make a clear distinction between partitioning and scheduling. The main reason for this can be traced to the historical development of the concepts. Mapping, being directly influenced by the parallel architecture to which it would be applied, has been viewed differently by the designers of different models [141, 161]. In some cases, scheduling has been used as an alternative term to describe the whole process of mapping. A recognisable trend is the use of 'scheduling' to describe a mapping process where the critical decisions are taken during run-time, while 'partitioning' has been used to imply that the critical decisions are taken during compile-time. In this sense, partitioning is more pertinent to this thesis. Nevertheless, the review presented in the next paragraphs adopts the commonly established mapping terminology.

[15] In the most common case, a parallel algorithm is represented by a directed acyclic graph whose vertices represent processes and whose edges represent communication paths; an undirected graph, whose nodes represent processing elements and edges the links between them, is used for the representation of the parallel computer. The problem consists of finding a mapping of the former graph onto the latter by optimising some objective functions; since the problem is NP-hard, one has to rely on efficient heuristics [41].

[16] In both approaches, for simplicity, it has been assumed that p divides n; a more in-depth discussion of the mathematics of partitioning is given in Section 4.2.

```
DOALL J=1,N
   DOALL I=1,N
      A(I,J)=A(I,J)+B(I,J)
   ENDDO
ENDDO
```

a) Code for matrix addition.



b) Execution time for `N` $= 2000$.

Figure 2.6: The effect of false sharing on performance.

However, in many parallel computers, the former approach may exhibit poor performance due to a phenomenon known as *false sharing*; this occurs when two processors access data in the same cache line, and one of the accesses is a 'write', thus causing the cache line to be exchanged between processors even though the processors access different parts of it. To illustrate the effects of false sharing on performance, consider the code fragment shown in Figure 2.6.a, which adds two $n \times n$ matrices. The `J` loop was parallelised by distributing the iterations among processors using both mapping schemes described: cyclic partitioning (CYC) and block partitioning (BLO). Performance results were obtained on the KSR1 for `N` $=$ 2000 and are presented graphically in Figure 2.6.b. It can be seen that assigning consecutive iterations to processors (i.e. block partitioning) results in better performance. Similar behaviour, performing Cholesky factorisation on the KSR1, is reported in [146].

The methods described presuppose that each iteration of the parallel loop performs the same amount of work and, therefore, that the workload assigned to each processor is nearly equal; however, this may not always be the case. For instance, two iterations may perform different amounts of work as a result of statements in the loop body (such as conditionals or loops) whose execution depends on the value of the index of the parallel loop. In the case of a perfect loop nest with multiple parallel loops, partitioning and scheduling can be performed along a parallel loop which presents an invariant workload for each iteration; a compiler technique to determine whether this is possible has been described in [168]. In the case of a double loop consisting of an outer parallel loop and an inner loop having bounds dependent on the index of the outer loop, Haghighat and Polychronopoulos [93, 94] suggest *balanced chunk scheduling*. This scheme attempts to distribute the total number of iterations of the loop body among processors as evenly as possible, as opposed to cyclic or block partitioning which distribute only the number of iterations of the outer loop as evenly as possible; the result is that the number of iterations of the parallel loop assigned to each processor varies, however, each processor executes the loop body almost the same number of times.

To illustrate balanced chunk scheduling, consider the code fragment shown in Figure 2.7.a, which adds two, upper triangular, 800×800 matrices. The `J` loop was parallelised and, using 16 processors, the iterations are distributed among processors as shown in Figure 2.7.b; thus, processor 1 executes iterations 1 through 200, processor 2 executes iterations 201 through 283, and so on.[17] Balanced chunk scheduling

[17] Using a number of processors, say p, which divides 16, the iterations assigned to each processor are found by merging the first $16/p$ groups of iterations described in Figure 2.7.b; thus, using 8 processors, processor 1 executes iterations 1 through 283, processor 2 executes iterations 284 through 400, and so on. Using a number of processors which does not divide 16, additional computation may be necessary. A methodology for partitioning the iterations for balanced chunk scheduling is described in Appendix B.

```
DOALL J=1,800
   DOALL I=1,J
      A(I,J)=B(I,J)+C(I,J)
   ENDDO
ENDDO
```

a) Code for upper triangular matrix addition.

1: 1-200	**2:**201-283	**3:**284-346
4:347-400	**5:**401-447	**6:**448-490
7:491-529	**8:**530-566	**9:**567-600
10:601-632	**11:**633-663	**12:**664-693
13:694-721	**14:**722-748	**15:**749-775
16:776-800		

b) Iterations of the J *loop assigned to each processor under balanced chunk scheduling using 16 processors.*

Figure 2.7: Balanced chunk scheduling.

is limited in that, in its standard form, it can be applied only to perfect loop nests; Haghighat and Polychronopoulos also restrict their discussion to double loops and assume that the number of iterations is known at compile-time. However, its main disadvantage lies in what is perceived as its main advantage: not distributing evenly the iterations of the outer loop is often a non-desirable option in practice.

In the case of more general loop nests, it has not been considered feasible to distribute the workload evenly at compile-time. Apart from variances in the execution time of loop iterations, additional overheads, mainly due to synchronisation and/or communication requirements (e.g. page faults, cache misses, interprocessor communication, etc.), may affect each processor in a different way. Thus, researchers have been led towards the development of run-time techniques for mapping.

2.3.1.2 Dynamic Loop Mapping Schemes

A simple method to achieve workload balance at run-time is to schedule one iteration at a time to each idle processor until all iterations are executed; this technique is known as *self-scheduling* [65, 208]. Its disadvantage is that it incurs a high overhead; assuming that the n iterations to be scheduled are stored in a central queue, processors have to access the queue n times, while, to avoid the same iteration being scheduled twice, access to the queue must be limited to one processor at a time. To reduce the overhead, it has been suggested that groups of k iterations should be formed and then scheduled together; this approach is known as *chunk scheduling* [135]. To determine an appropriate value for the chunk size, k, probabilistic analysis is used. It is concluded that, when the scheduling overhead is zero, the optimum chunk size is $k = 1$ iteration, while, when the scheduling overhead is large, the optimum chunk size is $k = \lceil n/p \rceil$ iterations, i.e. the maximum possible value for k. However, it is also argued that the choice $k = \lceil n/p \rceil$ is optimum for large n, regardless of the scheduling overhead.[18]

To reduce the scheduling overhead, while at the same time achieving good workload balance, it has been suggested that a decreasing chunk-size allocation should be performed; this scheme is known as *guided self-scheduling* [179]. Whenever a processor is idle, a chunk of $\lceil r/p \rceil$ iterations is assigned to it, where r is the number of remaining iterations to be scheduled and p is the number of processors; the initial value of r is n, the total number of loop iterations. One disadvantage of this scheme is that, towards the end of execution, many chunks containing a single iteration may be scheduled, thus causing significant overhead.

Guided self-scheduling may exhibit poor workload balance when the iterations scheduled during the first steps of the algorithm contain much more work than later iterations. To overcome this, a scheme called *factoring* [76] initially schedules chunks with a smaller number of iterations than guided self-scheduling. Thus, whenever a processor is idle, a chunk of $\lceil r/2p \rceil$ iterations is assigned to it.

The main concern of the above schemes is to offer better workload balance than chunk scheduling while keeping the scheduling overhead to a minimum. A compromise between these conflicting requirements is offered by *trapezoid self-scheduling* [215, 216], which schedules chunks of iterations based on a

[18] When $k = \lceil n/p \rceil$, chunk scheduling coincides with block partitioning. The argument put forward by Kruskal and Weiss [135] favours static schemes as opposed to dynamic schemes.

predetermined value for the maximum and minimum number of iterations each chunk may have. Assuming that these values are represented by k_h and k_l, respectively, and that the chunk size is gradually decreasing in a linear fashion each time a scheduling request is made, the total number of chunks to be scheduled is $s = \lceil 2n/(k_h + k_l) \rceil$; then, after each scheduling step, the chunk size is decreased by $(k_h - k_l)/(s-1)$. A more general approach to estimating the chunk size at each step, based on the characteristics of genetic algorithms, is described in [238].

An issue not addressed by the schemes described so far is that, especially in computers with a memory hierarchy, it may be preferable to execute certain iterations on certain processors in order to reuse data or to avoid false sharing; this need for locality exploitation was the motivation for developing *affinity scheduling* [152, 153]. This algorithm combines static and dynamic approaches. It first assigns $\lceil n/p \rceil$ loop iterations to each processor on the basis of block partitioning. At run-time, whenever a processor is idle, $1/p$-th of the remaining iterations are removed from the most heavily loaded processor and transferred to the idle processor.

2.3.1.3 Discussion

Dynamic loop mapping schemes attempt to create a balanced workload for each processor at run-time; however, achieving this goal may require additional communication (and run-time) overhead. Alternatively, a static loop mapping scheme may distribute work among processors in an unbalanced way, but it may also incur less communication and, by exploiting the memory hierarchy in a more efficient way, it may result in faster execution. In the context of automatic parallelisation, where decisions are usually taken on the basis of a machine-based economic model, dynamic loop mapping schemes do not provide any insight into the mapping phase. Thus, when modelling communication costs, researchers have preferred to use static schemes for mapping loops [87, 88].

2.3.2 Mapping Loops for Distributed Memory Computers

In distributed memory computers, given that communication is a dominant cost, parallelism is usually exploited by means of the data parallel paradigm (see Section 1.4.2.1). The elements of arrays are distributed among processors and each processor can be thought of as 'owning' the elements assigned to it. Thus, to minimise communication between processors, work is performed according to the data distribution. A processor always computes the values of the data it owns; data on the right-hand side of an assignment statement which are located in other processors are communicated to the processor performing the computation. This method is known as the *owner computes rule* [33].

To illustrate how this rule is applied, consider again the matrix addition shown in Figure 2.6.a. Assuming that both arrays, `A` and `B`, have been distributed among processors in equal blocks of consecutive columns, then the parallelisation of the loop is equivalent to block partitioning, as described in Section 2.3.1.1. However, in many cases, the array distribution must remain unchanged throughout the program; as a result, some loops may exhibit an imbalanced workload.

Applying the owner computes rule, the problem of mapping loops is substituted by the problem of finding an efficient data distribution. The latter is an NP-complete problem [134, 144], and has been the subject of extensive research which aims at making the compilers advanced enough to take a decision automatically [9, 23, 90, 91, 108, 133, 169, 222]. Historically, the origins of these ideas can be placed at the University of Bonn and the efforts for developing SUPERB, a parallelisation tool for distributed memory computers [241]; similar early research is also described in [33]. Part of the continuing research effort has led to the development of HPF [103, 131], although the latter includes only a subset of the possible ways of distributing an array. Usually, decisions concerning the data distribution are taken before applying any program transformations; thus additional analysis may be necessary before applying the latter [105, 243].

2.4 Effectiveness of Parallelising Compilers

Evaluating the effectiveness of parallelising compilers in a broad spectrum of applications is a fundamental requirement to demonstrate their feasibility and reveal possible inadequacies.

Some studies have concentrated on the effect that individual transformations have on the performance of parallelising compilers. In an early study, transformations used by Parafrase[19] were evaluated by being applied to a set of EISPACK[20] algorithms; it turned out that scalar expansion was the transformation that yielded the highest performance gains [53]. More recent studies have corroborated this; privatisation, of both scalars and arrays, is clearly important for enabling parallelisation of code [24, 177]. Many restructuring techniques have been found ineffective [25]; however, it has been argued that they can become more effective in the context of more advanced compiler technology [60]. The effect of restructuring transformations has also been studied in the context of transformations for the exploitation of data parallelism [106]. Other experimental studies have attempted to evaluate the effectiveness of data dependence techniques [83, 155, 200].

The evaluation of individual transformation techniques may indicate directions for possible research and improvement. However, to assess the overall performance of a parallelising compiler it is necessary to compare it with the performance achieved when using manual techniques for parallelisation. Some earlier approaches focused on the number of loops a parallelising compiler would parallelise, compared to the total number of parallel loops in a program, and the resulting speed-up achieved [143]. However, this approach can lead to false impressions; some loops may be more time-consuming than others and, therefore, it is more important for the compiler to be capable of parallelising the former, rather than the latter (where the overhead for initialising parallelism may outweigh the potential gains from parallel execution). In fact, parallelising compilers tend to parallelise everything 'parallelisable' without examining whether this might result in lower performance; we will return to this issue in Section 2.5.

Many studies of effectiveness have compared the running time of the code produced by various parallelising compilers to the running time of the code produced by programmers. These studies differ in the type of compilers examined and in the parallel computers and programs used; for typical studies we refer to [25, 61, 201], while a concise presentation of several studies with many references can be found in [12]. Despite the questionable validity of the particular choices in these studies, it appears that a consensus has been reached: in most cases parallelising compilers do not effect significant improvements in program performance.[21] This observation has raised the question of whether and where there is scope for potential improvements. Apart from suggestions for improving specific techniques, such as array privatisation [24], or for more robust dependence techniques [200, 201], the importance of having compilers capable of handling variables whose values are unknown at compile-time has been stressed [176]. The latter is examined further in the following chapters.

2.5 Discussion

At first glance, it seems reasonable to associate the ineffectiveness of automatic parallelisation techniques with the more general research questions raised by parallel computing (see [162, pp. 38–48] for a synopsis). In particular, issues such as the lack of appropriate analytical models of parallel computation, or the

[19] The Parafrase compiler was one of the earliest tools for automatic program parallelisation developed as part of a project started in the late 1970's at the University of Illinois [137]. As its name implies, Parafrase is a source-to-source restructuring tool capable of transforming sequential FORTRAN programs to a form suitable for execution on a parallel machine. A successor to Parafrase has been KAP, standing for Kuck's Automatic Parallelizer, a successful commercial automatic parallelisation tool. Historically, the first automatic parallelisation tool was developed for ILLIAC IV in the early 1970's; the tool was known as the Paralyzer [182] and, although limited in scope, it included the notion of data dependence.

[20] EISPACK [79, 202] is a library of FORTRAN subroutines for solving eigenvalue problems; alongside LINPACK it has been superseded by LAPACK, a library of FORTRAN 77 subroutines for solving the most commonly occurring problems in numerical linear algebra [8].

[21] This statement does not imply that there has been no progress in automatic parallelisation techniques through the years, nor that these techniques cannot be useful in a variety of cases; as Levine *et al.* [143] point out, loops that had been considered challenging in past years were easily parallelised by the systems they used. Instead, this statement should be used as an indication of the need for further research into new techniques and strategies.

diversity of parallel architectures, pose additional difficulties for the task of parallelising compilers; the latter aim at formalising a procedure which is of a rather experimental nature, based on a 'measure and modify' approach rather than an efficient model.

However, the capability for predicting program performance is necessary for a parallelising compiler. The main reason is that program transformations may have negative effects on execution performance. Consider again, for instance, the example presented in Figure 2.5. Although the transformed code can run in parallel, it introduces a non-unit stride access pattern to the array `A`; furthermore, the transformed loop nest is non-rectangular and, therefore, sophisticated mapping algorithms may be needed. These issues are likely to cause an additional overhead in the transformed code. On the other hand, by applying loop interchange (which is legal since no direction vector of the form $(1, -1)$ exists) to the original code, a unit stride access can be introduced. The execution time of the latter version was compared with that of the transformed, parallel code on the KSR1. Using a single processor, the amended sequential code is approximately 5 times faster than the transformed parallel; gains from parallelism result in better execution time for the transformed code only when using more than 8 processors.

Using a simpler example, consider the code for matrix addition, illustrated in Figure 2.6.a. It is trivial for a parallelising compiler to detect the parallelism of this code. However, KAP, for instance [122, Ch. 9], would parallelise the code regardless of the value of `N`, even if its value was known at compile-time. For sufficiently small values of `N`, the overheads due to parallelisation will outweigh the gains from parallel execution; on the KSR1, for example, the sequential version of the code was faster than any parallel version for values of `N` less than 50. Thus, KAP, having a strategy of parallelising anything parallelisable, may slow down the execution time of the parallel program. This problem has been pointed out in studies of the effectiveness of parallelising compilers [25], and it has also been reported from a scientific user's point-of-view [174, Sec. 2.4].

As mentioned earlier, one possible solution is to provide a parallelising compiler with a model for performance estimation. Adopting this position, in Section 1.4.3.2, we considered such a model based on the overheads incurred. Using this model, automatic parallelisation can be regarded as an optimisation problem, the solution of which is given by the sequence of transformations which minimises the time spent in overheads, t_o, according to Equation (1.1). Assuming, for instance, that an estimate for the execution time of the sequential program, t_s, is also known, it would be possible to overcome the problem discussed in the previous paragraph.

Certainly, we do not advocate that, having expressed automatic parallelisation as an optimisation problem, it is feasible to solve it; rather, the number of possible transformations is likely to be prohibitively high to allow exploration of the effect of all possible combinations at compile-time.[22] Even worse, accurate computation of the parameters of the model may be a difficult task. However, by making appropriate simplifying assumptions, similar models have been described and used recently [29, 64, 220].

Regardless of the performance model adopted, the most important implication of using any model and having to compute its parameters is that the compiler must be capable of extracting quantitative measures from programs. For this purpose, symbolic analysis techniques, i.e. techniques which can manipulate algebraic expressions containing variables of unknown value, are necessary. Such techniques have been considered, in the context of sequential programs, as a means of predicting the execution time or analysing the complexity of programs in terms of the input data [102, 221], but their use in the context of parallelising compilers appears to be more demanding; it would be desirable to apply them to derive expressions for all the overheads involved in Equation (1.1).

In several cases, measures for these sizes have been based on average values derived through program profiling information; for instance, this approach has been used in [196]. Its disadvantage is that information based on average values may be misleading when there is significant deviation from the average. This problem has been observed in [196, p. 157]; considering the case of a non-rectangular loop, such as that shown in Figure 2.7.a, information based on average values implicitly assumes that each iteration

[22] The use of linear programming techniques to find the appropriate choice of transformations for a loop is suggested in [199]; the authors do not address overhead issues, thereby making the application of this method questionable for practical purposes. Usually, parallelising compilers decide *a priori* the order in which the transformations are applied (for example, PTRAN [2], Parafrase-2 [181]), or they explore interactively a larger number of possible combinations, as in the case of Parascope [46].

of the outer loop takes the same time to execute, which, in this case, is not correct. As a potential solution, it is suggested that the use of symbolic expressions may lead to "greater rewards due to the extra information contained in them" [196, p. 157]. As already mentioned, in the context of parallelising compilers, symbolic analysis techniques have been considered crucial for advanced data dependence tests [28, 92], as well as for interprocedural analysis [99].

2.6 Concluding Remarks

The remainder of this thesis investigates the use of symbolic techniques in the context of estimating performance; in particular, Chapter 3 is devoted to the issue of counting loop iterations. Using this information, it is possible to evaluate the effects of compile-time loop mapping schemes; this issue is discussed in Chapter 4. From the examples presented earlier in this chapter, it has been observed that several transformations may alter the form of a loop nest from rectangular to non-rectangular. Thus, it becomes imperative to search for compile-time loop mapping schemes that are more robust than, for instance, balanced chunk scheduling. In combination with the program calculus emerging through the use of unimodular transformations and other unified frameworks, these mapping schemes may provide a useful tool for more sophisticated decision-making in automatic parallelisation.

Chapter 3

Counting Loop Iterations

3.1 Introduction

This chapter describes a framework for computing, symbolically, the number of times a statement in the body of a loop nest is executed. This information may be used by the compiler to derive estimates for the execution time of the loop nest or for particular overheads, such as load imbalance, as will be shown in Chapter 4. As a result, the compiler can assess the performance gains expected from particular parallelisations of the loop nest.

In the case of a single loop in which the loop bounds are integer expressions and there is a unit stride,[1] the number of loop iterations, n, is, by definition, given by

$$n = \sum_{i=a}^{b} 1 = \begin{cases} b-a+1, & \text{if } b \geq a, \\ 0, & \text{otherwise,} \end{cases} \tag{3.1}$$

where a, b are the lower and upper bounds of the loop, respectively; thus, each statement of the loop body is executed n times, assuming that the loop body contains no statements (such as conditionals) affecting the flow of control in the program. In the case where a statement is surrounded by more than one, say m loops, and, for each loop, the loop bounds are constant, integer expressions, known at compile-time, it is trivial to show that the statement will be executed a number of times, n, given by

$$n = \sum_{i_1=a_1}^{b_1} \sum_{i_2=a_2}^{b_2} \cdots \sum_{i_m=a_m}^{b_m} 1 = \prod_{i=1}^{m} n_i, \tag{3.2}$$

where $n_i = \sum_{i_j=a_j}^{b_j} 1$, and a_j, b_j are the lower and upper bounds, respectively, of the j-th loop, $1 \leq j \leq m$.

However, this formulation of the problem is not adequate to provide solutions in a variety of situations where the loop bounds are non-constant or are expressions whose value is unknown at compile-time. The simplest case in this class is a loop nest in which the index of an outer loop forms part of a bound expression of an inner loop; such is the case with any non-rectangular loop nest. Recall, for instance, the loop nest shown in Figure 2.7.a; in order to deal with this for the purposes of balanced chunk scheduling, Haghighat and Polychronopoulos [93] suggest actually executing the loop nest by assuming that a variable, initialised to zero, is incremented by one at each iteration of the loop nest. This approach, also adopted for computing sums by packages allowing symbolic computation, such as Mathematica [230], may be time-consuming.

Trying to find analogues of this problem, it is useful to observe that a loop index is usually confined by an inequality, such as $\alpha \leq i \leq \omega$, where α, ω are the lower and upper bounds of the loop, respectively. Assuming that a statement is surrounded by a number of loops and conditionals affecting its execution,

[1] Recall that for the purpose of this thesis the notion of loop is confined to `DO ... ENDDO` loops, or equivalently, if the loop can be executed in parallel, to `DOALL ... ENDDO` loops.

the entire set of corresponding inequalities define a *polytope*.[2] Then, the number of times the statement will be executed is equal to the number of integer points in the polytope; the entire set of these points corresponds to the notion of iteration space introduced in Section 1.4.2.2.

It is not known whether a polynomial algorithm for counting the number of integer points in a polytope exists; it appears that the problem is more complicated than the more widely studied problem of computing the volume of a polytope [20]. The latter, which can lead to an approximation of the number of integer points, is a #P-hard problem [58]; it has been shown that every approximate polynomial algorithm for computing the volume will return a result whose ratio of upper to lower bound will be an exponential expression in terms of the polytope dimensionality [17]. Such an approximate algorithm is described in [59]; for exact algorithms the reader is referred to [44, 139]. Nevertheless, for certain simple polytopes, corresponding to the kind of loop nests which are more likely to be dealt with in practice, it may be possible to use direct formulas to compute the volume; this approach is discussed further in Section 3.2.

The remainder of the chapter is devoted to the development of a methodology for evaluating summations, such as those in Equations (3.1) and (3.2), corresponding to loop nests with non-constant or unknown, at compile-time, bounds. The evaluation of summations to compute the number of iterations in a loop nest has also been considered by Nadia Tawbi [209, 210, 211]. Her approach concentrates on the description of an algorithm for splitting the computation into sub-parts which can be manipulated; issues, such as those arising from the presence of conditionals or complicated loop bounds expressions, are not addressed. A more in-depth study has been carried out by William Pugh [185], who considers the more general problem of counting solutions of *Presburger formulas*.[3] However, simplifying and verifying Presburger formulas may be prohibitively expensive [74]; furthermore, Pugh's framework incorporates techniques more complicated than those strictly necessary for counting loop iterations.

3.2 A Geometric Approach

The representation of a loop nest (or an iteration space) as a polytope allows us to deal with the problem from a geometrical point of view; its visualisation may provide a clearer understanding. This approach has been used often since the early years of Lamport's hyperplane method [138]; it has been used already in this thesis, in Section 2.2.1, where a Euclidean system of coordinates is employed to depict the iteration space and illustrate the dependences between loop iterations (see Figure 2.1.b).

In the general case, graphical representation of a polytope requires an algorithm for finding all of its vertices from a given set of constraints; such an algorithm is described in [42, Ch. 18]. For an exhaustive list of references, the reader is referred to [197, p. 224]; issues pertinent to the process of visualisation are discussed in [38].

The polytopes corresponding to the loop nests found in real programs are, for the most part, simple geometrical figures; it has become common to characterise the iteration space of such loop nests from the shape of the respective figures. In a double loop nest, for instance, a rectangular iteration space implies that the corresponding polytope is a rectangle; similarly, a triangular or a trapezoidal iteration space implies that the corresponding polytope is a triangle or a trapezium, respectively. The well-known formulas for computing the area of these shapes can be used as an approximation of the number of iteration points they contain.

To illustrate the above, consider the loop nests shown in Figures 3.1.a and 3.1.b; the respective iteration spaces for each loop nest, and the corresponding polytopes, illustrated by the dark shaded

[2] Geometrically, a polytope can be defined as a finite convex region of m-dimensional space enclosed by a finite number of hyperplanes [48, p. 126]; intuitively, a polytope may be considered as the general term of the sequence *point*, *segment*, *polygon*, *polyhedron*, ... Using an algebraic framework, a polytope is represented by the inequality $Ai \leq b$, where A is a $k \times m$ matrix, and i, b are vectors; m is the dimensionality of the polytope and k is the number of constraints. In our framework, m is the number of surrounding loops and k is equal to $2m$ if no conditionals are present. For a detailed introduction to the theory, the reader is referred to [197, Ch. 7]; its application in the context of automatic parallelisation is illustrated in [170].

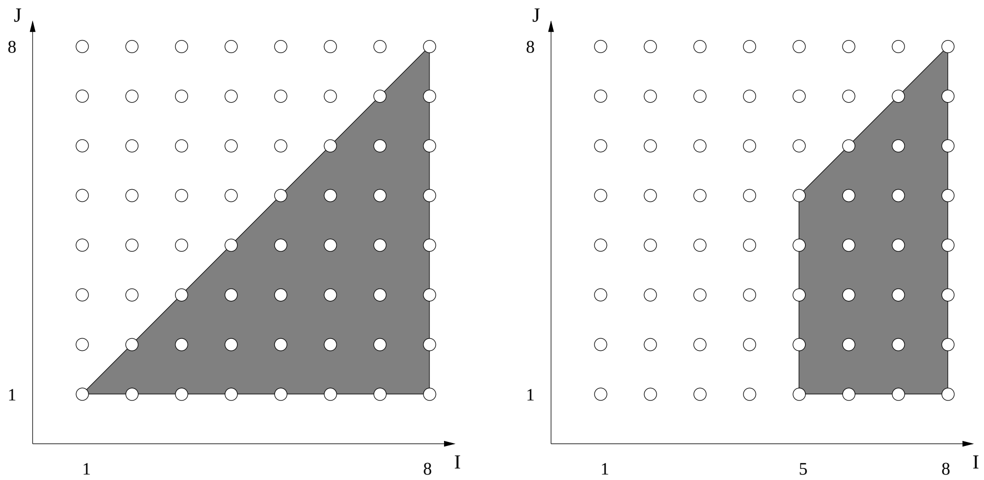
[3] Presburger formulas are those constructed by applying the first order logical connectives, $\wedge$, $\vee$, $\neg$, $\Rightarrow$, and/or quantifiers, $\forall$, $\exists$, to linear constraints [110].

```
DO I=1,N
   DO J=1,I
      (statements)
   ENDDO
ENDDO
```

a) A triangular loop nest.

```
DO I=5,N
   DO J=1,I
      (statements)
   ENDDO
ENDDO
```

b) A trapezoidal loop nest.



c) Geometrical representations of the above loop nests for $\mathtt{N} = 8$.

Figure 3.1: Examples of triangular and trapezoidal loop nests.

areas, for `N` $= 8$, are shown in Figure 3.1.c. Applying the well-known formula $bh/2$ for computing the area, a, of a triangle with base b and height h, we get $a = 32$ (assuming that $b = h = 8$; although the figure implies that the euclidean distance for b and h is 7, it is more realistic to regard the polytope as an approximation of a grid representing the iteration points in space); this is comparable with the value of 36, which represents the actual number of iteration points, as can be computed from the figure. Similarly, applying the formula $(b_1 + b_2)h/2$ for computing the area, a, of a trapezium with parallel sides b_1 and b_2 and distance between them h, we get $a = (5 + 8) \cdot 4/2 = 26$; this value coincides with the actual number of iteration points.

In general, the approximation error may be high; for instance, in the case of the loop nest shown in Figure 3.1.a it is `N`/2. Furthermore, it turns out that it is feasible to resort to an algebraic framework to obtain exact answers. Thus, the remainder of this thesis makes use of the geometric approach only as a means of visualising the techniques described.

3.3 An Algebraic Approach

As mentioned in Section 3.1, the number of times, n, that a statement surrounded by several loops is executed, can be computed by an expression of the form

$$n = \sum \sum \ldots \sum 1, \tag{3.3}$$

where each sum applies over the iteration space of a single loop. The assumptions made about the loop bounds (i.e. that they are constant, integer expressions whose value is known at compile-time) allow us to express these multiple sums as the product given in Equation (3.2); dropping these assumptions, the value of n can be computed by evaluating the $\sum$'s from right to left (inside-out, or from the innermost sum to the outermost) since (3.3) is an abbreviation for

$$n = \left(\sum \left(\sum \ldots \left(\sum 1\right) \ldots\right)\right). \tag{3.4}$$

The innermost sum in Equation (3.4) can be evaluated by applying the rule described in Equation (3.1). In order to proceed with the remaining terms, it is useful to recall some fundamental rules for manipulating sums; for further discussion the reader is referred to [85, Ch. 2].

Let K be a finite set of integers, k an index variable taking all values of K and a_k, b_k polynomial expressions in k; then the following three rules apply:

- The *distributive law*,
$$\sum_{k \in K} ca_k = c \sum_{k \in K} a_k,$$
where c is a constant expression, independent of k.

- The *associative law*,
$$\sum_{k \in K} (a_k + b_k) = \sum_{k \in K} a_k + \sum_{k \in K} b_k,$$
which holds for subtraction as well.

- The *commutative law*,
$$\sum_{k \in K} a_k = \sum_{f(j) \in K} a_{f(j)} = \sum_{j \in J} a_{f(j)},$$
where the function f is a one-to-one correspondence between J and K.

Using these three rules, additional properties of sums can be deduced. An important rule for combining different sets of integers follows: if K_1, K_2 are any sets of integers, then

$$\sum_{k \in K_1} a_k + \sum_{k \in K_2} a_k = \sum_{k \in (K_1 \cap K_2)} a_k + \sum_{k \in (K_1 \cup K_2)} a_k. \tag{3.5}$$

$$\begin{aligned}
1+2+3+\ldots+n &= \frac{n(n+1)}{2} \\
1^2+2^2+3^2+\ldots+n^2 &= \frac{n(n+1)(2n+1)}{6} \\
1^3+2^3+3^3+\ldots+n^3 &= \left(\frac{n(n+1)}{2}\right)^2 \\
1^4+2^4+3^4+\ldots+n^4 &= \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30} \\
1^5+2^5+3^5+\ldots+n^5 &= \frac{n^2(n+1)^2(2n^2+2n-1)}{12} \\
1^6+2^6+3^6+\ldots+n^6 &= \frac{n(n+1)(2n+1)(3n^4+6n^3-3n+1)}{42}
\end{aligned}$$

Table 3.1: Formulas for computing sums of powers of positive integers.

The above rule can be used for combining or splitting sums, as in

$$\sum_{k=l}^{u} a_k = \sum_{k=l}^{m} a_k + \sum_{k=m+1}^{u} a_k, \quad \text{where } l \leq m < u.$$

The rules outlined so far permit the manipulation and full evaluation of multiple sums, such as that described in Equation (3.3), as long as the indices of the sums are not involved in the summand. If the index of a sum is present in the summand, it is necessary to evaluate expressions of the form

$$\sum_i i^p. \tag{3.6}$$

In order to evaluate (3.6) we can use the so-called Bernoulli formula, named after Jacob Bernoulli (1654–1705) who first worked out formulas for sums of p-th powers:

$$\sum_{i=1}^{n} i^p = \frac{1}{p+1} \sum_{k=0}^{p} \binom{p+1}{k} B_k (n+1)^{p-k+1}, \quad \text{for all integers } p, n \geq 1, \tag{3.7}$$

where

$$\binom{p+1}{k} = \frac{(p+1)!}{k!\,(p+1-k)!}.$$

Equation (3.7) can be proved by induction; for the proof the reader is referred to [85, p. 270, p. 352]. The numbers B_k in Equation (3.7) are the *Bernoulli numbers*, defined by the recurrence relation:

$$\sum_{i=0}^{n} \binom{n+1}{i} B_i = 0, \quad \text{for all integers } n \geq 1, \text{ and } B_0 = 1.$$

Based on the above relation, the first few Bernoulli numbers are as follows: $B_1 = -1/2, B_2 = 1/6, B_3 = 0, B_4 = -1/30, B_5 = 0, B_6 = 1/42$.[4] Using these values, closed formulas for evaluating sums, such as the one in (3.6), for $p \leq 6$ are given in Table 3.1. Sums involving powers higher than the sixth

[4] Some books use a different notation for Bernoulli numbers: in [204], for instance, the first two terms, B_0, B_1, and all odd terms B_k for $k \geq 3$ are omitted from the sequence (the latter turn out to be always zero). The notation adopted in this text is the one followed by Knuth [128], which seems the most consistent.

virtually never occur in real problems; a sum of sixth powers would correspond to a loop nest of depth at least 7 where the bounds of 6 inner loops depend on the value of the index of a surrounding loop, which is, to say the least, an uncommon case.

Since $0^p = 0$, for $p \geq 1$, we have

$$\sum_{i=0}^{n} i^p = \sum_{i=1}^{n} i^p. \tag{3.8}$$

For the general case

$$\sum_{i=a}^{b} i^p, \quad \text{where } a < b,\, p \geq 1, \tag{3.9}$$

depending on the value of a, b the following subcases can be distinguished:

- If $a > 1$ then, using (3.5), we have

$$\sum_{i=a}^{b} i^p = \sum_{i=1}^{b} i^p - \sum_{i=1}^{a-1} i^p. \tag{3.10}$$

  The right-hand side in (3.10) can be computed based on (3.7).

- If $a = 0$ or $a = 1$ then (3.7) can be applied in a straightforward manner.

- If $a < 0$ then:

  – If $b > 0$ then

$$\sum_{i=a}^{b} i^p = \sum_{i=a}^{-1} i^p + \sum_{i=0}^{b} i^p. \tag{3.11}$$

  Applying the commutative law we replace i with $-i$ at the first sum of the right-hand side of (3.11) and we have

$$\sum_{i=a}^{-1} i^p = \sum_{i=1}^{-a} (-i)^p = (-1)^p \sum_{i=1}^{-a} i^p. \tag{3.12}$$

  Therefore, the right-hand side of (3.11) becomes

$$(-1)^p \sum_{i=1}^{-a} i^p + \sum_{i=0}^{b} i^p \tag{3.13}$$

  where both sums can be computed using (3.7).

  – If $b \leq 0$ then, applying again the commutative law,

$$\sum_{i=a}^{b} i^p = \sum_{i=-b}^{-a} (-i)^p = (-1)^p \sum_{i=-b}^{-a} i^p, \tag{3.14}$$

  where $0 \leq -b < -a$, and, therefore, (3.7) or (3.10) can be applied.

In order to illustrate the above, we present the following example.

```
C**  N integer, greater than 5
       DO I=5,N
          DO J=1,I
             DO K=1,J
                (statements)
             ENDDO
          ENDDO
       ENDDO
```

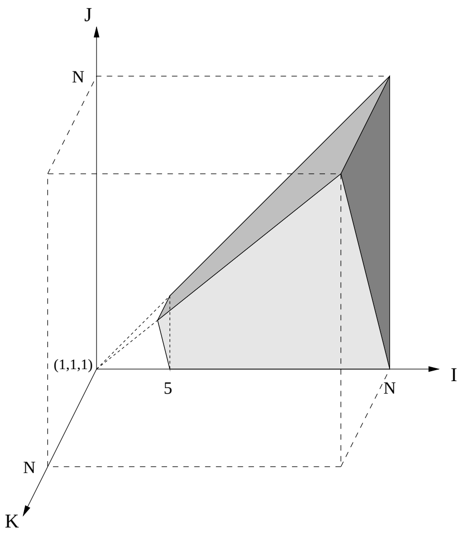


Figure 3.2: An example of a loop nest and its geometrical representation.

Example 3.1 Consider the loop nest shown on the left-hand side of Figure 3.2. The statements of the loop body are executed a number of times, I, given by

$$I = \sum_{i=5}^{n} \sum_{j=1}^{i} \sum_{k=1}^{j} 1.$$

Starting from the innermost sum, we have:

$$I = \sum_{i=5}^{n} \sum_{j=1}^{i} j.$$

Applying Bernoulli's formula for $p = 1$, we get:

$$I = \sum_{i=5}^{n} \frac{i(i+1)}{2} = \frac{1}{2} \left(\sum_{i=5}^{n} i^2 + \sum_{i=5}^{n} i \right).$$

Applying again Bernoulli's formula, (3.7), and (3.10), we end up with:

$$\begin{aligned} I &= \frac{1}{2} \left(\frac{n(n+1)(2n+1)}{6} - 30 + \frac{n(n+1)}{2} - 10 \right) = \frac{n^3 + 3n^2 + 2n - 120}{6} \\ &= \frac{(n-4)(n^2+7n+30)}{6}. \end{aligned}$$

□

3.3.1 Dealing with Non-Integer Loop Bounds

In the previous section, the lower and upper bounds of the loops, consequently the bounds of the sums, were always integers. However, in many real applications, rational loop bounds can be found. The way that these are treated by compilers may vary. In FORTRAN 77, for instance, the index variable can be either real or integer; the values of the loop bounds can be of any type, but, after evaluation, they are converted to the type of the index variable. In the case of an integer index variable, the conversion takes

```
INTEGER I
INTEGER A
DO I=1,A/4
   (statements)
ENDDO
```

```
INTEGER I
REAL A
DO I=1.25,A/4
   (statements)
ENDDO
```

a) Both loops perform the same number of iterations.

```
INTEGER I
REAL A
DO I=A/4,10
   (statements)
ENDDO
```

```
INTEGER I
REAL A
DO I=A/4+2,12
   (statements)
ENDDO
```

b) If $-8 < \mathtt{A} < -4$ *or* $-4 < \mathtt{A} < 0$ *then the loop on the right-hand side performs one iteration more than the loop on the left-hand side.*

Figure 3.3: Understanding loops with non-integer bounds.

place by omitting the decimal part of the real, i.e. the part to the right of the decimal point. To illustrate this, consider the examples shown in Figure 3.3.

In Figure 3.3.a, both loops perform the same number of iterations since the loop bounds are converted to integers.[5] In Figure 3.3.b, the loop of the right-hand side results by skewing (see Section 2.2.2.1) the loop of the left-hand side by 2; one would expect that both loops always perform the same number of iterations. However, for $-8 < \mathtt{A} < 0$ the quantity `A/4+2` remains positive and, because the conversion is done simply by removing the decimal part of the real value (whenever such a decimal part is non-zero), the loop on the right-hand side performs one iteration more than that on the left-hand side. On the other hand, if `A` was declared as an `INTEGER`, both loops would perform the same number of iterations for all values of `A`.

To avoid ambiguity in the interpretation of non-integer loop bounds by a compiler, and to ensure the development of a consistent algebraic representation, it is essential that, whenever required, non-integer loop bounds are converted to integers; this may be achieved by using the functions *floor*, $\lfloor\ \rfloor$, and *ceiling*, $\lceil\ \rceil$, which, for all real x, are defined as follows:

- $\lfloor x \rfloor$ = the greatest integer less than or equal to x.
- $\lceil x \rceil$ = the least integer greater than or equal to x.

From the above definitions, the following rules can be easily derived:

$$\lfloor x \rfloor = n \Longleftrightarrow x - 1 < n \le x, \tag{3.15}$$

$$\lceil x \rceil = n \Longleftrightarrow x \le n < x + 1, \tag{3.16}$$

$$\lceil x \rceil - \lfloor x \rfloor = \begin{cases} 0, & \text{if } x \text{ is an integer,} \\ 1, & \text{otherwise.} \end{cases} \tag{3.17}$$

A useful identity to convert ceilings to floors and *vice versa* is the following:

$$\left\lceil \frac{n}{m} \right\rceil = \left\lfloor \frac{n+m-1}{m} \right\rfloor, \quad \text{for } n, m \text{ integers.} \tag{3.18}$$

[5] However, if the index of the loop was a real variable, then the loop on the right-hand side would perform one iteration less for values of `A` which were multiples of 4.

By definition, the quantity $\lfloor n/m \rfloor$ is equal to the quotient of n divided by m, where m and n are positive integers. Therefore, using the notation $n \bmod m$ for the remainder of the division, we get the formula

$$n = m \left\lfloor \frac{n}{m} \right\rfloor + n \bmod m \Longleftrightarrow \left\lfloor \frac{n}{m} \right\rfloor = \frac{n - n \bmod m}{m}, \tag{3.19}$$

or, applying (3.18), we get the equivalent for ceilings:

$$\left\lceil \frac{n}{m} \right\rceil = \frac{n + m - 1 - (n + m - 1) \bmod m}{m}. \tag{3.20}$$

Both (3.19) and (3.20) can be generalised to arbitrary real numbers n, m, with $m \neq 0$ [85, p. 82].

Additional properties of the functions $\lfloor\ \rfloor$, $\lceil\ \rceil$, mod are discussed in [85, Ch. 3]. Note that FORTRAN 90 has been the first revision of this language to incorporate built-in functions conforming to the definitions given in the previous paragraphs; `FLOOR`, `CEILING` and `MODULO`, respectively. The function `MOD`, in FORTRAN 77, may give erroneous results compared to `MODULO`; this is because the former is defined as `MOD(M,N)=N-M*INT(N/M)`, where the function `INT` returns the integer part of the quantity passed as its argument. This coincides with the definition of mod, as given by Equation (3.19), only when its argument is a non-negative value. Further discussion of the differences between various definitions of the integer functions in programming languages may be found in [31].

Using floors and ceilings, and according to the way FORTRAN 77 interprets non-integer bounds, the number of iterations of the loops in Figure 3.3.a can be computed by evaluating the sum

$$\sum_{i=1}^{\lfloor a/4 \rfloor} 1 = \lfloor a/4 \rfloor, \quad \lfloor a/4 \rfloor \geq 1 \Longleftrightarrow a \geq 4.$$

In Figure 3.3.b, for the loop of the left-hand side, the number of iterations is given by

$$\begin{aligned} I_1 &= \sum_{i=\lfloor a/4 \rfloor}^{10} 1 = 10 - \lfloor a/4 \rfloor + 1 = 11 - \lfloor a/4 \rfloor, && \text{if } a \geq 0,\ \lfloor a/4 \rfloor \leq 10, \text{ or} \\ I_1 &= \sum_{i=\lceil a/4 \rceil}^{10} 1 = 10 - \lceil a/4 \rceil + 1 = 11 - \lceil a/4 \rceil, && \text{if } a < 0, \end{aligned}$$

and, for the loop of the right-hand side,

$$\begin{aligned} I_2 &= \sum_{i=\lfloor a/4+2 \rfloor}^{12} 1 = 12 - \lfloor a/4 + 2 \rfloor + 1 = 11 - \lfloor a/4 \rfloor, && \text{if } a \geq -8,\ \lfloor a/4 + 2 \rfloor \leq 12, \text{ or} \\ I_2 &= \sum_{i=\lceil a/4+2 \rceil}^{12} 1 = 12 - \lceil a/4 + 2 \rceil + 1 = 11 - \lceil a/4 \rceil, && \text{if } a < -8. \end{aligned}$$

Comparing I_1 and I_2 (that is, the summations corresponding to the two loops shown in Figure 3.3.b), we observe that they differ when $-8 \leq a < 0$, and the difference is given by

$$I_1 - I_2 = (11 - \lceil a/4 \rceil) - (11 - \lfloor a/4 \rfloor) = \lfloor a/4 \rfloor - \lceil a/4 \rceil,$$

which, according to (3.17), is 0 when $a/4$ is an integer and -1 otherwise; hence, for $a \in \{-1, -2, -3, -5, -6, -7\}$ the two loops in Figure 3.3.b do not perform the same number of iterations. To avoid phenomena like this in practice, the way that the loop bounds are to be interpreted should be declared explicitly.

3.3.1.1 Dealing with Non-Unit Stride

In real programs, floors and ceilings in loop bounds may also arise after applying unimodular transformations (see Section 2.2.3, or recall the example in Figure 2.5), loop generation techniques, i.e. techniques which attempt to reconstruct a loop structure to minimise overheads [7], or when partitioning loops (a case which is discussed in Chapter 4). When counting loop iterations, floors and/or ceilings may also arise when dealing with loops with non-unit stride. The summation symbol, $\sum$, as defined with a lower and an upper bound, implies a unit increment. In order to deal with loops having a non-unit stride, we need to find an *equivalent* loop (i.e., a loop with exactly the same number of iterations). The following lemma provides a solution to this problem.

Lemma 3.1 *The number of iterations of a loop with lower bound a, upper bound b, and incremental stride s, where $s > 0$, and a, b, s integers, is given by*

$$\sum_{i=0}^{\lfloor \frac{b-a}{s} \rfloor} 1 = \begin{cases} \lfloor \frac{b-a}{s} \rfloor + 1, & \textit{if } b \geq a, \\ 0, & \textit{otherwise.} \end{cases}$$

Proof: If $b < a$ then, by definition, no iterations are executed; thus, assuming that $b \geq a$, at least one loop iteration will be executed. The value of the loop index for the first iteration is a, while for the next iterations it will be successively $a+s, a+2s, a+3s, a+4s, \ldots$ Assume that there is a k such that the k-th iteration is executed while the $(k+1)$-th is not; then the number of iterations of the loop is k. In order to compute k we proceed as follows: the value of the loop index for the k-th iteration is $a + (k-1)s$, while the value of the loop index for the $(k+1)$-th iteration would be $a + ks$. Since the latter is not executed we have $a + ks > b$, while for the k-th iteration we have $a + (k-1)s \leq b$. Solving these two inequalities for k, we end up with

$$k > \frac{b-a}{s} \quad \text{and} \quad k \leq \frac{b-a}{s} + 1,$$

but, because of (3.15), these inequalities imply that

$$k = \left\lfloor \frac{b-a}{s} + 1 \right\rfloor = \left\lfloor \frac{b-a}{s} \right\rfloor + 1 = \sum_{i=0}^{\lfloor \frac{b-a}{s} \rfloor} 1.$$

QED

Using the same reasoning for a negative stride, $s < 0$, it can be proved easily that the number of iterations is also given by

$$\sum_{i=0}^{\lfloor \frac{b-a}{s} \rfloor} 1 = \sum_{i=0}^{\lfloor \frac{a-b}{-s} \rfloor} 1.$$

However, the most important consequence of Lemma 3.1 is that it establishes a one-to-one correspondence between the sequences $a, a+s, \ldots, a+\lfloor \frac{b-a}{s} \rfloor s$ and $0, 1, \ldots, \lfloor \frac{b-a}{s} \rfloor$. Based on this result, it becomes possible to evaluate sums whose index is present in the summand and takes its values from a sequence of integers with an incremental stride s, $s > 1$; this case may arise when counting the iterations of a double loop nest where the outer loop has a non-unit stride while the bounds of the inner loop are a function of the index of the outer loop. Thus, assuming that the notation

$$\sum_{i=a}^{b,s}, \quad \text{where } s \neq 0,$$

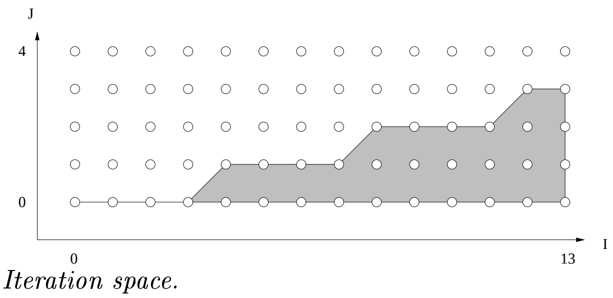
denotes a sum where i successively takes the values $a, a+s, a+2s, \ldots$ until, for some i, $i > b$, then, applying the commutative law, we get

$$\sum_{i=a}^{b,s} i^p = \sum_{i=0}^{\lfloor \frac{b-a}{s} \rfloor} (a+is)^p, \quad p \geq 0, p \text{ integer},$$

where the expression on the right-hand side can be evaluated as a polynomial.

```
C**  N positive integer
       DO I=0,N
          DO J=0,FLOOR(I/4)
             (statements)
          ENDDO
       ENDDO
```

a) Code.



b) Iteration space.

Figure 3.4: An example of a loop nest with an integer function in a loop bound.

3.3.1.2 Summations over Integer Functions

Sums over expressions involving floors and/or ceilings can be evaluated easily; as long as the index of the sum is not contained inside a floor and/or ceiling, or can be moved out of a floor and/or ceiling (note that, for any integer n, $\lfloor n+x \rfloor = n + \lfloor x \rfloor$ and $\lceil n+x \rceil = n + \lceil x \rceil$), these integer functions can be treated as symbolic constants. However, the possibility of having sums of the form

$$\sum_i \lfloor \frac{ai+b}{c} \rfloor \quad \text{or} \quad \sum_i \lceil \frac{ai+b}{c} \rceil, \quad a, b, c \text{ integers, } c \neq 0,$$

which cannot be analysed using the methods described so far, has directed researchers towards the development of special techniques for their evaluation.

Nadia Tawbi [209, Sec. II.6] suggests that the relative error may be kept small by approximating these integer functions as follows:

$$\lfloor \frac{ai+b}{c} \rfloor \simeq \frac{ai+b}{c} - \overline{E}_f, \quad \text{and} \quad \lceil \frac{ai+b}{c} \rceil \simeq \frac{ai+b}{c} + \overline{E}_c,$$

where $\overline{E}_f$, $\overline{E}_c$ are estimates for the absolute value of the difference between the rational quantity $(ai+b)/c$ and its floor or ceiling, respectively. Based on Equations (3.19) and (3.20), Tawbi presents an example where $\overline{E}_f$ and $\overline{E}_c$ are computed as the average of the expected values for this difference. In her example, the relative error is small; however, no formal justification is given, nor is the general case discussed.

William Pugh [185] presents a more complete description, distinguishing between three different approaches to the problem, namely: *compute symbolic answers*, *compute approximate answers*, and *splinter the problem*. Rather hastily, it is concluded that these sums cannot be evaluated symbolically. Different ways of deriving approximations for the values of floors or ceilings are suggested; these can be computed by using an upper or a lower bound, or the average between the two of them. Finally, splintering refers to breaking the problem into disjoint subcases, each subcase having no integer functions.

To illustrate the above, we consider the following example:

Example 3.2 Evaluate the expression

$$I = \sum_{i=0}^{n} \sum_{j=0}^{\lfloor i/4 \rfloor} 1, \quad n \geq 0,$$

which computes the number of times the statements in the body of the loop nest shown in Figure 3.4.a are executed.

First, we attempt to compute a symbolic answer. Starting with the evaluation of the inner sum we have

$$I = \sum_{i=0}^{n} (\lfloor i/4 \rfloor + 1) = \sum_{i=0}^{n} \lfloor i/4 \rfloor + n + 1.$$

Based on what has been discussed so far, it is not possible to analyse this expression further. Thus, two options are available: either approximate the value of $\lfloor i/4 \rfloor$ or split the problem by considering all possible forms of i with respect to the remainder left when divided by 4 (i.e., i modulo 4).

Approximating the value of $\lfloor i/4 \rfloor$ can be done in three different ways. Based on Equation (3.15), an upper bound, U, and a lower bound, L, can be computed; the average of these two values, A, can be used as a potentially better approximation. Thus,

$$U = i/4, \quad L = (i-3)/4, \quad A = (U+L)/2 = (2i-3)/8.$$

In each case, respectively, the computation of I yields the following results:

$$\begin{aligned}
\text{Using } U, \quad I &= \sum_{i=0}^{n} (i/4) + n + 1 = (n^2 + 9n + 8)/8. \\
\text{Using } L, \quad I &= \sum_{i=0}^{n} ((i-3)/4) + n + 1 = (n^2 + 3n + 2)/8. \\
\text{Using } A, \quad I &= \sum_{i=0}^{n} ((2i-3)/8) + n + 1 = (n^2 + 6n + 5)/8.
\end{aligned}$$

Since the computations are based on approximate values of $\lfloor i/4 \rfloor$, which may be non-integer, the value of I may also be non-integer; whenever this is the case, the value of I can be rounded to the nearest integer. Moreover, the value of I resulting from using the average approximation, A (Tawbi's approach), coincides with the value of I computed as the average of the values resulting from using the upper bound U and the lower bound L; in the particular example, this is a consequence of the linear properties of the approximated expression and it is not expected to be always true (consider, for instance, the approximation of $\lfloor i/4 \rfloor^2$). Finally, note that, in order to get an upper or lower bound for I, floors and/or ceilings should be approximated by either their upper or lower bound, depending on their sign; in this example, the positive sign for $\lfloor i/4 \rfloor$ implies that U results in an upper bound for I, respectively, L results in a lower bound.

Splintering the problem of computing $\lfloor i/4 \rfloor$ would require examination of each possible form for i with respect to the remainder left when i is divided by 4; thus, i can have one of the forms $4k$, $4k+1$, $4k+2$, $4k+3$. In each of these cases, $\lfloor 4k/4 \rfloor = \lfloor (4k+1)/4 \rfloor = \lfloor (4k+2)/4 \rfloor = \lfloor (4k+3)/4 \rfloor = k$. Furthermore, since $0 \le i \le n$, in the first case we have $0 \le 4k \le n$, hence $0 \le k \le \lfloor n/4 \rfloor$; similarly, in the second case $0 \le k \le \lfloor (n-1)/4 \rfloor$, in the third case $0 \le k \le \lfloor (n-2)/4 \rfloor$, and in the fourth case $0 \le k \le \lfloor (n-3)/4 \rfloor$. Since the four cases are disjoint, we have

$$\begin{aligned}
I &= \sum_{i=0}^{n} \lfloor i/4 \rfloor + n + 1 = \sum_{k=0}^{\lfloor n/4 \rfloor} k \; + \sum_{k=0}^{\lfloor (n-1)/4 \rfloor} k \; + \sum_{k=0}^{\lfloor (n-2)/4 \rfloor} k \; + \sum_{k=0}^{\lfloor (n-3)/4 \rfloor} k \; + n + 1 \\
&= \lfloor \frac{n}{4} \rfloor \frac{\lfloor \frac{n}{4} \rfloor + 1}{2} + \lfloor \frac{n-1}{4} \rfloor \frac{\lfloor \frac{n-1}{4} \rfloor + 1}{2} + \lfloor \frac{n-2}{4} \rfloor \frac{\lfloor \frac{n-2}{4} \rfloor + 1}{2} + \lfloor \frac{n-3}{4} \rfloor \frac{\lfloor \frac{n-3}{4} \rfloor + 1}{2} + n + 1.
\end{aligned}$$

Splintering the problem again, and examining the form of n, we have:

$$I = \begin{cases} (n^2 + 6n + 8)/8, & \text{if } 4|n, \\ (n^2 + 6n + 9)/8, & \text{if } 4|(n-1), \\ (n^2 + 6n + 8)/8, & \text{if } 4|(n-2), \\ (n^2 + 6n + 5)/8, & \text{if } 4|(n-3). \end{cases} \tag{3.21}$$

These expressions provide closed forms for computing exactly the number of times the body of the loop nest shown in Figure 3.4 is executed. Comparing these expressions with that obtained using the average approximation, A, for the value of $\lfloor i/4 \rfloor$, it can be seen that the maximum absolute error is $1/2$, which becomes zero if non-integer results are rounded to the nearest integer. However, in general, the error may be significantly high; for instance, evaluating the expression

$$I = \sum_{i=1}^{n} \sum_{j=0}^{10} \sum_{k=0}^{\lfloor j/40 \rfloor} 1, \quad n \ge 0,$$

using the average approximation $A = (2j-39)/80$ for the value of $\lfloor j/40 \rfloor$ returns the approximate result $I = 7n$, while the exact result is $I = 11n$. □

Splintering may be a useful technique for computing exact values of summations in cases where the problem can be split into a small number of subcases, as in Example 3.2. However, it may be totally impractical, when the number of subcases is large, or non-applicable, when symbolic constants are involved. Consider, for instance, an upper bound, such as $\lfloor i/a \rfloor$, where the value of a is not known at compile-time. In this case, the only way to get an exact answer is to find a closed form for symbolic sums such as

$$\sum_{x=0}^{a} \left\lfloor \frac{x}{b} \right\rfloor^n, \text{ or } \sum_{x=0}^{a} \left\lceil \frac{x}{b} \right\rceil^n, \quad \text{for } a, b, n \text{ integers, } a, b > 0,\ n \geq 0.$$

Based on (3.19) or (3.20), respectively, the problem can be transformed to that of evaluating sums of the form

$$\sum_{x=0}^{a} x^m (x \bmod b)^n, \quad \text{for } a, b, m, n \text{ integers, } a, b > 0,\ m, n \geq 0.$$

The following lemma provides a closed formula for dealing with these sums.

Lemma 3.2 *For any positive integers a, b and non-negative integers m, n, the sum*

$$\sum_{x=0}^{a} x^m (x \bmod b)^n, \tag{3.22}$$

is equal to

$$\sum_{x=0}^{a} x^{m+n}, \quad \textit{if } a < b,$$

or,

$$\sum_{i=0}^{\lfloor a/b \rfloor - 1} \sum_{x=0}^{b-1} (x+ib)^m x^n + \sum_{x=0}^{a \bmod b} (x + b\lfloor a/b \rfloor)^m x^n, \quad \textit{if } a \geq b.$$

Proof: If $a < b$, it can be seen that, for all x, $0 \leq x \leq a$, it is the case that $(x \bmod b) = x$ and, therefore, (3.22) becomes

$$\sum_{x=0}^{a} x^{m+n} \tag{3.23}$$

which can be computed as described in Section 3.3.

In the case where $a \geq b$, (3.22) becomes

$$\sum_{x=0}^{b\lfloor a/b \rfloor - 1} x^m (x \bmod b)^n + \sum_{x=b\lfloor a/b \rfloor}^{a} x^m (x \bmod b)^n.$$

The first sum can be analysed as

$$\begin{aligned} \sum_{x=0}^{b\lfloor a/b \rfloor - 1} x^m (x \bmod b)^n &= \sum_{x=0}^{b-1} x^m (x \bmod b)^n + \sum_{x=b}^{2b-1} x^m (x \bmod b)^n + \\ &+ \sum_{x=2b}^{3b-1} x^m (x \bmod b)^n + \ldots + \sum_{x=b(\lfloor \frac{a}{b} \rfloor - 1)}^{b\lfloor \frac{a}{b} \rfloor - 1} x^m (x \bmod b)^n, \end{aligned}$$

which can be rewritten as

$$\sum_{i=0}^{\lfloor a/b \rfloor - 1} \sum_{x=0}^{b-1} (x+ib)^m ((x+ib) \bmod b)^n. \tag{3.24}$$

Since ib is always a multiple of b, we have $(x+ib) \bmod b = x \bmod b$ and, because $0 \le x < b$, $x \bmod b = x$. Therefore, (3.24) becomes

$$\sum_{i=0}^{\lfloor a/b \rfloor - 1} \sum_{x=0}^{b-1} (x+ib)^m x^n,$$

which can be computed as described by (3.7).

It remains to compute

$$\sum_{x=b\lfloor a/b \rfloor}^{a} x^m (x \bmod b)^n,$$

which is equal to

$$\sum_{x=0}^{a-b\lfloor a/b \rfloor} (x + b\lfloor a/b \rfloor)^m ((x + b\lfloor a/b \rfloor) \bmod b)^n.$$

Since $a - b\lfloor a/b \rfloor = a \bmod b < b$, we have $((x + b\lfloor a/b \rfloor) \bmod b) = x$; therefore, the summation becomes

$$\sum_{x=0}^{a \bmod b} (x + b\lfloor a/b \rfloor)^m x^n.$$

Hence, if $a \ge b$, we have derived

$$\sum_{x=0}^{a} x^m (x \bmod b)^n = \sum_{i=0}^{\lfloor a/b \rfloor - 1} \sum_{x=0}^{b-1} (x+ib)^m x^n + \sum_{x=0}^{a \bmod b} (x + b\lfloor a/b \rfloor)^m x^n. \tag{3.25}$$

QED

In order to apply Lemma 3.2 to the summation of Example 3.2, the expression

$$I = \sum_{i=0}^{n} \lfloor i/4 \rfloor + n + 1$$

is transformed to

$$I = \sum_{i=0}^{n} \left(\frac{i - i \bmod 4}{4} \right) + n + 1 = \frac{n(n+1)}{8} - \frac{1}{4} \sum_{i=0}^{n} (i \bmod 4) + n + 1,$$

which, after applying Lemma 3.2 and evaluating the sum (assuming that $n \ge 0$), becomes

$$I = \frac{n^2 + 6n + 8 + 2(n \bmod 4) - (n \bmod 4)^2}{8}.$$

It can be verified that the above expression is equivalent to that given by (3.21), obtained after splintering the problem in Example 3.2.

In the general case, when the lower bound is non-zero, the rules described by Equations (3.8) through (3.14) must be applied before using Lemma 3.2. Furthermore, the commutative law can be used to transform sums of the form

$$\sum_{x=0}^{a} x^m ((x+c) \bmod b)^n, \quad \text{for } a, b, c, m, n \text{ integers, } a, b > 0,\ m, n \ge 0,$$

to the form

$$\sum_{y=c}^{a+c} (y-c)^m (y \bmod b)^n, \tag{3.26}$$

or sums of the form

$$\sum_{x=0}^{a} x^m((cx) \bmod b)^n, \quad \text{for } a,b,c,m,n \text{ integers, } a,b>0,\ m,n \geq 0,$$

to the form

$$\sum_{x=0}^{a} \sum_{y=cx}^{y=cx} x^m (y \bmod b)^n. \tag{3.27}$$

After applying the rules described by Equations (3.8) through (3.14) to both (3.26) and (3.27), the resulting sums can be evaluated according to Lemma 3.2.

3.3.2 Dealing with Constraints

In the examples examined so far, it has been assumed that, for any loop, the lower bound is less than or equal to the upper bound. However, in the general case, with variables whose values may be unknown at compile-time, the evaluation of sums must follow Equation (3.1) strictly (or Equations (3.8) through (3.14) when the sum is applied over a function of its index). In other words, the accurate evaluation of sums must be undertaken according to the constraints that apply; these conditions must be taken into account throughout the whole process of evaluating any summation. In order to illustrate the above, we consider the following example:

Example 3.3 Cosnard and Loi [47] attempt to compute

$$S = \sum_{i=l_1}^{u_1} \sum_{j=l_2+l_1-i}^{u_2} 1.$$

Starting from the innermost sum, we have:

$$S = \begin{cases} \displaystyle\sum_{i=l_1}^{u_1} (u_2 - (l_2 + l_1 - i) + 1), & \text{if } l_2 + l_1 - i \leq u_2 \Longleftrightarrow l_2 + l_1 - u_2 \leq i, \\ 0, & \text{otherwise.} \end{cases}$$

Before evaluating the remaining sum, the condition which has already been stated must be considered. Since it defines a lower bound for i, it must be compared with the lower bound of the sum having i as its index. Thus:

$$S = \begin{cases} \displaystyle\sum_{i=l_1}^{u_1} (i + u_2 - l_2 - l_1 + 1), & \text{if } l_1 \geq l_2 + l_1 - u_2 \Longleftrightarrow u_2 \geq l_2, \\ \displaystyle\sum_{i=l_2+l_1-u_2}^{u_1} (i + u_2 - l_2 - l_1 + 1), & \text{if } l_1 < l_2 + l_1 - u_2 \Longleftrightarrow u_2 < l_2, \\ 0, & \text{otherwise.} \end{cases}$$

The next step, before evaluating each sum, is to ensure that the upper bound is greater than the lower bound. We have:

$$S = \begin{cases} \displaystyle\sum_{i=l_1}^{u_1} (i + u_2 - l_2 - l_1 + 1), & \text{if } (u_2 \geq l_2) \wedge (l_1 \leq u_1), \\ \displaystyle\sum_{i=l_2+l_1-u_2}^{u_1} (i + u_2 - l_2 - l_1 + 1), & \text{if } (u_2 < l_2) \wedge (l_2 + l_1 - u_2 \leq u_1), \\ 0, & \text{otherwise.} \end{cases}$$

Since the polynomials in both summations involve i, the rules developed in Equations (3.8) through (3.14) must be applied. The following cases result:

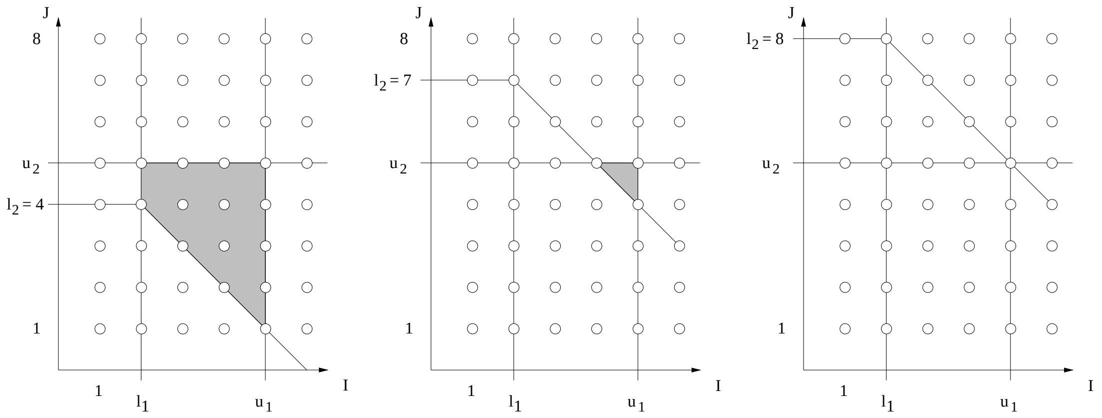


Figure 3.5: Geometrical representation of the summation of Example 3.3 for three different values of l_2.

- If $((l_1 \geq 0) \vee (l_1 < 0 \wedge u_1 > 0)) \wedge (u_2 \geq l_2) \wedge (l_1 \leq u_1)$, then $S = (1 - l_1 + u_1)(2 - l_1 - 2l_2 + u_1 + 2u_2)/2$.
- If $(l_1 < 0) \wedge (u_1 \leq 0) \wedge (u_2 \geq l_2) \wedge (l_1 \leq u_1)$, then $S = (l_1 + u_1 - l_1^2 - u_1^2)/2 + (u_1 - l_1 + 1)(u_2 - l_2 - l_1 + 1)$.
- If $((l_2 + l_1 - u_2 \geq 0) \vee (l_2 + l_1 - u_2 < 0 \wedge u_1 > 0)) \wedge (u_2 < l_2) \wedge (l_2 + l_1 - u_2 \leq u_1)$, then $$S = (2 - 3l_1 + l_1^2 - 3l_2 + 2l_1l_2 + l_2^2 + 3u_1 - 2l_1u_1 - 2l_2u_1 + u_1^2 + 3u_2 - 2l_1u_2 - 2l_2u_2 + 2u_1u_2 + u_2^2)/2.$$
- If $(l_2 + l_1 - u_2 < 0) \wedge (u_1 \leq 0) \wedge (u_2 < l_2) \wedge (l_2 + l_1 - u_2 \leq u_1)$, then $$S = (2 - 3l_1 + l_1^2 - 3l_2 + 2l_1l_2 + l_2^2 + 3u_1 - 2l_1u_1 - 2l_2u_1 - u_1^2 + 3u_2 - 2l_1u_2 - 2l_2u_2 + 2u_1u_2 + u_2^2)/2.$$
- If none of the above conditions is satisfied, then $S = 0$.

Assuming that $l_1 = 2, u_1 = 5, u_2 = 5$, the geometrical representation for three different values of l_2, namely 4, 7, and 8, is given in Figure 3.5. Replacing the values of l_1, u_1, and u_2, in the analytical formulas for S, we deduce that

$$S = \begin{cases} 30 - 4l_2, & \text{if } l_2 \leq 5, \\ (90 - 19l_2 + l_2^2)/2, & \text{if } (l_2 > 5) \wedge (l_2 \leq 8), \\ 0, & \text{otherwise.} \end{cases} \tag{3.28}$$

Therefore, for $l_2 = 4, S = 14$, for $l_2 = 7, S = 3$, and for $l_2 = 8, S = 1$, which can also be verified from Figure 3.5. Instead, for any value of l_2, Cosnard and Loi [47] use the formula $S = 30 - 4l_2$; they observe the error and they point out that, in the general case, this is not bounded. This can be verified in this example, where the absolute error, e, is not bounded, since

$$\lim_{l_2 \to +\infty} e = \lim_{l_2 \to +\infty} |(30 - 4l_2) - 0| = +\infty.$$

□

The previous example emphasises the importance of splitting the evaluation of a summation into different expressions each of which holds under a given set of constraints. This split must be considered as an additive process; for instance, Equation (3.28) may also be written as

$$S = (30 - 4l_2)[l_2 \leq 5] + ((90 - 19l_2 + l_2^2)/2)[5 < l_2 \leq 8],$$

```
DO I=1,N
   DO J=1,I
      IF ((I+J).LE.N) THEN
         (statements)
      ELSE
         (statements)
      ENDIF
   ENDDO
ENDDO
```

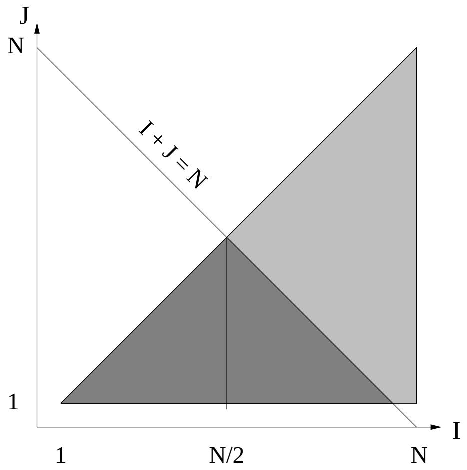


Figure 3.6: An example of a loop nest with conditionals.

where the result of $[exp]$ is 1 if exp is true, or 0 if exp is false, and exp is a true-or-false statement.[6]

Splitting the evaluation of a summation may also be necessary when the functions *min* or *max* are involved in the loop bounds; these functions may appear after applying unimodular transformations (recall the example in Figure 2.5) or loop generation techniques [7]. In this case, before considering any constraints necessary for the evaluation of a summation, the first step must be the elimination of *min* and/or *max* by splitting the problem appropriately. For instance,

$$\sum_{i=\min(a,b)}^{\max(c,d)} 1 = \begin{cases} \sum_{i=a}^{c} 1, & \text{if } (a \le b) \wedge (c \ge d), \\ \sum_{i=a}^{d} 1, & \text{if } (a \le b) \wedge (d > c), \\ \sum_{i=b}^{c} 1, & \text{if } (b < a) \wedge (c \ge d), \\ \sum_{i=b}^{d} 1, & \text{if } (b < a) \wedge (d > c), \end{cases} \tag{3.29}$$

for a, b, c, d integers (if they are not integers, floors and/or ceilings must be employed, as appropriate). Whenever *min* and/or *max* have more than two parameters, they can be evaluated in a similar way; in general, a sum having k *min* and/or *max*, with each of the latter having m_i parameters, $i = 1, 2, \ldots, k$, must be split into $\prod_{i=1}^{k} m_i$ different subcases, each subcase holding when a logical expression consisting of k conditions is true.

Constraints may also arise when counting the number of iterations of statements surrounded by conditionals, such as `IF` ... `THEN` ... `ELSE`, which are in the body of a loop nest and whose execution depends on the value of the loop indices. The following example examines a case of this nature.

Example 3.4 Consider the loop nest shown in Figure 3.6. The shaded area represents the whole iteration space, while the dark shaded area represents that part of the iteration space for which the logical expression of the `IF` statement is true.

[6] This notation is adopted in [85, p. 24]; the idea is attributed to Kenneth Iverson, who introduced it in his programming language APL [114].

The number of times the IF statement is executed is given by

$$I_{tot} = \sum_{i=1}^{n} \sum_{j=1}^{i} 1 = \begin{cases} n(n+1)/2, & \text{if } n \geq 1, \\ 0, & \text{otherwise,} \end{cases}$$

while the number of times the statements following the THEN branch of the conditional are executed is given by

$$I_{then} = \begin{cases} \displaystyle\sum_{i=1}^{n} \sum_{j=1}^{i} 1, & \text{if } i+j \leq n, \\ 0, & \text{otherwise.} \end{cases}$$

Since, whenever the statements following THEN are executed, those following ELSE are not, and *vice versa*, it is apparent that the number of times the latter are executed is given by $I_{else} = I_{tot} - I_{then}$. To evaluate I_{then} we have to consider first the condition $j \leq n - i$, which defines an upper bound for j. Therefore, comparing this value with the existing upper bound i, we have:

$$\begin{aligned} I_{then} &= \begin{cases} \displaystyle\sum_{i=1}^{n} \sum_{j=1}^{i} 1, & \text{if } i \leq n-i, \\ \displaystyle\sum_{i=1}^{n} \sum_{j=1}^{n-i} 1, & \text{if } i > n-i, \\ 0, & \text{otherwise,} \end{cases} \\ &= \begin{cases} \displaystyle\sum_{i=1}^{n} i, & \text{if } (i \leq n-i) \wedge (i \geq 1), \\ \displaystyle\sum_{i=1}^{n} (n-i), & \text{if } (i > n-i) \wedge (n-i \geq 1), \\ 0, & \text{otherwise,} \end{cases} \\ &= \begin{cases} \displaystyle\sum_{i=1}^{n} i, & \text{if } 1 \leq i \leq n/2, \\ \displaystyle\sum_{i=1}^{n} (n-i), & \text{if } n/2 < i \leq n-1, \\ 0, & \text{otherwise.} \end{cases} \end{aligned}$$

Since i takes only integer values, we introduce floors to ensure that $n/2$ is always an integer and, comparing the restrictions of the right-hand side with the bounds of the sum, we get

$$\begin{aligned} I_{then} &= \begin{cases} \displaystyle\sum_{i=1}^{\lfloor n/2 \rfloor} i, & \text{if } \lfloor n/2 \rfloor \geq n, \\ \displaystyle\sum_{i=1}^{n} i, & \text{if } \lfloor n/2 \rfloor < n, \\ \displaystyle\sum_{i=\lfloor n/2 \rfloor + 1}^{n-1} (n-i), & \text{if } \lfloor n/2 \rfloor \geq 0, \\ \displaystyle\sum_{i=1}^{n-1} (n-i), & \text{if } \lfloor n/2 \rfloor < 0, \\ 0, & \text{otherwise,} \end{cases} \\ &= \begin{cases} \lfloor n/2 \rfloor (\lfloor n/2 \rfloor + 1)/2, & \text{if } n \geq 2, \\ n(n-1)/2 - n\lfloor n/2 \rfloor + \lfloor n/2 \rfloor (\lfloor n/2 \rfloor + 1)/2, & \text{if } n \geq 3, \\ 0, & \text{otherwise.} \end{cases} \end{aligned}$$

At this point we observe that, although the above equation results in three different expressions for the value of I_{then}, the first two become true for overlapping values of n, i.e., $n \geq 3$. These two expressions can be simplified, as follows:[7]

$$
\begin{aligned}
I_{then} &= \begin{cases} \lfloor n/2 \rfloor (\lfloor n/2 \rfloor + 1) + n(n-1)/2 - n\lfloor n/2 \rfloor, & \text{if } n \geq 3, \\ 1, & \text{if } n = 2, \\ 0, & \text{otherwise}, \end{cases} \\
&= \begin{cases} n^2/4, & \text{if } (n \geq 2) \wedge (2 \mid n), \\ (n^2-1)/4, & \text{if } (n \geq 3) \wedge (2 \nmid n), \\ 0, & \text{otherwise}. \end{cases}
\end{aligned}
$$

□

The same loop nest, for $n = 2m$, has been examined by Haghighat and Polychronopoulos [93] and subsequently by Pugh [185]. Since, in their example, the upper bound of the outer loop is always a multiple of two, they avoid using floors; their result corresponds with the first case of the results in our example (i.e. when $2 \mid n$). Different approaches are also followed; Haghighat and Polychronopoulos base their analysis on a rather complex framework of rules which handle in a unified way the evaluation of summations alongside the corresponding constraints, while Pugh makes use of Presburger formulas. The former approach provides a clear notation for representing the loop nests, but the evaluation of the summations and the resulting expressions are complex; the latter approach follows a more abstract representation, which permits easier evaluation, but at the expense of a more sophisticated (and rather incompletely described) framework. Our approach provides a compromise by considering the evaluation of summations and any corresponding constraints at one and the same time.

In order to provide a concrete way of representing a loop nest containing conditional statements by a summation, it is helpful to recall Equation (3.29). Since an `IF` statement adds, by definition, a constraint to the evaluation of a summation, the reverse process can also be followed; thus, these constraints can be replaced, appropriately, by a `MIN` or a `MAX` function within the bounds of the sum corresponding to the first loop which precedes the `IF` statement in the program code and whose index is part of the logical expression of the `IF`. However, this conversion can be performed only in the case where the `IF` statement consists of conditions connected by a logical *and* ($\wedge$). Whenever a logical *or* ($\vee$) occurs, de Morgan's law ($A \vee B = \neg(\neg A \wedge \neg B)$) can be used to eliminate it. Note that any resulting negations can be removed; for instance, assuming that A corresponds to $i > l$, then $\neg A$ is equivalent to $i \leq l$. Similarly, a conditional such as

```
IF NOT(logical.expression) THEN
   (statements.1)
ELSE
   (statements.2)
ENDIF
```

can be rewritten as

```
IF (logical.expression) THEN
   (statements.2)
ELSE
   (statements.1)
ENDIF
```

[7] This reduction is associated with a specific property of this loop nest, which is considered in the next chapter; in particular, each of the first two expressions results from evaluating a sum over those values of i which render the loop nest *canonical* (see Definition 4.1). In this context, the code is examined again in Section 4.3.3. Geometrically, as can be seen in Figure 3.6, each sum corresponds to a right-angled triangle; considering the dark shaded triangle (which represents that part of the iteration space for which the logical expression of the `IF` statement is true), the two right-angled triangles result by drawing the perpendicular from the vertex, facing the side corresponding to the `I` axis, to this side.

```
DO I=1,N
   DO J=1,I
      IF (I.GT.10) THEN
         (statements.1)
      ELSE
         DO K=1,I
            IF ((J+I.LT.N).OR.(K+I.GT.N)) THEN
               (statements.2)
            ...
```

Figure 3.7: An example of a loop nest with conditionals.

An alternative way of dealing with logical expressions connected by $\vee$ is to recognise that a summation alongside any logical conditions returns a set of iteration points (cf. the definition of a polytope in Section 3.1). Thus, for a given summation, assuming that the conditions A, B return the sets A', B', respectively, $A \wedge B$ returns $A' \cap B'$, and $A \vee B$ returns $A' \cup B' = A' + B' - A' \cap B'$; the right-hand side of the latter equation indicates how to evaluate a summation to which a constraint of the type $A \vee B$ is applied.

In order to illustrate this, consider the code fragment shown in Figure 3.7. The number of times, I_1, the statements labelled as `statements.1` are executed, is given by

$$I_1 = \sum_{i=max(1,10)}^{n} \sum_{j=1}^{i} 1,$$

while the number of times, I_2, the statements labelled as `statements.2` are executed is given by

$$I_2 = \left(\sum_{i=1}^{\min(n,10)} \sum_{j=1}^{\min(i,n-i)} \sum_{k=1}^{i} 1 \right) + \left(\sum_{i=1}^{\min(n,10)} \sum_{j=1}^{n} \sum_{k=\max(1,n-i)}^{i} 1 \right) - \left(\sum_{i=1}^{\min(n,10)} \sum_{j=1}^{\min(i,n-i)} \sum_{k=\max(1,n-i)}^{i} 1 \right).$$

3.4 An Algorithm for Counting Loop Iterations

Section 3.3 describes the techniques needed for counting, symbolically, the number of times a statement located in the body of a loop nest is executed. An algorithm epitomising this entire procedure is shown in Figure 3.8. The algorithm accepts as its input a set of loops and/or `IF` statements (which are ordered in the set according to their ordering in the program) surrounding a statement, say `s`; it is assumed that the loops are of the `DO` ... `ENDDO` type and that the body of the corresponding loop nest contains no `GOTO` statements. After building an internal representation for the loops and/or `IF`s surrounding `s` (this is done by the first two *for* loops in the algorithm), the algorithm gradually evaluates each sum, from the innermost to the outermost, for every different outcome (an outcome is defined as the expression which returns the number of times `s` is executed when certain constraints hold); outcomes may be added or removed as appropriate. When all sums have been evaluated, the algorithm returns the resulting expressions as well as the constraints under which each expression holds.

Clearly, the execution time of the algorithm is dominated by the third part, i.e., the *for* and the *repeat* loops. The *for* loop is executed a number of times equal to the number of nested Σ's involved in the summation; their number corresponds to the number of loops surrounding `s`. The execution of the *repeat* loop depends on the number of different outcomes, which, in turn, depends on the number of variables (whose values are unknown at compile-time) that are found in the bounds of the loops and/or `IF`s surrounding `s`. In the worst-case, given that each new constraint (added, for instance, due to the presence of a symbolic variable) may double the number of possible non-zero outcomes (cf. Example 3.3 or Example 3.4), this number may grow exponentially. However, in practice, the most likely case is that a loop has a symbolic variable in only one bound [92]; given also that loop nests of depth greater than 3 are rarely encountered, no significant aberrations in the behaviour of the algorithm are expected.

Algorithm Counting_Loop_Iterations(`s`, L)
  INPUT: A statement, `s`, and an ordered set, L, of loops and `IF`s surrounding `s`
  OUTPUT: A set of symbolic expressions, S, and constraints, C

$n := 0$; $m := 1$; $s_{10} := 1$; $c_1 \leftarrow \varnothing$
/⋆ *n is the number of sums, m is the number of outcomes* ⋆/

for every loop surrounding `s`, scanning from the innermost to the outermost
  create a sum $\Sigma(i, l, u, s, s_{1n})$
  $n := n + 1$
  $s_{1n} \leftarrow \Sigma(i, l, u, s, s_{1,n-1})$
endfor

for every `IF` surrounding `s`, scanning from the innermost to the outermost
  if there exist conditions in the `IF` depending on the index of a surrounding loop
    while there exist conditions in the logical expression not connected by $\wedge$
      apply deMorgan's Law to transform the logical expression
    endwhile
    for every loop surrounding the `IF`, scanning from innermost to outermost
      if the loop index is part of the logical expression of the `IF`
        assign conditions to the bounds of the corresponding sum
        remove this condition from the logical expression
      endif
    endfor
  endif
endfor

for $j := 1$ **to** n
  $i := 1$
  repeat
    get the sum, s_{ij}, and the corresponding constraint, c_i
    if the index of the sum, $s_{ij}.i$, occurs in a condition of the constraint c_i
      add *min* and/or *max* to the bounds, as in (3.29)
      remove the corresponding conditions
    endif
    eliminate all *min*, *max* from the bounds of the current sum, s_{ij}
      (possibly, increase m and add new sums/constraints)
    ensure the correctness of the evaluation with respect to the upper and lower bounds
      (possibly, increase m and add new sums/constraints)
    evaluate sums
    $i := i + 1$
  until $i > m$
  simplify constraints (possibly, decrease m and remove some sums/constraints)
endfor

return($\{s_{11}, s_{21}, \ldots, s_{m1}\}$,$\{c_1, c_2, \ldots, c_m\}$)

Figure 3.8: An algorithm for counting loop iterations.

3.5 Practical Considerations

The algorithm presented in Section 3.4, in conjunction with a strategy for counting the floating point operations performed by each statement, such as that adopted by [84, p. 19], can provide a parallelising compiler with an effective symbolic performance estimator. Such an estimator may be used, for instance, to prevent the parallelisation of loop nests for which the resulting overheads due to parallelisation are likely to outweigh the gains from parallel execution (a problem that has been highlighted in Section 2.5). Assuming that, for any parallelisable loop nest, an estimate for the *minimum grain size* of the computation (i.e., the number of associated floating point operations required to be executed in each processor in order to achieve some performance gains on a particular architecture) is known, then the compiler can introduce the appropriate conditional statement to determine whether this number is exceeded; if so, it is worth executing the loop nest in parallel.

In order to illustrate this, consider the code fragment shown in Figure 3.9.a, which implements the addition of two upper triangular matrices. Assuming that `N` is not known at compile-time, the transformed code, shown in Figure 3.9.b, provides a way of executing the loop nest in parallel only if performance is likely to improve, that is, the work present in the loop nest (expressed, for instance, in floating points operations, `FLOPS`, where one floating point operation is associated with the work of the assignment statement in the loop body) exceeds a predefined minimum (`MIN_GRAIN`). A value for the latter, on a given architecture, can be obtained experimentally. Indicatively, the code shown in Figure 3.9.a was executed on the KSR1 for varying values of `N` and number of processors. The resulting performance is depicted graphically in Figure 3.9.c. Performance gains appear only for values of `N` greater than 140; thus, on the KSR1, a value of $140 \cdot 141/2 = 9870$ can be chosen for `MIN_GRAIN`.[8]

Essentially, the experiments run on the KSR1 using the code shown in Figure 3.9.a aim to determine the overhead due to parallel start-up (see Section 1.4.3.2); based on the latter, the introduction of the conditional in the code of Figure 3.9.b reflects a simplified model for ensuring that any gains from parallelisation of the loop nest outweigh this particular overhead. However, the execution time of the parallel program may also be affected by other sources of overhead. For instance, in the transformed code in Figure 3.9.b, it has been assumed implicitly that each processor is assigned the same, or nearly the same, amount of work. If this were not the case, an overhead due to load imbalance would arise; this may have a more significant impact on performance than the overhead due to parallel start-up. Distributing loop iterations across the processors in as balanced a fashion as possible can also be achieved using the algorithm of Figure 3.8; this forms the subject of Chapter 4.

3.6 Concluding Remarks

This chapter has described a methodology for computing the number of times that a statement surrounded by a number of loops is executed; this number may be given in terms of the symbolic variables involved in the loop bounds. The methodology follows an algebraic representation based on the evaluation of nested sums. The manipulation of the latter is achieved in a fashion which is closely related to the way loop nests actually work in practice; hence, the algorithmic aspects of this process have been stressed. Emphasis has also been placed on computing exact expressions in cases where previous research has failed, that is, whenever integer functions are specified in the loop bounds.

Although this methodology, incorporated in a compiler, may be used to estimate performance, it may also provide a tool for reasoning when developing strategies aiming to minimise particular overheads; this is demonstrated in the next chapter.

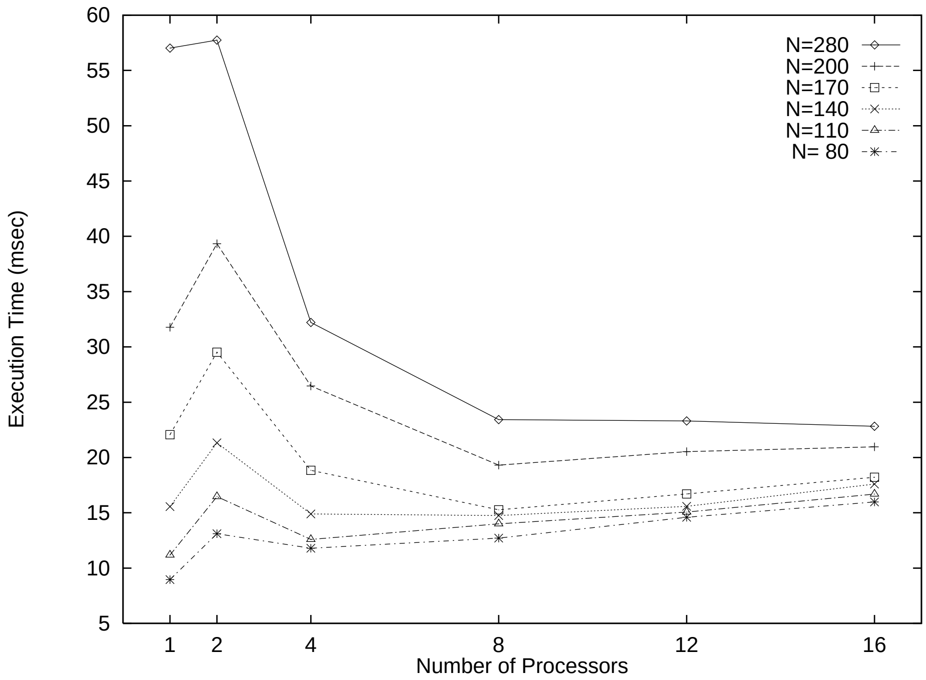
[8] A close observation of Figure 3.9.c shows that for `N` = 140 performance gains (although relatively small) arise only when using four or eight processors; even for higher values of `N`, some performance degradation occurs when running on two processors. As a result, a more accurate estimate must be based on different values for `MIN_GRAIN` depending on the number of processors used. However, it is questionable whether such an elaborate analysis is worth carrying out, given that the main objective for the compiler is rather an indication than an accurate prediction (which may be difficult to obtain, since the latter may be susceptible to possible execution time deviations during the experimental measurements).

```
DOALL J=1,N
   DOALL I=1,J
      A(I,J)=B(I,J)+C(I,J)
   ENDDO
ENDDO
```

a) Original code.

```
FLOPS=N*(N+1)/2
IF (FLOPS.GT.MIN_GRAIN) THEN
   DOALL J=1,N
      DOALL I=1,J
         A(I,J)=B(I,J)+C(I,J)
      ENDDO
   ENDDO
ELSE
   DO J=1,N
      DO I=1,J
         A(I,J)=B(I,J)+C(I,J)
      ENDDO
   ENDDO
ENDIF
```

b) Transformed code.



c) Execution time of the original code for different values of `N` *on the KSR1.*

Figure 3.9: Studying the overhead due to parallel start-up.

Chapter 4

Load Balancing

4.1 Introduction

The previous chapter presented a methodology for symbolically computing the iterations associated with any statement in the body of a loop nest. In this chapter, this methodology forms the basis for development of a strategy for mapping loop nests onto processors in such a way that load imbalance is minimised. Recall the classification of overheads given in Section 1.4.3.2; load imbalance refers to the overhead caused by poor distribution of parallelised *computational work* (hereinafter called *workload* or simply *work*) amongst processors. Reducing the overhead due to load imbalance requires distribution of the workload as evenly as possible.

Assuming that the workload, W, is distributed amongst p processors, and processor i, $0 \leq i < p$, is assigned a workload equal to W_i, then if, for all i,

$$W_i = \frac{W}{p},$$

there exists a *perfect load balance.* Whenever there is a processor i for which the difference $L_i = W_i - W/p$ is non-zero, then the workload assigned to processor i exhibits an *imbalance* equal to L_i; positive values of L_i indicate that processor i is assigned more workload than the average share and the processor may be considered 'overloaded', while negative values of L_i indicate that the particular processor is assigned less workload than the average share. The processor with the highest value of imbalance (which, consequently, is the most heavily loaded) determines the overall overhead of the parallel computation due to load imbalance. Thus, a given distribution of the workload W amongst p processors exhibits a *load imbalance*, L, given by

$$L = \max_{0 \leq i < p}(L_i) = \max_{0 \leq i < p}(W_i - \frac{W}{p}) = W_{max} - \frac{W}{p}, \tag{4.1}$$

where W_{max} represents the workload assigned to the processor with the highest value of imbalance. Reducing the overhead due to load imbalance is equivalent to finding $W_0, W_1, \ldots, W_{p-1}$ such that L in Equation (4.1) is minimised.[1]

The impact of the overhead due to load imbalance on the overall overhead depends on the value of the remaining overheads, as well as, on the machine used. Recalling Equation (1.1), the value of L, which is a machine-independent metric, is scaled according to machine-dependent characteristics to derive F_{LI}; thus, the same value of L may, in practice, result in a different overhead on different architectures. To assess the impact of load imbalance on performance, it is useful to introduce *relative load imbalance*,

[1] Clearly, $W = \sum_{i=0}^{p-1} W_i$, where W, W_i may be measured, for instance, in terms of floating point operations. Furthermore, assuming that W_i' represents the set of those statements whose workload is W_i, then $W' = \cup_{i=0}^{p-1} W_i'$ and, for any two sets, W_i', W_j', $0 \leq i, j < p$, $i \neq j$, it must be the case that $W_i' \cap W_j' = \varnothing$.

defined as

$$L_R = \frac{L}{W_{max}} = \frac{W_{max} - \frac{W}{p}}{W_{max}} = 1 - \frac{W}{pW_{max}}. \tag{4.2}$$

It can be shown easily that L_R takes values in the interval $[0, 1-1/p]$. Values close to zero denote a small impact on performance and, assuming that load imbalance is the only non-zero overhead (or the most dominant), a close-to-linear speed-up is expected; values close to $1 - 1/p$ denote a highly imbalanced workload distribution, and no significant gains from parallelisation can be expected.

As mentioned in Section 1.4.3.2, when attempting to reduce one source of overhead it is essential to avoid options which may increase other sources of overhead. In Section 2.3.1.1 it is noted that, in certain cases, when mapping loop iterations onto processors in a round-robin fashion (cyclic partitioning), an additional communication overhead, false sharing, may arise; thus, a primary requirement for the development of the methods described in the following sections is to assign as many consecutive loop iterations as possible to the same processor. Furthermore, for scalability, a second requirement is to partition the iterations of the outer loop into chunks of equal, or nearly equal, size (as opposed to the unequal sized chunks used in balanced chunk scheduling).

Based on these requirements, Section 4.2 describes possible ways of partitioning loop nests in the case where each and every loop iteration performs exactly the same amount of work. Subsequent sections examine cases where different loop iterations may perform different amounts of work; Section 4.3 considers the case where this is a result of inner loops, at the same level, whose execution depends on the index of the outer loop, while Section 4.4 considers the general case of having a number of nested loops whose execution depends on the index of the outer loop.

4.2 The Problem of Loop Partitioning

Starting with the problem of partitioning the loop iterations of a single loop, it is useful to recall the general, combinatorial problem of partitioning n things into m groups as equally as possible, where n, m are positive integers; this problem has been analysed in [85, pp. 83–85]. It is observed that the most even distribution would consist of $(n \bmod m)$ groups each containing $\lceil n/m \rceil$ things and $n - (n \bmod m)$ groups each containing $\lfloor n/m \rfloor$ things. Because of Equation (3.17), the difference between the number of elements in any two groups is at most 1.

In the context of loops, the requirements to avoid false sharing and to partition into equal-sized chunks should be respected when splitting the loop iterations. Thus, to distribute the loop iterations of a single loop with lower bound l and upper bound u across p processors, we have to find an analytical expression for $l_k, u_k, 0 \le k < p$, such that

$$\sum_{i=l}^{u} 1 = \sum_{i=l_0}^{u_0} 1 + \sum_{i=l_1}^{u_1} 1 + \ldots + \sum_{i=l_{p-1}}^{u_{p-1}} 1 = \sum_{k=0}^{p-1} \sum_{j=l_k}^{u_k} 1, \tag{4.3}$$

where $u_k = l_{k+1} - 1$ for $0 \le k < p-1$ and $u_{p-1} = u$,[2] and, for all l_k, u_k,

$$\left\lfloor \frac{n}{p} \right\rfloor \le \sum_{j=l_k}^{u_k} 1 \le \left\lceil \frac{n}{p} \right\rceil, \tag{4.4}$$

where $n = u - l + 1 = u_{p-1} - l_0 + 1$. It is not hard to verify that the following three relations satisfy both (4.3) and (4.4):

- **A. Partitioning by decreasing order:**

$$l_k = l + k\lfloor n/p \rfloor + \min(n \bmod p, k), \quad k = 0, 1, 2, \ldots, p-1.$$

[2] This restriction ensures that no two partitions are assigned the same loop iteration; clearly, if S is the set containing all possible values of the index of the outer loop, that is, $S = \{x : x \text{ integer and } l \le x \le u\}$, and S_k, $0 \le k < p$, the set of values assigned to the k-th partition, then, for any two different partitions, say k_1, k_2, it must be the case that $S_{k_1} \cap S_{k_2} = \varnothing$, while, for all partitions, $\bigcup_{k=0}^{p-1} S_k = S$.

In this case the partitions are arranged in *decreasing* order of their number of iterations, i.e. the first ($n \bmod p$) loops have $\lceil n/p \rceil$ iterations, whilst the rest have $\lfloor n/p \rfloor$. In general, the loop with bounds (l_k, u_k), i.e. the k-th partition, $0 \le k < p$, has $\lceil (n-k)/p \rceil$ iterations.

- **B. Partitioning by increasing order:**

$$l_k = l + k\lfloor n/p \rfloor + \max(0, k - p + (n \bmod p)), \quad k = 0, 1, 2, \ldots, p-1.$$

In this case the partitions are arranged in *increasing* order of their number of iterations, i.e. the last ($n \bmod p$) loops have $\lceil n/p \rceil$ iterations, whilst the rest have $\lfloor n/p \rfloor$. In general, the loop with bounds (l_k, u_k), $0 \le k < p$, has $\lfloor (n+k)/p \rfloor$ iterations.

- **C. Partitioning by modular order:**

$$l_k = l + \lfloor kn/p \rfloor, \quad k = 0, 1, 2, \ldots, p-1.$$

In this case, the partitions are not sorted depending on the order of their number of iterations; the loop with bounds (l_k, u_k), $0 \le k < p$, has $\lfloor n(k+1)/p \rfloor - \lfloor nk/p \rfloor$ iterations, i.e. $\lfloor n/p \rfloor$, if $kn \bmod p > n \bmod p$, and $\lceil n/p \rceil$ otherwise.

Clearly, if n is a multiple of p, all three relations reduce to

$$l_k = l + kn/p, \quad k = 0, 1, 2, \ldots, p-1.$$

In this case, the loop can be partitioned into *equal partitions*, each consisting of n/p iterations.[3]
The results of the preceding analysis can be summarised in the following lemma:

Lemma 4.1 *Any single loop can be partitioned into p partitions with each partition having a number of iterations which is either the same for all partitions, if the total number of iterations of the loop is a multiple of p, or differs by at most 1 from that of any other partition, otherwise.*

The proof can be drawn immediately from the discussion of the previous paragraphs.
In the context of KSR FORTRAN parallel programming, applying partitioning by decreasing order to the simple parallel loop shown in Figure 4.1.a would result in the transformed code shown in Figure 4.1.b. Each one of the p processors available executes the code enclosed within the `PARALLEL REGION` and `END PARALLEL REGION` directives, but operating upon different data values; this is achieved by means of a library function, `IPR_MID()`, which returns an integer between 0 and $p-1$, depending on which processor is executing the code. Note also, that the variables `K`, `LK`, `UK`, `I` are declared as `PRIVATE`, that is, each processor has its own copy of the variable.

The techniques of partitioning described so far can also be applied to the outer loop of a nest of loops, whether perfectly nested or not. Assuming that the bounds of the inner loops are not a function of the index of the outer loop, nor are there any conditional statements in the loop body whose execution depends on the value of the index of the outer loop (and, as a consequence, each iteration of the outer loop performs the same amount of work), then perfect load balance may be achieved. If this is not possible (because the number of iterations of the outer loop is not a multiple of the number of processors), then, in the case of a perfectly nested loop, partitioning may be applied to the iterations of more than one loop at the same time.

[3] Several other partitioning schemes satisfying both (4.3) and (4.4) may be found; the three schemes presented are preferred because they provide some order and are characterised by simple expressions. Additional partitioning schemes may be derived by relaxing the requirement for a lower bound in (4.4), which does not affect the value of load imbalance as given by (4.1); such a scheme, where $l_k = l + k\lceil n/p \rceil$, $0 \le k < p$, is described, for instance, in [39] and implemented, albeit in a slightly different context, by MARS, a parallelising compiler developed jointly by the University of Manchester and IRISA, Rennes (France) [29]. However, in this case, all processors but one are assigned the highest possible number of iterations, $\lceil n/p \rceil$, and, consequently, the last processor may be underutilised; in practice, this option may increase the chances of slowing down a program's performance.

```
       DOALL I=L,U
          (statements)
       ENDDO
```

a) Unpartitioned loop structure.

```
       N=U-L+1
C*KSR* PARALLEL REGION(NUMTHREADS=P, PRIVATE=K,LK,UK,I)
       K=IPR_MID()
       LK=L+K*FLOOR(N/P)+MIN(MOD(N,P),K)
       UK=L+(K+1)*FLOOR(N/P)+MIN(MOD(N,P),K+1)-1
       DO I=LK,UK
          (statements)
       ENDDO
C*KSR* END PARALLEL REGION
```

b) Transformation for loop partitioning.

Figure 4.1: An example of loop partitioning.

```
       DOALL I=L1,U1
          DOALL J=L2,U2
             (statements)
          ENDDO
       ENDDO
```

a) Unpartitioned loop structure.

```
C ---- P1 and P2 are given; apparently, P=P1*P2 ----
       N=U1-L1+1
       M=U2-L2+1
C*KSR* PARALLEL REGION(NUMTHREADS=P,
C*KSR*&                PRIVATE=K,K1,K2,LK1,UK1,LK2,UK2,I,J)
       K=IPR_MID()
       K1=FLOOR(K/P2)
       K2=MOD(K,P2)
       LK1=L1+K1*FLOOR(N/P1)+MIN(MOD(N,P1),K1)
       UK1=L1+(K1+1)*FLOOR(N/P1)+MIN(MOD(N,P1),K1+1)-1
       LK2=L2+K2*FLOOR(M/P2)+MIN(MOD(M,P2),K2)
       UK2=L2+(K2+1)*FLOOR(M/P2)+MIN(MOD(M,P2),K2+1)-1
       DO I=LK1,UK1
          DO J=LK2,UK2
             (statements)
          ENDDO
       ENDDO
C*KSR* END PARALLEL REGION
```

b) Transformation for loop partitioning.

Figure 4.2: An example of partitioning along more than one loop at a time.

To illustrate this, consider the double loop shown in Figure 4.2.a. Assume that the number of iterations of the outer loop is $n =$ `U1-L1+1` $= 20$, of the inner loop is $m =$ `U2-L2+1` $= 15$, and there are $p = 6$ processors available. Partitioning along the index of the outer loop would result in two processors performing the statements of the loop body 60 times each, and four processors performing them 45 times each, thus leading to a relative load imbalance $L_R = 1/6$; instead, partitioning the outer loop into two partitions and the inner loop into three partitions results in a perfect load balance. In the general case, partitioning the outer loop into p_1 partitions and the inner loop into p_2 partitions, where $p = p_1p_2$, would result in the transformed code shown in Figure 4.2.b; two questions immediately arise: 'When is partitioning along several loops preferable?' and 'How many processors should be assigned to each loop?' A brief analysis for a double loop follows.

Assume that the number of iterations of the outer loop is n, of the inner loop is m, and there are p processors. Clearly, if p divides n then partitioning along the index of the outer loop leads to a perfect load balance. If p does not divide n but p divides nm, then there exist p_1, p_2, where $p = p_1p_2$, such that $p_1|n$ and $p_2|m$;[4] the values of p_1 and p_2 may be computed using either $p_1 = \gcd(n,p)$, $p_2 = p/p_1$ or $p_2 = \gcd(m,p)$, $p_1 = p/p_2$. In the more general case, where $p \nmid mn$, the problem becomes that of finding p_1, p_2, $p = p_1p_2$, such that $\lceil \frac{n}{p_1} \rceil \lceil \frac{m}{p_2} \rceil$ is minimised (this expression can be easily derived from (4.1)). Solving this problem may not be feasible; alternatively, applying loop coalescing (see Section 2.2.2.4) before partitioning may be considered. Since the inequality $\lceil nm/(p_1p_2) \rceil \leq \lceil n/p_1 \rceil \lceil m/p_2 \rceil$ always holds,[5] the transformed coalesced loop is expected to result in a smaller load imbalance than the original loop nest.

4.2.1 Dealing with Conditionals

The presence, in the loop body, of conditionals whose execution does not depend on the value of the index of the outer loop does not affect the load imbalance resulting from the partitioning schemes described so far; each iteration of the outer loop still performs the same amount of work and, consequently, perfect load balance can be achieved. In the special case where the execution of a conditional depends on a condition involving the index of the outer loop and a constant, then, applying index set splitting (see Section 2.2.2.3) prior to partitioning, the conditional may be removed.

To illustrate this, consider the code shown in Figure 4.3.a; splitting the iterations of the outer loop into those rendering the condition of the `IF` statement true and those rendering it false, the conditional can be removed, and the code shown in Figure 4.3.b results.[6] Each of the loops in the transformed code may be partitioned using any of the three partitioning schemes described in Section 4.2; for both loops, if the number of iterations is a multiple of the number of processors, p, then perfect load balance can be achieved. In the general case, let n, m be the number of loop iterations, and W_1, W_2 the amount of work in the body of each loop, respectively; assuming that the same partitioning scheme (either by decreasing or increasing order) has been applied to both loops, then the expected load imbalance is equal to:

$$L = \left\lceil \frac{n}{p} \right\rceil W_1 + \left\lceil \frac{m}{p} \right\rceil W_2 - \frac{nW_1 + mW_2}{p}.$$

However, in the case where $(n \bmod p) + (m \bmod p) \leq p$, the load imbalance can be reduced to

$$L = \max\left(\left\lceil \frac{n}{p} \right\rceil W_1 + \left\lfloor \frac{m}{p} \right\rfloor W_2, \left\lfloor \frac{n}{p} \right\rfloor W_1 + \left\lceil \frac{m}{p} \right\rceil W_2 \right) - \frac{nW_1 + mW_2}{p}.$$

[4] This is a corollary of two basic theorems of number theory: Euclid's First Theorem — *If p is a prime, and $p|mn$, then $p|m$ or $p|n$* — and the Fundamental Theorem of Arithmetic — *Every positive integer greater than 1 can be expressed as a product of primes in one way only* — [96, p. 3].

[5] The proof is straightforward. Let $x = n/p_1 > 0$ and $y = m/p_2 > 0$ and assume $\overline{x} = \lceil x \rceil - x$ and $\overline{y} = \lceil y \rceil - y$; clearly $0 \leq \overline{x}, \overline{y} < 1$. Then, $\overline{x}\,\overline{y} \leq \overline{x} + \overline{y} \leq \lceil y \rceil \overline{x} + \lceil x \rceil \overline{y}$. Hence, $\lceil x \rceil \lceil y \rceil + \overline{x}\,\overline{y} - \lceil y \rceil \overline{x} - \lceil x \rceil \overline{y} \leq \lceil x \rceil \lceil y \rceil \iff (\lceil x \rceil - \overline{x})(\lceil y \rceil - \overline{y}) \leq \lceil x \rceil \lceil y \rceil \iff \lceil xy \rceil \leq \lceil x \rceil \lceil y \rceil$, **QED**.

[6] In this example, it has been implicitly assumed that the values of `L`, `U`, `A` are known at compile-time and that `L≤A<U`; if some of these values were not known, then, in the transformed code, the upper bound of the first loop should have been `MIN(A,U)` and the lower bound of the second loop `MAX(A+1,L)`.

```
DOALL I=L,U
   (statements.1)
   IF (I.GT.A) THEN
      (statements.2)
   ELSE
      (statements.3)
   ENDIF
ENDDO
```

a) Loop with conditional.

```
DOALL I=L,A
   (statements.1)
   (statements.3)
ENDDO
DOALL I=A+1,U
   (statements.1)
   (statements.2)
ENDDO
```

b) After index set splitting.

```
       N=A-L+1
       M=U-A
C*KSR* PARALLEL REGION(NUMTHREADS=P, PRIVATE=I,K,LK,UK)
       K=IPR_MID()
       LK=L+K*FLOOR(N/P)+MIN(MOD(N,P),K)
       UK=L+(K+1)*FLOOR(N/P)+MIN(MOD(N,P),K+1)-1
       DO I=LK,UK
          (statements.1)
          (statements.3)
       ENDDO
       LK=A+1+K*FLOOR(M/P)+MAX(0,K-P+MOD(M,P))
       UK=A+1+(K+1)*FLOOR(M/P)+MAX(0,K+1-P+MOD(M,P))-1
       DO I=LK,UK
          (statements.1)
          (statements.2)
       ENDDO
C*KSR* END PARALLEL REGION
```

c) After loop partitioning.

Figure 4.3: An example of partitioning loops containing conditionals.

This is achieved by partitioning one of the loops by decreasing order and the other one by increasing order. This approach has been followed in the code shown in Figure 4.3.c.

An alternative approach to partitioning the code shown in Figure 4.3.b is to make use of the fact that the two loops can be executed concurrently by assigning to each loop a number of processors which is proportional to the corresponding work. The number of processors for the first loop, p_1, may be either $\lfloor p\frac{nW_1}{nW_1+mW_2}\rfloor$ or $\lceil p\frac{nW_1}{nW_1+mW_2}\rceil$, while the number of processors for the second loop is $p_2 = p - p_1$; the choice of p_1, p_2 should minimise the load imbalance, which, assuming that $p_1, p_2 \neq 0$, is given by

$$L = \max\left(\left\lceil\frac{n}{p_1}\right\rceil W_1, \left\lceil\frac{m}{p_2}\right\rceil W_2\right) - \frac{nW_1+mW_2}{p}.$$

This approach may sometimes lead to a smaller load imbalance [195]; however, its main drawback stems from the parallel start-up overhead caused by having to compute the values of p_1 and p_2. Furthermore, these values are based on estimates for W_1, W_2, which may introduce an additional error. Hence, this approach will not be considered further.

4.3 Partitioning Non-Rectangular Loop Nests of Depth 2

The partitioning schemes described in Section 4.2 result in perfect load balance, or a small value of load imbalance, when each iteration of the outer loop performs the same amount of work; this implies that the index of the outer loop cannot form part of the bounds of an inner loop, nor of an `IF` statement's condition (unless compared to a constant value, when the `IF` may be removed, as shown in Section 4.2.1). A simple example where the former constraint is not applicable is that of a triangular or trapezoidal loop nest (see Figure 3.1 in Section 3.2). Consider, for instance, the loop shown in Figure 3.1.a, i.e., a double loop with the index of the outer loop, i, taking values from 1 to n, and the index of the inner loop, j, taking values from 1 to i. Assuming that the loop nest is mapped onto p processors, and $p|n$, then, applying any of the partitioning schemes described in Section 4.2, a load imbalance, L, equal to

$$L = \frac{n^2(p-1)}{2p^2}W,$$

where W is the amount of work in the loop body, results; the relative load imbalance, L_R, is equal to

$$L_R = \frac{np-n}{2np+p-n} = \frac{1}{1+\frac{1+\frac{1}{n}}{1-\frac{1}{p}}},$$

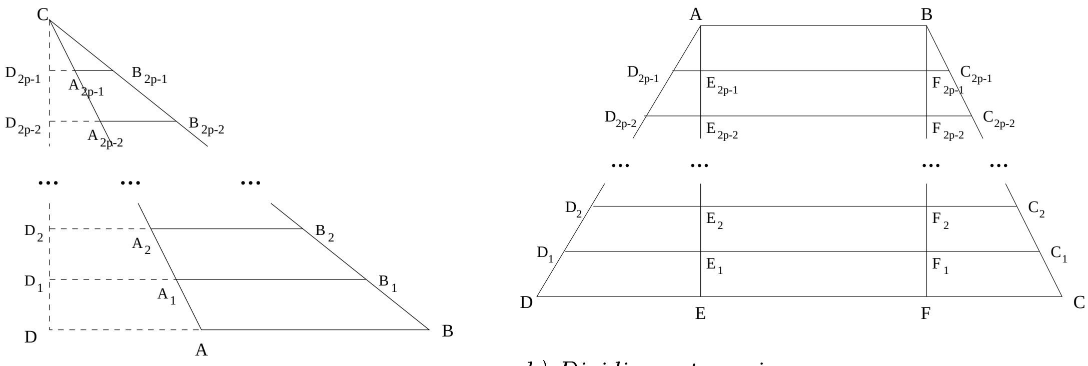
which, for $p, n \geq 2$, has a lower bound of $1/4$. Thus, it is apparent that the partitioning schemes described so far are not sufficient to minimise load imbalance, and that new schemes should be devised.

To illustrate a strategy for a more suitable partitioning scheme, recall the principles of the geometric approach described in Section 3.2. The problem can be expressed as that of finding a way to divide the area of a triangle or a trapezium into p parts of equal area (including dividing the area into more than p parts which are then grouped appropriately); at the same time, one of the sides of the triangle/trapezium must also be divided into segments of equal length, thus conforming to the requirement, mentioned in Section 4.1, for equal, or nearly equal, partitions along the index of the outer loop. It turns out that a solution exists, as shown by the next lemma.

Lemma 4.2 *By drawing lines parallel to one side of any triangle ABC of area α in such a way that these lines cut any other side into $2p$ segments of equal length, it is possible to form, by appropriate pairing, p pieces of area α/p each.*

Proof: Consider the triangle ABC shown in Figure 4.4.a. Let AB be the side of the triangle to which parallel lines are drawn, and assume that they cut BC into $2p$ segments of equal length, i.e.,

$$BB_1 = B_1B_2 = \ldots = B_{2p-2}B_{2p-1} = B_{2p-1}C = \frac{BC}{2p}.$$



a) Dividing a triangle. *b) Dividing a trapezium.*

Figure 4.4: Dividing a triangle/trapezium to form p pieces of equal area.

Then it is known that the parallel lines also cut AC and CD (which is drawn perpendicular to AB) into $2p$ segments of equal length,[7] i.e.,

$$DD_1 = D_1D_2 = \ldots = D_{2p-2}D_{2p-1} = D_{2p-1}C = \frac{CD}{2p}.$$

Because the triangles $A_{2p-1}B_{2p-1}C$, $A_{2p-2}B_{2p-2}C$, $\ldots$, A_1B_1C, ABC are similar,

$$A_{2p-1}B_{2p-1} = \frac{A_{2p-2}B_{2p-2}}{2} = \ldots = \frac{A_1B_1}{2p-1} = \frac{AB}{2p}.$$

Thus, assuming that the area of the triangle $A_{2p-1}B_{2p-1}C$ is $(A_{2p-1}B_{2p-1}C) = \epsilon$,

$$(A_{2p-2}B_{2p-2}C) = 2^2\epsilon, (A_{2p-k}B_{2p-k}C) = k^2\epsilon, 1 \le k \le 2p-1, (ABC) = 4p^2\epsilon.$$

Therefore $(A_{2p-1}B_{2p-1}B_{2p-2}A_{2p-2}) = 3\epsilon$,

$$(A_{2p-k}B_{2p-k}B_{2p-k-1}A_{2p-k-1}) = (2k+1)\epsilon, 1 \le k < 2p-1, (A_1B_1BA) = (4p-1)\epsilon,$$

where $\epsilon = \alpha/(4p^2)$. Considering the set, S, of the $2p$ shapes into which the parallel lines drawn divide the triangle ABC, i.e.,

$$S = \{A_{2p-1}B_{2p-1}C, A_{2p-1}B_{2p-1}B_{2p-2}A_{2p-2}, \ldots, A_1B_1BA\},$$

it can be seen that each of the p pairs consisting of the shape with the k-th largest area and the shape with the k-th smallest area, $1 \le k \le p$, has a total area equal to $(2k+1)\epsilon + (2(2p-k-1)+1)\epsilon = \frac{\alpha}{p}$. **QED**

Lemma 4.2 can be easily extended to any trapezium, with the restriction that the division into equal segments should be drawn along the non-parallel sides of the trapezium; that is, the parallel lines must be drawn parallel to the trapezium's parallel sides, as illustrated in Figure 4.4.b. A corollary of these results is that any two-dimensional shape can be partitioned into partitions of equal area by drawing a number of parallel lines; the lines are first drawn in a way that they split the given shape into triangles and trapeziums, and, then, the above-mentioned techniques are applied to each of these in turn.

However, as discussed in Section 3.2, the geometric approach is not adequate for an exact representation framework; thus, in the following sections, the merits of the above partitioning scheme are evaluated using an algebraic representation framework.

```
DOALL i = l1, u1
      (statements.1)
      DO j = l21 i + l22, u21 i + u22
         (statements.2)
      ENDDO
      (statements.3)
ENDDO
```

Figure 4.5: The general form of a triangular/trapezoidal loop nest.

4.3.1 Canonical Loop Nests

The loop nests examined in this section have the general form shown in Figure 4.5. It is assumed that the sets of statements labelled `statements.1`, `statements.2`, and `statements.3` do not include statements whose execution depends on the value of the index of a surrounding loop (i, as well as j, for the second set); hence, the workload corresponding to each set of statements remains the same for any iteration of the outer loop.[8] Furthermore, it is assumed that the second set (`statements.2`) contains at least one statement, i.e. it is not empty (if it was empty, then each iteration of the outer loop would perform the same amount of work and, consequently, the partitioning schemes described in Section 4.2 could be applied); however, the first and third sets of statements may be empty (if both are empty, then the loop is perfectly nested).

First, the subclass of *canonical* loop nests is examined, defined as follows:

Definition 4.1 *Consider the loop nest shown in Figure 4.5; this loop nest is* **canonical** *if and only if* $u_1 > l_1$, $u_{21} \neq l_{21}$ *and, for all* i, $l_1 \leq i \leq u_1$, *the inequality* $u_{21}i + u_{22} \geq l_{21}i + l_{22}$ *always holds.*

In the remainder of this chapter it is further assumed that $l_1 \geq 0$. This assumption does not affect the generality of the approach since, by applying normalisation (see Section 2.2.2.4), the loop bounds of the outer loop can be transformed so as to meet this assumption.

Based on Definition 4.1, the equivalent of Lemma 4.2, for canonical loop nests, can be expressed as follows:

Theorem 4.1 *Consider a canonical loop nest; if the index of the outer loop can be partitioned into* $2p$ *equal partitions, then the loop nest can be partitioned into* p *partitions of equal workload.*

Proof: Let $n = u_1 - l_1 + 1$ be the number of iterations of the outer loop. Since the outer loop can be partitioned into $2p$ equal partitions then, according to the discussion preceding Lemma 4.1, $2p$ divides n. Assuming that the work corresponding to statements outside the inner loop is W_I, and the work corresponding to the statements inside the inner loop is W_J, then the k-th partition of the outer loop will perform work W_k, equal to

$$W_k = \frac{n}{2p} W_I + I_k W_J, \tag{4.5}$$

where I_k is the number of times the statements inside the inner loop are executed by the k-th partition of the outer loop. We need to find which partition of the loop, say the x-th, $x \neq k$, makes the expression $W_k + W_x$ constant for all k, x (i.e., independent of k, x). Moreover, given that the loop nest must be partitioned into p partitions of equal workload, $W_k + W_x$ must be equal to W/p, where W is the total

[7] A fundamental property of Euclidean Geometry, a consequence of Proposition 2 in Book VI of Euclid's Elements: *If a straight line be drawn parallel to one of the sides of a triangle, it will cut the sides of the triangle proportionally* [101].

[8] This implies that the j loop may be surrounded by `DO` ... `ENDDO` loops which perform the same number of iterations regardless of the value of i; such loops may also exist in any of the three mentioned sets of statements. Thus, literally, the depth of a loop nest may be higher than 2; however, for simplicity, in this and subsequent discussion, such loops, which do not affect the strategy followed, are omitted.

work in the loop nest, i.e., $W = \sum_{k=0}^{2p-1} W_k$. Therefore,

$$\begin{aligned} (\frac{n}{2p}W_I + I_k W_J) + (\frac{n}{2p}W_I + I_x W_J) &= \frac{1}{p}\sum_{i=0}^{2p-1}(\frac{n}{2p}W_I + I_i W_J) \Longleftrightarrow \\ I_k + I_x &= \frac{1}{p}\sum_{i=0}^{2p-1} I_i, \end{aligned} \tag{4.6}$$

and the problem reduces to making $I_k + I_x$ constant. We proceed by computing a closed formula for I_k; the k-th partition of the outer loop will execute the statements inside the inner loop a number of times, I_k, equal to:

$$I_k = \sum_{i=l_1+kn/2p}^{l_1+(k+1)n/2p-1} \sum_{j=l_{21}i+l_{22}}^{u_{21}i+u_{22}} 1.$$

Since the loop nest is canonical, by hypothesis, $u_{21}i + u_{22} \geq l_{21}i + l_{22}$. Thus, evaluating the innermost sum, we get:

$$\begin{aligned} I_k &= \sum_{i=l_1+kn/2p}^{l_1+(k+1)n/2p-1} ((u_{21}-l_{21})i + (u_{22}-l_{22}+1)) \\ &= (u_{21}-l_{21})\frac{n(n+2kn+4l_1p-2p)}{8p^2} + (u_{22}-l_{22}+1)\frac{n}{2p}. \end{aligned} \tag{4.7}$$

Observing that I_k in (4.7) has the form $Ak + B$, A, B constants, then, replacing I_k in (4.6), we have

$$(Ak+B) + (Ax+B) = \frac{1}{p}\sum_{i=0}^{2p-1}(Ai+B) \Longleftrightarrow A(k+x) + 2B = A(2p-1) + 2B.$$

Therefore, $x = 2p - 1 - k$; indeed,

$$I_k + I_{2p-1-k} = \frac{n}{2p}((u_{21}-l_{21})(n+2l_1-1) + 2(u_{22}-l_{22}+1)),$$

which is a constant expression independent of k. It remains to find the sets of values for k and $2p-1-k$. Let $k \in S_1$, $2p-1-k \in S_2$. Clearly, $S_1 \cap S_2 = \varnothing$ and $S_1 \cup S_2 = \{0, 1, 2, \ldots, 2p-1\}$. Since, for any k such that $0 \leq k < p$,

$$0 \leq k \leq p-1 \Longleftrightarrow 0 \geq -k \geq 1-p \Longleftrightarrow 2p-1 \geq 2p-1-k \geq p,$$

then $S_1 = \{0, 1, 2, \ldots, p-1\}$ and $S_2 = \{p, p+1, \ldots, 2p-1\}$.

Therefore, the loop nest can be partitioned into p partitions of equal workload, where the k'-th partition of the loop nest, $0 \leq k' < p$, consists of the k-th and $(2p-k-1)$-th partition along the index of the outer loop. **QED**

Corollary 4.1 *Consider any loop nest which is partitioned into $2p$ equal partitions along the index of the outer loop. If, for all k, $0 \leq k \leq 2p-1$, the k-th partition has work equal to $Ak + B$, A, B constant and $A \neq 0$, then the loop nest can be partitioned into p partitions of equal workload.*

In Corollary 4.1, the restriction for a canonical loop nest is incorporated into the requirement related to the nature of the workload of the k-th partition (note that the restrictions associated with a canonical loop were used to evaluate the summation which derived I_k in Equation (4.7)). In the case where $A = 0$, each partition has the same workload and, although Corollary 4.1 still holds, it is not necessary to apply the partitioning scheme described in the proof of Theorem 4.1. Instead, the partitioning schemes described in Section 4.2 can be applied.

Based on this formulation, it is easy to extend Theorem 4.1 to cover cases where the loop nest contains more than one loop at level 2:

```
DOALL i = l1, u1
      (statements.1)
      DO j1 = l11 i + l12, u11 i + u12
         (statements.2)
      ENDDO
      (statements.3)
      DO j2 = l21 i + l22, u21 i + u22
         (statements.4)
      ENDDO
      ...
      DO jm = lm1 i + lm2, um1 i + um2
         (statements.2m)
      ENDDO
      (statements.2m+1)
ENDDO
```

Figure 4.6: The general form of multiple triangular/trapezoidal loop nests.

Corollary 4.2 *Consider the loop nest shown in Figure 4.6. Assume that the body of the outer loop contains m, $m > 1$, consecutive loops whose bounds depend on the index of the outer loop and, for all loops and for all i, $l_1 \leq i \leq u_1$, it is the case that $l_{k1}i + l_{22} \leq u_{k1}i + u_{k2}$, $1 \leq k \leq m$; if the index of the outer loop can be partitioned into $2p$ equal partitions, then the loop nest can be partitioned into p partitions of equal workload.*

Proof: Assuming that W_I is the work corresponding to the statements surrounded only by the outermost loop (i.e., `statements.1`, `statements.3`, ..., `statements.2m+1`), W_{J_x}, $1 \leq x \leq m$, is the work corresponding to the body of the loop with index J_x (i.e., `statements.2x`), I_{xk} is the number of times the respective statements (i.e., `statements.2x`) are executed by the k-th partition, and $n = u_1 - l_1 + 1$, then the k-th partition of the outer loop performs work W_k, equal to

$$W_k = \frac{n}{2p}W_I + I_{1k}W_{J_1} + I_{2k}W_{J_2} + \ldots + I_{mk}W_{J_m}.$$

Since, for each of the inner loops, the upper bound is always greater than or equal to the lower bound, by hypothesis, then, according to (4.7) (see proof of Theorem 4.1), I_{xk}, $1 \leq x \leq m$, has the form $ak + b$, where a, b are constants. Thus, the k-th partition of the outer loop has work equal to $Ak + B$, A, B constants, and, according to Corollary 4.1, if $A \neq 0$ (i.e., when there is at least one y, $1 \leq y \leq m$, such that $u_{y1} \neq l_{y1}$ and $\sum_{j=1}^{m}(u_{j1} - l_{j1})W_{J_j} \neq 0$), the loop nest can be partitioned into p partitions of equal workload. **QED**

Corollary 4.2 can also be used to cover cases where the bound of an inner loop depends on the indices of, say, two outer loops. Removing the innermost of the two outer loops (by replicating the loop body a number of times equal to the total number of loop iterations and changing the value of the loop index appropriately), the resulting loop nest has the general form shown in Figure 4.6. Thus, the partitioning scheme described in the proof of Theorem 4.1 can be applied.

Finally, the following corollary may be useful when a partitioning strategy beyond the loop nest level is applied:

Corollary 4.3 *Consider a canonical loop nest which is partitioned along the index of the outer loop into $2mp$ equal partitions; then the loop nest can be partitioned into p partitions of equal workload.*

Proof: According to Theorem 4.1, if the index of the outer loop can be partitioned into $2mp$ equal partitions, then the loop nest can be partitioned into mp partitions of equal workload. Since the mp partitions have the same workload, by merging any of them into groups of m partitions each, the loop

```
C  U1 is greater than L1
       DOALL I=L1,U1
             DO J=L1,I
                (statements)
             ENDDO
       ENDDO
```

a) Unpartitioned loop nest.

```
C      ---assuming that MODULO(N,2*P)=0---
       N=U1-L1+1
C*KSR* PARALLEL REGION(NUMTHREADS=P,PRIVATE=I,J,K,LK,UK)
       K=IPR_MID()
       LK=L1+K*N/(2*P)
       UK=L1+(K+1)*N/(2*P)-1
       DO I=LK,UK
          DO J=L1,I
             (statements)
          ENDDO
       ENDDO
       K=2*P-K-1
       LK=L1+K*N/(2*P)
       UK=L1+(K+1)*N/(2*P)-1
       DO I=LK,UK
          DO J=L1,I
             (statements)
          ENDDO
       ENDDO
C*KSR* END PARALLEL REGION
```

b) After loop partitioning.

Figure 4.7: Partitioning a triangular loop.

nest is partitioned into p partitions of equal workload. **QED**

In order to illustrate the partitioning technique used to formulate Theorem 4.1, consider the triangular loop nest shown in Figure 4.7.a. Based on the general form for a triangular loop nest, as depicted in Figure 4.5, this example is a special case in which $l_{21} = 0$, $l_{22} = l_1$, $u_{21} = 1$, $u_{22} = 0$; these values satisfy the conditions for a canonical loop nest, as required by Definition 4.1. Thus, assuming that the number of iterations of the outer loop, n, is a multiple of $2p$, where p is the number of processors used, the partitioning technique described in the proof of Theorem 4.1 leads to perfect load balance; the resulting code, after loop partitioning, is shown in Figure 4.7.b.

The same technique may be applied in the general case, where n is not a multiple of $2p$, provided that the outer loop is partitioned according to one of the three partitioning schemes described in Section 4.2. For instance, consider again the loop nest shown in Figure 4.7.a. Let `L1` $= 1$, `U1` $= 26$, and assume that the loop nest is to be partitioned across 3 processors. Then, the workload, W_k, of the k-th partition of the loop nest, $0 \leq k < 3$, is given by

$$W_k = \sum_{i=l_k}^{u_k} \sum_{j=1}^{i} W + \sum_{i=l_{5-k}}^{u_{5-k}} \sum_{j=1}^{i} W,$$

where $l_{k'}, u_{k'}$, $0 \leq k' < 6$, are the bounds of the k'-th partition along the index of the outer loop, and W is the workload represented by the `statements` in the loop body. Depending on the partitioning scheme used for the index of the outer loop, the value of W_k is as follows:

- Applying partitioning by decreasing order, then $W_0 = 113W$, $W_1 = 122W$, $W_2 = 116W$; thus, according to Equation (4.1), the resulting load imbalance is equal to $L = 5W$.
- Applying partitioning by increasing order, then $W_0 = 130W$, $W_1 = 121W$, $W_2 = 100W$; thus, according to Equation (4.1), the resulting load imbalance is equal to $L = 13W$.
- Applying partitioning by modular order, then $W_0 = 130W$, $W_1 = 104W$, $W_2 = 117W$; thus, according to Equation (4.1), the resulting load imbalance is equal to $L = 13W$.

It can be seen that each partitioning scheme results in different distribution of the workload, thus leading to different values for load imbalance; the smallest value is achieved when partitioning by decreasing order. In the general case, in order to find which partitioning scheme returns the smallest value for load imbalance, a comparison was made between partitioning by decreasing order and partitioning by increasing order; the results of this analysis are summarised in the following lemma:

Lemma 4.3 *Consider a canonical perfect loop nest (see Definition 4.1) which is partitioned along the index of the outer loop into $2p$ partitions applying partitioning either by increasing or by decreasing order. Partitioning the loop nest into p partitions where the k-th partition, $0 \leq k < p$, consists of the k-th and the $(2p-k-1)$-th partition along the index of the outer loop, and assuming that the number of iterations of the outer loop, n, is not a multiple of $2p$, then:*

- *Partitioning by decreasing order results in smaller load imbalance than partitioning by increasing order when either $u_{21} > l_{21}$ and $n \bmod 2p < p$ or $u_{21} < l_{21}$ and $n \bmod 2p > p$.*
- *Partitioning by increasing order results in smaller load imbalance than partitioning by decreasing order when either $u_{21} < l_{21}$ and $n \bmod 2p < p$ or $u_{21} > l_{21}$ and $n \bmod 2p > p$.*

When $n \bmod 2p = p$, both partitioning schemes result in the same load imbalance.

Proof: See Appendix A.

4.3.2 Generalised Loop Nests

Section 4.3.1 concentrated on the partitioning of canonical loop nests, as introduced in Definition 4.1. This section re-considers loop nests having the general form shown in Figure 4.5, but without the restrictions associated with Definition 4.1.

First, we prove that, if there are no values of i for which the loop nest is canonical, then the loop nest is rectangular; it is assumed that $l_1 \leq u_1$, i.e., the loop nest is not empty.

Theorem 4.2 *Consider the loop nest shown in Figure 4.5; then, either there is a subset of the iteration space of the outer loop for which the loop nest is canonical, or the loop nest is rectangular.*

Proof: If, for all i, $l_1 \leq i \leq u_1$, it is the case that $u_{21}i + u_{22} < l_{21}i + l_{22}$, then the upper bound of the inner loop is always smaller than the lower bound. Therefore, the statements inside the inner loop will never be executed; hence, the loop nest is rectangular. Furthermore, if $u_{21} = l_{21}$, the inner loop performs a constant number of iterations regardless of the value of i, therefore the loop nest is also rectangular. Conversely, if there are some i, say $l'_1 \leq i \leq u'_1$, where $l_1 \leq l'_1 \leq u'_1 \leq u_1$, for which $u_{21}i + u_{22} \geq l_{21}i + l_{22}$, then these i satisfy the conditions for a canonical loop nest according to Definition 4.1. **QED**

Theorem 4.2 provides the basis for partitioning any loop nest of depth 2. If the original loop nest is neither rectangular nor canonical then, applying index set splitting, it may be split into a canonical and a rectangular loop nest. Since partitioning schemes for each of these two classes of loop nests have already been described, a load balanced partition may be achieved.

Considering again the loop nest shown in Figure 4.5, and assuming that there are some values of i for which the loop nest is canonical, then these satisfy the following inequality:

$$u_{21}i + u_{22} \geq l_{21}i + l_{22} \Longleftrightarrow \begin{cases} i \geq (l_{22} - u_{22})/(u_{21} - l_{21}), & \text{if } u_{21} > l_{21}, \\ i \leq (l_{22} - u_{22})/(u_{21} - l_{21}), & \text{if } u_{21} < l_{21}. \end{cases}$$

Then the loop nest is split into a rectangular loop nest having work W_1 equal to

$$W_1 = \begin{cases} \displaystyle\sum_{i=l_1}^{\lceil r \rceil - 1} W_I, & \text{if } u_{21} > l_{21}, \\ \displaystyle\sum_{i=\lfloor r \rfloor + 1}^{u_1} W_I, & \text{if } u_{21} < l_{21}, \end{cases}$$

and a canonical loop nest having work W_2 equal to

$$W_2 = \begin{cases} \displaystyle\sum_{i=\lceil r \rceil}^{u_1} (W_I + \sum_{j=l_{21}i+l_{22}}^{u_{21}i+u_{22}} W_J), & \text{if } u_{21} > l_{21}, \\ \displaystyle\sum_{i=l_1}^{\lfloor r \rfloor} (W_I + \sum_{j=l_{21}i+l_{22}}^{u_{21}i+u_{22}} W_J), & \text{if } u_{21} < l_{21}, \end{cases}$$

where $r = (l_{22} - u_{22})/(u_{21} - l_{21})$. Each of the resulting loop nests may be partitioned by means of an appropriate partitioning scheme; using p processors, in the case where $u_{21} > l_{21}$, perfect load balance is achieved if $p|(\lceil r \rceil - l_1)$ and $p|(u_1 - \lceil r \rceil + 1)$, while, in the case where $u_{21} < l_{21}$, perfect load balance is achieved if $p|(u_1 - \lfloor r \rfloor)$ and $p|(\lfloor r \rfloor - l_1 + 1)$. Alternatively, the number of processors assigned to each loop nest can be made proportional to their workload; thus, the number of processors p_1 assigned to the rectangular loop may be either $\lfloor p\frac{W_1}{W_1+W_2} \rfloor$ or $\lceil p\frac{W_1}{W_1+W_2} \rceil$ depending on which expression minimises the load imbalance, while the number of processors p_2 assigned to the canonical loop nest is $p_2 = p - p_1$. But, the latter approach may require undue additional computational overhead in exchange for insignificant performance gains.

Theorem 4.2 can be extended to cover the case where there is more than one inner loop whose bounds depend on the index of the outer loop, such as the loop nest shown in Figure 4.6. In this case, it may be necessary to split the loop nest into more than two parts; one may be rectangular, while the remaining will contain either only one inner loop, whose bounds are a function of the index of the outer loop, or more than one loop, each satisfying the conditions required for Corollary 4.2. To find the appropriate ranges of values for splitting the outer loop index i, we have to consider the values which make each one of the inner loops canonical with respect to the outer loop; thus, the j-th inner loop, $1 \leq j \leq m$, is canonical for those values which satisfy the following system of inequalities:

$$l_1 \leq i \leq u_1 \quad \text{and} \quad l_{j1}i + l_{j2} \leq u_{j1}i + u_{j2}.$$

Repeating this procedure for each of the inner loops, the interval $[l_1, u_1]$ is split into a maximum of $m+1$ subintervals in each of which some of the inner loops are canonical, with respect to the outer loop, while others are not; this implies that, for the particular values of i, the latter are not executed. The interval where none of the inner loops is canonical with respect to the outer loop implies that, for these values of i, the loop nest is rectangular. Splitting the loop nest into as many loop nests as the number of subintervals, and eliminating the inner loops which are not executed in each particular case, are the steps that must be taken before partitioning.

To illustrate the above, consider the following example:

Example 4.1 Consider the loop nest shown in Figure 4.8.a. The inner loops do not meet the requirements for Corollary 4.2; thus, we consider the values of `I` which make each of the inner loops canonical with respect to the outer loop, namely `101` $\leq$ `I` $\leq$ `1000` for the first of the inner loops and `1` $\leq$ `I` $\leq$ `900` for the second. Based on these values, the loop nest can be split as shown in Figure 4.8.b. Assuming that the number of processors, `P`, divides 50, the resulting code can be partitioned as shown in Figure 4.8.c; then perfect load balance is achieved.

Instead, if the partitioning scheme described in the proof of Theorem 4.1 had been applied directly to the loop nest shown in Figure 4.8.a, and assuming that $p = 10$ processors were used and the amount of work in the body of each of the inner loops is W, a load imbalance equal to $9690W$ would result; applying the partitioning schemes described in Section 4.2, a load imbalance equal to $48465W$ would result. In the general case, where p may not be a divisor of the number of iterations, the results of Lemma 4.3 must be used, by adding appropriate conditionals to compute the bounds of the partitioned loops. □

4.3.3 Dealing with Conditionals

As illustrated in Section 4.2.1, conditionals which affect the amount of work each iteration of the outer loop performs (i.e., conditionals whose execution depends on the value of the index of the outer loop) may be removed by index set splitting. The partitioning scheme described in the proof of Theorem 4.1 allows us to deal with a wider class of conditionals than those dealt with in Section 4.2.1.

Consider, for instance, the code shown in Figure 4.9.a, in which there is a conditional whose execution depends on the value of the indices of two surrounding loops, one of which is the outer loop. Applying index set splitting to the innermost loop whose index affects the conditional's execution, i.e., the `J` loop, the conditional may be removed, and the code shown in Figure 4.9.b results. The next step is to remove the `MIN` and `MAX` from the loop bounds of the resulting code; this can be done directly if, at compile-time, it is known that `U1`$\leq$`U2` or `L1`$\geq$`U2` (then `MIN(I,U2)` is replaced by `I` or `U2` respectively), and `L1+1`$\geq$`L2` or `U1+1`$\leq$`L2` (then `MAX(I+1,L2)` is replaced by `I+1` or `L2` respectively). In the general case, the `MIN` and `MAX` can be replaced by `IF` statements; since the resulting conditionals involve the index of the outer loop and a constant, they can be removed, as described in Section 4.2.1. The final result of these transformations is a conditional-free loop nest which can be partitioned using the partitioning schemes described so far.

The same strategy would apply if a bound of the `J` loop depended on the index of the outer loop. Note that, in this case, conditionals whose execution depends only on the value of `J`, and not of `I`, also affect the work performed by each iteration of the outer loop, since the value of `J` depends on `I`. However, applying index set splitting, these conditionals can also be removed.

```
      DOALL I=1,1000
            DO J=200,2*I-1
               (statements.1)
            ENDDO
            DO J=I+100,1000
               (statements.2)
            ENDDO
      ENDDO
```

a) Unpartitioned loop nest.

```
      DOALL I=1,100
            DO J=I+100,1000
               (statements.2)
            ENDDO
      ENDDO
      DOALL I=101,900
            DO J=200,2*I-1
               (statements.1)
            ENDDO
            DO J=I+100,1000
               (statements.2)
            ENDDO
      ENDDO
      DOALL I=901,1000
            DO J=200,2*I-1
               (statements.1)
            ENDDO
      ENDDO
```

b) After index set splitting.

```
C*KSR* PARALLEL REGION(NUMTHREADS=P,
C*KSR*&     PRIVATE=I,J,KA,KB,LK,UK)
      KA=IPR_MID()
      KB=2*P-KA-1
C --- first loop nest ---
      LK=1+KA*100/(2*P)
      UK=1+(KA+1)*100/(2*P)-1
      DO I=LK,UK
         DO J=I+100,1000
            (statements.2)
         ENDDO
      ENDDO
      LK=1+KB*100/(2*P)
      UK=1+(KB+1)*100/(2*P)-1
      DO I=LK,UK
         DO J=I+100,1000
            (statements.2)
         ENDDO
      ENDDO
C --- second loop nest ---
      LK=101+KA*800/(2*P)
      UK=101+(KA+1)*800/(2*P)-1
      DO I=LK,UK
         DO J=200,2*I-1
            (statements.1)
         ENDDO
         DO J=I+100,1000
            (statements.2)
         ENDDO
      ENDDO
      LK=101+KB*800/(2*P)
      UK=101+(KB+1)*800/(2*P)-1
      DO I=LK,UK
         DO J=200,2*I-1
            (statements.1)
         ENDDO
         DO J=I+100,1000
            (statements.2)
         ENDDO
      ENDDO
C --- third loop nest ---
      LK=901+KA*100/(2*P)
      UK=901+(KA+1)*100/(2*P)-1
      DO I=LK,UK
         DO J=200,2*I-1
            (statements.1)
         ENDDO
      ENDDO
      LK=901+KB*100/(2*P)
      UK=901+(KB+1)*100/(2*P)-1
      DO I=LK,UK
         DO J=200,2*I-1
            (statements.1)
         ENDDO
      ENDDO
C*KSR* END PARALLEL REGION
```

c) After loop partitioning.

Figure 4.8: Partitioning a loop nest having multiple triangular/trapezoidal loops.

```
DOALL I=L1,U1
   (statements.1)
   DO J=L2,U2
      (statements.2)
      IF (I.GT.J) THEN
         (statements.3)
      ELSE
         (statements.4)
      ENDIF
   ENDDO
ENDDO
```

a) Loop with conditional.

```
DOALL I=L1,U1
   (statements.1)
   DO J=L2,MIN(I,U2)
      (statements.2)
      (statements.4)
   ENDDO
   DO J=MAX(I+1,L2),U2
      (statements.2)
      (statements.3)
   ENDDO
ENDDO
```

b) After index set splitting.

Figure 4.9: Partitioning loops which contain conditionals.

```
DOALL I=1,N
   DO J=1,I
      IF ((I+J).LE.N) THEN
         (statements.1)
      ELSE
         (statements.2)
      ENDIF
   ENDDO
ENDDO
```

a) Original Code.

```
DOALL I=1,N
   DO J=1,MIN(N-I,I)
      (statements.1)
   ENDDO
   DO J=MAX(N-I+1,1),I
      (statements.2)
   ENDDO
ENDDO
```

b) Eliminating the `IF`*.*

```
DOALL I=1,N
   IF (N-I.LT.I)
      DO J=1,N-I
         (statements.1)
      ENDDO
   ELSE
      DO J=1,I
         (statements.1)
      ENDDO
   ENDIF
   DO J=N-I+1,I
      (statements.2)
   ENDDO
ENDDO
```

c) Eliminating `MIN` *and* `MAX`*.*

```
DOALL I=1,FLOOR(N/2)
   DO J=1,I
      (statements.1)
   ENDDO
ENDDO
DOALL I=FLOOR(N/2)+1,N-1
   DO J=1,N-I
      (statements.1)
   ENDDO
   DO J=N-I+1,I
      (statements.2)
   ENDDO
ENDDO
DO J=1,N
   (statements.2)
ENDDO
```

d) After index set splitting.

Figure 4.10: An example of a loop nest with conditionals.

```
DOALL i = l_1, u_1
      (statements.1)
      DO j_2 = l_21 i + l_22, u_21 i + u_22
         (statements.2)
         DO j_3 = l_31 i + l_32 j_2 + l_33, u_31 i + u_32 j_2 + u_33
            ...
            (statements, including m - 4 DO loops)
            ...
            DO  j_m = l_m1 i + l_m2 j_2 + ... + l_m,m-1 j_m-1 + l_mm,
+                     u_m1 i + u_m2 j_2 + ... + u_m,m-1 j_m-1 + u_mm
               (statements.m)
            ENDDO
            ...
            (statements, including m - 4 ENDDO)
            ...
         ENDDO
         (statements.2m-2)
      ENDDO
      (statements.2m-1)
ENDDO
```

Figure 4.11: A canonical loop nest of depth m.

We now return to the loop nest considered in Example 3.4; the corresponding code is shown in Figure 4.10.a. In order to transform this to canonical form, we first remove the conditional, by splitting the `J` loop appropriately (see Figure 4.10.b), and we then remove the resulting `MIN` and `MAX` from the bounds of the `J` loop (see Figures 4.10.c and 4.10.d). The code shown in Figure 4.10.d can be used to verify the result obtained in Example 3.4 (recall that the latter considers the problem of counting the number of times the statements labelled `statements.1` are executed).

4.4 Partitioning Non-Rectangular Loop Nests of any Depth

The partitioning scheme described in Section 4.3 is based on the notion of a canonical loop nest, whose property that, upon partitioning along the index of the outer loop, the k-th partition of the loop nest performs work equal to $Ak + B$ for some constants A, B, makes it possible to partition a class of loop nests in such a way that perfect load balance is achieved. This section generalises the concept of a canonical loop nest for loop nests containing more than one inner loop, nested within each other, whose bounds depend on the index of the outer loop; this extended notion forms the basis for the subsequent description of partitioning schemes for such loop nests.

4.4.1 Canonical Loop Nests

We first generalise the notion of the canonical loop nest as follows:

Definition 4.2 *Consider the loop nest of depth m, $m \geq 2$, shown in Figure 4.11, where the body of the outer loop contains $m-1$ nested loops; this loop nest is* **canonical** *if and only if, $u_1 > l_1$ and, for all $i, j_2, j_3, \ldots, j_m$, the following inequalities always hold*

$$\begin{gathered} l_{21}i + l_{22} \leq u_{21}i + u_{22} \\ l_{31}i + l_{32}j_2 + l_{33} \leq u_{31}i + u_{32}j_2 + u_{33} \\ \cdots \\ l_{m1}i + l_{m2}j_2 + \ldots + l_{mm} \leq u_{m1}i + u_{m2}j_2 + \ldots + u_{mm} \end{gathered}$$

where, for each of j_k, $2 \le k \le m$, at least one of the differences $(l_{k1} - u_{k1})$, $(l_{k2} - u_{k2}), \ldots, (l_{k,k-1} - u_{k,k-1})$ is non-zero.[9]

Clearly, for $m = 2$, Definition 4.2 coincides with Definition 4.1, and it has been shown (see Theorem 4.1) that, for a loop nest conforming to the requirements of Definition 4.1, $2p$ equal partitions along the index of the outer loop suffice to partition the loop nest into p partitions of equal workload. The next two theorems demonstrate that, for the general case of a loop nest satisfying the requirements of Definition 4.2, $2p^{m-1}$ partitions along the index of the outer loop are needed to partition the loop nest into p partitions of equal workload.

Theorem 4.3 *Consider any loop nest of the form described in Definition 4.2. Assume that the outer loop is partitioned into $2p^{m-1}$ equal partitions; then, for all k, $0 \le k \le 2p^{m-1} - 1$, the k-th partition has a workload, W_k, equal to*

$$W_k = \sum_{i=0}^{m-1} C_i k^i, \quad C_i \ \textit{constants.}$$

Proof: Let $n = u_1 - l_1 + 1$ be the number of iterations of the outer loop; since the outer loop can be partitioned into $2p^{m-1}$ equal partitions then, according to Lemma 4.1, $2p^{m-1}$ divides n. Assuming that W_I is the work corresponding to the statements which are executed only by the outer loop (i.e., `statements.1` and `statements.2m-1`), W_{J_i}, $2 \le i \le m-1$, is the work corresponding to the statements which are executed by the loop with index j_i but not by the loop with index j_{i+1}, and W_{J_m} is the work corresponding to the statements which are in the body of the loop with index j_m, then the k-th partition of the loop nest will perform work W_k, equal to:

$$W_k = \frac{n}{2p^{m-1}} W_I + I_{J_2} W_{J_2} + \ldots + I_{J_m} W_{J_m} = \frac{n}{2p^{m-1}} W_I + \sum_{j=2}^{m} I_{J_j} W_{J_j}, \tag{4.8}$$

where I_{J_j}, $2 \le j \le m$, is the number of times the statements corresponding to work W_{J_j} are executed by the k-th partition. Proceeding to the computation of a closed formula for I_{J_j}, we have to evaluate the summation:

$$I_{J_j} = \sum_{i=l_k}^{u_k} \sum_{j_2=l_{21}i+l_{22}}^{u_{21}i+u_{22}} \sum_{j_3=l_{31}i+l_{32}j_2+l_{33}}^{u_{31}i+u_{32}j_2+u_{33}} \cdots \sum_{j_j=l_{j1}i+l_{j2}j_2+\ldots+l_{jj}}^{u_{j1}i+u_{j2}j_2+\ldots+u_{jj}} 1,$$

where $l_k = l_1 + kn/2p^{m-1}$ and $u_k = l_1 + (k+1)n/2p^{m-1} - 1$. Since the loop nest is canonical by hypothesis, the upper bound of any of these sums is always greater than or equal to the corresponding lower bound; thus, the innermost sum can be evaluated as follows:

$$I_{J_j} = \sum_{i=l_k}^{u_k} \sum_{j_2=l_{21}i+l_{22}}^{u_{21}i+u_{22}} \sum_{j_3=l_{31}i+l_{32}j_2+l_{33}}^{u_{31}i+u_{32}j_2+u_{33}} \cdots \sum_{j_{j-1}=l_{j-1,1}i+l_{j-1,2}j_2+\ldots+l_{j-1,j-1}}^{u_{j-1,1}i+u_{j-1,2}j_2+\ldots+u_{j-1,j-1}} f_1,$$

where f_1 is a first degree polynomial in $i, j_2, j_3, \ldots, j_{j-1}$. Repeating this procedure for the remaining summations (recall the rules for evaluating sums and Bernoulli's formula from Section 3.3), we end up with

$$I_{J_j} = \sum_{i=l_k}^{u_k} f_{j-1} = \sum_{i=1}^{u_k} f_{j-1} - \sum_{i=1}^{l_k-1} f_{j-1},$$

where f_{j-1} is a $(j-1)$-th degree polynomial in i; evaluating the two final summations, a $(j-1)$-th degree polynomial in k results, hence:

$$I_{J_j} = \sum_{i=0}^{j-1} A_{ij} k^i, \quad \text{for } A_{ij} \text{ constants.}$$

[9] This restriction guarantees that the number of iterations of each inner loop depends on the value of the index of at least one of the surrounding loops.

Therefore, Equation (4.8) can be written as

$$\begin{aligned}
W_k &= \frac{n}{2p^{m-1}} W_I + \sum_{j=2}^{m} I_{J_j} W_{J_j} \\
&= \frac{n}{2p^{m-1}} W_I + \sum_{j=2}^{m} \left(\sum_{i=0}^{j-1} A_{ij} k^i \right) W_{J_j} \\
&= \frac{n}{2p^{m-1}} W_I + \sum_{j=2}^{m} \left(A_{0j} + A_{1j} k + A_{2j} k^2 + \ldots + A_{j-1,j} k^{j-1} \right) W_{J_j} \\
&= \frac{n}{2p^{m-1}} W_I + \sum_{j=2}^{m} A_{0j} W_{J_j} + \sum_{j=2}^{m} A_{1j} W_{J_j} k + \sum_{j=3}^{m} A_{2j} W_{J_j} k^2 + \ldots \\
&\qquad \ldots + \sum_{j=m}^{m} A_{m-1,m} W_{J_j} k^{m-1}.
\end{aligned}$$

By replacing, in the last equation,

$$\sum_{j=i+1}^{m} A_{ij} W_{J_j}, 1 \le i \le m-1,$$

with C_i, and

$$\frac{n}{2p^{m-1}} W_I + \sum_{j=2}^{m} A_{0j} W_{J_j}$$

with C_0 we get:

$$W_k = C_0 + C_1 k + \ldots + C_{m-1} k^{m-1} = \sum_{i=0}^{m-1} C_i k^i.$$

QED

Theorem 4.4 *Consider any loop nest which is partitioned along the index of the outer loop into $2p^{m-1}$, $m \ge 2$, equal partitions. If, for all k, $0 \le k \le 2p^{m-1} - 1$, the k-th partition has workload, W_k, equal to*

$$W_k = \sum_{i=0}^{m-1} C_i k^i, \quad \textit{where } C_i \textit{ constants,}$$

then the loop nest can be partitioned into p partitions of equal workload.

Proof: Assume that a way of partitioning the loop nest into p partitions of equal workload is found such that each partition, k', $0 \le k' \le p-1$, of the loop nest consists of $2p^{m-1}/p = 2p^{m-2}$, $m \ge 2$, partitions along the index of the outer loop and performs work $W'_{k'}$; then, for all k', it must be the case that:

$$W'_0 = W'_1 = W'_2 = \ldots = W'_{p-1}.$$

Let $S_{k'} = \{s_{k'1}, s_{k'2}, \ldots, s_{k'2p^{m-2}}\}$ be the set of partitions along the index of the outer loop which compose the k'-th partition of the loop nest; clearly, the integers s_{ij}, $0 \le i < p$, $1 \le j \le 2p^{m-2}$ (that is, the elements of all the sets $S_{k'}$, $0 \le k' < p$), are a permutation of the integers $0, 1, 2, \ldots, 2p^{m-1} - 1$. Then, the workload, $W'_{k'}$, of the k'-th partition of the loop nest is given by

$$W'_{k'} = W_{s_{k'1}} + W_{s_{k'2}} + \ldots + W_{s_{k'2p^{m-2}}} = \sum_{i=1}^{2p^{m-2}} W_{s_{k'i}} = \sum_{i=1}^{2p^{m-2}} \sum_{j=0}^{m-1} C_j s^j_{k'i}.$$

Since, for any two distinct partitions x, y of the loop nest, $0 \le x, y < p$ and $x \neq y$, $W'_x = W'_y \iff W'_x - W'_y = 0$, we have:

$$
\begin{aligned}
W'_x - W'_y &= \sum_{i=1}^{2p^{m-2}} \sum_{j=0}^{m-1} C_j s_{xi}^j - \sum_{i=1}^{2p^{m-2}} \sum_{j=0}^{m-1} C_j s_{yi}^j \\
&= \sum_{i=1}^{2p^{m-2}} \left(C_0 + C_1 s_{xi} + C_2 s_{xi}^2 + \ldots + C_{m-1} s_{xi}^{m-1}\right) - \\
&\quad \sum_{i=1}^{2p^{m-2}} \left(C_0 + C_1 s_{yi} + C_2 s_{yi}^2 + \ldots + C_{m-1} s_{yi}^{m-1}\right) \\
&= \sum_{i=1}^{2p^{m-2}} \left(C_1(s_{xi} - s_{yi}) + C_2(s_{xi}^2 - s_{yi}^2) + \ldots + C_{m-1}(s_{xi}^{m-1} - s_{yi}^{m-1})\right) \\
&= \left(\sum_{i=1}^{2p^{m-2}} s_{xi} - \sum_{i=1}^{2p^{m-2}} s_{yi}\right) C_1 + \left(\sum_{i=1}^{2p^{m-2}} s_{xi}^2 - \sum_{i=1}^{2p^{m-2}} s_{yi}^2\right) C_2 + \ldots \\
&\quad + \left(\sum_{i=1}^{2p^{m-2}} s_{xi}^{m-1} - \sum_{i=1}^{2p^{m-2}} s_{yi}^{m-1}\right) C_{m-1}.
\end{aligned}
$$

The above expression must be equal to zero. A solution is given when the coefficients of the C_i, $1 \le i \le m-1$, are all equal to zero. In this case, the problem reduces to solving the following system of equations:

$$
\begin{cases}
\sum_{i=1}^{2p^{m-2}} s_{xi} &= \sum_{i=1}^{2p^{m-2}} s_{yi} \\
\sum_{i=1}^{2p^{m-2}} s_{xi}^2 &= \sum_{i=1}^{2p^{m-2}} s_{yi}^2 \\
&\ldots \\
\sum_{i=1}^{2p^{m-2}} s_{xi}^{m-1} &= \sum_{i=1}^{2p^{m-2}} s_{yi}^{m-1}.
\end{cases}
$$

Generalising the above for all partitions of the loop nest, we get the following system of equations:

$$
\begin{cases}
\sum_{i=1}^{2p^{m-2}} s_{0i} = \sum_{i=1}^{2p^{m-2}} s_{1i} = \sum_{i=1}^{2p^{m-2}} s_{2i} = \ldots = \sum_{i=1}^{2p^{m-2}} s_{p-1,i} \\
\sum_{i=1}^{2p^{m-2}} s_{0i}^2 = \sum_{i=1}^{2p^{m-2}} s_{1i}^2 = \sum_{i=1}^{2p^{m-2}} s_{2i}^2 = \ldots = \sum_{i=1}^{2p^{m-2}} s_{p-1,i}^2 \\
\ldots \\
\sum_{i=1}^{2p^{m-2}} s_{0i}^{m-1} = \sum_{i=1}^{2p^{m-2}} s_{1i}^{m-1} = \sum_{i=1}^{2p^{m-2}} s_{2i}^{m-1} = \ldots = \sum_{i=1}^{2p^{m-2}} s_{p-1,i}^{m-1}
\end{cases}
\tag{4.9}
$$

Thus, all sets $S_{k'}$, $0 \le k' < p$, must have equal sums of their elements, equal sums of the squares of their elements, and so on, up to the $(m-1)$-th power. Hence, if we find a method of partitioning all the integers from 0 to $2p^{m-1} - 1$ into the p sets $S_0, S_1, \ldots, S_{p-1}$ so that (4.9) holds, the theorem has been proved. We intend to find a solution using induction on m.

A solution for $m = 2$ has been given in Theorem 4.1.

For the case $m = 3$, we consider the p pairs given by

$$\left\{2pi + (k' + i) \bmod p, 2p(i + 1) - 1 - (k' + i) \bmod p\right\},$$

where $0 \leq i < p$. We observe that the sum of the elements of each pair is independent of k'. Furthermore, the sum of the squares of the elements of all the pairs is constant; this is because, for all k', $0 \leq k' < p$, the value of

$$((k'+0) \bmod p)^2 + ((k'+1) \bmod p)^2 + \ldots + ((k'+p-1) \bmod p)^2$$

is equal to $0^2 + 1^2 + 2^2 + \ldots + (p-1)^2$. Given also that, for all i, k', any two such pairs have different elements, we have found a way of partitioning the integers $0, 1, 2, \ldots, 2p^2-1$ such that they yield a solution to (4.9). Therefore, for $m = 3$, the set of partitions along the index of the outer loop which compose the k'-th partition of the loop nest is given by:

$$S_{k'} = \{2pi + (k'+i) \bmod p : 0 \leq i < p\} \cup \{2p(i+1) - 1 - (k'+i) \bmod p : 0 \leq i < p\}. \tag{4.10}$$

Suppose that, for some $m, m \geq 3$, the set of partitions along the index of the outer loop which compose the k'-th partition of the loop nest, $S_{k'}$, is given by:

$$\begin{aligned} S_{k'} \quad = \quad & \left\{ 2pi + (k' + \sum_{j=0}^{m-3} \lfloor i/p^j \rfloor) \bmod p : 0 \leq i < p^{m-2} \right\} \bigcup \\ & \left\{ 2p(i+1) - 1 - (k' + \sum_{j=0}^{m-3} \lfloor i/p^j \rfloor) \bmod p : 0 \leq i < p^{m-2} \right\}. \end{aligned} \tag{4.11}$$

Then we have to prove that (4.11) also provides a solution to (4.9) when m is replaced by $m+1$. Consider the p^2 sets $S'_{tk'}$, $0 \leq t < p$, $0 \leq k' < p$, each consisting of $2p^{m-2}$ elements and defined as follows:

$$\begin{aligned} S'_{tk'} \quad = \quad & \left\{ 2p(i + tp^{m-2}) + \left(k' + t + \sum_{j=0}^{m-3} \lfloor i/p^j \rfloor \right) \bmod p : 0 \leq i < p^{m-2} \right\} \bigcup \\ & \left\{ 2p(i + tp^{m-2} + 1) - 1 - \left(k' + t + \sum_{j=0}^{m-3} \lfloor i/p^j \rfloor \right) \bmod p : 0 \leq i < p^{m-2} \right\}. \end{aligned}$$

Clearly, for any k', the sets $S'_{0k'}$, $0 \leq k' < p$, have an equal sum for each of the 1-st, 2-nd, ..., $(m-1)$-th powers of their elements, because of (4.11). The same also holds for each of the sets $S'_{1k'}, S'_{2k'}, \ldots, S'_{p-1,k'}$, $0 \leq k' < p$ (note that their elements can be written in the form $s'_{0k'j} + c$, where $s'_{0k'j}$ are the elements of the set $S'_{0k'}$ and c is a constant). Thus, the sets $S'_{k'}$, where $S'_{k'} = S'_{0k'} \cup S'_{1k'} \cup \ldots \cup S'_{p-1,k'}$, have an equal sum for each of the 1-st, 2-nd, ..., $(m-1)$-th powers of their elements. It can be shown that the sum of the m-th powers of their elements is also equal; this is because, for all k', $0 \leq k' < p$, the value of

$$((k'+0+\sigma_i) \bmod p)^m + ((k'+1+\sigma_i) \bmod p)^m + \ldots + ((k'+p-1+\sigma_i) \bmod p)^m,$$

where $\sigma_i = \sum_{j=0}^{m-3} \lfloor i/p^j \rfloor$, is equal to $0^m + 1^m + 2^m + \ldots + (p-1)^m$. Hence, the sets $S'_{k'}$ provide a solution to (4.9) when m is replaced by $m+1$. However, the sets $S'_{k'}$ are identical with the sets $S_{k'}$ in (4.11). This is because the quantity $i + tp^{m-2}$, $0 \leq i < p^{m-2}$, $0 \leq t < p$, returns the same values as the quantity i, $0 \leq i < p^{m-1}$, and, similarly, the quantity

$$\left(k' + t + \sum_{j=0}^{m-3} \lfloor i/p^j \rfloor \right) \bmod p = \left(k' + \sum_{j=0}^{m-2} \lfloor (i + tp^{m-2})/p^j \rfloor \right) \bmod p,$$

$0 \leq i < p^{m-2}$, $0 \leq t < p$, returns the same values as the quantity

$$\left(k' + \sum_{j=0}^{m-2} \lfloor i/p^j \rfloor \right) \bmod p, \quad 0 \leq i < p^{m-1}.$$

Thus, the general solution of (4.9) is given by (4.11), and, therefore, the loop nest can be partitioned into p partitions of equal workload. **QED**

Theorems 4.3 and 4.4 can be summarised as follows:

Corollary 4.4 *Consider a canonical loop nest of depth m; if the index of the outer loop can be partitioned into $2p^{m-1}$ equal partitions, then the loop nest can be partitioned into p partitions of equal workload.*

The proof of Theorem 4.4 shows that the problem of partitioning the loop nest into p partitions of equal workload (assuming that the index of the outer loop is partitioned into $2p^{m-1}$ equal partitions) can be reduced to that of partitioning all the integers from 0 to $2p^{m-1}-1$ into p sets, say S_i, where $0 \leq i \leq p-1$, such that the sum of the elements of every set is the same, as is the sum of the elements squared, the sum of the elements cubed, and so on, up to the $(m-1)$-th power; then the elements of all the sets S_i, as given by (4.11), say $s_{i1}, s_{i2}, \ldots, s_{i,2p^{m-2}}$, provide a solution to the system of $(m-1)(p-1)$ equations

$$\sum_{i=1}^{x} s_{0i}^h = \sum_{i=1}^{x} s_{1i}^h = \sum_{i=1}^{x} s_{2i}^h = \ldots = \sum_{i=1}^{x} s_{p-1,i}^h, \quad 1 \leq h \leq m-1, \tag{4.12}$$

for $x = 2p^{m-2}$. Finding non-trivial solutions (i.e., solutions where no two sets $\{s_{ui}\}$ and $\{s_{vi}\}$, with $u \neq v$, are permutations of one another) of the above system of equations has been a long-standing problem in Number Theory, known as the *Prouhet-Tarry-Escott* problem [96, §21.9]. Its history dates from 1851, when Prouhet, in a memoir submitted to the French Academy of Sciences, gave a rule for partitioning the integers from 0 to p^m-1 into p sets of p^{m-1} members, thus providing a solution of (4.12) for $x = p^{m-1}$ [183]. In short, Prouhet's rule consists of assigning each integer i, in the range $[0, p^m-1]$, to the set S_v, $0 \leq v \leq p-1$, where v results as the modulus p of the sum of the digits of i written in base p [232]; this approach coincides with ours in the case $p = 2$.[10] However, in the general case, Prouhet's approach, if applied to Theorem 4.4, would require p^m equal partitions along the index of the outer loop, that is a number higher by a factor of $p/2$ than the one suggested by the theorem.

In order to illustrate the results of Theorems 4.3 and 4.4, we consider the following example:

Example 4.2 Consider the loop nest shown in Figure 4.12.a. Assuming that `N` > 1, then the inequalities `-2` $\leq$ `3*I-1` and `J+I` $\leq$ `5*I+2` always hold, while, for each inequality, the coefficients of `I` are non-zero; hence, the requirements of Definition 4.2 are satisfied and the loop nest is canonical for $m = 3$. Thus, based on Theorems 4.3 and 4.4, and assuming that the number of iterations of the outer loop, `N`, is a multiple of $2p^2$, partitioning the loop nest according to (4.10) leads to perfect load balance; the partitioned loop nest is shown in Figure 4.12.b.

To provide an intuitive view of this partitioning scheme, the corresponding geometrical representation of the original loop nest is shown in Figure 4.13.a. Assuming that two processors are used, the index of the outer loop is partitioned into 8 partitions; the corresponding polytope for each partition is shown in Figure 4.13.b. Then, applying (4.10), the first processor is assigned partitions 0, 3, 5, and 6, while the second processor is assigned partitions 1, 2, 4, and 7; in geometrical terms, the polytopes have been assigned to two groups in such a way that the total volume of the polytopes in each group is the same. □

In the general case, where the number of iterations of the outer loop, n, is not a multiple of $2p^{m-1}$, the partitioning technique suggested in the proof of Theorem 4.4 can be applied, provided that the outer loop is partitioned according to one of the three partitioning schemes described in Section 4.2. In this case, a small value of load imbalance is expected.

Theorems 4.3 and 4.4 can also be extended to cover cases where there are more than one inner loops at the same level (i.e., loops which are surrounded only by the same outer loops) whose bounds depend on the index of a surrounding loop (cf. Corollary 4.2 for the case of loop nests of depth 2); the necessary requirement is that, for any loop, the lower bound is always less than or equal to the upper bound.

[10] Finding non-trivial solutions of (4.12) for $p = 2$ is known as the Tarry-Escott problem and has attracted most of the interest in the problem [30, 80].

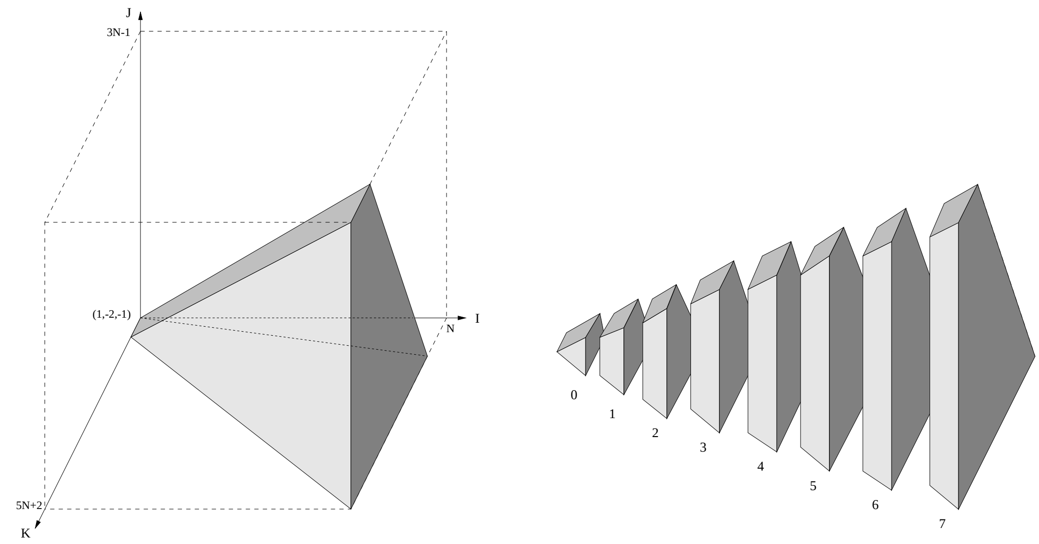
```
      DOALL I=1,N
         DO J=-2,3*I-1
            DO K=J+I,5*I+2
               (statements)
            ENDDO
         ENDDO
      ENDDO
```

a) Unpartitioned loop nest.

```
C  --- assuming that MODULO(N,2*P*P)=0 ---
C*KSR* PARALLEL REGION(NUMTHREADS=P,PRIVATE=I,J,K,LK,UK,K1,K2)
      K1=IPR_MID()
      DO II=0,P-1
         K2=2*P*II+MOD(K1+II,P)
         LK=1+K2*N/(2*P*P)
         UK=1+(K2+1)*N/(2*P*P)-1
         DO I=LK,UK
            DO J=-2,3*I-1
               DO K=J+I,5*I+2
                  (statements)
               ENDDO
            ENDDO
         ENDDO
         K2=2*P*(II+1)-1-MOD(K1+II,P)
         LK=1+K2*N/(2*P*P)
         UK=1+(K2+1)*N/(2*P*P)-1
         DO I=LK,UK
            DO J=-2,3*I-1
               DO K=J+I,5*I+2
                  (statements)
               ENDDO
            ENDDO
         ENDDO
      ENDDO
C*KSR* END PARALLEL REGION
```

b) After loop partitioning.

Figure 4.12: An example of partitioning a loop nest of depth 3.



a) Unpartitioned loop nest. *b) Partitioning into 8 partitions.*

Figure 4.13: Geometrical representation of the loop nest shown in Figure 4.12.

4.4.2 Generalised Loop Nests

Section 4.3.2 illustrates how to apply index set splitting to transform non-canonical loop nests of depth 2 (that is, loop nests having the general form shown in Figure 4.5 but not conforming to the constraints in Definition 4.1) into consecutive loop nests each of which satisfies the requirements for partitioning as described in the proof of Theorem 4.1. The same approach can be used for non-canonical loop nests of depth greater than 2 (that is, loop nests having the general form shown in Figure 4.11 but not conforming to the constraints in Definition 4.2). However, in the latter case, it may be necessary to apply index set splitting not only to the outer loop but also to some or all of the inner loops; whether or not this is necessary is determined by considering successively for each loop those index values which make it canonical with respect to the surrounding loops.

Considering again the loop nest shown in Figure 4.11, the first step consists of finding the values of i which satisfy the inequalities

$$l_1 \le i \le u_1 \quad \text{and} \quad l_{21}i + l_{22} \le u_{21}i + u_{22}.$$

If no such values exist, then the loop with index j_2 is never executed. If there are such values, given by $l_1 \le i \le u_1$, the loop with index j_2 is always executed; therefore, this loop is canonical with respect to the outer loop, and no index set splitting is required. Conversely, if there is a subset of the values of i which satisfies both inequalities, say $l_1 \le i \le u'_1$, where $u'_1 < u_1$, then the outer loop must be split into two consecutive loops, the first of which corresponds to the values given by $l_1 \le i \le u'_1$, while the second corresponds to $u'_1 + 1 \le i \le u_1$; clearly, for the former values, the loop with index j_2 is canonical with respect to the outer loop, while, for the latter, it is never executed.

If, as a result of the previous step, there are some values of i for which the two outermost loops are canonical, the next step consists of finding the values of i, j_2 which make the three outermost loops canonical; these values must satisfy the system of inequalities

$$\begin{gathered} l'_1 \le i \le u'_1 \\ l_{21}i + l_{22} \le j_2 \le u_{21}i + u_{22} \\ l_{31}i + l_{32}j_2 + l_{33} \le u_{31}i + u_{32}j_2 + u_{33}, \end{gathered}$$

```
DOALL I=1,1000
      DO J=1,I
         DO K=2*I-J,1000
            (statements.1)
         ENDDO
      ENDDO
      (statements.2)
      DO J=2*I-500,1000
         DO K=I+J,1000
            (statements.3)
         ENDDO
      ENDDO
ENDDO
```

a) Unpartitioned loop nest.

```
DOALL I=1,750
      DO J=MAX(1,2*I-1000),I
         DO K=2*I-J,1000
            (statements.1)
         ENDDO
      ENDDO
      (statements.2)
      DO J=2*I-500,MIN(1000,1000-I)
         DO K=I+J,1000
            (statements.3)
         ENDDO
      ENDDO
ENDDO
DOALL I=751,1000
      DO J=MAX(1,2*I-1000),I
         DO K=2*I-J,1000
            (statements.1)
         ENDDO
      ENDDO
      (statements.2)
ENDDO
```

b) After initial index set splitting.

```
DOALL I=1,500
      DO J=1,I
         DO K=2*I-J,1000
            (statements.1)
         ENDDO
      ENDDO
      (statements.2)
      DO J=2*I-500,1000-I
         DO K=I+J,1000
            (statements.3)
         ENDDO
      ENDDO
ENDDO
DOALL I=501,750
      DO J=2*I-1000,I
         DO K=2*I-J,1000
            (statements.1)
         ENDDO
      ENDDO
      (statements.2)
      DO J=2*I-500,1000-I
         DO K=I+J,1000
            (statements.3)
         ENDDO
      ENDDO
ENDDO
DOALL I=751,1000
      DO J=2*I-1000,I
         DO K=2*I-J,1000
            (statements.1)
         ENDDO
      ENDDO
      (statements.2)
ENDDO
```

c) Transformed code.

Figure 4.14: Transforming generalised loop nests to canonical loop nests.

where the first inequality corresponds to those values of i that make the two outermost loops canonical.

The same procedure is repeated for each loop, successively, until there are no remaining loops or else a given system of, say k, $2 \leq k \leq m$, inequalities has no solutions (this would imply that the loop with index j_k is never executed for these values of $i, j_2, \ldots, j_{k-1}$ that render the $k-1$ outermost loops canonical). Note that, in the case where the original loop nest contains more than one consecutive loop at some level (an analogue of the loop nest shown in Figure 4.6, but for loop nests of depth greater than 2), the same procedure should be applied for each loop separately.

In order to illustrate the above, consider the following example:

Example 4.3 Consider the loop nest shown in Figure 4.14.a. Since there are two consecutive loop nests in the body of the `I` loop, the procedure described above must be applied separately for each of them.

For the first loop nest, the `J` loop is canonical with respect to the outer loop; the `K` loop is canonical (with respect to the surrounding loops) when `2*I-J` $\leq$ `1000` $\Longleftrightarrow$ `J` $\geq$ `2*I-1000`. Thus, the `J` loop must

```
      F1=FLOOR(500/(2*P*P))
      F2=FLOOR(250/(2*P*P))
      M1=MOD(500,2*P*P)
      M2=MOD(250,2*P*P)
C*KSR* PARALLEL REGION(NUMTHREADS=P,
C*KSR*&PRIVATE=I,J,K,II,LK,UK,K1,K2)
      K1=IPR_MID()
      DO II=0,P-1
C --- first loop nest ---
         K2=2*P*II+MOD(K1+II,P)
         LK=1+K2*F1+MIN(M1,K2)
         UK=1+(K2+1)*F1+MIN(M1,K2+1)-1
         DO I=LK,UK
            DO J=1,I
               DO K=2*I-J,1000
                  (statements.1)
               ENDDO
            ENDDO
            (statements.2)
            DO J=2*I-500,1000-I
               DO K=I+J,1000
                  (statements.3)
               ENDDO
            ENDDO
         ENDDO
         K2=2*P*(II+1)-1-MOD(K1+II,P)
         LK=1+K2*F1+MIN(M1,K2)
         UK=1+(K2+1)*F1+MIN(M1,K2+1)-1
         DO I=LK,UK
            DO J=1,I
               DO K=2*I-J,1000
                  (statements.1)
               ENDDO
            ENDDO
            (statements.2)
            DO J=2*I-500,1000-I
               DO K=I+J,1000
                  (statements.3)
               ENDDO
            ENDDO
         ENDDO
C --- second loop nest ---
         K2=2*P*II+MOD(K1+II,P)
         LK=501+K2*F2+MIN(M2,K2)
         UK=501+(K2+1)*F2+MIN(M2,K2+1)-1
         DO I=LK,UK
            DO J=2*I-1000,I
               DO K=2*I-J,1000
                  (statements.1)
               ENDDO
            ENDDO
            (statements.2)
            DO J=2*I-500,1000-I
               DO K=I+J,1000
                  (statements.3)
               ENDDO
            ENDDO
         ENDDO
         K2=2*P*(II+1)-1-MOD(K1+II,P)
         LK=501+K2*F2+MIN(M2,K2)
         UK=501+(K2+1)*F2+MIN(M2,K2+1)-1
         DO I=LK,UK
            DO J=2*I-1000,I
               DO K=2*I-J,1000
                  (statements.1)
               ENDDO
            ENDDO
            (statements.2)
            DO J=2*I-500,1000-I
               DO K=I+J,1000
                  (statements.3)
               ENDDO
            ENDDO
         ENDDO
C --- third loop nest ---
         K2=2*P*II+MOD(K1+II,P)
         LK=751+K2*F2+MIN(M2,K2)
         UK=751+(K2+1)*F2+MIN(M2,K2+1)-1
         DO I=LK,UK
            DO J=2*I-1000,I
               DO K=2*I-J,1000
                  (statements.1)
               ENDDO
            ENDDO
            (statements.2)
         ENDDO
         K2=2*P*(II+1)-1-MOD(K1+II,P)
         LK=751+K2*F2+MIN(M2,K2)
         UK=751+(K2+1)*F2+MIN(M2,K2+1)-1
         DO I=LK,UK
            DO J=2*I-1000,I
               DO K=2*I-J,1000
                  (statements.1)
               ENDDO
            ENDDO
            (statements.2)
         ENDDO
C --- end ---
      ENDDO
C*KSR* END PARALLEL REGION
```

Figure 4.15: Partitioning the code shown in Figure 4.14.c.

be split into two consecutive loops depending on which values of J render the K loop canonical; the bounds of the first loop will be 1 and MAX(1,2*I-1000)-1, and of the second loop MAX(1,2*I-1000) and I. Since the body of the J loop does not contain statements other than the K loop, no statements are executed when 1 $\leq$ J $\leq$ MAX(1,2*I-1000)-1; hence, the corresponding loop can be eliminated.

For the second loop nest, the J loop is canonical with respect to the outer loop when 2*I-500 $\leq$ 1000 $\Longleftrightarrow$ I $\leq$ 750; the index of the I loop is split accordingly. Furthermore, the K loop is canonical when I+J $\leq$ 1000 $\Longleftrightarrow$ J $\leq$ 1000-I. The code resulting after applying the appropriate transformations is shown in Figure 4.14.b. Evaluating MAX(1,2*I-1000), by replacing it with appropriate conditionals which are then removed using index set splitting, as described in Section 4.3.3, results in the code shown in Figure 4.14.c (note that MIN(1000,1000-I) is always equal to 1000-I since I takes only positive values). Finally, partitioning the index of the outer loop, I, by decreasing order and grouping the partitions according to the scheme described in the proof of Theorem 4.4, the partitioned code is obtained, as shown in Figure 4.15. Perfect load balance can be achieved using 5 processors; in general, the value of load imbalance tends to be comparatively low.[11] □

[11] Indicatively, we computed the relative load imbalance, L_R, as given by Equation (4.2), for a varying number of

4.5 Concluding Remarks

A strategy for mapping loop nests onto parallel processors at compile-time has been presented in this chapter; primarily, this strategy aims to minimise the overhead due to load imbalance. Measures for quantifying the latter have also been described.

Based on this strategy, from an algorithmic point of view, the first step before partitioning a given loop nest is the removal of conditionals whose execution depends on the value of the index of the outermost, parallel loop; this is achieved by splitting the outermost loop appropriately. Applying further splitting, the resulting loop nests are transformed to a canonical form; the proofs of Theorems 4.1, 4.3, and 4.4 describe how to partition loop nests having the latter form in a way that leads to perfect load balance.

In the context of a parallelising compiler, an advantage of this strategy is that it readily provides estimates for load imbalance; as discussed in Chapter 2, such estimates may be essential for predicting performance. In addition, the strategy generally exhibits good behaviour (at least according to the quantitative measures used in this chapter); evaluating its performance in practice is the subject of the following chapter.

processors, assuming that all three sets of statements labelled `statements.1`, `statements.2`, and `statements.3` perform the same amount of work; as can be seen from the following table, the value of relative load imbalance remains small:

Processors	2	3	4	5	6	7	8
L_R	.00068	.00090	.00111	.00000	.00239	.01274	.00217
Processors	9	10	12	14	16	20	25
L_R	.01333	.00073	.00300	.00799	.00613	.00160	.00138

Chapter 5

Evaluation and Experimental Results

5.1 Introduction

In order to evaluate the performance gains of the mapping strategy described in Chapter 4, a series of experiments has been conducted. Two routes have been followed for analysis of the results: the first compares the values of load imbalance, computed as suggested in Chapter 4, when applying different mapping schemes; the second compares the resulting execution times on the KSR1. Our objective has been not only to evaluate the effectiveness of the schemes introduced in Chapter 4, but also to establish the appropriateness of the theoretical values for load imbalance as a means of justifying the selection of a particular mapping scheme.

In essence, five loop mapping schemes have been evaluated; namely the following (for convenience, each scheme has been given an acronym, shown in parenthesis, which is used as a shorthand in the remainder of this chapter):

- *Chunk scheduling* (KAP), as described in Section 2.3.1.2 and implemented via the KSR FORTRAN compiler by means of the `TILE` directive [123]; the latter is the approach followed for loop parallelisation by the KSR edition of the KAP parallelising compiler [122], using a number of chunks which is always equal to the number of processors (the default value of the `TILE` directive).
- *Block partitioning* (MARS), based on blocks composed of consecutive loop iterations, as described in Section 2.3.1.1 and corresponding to the implementation of the MARS experimental parallelising compiler (see footnote 3 in Section 4.2).
- *Cyclic partitioning* (CYC), as described in Section 2.3.1.1.
- *Balanced chunk scheduling* (BCS), as described in Section 2.3.1.1 and further analysed in Appendix B.
- The partitioning schemes, described in Chapter 4, based on the notion of a *canonical loop nest* (CAN). In some cases, when more than one of the schemes based on the notion of a canonical loop nest are used on the same benchmark, a suffix is added to the CAN shorthand; these cases are further discussed in Section 5.2.

Remember that partitioning takes place at *run-time* when applying KAP, while it takes place at *compile-time* when applying any of the remaining four schemes.[1]

[1] In the context of the KSR1 implementation, the former case corresponds to parallelisation based on the `TILE` directive, while the latter cases correspond to parallelisation based on the `PARALLEL REGION` directive [123]. Although it appears that this variation in the implementation constitutes the sole difference between KAP and MARS, it must be mentioned that

5.2 Benchmark Programs

The selected mapping schemes are tested on a suite of five benchmark programs which have been chosen to present a range of different features; their common characteristic is that they comprise non-rectangular loop nests, that is, the amount of work performed by each iteration of the outermost parallel loop varies from iteration to iteration.

The benchmark suite contains the following programs:

- *Upper Triangular Matrix Addition*: This application is chosen as a classical example of a loop nest where each iteration of the outer loop performs a different amount of work; the corresponding code is shown in Figure 5.1.a.
- *Adjoint Convolution*: This benchmark has been used to evaluate the effectiveness of dynamic loop mapping strategies [76, 153]; the code is shown in Figure 5.1.b. Although the code consists of a loop nest which has a form similar to that of upper triangular matrix multiplication, the size of the arrays involved is considerably smaller.
- *Upper Triangular Matrix Multiplication*: The multiplication of two upper triangular $n \times n$ matrices has been chosen as an example of a loop nest where there are two inner loops whose number of iterations depends on the index of the outer loop; the corresponding code is shown in Figure 5.1.c.
- *Banded SYR2K*: Banded symmetric rank-$2k$ update (SYR2K [56]) has been chosen as an example of a loop nest with more complicated bounds; the corresponding code is shown in Figure 5.1.d.[2]
- TRED2: The fifth and final benchmark, TRED2, is a routine of approximately 100 lines of code taken from the eigenvalue solver package EISPACK [79, 202] which has been used extensively for experimental measurements [90, 91, 167, 169]; the version of the code we use is shown in Appendix C.2. In order to parallelise the code, we considered three loop nests which between them account for over 93% of the overall execution time when `N` = 256, and over 99% of the overall execution time when `N` = 1024. These loop nests are marked in the code presented in Appendix C.2 and they are also reproduced in Figure 5.2. All three are nested in the body of an outer sequential loop with lower bound `2`, upper bound `N`, and loop index `II` (for the first two), and `I` (for the third).

For all five benchmarks, KAP, MARS and CYC are implemented. BCS is implemented only for the first three benchmarks; it is not applicable to banded SYR2K (because of the `MIN` and `MAX` functions in the loop bounds) and the parallelised loops of TRED2 (which are not perfectly nested).

The implementation details of the schemes based on the properties of the canonical loop nests deserve further explanation.

The loop nests corresponding to the first two benchmarks, upper triangular matrix addition, and adjoint convolution, have depth 2 and are canonical, according to Definition 4.1; thus, the partitioning scheme described in the proof of Theorem 4.1 can lead to perfect load balance. Furthermore, since both

parallelisation in MARS is based on a strategy of partitioning data rather than loop iterations [29]. In the case where the loop indices are also the subscripts of the arrays present in a loop nest, this approach may coincide with chunk scheduling; as a result, in subsequent examples, KAP and MARS are treated as having identical values of load imbalance.

[2] SYR2K performs the operation $C \leftarrow C + AB^T + BA^T$, where C is a $n \times n$ symmetric matrix and A, B are $n \times k$ matrices. The banded version in Figure 5.1.c has been transformed for parallelism and locality as described in [120]; the original code has the following form:

```
      DO I=1,N
         DO J=I,MIN(I+2*BB-2,N)
            DO K=MAX(I-BB+1,J-BB+1,1),MIN(I+BB-1,J+BB-1,N)
               C(I,J-I+1)=C(I,J-I+1)+A(K,I-K+BB)*B(K,J-K+BB)
     &                              +A(K,J-K+BB)*B(K,I-K+BB)
            ENDDO
         ENDDO
      ENDDO
```

```
DOALL J=1,N
   DO I=1,J
      A(I,J)=B(I,J)+C(I,J)
   ENDDO
ENDDO
```

a) Upper Triangular Matrix Addition.

```
DOALL I=1,N
   DO J=I,N
      A(I)=A(I)+B(J)*C(J-I+1)
   ENDDO
ENDDO
```

b) Adjoint Convolution.

```
DOALL J=1,N
   DO I=1,J
      DO K=I,J
         A(I,J)=A(I,J)+B(I,K)*C(K,J)
      ENDDO
   ENDDO
ENDDO
```

c) Upper Triangular Matrix Multiplication.

```
 DOALL I=1,MIN(N,2*BB-1)
    DO J=MAX(1-BB,1-N),MIN(BB-I,N-I)
       DO K=MAX(1,I+J),MIN(N+J,N)
          C(-I-J+K+1,I)=C(-I-J+K+1,I)+A(K,-I-J+BB+1)*B(K,-J+BB)
&                                   +A(K,-J+BB)*B(K,-I-J+BB+1)
       ENDDO
    ENDDO
 ENDDO
```

d) Banded SYR2K.

Figure 5.1: Benchmark applications.

```
L=N+1-II
DOALL J=1,L
   Z(J,L+1)=D(J)
   G=E(J)
   DO K=1,J
      G=G+Z(J,K)*D(K)
   ENDDO
   DO K=J+1,L
      G=G+Z(K,J)*D(K)
   ENDDO
   E(J)=G
ENDDO
```

a) First loop nest.

```
 L=N+1-II
 DOALL J=1,L
    TEMP1=D(J)
    TEMP2=E(J)
    DO K=J,L
       Z(K,J)=Z(K,J)
&            -TEMP1*E(K)
&            -TEMP2*D(K)
    ENDDO
 ENDDO
```

b) Second loop nest.

```
L=I-1
DOALL J=1,L
   G=0.0D0
   DO K=1,L
      G=G+Z(K,I)*Z(K,J)
   ENDDO
   DO K=1,L
      Z(K,J)=Z(K,J)-G*D(K)
   ENDDO
ENDDO
```

c) Third loop nest.

Figure 5.2: Parallelised loops in TRED2.

Benchmark	Cano-nical	KAP	MARS	CYC	BCS	CAN		
						CAN-2	CAN-3	CAN-3t
Upper Triangular Matrix Addition	yes	•	•	•	•	•		
Adjoint Convolution	yes	•	•	•	•	•		
Upper Triangular Matrix Multiplication	yes	•	•	•	•	•	•	
Banded SYR2K	no	•	•	•		•	•	•
TRED2	yes	•	•	•		•		

Table 5.1: Benchmarks and mapping schemes implemented.

loop nests are perfectly nested, the results of Lemma 4.3 can be used whenever `N` is not a multiple of $2p$, where p is the number of processors used. This version is denoted by CAN.

The loop nests corresponding to upper triangular matrix multiplication and banded SYR2K have depth 3 and, in each case, there are two nested inner loops having bounds depending on the index of the outermost loop. Thus, the partitioning scheme described in the proof of Theorems 4.3 and 4.4 (for $m = 3$) is implemented for both cases; this scheme is denoted by CAN-3. Note that the former loop nest is canonical, according to Definition 4.2 (and therefore perfect load balance may be achieved), but the latter is not. However, converting the `MIN` and `MAX` functions to `IF` statements, and removing the latter by index set splitting (see Section 4.3.3), banded SYR2K can be transformed into four consecutive canonical loop nests (this transformation is presented in Appendix C.1), which, upon partitioning (see Figure C.8), may lead to perfect load balance; this version is denoted by CAN-3t. For comparison, the partitioning scheme described in the proof of Theorem 4.1 is also implemented; this version is denoted by CAN-2.[3]

Finally, the first two parallelised loops of TRED2 are canonical, in accordance with Definition 4.1 (or as extended by Corollary 4.2, for the first loop nest); thus, the partitioning scheme described in the proof of Theorem 4.1 may lead to perfect load balance. The corresponding partitioned version of the program is denoted by CAN.[4]

The situation is summarised in Table 5.1. For each benchmark, it is shown whether or not the corresponding loop nest is canonical, and a • entry indicates that a particular mapping scheme has been implemented. For brevity, in the following sections, CAN-2 (that is, the mapping scheme described in the proof of Theorem 4.1) is shortened to CAN whenever this is the sole mapping scheme based on the notion of a canonical loop nest.

5.3 Evaluating the Load Imbalance

Firstly, a comparison of the load imbalance, L, and the relative load imbalance, L_R, resulting after applying each mapping scheme to each benchmark, and computed as suggested by Equations (4.1) and (4.2), respectively, is carried out.

For the first four benchmarks (which comprise perfectly nested loops), the load imbalance is computed in terms of the work associated with the assignment statement of the loop body;[5] for the fifth benchmark,

[3] For both upper triangular matrix multiplication and banded SYR2K, whenever `N` is not a multiple of $2p$ (for CAN-2), or $2p^2$ (for CAN-3), partitioning along the index of the outermost loop takes place by decreasing or increasing order depending on which scheme returns the smallest value of load imbalance. For CAN-3t, partitioning is based on a fixed strategy of partitioning the index of the outermost loop by: *a)* decreasing order in the first and second loop nests, and *b)* increasing order in the third and fourth loop nests (see Figure C.8 in Appendix C.1). This strategy is not optimal; a smaller value of load imbalance may be found after comparing the load imbalance resulting from each possible combination of partitioning for the four loop nests (a total of $4p^2$ combinations, assuming that the two loop nests consisting of only one iteration of the outer loop can be executed by any processor). However, the number of different combinations may be too high, and the potential benefit in load balance too small, to make it worthwhile pursuing such a comparison.

[4] Note that the third loop nest is rectangular; therefore, the partitioning schemes described in Section 4.2 can be applied. Furthermore, since the first two loop nests are not perfectly nested, the results of Lemma 4.3 are not applicable; thus, in both cases, the index of the outer loop is partitioned by decreasing order.

[5] This means that a value of 100 denotes a load imbalance equal to $100W$, where W is the amount of work corresponding to the assignment statement comprising the loop body.

N	Mapping scheme	Number of processors									
		2		4		8		12		16	
		L	L_R	L	L_R	L	L_R	L	L_R	L	L_R
400	KAP/MARS	20000	.333	15000	.428	8750	.466	5472	.450	4688	.483
	CYC	100	.002	150	.007	175	.017	185	.027	188	.036
	BCS	86	.002	119	.006	69	.007	280	.040	298	.056
	CAN	0	.000	0	.000	0	.000	117	.017	188	.036
800	KAP/MARS	80000	.333	60000	.428	35000	.466	21747	.449	18750	.484
	CYC	200	.001	300	.004	350	.009	368	.014	375	.018
	BCS	261	.002	161	.002	393	.010	375	.014	549	.027
	CAN	0	.000	0	.000	0	.000	236	.009	0	.000
1200	KAP/MARS	180000	.333	135000	.428	78750	.466	55000	.478	42188	.484
	CYC	300	.001	450	.002	525	.006	550	.009	563	.012
	BCS	324	.001	170	.001	724	.008	654	.011	663	.014
	CAN	0	.000	0	.000	0	.000	0	.000	563	.012
1600	KAP/MARS	320000	.333	240000	.428	140000	.467	86992	.449	75000	.484
	CYC	400	.001	600	.002	700	.004	735	.007	750	.009
	BCS	254	.000	845	.003	499	.003	987	.009	800	.010
	CAN	0	.000	0	.000	0	.000	467	.004	0	.000

Table 5.2: Expected load imbalance and relative load imbalance for upper triangular matrix addition.

TRED2 (where the parallelised loops are not perfectly nested), the load imbalance is computed as CPU cycles.[6] The results, with L rounded to the nearest integer, are shown in Tables 5.2 (upper triangular matrix addition), 5.3 (adjoint convolution), 5.4 (upper triangular matrix multiplication), 5.5 (banded SYR2K), and 5.6 (TRED2).

As a general trend, KAP and MARS exhibit a high value of load imbalance while the remaining schemes exhibit significantly smaller values; the CAN schemes generally exhibit the smallest value of load imbalance.

Some further remarks can be made concerning the results shown in Table 5.4 (for upper triangular matrix multiplication) and Table 5.5 (for banded SYR2K), where more than one scheme based on the notion of canonical loop nests is implemented. In both tables, CAN-3 exhibits, on average, the smallest value of load imbalance.[7] For banded SYR2K, CAN-3t, albeit composed of canonical loop nests, results in a smaller value of load imbalance than CAN-3 only on 2 processors; the reason may be attributed mainly to the nature of the partitioned code (for instance, two of the loop nests contain only a single iteration), as well as to the use of a fixed strategy for partitioning. For upper triangular matrix multiplication, CAN-2 exhibits a value of load imbalance which is significantly higher than that of CAN-3 (see Table 5.4); this is also the case for $\mathtt{N} = 1024$, $\mathtt{BB} = 256$, in banded SYR2K. However, for $\mathtt{N} = 512$, $\mathtt{BB} = 64$, CAN-2 exhibits a value of load imbalance which is comparable to that of CAN-3 (even outperforming it on 2 processors) and always less that that of CAN-3t; the relatively small number of iterations seems to be the reason for this.

The above remarks indicate that, as a general approach, partitioning using the scheme corresponding to the depth of the loop nest is preferable. Although, in certain cases, schemes other than those described in Chapter 4 may lead to a smaller value of load imbalance, the differences are not expected to be significant in practice; determining the trade-off is an interesting exercise only from a numerical point of view.

[6] In order to compute the load imbalance, we consider the amount of work performed by the body of each loop in the nest. Thus, for the first loop nest, let W_J represent the amount of work performed by the three assignment statements which are executed only by the J loop, W_{K_1} represent the amount of work performed by the body of the first K loop, and W_{K_2} represent the amount of work performed by the body of the second K loop; similarly, we consider W_J and W_K for the second loop nest, and W_J, W_{K_1} and W_{K_2} for the third loop nest. An estimate for each of these values is based on the number of corresponding CPU cycles (as indicated by translating each FORTRAN statement to assembly language), namely: $W_J = 97$, $W_{K_1} = 51$, and $W_{K_2} = 51$, for the first loop nest, $W_J = 50$, and $W_K = 53$, for the second loop nest, and $W_J = 17$, $W_{K_1} = 46$, and $W_{K_2} = 39$, for the third loop nest.

[7] In certain cases, the number of partitions along the index of the outermost loop required for CAN-3 exceeds the number of iterations available (for instance, in upper triangular matrix multiplication, for $\mathtt{N} = 256$ and 12 or more processors, as well as, in banded SYR2K, for $\mathtt{N} = 512$, $\mathtt{BB} = 64$ and 8 or more processors); even so, CAN-3 results in the smallest value of load imbalance.

N	Mapping scheme	Number of processors									
		2		4		8		12		16	
		L	L_R	L	L_R	L	L_R	L	L_R	L	L_R
8000	KAP/MARS	8000000	.333	6000000	.429	3500000	.467	2446889	.478	1875000	.484
	CYC	2000	.000	3000	.000	3500	.001	3668	.001	3750	.002
	BCS	1653	.000	1000	.000	2114	.001	3030	.001	3417	.002
	CAN	0	.000	0	.000	0	.000	2336	.001	0	.000
16000	KAP/MARS	32000000	.333	24000000	.429	14000000	.467	9787556	.478	7500000	.484
	CYC	4000	.000	6000	.000	7000	.000	7335	.001	7500	.001
	BCS	4955	.000	4704	.000	10732	.001	6292	.001	8275	.001
	CAN	0	.000	0	.000	0	.000	4667	.000	0	.000

Table 5.3: Expected load imbalance and relative load imbalance for adjoint convolution.

N	Mapping scheme	Number of processors									
		2		4		8		12		16	
		L	L_R	L	L_R	L	L_R	L	L_R	L	L_R
256	KAP/MARS	1056768	.428	923648	.566	577024	.620	356749	.602	319360	.644
	CYC	8256	.006	12416	.017	14560	.040	15331	.061	15760	.082
	BCS	382	.000	13271	.018	7183	.020	6329	.026	9828	.053
	CAN-2	262144	.156	229376	.245	143360	.288	82091	.258	79360	.310
	CAN-3	0	.000	0	.000	0	.000	50	.000	512	.003
1024	KAP/MARS	67239936	.428	58818560	.567	36757504	.621	22978604	.606	20346880	.645
	CYC	131328	.001	197120	.004	230272	.010	241550	.016	247360	.022
	BCS	151255	.002	118940	.003	193075	.009	322800	.021	281770	.025
	CAN-2	16777216	.158	14680064	.247	9175040	.290	6228806	.294	5079040	.312
	CAN-3	0	.000	0	.000	0	.000	48713	.003	0	.000

Table 5.4: Expected load imbalance and relative load imbalance for upper triangular matrix multiplication.

N, BB	Mapping scheme	Number of processors									
		2		4		8		12		16	
		L	L_R	L	L_R	L	L_R	L	L_R	L	L_R
512, 64	KAP/MARS	1004896	.350	764592	.450	447832	.490	331685	.516	240300	.507
	CYC	15360	.008	23056	.024	26936	.055	28645	.084	28940	.110
	CAN-2	992	.001	19216	.020	17920	.037	12597	.039	9920	.041
	CAN-3	8192	.004	1024	.001	128	.000	1633	.005	560	.002
	CAN-3t	2080	.001	31248	.032	60360	.114	73227	.191	31668	.120
1024, 256	KAP/MARS	30758272	.367	23767744	.473	13981024	.513	9924928	.529	7514800	.531
	CYC	114688	.002	172096	.006	200928	.015	211168	.023	215600	.031
	CAN-2	1851264	.034	1478464	.053	1146880	.079	692496	.073	537360	.075
	CAN-3	524288	.010	65536	.002	8192	.001	22392	.003	1024	.000
	CAN-3t	128	.000	244800	.009	252960	.019	510110	.054	452880	.064

Table 5.5: Expected load imbalance and relative load imbalance for banded SYR2K.

N	Mapping scheme	loop nest	Number of processors									
			2		4		8		12		16	
			L	L_R	L	L_R	L	L_R	L	L_R	L	L_R
256	KAP/	1st	424000	.003	634368	.009	736288	.020	776208	.031	780720	.042
	MARS	2nd	37054016	.331	28117696	.429	16780736	.473	11990030	.490	9389120	.501
		3rd	697408	.003	1043392	.009	1210944	.020	1276609	.031	1283840	.042
		all 3	38175424	.077	29795456	.116	18727968	.141	14042848	.156	11453680	.168
	CYC	1st	424000	.003	634368	.009	736288	.020	776208	.031	780720	.042
		2nd	437376	.006	657760	.017	771344	.040	812171	.061	834920	.082
		3rd	697408	.003	1043392	.009	1210944	.020	1276609	.031	1283840	.042
		all 3	1558784	.003	2335520	.010	2718576	.023	2864989	.036	2899480	.049
	CAN	1st	424000	.003	634368	.009	736288	.020	776208	.031	780720	.042
		2nd	327136	.004	565520	.015	698280	.036	771458	.058	756180	.075
		3rd	697408	.003	1043392	.009	1210944	.020	1276609	.031	1283840	.042
		all 3	1448544	.003	2243280	.010	2645512	.023	2824276	.036	2820740	.047
1024	KAP/	1st	6709504	.001	10057728	.002	11718784	.005	12295824	.008	12523200	.011
	MARS	2nd	2371197184	.333	1783614208	.429	1046515456	.468	735163940	.481	567115520	.488
		3rd	11145472	.001	16707328	.002	19466496	.005	20425047	.008	20802560	.011
		all 3	2389052160	.076	1810379264	.111	1077700736	.129	767884810	.137	600441280	.142
	CYC	1st	6709504	.001	10057728	.002	11718784	.005	12295824	.008	12523200	.011
		2nd	6959616	.001	10446208	.004	12203072	.010	12800705	.016	13108640	.022
		3rd	11145472	.001	16707328	.002	19466496	.005	20425047	.008	20802560	.011
		all 3	24814592	.001	37211264	.003	43388352	.006	45521575	.009	46434400	.013
	CAN	1st	6709504	.001	10057728	.002	11718784	.005	12295824	.008	12523200	.011
		2nd	5216128	.001	9100352	.004	11340960	.009	12239432	.015	12488400	.021
		3rd	11145472	.001	16707328	.002	19466496	.005	20425047	.008	20802560	.011
		all 3	23071104	.001	35865408	.002	42526240	.006	44960302	.009	45814160	.012

Table 5.6: Expected load imbalance and relative load imbalance for the three parallelised loops of TRED2.

Mapping scheme	Number of processors							
	2		4		6		8	
	%	L_R	%	L_R	%	L_R	%	L_R
CYC	7.3(00.0)	.0012	2.6(00.0)	.0035	2.9(00.0)	.0057	2.2(00.0)	.0080
BCS	17.9(17.7)	.0008	22.8(19.9)	.0027	18.4(17.9)	.0050	14.5(14.2)	.0076
CAN	82.3(74.8)	.0003	80.0(74.7)	.0013	81.9(78.7)	.0024	85.8(83.3)	.0035
Mapping scheme	Number of processors							
	10		12		14		16	
	%	L_R	%	L_R	%	L_R	%	L_R
CYC	2.0(00.0)	.0103	1.8(00.0)	.0125	2.2(00.0)	.0148	1.5(00.0)	.0170
BCS	11.9(11.7)	.0102	10.7(10.7)	.0131	7.3(07.2)	.0161	7.5(07.3)	.0188
CAN	88.3(86.1)	.0047	89.3(87.5)	.0058	92.8(90.5)	.0070	92.7(91.0)	.0081

Table 5.7: Fraction of cases (%) where each mapping scheme returns the smallest value of expected load imbalance, and average relative load imbalance of triangular matrix addition for all $\texttt{N} \in [400, 1600]$.

In all the cases examined in Table 5.2, CAN exhibits the smallest value of load imbalance. However, in most of these cases, N is a multiple of $2p$, where p is the number of processors used, and, as a result of Theorem 4.1, CAN leads to perfect load balance. In order to examine the behaviour of CAN, as compared to CYC and BCS, in a wider range of cases, we computed the expected load imbalance for these three schemes for all (integer) values of N in the interval $[400, 1600]$. Table 5.7 shows, in percentage terms, the number of times each scheme would result in the smallest value of load imbalance,[8] as well as the average value of the relative load imbalance, L_R, using a varying number of processors. It can be seen that CAN returns the smallest value of load imbalance in more than 80% of the cases; this fraction tends to increase as the number of processors increases.

However, the difference in load imbalance between the three mapping schemes is relatively small (as can also be inferred by observing the corresponding values of relative load imbalance), and its overall effect on the actual performance may not be significant.

Apart from evaluating the load imbalance exhibited by each mapping scheme, the results of Tables 5.2 through 5.6 can also be used to estimate the performance in each case; such an estimate would be based on the assumption that load imbalance is the dominant overhead (see Section 1.4.3.2). To what extent this assumption leads to realistic estimates is indicated by the experimental results presented in the next section.

5.4 Experimental Results

The parallelised programs for each benchmark are run on the KSR1 for the same problem sizes used in Section 5.3. The methodology consists of running each parallelised program five times, for any given problem size and number of processors. Runs are grouped in *experiments*; each experiment consists of one run of each parallelised program for a given problem size and for a varying number of processors. As far as possible, experiments are run when no jobs are sent by other users of the KSR1; whenever the execution time of a particular program run exhibits a significant deviation from other runs of the same program and with the same parameters, the whole experiment is repeated. The final execution times shown for each particular case are computed as the arithmetic mean of the execution time of all the respective runs carried out.

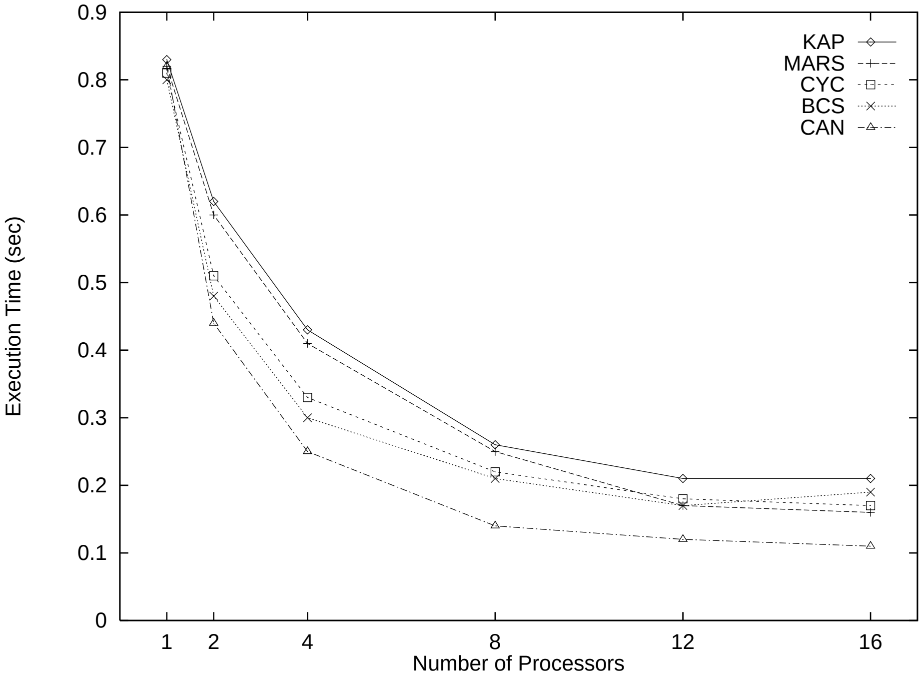
The execution time, on the KSR1, of the parallelised programs for upper triangular matrix addition for two different values of N is depicted graphically in Figure 5.3.a ($\mathtt{N} = 800$) and Figure 5.3.b ($\mathtt{N} = 1600$). CAN performs consistently best and KAP performs worst.[9] While this might have been anticipated from the values of load imbalance shown in Table 5.2, the latter do not suffice to provide an adequate explanation for the performance of MARS, CYC, and BCS; in particular, even though CYC and BCS have an expected value of load imbalance comparable to that of CAN, they run significantly slower than CAN (from approximately 10% up to 70%), while MARS, which has a significantly higher expected value of load imbalance than the remaining schemes, tends to be comparable with BCS and CYC, even outperforming them on 16 processors.

An analysis of the timing results shows that a large amount of the overall execution time (from approximately 30%, when using 2 processors, up to 75%, when using 16 processors) constitutes *system time* (i.e., time spent on system operations such as memory handling or library calls); this is due to the relatively large size of the arrays involved (approximately 4.9 Mbytes each when $\mathtt{N} = 800$, or 19.5 Mbytes each when $\mathtt{N} = 1600$). Thus, load imbalance constitutes only a fraction of the overall overhead.

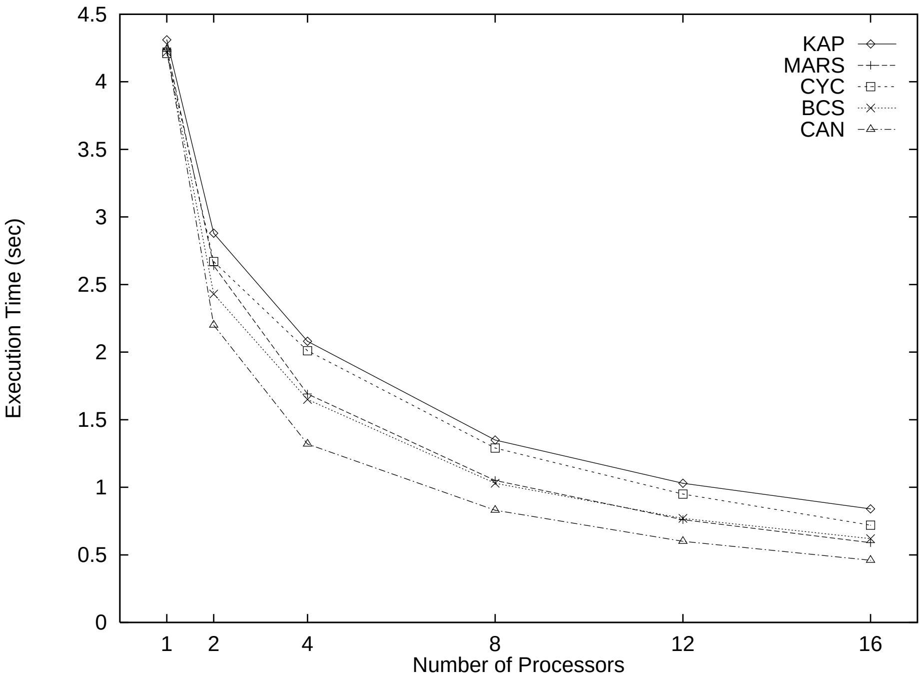
The execution time of the partitioned programs for adjoint convolution is shown in Figures 5.4.a ($\mathtt{N} = 8000$) and 5.4.b ($\mathtt{N} = 16000$). In both graphs, KAP and MARS perform worst of all while CYC, BCS, and CAN perform best of all; a slight variation arises for CYC when $\mathtt{N} = 8000$ and $p = 16$, which seems

[8] The number enclosed by parentheses shows, again in percentage terms, the number of times for which *only* the respective mapping scheme would result in the smallest value of load imbalance. Note that, in several cases, more than one scheme may result in the same, smallest value of load imbalance (cf., for instance, the case $\mathtt{N} = 1200$, $p = 16$, in Table 5.2, where both CYC and CAN exhibit the same value of load imbalance which is the smallest amongst the mapping schemes considered); consequently, the percentage values for CYC, BCS and CAN do not necessarily add up to 100.

[9] In this section, the phrase 'the code partitioned by' is implied before any acronym.

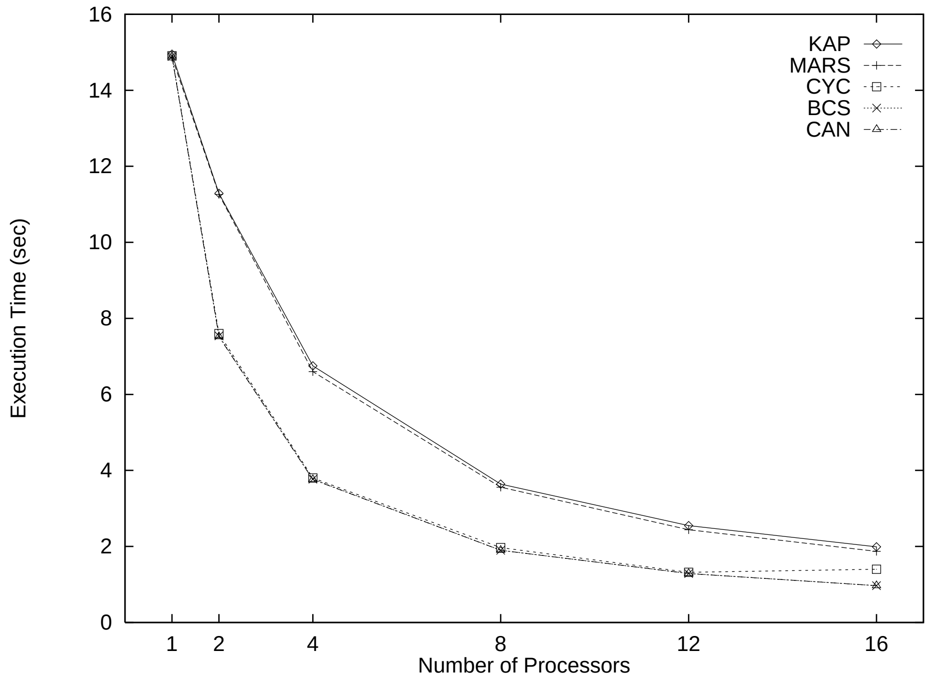


a) $\mathtt{N} = 800$.

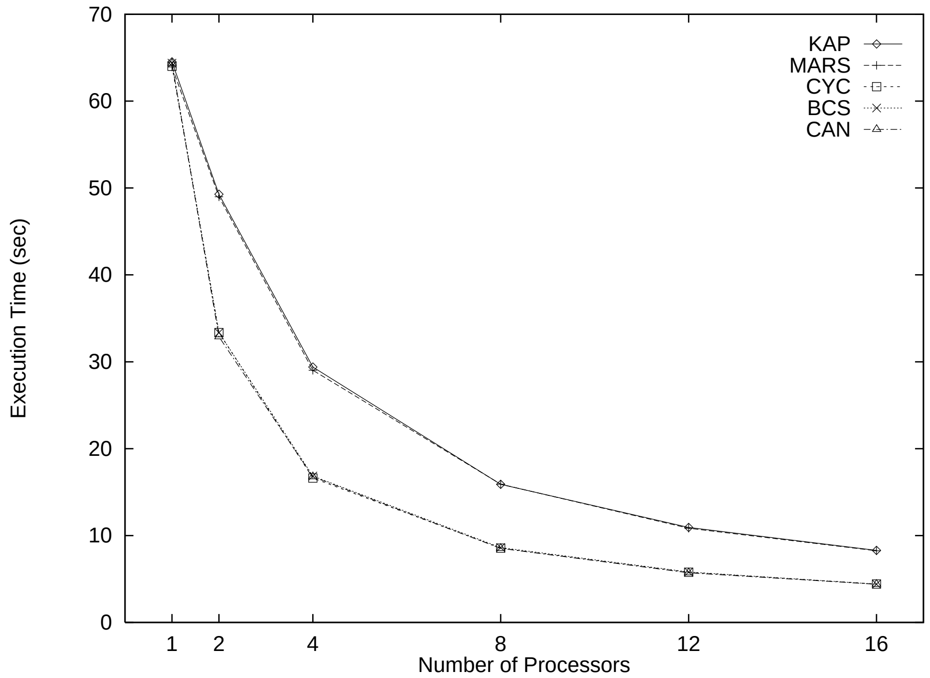


b) $\mathtt{N} = 1600$.

Figure 5.3: Execution time of mapping schemes on the KSR1 for upper triangular matrix addition.

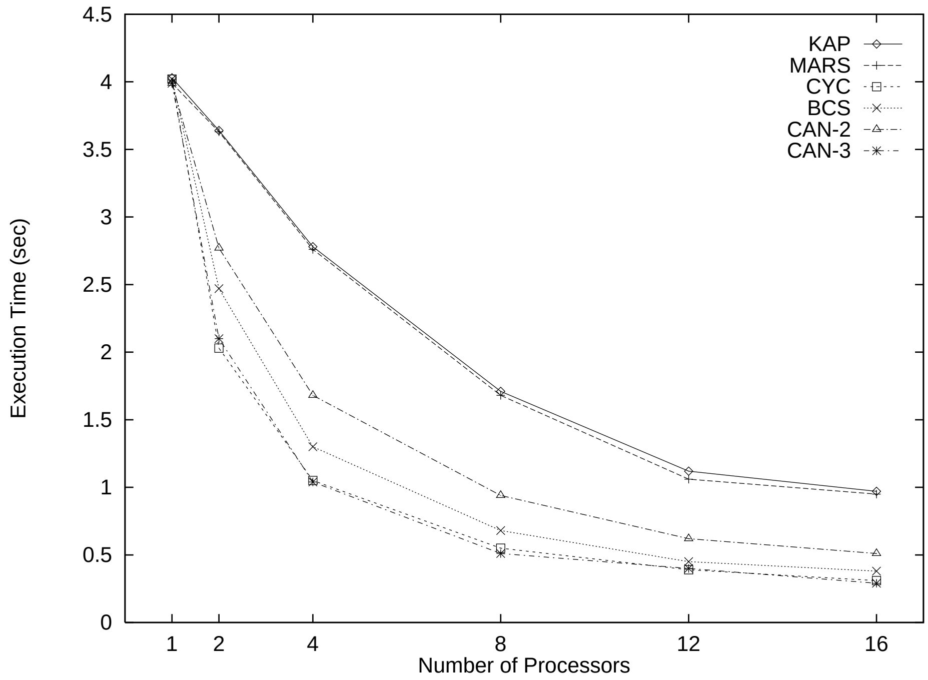


a) $\mathtt{N} = 8000$.

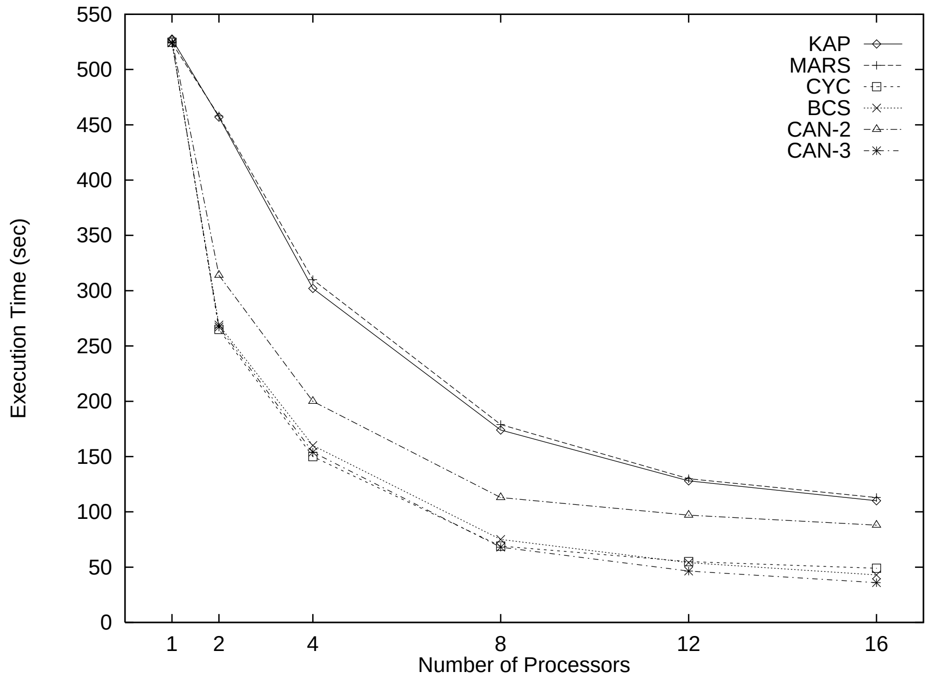


b) $\mathtt{N} = 16000$.

Figure 5.4: Execution time of mapping schemes on the KSR1 for adjoint convolution.

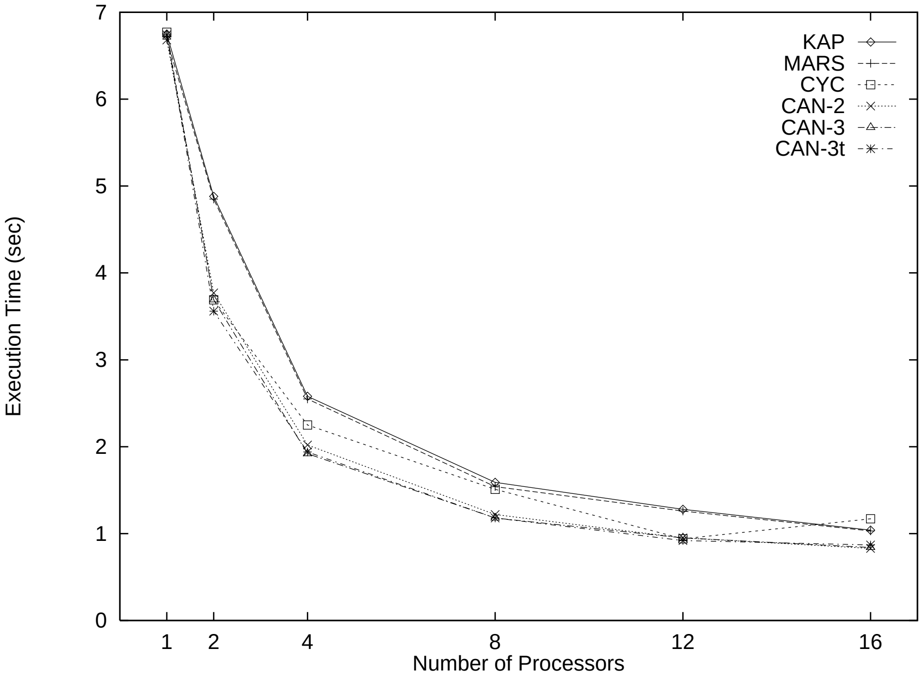


a) $\mathtt{N} = 256$.

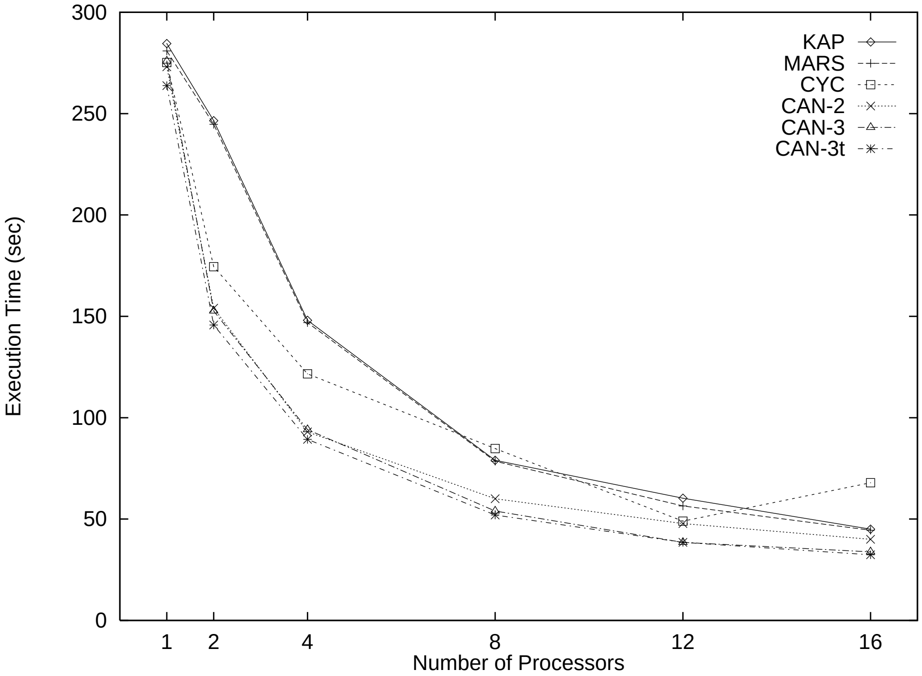


b) $\mathtt{N} = 1024$.

Figure 5.5: Execution time of mapping schemes on the KSR1 for upper triangular matrix multiplication.



a) N $= 512$, BB $= 64$.



b) N $= 1024$, BB $= 256$.

Figure 5.6: Execution time of mapping schemes on the KSR1 for banded SYR2K.

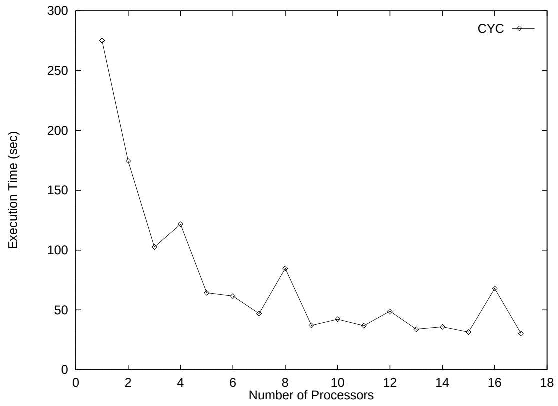


Figure 5.7: Execution time of CYC on the KSR1 for banded SYR2K when N = 1024, BB = 256.

to be due to false sharing (see Section 2.3.1.1). The execution times obtained are consistent with those anticipated from the expected values of load imbalance shown in Table 5.3; in contrast with the results for upper triangular matrix addition, the relatively smaller size of the arrays in this benchmark renders load imbalance the dominant overhead.

Also consistent with the performance anticipated from the expected values of load imbalance shown in Table 5.4 are the execution times of the partitioned programs for upper triangular matrix multiplication shown in Figures 5.5.a (N = 256) and 5.5.b (N = 1024). In both graphs, KAP and MARS perform worst of all while CAN-3 performs best; the performance of CAN-3 is comparable with that of CYC and BCS.

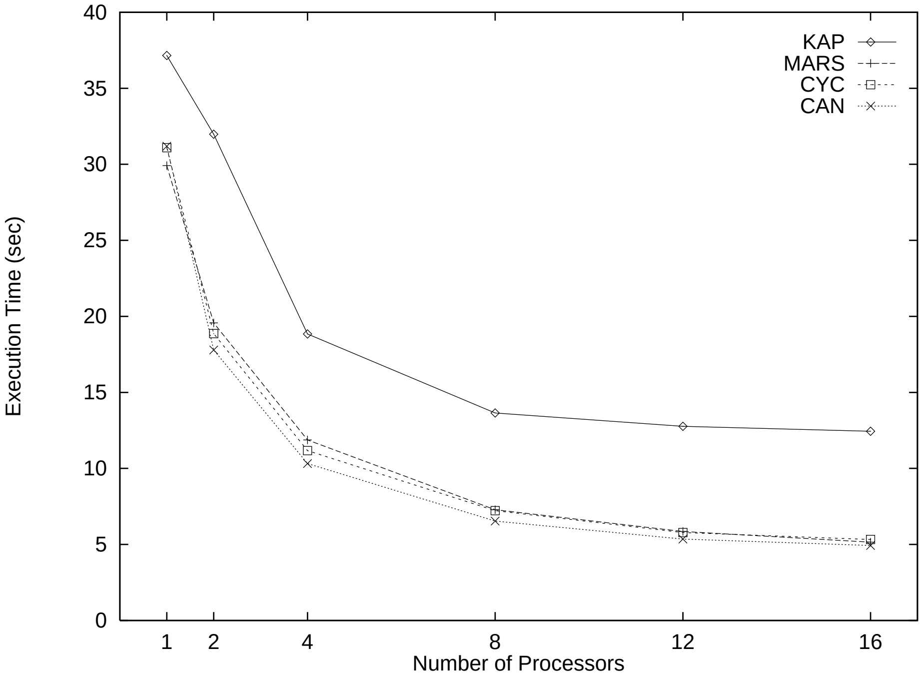
The execution time of the parallelised programs for banded SYR2K is depicted in Figures 5.6.a (N = 512, BB = 64) and (N = 1024, BB = 256). In the first case, in Figure 5.6.a, KAP and MARS perform worst of all, except when running on 16 processors where CYC performs worst of all. CAN-3t performs best of all, while equally good results are achieved by CAN-3 and, to some extent, CAN-2. CYC exhibits somewhat odd behaviour; it performs nearly best of all when running on 12 processors, but performs worst of all when running on 16 processors, and nearly worst when running on 8 processors.

Similar remarks can be made about the results depicted in Figure 5.6.b. CAN-3t performs best of all when using fewer than 12 processors, while its performance is comparable to that of CAN-3, particularly when using more than 8 processors. KAP and MARS perform worst of all except when using 8 or 16 processors; in these cases, CYC, whose performance follows the same odd pattern of behaviour as in Figure 5.6.a, performs worst of all.

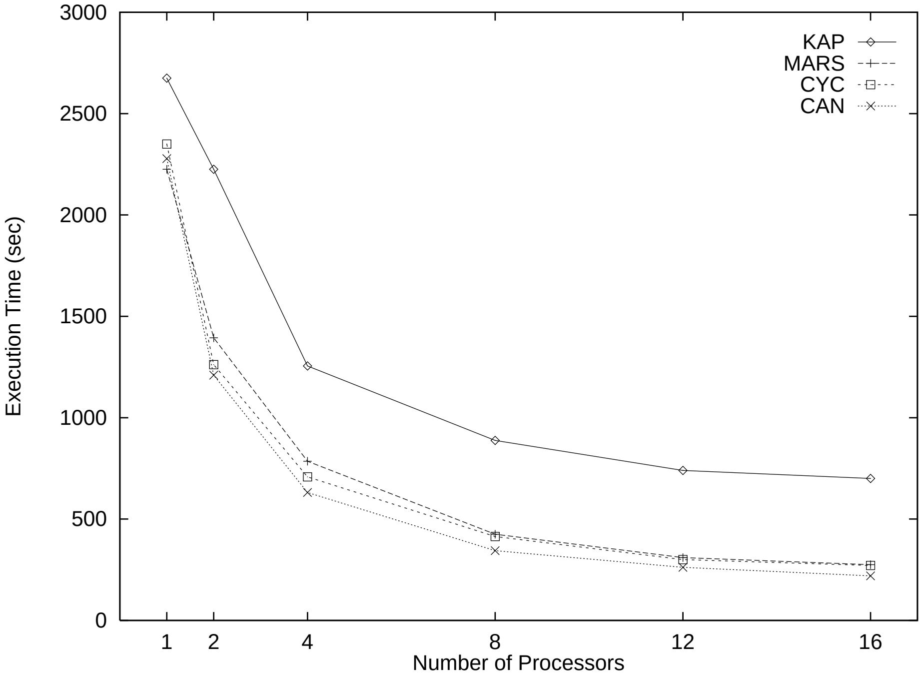
The anomalies in CYC's performance are due to poor cache behaviour, resulting in a high number of cache block replacements in cases where the number of processors is a power of 2. To illustrate this phenomenon, CYC is run, for N = 1024 and BB = 256, using all possible numbers of processors from 1 to 17; its performance is depicted in Figure 5.7. It seems that, the larger the number of 2's in the integer factorisation of p (the number of processors), the worse the performance becomes.

Another interesting observation, in connection with the load imbalance computed in Table 5.5 and the actual performance depicted in Figure 5.6, is that, although CAN-3t nearly always exhibits a higher load imbalance than CAN-3 (except when the number of processors is 2), its actual performance is generally better than that of CAN-3 (except when running on more than 12 processors, where the difference in the anticipated load imbalance between the two partitioning schemes becomes comparatively higher); the performance gains of CAN-3t are most likely due to the absence of MIN and MAX functions in the loop bounds.

Finally, the execution time of the parallelised programs for TRED2 is depicted in Figure 5.8.a (N = 256) and 5.8.b (N = 1024). In both graphs, KAP performs worst of all; its poor performance is probably due to the comparatively high parallel start-up overhead incurred by the TILE directive (used for parallelisation by KAP), as opposed to the PARALLEL REGION directive (used for parallelisation



a) N = 256.



b) N = 1024.

Figure 5.8: Execution time of partitioning schemes on the KSR1 for TRED2.

by the remaining schemes), rather than the parallelisation strategy *per se*. In both graphs, CAN performs best of all, while CYC outperforms MARS when using fewer than 16 processors; in the latter case, it seems that the performance losses of CYC, caused by false sharing, outweigh its benefits, due to load imbalance, compared with MARS.

5.5 Concluding Remarks

The experimental results show that appropriately selected mapping schemes based on the notion of a canonical loop nest have a performance advantage over mapping schemes such as chunk scheduling, block partitioning, cyclic partitioning, or balanced chunk scheduling. It is worth remembering that the first two schemes correspond to those applied by the existing parallelising compilers, KAP and MARS, respectively, while the last two schemes are among those static loop mapping schemes which are preferred for the parallelisation of non-rectangular loop nests. Thus, the mapping schemes based on the notion of a canonical loop nest can be regarded as an extension to the mapping strategies applied by parallelising compilers in order to deal with non-rectangular loop nests.

The experimental results are generally consistent with the anticipated performance from the computed values of load imbalance. Whenever there has been a deviation from the latter (as, for instance, in upper triangular matrix addition or, to some extent, in banded SYR2K) this is due to the presence of additional overheads which diminish the impact of load imbalance. However, even in such cases, the performance of the mapping schemes based on the notion of a canonical loop nest appears to be less affected than that of other schemes.

Chapter 6

Conclusions

6.1 Summary of Results

This thesis has been motivated by the problem of mapping loop structures onto parallel computers. This is a well-studied topic in its own right, but, in our investigation, it is viewed as an instance of the more general problem of automatic parallelisation. The latter was originally examined as, mostly, a problem of detecting the parallelism inherent in programs, usually at the loop level. However, as discussed in Section 2.5, a policy of parallelising all parallelisable loop nests in a program does not necessarily yield the best performance. Parallelism must be exploited subject to its contribution towards performance gains; thus, whether it is worth applying a transformation (or a sequence of transformations) must be judged on this basis.

Following this theme, we considered in Section 1.4.3.2 a model, based on overheads, for performance prediction which a compiler might use in principle to predict the performance of the various forms of parallel code it might generate. Similar models, albeit using slightly different parameters, have been suggested recently, but it has been beyond the scope of this thesis to evaluate the merits of each of these. For our purposes, the most important consequence of using *any* model is that its accuracy in predicting performance is highly dependent on the amount of information on the program's execution behaviour with which the compiler is provided. This information may be limited, for instance, when decisions regarding the way parallelism is mapped onto a parallel architecture are deferred from compile-time to run-time; thus, we have assessed that it is desirable to investigate compile-time schemes for mapping parallel loop nests.

Using a performance model also implies that the compiler should be capable of extracting quantitative information from programs. In the context of loop nests, significant information lies in the number of times that statements of the loop body will be executed. Computing this number, at compile-time (and taking into account that symbolic variables may be involved), is the subject of Chapter 3. The suggested methodology follows an approach closely related to the way that loop nests are actually executed, thus providing an immediate explanation for certain rules that are applied (cf. Example 3.4); the algorithmic aspects of this computation are also stressed. Particular emphasis is placed on counting the number of iterations *exactly* in cases where integer functions are involved in the loop bounds. Previous related work [185, 209] gave approximate answers, but, as shown in Section 3.3.1.2, this can lead to large errors; recent work [43] computes exact answers, but only when non-symbolic variables are involved and requiring a rather expensive framework.

The methodology developed in Chapter 3 led to the introduction of the notion of canonical loop nests; these are nests of loops for each of which the upper bound is always greater than or equal to the lower bound (assuming a unit stride), and both bounds are linear expressions of the surrounding loop indices. As described in Chapter 4, canonical loop nests (upon partitioning into equal partitions along the index of the outermost loop) exhibit a workload distribution for which a standard strategy of partitioning the loop nest into partitions of equal workload can be applied; at the heart of this strategy, a solution to an old problem of Number Theory is given. In the general case, non-canonical loop nests can be handled by splitting them into a number of consecutive canonical loop nests; essentially, this makes it possible to partition any loop nest whose bounds are constant or linear functions of the loop indices.

In formulating this partitioning strategy, the primary goal has been that of minimising the overhead due to load imbalance; that is, distributing the workload *as evenly as possible*. However, this goal has not been pursued in isolation of the objective of minimising the overall overhead; thus, care has been taken to avoid options that may increase other sources of overhead. According to the experimental results, presented in Chapter 5, where mapping schemes based on the notion of the canonical loop nest generally exhibit better performance, it can be argued that this aim has been accomplished. It is particularly encouraging to find a connection between measured performance and the theoretical values for load imbalance.

In Section 1.3, it is asserted that symbolic analysis techniques can improve the compiler's 'knowledge' of a program, thus leading to more efficient parallelisation schemes; the work subsequently described can be used to justify this position. As a side issue, it is interesting to note that this work is based on a methodology for solving the computationally expensive problem of counting the number of integer points in a polytope (see Section 3.1); however, the cases usually encountered by parallelising compilers are simple instances of the general problem for which it is feasible to get answers quickly. An analogy can be drawn with the problem of finding data dependences which, as mentioned in Section 2.2.1, is NP-complete; this has not prevented the development of efficient implementations [184].

6.2 Critique

Returning to the most elegant result of this thesis, namely Theorems 4.3 and 4.4, one may feel uncomfortable that the number of partitions along the index of the outermost loop required to achieve a perfectly balanced workload distribution is related exponentially to the number of processors available; this fact can be used to argue that, in practice, many more partitions may be required than the number of iterations available. Such a case is encountered in the experiments presented in Chapter 5 (see the results for more than 12 processors in Figures 5.5.a or 5.6.a); it does not seem to affect the performance of the corresponding methods, but, in certain cases, a trade-off between the different variations may be found. Nevertheless, we believe that the importance of our result lies in connecting the depth of a loop nest, whose bounds are linear expressions of the surrounding indices, with an appropriate partitioning scheme.

The techniques described in this thesis do not consider problems connected with sparse computations; the latter are usually manipulated by indirect references which cannot be interpreted at compile-time. However, parallelising compiler technology, as a whole, is not mature enough to deal with these cases; more advanced and aggressive techniques than those currently used are needed.

From a global point of view, one may regard with scepticism the idea of compile-time mapping; it can be argued that the sheer number of hard-to-foresee factors will force deferment of key decisions to run-time. As discussed in Chapter 1, accepting this argument in its entirety contradicts the goal of automatic parallelisation and limits a compiler's capabilities; thus, a compromise must be reached. Nonetheless, the loop nests examined in this thesis belong to the class for which there are seemingly strong reasons to prefer a fully compile-time approach for mapping.

6.3 Further Work

The research presented in this thesis can be extended along two main paths. Firstly, using the techniques described in Chapter 3, quantitative estimates for sources of overheads other than load imbalance can be computed; secondly, the strategy presented in Chapter 4 can be enhanced in various ways.

Given that communication is an increasingly expensive source of overhead, even on shared memory parallel computers, quantitative estimates are highly desirable; this has been discussed partly in [72]. However, the general case constitutes a problem more demanding than load imbalance, mainly due to the different levels of memory hierarchy and the additional constraints involved.

The exploitation of data parallelism, rather than loop parallelism, is also an issue to consider. As mentioned in Section 1.4.2.1, in many cases, the former turns out to be equivalent to the latter; thus, finding an appropriate array distribution may result as a trivial consequence of having a one-to-one correspondence between loop iterations and array elements. However, requirements, such as keeping the same array distribution strategy throughout a program consisting of multiple loop nests, may inflict inevitable communication overhead; in this case, techniques are needed to trade off load imbalance against communication.

On a different matter, the partitioning strategy presented in Chapter 4 can be amended to cover cases where integer functions exist within the loop bounds. Quantitative evaluation of the various trade-offs arising may also be carried out, but it is not expected to be of high significance in practice. More important, perhaps, is to consider a *blocked* version of the strategy, wherein it is combined with loop tiling (see Section 2.2.2.2). At a theoretical level, a proof that the number of partitions along the index of the outer loop suggested by Theorem 4.4 is optimal would be interesting.

Finally, remember that this thesis was motivated by a practical problem. Thus, a full implementation of the presented work, tested on a wide spectrum of applied problems, may pose new challenges for further investigation. Adopting Donald Knuth's words, that "the best theory is inspired by practice and the best practice is inspired by theory" [130], then this task is perhaps the most urgently required sequel.

Appendix A

Proof of Lemma 4.3

Lemma 4.3 *Consider a canonical perfect loop nest (see Definition 4.1) which is partitioned along the index of the outer loop into $2p$ partitions applying partitioning either by increasing or by decreasing order. Partitioning the loop nest into p partitions where the k-th partition, $0 \leq k < p$, consists of the k-th and the $(2p-k-1)$-th partition along the index of the outer loop, and assuming that the number of iterations of the outer loop, n, is not a multiple of $2p$, then:*

- *Partitioning by decreasing order results in smaller load imbalance than partitioning by increasing order when either $u_{21} > l_{21}$ and $n \bmod 2p < p$ or $u_{21} < l_{21}$ and $n \bmod 2p > p$.*
- *Partitioning by increasing order results in smaller load imbalance than partitioning by decreasing order when either $u_{21} < l_{21}$ and $n \bmod 2p < p$ or $u_{21} > l_{21}$ and $n \bmod 2p > p$.*

When $n \bmod 2p = p$, both partitioning schemes result in the same load imbalance.

Proof: Consider the canonical perfect loop nest shown in Figure A.1. Assuming that the work corresponding to the statements in the loop body is W, then the k-th partition along the index of the outer loop performs work W_k equal to

$$W_k = I_k W,$$

where I_k is the number of times the statements of the loop body are executed. Then, the k-th partition of the loop nest (which consists of the k-th and the $(2p-k-1)$-th partition along the index of the outer loop), $0 \leq k < p$, performs work equal to

$$W_k = (I_k + I_{2p-k-1})W.$$

Assuming that $n = u - l + 1$, and

$$M_k = \min(n \bmod 2p, k)$$

or

$$M_k = \max(0, k - 2p + (n \bmod 2p)),$$

```
DOALL i = l,u
        DO j = l21 i + l22, u21 i + u22
           (statements)
        ENDDO
ENDDO
```

Figure A.1: A canonical perfect loop nest.

depending on whether partitioning is applied by decreasing or increasing order, respectively, then I_k is given by

$$\begin{aligned}
I_k &= \sum_{i=l+k\lfloor n/2p\rfloor+M_k}^{l+(k+1)\lfloor n/2p\rfloor+M_{k+1}-1} \sum_{j=l_{21}i+l_{22}}^{u_{21}i+u_{22}} 1 \\
&= \sum_{i=l+k\lfloor n/2p\rfloor+M_k}^{l+(k+1)\lfloor n/2p\rfloor+M_{k+1}-1} ((u_{21}-l_{21})i+(u_{22}-l_{22}+1)) \\
&= (u_{21}-l_{21})\,S_k+(u_{22}-l_{22}+1)\left(\left\lfloor\frac{n}{2p}\right\rfloor+M_{k+1}-M_k\right), \qquad \text{(A.1)}
\end{aligned}$$

where S_k is equal to

$$S_k=(M_{k+1}-M_k)\left(l+\frac{1+M_{k+1}+M_k}{2}+k\lfloor\frac{n}{2p}\rfloor\right)+\lfloor\frac{n}{2p}\rfloor\left(M_{k+1}+l+\lfloor\frac{n}{2p}\rfloor(k+\frac{1}{2})-\frac{1}{2}\right).$$

Let

$$a=u_{21}-l_{21},\quad b=u_{22}-l_{22}+1,\quad c=\lfloor\frac{n}{2p}\rfloor(u_{22}-l_{22}+1).$$

Then I_k in (A.1) can be written as $aS_k+b(M_{k+1}-M_k)+c$; thus, we have

$$I_k+I_{2p-k-1}=a(S_k+S_{2p-k-1})+b(M_{k+1}-M_k+M_{2p-k}-M_{2p-k-1})+2c, \qquad \text{(A.2)}$$

where

$$\begin{aligned}
S_k+S_{2p-k-1} &= (M_{k+1}-M_k+M_{2p-k}-M_{2p-k-1})(l-\frac{1}{2}) \\
&+\frac{1}{2}(M_{k+1}^2-M_k^2+M_{2p-k}^2-M_{2p-k-1}^2)+\lfloor\frac{n}{2p}\rfloor(M_{2p-k}-M_{2p-k-1})(2p-k) \\
&+k\lfloor\frac{n}{2p}\rfloor(M_{k+1}-M_k)+\lfloor\frac{n}{2p}\rfloor(2p\lfloor\frac{n}{2p}\rfloor+2l-1+M_{2p-k-1}+M_{k+1}).
\end{aligned}$$

In order to remove the M's from the above expression we distinguish between:

I. Partitioning by decreasing order: (i.e. $M_k=\min(n \bmod 2p, k)$)

- **A.** For the partitions in which $n \bmod 2p \le k$, we have $M_k=M_{k+1}=M_{2p-k-1}=M_{2p-k}=n \bmod 2p$. Hence,

$$S'_{1A}=S_k+S_{2p-k-1}=\left\lfloor\frac{n}{2p}\right\rfloor\left(2p\left\lfloor\frac{n}{2p}\right\rfloor+2(n \bmod 2p)+2l-1\right).$$

- **B.** For the partitions in which $n \bmod 2p > k$ and $n \bmod 2p \le 2p-k-1$, we have $M_k=k, M_{k+1}=k+1, M_{2p-k-1}=M_{2p-k}=n \bmod 2p$. Hence,

$$S'_{1B}=S_k+S_{2p-k-1}=\left\lfloor\frac{n}{2p}\right\rfloor\left(2p\left\lfloor\frac{n}{2p}\right\rfloor+(n \bmod 2p)+2l+2k\right)+k+l.$$

- **C.** Finally, for the partitions in which $n \bmod 2p > k$ and $n \bmod 2p > 2p-k-1$, we have $M_k=k, M_{k+1}=k+1, M_{2p-k-1}=2p-k-1, M_{2p-k}=2p-k$. Hence,

$$S'_{1C}=S_k+S_{2p-k-1}=\left(\left\lfloor\frac{n}{2p}\right\rfloor+1\right)\left(2p\left\lfloor\frac{n}{2p}\right\rfloor+2p+2l-1\right).$$

II. Partitioning by increasing order: (i.e. $M_k = \max(0, k - 2p + (n \bmod 2p)))$

- **A.** For the partitions in which $n \bmod 2p \leq k$, we have $M_k = M_{k+1} = M_{2p-k-1} = M_{2p-k} = 0$. Hence,
$$S'_{2A} = S_k + S_{2p-k-1} = \left\lfloor \frac{n}{2p} \right\rfloor \left(2p \left\lfloor \frac{n}{2p} \right\rfloor + 2l - 1 \right).$$

- **B.** For the partitions in which $n \bmod 2p > k$ and $n \bmod 2p \leq 2p - k - 1$, we have $M_k = M_{k+1} = 0, M_{2p-k-1} = (n \bmod 2p) - k - 1, M_{2p-k} = (n \bmod 2p) - k$. Hence,
$$\begin{aligned} S'_{2B} = S_k + S_{2p-k-1} &= \left\lfloor \frac{n}{2p} \right\rfloor \left(2p \left\lfloor \frac{n}{2p} \right\rfloor + (n \bmod 2p) + 2p + 2l - 2k - 2 \right) \\ &\quad + (n \bmod 2p) + l - k - 1. \end{aligned}$$

- **C.** Finally, for the partitions in which $n \bmod 2p > k$ and $n \bmod 2p > 2p - k - 1$, we have $M_k = k - 2p + (n \bmod 2p)$, $M_{k+1} = k + 1 - 2p + (n \bmod 2p)$, $M_{2p-k-1} = (n \bmod 2p) - k - 1$, $M_{2p-k} = (n \bmod 2p) - k$. Hence,
$$S'_{2C} = S_k + S_{2p-k-1} = \left(\left\lfloor \frac{n}{2p} \right\rfloor + 1 \right) \left(2p \left\lfloor \frac{n}{2p} \right\rfloor + 2(n \bmod 2p) - 2p + 2l - 1 \right).$$

Observing the above-mentioned relations, regardless of the partitioning scheme used, it can be noticed that:

- If $n \bmod 2p < p$ then $S_k + S_{2p-k-1}$ results from the formula under **A** or **B**.
- If $n \bmod 2p = p$ then $S_k + S_{2p-k-1}$ results from the formula under **B**.
- If $n \bmod 2p > p$ then $S_k + S_{2p-k-1}$ results from the formula under **B** or **C**.

Now, we turn to the original problem of finding under which conditions each of the two partitioning schemes returns the highest load imbalance. We introduce the notation $M'_j, j \in \{A, B, C\}$, to represent the value of $M_{2p-k} - M_{2p-k-1} + M_{k+1} - M_k$ in each of the above-mentioned cases; apparently, $M'_A = 0$, $M'_B = 1$, $M'_C = 2$. Then, recalling (4.1), the load imbalance is given by

$$\begin{aligned} L &= \max_{0 \leq k < p} \left((I_k + I_{2p-k-1})W - \frac{n}{2p} \left(a(n + 2l - 1) + 2b \right) W \right) \\ &= I_{max} W - \frac{n}{2p} \left(a(n + 2l - 1) + 2b \right) W, \end{aligned} \tag{A.3}$$

where I_{max} results from those $S'_{ij}, M'_j, i \in \{1, 2\}, j \in \{A, B, C\}$, which maximise the expression

$$aS'_{ij} + bM'_j + 2c. \tag{A.4}$$

We want to reject the partitioning scheme which maximises (A.4) and compute the value of I_{max} (and consequently the load imbalance, by replacing I_{max} in (A.3)) returned by the other partitioning scheme. In order to do this, we proceed as follows:

- If $n \bmod 2p < p$ then:
  - If $a, b \geq 0$ then (A.4) is maximised when S'_{ij}, M'_j become maxima. We observe that, for all k, $S'_{2B} > S'_{1B}, S'_{1A}, S'_{2A}$ and $M'_B > M'_A$. Therefore, partitioning by decreasing order leads to a smaller value of I_{max}, which is given by S'_{1B}, M'_B:
$$I_{max} = a \left(n \lfloor \frac{n}{2p} \rfloor + (2 \lfloor \frac{n}{2p} \rfloor + 1)(n \bmod 2p + l - 1) \right) + b + 2c. \tag{A.5}$$

Equation (A.5) also holds if $a > 0$ and $b < 0$. Then (A.4) is maximised when S'_{ij} becomes maximum and M'_j becomes minimum. As before, S'_{2B} satisfies the first criterion, but M'_B is not a minimum. However, observing that $a(S'_{2B} - S'_{1A}) > -b$ by hypothesis (recall that $al = (u_{21} - l_{21})l \geq l_{22} - u_{22} = 1 - b$ from Definition 4.1), a smaller value of I_{max} is again achieved under partitioning by decreasing order, and is given by S'_{1B}, M'_B.

– If $a < 0$ then, by hypothesis, $b > 0$, therefore (A.4) is maximised when S'_{ij} becomes minimum and M'_j becomes maximum. We observe that $S'_{2A} < S'_{1A}, S'_{1B}, S'_{2B}$; therefore, S'_{2A} is a minimum solution but M'_A is not. Observing that $a(S'_{1B} - S'_{2A}) > -b$, by hypothesis, and $S'_{1B} < S'_{2B}$, then (A.4) is maximised for S'_{1B}, M'_B. Therefore, partitioning by increasing order leads to a smaller value of I_{max}, which is given by S'_{2B}, M'_B and is equal to:

$$I_{max} = a\left(\lfloor\frac{n}{2p}\rfloor(n + 2p + 2l - 2(n \bmod 2p)) + l\right) + b + 2c. \tag{A.6}$$

- If $n \bmod 2p = p$ then we observe that $S'_{1B} + S'_{2B}$ is constant (i.e. independent of k) and the value of S'_{1B} for a given k, $0 \leq k < p$, is the same with the value of S'_{2B} for $p - k - 1$. Therefore, the two schemes return the same value of I_{max}, given by:

$$I_{max} = a\left(\lfloor\frac{n}{2p}\rfloor(n + 2p + 2l - 2) + p + l - 1\right) + b + 2c. \tag{A.7}$$

- If $n \bmod 2p > p$ then, as before, we have:

  – If $a, b \geq 0$ then we observe that $S'_{1C} > S'_{2B}, S'_{2C}, S'_{1B}$ and $M'_C > M'_B$. Therefore, a smaller value of I_{max} is achieved when partitioning by increasing order and is given by S'_{2B} or S'_{2C} as follows:
  If $(\lfloor n/2p\rfloor + 1)(n \bmod 2p + 1 - 2p) > 1 - l - (b/a)$ then

$$I_{max} = a\left((\lfloor\frac{n}{2p}\rfloor + 1)(n + (n \bmod 2p) - 2p + 2l - 1)\right) + 2b + 2c, \tag{A.8}$$

  else

$$I_{max} = a\left(\lfloor\frac{n}{2p}\rfloor(n + 2p + 2l - 2) + (n \bmod 2p) + l - 1\right) + b + 2c. \tag{A.9}$$

  Equations (A.8) and (A.9) also hold when $a > 0$ and $b < 0$. Then (A.4) is maximised when S'_{ij} becomes maximum and M'_j becomes minimum. Observing that $a(S'_{1C} - S'_{1B}) > -b$ and $a(S'_{1C} - S'_{2B}) > -b$ by hypothesis, then a smaller value of load imbalance is achieved when partitioning by increasing order and is given by (A.8) or (A.9), as before.

  – If $a < 0$ then, by hypothesis, $b > 0$ and therefore (A.4) is maximised when S'_{ij} becomes minimum and M'_j becomes maximum. We observe that $S'_{2C} < S'_{1C}, S'_{1B}$ therefore partitioning by decreasing order results in a smaller value of I_{max}, which is given by:

$$I_{max} = a\left(\lfloor\frac{n}{2p}\rfloor(n + 2l) + l\right) + 2b + 2c. \tag{A.10}$$

The analysis so far can be summarised as follows:

- Partitioning by decreasing order results in smaller load imbalance than partitioning by increasing order when either $u_{21} > l_{21}$ and $n \bmod 2p < p$ or $u_{21} < l_{21}$ and $n \bmod 2p > p$.

- Partitioning by increasing order results in smaller load imbalance than partitioning by decreasing order when either $u_{21} < l_{21}$ and $n \bmod 2p < p$ or $u_{21} > l_{21}$ and $n \bmod 2p > p$.

QED

Based on the above, an upper bound for the load imbalance, L, can be computed. Observing Equations (A.5) through (A.10), it can be seen that the maximum value for I_{max}, and consequently L, is attained at Equation (A.8). Considering that $(n \bmod 2p) \leq 2p-1$ and $\lfloor n/2p \rfloor \leq n/2p$, we have:

$$\begin{aligned} L &= \left(a\left((\lfloor \frac{n}{2p} \rfloor + 1)(n + (n \bmod 2p) - 2p + 2l - 1) \right) + 2b + 2c \right) W \\ &\quad - \frac{n}{2p}\left(a(n+2l-1) + 2b \right) W \\ &\leq \left(a\left((\frac{n}{2p} + 1)(n + 2l - 2) \right) + 2b(\frac{n}{2p} + 1) \right) W - \frac{n}{2p}\left(a(n+2l-1) + 2b \right) W \\ &\leq \left(an\left(1 - \frac{1}{2p}\right) + 2al + 2b - 2a \right) W. \end{aligned}$$

The above quantity can be used to compute an upper bound for the relative load imbalance, L_R (assuming that $L \neq 0$):

$$\begin{aligned} L_R &= \frac{L}{L + \frac{n}{2p}\left(a(n+2l-1)+2b\right)W} = \frac{1}{1 + \frac{\frac{n}{2p}(a(n+2l-1)+2b)W}{L}} \\ &\leq \frac{1}{1 + \frac{\frac{n}{2p}(an+2al-a+2b)}{\left(an(1-\frac{1}{2p})+2al+2b-2a\right)}} \leq \frac{1}{1 + \frac{n}{2p}}. \end{aligned}$$

Clearly, when $n \gg 2p$, the relative load imbalance is close to zero.

It is interesting to notice that, in several cases, perfect load balance may be achieved, even though the index of the outermost, parallel loop is not partitioned into equal partitions. This can happen when Equation (A.2) returns the same value for all k, $0 \leq k < p$, that is, for any two x, y, $0 \leq x, y < p$, $x \neq y$, it must be the case that

$$\begin{aligned} &a(S_x + S_{2p-x-1} - S_y - S_{2p-y-1}) = \\ &\quad b(M_{y+1} - M_y + M_{2p-y} - M_{2p-y-1} - M_{x+1} + M_x - M_{2p-x} + M_{2p-x-1}). \end{aligned} \tag{A.11}$$

It can be seen that S'_{1B} and S'_{2B} depend on the value of k, while S'_{1A}, S'_{1C}, S'_{2A}, S'_{2C}, are constant for any value of k; thus, Equation (A.11) does not hold whenever both $S_x + S_{2p-x-1}$ and $S_y + S_{2p-y-1}$ are computed by S'_{1B} or S'_{2B}. Therefore, perfect load balance can be achieved only if S'_{1B} or S'_{2B} are 'used' once; this occurs when $n \bmod 2p$ has a value of 1 or $2p-1$ (in which case S'_{1B} or S'_{2B} are used only for $k=0$). Thus, substituting in (A.11), perfect load balance may arise when

$$a(S'_{1A} - S'_{1B}) = b, \quad \text{if } (n \bmod 2p) = 1,$$

or,

$$a(S'_{1B} - S'_{1C}) = b, \quad \text{if } (n \bmod 2p) = 2p-1,$$

when partitioning by decreasing order, or,

$$a(S'_{2A} - S'_{2B}) = b, \quad \text{if } (n \bmod 2p) = 1,$$

or,

$$a(S'_{2B} - S'_{2C}) = b, \quad \text{if } (n \bmod 2p) = 2p-1,$$

when partitioning by increasing order.

It can be seen that, if $(n \bmod 2p) = 1$, the first and third of the above equations have no solution (therefore, it is impossible to achieve perfect load balance), while, if $(n \bmod 2p) = 2p-1$, we have that:

- Partitioning by decreasing order results in perfect load balance if $a(u+1)+b=0$.
- Partitioning by increasing order results in perfect load balance if $a(l-1)+b=0$.

Appendix B

Balanced Chunk Scheduling

Balanced chunk scheduling is presented in Section 2.3.1.1 as a method of partitioning the outer loop of a perfect loop nest of depth 2 whose inner loop has a bound which depends on the index of the outer loop. The idea, originally proposed in [93], is to distribute the total number of iterations of the loop body among processors as evenly as possible so that load imbalance is minimised; this section provides a more general description, applying techniques used earlier in this thesis, and extends the method to other classes of loop nests.

In the case of a canonical double loop, such as the one shown in Figure A.1, partitioning the loop nest for p processors requires finding l_k, u_k, $0 \leq k < p$, where $\sum_{i=l}^{u} 1 \sum_{j=l_{21}i+l_{22}}^{u_{21}i+u_{22}} = \sum_{k=0}^{p-1} \sum_{i=l_k}^{u_k} \sum_{j=l_{21}i+l_{22}}^{u_{21}i+u_{22}} 1$, $l_0 = l$, $u_{p-1} = u$, and $l_k = u_{k-1} + 1$ for $1 \leq k < p$, such that the expression

$$\max_k \Big(\sum_{i=l_k}^{u_k} \sum_{j=l_{21}i+l_{22}}^{u_{21}i+u_{22}} 1 \Big)$$

is minimised.

Let I be the total number of iterations of the loop body, that is,

$$I = \sum_{i=l}^{u} \sum_{j=l_{21}i+l_{22}}^{u_{21}i+u_{22}} 1.$$

Then perfect load balance is achieved when $p|I$ and there are integers l_k, u_k satisfying the equation

$$\sum_{i=l_k}^{u_k} \sum_{j=l_{21}i+l_{22}}^{u_{21}i+u_{22}} 1 = \frac{I}{p},$$

for all k, $0 \leq k < p$.

Given that $l_0 = l$ and $l_k = u_{k-1} + 1$, $1 \leq k < p$, the problem of finding whether there is perfect load balance can be expressed as that of finding integers u_k, $0 \leq k < p$, such that

$$\sum_{i=l}^{u_k} \sum_{j=l_{21}i+l_{22}}^{u_{21}i+u_{22}} 1 = \frac{I(k+1)}{p}. \tag{B.1}$$

Since the loop nest is canonical, and assuming that $l \geq 0$, then (B.1) can be rewritten as

$$(u_k - l + 1)(au_k + al + 2b) = \frac{2I(k+1)}{p},$$

where $a = u_{21} - l_{21} \neq 0$ and $b = u_{22} - l_{22} + 1$; the solutions of this equation, for u_k, are given by

$$u_k = \frac{-p(a+2b) \pm \sqrt{p^2(a+2b)^2 - 4ap(-2I - 2Ik + 2bp + alp - 2blp - apl^2)}}{2ap}. \tag{B.2}$$

Since $l \geq 0$, it must be the case that $u_k \geq 0$, hence the negative solution can be rejected. However, it appears that integer solutions for u_k are not common;[1] thus, rounding u_k to the nearest integer (i.e. $\lfloor u_k + 0.5 \rfloor$) a small value of load imbalance is expected.

The above are illustrated in the following example:

Example B.1 Consider upper triangular matrix addition, which has been used as a benchmark in Chapter 5. The corresponding code (see Figure 5.1.a) is a special case of the loop nest shown in Figure A.1 for $l = 1$, $u = n$, $l_{21} = 0$, $l_{22} = 1$, $u_{21} = 1$, $u_{22} = 0$. Then $a = u_{21} - l_{21} = 1$ and $b = u_{22} - l_{22} + 1 = 0$; thus, substituting in (B.2) and rejecting the negative solutions, the values of u_k, $0 \leq k < p$, are given by

$$u_k = \frac{-p + \sqrt{p^2 - 4p(-2I - 2Ik)}}{2p}.$$

Assuming that $n = 800$ (hence, $I = 800 \cdot 801/2 = 320400$) and $p = 16$, the following solutions are obtained:

$$\begin{array}{llll} u_0 = 199.626, & u_1 = 282.520, & u_2 = 346.127, & u_3 = 399.750, \\ u_4 = 446.993, & u_5 = 489.704, & u_6 = 528.981, & u_7 = 565.539, \\ u_8 = 599.875, & u_9 = 632.351, & u_{10} = 663.240, & u_{11} = 692.753, \\ u_{12} = 721.061, & u_{13} = 748.299, & u_{14} = 774.581, & u_{15} = 800.000. \end{array}$$

Rounding the above solutions to the nearest integer, the index of the outer loop can be partitioned accordingly (see also Figure 2.7.b). □

In the case of a non-canonical double loop, balanced chunk scheduling can be applied after eliminating those values of the index of the outer loop which render the loop nest non-canonical (and for which the j loop is not executed, and therefore no statement at all is executed). More demanding is the case of a canonical loop nest of depth 2 which is not perfect; then, assuming that W_I is the amount of work corresponding to the statements which are executed only by the outermost loop, and W_J is the amount of work corresponding to the statements which are executed by both the outermost and the innermost loops, partitioning the loop nest into p processors requires finding l_k, u_k, $0 \leq k < p$, where $\sum_{i=l}^{u} 1 = \sum_{k=0}^{p-1} \sum_{i=l_k}^{u_k} 1$, $l_0 = l$, $u_{p-1} = u$, and $l_k = u_{k-1} + 1$ for $1 \leq k < p$, such that the expression

$$\max_k \left(\sum_{i=l_k}^{u_k} (W_I + \sum_{j=l_{21}i+l_{22}}^{u_{21}i+u_{22}} W_J) \right) = \max_k \left(W_J \sum_{i=l_k}^{u_k} (\frac{W_I}{W_J} + \sum_{j=l_{21}i+l_{22}}^{u_{21}i+u_{22}} 1) \right)$$

is minimised. Assuming that $r = W_I/W_J$, perfect load balance is achieved when there are integers l_k, u_k satisfying the equation

$$\sum_{i=l_k}^{u_k} (r + \sum_{j=l_{21}i+l_{22}}^{u_{21}i+u_{22}} 1) = \frac{1}{p} \sum_{i=l}^{u} (r + \sum_{j=l_{21}i+l_{22}}^{u_{21}i+u_{22}} 1),$$

for all k, $0 \leq k < p$.

Given that $l_0 = l$ and $l_k = u_{k-1} + 1$, $1 \leq k < p$, finding whether perfect load balance exists is equivalent to finding integers u_k, $0 \leq k < p$, such that

$$\sum_{i=l}^{u_k} (r + \sum_{j=l_{21}i+l_{22}}^{u_{21}i+u_{22}} 1) = \frac{k+1}{p} \sum_{i=l}^{u} (r + \sum_{j=l_{21}i+l_{22}}^{u_{21}i+u_{22}} 1).$$

Since the loop nest is canonical, and assuming that $l \geq 0$, the above equation can be rewritten as

$$(u_k - l + 1)(au_k + al + 2b + 2r) = \frac{2I(k+1)}{p},$$

[1] For instance, assuming that $a = 1, b = 0, l = 1, p = 2$, then u_0 is an integer if and only if the quantity $\sqrt{1 + 4I}$ is an odd integer. Since $I = n(n+1)/2$, the problem is equivalent to solving the diophantine equation $2n^2 + 2n + 1 = (2w+1)^2$ or $n^2 + (n+1)^2 = (2w+1)^2$; the latter has only six solutions for $0 < n < 100000$, namely for $n \in \{3, 20, 119, 696, 4059, 23660\}$.

where $a = u_{21} - l_{21} \neq 0$, $b = u_{22} - l_{22} + 1$ and $I = (u - l + 1)(au + al + 2b + 2r)/2$; solving for u_k, we get

$$u_k = \frac{-pA \pm \sqrt{p^2A^2 - 4ap(pA - ap - lpA + 2alp - apl^2 - 2I - 2Ik)}}{2ap}, \tag{B.3}$$

where $A = a + 2b + 2r$. It can be seen that for $r = 0$ (that is, when the loop is perfectly nested), (B.3) reduces to (B.2).

In the general case of a non-canonical loop nest of depth 2, it may be necessary to rely on two different expressions for computing u_k; one will cover the range of values which render the loop nest canonical, and another will cover the remaining values. More expressions may be needed when dealing with non-canonical loop nests of depth greater than 2. For canonical loop nests of depth greater than 2, the values of u_k can still be obtained by solving a polynomial of degree equal to the depth of the loop nest. The computational cost of finding these solutions, as well as the need for an estimate of the amount of work in the loop body of a non-perfect loop nest are the drawbacks of balanced chunk scheduling.

Appendix C

Codes Omitted

C.1 Transforming Banded SYR2K into Canonical Form

In this section, we demonstrate how to transform the non-canonical loop nest shown in Figure 5.1.d (banded SYR2K) to consecutive, canonical loop nests. In brief, this process consists of the following stages:

- Replace each `MIN` and/or `MAX` function from the bounds of the outermost loop containing any by an appropriate `IF` statement; eliminate the `IF` statement by splitting accordingly the iterations of the first loop preceding the `IF` and whose index is part of the logical expression of the `IF` (see Section 4.3.3). Repeat the above, for all loops from the outermost to the innermost, until the code contains no `MIN` and/or `MAX` functions and no `IF` statements.
- Transform any non-canonical loop nest into consecutive canonical loop nests as discussed in Section 4.4.2.

The original code, assuming that `N` $>$ `2*BB-1`, is shown in Figure C.1. After elimination of the `MIN` and `MAX` functions from the `J` loop (see Figure C.2), the resulting code is shown in Figure C.3. In the process of eliminating the `MIN` and `MAX` functions from the `K` loop, new `MIN` and `MAX` functions may result in the surrounding loops; these can be eliminated accordingly (see Figures C.4, C.5, and C.6). After transforming the resulting code (see Figure C.7.a) into a code containing only canonical loop nests (see Figure C.7.c), the latter can be partitioned as shown in Figure C.8.

C.2 TRED2

The code for TRED2, omitted from Section 5.2, is given in full in Figure C.9.

```
DOALL I=1,2*BB-1
   DO J=MAX(1-BB,1-N),MIN(BB-I,N-I)
      DO K=MAX(1,I+J),MIN(N+J,N)
         (statement)
      ENDDO
   ENDDO
ENDDO
```

Figure C.1: Banded SYR2K.

```
DOALL I=1,2*BB-1
   IF (1-BB.GT.1-N) THEN
      DO J=1-BB,MIN(BB-I,N-I)
         DO K=MAX(1,I+J),MIN(N+J,N)
            (statement)
         ENDDO
      ENDDO
   ELSE
      DO J=1-N,MIN(BB-I,N-I)
         DO K=MAX(1,I+J),MIN(N+J,N)
            (statement)
         ENDDO
      ENDDO
   ENDIF
ENDDO
```

a) Removing the `MAX` *function.*

```
DOALL I=1,2*BB-1
   IF (BB-I.LT.N-I) THEN
      DO J=1-BB,BB-I
         DO K=MAX(1,I+J),MIN(N+J,N)
            (statement)
         ENDDO
      ENDDO
   ELSE
      DO J=1-BB,N-I
         DO K=MAX(1,I+J),MIN(N+J,N)
            (statement)
         ENDDO
      ENDDO
   ENDIF
ENDDO
```

b) Removing the `MIN` *function.*

Figure C.2: Removing the `MIN` and `MAX` functions from the `J` loop.

```
DOALL I=1,2*BB-1
   DO J=1-BB,BB-I
      DO K=MAX(1,I+J),MIN(N+J,N)
         (statement)
      ENDDO
   ENDDO
ENDDO
```

Figure C.3: The transformed `J` loop has no `MIN` and `MAX`.

```
DOALL I=1,2*BB-1
   DO J=1-BB,BB-I
      IF (1.GE.I+J) THEN
         DO K=1,MIN(N+J,N)
            (statement)
         ENDDO
      ELSE
         DO K=I+J,MIN(N+J,N)
            (statement)
         ENDDO
      ENDIF
   ENDDO
ENDDO
```

a) Removing the `MAX` *function.*

```
DOALL I=1,2*BB-1
   DO J=1-BB,MIN(BB-I,1-I)
      DO K=1,MIN(N+J,N)
         (statement)
      ENDDO
   ENDDO
   DO J=MAX(2-I,1-BB),BB-I
      DO K=I+J,MIN(N+J,N)
         (statement)
      ENDDO
   ENDDO
ENDDO
```

b) Eliminating the conditional.

Figure C.4: Removing the `MAX` function from the `K` loop.

```
DOALL I=1,2*BB-1
   IF (BB-I.LE.1-I) THEN
      DO J=1-BB,BB-I
         DO K=1,MIN(N+J,N)
            (statement)
         ENDDO
      ENDDO
   ELSE
      DO J=1-BB,1-I
         DO K=1,MIN(N+J,N)
            (statement)
         ENDDO
      ENDDO
   ENDIF
   IF (2-I.GE.1-BB) THEN
      DO J=2-I,BB-I
         DO K=I+J,MIN(N+J,N)
            (statement)
         ENDDO
      ENDDO
   ELSE
      DO J=1-BB,BB-I
         DO K=I+J,MIN(N+J,N)
            (statement)
         ENDDO
      ENDDO
   ENDIF
ENDDO
```

a) Replacing by a conditional.

```
DOALL I=1,MIN(BB,2*BB-1)
   DO J=1-BB,1-I
      DO K=1,MIN(N+J,N)
         (statement)
      ENDDO
   ENDDO
   DO J=2-I,BB-I
      DO K=I+J,MIN(N+J,N)
         (statement)
      ENDDO
   ENDDO
ENDDO
DOALL I=MAX(BB+1,1),2*BB-1
   DO J=1-BB,1-I
      DO K=1,MIN(N+J,N)
         (statement)
      ENDDO
   ENDDO
   DO J=1-BB,BB-I
      DO K=I+J,MIN(N+J,N)
         (statement)
      ENDDO
   ENDDO
ENDDO
```

b) Eliminating the conditional.

Figure C.5: Removing the `MIN` and `MAX` functions from the `J` loop.

```
DOALL I=1,BB
   DO J=1-BB,1-I
      IF (N+J.LE.N)
         DO K=1,N+J
            (statement)
         ENDDO
      ELSE
         DO K=1,N
            (statement)
         ENDDO
      ENDIF
   ENDDO
   DO J=2-I,BB-I
      IF (N+J.LE.N)
         DO K=I+J,N+J
            (statement)
         ENDDO
      ELSE
         DO K=I+J,N
            (statement)
         ENDDO
      ENDIF
   ENDDO
ENDDO
DOALL I=BB+1,2*BB-1
   DO J=1-BB,BB-I
      IF (N+J.LE.N)
         DO K=I+J,N+J
            (statement)
         ENDDO
      ELSE
         DO K=I+J,N
            (statement)
         ENDDO
      ENDIF
   ENDDO
ENDDO
```

a) Replacing by a conditional.

```
DOALL I=1,BB
   DO J=1-BB,MIN(1-I,0)
      DO K=1,N+J
         (statement)
      ENDDO
   ENDDO
   DO J=MAX(1-BB,1),1-I
      DO K=1,N
         (statement)
      ENDDO
   ENDDO
   DO J=2-I,MIN(BB-I,0)
      DO K=I+J,N+J
         (statement)
      ENDDO
   ENDDO
   DO J=MAX(2-I,1),BB-I
      DO K=I+J,N
         (statement)
      ENDDO
   ENDDO
ENDDO
DOALL I=BB+1,2*BB-1
   DO J=1-BB,MIN(BB-I,0)
      DO K=I+J,N+J
         (statement)
      ENDDO
   ENDDO
   DO J=MAX(1-BB,1),BB-I
      DO K=I+J,N
         (statement)
      ENDDO
   ENDDO
ENDDO
```

b) Eliminating the conditional.

Figure C.6: Removing the `MIN` function from the `K` loop.

```
DOALL I=1,BB
   DO J=1-BB,1-I
      DO K=1,N+J
         (statement)
      ENDDO
   ENDDO
   DO J=1,1-I
      DO K=1,N
         (statement)
      ENDDO
   ENDDO
   DO J=2-I,0
      DO K=I+J,N+J
         (statement)
      ENDDO
   ENDDO
   DO J=1,BB-I
      DO K=I+J,N
         (statement)
      ENDDO
   ENDDO
ENDDO
DOALL I=BB+1,2*BB-1
   DO J=1-BB,BB-I
      DO K=I+J,N+J
         (statement)
      ENDDO
   ENDDO
   DO J=1,BB-I
      DO K=I+J,N
         (statement)
      ENDDO
   ENDDO
ENDDO
```

a) Some loops are never executed.

```
DOALL I=1,BB
   DO J=1-BB,1-I
      DO K=1,N+J
         (statement)
      ENDDO
   ENDDO
   DO J=2-I,0
      DO K=I+J,N+J
         (statement)
      ENDDO
   ENDDO
   DO J=1,BB-I
      DO K=I+J,N
         (statement)
      ENDDO
   ENDDO
ENDDO
DOALL I=BB+1,2*BB-1
   DO J=1-BB,BB-I
      DO K=I+J,N+J
         (statement)
      ENDDO
   ENDDO
ENDDO
```

b) The first loop nest is not canonical.

```
DOALL I=1,1
   DO J=1-BB,1-I
      DO K=1,N+J
         (statement)
      ENDDO
   ENDDO
   DO J=1,BB-I
      DO K=I+J,N
         (statement)
      ENDDO
   ENDDO
ENDDO
DOALL I=2,BB-1
   DO J=1-BB,1-I
      DO K=1,N+J
         (statement)
      ENDDO
   ENDDO
   DO J=2-I,0
      DO K=I+J,N+J
         (statement)
      ENDDO
   ENDDO
   DO J=1,BB-I
      DO K=I+J,N
         (statement)
      ENDDO
   ENDDO
ENDDO
DOALL I=BB,BB
   DO J=1-BB,1-I
      DO K=1,N+J
         (statement)
      ENDDO
   ENDDO
   DO J=2-I,0
      DO K=I+J,N+J
         (statement)
      ENDDO
   ENDDO
ENDDO
DOALL I=BB+1,2*BB-1
   DO J=1-BB,BB-I
      DO K=I+J,N+J
         (statement)
      ENDDO
   ENDDO
ENDDO
```

c) All loop nests are canonical.

Figure C.7: Transforming the code into consecutive canonical loop nests.

```
      NN=MIN(N,2*BB-1)
      PP=2*P*P
      F1=INT(1/PP)
      M1=MOD(1,PP)
      F2=INT((BB-2)/PP)
      M2=MOD((BB-2),PP)
      F3=INT(1/PP)
      M3=MOD(1,PP)
      F4=INT((BB-1)/PP)
      M4=MOD((BB-1),PP)
C*KSR* PARALLEL REGION(NUMTHREADS=P,
C*KSR*&PRIVATE=(K2,LK,UK,K,II,K1,I,J))
      K1=IPR_MID()
      DO II=0,P-1
C === first subset ===
         K2=2*P*II+MOD(K1+II,P)
C --- first loop nest ---
         LK=1+K2*F1+MIN(M1,K2)
         UK=1+(K2+1)*F1-1+MIN(M1,K2+1)
         DO I=LK,UK
            DO J=1-BB,0
               DO K=1,N+J
                  (statement)
               ENDDO
            ENDDO
            DO J=1,BB-1
               DO K=1+J,N
                  (statement)
               ENDDO
            ENDDO
         ENDDO
C --- second loop nest ---
         LK=2+K2*F2+MIN(M2,K2)
         UK=2+(K2+1)*F2-1+MIN(M2,K2+1)
         DO I=LK,UK
            DO J=1-BB,1-I
               DO K=1,N+J
                  (statement)
               ENDDO
            ENDDO
            DO J=2-I,0
               DO K=I+J,N+J
                  (statement)
               ENDDO
            ENDDO
            DO J=1,BB-I
               DO K=I+J,N
                  (statement)
               ENDDO
            ENDDO
         ENDDO
C --- third loop nest ---
         LK=BB+K2*F3+MAX(0,K2-PP+M3)
         UK=BB+(K2+1)*F3-1+MAX(0,K2+1-PP+M3)
         DO I=LK,UK
            DO J=1-BB,1-I
               DO K=1,N+J
                  (statement)
               ENDDO
            ENDDO
            DO J=2-I,0
               DO K=I+J,N+J
                  (statement)
               ENDDO
            ENDDO
         ENDDO
C --- fourth loop nest ---
         LK=BB+1+K2*F4+MAX(0,K2-PP+M4)
         UK=BB+1+(K2+1)*F4-1+MAX(0,K2+1-PP+M4)
         DO I=LK,UK
            DO J=1-BB,BB-I
               DO K=I+J,N+J
                  (statement)
               ENDDO
            ENDDO
         ENDDO
C === second subset ===
         K2=2*P*(II+1)-1-MOD(K1+II,P)
C --- first loop nest ---
         LK=1+K2*F1+MIN(M1,K2)
         UK=1+(K2+1)*F1-1+MIN(M1,K2+1)
         DO I=LK,UK
            DO J=1-BB,0
               DO K=1,N+J
                  (statement)
               ENDDO
            ENDDO
            DO J=1,BB-1
               DO K=1+J,N
                  (statement)
               ENDDO
            ENDDO
         ENDDO
C --- second loop nest ---
         LK=2+K2*F2+MIN(M2,K2)
         UK=2+(K2+1)*F2-1+MIN(M2,K2+1)
         DO I=LK,UK
            DO J=1-BB,1-I
               DO K=1,N+J
                  (statement)
               ENDDO
            ENDDO
            DO J=2-I,0
               DO K=I+J,N+J
                  (statement)
               ENDDO
            ENDDO
            DO J=1,BB-I
               DO K=I+J,N
                  (statement)
               ENDDO
            ENDDO
         ENDDO
C --- third loop nest ---
         LK=BB+K2*F3+MAX(0,K2-PP+M3)
         UK=BB+(K2+1)*F3-1+MAX(0,K2+1-PP+M3)
         DO I=LK,UK
            DO J=1-BB,1-I
               DO K=1,N+J
                  (statement)
               ENDDO
            ENDDO
            DO J=2-I,0
               DO K=I+J,N+J
                  (statement)
               ENDDO
            ENDDO
         ENDDO
C --- fourth loop nest ---
         LK=BB+1+K2*F4+MAX(0,K2-PP+M4)
         UK=BB+1+(K2+1)*F4-1+MAX(0,K2+1-PP+M4)
         DO I=LK,UK
            DO J=1-BB,BB-I
               DO K=I+J,N+J
                  (statement)
               ENDDO
            ENDDO
         ENDDO
      ENDDO
C*KSR* END PARALLEL REGION
```

Figure C.8: Partitioning banded SYR2K (version CAN-3t in Figure 5.6).

```
   DO J=1,N
      DO I=1,N
         Z(I,J)=A(I,J)
      ENDDO
      D(J)=A(N,J)
   ENDDO
   IF (N.NE.1) THEN
      DO II=2,N
         H=0.0D0
         SCALE =0.0D0
         IF (II.LE.N-1) THEN
            DO K=1,N+1-II
               SCALE=SCALE+DABS(D(K))
            ENDDO
            IF (SCALE.EQ.0.0D0) THEN
               E(N+2-II)=D(N+1-II)
               DO J=1,N+1-II
                  D(J)=Z(N+1-II,J)
                  Z(N+2-II,J)=0.0D0
                  Z(J,N+2-II)=0.0D0
               ENDDO
            ELSE
               DO K=1,N+1-II
                  D(K)=D(K)/SCALE
                  H=H+D(K)*D(K)
               ENDDO
               G=-DSIGN(DSQRT(H),D(N+1-II))
               E(N+2-II)=SCALE*G
               H=H-D(N+1-II)*G
               D(N+1-II)=D(N+1-II)-G
               DO J=1,N+1-II
                  E(J)=0.0D0
               ENDDO
               L=N+1-II
C --- first parallelised loop nest ---
               DOALL J=1,L
                  Z(J,L+1)=D(J)
                  G=E(J)
                  DO K=1,J
                     G=G+Z(J,K)*D(K)
                  END DO
                  DO K=J+1,L
                     G=G+Z(K,J)*D(K)
                  END DO
                  E(J)=G
               ENDDO
C --- end of first parallelised loop nest ---
               F=0.0D0
               DO J=1,N+1-II
                  E(J)=E(J)/H
                  F=F+E(J)*D(J)
               ENDDO
               HH=F/(H+H)
               DO J=1,N+1-II
                  E(J)=E(J)-HH*D(J)
               ENDDO
C --- second parallelised loop nest ---
               DOALL J=1,L
                  TEMP1=D(J)
                  TEMP2=E(J)
                  DO K=J,L
                     Z(K,J)=Z(K,J)-TEMP1*E(K)
    &                                    -TEMP2*D(K)
                  ENDDO
               ENDDO
C --- end of second parallelised loop nest ---
               DO J=1,L
                  D(J)=Z(L,J)
                  Z(L+1,J)=0.0D0
               ENDDO
            ENDIF
         ELSE
            E(N+2-II)=D(N+1-II)
            DO J=1,N+1-II
               D(J)=Z(N+1-II,J)
               Z(N+2-II,J)=0.0D0
               Z(J,N+2-II)=0.0D0
            ENDDO
         ENDIF
         D(N+2-II)=H
      ENDDO
      DO I=2,N
         Z(N,I-1)=Z(I-1,I-1)
         Z(I-1,I-1)=1.0D0
         IF (D(I).NE.0.0D0) THEN
            DO K=1,I-1
               D(K)=Z(K,I)/D(I)
            ENDDO
            L=I-1
C --- third parallelised loop nest ---
            DOALL J=1,L
               G=0.0D0
               DO K=1,L
                  G=G+Z(K,I)*Z(K,J)
               ENDDO
               DO K=1,L
                  Z(K,J)=Z(K,J)-G*D(K)
               ENDDO
            ENDDO
C --- end of third parallelised loop nest ---
         ENDIF
         DO K=1,I-1
            Z(K,I)=0.0D0
         ENDDO
      ENDDO
   ENDIF
   DO I=1,N
      D(I)=Z(N,I)
      Z(N,I)=0.0D0
   ENDDO
   Z(N,N)=1.0D0
   E(1)=0.0D0
```

Figure C.9: The code for TRED2.

Bibliography

[1] V. Adve, A. Carle, E. Granston, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, J. Mellor-Crummey, S. Warren, "Requirements for Data-Parallel Programming Environments", *IEEE Parallel & Distributed Technology*, **2**-3, Fall 1994, pp. 48–58.

[2] F. Allen, M. Burke, P. Charles, R. Cytron, J. Ferrante, "An Overview of the PTRAN Analysis System for Multiprocessing", *Journal of Parallel and Distributed Computing*, **5**-5, Oct. 1988, pp. 617–640.

[3] F. E. Allen, J. Cocke, "A Catalogue of Optimizing Transformations", in R. Rustin (Ed.), *Design and Optimization of Compilers*, Prentice-Hall, 1972, pp. 1–30.

[4] J. R. Allen, K. Kennedy, C. Porterfield, J. Warren, "Conversion of control dependence to data dependence", *Proceedings of the 10th ACM Symposium on Principles of Programming Languages* (Jan. 1983), ACM Press, pp. 177–189.

[5] R. Allen, K. Kennedy, "Automatic Translation of Fortran Programs to Vector Form", *ACM Transactions on Programming Languages and Systems*, **9**-4, Oct. 1987, pp. 491–542.

[6] G. Amdahl, "Validity of the single-processor approach to achieving large scale computing capabilities", *AFIPS Conference Proceedings*, **30**, 1967, pp. 483–485.

[7] C. Ancourt, F. Irigoin, "Scanning Polyhedra with DO Loops", in *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming* (Williamsburg, Virginia, Apr. 1991), *ACM SIGPLAN Notices*, **26**-7, July 1991, pp. 39–50.

[8] E. Anderson, Z. Bai, C. H. Bischof, J. W. Demmel, J. J. Dongarra, J. J. Du Croz, A. Greenbaum, S. J. Hammarling, A. McKenney, S. Ostrouchov, D. C. Sorensen, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, 1992.

[9] J. M. Anderson, M. S. Lam, "Global Optimizations for Parallelism and Locality on Scalable Parallel Machines", *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation* (Albuquerque, NM, June 1993), *ACM SIGPLAN Notices*, **28**-6, June 1993, pp. 112–125.

[10] D. F. Bacon, S. L. Graham, O. J. Sharp, "Compiler Transformations for High-Performance Computing", *ACM Computing Surveys*, **26**-4, Dec. 1994, pp. 345–420.

[11] M. C. Ballian, "Critical Path Analysis for the Manchester Dataflow Machine", MSc Thesis, Department of Computer Science, University of Manchester, 1989.

[12] U. Banerjee, R. Eigenmann, A. Nicolau, D. Padua, "Automatic Program Parallelization", *Proceedings of the IEEE*, **81**-2, Feb. 1993, pp. 211–243; also available as CSRD Report 1250, Center for Supercomputing Research & Development, University of Illinois at Urbana-Champaign, Nov. 1992.

[13] U. Banerjee, *Dependence Analysis for Supercomputing*, Kluwer Academic Publishers, 1988.

[14] U. Banerjee, "A Theory of Loop Permutations", in D. Gelernter, A. Nicolau, and D. Padua (Eds.), *Languages and Compilers for Parallel Computing*, MIT Press, 1990, pp. 54–74.

[15] U. Banerjee, "Unimodular Transformations of Double Loops", in A. Nicolau, D. Gelernter, T. Gross, and D. Padua (Eds.), *Advances in Languages and Compilers for Parallel Processing*, MIT Press, 1991, pp. 192–219.

[16] U. Banerjee, *Loop Transformations for Restructuring Compilers: The Foundations*, Kluwer Academic Publishers, 1993.

[17] I. Bárány, Z. Füredi, "Computing the Volume is Difficult", *Proceedings of the* 18*th Annual ACM Symposium on Theory of Computing*, 1986, pp. 442–447.

[18] M. Barnett, C. Lengauer, "Unimodularity Considered Non-Essential", in L. Bougé, M. Cosnard, Y. Robert, D. Trystram (Eds.), *Proceedings of CONPAR '92*, LNCS **634**, Springer-Verlag, 1992, pp. 659–664.

[19] M. Barnett, C. Lengauer, "Loop Parallelization and Unimodularity", Technical Report ECS-LFCS-92-197, Department of Computer Science, University of Edinburgh, Jan. 1992.

[20] A. I. Barvinok, "Computing the Volume, Counting Integral Points, and Exponential Sums", *Discrete & Computational Geometry*, **10**-2, 1993, pp. 123–141.

[21] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum, J. Martin, "The PERFECT Club Benchmarks: Effective Performance Evaluation of Supercomputers", *International Journal of Supercomputer Applications*, **3**-3, Fall 1989, pp. 5–40; also available as CSRD Report 827, Center for Supercomputing Research & Development, University of Illinois at Urbana-Champaign, May 1989.

[22] A. J. C. Bik, H. A. G. Wijshoff, "Implementation of Fourier-Motzkin Elimination", Technical Report TR94-42, Department of Computer Science, University of Leiden, The Netherlands, 1994.

[23] R. Bixby, K. Kennedy, U. Kremer, "Automatic Data Layout Using 0-1 Integer Programming", *IFIP Transactions A – Computer Science and Technology*, **50**, 1994, pp. 111–122; an earlier version is available as Technical Report CRPC-TR93349-S, Center for Research on Parallel Computation, Rice University, Nov. 1993.

[24] W. Blume, R. Eigenmann, J. Hoeflinger, D. Padua, P. Petersen, L. Rauchwerger, P. Tu, "Automatic Detection of Parallelism: A Grand Challenge for High-Performance Computing", *IEEE Parallel & Distributed Technology*, **2**-3, Fall 1994, pp. 37–47; also available as CSRD Report 1348, Center for Supercomputing Research & Development, University of Illinois at Urbana-Champaign.

[25] W. Blume, R. Eigenmann, "Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs", *IEEE Transactions on Parallel and Distributed Systems*, **3**-6, Nov. 1992, pp. 643–656; also available as CSRD Report 1218, Center for Supercomputing Research & Development, University of Illinois at Urbana-Champaign, May 1992.

[26] W. Blume, R. Eigenmann, "The Range Test: A Dependence Test for Symbolic, Non-linear Expressions", in *Proceedings of Supercomputing '94* (Washington D. C., Nov. 1994), IEEE Computer Society Press, pp. 528–537; also available as CSRD Report 1345, Center for Supercomputing Research & Development, University of Illinois at Urbana-Champaign.

[27] W. Blume, R. Eigenmann, "An overview of Symbolic Analysis Techniques Needed for the Effective Parallelization of the Perfect Benchmarks", in *Proceedings of the 1994 International Conference in Parallel Processing* (St. Charles, Illinois, 1994).

[28] W. J. Blume, *Symbolic Analysis Techniques for Effective Automatic Parallelization*, PhD Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1995.

[29] F. Bodin, M. O'Boyle, "A Compiler Strategy for Shared Virtual Memories", in B. K. Szymanski, B. Sinharoy (Eds.), *Languages, Compilers and Run-Time Systems for Scalable Computers*, Kluwer Academic Publishers, 1996, pp. 57–69.

[30] P. Borwein, C. Ingalls, "The Prouhet-Tarry-Escott Problem Revisited", *L' Enseignement Mathématique*, **40**, 1994, pp. 3–27.

[31] R. T. Boute, "The Euclidean Definition of the Functions div and mod", *ACM Transactions on Programming Languages and Systems*, **14**-2, Apr. 1992, pp. 127–144.

[32] H. Burkhart, R. Millen, "Performance-Measurement Tools in a Multiprocessor Environment", *IEEE Transactions on Computers*, **38**-5, May 1989, pp. 725–737.

[33] D. Callahan, K. Kennedy, "Compiling programs for distributed-memory multiprocessors", *Journal of Supercomputing*, **2**-2, Oct. 1988, pp. 151–169.

[34] D. Cann, "Retire Fortran? A Debate Rekindled", *Communications of the ACM*, **35**-8, Aug. 1992, pp. 81–89.

[35] E. A. Carmona, M. D. Rice, "Modeling the Serial and Parallel Fractions of a Parallel Algorithm", *Journal of Parallel and Distributed Computing*, **13**-3, Nov. 1991, pp. 286–298.

[36] S. Carr, K. S. McKinley, C.-W. Tseng, "Compiler Optimizations for Improving Data Locality", in *Proceedings of the* 6*th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, Oct. 1994), *ACM SIGPLAN Notices*, **29**-11, Nov. 1994, pp. 252–262; also available as Technical Report TR94-234, Department of Computer Science, Rice University, July 1994.

[37] S. Carr, *Memory-Hierarchy Management*, PhD Thesis, Department of Computer Science, Rice University, Sep. 1992; also available as Technical Report TR92-190, Oct. 1992.

[38] Z. S. Chamski, G. A. Hedayat, "Interactive Visualization of High-Dimension Iteration and Data Sets", in W. K. Giloi, S. Jähnichen, B. D. Shriver (Eds.), *Programming Models for Massively Parallel Computers*, IEEE Computer Society Press, 1995, pp. 189–196.

[39] Z. S. Chamski, M. F. P. O'Boyle, "Practical Loop Generation", in *Proceedings of the* 29*th Annual Hawaii International Conference on System Sciences* (Vol. I, Software Technology and Architecture), IEEE Computer Society Press, 1996, pp. 223–232.

[40] B. Chapman, P. Mehrotra, H. Zima, "Programming in Vienna Fortran", *Scientific Programming*, **1**-1, Fall 1992, pp. 31–50.

[41] V. Chaudhary, J. K. Aggarwal, "A Generalized Scheme for Mapping Parallel Algorithms", *IEEE Transactions on Parallel and Distributed Systems*, **4**-3, Mar. 1993, pp. 328–346.

[42] V. Chvátal, *Linear Programming*, W. H. Freeman and Company, New York, 1983.

[43] P. Clauss, "Counting Solutions to Linear and Nonlinear Constraints through Ehrhart polynomials: Applications to Analyze and Transform Scientific Programs", in *Proceedings of the 1996 International Conference on Supercomputing* (Philadelphia, May 1996), ACM Press, pp. 278–285.

[44] J. Cohen, T. Hickey, "Two Algorithms for Determining Volumes of Convex Polyhedra", *Journal of the ACM*, **26**-3, July 1979, pp. 401–414.

[45] J.-F. Collard, "Automatic Parallelization of WHILE Loops using Speculative Execution", *International Journal of Parallel Programming*, **23**-2, 1995, pp. 191–219; an earlier version is available as Research Report 93-38, Laboratoire de l' Informatique du Parallélisme, École Normale Supérieure de Lyon, France, Nov. 1993.

[46] K. D. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, S. K. Warren, "The ParaScope Parallel Programming Environment", *Proceedings of the IEEE*, **81**-2, Feb. 1993, pp. 244–263.

[47] M. Cosnard, M. Loi, "Automatic Task Graph Generation Techniques", in *Proceedings of the 28th Annual Hawaii International Conference on System Sciences* (Vol. II, Software Technology), IEEE Computer Society Press, 1995, pp. 113–122.

[48] H. S. M. Coxeter, *Regular Polytopes*, Dover Publications, 1973.

[49] M. E. Crovella, *Performance Prediction and Tuning of Parallel Programs*, PhD Thesis, Department of Computer Science, University of Rochester, 1994.

[50] M. E. Crovella, T. J. LeBlanc, "Parallel Performance Prediction Using Lost Cycles Analysis", in *Proceedings of Supercomputing '94* (Washington D. C., Nov. 1994), IEEE Computer Society Press, pp. 600–609; a more detailed version is available as Technical Report 479, Department of Computer Science, University of Rochester, Dec. 1993.

[51] L. A. Crowl, "How to Measure, Present, and Compare Parallel Performance", *IEEE Parallel & Distributed Technology*, Spring 1994, pp. 9–25.

[52] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, T. Von Eicken, "LogP: Towards a Realistic Model of Parallel Computation", in *Proceedings of the 4th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming* (San Diego, May 1993), *ACM SIGPLAN Notices*, **28**-7, July 1993, pp. 1–12.

[53] R. Cytron, D. J. Kuck, A. V. Veidenbaum, "The effect of restructuring compilers on program performance for high-speed computers", *Computer Physics Communications*, **37**, July 1985, pp. 39–48.

[54] R. Cytron, J. Ferrante, V. Sarkar, "Experiences Using Control Dependence in PTRAN", in D. Gelernter, A. Nicolau, D. Padua (Eds.), *Languages and Compilers for Parallel Computing*, Research Monographs in Parallel and Distributed Computing Series, MIT Press, 1990, pp. 186–212.

[55] J. J. Dongarra, A. R. Hinds, "Unrolling Loops in FORTRAN", *Software–Practice and Experience*, **9**-3, Mar. 1979, pp. 219–226.

[56] J. J. Dongarra, J. Du Croz, S. Hammarling, I. Duff, "A Set of Level 3 Basic Linear Algebra Subprograms", *ACM Transactions on Mathematical Software*, **16**-1, Mar. 1990, pp. 1–17.

[57] M. A. Driscoll, W. R. Daasch, "Accurate Predictions of Parallel Program Execution Time", *Journal of Parallel and Distributed Computing*, **25**-1, Feb. 1995, pp. 16–30.

[58] M. E. Dyer, A. M. Frieze, "On the Complexity of Computing the Volume of a Polyhedron", *SIAM Journal on Computing*, **17**-5, Oct. 1988, pp. 967–974.

[59] M. Dyer, A. Frieze, R. Kannan, "A Random Polynomial-Time Algorithm for Approximating the Volume of Convex Bodies", *Journal of the ACM*, **38**-1, Jan. 1991, pp. 1–17.

[60] R. Eigenmann, J. Hoeflinger, G. Jaxon, D. Padua, "The Cedar Fortran Project", CSRD Report 1262, Center for Supercomputing Research & Development, University of Illinois at Urbana-Champaign, 1992.

[61] R. Eigenmann, J. Hoeflinger, Z. Li, D. Padua, "Experience in the Automatic Parallelization of Four Perfect-Benchmark Programs", in U. Banerjee, D. Gelernter, A. Nicolau, D. Padua (Eds.), *Languages and Compilers for Parallel Computing* (4th International Workshop, Aug. 1991), LNCS **589**, Springer-Verlag, 1992, pp. 65–83.

[62] C. Eisenbeis, J.-C. Sogno, "A general algorithm for data dependence analysis", in *Proceedings of the 1992 International Conference on Supercomputing* (Washington D. C. July 1992), ACM Press, pp. 292–302; also available as Rapport de Recherche RR-1699, INRIA, May 1992.

[63] V. Faber, O. M. Lubeck, A. B. White, "Comments on the paper 'Parallel efficiency can be greater than unity'", *Parallel Computing*, **4**-2, Apr. 1987, pp. 209–210.

[64] T. Fahringer, "Estimating and Optimizing Performance for Parallel Programs", *IEEE Computer*, **28**-11, Nov. 1995, pp. 47–56.

[65] Z. Fang, P. Tang, P.-C. Yew, C.-Q. Zhu, "Dynamic Processor Self-Scheduling for General Parallel Nested Loops", *IEEE Transactions on Computers*, **39**-7, Jul. 1990, pp. 919–929.

[66] P. Feautrier, "Array Expansion", in *Proceedings of the 1988 International Conference on Supercomputing* (St. Malo, July 1988), pp. 429–441.

[67] P. Feautrier, "Data-Flow Analysis of Array and Scalar References", *International Journal of Parallel Programming*, **20**-1, 1991, pp. 23–53.

[68] P. Feautrier, "Automatic Parallelization Techniques", tutorial given in *CONPAR '92* (Lyon, France, Sep. 1992).

[69] J. T. Feo, D. C. Cann, R. R. Oldehoeft, "A Report on the Sisal Language Project", *Journal of Parallel and Distributed Computing*, **10**-4, Dec. 1990, pp. 349–366.

[70] A. Fernández, J. M. Llabería, M. Valero-García, "Loop Transformation Using Nonunimodular Matrices", *IEEE Transactions on Parallel and Distributed Systems*, **6**-8, Aug. 1995, pp. 832–840.

[71] J. Ferrante, K. J. Ottenstein, J. D. Warren, "The program dependency graph and its uses in optimization", *ACM Transactions on Programming Languages and Systems*, **9**-3, June 1987, pp. 319–349.

[72] J. Ferrante, V. Sarkar, W. Thrash, "On Estimating and Enhancing Cache Effectiveness", in U. Banerjee, D. Gelernter, A. Nicolau, D. Padua (Eds.), *Languages and Compilers for Parallel Computing* (4th International Workshop, Aug. 1991), LNCS **589**, Springer-Verlag, 1992, pp. 328–343.

[73] D. Fischer, "On Superlinear Speedups", *Parallel Computing*, **17**-6&7, Sep. 1991, pp. 695–697.

[74] M. J. Fischer, M. O. Rabin, "Super-exponential Complexity of Presburger Arithmetic", *SIAM-AMS Proceedings of the Symposium on Real Computational Processes*, **7**, 1974, pp. 27–41.

[75] M. J. Flynn, "Some Computer Organizations and Their Effectiveness", *IEEE Transactions on Computers*, **21**-9, Sep. 1972, pp. 948–960.

[76] S. Flynn Hummel, E. Schonberg, L. E. Flynn, "Factoring: A Method for Scheduling Parallel Loops", *Communications of the ACM*, **35**-8, Aug. 1992, pp. 90–101.

[77] G. C. Fox, R. D. Williams, P. C. Messina, *Parallel Computing Works!*, Morgan Kaufmann, 1994; also available as
`http://www.npac.syr.edu/copywrite/pcw/`

[78] K. A. Gallivan, R. J. Plemmons, A. H. Sameh, "Parallel Algorithms for Dense Linear Algebra Computations", *SIAM Review*, **32**-1, Mar. 1990, pp. 54–135.

[79] B. S. Garbow, J. M. Boyle, J. J. Dongarra, C. B. Moler, *Matrix Eigensystem Routines – EISPACK Guide Extension*, LNCS **51**, Springer-Verlag, 1977.

[80] H. Gauchman, I. Rosenholtz, "The Tarry-Escott Problem", *American Mathematical Monthly*, **102**-9, Nov. 1995, pp. 843-844.

[81] M. Girkar, C. D. Polychronopoulos, "Automatic Extraction of Functional Parallelism from Ordinary Programs", *IEEE Transactions on Parallel and Distributed Systems*, **3**-2, Mar. 1992, pp. 166–178.

[82] M. Girkar, C. D. Polychronopoulos, "The Hierarchical Task Graph as a Universal Intermediate Representation", *International Journal of Parallel Programming*, **22**-5, 1994, pp. 519–551.

[83] G. Goff, K. Kennedy, C.-W. Tseng, "Practical Dependence Testing", in *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation* (Toronto, June 1991), *ACM SIGPLAN Notices*, **26**-6, June 1991, pp. 15–29.

[84] G. H. Golub, C. F. Van Loan, *Matrix Computations*, The Johns Hopkins University Press, 1989.

[85] R. L. Graham, D. E. Knuth, O. Patashnik, *Concrete Mathematics*, Addison-Wesley, 1991.

[86] A. Y. Grama, A. Gupta, V. Kumar, "Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures", *IEEE Parallel & Distributed Technology*, **1**-3, Aug. 1993, pp. 12–21.

[87] E. Granston, T. Montaut, F. Bodin, "Loop Transformations to Prevent False Sharing", *International Journal of Parallel Programming*, **23**-4, 1995, pp. 263–301; also available as Technical Report CRPC-TR95528, Center for Research on Parallel Computation, Rice University, May 1995.

[88] E. D. Granston, H. A. G. Wijshoff, "Managing Pages in Shared Virtual Memory Systems: Getting the Compiler into the Game", in *Proceedings of the 1993 International Conference on Supercomputing* (Tokyo, July 1993), ACM Press, pp. 11–20.

[89] T. Gross, D. R. O'Hallaron, J. Subhlok, "Task Parallelism in a High Performance Fortran Framework", *IEEE Parallel & Distributed Technology*, **2**-3, Fall 1994, pp. 16–26.

[90] M. Gupta, *Automatic Data Partitioning on Distributed Memory Computers*, PhD Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Sep. 1992.

[91] M. Gupta, P. Banerjee, "Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers", *IEEE Transactions on Parallel and Distributed Systems*, **3**-2, Mar. 1992, pp. 179–193.

[92] M. Haghighat, C. Polychronopoulos, "Symbolic Dependence Analysis for High-Performance Parallelizing Compilers", in A. Nicolau, D. Gelernter, T. Gross, D. Padua (Eds.), *Advances in Languages and Compilers for Parallel Processing*, MIT Press, 1991, pp. 310–330.

[93] M. R. Haghighat, C. D. Polychronopoulos, "Symbolic Analysis: A Basis for Parallelization, Optimization, and Scheduling of Programs", in U. Banerjee, D. Gelernter, A. Nicolau, D. Padua (Eds.), *Languages and Compilers for Parallel Computing* (6th International Workshop, Aug. 1993), LNCS **768**, Springer-Verlag, 1994, pp. 567–585; also available as CSRD Report 1314, Center for Supercomputing Research & Development, University of Illinois at Urbana-Champaign.

[94] M. R. Haghighat, C. D. Polychronopoulos, "Symbolic Analysis for Parallelizing Compilers", *ACM Transactions on Programming Languages and Systems*, **18**-4, July 1996, pp. 477–518.

[95] M. W. Hall, B. R. Murphy, S. P. Amarasinghe, S.-W. Liao, M. S. Lam, "Interprocedural Analysis for Parallelization", in C.-H. Huang, P. Sadayappan, U. Banerjee, D. Gelernter, A. Nicolau, D. Padua (Eds.), *Languages and Compilers for Parallel Computing* (8th International Workshop, Aug. 1995), LNCS **1033**, Springer-Verlag, 1996, pp. 61–80.

[96] G. H. Hardy, E. M. Wright, *An Introduction to the Theory of Numbers*, Clarendon Press, Oxford, 5th edition, 1979.

[97] T. J. Harris, "A Survey of PRAM Simulation Techniques", *ACM Computing Surveys*, **26**-2, June 1994, pp. 187–206.

[98] P. J. Hatcher, M. J. Quinn, A. J. Lapadula, B. K. Seevers, R. J. Anderson, R. R. Jones, "Data-Parallel Programming on MIMD Computers", *IEEE Transactions on Parallel and Distributed Systems*, **2**-3, July 1991, pp. 377–383.

[99] P. Havlak, *Interprocedural Symbolic Analysis*, PhD Thesis, Department of Computer Science, Rice University, May 1994; also available as Technical Report CRPC-TR94451-S, Center for Research on Parallel Computation, Rice University, May 1994.

[100] T. L. Heath, *Diophantus of Alexandria*, Cambridge University Press, 1910.

[101] T. L. Heath, *The Thirteen Books of Euclid's Elements*, vol. II (Books III–IX), Cambridge University Press, 1908.

[102] T. Hickey, J. Cohen, "Automating Program Analysis", *Journal of the ACM*, **35**-1, Jan. 1988, pp. 185–220.

[103] High Performance Fortran Forum, "High Performance Fortran Language Specification", *Scientific Programming*, **2**-1&2, Spring and Summer 1993, pp. 1–170.

[104] W. D. Hillis, L. W. Tucker, "The CM-5 Connection Machine: A Scalable Supercomputer", *Communications of the ACM*, **36**-11, Nov. 1993, pp. 31–40.

[105] S. Hiranandani, K. Kennedy, C.-W. Tseng, "Compiling Fortran D for MIMD Distributed-Memory Machines", *Communications of the ACM*, **35**-8, Aug. 1992, pp. 66–80.

[106] S. Hiranandani, K. Kennedy, C.-W. Tseng, "Evaluating Compiler Optimizations for Fortran D", *Journal of Parallel and Distributed Computing*, **21**-1, Apr. 1994, pp. 27–45.

[107] R. W. Hockney, C. R. Jesshope, *Parallel Computers 2: Architecture, Programming and Algorithms*, IOP Publishing Ltd, 1992.

[108] C.-H. Huang, P. Sadayappan, "Communication-Free Hyperplane Partitioning of Nested Loops", *Journal of Parallel and Distributed Computing*, **19**-2, Oct. 1993, pp. 90–102; an earlier, shorter version is available in U. Banerjee, D. Gelernter, A. Nicolau, D. Padua (Eds.), *Languages and Compilers for Parallel Computing* (4th International Workshop, Aug. 1991), LNCS **589**, Springer-Verlag, 1992, pp. 186–200.

[109] J. Hummel, L. J. Hendren, A. Nicolau, "A General Data Dependence Test for Dynamic, Pointer-Based Data Structures", in *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation* (Orlando, June 1994), *ACM SIGPLAN Notices*, **29**-6, June 1994, pp. 218–229.

[110] W. Hussak, *Decidability in Temporal Presburger Arithmetic*, MSc Thesis, Department of Computer Science, University of Manchester, Feb. 1987; also available as Technical Report UMCS-87-11-4.

[111] INMOS Ltd, *Occam 2 Reference Manual*, Prentice Hall International, 1988.

[112] International Organization for Standardization (ISO), *Fortran 90*, ISO/IEC 1539, July 1991.

[113] F. Irigoin, R. Triolet, "Supernode Partitioning", in *Proceedings of the* 15*th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (San Diego, CA, Jan. 1988), ACM Press, pp. 319–329.

[114] K. E. Iverson, *A Programming Language*, Wiley, 1962.

[115] A. H. Karp, R. G. Babb, "A Comparison of 12 Parallel Fortran Dialects", *IEEE Software*, **5**-5, Sep. 1988, pp. 52–67.

[116] A. H. Karp, H. P. Flatt, "Measuring Parallel Processor Performance", *Communications of the ACM*, **33**-5, May 1990, pp. 539–543.

[117] R. M. Karp, V. Ramachandran, "Parallel Algorithms for Shared-Memory Machines", in J. Van Leeuwen (Ed.), *Handbook of Theoretical Computer Science (Volume A: Algorithms and Complexity)*, Elsevier, 1990, pp. 871–941.

[118] W. Kelly, W. Pugh, "Using Affine Closure to Find Legal Reordering Transformations", *International Journal of Parallel Programming*, **23**-4, 1995, pp. 303–325.

[119] W. Kelly, W. Pugh, "Finding Legal Reordering Transformations using Mappings", in K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, D. Padua (Eds.), *Languages and Compilers for Parallel Computing* (7th International Workshop, Aug. 1994), LNCS **892**, Springer-Verlag, 1995, pp. 107–124; also available as Technical Report CS-TR-3297, Department of Computer Science, University of Maryland, June 1994.

[120] W. Kelly, W. Pugh, "A Framework for Unifying Reordering Transformations", Technical Report CS-TR-3193, Department of Computer Science, University of Maryland, Apr. 1993; this is a revised version of Technical Report CS-TR-2995, Nov. 1992.

[121] W. Kelly, W. Pugh, "Determining Schedules based on Performance Estimation", *Parallel Processing Letters*, **4**-3, Sep. 1994, pp. 205–219; also available as Technical Report CS-TR-3108, Department of Computer Science, University of Maryland, Dec. 1993.

[122] Kendall Square Research, *KSR Fortran Programming*, 170 Tracer Lane, Waltham, MA 02154-1379, 1991.

[123] Kendall Square Research, *KSR Parallel Programming*, 170 Tracer Lane, Waltham, MA 02154-1379, 1993.

[124] K. Kennedy, K. S. McKinley, "Loop Distribution with Arbitrary Control Flow", in *Proceedings of Supercomputing 90* (New York, Nov. 1990), IEEE Computer Society Press, pp. 407–416.

[125] K. Kennedy, K. McKinley, "Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution", in U. Banerjee, D. Gelernter, A. Nicolau, D. Padua (Eds.), *Languages and Compilers for Parallel Computing* (6th International Workshop, Aug. 1993), LNCS **768**, Springer-Verlag, 1994, pp. 301–320; also available as Technical Report CRPC-TR94370, Center for Research on Parallel Computation, Rice University, Jan. 1994.

[126] S. Kertzner, "The Linear Diophantine Equation", *American Mathematical Monthly*, **88**-3, Mar. 1981, pp. 200–203.

[127] D. E. Knuth, "An Empirical Study of FORTRAN programs", *Software–Practice and Experience*, **1**, 1971, pp. 105–133.

[128] D. E. Knuth, *The Art of Computer Programming: Fundamental Algorithms*, Addison-Wesley, 1973.

[129] D. E. Knuth, "Big Omicron and Big Omega and Big Theta", *ACM SIGACT News*, **8**-2, Apr. 1976, pp. 18–24.

[130] D. E. Knuth, "Theory and Practice", *Theoretical Computer Science*, **90**, 1991, pp. 1–15.

[131] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr., M. E. Zosel, *The High Performance Fortran Handbook*, MIT Press, 1994.

[132] X. Kong, D. Klappholz, K. Psarris, "The I Test: An Improved Dependence Test for Automatic Parallelization and Vectorization", *IEEE Transactions on Parallel and Distributed Systems*, **2**-3, July 1991, pp. 342–349.

[133] U. Kremer (Ed.), *Proceedings of the Workshop on Automatic Data Layout and Performance Prediction*, Technical Report CRPC-TR95548, Center for Research on Parallel Computation, Rice University, Apr. 1995.

[134] U. Kremer, "NP-completeness of Dynamic Remapping", Technical Report CRPC-TR93330-S, Center for Research on Parallel Computation, Rice University, Aug. 1993.

[135] C. Kruskal, A. Weiss, "Allocating Independent Subtasks on Parallel Processors", *IEEE Transactions on Software Engineering*, **11**-10, Oct. 1985, pp. 1001–1016.

[136] D. J. Kuck, "A Survey of Parallel Machine Organization and Programming", *ACM Computing Surveys*, **9**-1, Mar. 1977, pp. 29–59.

[137] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, M. Wolfe, "Dependence Graphs and Compiler Optimizations", in *Proceedings of the 8th ACM Symposium on the Principles of Programming Languages*, Jan. 1981, pp. 207–218.

[138] L. Lamport, "The Parallel Execution of DO loops", *Communications of the ACM*, **17**-2, Feb. 74, pp. 83–93.

[139] J. Lawrence, "Polytope Volume Computation", *Mathematics of Computation*, **57**-195, July 1991, pp. 259–271.

[140] B. Leasure (Ed.), *PCF Fortran: Language Definition*, The Parallel Computing Forum, Aug. 1990.

[141] P.-Z. Lee, Z. M. Kedem, "Mapping Nested Loop Algorithms into Multidimensional Systolic Arrays", *IEEE Transactions on Parallel and Distributed Systems*, **1**-1, Jan. 1990, pp. 64–76; a shorter version is available in F. Ris, P. M. Kogge (Eds.), *Proceedings of the 1989 International Conference on Parallel Processing* (Vol. III Algorithms and Applications), The Pennsylvania State University Press, pp. 206–210.

[142] T. Lehr, Z. Segall, D. F. Vrsalovic, E. Caplan, A. L. Chung, C. E. Fineman, "Visualizing Performance Debugging", *IEEE Computer*, **22**-10, Oct. 1989, pp. 38–51.

[143] D. Levine, D. Callahan, J. Dongarra, "A comparative study of automatic vectorizing compilers", *Parallel Computing*, **17**-10&11, Dec. 1991, pp. 1223–1244.

[144] J. Li, M. Chen, "Index Domain Alignment: Minimizing Cost of Cross-Referencing Between Distributed Arrays", in *Frontiers '90: The Third Symposium on the Frontiers of Massively Parallel Computation* (University of Maryland, Oct. 1990), IEEE Computer Society Press, pp. 424–433.

[145] W. Li, K. Pingali, "A Singular Loop Transformation Framework Based on Non-Singular Matrices", in U. Banerjee, D. Gelernter, A. Nicolau, D. Padua (Eds.), *Languages and Compilers for Parallel Computing* (5th International Workshop, Aug. 1992), LNCS **757**, Springer-Verlag, 1993, pp. 391–405.

[146] W. Li, "Compiler Optimizations for Cache Locality and Coherence", Technical Report 504, Department of Computer Science, University of Rochester, Apr. 1994.

[147] Z. Li, P.-C. Yew, C.-Q. Zhu, "An Efficient Data Dependence Analysis for Parallelizing Compilers", *IEEE Transactions on Parallel and Distributed Systems*, **1**-1, Jan. 1990, pp. 26–34.

[148] Z. Li, "Array Privatization for Parallel Execution of Loops", in *Proceedings of the 1992 International Conference on Supercomputing* (Washington D. C., July 1992), ACM Press, pp. 313–322.

[149] D. J. Lilja, "Exploiting the Parallelism Available in Loops", *IEEE Computer*, **27**-2, Feb. 1994, pp. 13–26.

[150] D. B. Loveman, "Program Improvement by Source-to-Source Transformation", *Journal of the ACM*, **24**-1, Jan. 1977, pp. 121–145.

[151] N. Manjikian, T. S. Abdelrahman, "Fusion of Loops for Parallelism and Locality", Technical Report CSRI-315, Computer Systems Research Institute, University of Toronto, Feb. 1995; a shorter version is also available in *Proceedings of the International Conference on Parallel Processing* (Aug. 1995).

[152] E. P. Markatos, T. J. LeBlanc, "Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors", *IEEE Transactions on Parallel and Distributed Systems*, **5**-4, Apr. 1994, pp. 379–400; an earlier, shorter version is also available in *Proceedings of Supercomputing '92* (Minneapolis, Nov. 1992), IEEE Computer Society Press, pp. 104–113.

[153] E. Markatos, *Scheduling for Locality in Shared-Memory Multiprocessors*, PhD Thesis, Department of Computer Science, University of Rochester, 1993.

[154] V. Maslov, "Delinearization: an Efficient Way to Break Multiloop Dependence Equations", in *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation* (San Francisco, June 1992), *ACM SIGPLAN Notices*, **27**-7, July 1992, pp. 152–161.

[155] D. E. Maydan, J. L. Hennessy, M. S. Lam, "Efficient and Exact Data Dependence Analysis", in *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation* (Toronto, June 1991), *ACM SIGPLAN Notices*, **26**-6, June 1991, pp. 1–14.

[156] H. G. Mayer, M. Wolfe, "InterProcedural Alias Analysis: Implementation and Empirical Results", *Software–Practice and Experience*, **23**-11, Nov. 1993, pp. 1201–1233.

[157] K. S. McKinley, *Automatic and Interactive Parallelization*, PhD Thesis, Rice University, Texas, Apr. 1992; also available as TR92-182, Department of Computer Science, Rice University.

[158] K. S. McKinley, S. Carr, C.-W. Tseng, "Improving Data Locality with Loop Transformations", *ACM Transactions on Programming Languages and Systems*, **18**-4, July 1996, pp. 424–453.

[159] J. J. Modi, *Parallel Algorithms and Matrix Computation*, Clarendon Press, Oxford, 1988.

[160] M. Y. Mohd-Saman, D. J. Evans, "Inter-procedural analysis for parallel computing", *Parallel Computing*, **21**-2, Feb. 1995, pp. 315–338.

[161] D. I. Moldovan, J. A. B. Fortes, "Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays", *IEEE Transactions on Computers*, **35**-1, Jan. 1986, pp. 1–12.

[162] D. I. Moldovan, *Parallel Processing: From Applications to Systems*, Morgan Kaufmann, 1993.

[163] L. J. Mordell, *Diophantine Equations*, Academic Press, 1969.

[164] J. S. Morgan, J. L. Schonfelder, *Programming in Fortran 90*, Alfred Waller Ltd, 1993.

[165] N.A. Software Ltd., *Fortran 90 Plus. User's Guide*, Liverpool, U.K., 1994.

[166] M. G. Norman, P. Thanisch, "Models of Machines and Computation for Mapping in Multicomputers", *ACM Computing Surveys*, **25**-3, Sep. 1993, pp. 263–302.

[167] M. F. P. O'Boyle, J. M. Bull, "Expert Programmer versus Parallelizing Compiler: A Comparative Study of Two Approaches for Distributed Shared Memory", *Scientific Programming*, **5**-1, Spring 1996, pp. 63–88.

[168] M. O'Boyle, G. A. Hedayat, "Load Balancing of Parallel Affine Loops by Unimodular Transformations", in W. Joosen and E. Milgrom (Eds.), *Parallel Computing: From Theory to Sound Practice* (Proceedings of the European Workshop on Parallel Computing, EWPC '92), IOS Press, 1992, pp. 192–203; also available as Technical Report UMCS-92-1-1, Department of Computer Science, University of Manchester, 1992.

[169] M. O'Boyle, "A Data Partitioning Algorithm for Distributed Memory Compilation", in C. Halatsis, D. Maritsas, G. Philokyprou, S. Theodoridis (Eds.), *PARLE '94 Proceedings*, LNCS **817**, Springer-Verlag, 1994, pp. 61–72; a more detailed version is available as Technical Report UMCS-93-7-1, Department of Computer Science, University of Manchester, 1993.

[170] M. F. P. O'Boyle, *Program and Data Transformations for Efficient Execution on Distributed Memory Architectures*, PhD Thesis, Department of Computer Science, University of Manchester, Jan. 1992; also available as Technical Report UMCS-93-1-6, 1993.

[171] D. A. Padua, M. J. Wolfe, "Advanced Compiler Optimizations for Supercomputers", *Communications of the ACM*, **29**-12, Dec. 1986, pp. 1184–1201.

[172] C. M. Pancake, D. Bergmark, "Do Parallel Languages Respond to the Needs of Scientific Programmers?", *IEEE Computer*, **23**-12, Dec. 1990, pp. 13–23.

[173] C. M. Pancake, "Software Support for Parallel Computing: Where Are We Headed?", *Communications of the ACM*, **34**-11, Nov. 1991, pp. 52–64.

[174] P. Papadimitriou, *Parallel Solution of SVD-Related Problems, with Applications*, PhD Thesis, Department of Mathematics, University of Manchester, 1993.

[175] N. Paris, "POMPC: A C-Language for Data Parallelism", *International Journal of Modern Physics C – Physics and Computers*, **4**-1, 1993, pp. 85–96.

[176] P. M. Petersen, D. A. Padua, "Machine-Independent evaluation of Parallelizing Compilers", CSRD Report 1173, Center for Supercomputing Research & Development, University of Illinois at Urbana-Champaign, Jan. 1992.

[177] K. L. Pieper, *Parallelizing Compilers: Implementation and Effectiveness*, PhD Thesis, Department of Computer Science, Stanford University, 1993.

[178] J. Plevyak, A. A. Chien, V. Karamcheti, "Analysis of Dynamic Structures for Efficient Parallel Execution", in U. Banerjee, D. Gelernter, A. Nicolau, D. Padua (Eds.), *Languages and Compilers for Parallel Computing* (6th International Workshop, Aug. 1993), LNCS **768**, Springer-Verlag, 1994, pp. 37–56.

[179] C. D. Polychronopoulos, D. J. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers", *IEEE Transactions on Computers*, **36**-12, Dec. 1987, pp. 1425–1439.

[180] C. D. Polychronopoulos, *Parallel Programming and Compilers*, Kluwer Academic Publishers, 1988.

[181] C. D. Polychronopoulos, M. B. Girkar, M. R. Haghighat, C. L. Lee, B. P. Leung, D. A. Schouten, "Parafrase-2: An environment for parallelizing, partitioning, synchronizing and scheduling programs on multiprocessors", in *Proceedings of the 1989 International Conference on Parallel Processing* (Vol. II Software), The Pennsylvania State University Press, pp. 39–48.

[182] D. L. Presberg, N. W. Johnson, "The Paralyzer: IVTRAN's Parallelism Analyzer and Synthesizer", *ACM SIGPLAN Notices*, **10**-3, Mar. 1975, pp. 9–16.

[183] M. E. Prouhet, "Mémoire sur quelques relations entre les puissances des nombres", *Comptes Rendus Hebdomadaires de l' Académie des Sciences*, **33**, 1851, p. 225.

[184] W. Pugh, "A Practical Algorithm for Exact Array Dependence Analysis", *Communications of the ACM*, **35**-8, Aug. 1992, pp. 102–114.

[185] W. Pugh, "Counting Solutions to Presburger Formulas: How and Why", in *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation* (Orlando, June 1994), *ACM SIGPLAN Notices*, **29**-6, June 1994, pp. 121–134; also available as Technical Report CS-TR-3234, Department of Computer Science, University of Maryland, Mar. 1994.

[186] W. Pugh, D. Wonnacott, "Static Analysis of Upper and Lower Bounds on Dependences and Parallelism", *ACM Transactions on Programming Languages and Systems*, **16**-4, July 1994, pp. 1248–1278; also available as Technical Report CS-TR-3250, Department of Computer Science, University of Maryland, March 1994.

[187] W. Pugh, D. Wonnacott, "Going Beyond Integer Programming with the Omega Test to Eliminate False Dependences", *IEEE Transactions on Parallel and Distributed Systems*, **6**-2, Feb. 1995, pp. 204–211.

[188] J. Ramanujam, P. Sadayappan, "Tiling Multidimensional Iteration Spaces for Multicomputers", *Journal of Parallel and Distributed Computing*, **16**-2, Oct. 1992, pp. 108–120.

[189] J. Ramanujam, "Non-unimodular Transformations of Nested Loops", in *Proceedings of Supercomputing '92* (Minneapolis, Nov. 1992), IEEE Computer Society Press, pp. 214–223.

[190] L. Rauchwerger, D. Padua, "Parallelizing WHILE Loops for Multiprocessor Systems", CSRD Report 1349, Center for Supercomputing Research & Development, University of Illinois at Urbana-Champaign, 1993.

[191] G. D. Riley, *Techniques for Improving the Performance of Parallel Computations*, MSc Thesis, Department of Computer Science, University of Manchester, in preparation.

[192] Y. Robert, S. W. Song, "Revisiting cycle shrinking", *Parallel Computing*, **18**-5, May 1992, pp. 481–496; also available as "New Techniques for Cycle Shrinking", in D. Etiemble, J.-C. Syre (Eds.), *PARLE '92. Parallel Architectures and Languages Europe*, LNCS **605**, Springer-Verlag, 1992, pp. 449–464.

[193] R. Sakellariou, "Exploiting Parallelism in Integer Factorisation Algorithms", MSc Thesis, Department of Computer Science, University of Manchester, 1992.

[194] R. Sakellariou, "A Computational Study of Parallel Algorithms for the All-Pairs Shortest Path Problem", in E. A. Lipitakis (Ed.), *Proceedings of the 2nd Hellenic-European Conference on Mathematics and Informatics* (Athens, Sep. 1994), Hellenic Mathematical Society, pp. 839–846.

[195] R. Sakellariou, "Partitioning Loop Nests Containing Conditionals for Automatic Parallelisation", *Proceedings of the 3rd Hellenic-European Conference on Mathematics and Informatics* (Athens, Sep. 1996), to be published.

[196] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*, Research Monographs in Parallel and Distributed Computing Series, MIT Press, 1989.

[197] A. Schrijver, *Theory of Linear and Integer Programming*, Wiley, 1986.

[198] W. Shang, M. T. O'Keefe, J. A. B. Fortes, "On Loop Transformations for Generalized Cycle Shrinking", *IEEE Transactions on Parallel and Distributed Systems*, **5**-2, Feb. 1994, pp. 193–204.

[199] W. Shang, J. A. B. Fortes, "Time Optimal Linear Schedules for Algorithms with Uniform Dependencies", *IEEE Transactions on Computers*, **40**-6, June 1991, pp. 723–742.

[200] Z. Shen, Z. Li, P.-C. Yew, "An Empirical Study of Fortran Programs for Parallelizing Compilers", *IEEE Transactions on Parallel and Distributed Systems*, **1**-3, July 1990, pp. 356–364.

[201] J. P. Singh, J. L. Hennessy, "An Empirical Investigation of the Effectiveness and Limitations of Automatic Parallelisation", in *Proceedings of the International Symposium on Shared Memory Multiprocessing*, Information Processing Society of Japan, Tokyo, Apr. 1991, pp. 25–36; also available as Technical Report CSL-TR-91-462, Computer Systems Laboratory, Stanford University.

[202] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, C. B. Moler, *Matrix Eigensystem Routines – EISPACK Guide (Second Edition)*, LNCS **6**, Springer-Verlag, 1976.

[203] E. Speckenmeyer, "Is Average Superlinear Speedup Possible?", in E. Börger, H. Kleine Büning, M. M. Richter (Eds.), *CSL '88: 2nd Workshop on Computer Science Logic* (Duisburg, Oct. 1988), LNCS **385**, Springer-Verlag, 1989, pp. 301–312.

[204] M. R. Spiegel, *Mathematical Handbook of Formulas and Tables*, Schaum's Outline Series, McGraw-Hill, 1993.

[205] X.-H. Sun, J. Gustafson, "Toward a better parallel performance metric", *Parallel Computing*, **17**-10&11, Dec. 1991, pp. 1093–1109.

[206] X.-H. Sun, D. T. Rover, "Scalability of Parallel Algorithm-Machine Combinations", *IEEE Transactions on Parallel and Distributed Systems*, **5**-6, June 1994, pp. 599–613.

[207] B. K. Szymanski (Ed.), *Parallel Functional Languages and Compilers*, ACM Press & Addison-Wesley, 1991.

[208] P. Tang, P.-C. Yew, "Processor Self-Scheduling for Multiple Nested Parallel Loops", in *Proceedings of the 1986 International Conference on Parallel Processing*, pp. 528–535.

[209] N. Tawbi, *Parallélisation Automatique: Estimation des Durées d' Exécution et Allocation Statique de Processeurs*, PhD Thesis, Université Pierre et Marie Curie, Paris, April 1991.

[210] N. Tawbi, "Estimation of Nested Loops execution time by Integer Arithmetic in Convex Polyhedra", in *Proceedings of the 8th International Parallel Processing Symposium*, IEEE Computer Society Press, 1994, pp. 217–221.

[211] N. Tawbi, P. Feautrier, "Processor Allocation and Loop Scheduling on Multiprocessor Computers", in *Proceedings of the 1992 International Conference on Supercomputing* (Washington D. C., July 1992), ACM Press, pp. 63–71.

[212] K. Tödter, C. Hammer, "PARC++: A Parallel C++", *Software–Practice and Experience*, **25**-6, June 1995, pp. 623–636.

[213] P. Tu, D. Padua, "Array Privatization for Shared and Distributed Memory Machines", *ACM SIGPLAN Notices*, **28**-1, Jan. 1993, pp. 64–67.

[214] P. Tu, D. Padua, "Automatic Array Privatization", in U. Banerjee, D. Gelernter, A. Nicolau, D. Padua (Eds.), *Languages and Compilers for Parallel Computing* (6th International Workshop, Aug. 1993), LNCS **768**, Springer-Verlag, 1994, pp. 500–521.

[215] T. H. Tzen, *Advanced Loop Parallelization: Dependence Uniformization and Trapezoid Self-Scheduling*, PhD Thesis, Department of Computer Science, Michigan State University, 1992.

[216] T. H. Tzen, L. M. Ni, "Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers", *IEEE Transactions on Parallel and Distributed Systems*, **4**-1, Jan. 1993, pp. 87–98.

[217] L. G. Valiant, "A Bridging Model for Parallel Computation", *Communications of the ACM*, **33**-8, Aug. 1990, pp. 103–111.

[218] G. von Laszewski, M. Parashar, A. G. Mohamed, G. C. Fox, "On the Parallelization of Blocked LU Factorization Algorithms on Distributed Memory Architectures", in *Proceedings of Supercomputing '92* (Minneapolis, Nov. 1992), IEEE Computer Society Press, pp. 170–179.

[219] D. R. Wallace, "Dependence of Multi-Dimensional Array References", in *Proceedings of the 1988 International Conference on Supercomputing*, (St. Malo, July 1988), ACM Press, pp. 418–428.

[220] K.-Y. Wang, "Precise Compile-Time Performance Prediction for Superscalar-Based Computers", in *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation* (Orlando, June 1994), *ACM SIGPLAN Notices*, **29**-6, June 1994, pp. 73–84.

[221] B. Wegbreit, "Mechanical Program Analysis", *Communications of the ACM*, **18**-9, Sep. 1975, pp. 528–539.

[222] S. Wholey, "Automatic Data Mapping for Distributed-Memory Parallel Computers", in *Proceedings of the 1992 International Conference on Supercomputing* (Washington D. C., July 1992), ACM Press, pp. 25–34.

[223] M. E. Wolf, *Improving Locality and Parallelism in Nested Loops*, PhD Thesis, Department of Computer Science, Stanford University, Aug. 1992; also available as Technical Report CSL-TR-92-538, Aug. 1992.

[224] M. E. Wolf, M. S. Lam, "A Data Locality Optimizing Algorithm", in *Proceedings of ACM SIGPLAN '91 Conference on Programming Language Design and Implementation* (Toronto, June 1991), *ACM SIGPLAN Notices*, **26**-6, June 1991, pp. 30–44.

[225] M. Wolfe, *Optimizing Supercompilers for Supercomputers*, Research Monographs in Parallel and Distributed Computing Series, MIT Press, 1989.

[226] M. J. Wolfe, C. Tseng, "The Power Test for Data Dependence", *IEEE Transactions on Parallel and Distributed Systems*, **3**-5, Sep. 1992, pp. 591–601.

[227] M. Wolfe, "Beyond Induction Variables", in *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation* (San Francisco, June 1992), *ACM SIGPLAN Notices*, **27**-7, July 1992, pp. 162–174.

[228] M. Wolfe, "The Definition of Dependence Distance", *ACM Transactions on Programming Languages and Systems*, **16**-4, July 1994, pp. 1114-1116.

[229] M. Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley, 1996.

[230] S. Wolfram, *Mathematica: A System for Doing Mathematics by Computer*, Addison-Wesley, 1988.

[231] J. Worlton, "Toward a Science of Parallel Computation", *Computational Mechanics Advances & Trends, AMD*, **75**, 1987, pp. 23–35.

[232] E. M. Wright, "Prouhet's 1851 Solution of the Tarry-Escott Problem of 1910", *American Mathematical Monthly*, **66**, 1959, pp. 199–201.

[233] Y. Wu, T. G. Lewis, "Parallelizing WHILE Loops", in *Proceedings of the 1990 International Conference on Parallel Processing* (Vol. II Software), Aug. 1990, The Pennsylvania State University Press, pp. 1–8.

[234] Z. Xu, K. Hwang, "Modeling Communication Overhead: MPI and MPL Performance on the IBM SP2", *IEEE Parallel & Distributed Technology*, **4**-1, Spring 1996, pp. 9–23.

[235] J. Xue, "Automating non-unimodular loop transformations for massive parallelism", *Parallel Computing*, **20**-5, May 1994, pp. 711–728.

[236] T. Yang, A. Gerasoulis, "PYRROS: Static Task Scheduling and Code Generation for Message Passing Multiprocessors", in *Proceedings of the 1992 International Conference on Supercomputing* (Washington D. C., July 1992), ACM Press, pp. 428–437.

[237] A. P. Yershov, "ALPHA–An Automatic Programming System of High Efficiency", *Journal of the ACM*, **13**-1, Jan. 1966, pp. 17–24.

[238] K. K. Yue, D. J. Lilja, "Parameter Estimation for a Generalized Parallel Loop Scheduling Algorithm", in *Proceedings of the* 28*th Annual Hawaii International Conference on System Sciences* (Vol. II, Software Technology), IEEE Computer Society Press, 1995, pp. 187–188; a more detailed version is available as Technical Report TR94-51, Department of Computer Science, University of Minnesota, 1994.

[239] X. Zhang, R. Castañeda, E. W. Chan, "Spin-Lock Synchronization on the Butterfly and KSR1", *IEEE Parallel & Distributed Technology*, **2**-1, Spring 1994, pp. 51–63.

[240] X. Zhang, Y. Yan, K. He, "Latency Metric: An Experimental Method for Measuring and Evaluating Parallel Program and Architecture Scalability", *Journal of Parallel and Distributed Computing*, **22**-3, Sep. 1994, pp. 392–410.

[241] H. P. Zima, H.-J. Bast, H. M. Gerndt, "SUPERB: A tool for semi-automatic MIMD/SIMD parallelization", *Parallel Computing*, **6**-1, Jan. 1988, pp. 1–18.

[242] H. Zima, B. Chapman, *Supercompilers for Parallel and Vector Computers*, ACM Press & Addison-Wesley, 1990.

[243] H. P. Zima, B. M. Chapman, "Compiling for Distributed-Memory Systems", *Proceedings of the IEEE*, **81**-2, Feb. 1993, pp. 264–287.