Programming Research Group

BSP SCHEDULING OF REGULAR PATTERNS OF COMPUTATION

Radu Calinescu

PRG-TR-1-97

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

BSP Scheduling of Regular Patterns of Computation

Radu Calinescu

January 1997

Abstract

One of the major challenges of the current research in the field of parallel computing is the development of a realistic underlying framework for the design and programming of general purpose parallel computers. The bulk-synchronous parallel (BSP) model is largely viewed as the most suitable candidate for this role, as it offers support for both the design of scalable parallel architectures and the generation of portable parallel code.

However, when considering the development of portable parallel software within the framework of the BSP model, one cannot disregard the existence of a broad basis of efficient sequential and PRAM solutions for the most various classes of problems. In fact, the recent emergence of reliable techniques for the identification of the potential parallelism of a sequential program has rendered the automatic parallelisation of existing sequential code more compelling than ever.

At first sight, BSP simulation of PRAMs appears to be the ideal strategy for taking advantage of this wealth of potential parallelism. Unfortunately, PRAM simulation heavily relies on the future manufacturing of parallel architectures providing powerful multithreading support and fast context switching/address translation capabilities. At the same time, simulation has been proven to yield inefficient solutions when computations on dense data structures are involved.

This report describes the first stage of a project proposing BSP scheduling as an alternative to PRAM simulation in the generation of portable parallel code. After introducing the BSP programming and cost model, the report presents a brief overview of the current trends in the identification and exploitation of potential parallelism. Then, techniques for the BSP scheduling of many regular patterns of computation which occur frequently in imperative programs are devised and analysed in terms of the BSP cost model. Finally, a whole chapter is dedicated to the mapping of generic loop nests on BSP computers. A discussion of the further stages of the project concludes the report.

Contents

1 Introduction

Although regarded as the indubitable near-future replacement of sequential computing for more than two decades, parallel computing largely failed to become so. As a result of the research that focused on the causes of this unsuccess, the concept of *general purpose parallel computing* has emerged [47, 66] as the proper way forward from this impasse. Consequently, a parallel computing model encompassing the characteristics that made the von Neumann model so successful in sequential computing is deemed necessary [48, 65] to trigger the long-expected transition. In order to achieve this goal, the new model must consistently permit both the design of general purpose parallel architectures and the development of portable software for them. Candidates for this role have not delayed their appearance: the XPRAM model of Valiant [66], the logP model of Culler et al. [17], the WPRAM model of Nash et al. [51] are but a few examples of possible such models. The most interesting proposal, however, is the bulk-synchronous parallel (BSP) model introduced by Valiant in [65].

Since its emergence in the early 1990s, the BSP model has continuously attracted the attention of many parallel algorithm and software designers. This attention is justified not only by the simplicity, elegance and generality of the model, but also by the fact that parallel applications developed around the BSP model are portable and costable. Indeed, the BSP programming discipline yields programs that are transportable across any general purpose parallel computer, and whose cost can be accurately analysed. As a result, BSP algorithms approaching the most various areas have been designed [65, 47, 48, 8, 26, 10, 11], and are currently used within larger applications.

Without understating the usefulness of this approach to portable parallel code design, one cannot neglect the existence of a large body of efficient sequential code, currently in use on many sequential computers. As powerful techniques for the extraction of the potential parallelism from sequential programs have been devised over the last two decades [4, 5, 53, 72], the exploitation of this body of sequential code seems more appealing than ever. Nevertheless, before the automatic derivation of portable parallel programs becomes a reality, reliable methods for the mapping of this potential parallelism on a general purpose parallel computer must be devised.

The availability of techniques for virtual parallelism identification from sequential code is not the only motivation of such an undertaking. Indeed, one can mention at least two other sources of virtual parallelism. The first source is the Parallel Random Access Machine (PRAM) model [24]—a theoretical, idealised model of parallel computing which has been at the very core of parallel algorithm design ever since parallel computing became a notable field of its own. This privileged position has led to a broad base of efficient PRAM algorithms being developed in the most various areas [27]; each such algorithm can also be viewed as a virtual parallel program. The second source of virtual parallel code is represented by the high-level programming languages with [60] explicit parallelism but implicit partitioning and scheduling. These PRAM-style programming languages have been repeatedly proposed as the norm for parallel software design [60, 67], and may become a major source of virtual parallelism in the future.

In order to take advantage of this wealth of virtual parallel code, the potential parallelism made explicit by this code must be mapped onto real general purpose parallel computers. As envisaged in [65], two ways of developing BSP (i.e., portable and costable) parallel applications exist. The first one is called *automatic mode programming*, and consists of directly executing virtual parallel code onto a BSP simulated PRAM. The second solution, called *direct mode programming*, requires that the programmer retains full control of the memory and communication management. Apparently, the automatic mode represents the solution for the exploitation of the existing basis of virtual parallel code. However, although optimally efficient techniques for PRAM simulation on BSP computers have been developed [66], their serious limitations render the automatic mode programming unfeasible. Indeed, the BSP simulations of PRAM place restrictive demands on the target parallel architecture, requiring powerful multithreading support and fast context switching/address translation capabilities. Then, when locality exists in the original virtual parallel code, simulation leads to solutions hiding unnecessarily large multiplicative constants behind the optimal complexity notations.

As a consequence, the only realistic solution is to exploit the potential parallelism of an application in the context of the direct mode programming. So far, this method of devising BSP code has largely been viewed as requiring the involvement of the programmer, i.e., as a fully manual method. This report aims to prove that this image of the direct mode programming is misleading, by providing scheduling techniques for the automatic development of direct BSP code. The usage of scheduling for the mapping of potential parallelism on a parallel computer is not an entirely new idea. However, the great majority of the scheduling approaches proposed so far have only tackled the generation of architecture-specific parallel code, yielding non-portable programs.

The strategy adopted by the project whose first stage is described in this report is to identify common patterns of computation and to devise BSP scheduling techniques for these patterns. Once a collection of such scheduling techniques is available, the BSP scheduling of whole virtual parallel programs will be approached. This will be done by decomposing the virtual parallel programs into skeletons that match one of the computation patterns for which a BSP scheduling technique exists, and scheduling these code skeletons accordingly.

Several benefits are expected as a result of undertaking this project. Firstly, this research will provide a set of techniques permitting the utilisation of existing virtual parallel code as a front-end for the derivation of portable parallel software (and, in the long run, the utilisation of sequential code for the same purpose). Secondly, the realisation of the project's ultimate goal will considerably ease the task of parallel software designers, by enabling the use of PRAM-style high level languages as realistic parallel programming environments. Finally—and possibly most important—our study will provide an insight into the intricate mechanisms of the direct mode programming of general purpose parallel computers.

The current report approaches the BSP scheduling of regular patterns of computation that appear frequently in imperative programs. The report is organised as follows. In Chapter 2, the structure of a bulk-synchronous parallel computer, and the BSP programming and cost model are briefly presented. Also in Chapter 2, we introduce the pseudocode used to describe the BSP schedules devised in this report. Chapter 3 comprises a brief survey on potential parallelism identification through data dependence analysis and code transformation, and defines data dependence types and representations used throughout the report. An overview of classical techniques for the simulation and scheduling of potential parallelism, as well as a presentation of the previous work on portable parallel code derivation through scheduling are included in Chapter 4. Then, in Chapter 5, techniques for the BSP scheduling of various regular patterns of computation are devised and analysed in terms of the BSP cost model. Chapter 6 is dedicated to the BSP scheduling of generic loop nests, i.e., of loop nests that do not match any regular pattern of computation. Finally, a short summary of the results obtained so far and the first conclusions emerging from these results are presented in Chapter 7.

2 The bulk-synchronous parallel model

The bulk-synchronous parallel (BSP) model of parallel computation was introduced by L.G. Valiant [65] in the early 90s, being further developed over the recent years by researchers from Oxford and Harvard Universities [47, 26, 16, 48]. The primary purpose of the model is to act as a "bridging model" [65] mitigating the discrepancies between parallel architectures and the software executed on them. At the same time, the BSP model is intended to be a universal model of parallel computing, able to provide a reliable underlying framework for the devising of both scalable parallel architectures and portable parallel software. The new model encountered an immediate success, largely due to a simplicity, elegance and generality which made it to be regarded as the parallel computing equivalent of the von Neumann model of sequential computation [65, 47]. This chapter overviews the characteristics of BSP model, discussing the reasons for its power, and the novelty of its approach.

2.1 Bulk-synchronous parallel computers

A bulk-synchronous parallel computer is defined [65] as a system comprising three elements: a set of processor/memory pairs (or *units*), a communication network permitting point-to-point message delivery between pairs of units, and a mechanism for the efficient barrier synchronisation of the processors. No special broadcasting or combining facilities are assumed. The only requirement is that the communication network provides uniformly efficient non-local memory access, and this requirement can typically be satisfied by using two-phase randomised routing [66]. Similarly to the definition of the von Neuman computer, this definition is general enough to describe all existing and future types of parallel computers. At the same time, the definition clearly specifies the basic building blocks that, in one form or another, must be included into any feasible parallel computer.

A bulk-synchronous parallel computer is fully characterised by four parameters:

- s, the processor speed;
- p, the number of processor/memory units;
- L, the synchronisation parameter;
- g, the communication parameter.

The parameter s gives the speed of the processors in floating point operations (or *flops*) per second. The parameter L was originally defined as the *synchronisation periodicity* or the minimum distance between successive barrier synchronisations [65, 47]. However, due to its role in assessing the cost of a BSP algorithm, L is largely viewed nowadays as the cost of performing a barrier synchronisation of all processors. In order to express this cost in a way which is coherent with the cost of local computations, the parameter L is normalised with respect with s, i.e., it is measured in flops rather than in real time units. Typically, the hardware of the parallel computer imposes a lower bound on the value of L, bound which may or may not be met by the software layer.

The communication parameter g can also be defined in several equivalent ways. First, g can be regarded [47] as the ratio between the total number of local operations performed by all processors in a time unit and the total number of words delivered by the commu-

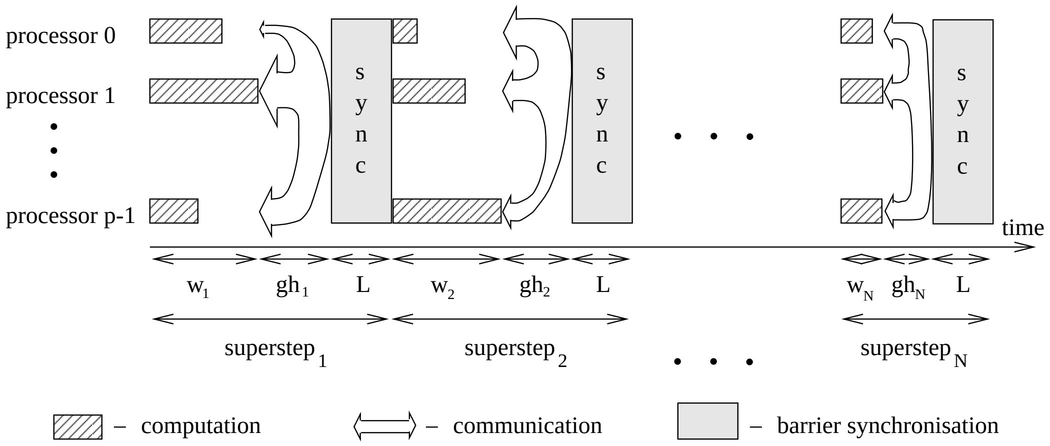


Figure 1: A bulk-synchronous parallel computation.

nication network in a time unit. This interpretation of g indicates that, for an algorithm to be efficient, at least g local operations need to be performed for each data transfer. Accordingly, g can also be viewed as a measure of the throughput of the communication network. However, we prefer the definition in [65], which relates g to the cost of realising a so-called h-relation. An h-relation is a communication pattern in which any processor sends and receives at most h items of data (or words); for conformity with the definition of L, we will assume that the length of a word is that of a floating point data. Then, the cost of realising an h-relation is $g\max\{h, h_0\}$, where h_0 is a threshold value accommodating the start-up cost of the communication. In other words, if $h \geq h_0$, the implementation of an h-relation costs gh. This result shows that in order to achieve efficient communication, the size of the exchanged messages must be large enough. The parameter g is also normalised with respect to s, i.e, it is expressed in flops per floating point word.

As shown in [48], any parallel system can be regarded as a BSP computer, and represents a point in the (p, L, g) space of BSP computers. As we shall see in the following sections, it is exactly this abstraction from the hardware details of the machine that allows the development of portable and costable parallel software.

2.2 The BSP programming model

A BSP computation consists of a sequence of supersteps (Figure 1). The processors proceed asynchronously through each superstep, and barrier synchronise at the end of the supersteps—hence the name of "bulk-synchronous" parallel. Within a superstep, the processors are allowed to independently execute operations on locally held data and/or to initiate read/write requests for non-local data. However, the non-local memory accesses initiated during a superstep take effect only when all the processors reach the barrier synchronisation that ends that superstep.

The crucial benefit of this programming model is [65, 48] the separation of the three main components of a parallel application, namely computation, communication, and synchronisation. As a result, the cost of each of the three components can be independently and accurately assessed.

2.3 The BSP cost model

The BSP cost model is compositional: the cost of a BSP program is simply the sum of the costs of its constituent supersteps. Several expressions have been proposed for the cost of a single superstep. Thus, in [8], the authors adopted the expression

$$cost(i) = \max\{L, w_i, gh_i\}$$

for the cost of superstep i, $1 \leq i \leq N$ of a N-superstep BSP program, where w_i represents the maximum number of local operations executed by any processor during superstep i, and h_i is the maximum number of words sent or received by any processor in superstep i. This formulation of the BSP cost accounts for a desired overlapping of computation and communication, and considers one of the synchronisation mechanisms proposed in [65], namely that in which the system checks the termination of a superstep every L time units.

A more conservative alternative is [26] to charge a cost of

$$\max\{L, w_i + gh_i\}$$

for the execution of superstep i of a BSP computation. This cost is consistent with the fact that real parallel computers are often unable to fully overlap computation and communication. Indeed, as pointed out in [49], on many computers the most costly communication operation is the transfer of data from the operating system to the application buffers, and this operation typically requires processor participation.

Since a specialised mechanism for processor synchronisation is seldom available on the existing parallel computers, barrier synchronisations are usually implemented through inter-processor communication. As a consequence, it is often the case that the situation in Figure 1 arises, i.e., that the cost of synchronisation must be charged in addition to the computation and communication costs. Accordingly, the cost of superstep i, $1 \leq i \leq N$ is in this case

$$cost(i) = L + w_i + gh_i, \qquad (1)$$

with L representing a measure of the latency of the communication network. It is this expression of the cost of a superstep that we will use throughout this report. However, it is worth noticing that all the expressions of the BSP cost presented in this section are equivalent within a small multiplicative constant, and the consistent use of any of them would lead to similar results.

So far we considered that, irrespective of the value of h_i—the maximum number of words sent or received by any processor during superstep i—the contribution of communication to the cost of superstep i is gh_i. This is apparently against the requirement that

h must be greater than or equal to a threshold value h_0 for the cost of an h-relation to be gh. The explanation is that the parameter L is large enough to cover for the start-up cost of an h-relation (i.e., $L \geq gh_0$). Therefore, it is always safe to consider that the realisation of an h relation within a superstep brings a contribution of gh to the cost of that superstep.

The standard way of assessing the efficiency of parallel algorithms is to compare them with the best (known) sequential algorithm that solves the same problem. The same strategy applies to BSP algorithms. Assume for instance that the sequential cost of solving a problem of size n is $T(n)$, and the cost of an N-superstep BSP algorithm solving the same problem is

$$NL + \sum_{i=1}^{N} w_i + g\sum_{i=1}^{N} h_i = N(n,p)L + W(n,p) + gH(n,p).$$

Several ways of evaluating the efficiency of BSP algorithms have been proposed so far. In [48], a BSP algorithm is considered to be efficient if $W(n,p) = T(n)/p$, and $N(n,p)$ and $H(n,p)$ are as small as possible. Also, in [26], two performance metrics, $\pi = pW(n,p)/T(n)$ and $\mu = gH(n,p)/(T(n)/p)$ are defined, and a BSP algorithm is regarded as optimal if $\pi = 1 + \mathrm{o}(1)$ and $\mu = \mathrm{o}(1)$ or $\mu \leq 1$.

We will assess the optimality of the BSP schedules devised in this report in a very similar way. Thus, we will say that a BSP schedule is k-optimal in computation if $pW(n,p)/T(n) = k + \mathrm{o}(1)$. For each schedule with $N(n,p)/W(n,p) = \mathrm{o}(1)$ and $H(n,p)/W(n,p) = \mathrm{o}(1)$, we will also strive to find a threshold n_0 such that for $n \geq n_0$, $W(n,p) \gg \max\{N(n,p)L,\ gH(n,p)\}$ (a computation cost which is a order of magnitude larger than the communication and synchronisation overheads will be viewed as sufficient for practical purposes). Whenever this is possible, we will say that the schedule is k-optimal for $n \geq n_0$, and expect to obtain a p/k speedup for a practical implementation of the schedule.

2.4 The development of BSP applications

Two methods of developing a BSP application have been envisaged in [65]. The first method, called *automatic mode programming*, consists of writing the program in a PRAM style (i.e., without explicitly addressing the memory and communication management issues) and simulating the resulting code onto the target BSP computer. Optimally-efficient BSP simulations of PRAMs have been devised [65, 66] for this purpose. The second method, called *direct mode programming* allows the programmer to retain control of memory and communication management. As emphasised in [65], the latter approach "avoids the overheads of automatic memory management and may exploit the relative advantage in throughput of computation over communication that may exist".

Despite the solid theoretical results underlying PRAM simulation, practical BSP implementations of PRAM simulation are still in their infancy. Direct BSP algorithms and applications, on the other hand, have been extremely successful from the very early days of the BSP model. Besides the simplicity of designing and analysing direct BSP algorithms, this success is also due to the appearance of several very powerful environments for the development of portable BSP software. These environments include the

Oxford BSP Library [50] and the Green BSP Library [29], culminating with the recent proposal [30] and implementation of a BSP Worldwide standard library called BSPlib. All these libraries comprise a limited number of functions that can be called from C, Fortran, and other imperative languages with a similar memory model. Furthermore, the libraries have been implemented on a large number of parallel architectures, using native or generic primitives available on these machines.

2.5 BSP pseudocode

The sequential code whose mapping on a BSP computer is approached in this report, as well as the resulting BSP schedules are described in pseudocode. This subsection presents the conventions used in our pseudocode.

First, for the sake of brevity, variables will not be explicitly declared. We will implicitly assume that loop indices are integers, whilst other variables and array elements will be assumed to be of floating point type, unless otherwise specified. Second, indentation will be used to indicate block limits instead of `begin`/`end` or similar constructs. Finally, less important and simple parts of a schedule will be described informally rather then being specified in detail.

Three broad classes of statements will be used to describe a piece of code: assignments, conditional statements, and repetitive statements. All three types of statements have the same interpretation as in a typical imperative programming language, such as C or Fortran. The generic form of an assignment statement is

```
variable = expression
```

where *expression* is an expression whose type coincides with that of the variable on the left-hand side of the assignment. A conditional statement has the generic form:

```
if condition then
  statement
else
  statement
```

where *condition* is a boolean condition, and the `else` part is optional. Two types of repetitive statements are used, the first one to describe sequential loops:

```
for index=lower_bound,upper_bound do
  statement
```

and the second to describe parallel loops:

```
forall index=lower_bound,upper_bound do in parallel
  statement
```

Finally, we will use the constructs `BSP_start_superstep` and `BSP_end_superstep` to mark the beginning and the end of supersteps when describing BSP code.

$S2_{(y_1,y_2,\dots,y_K)}$	$S1_{(x_1,x_2,\dots,x_K)}$	
	read	write
read	—	$S1_{(x_1,x_2,\dots,x_K)}\delta S2_{(y_1,y_2,\dots,y_K)}$
write	$S1_{(x_1,x_2,\dots,x_K)}\overline{\delta}S2_{(y_1,y_2,\dots,y_K)}$	$S1_{(x_1,x_2,\dots,x_K)}\delta^{o}S2_{(y_1,y_2,\dots,y_K)}$

Table 1: The type of the dependence between two statement instances (at least one of which modifies the memory location accessed by both instances) depends only on the type of accesses to the commonly used variable, and on their execution order. In this table, the lexicographical order of the index vectors is $(x_1, x_2, \dots, x_K) < (y_1, y_2, \dots, y_K)$.

3 Data dependence analysis and code transformation

3.1 Data dependence analysis

Three types of data dependence between two statement *instances* (the term 'instance' refers to an actual execution of a statement in the program; if a statement belongs to a loop body and is executed many times, each such execution is considered a different instance of that statement) may prevent them from being concurrently executed (Figure 2):

- *true dependence* (or *flow dependence*), when one of the statement instances reads a variable earlier modified by the other;
- *antidependence*, when one of the statement instances uses the value of a variable which is later written by the other statement instance;
- *output dependence*, when both statement instances write the same memory location.

In each of the three cases, any *legal* transformation of the original program must preserve the order in which the two statement instances are executed (exceptions such as the output dependence of two statement instances that assign the same value to a variable, etc. are not discussed here). As shown in Table 1, the type of a data dependence depends only on the order and type of accesses made to the commonly used location of memory.

There is also a control dependence that may exist between two statement instances, namely when the execution of one statement is conditioned by the result of evaluating the other. However, such dependences may be systematically converted to data dependences [2], and therefore no separate consideration of control dependences is necessary.

Numerous tests, varying from simple and approximate [5, 72] to complex and exact [56] have been devised to identify whether two array references address the same memory location. Whenever an approximate test fails to prove independence, dependence is unconditionally assumed.

Prior to its usage for the extraction of potential parallelism from the analysed sequential code, the dependence information acquired as a result of this analysis is "encoded" in a concise form. Three representations of this information are most popular:

```
        for i₁=1,n₁-1 do
          for i₂=0,n₂-2 do
S1:         a[i₁,i₂]=...
S2:         ...=f(a[i₁-1,i₂+1])
```

(a) true dependence: the value of $a[x_1, x_2]$ modified by the instance of S1 for $(i_1, i_2) = (x_1, x_2)$ is later used by the instance of S2 for $(i_1, i_2) = (x_1 + 1, x_2 - 1)$. This data dependence is denoted $\mathrm{S1}_{(x_1,x_2)}\delta\,\mathrm{S2}_{(x_1+1,x_2-1)}$.

```
        for i=0,n-2 do
S1:       a[i] = f(a[i+1])
```

(b) antidependence: the value of $a[x]$ read by the instance of S1 for $i = x$ is later written by the instance of S1 for $i = x + 1$. The notation used for this type of data dependence is $\mathrm{S1}_{(x)}\overline{\delta}\,\mathrm{S1}_{(x+1)}$.

```
        for i₁=0,n₁-1 do
          for i₂=1,n₂-1 do
S1:         a[i₁,i₂]=...
S2:         a[i₁,i₂-1]=...
```

(c) output dependence: the variable $a[x_1, x_2]$ is first written by the instance of S1 for $(i_1, i_2) = (x_1, x_2)$, then by the instance of S2 for $(i_1, i_2) = (x_1, x_2 + 1)$. The output dependence is denoted $\mathrm{S1}_{(x_1,x_2)}\delta^{\mathrm{o}}\mathrm{S2}_{(x_1,x_2+1)}$.

Figure 2: Three types of data dependence ((a)–(c)) may require that two statement instances are executed in a fixed order in any legal transformation of a given program. A generic dependence between two statement instances $\mathrm{S1}_{(x_1,x_2,\ldots,x_K)}$ and $\mathrm{S2}_{(y_1,y_2,\ldots,y_K)}$ is denoted $\mathrm{S1}_{(x_1,x_2,\ldots,x_K)}\delta^*\mathrm{S2}_{(y_1,y_2,\ldots,y_K)}$ (i.e., $\delta^* = \delta + \overline{\delta} + \delta^{\mathrm{o}}$).

- The *statement dependence graph* [3] is a directed graph representation of the data dependences. In this graph, each statement belonging to the analysed loop body is assigned a vertex, and each dependence relation between two statement instances is codified as an edge joining the vertices associated with the two statements. The edges are labeled with the level of the innermost loop for which the dependence still holds; this is the minimal information needed to decide whether a given statement may be vectorised from a certain loop level inwards.

- The *distance vector* [39] and the *direction vector* [72] representations of data dependences store instead of several individual dependences only the difference between the index vectors, respectively the relations (i.e., '$<$', '$>$', '$\neq$', etc.) between the components of the index vectors for which the data dependence exists. For instance, for the three examples in Figure 2, the distance vectors are: (a) $(1, -1)$, (b) (1), (c) $(0, 1)$; the direction vectors for the same code samples are: (a) $(<, >)$, (b) $(<)$, (c) $(=, <)$. It is worth noticing that the first non-zero (non-'$=$') component of a distance (direction) vector is always positive ('$<$'). The information provided by these two representations of data dependences is much more detailed than that preserved by the statement dependence graph, and is useful in deciding the legality of many transformations that can be applied to a loop nest. Several variations of these two representations have also been proposed. For instance, Sarkar and Thekkath [62] have devised a type of dependence vector whose elements are distance values whenever such distances exist as constants, or are direction components otherwise. Similarly, Wolf and Lam [70] have described a dependence vector whose components are neither values, nor directions, but finite integer intervals; thus, many distance vectors may be represented in a more compact way. Finally, in [35], Irigoin and Triolet have introduced the so-called *dependence cone*, i.e., the description of the entire set of dependence vectors of a loop nest by the multi-dimensional cone they span in the K-dimensional Euclidean space, where K is the depth of the loop nest. A comprehensive evaluation of the relations among all these representations, as well as the specification of the less precise representation needed to asses the legality of various code transformations can be found in [74].

- The *communication-free partition* of the iteration space [44] represents a partitioning of the iteration space of a nested loop into subsets of iterations which share no common memory reference. Consequently, the iterations from different such subsets may be considered for parallel execution.

3.2 Code transformation techniques

In order to expose the potential parallelism of a sequential program, the data dependence analysis phase must be followed by a phase in which the dependence information is used to operate legal transformations into the source sequential code. The result of these transformations is a code representation of the potential parallelism in the analysed sequential program. Typically, this *virtual parallel code* cannot be directly executed on a parallel computer, but needs to be adjusted to fit the characteristics of the target machine.

As this phase has been by far the most explored area related to sequential code parallelisation, we are only able to include here a brief listing of the main approaches proposed so far. For each approach, a number of references are provided for the reader interested in learning more about a specific transformation or method. Comprehensive surveys of the results mentioned in the remainder of this section are available in [4, 53, 59, 72].

A first technique worth mentioning is the π-block-based vectorisation proposed by Allen and Kennedy in [3]. This technique makes use of the statement dependence graph, and attempts to vectorise loop nests starting from the outermost loop in the nest. In each step, the so-called π-blocks (i.e, the maximal cycles or individual vertices belonging to no cycle) of the dependence graph are identified, and processed in topological-sorted order as follows. If the π-block to be processed consists of a single vertex, then vector code is generated for it; otherwise (i.e., if the π-block is a cycle), the current loop is left unchanged (i.e., sequential) and the vectorisation of innerer loops of the π-block is attempted after eliminating the edges labeled with the level of the current loop.

A second technique providing a code representation of the potential parallelism of a loop nest is the mapping of a communication-free partition of the iteration space on a virtual processor space (i.e., on a PRAM machine). A method to automatically generate this mapping and obtain the corresponding (PRAM-like) parallel code is presented in [44]. The main limitation of this technique is that feasible (e.g., well balanced) communication-free partitions may be impossible to obtain in many cases.

A very rich collection of loop transformations based on distance or direction vectors has been developed with the ultimate goal of concurrentising one or some of the loops in a nested loop. This collection includes (but is not limited to) the following transformation techniques:

- loop interchange—the swapping of the position of two loops in a perfect loop nest [3, 72], and loop permutation—the generalisation of loop interchange [6];
- loop fusion—the joining of two loops [21], and loop fission—the separation of a single loop into two loops [38];
- loop reversal—the execution of a loop in reversed iteration order [69];
- loop coalescing—the reduction of a loop nest to a single loop with the index taking one value for each point in the original iteration space [52, 55];
- loop unrolling—the replication of the loop body and shrinking of the iteration space [19];
- strip mining—the partitioning of the index range of a loop into strips [45], and tiling—the generalisation of strip mining to multiple loops [35, 57, 73];
- loop skewing—a very useful transformation which "skews" the iteration space such as to align independent points in it [35, 40, 57, 72];
- loop concurrentisation—the specification that a loop may be concurrently executed [72, 74].

The devising of this avalanche of code transformation techniques has been accompanied by significant research efforts aiming at the development of unifying frameworks for their validation and application. The most promising approaches proposed so far are described in the following.

The Banerjee's unimodular transformation framework [7, 71] is an approach which uses affine transformations whose transformation matrix determinant has value ± 1. This framework is able to model loop permutation, reversal and skewing. A generalisation of the unimodular framework to general nonsingular affine transformations is provided in [43].

Kelly and Pugh [37] have developed a framework that expresses compositions of loop interchange, skewing, reversal, fusion, fission, etc. as 1–1 affine mappings of the original iteration space to a new iteration space, possibly with a different dimensionality.

In [41], Lengauer extends the use of the polytope model from the systolic design area to loop parallelisation, defining a generic transformation of the iteration space in terms of space-time mappings from an original source polytope to a target polytope.

Finally, Feautrier [22, 23] describes an integrated framework using piece-wise affine "schedules" that map independent points in the iteration space to the same point of the (possibly multi-dimensional) execution time. These schedules are chosen such as to maximise parallelism, playing the role of combinations of usual loop transformations.

The application of the code transformations techniques described in this section, either individually or using one of the above mentioned frameworks, produces a virtual parallel program, i.e., a program in which the potential parallelism of the initial sequential code appears explicitly (e.g., a PRAM-style or a vector program). It is worth mentioning that such a program may also be the result of programming in a parallel programming language with explicit parallelism but implicit partitioning and scheduling [60].

4 Simulation and scheduling of potential parallelism

In order to exploit the potential parallelism identified in a sequential program, this parallelism must be mapped onto a real parallel computer. Two strategies can be used to accomplish this goal. The former strategy, called *simulation*, consists of writing a PRAM version of the original code (i.e., a version in which all the potential parallelism appears explicitly), and simulating the resulting PRAM program on the target parallel computer. The simulation technique that is used does not depend on the characteristics of the application to be parallelised, but only on the structure of the target parallel architecture. The latter strategy, called *scheduling*, distributes the data and the computation of the original program across the parallel machine in accordance with both the parameters of the parallel architecture and the structure of the program itself.

Potential parallelism exploitation through simulation is simpler. Indeed, once a PRAM simulation technique appropriate for the considered parallel computer is devised, it can be repeatedly used to parallelise any application. However, as we will show in this chapter, scheduling typically yields parallel programs that outperform those induced by simulation. This is due to the fact that scheduling takes advantage of the characteristics of the individual application when developing a parallel version of it.

4.1 Potential parallelism simulation

The straightforward technique for the mapping of potential parallelism on a real parallel architecture is simulation. Although potential parallelism simulation on shared-memory systems has been considered and it poses certain problems itself [64], in this section we shall only refer to simulation on distributed-memory multiprocessors, as this class of machines is far more common and is directly related to our project. For a distributed-memory machine, virtual parallelism simulation requires two distinct operations, namely the even partitioning of the parallel threads of computation among the available processors, and the uniform distribution of memory references among the local memories. The goal of the first operation is obvious; the latter one aims [65] at allocating memory references in a manner which prevents uneven memory accesses, and hence assures that no overloaded processor-memory unit will exist to slow down the overall computation.

As for many years now it has been acknowledged as the most convenient framework for the design and analysis of parallel algorithms, the PRAM model has also constituted the front-end of most simulation approaches. This choice is highly justified by the existence of a broad collection of efficient PRAM algorithms, by the easiness of programming in PRAM-style high level languages, as well as by the encouraging results achieved in the automatic translation of sequential code to PRAM code (see Chapter 3 for details).

Consider the simulation of a v-processor PRAM program using a storage of size m on a p-processor parallel computer; the following performance metrics are used to asses the quality of the simulation [33]:

- the *slowdown* or the *delay* of the simulation; if the PRAM program requires T steps and the simulation takes $T' = \delta T$ steps, then the slowdown is δ;

- the *memory blowup* of the simulation; if the simulation requires $m' = \beta m$ memory to execute the PRAM program, then β is the memory blowup;

- the *efficiency* of the simulation is $\varepsilon = W/W'$, where $W = vT$ and $W' = pT'$ represent the amounts of work corresponding to the PRAM program and to its simulation, respectively.

The simulation is considered (*work*) *optimal* or *optimally efficient* if its efficiency ε is independent of T, v, and p. It is easy to see that if a simulation is optimal, then $\delta = \mathrm{O}(v/p)$; the ratio v/p is called the *parallel slackness* of the simulation.

Any PRAM simulation is fully characterised by the memory management technique and the routing technique it uses, and by its target parallel architecture. According to these characteristics, several classifications of PRAM simulation are possible:

- simulations targeted at special purpose parallel architectures (e.g., mesh-connected computers [42, 54], optical-communication parallel computers [28], etc.), or at general purpose parallel computers [65, 66];

- PRAM simulations using deterministic [34, 54] or randomised [42, 65, 66] techniques for memory management and routing;

- simulations requiring supra-unitary parallel slackness [42, 65, 66] or not.

Although more desirable due to their easily controllable behaviour, deterministic PRAM simulations suffer from poor worst-case performance (e.g., for the straightforward distribution of the PRAM storage among local memories, the worst-case slowdown is as high as $\Omega(p)$[1]). It is true that better results can be obtained employing for instance multiple copies of the simulated PRAM storage. However, the best theoretical results in terms of slowdown and memory blowup require [34] the use of expander graphs for the memory management and routing schemes, being of little practical utility. Therefore, in the remainder of this section we will focus on randomised approaches; for more background on deterministic simulation see the excellent survey in [33].

In randomised PRAM simulation, the memory management is realised through *hashing*, i.e., by using an appropriate hash function for the mapping of the PRAM memory locations to 'randomly'-distributed locations in the p local memories of the target architecture. To be efficient, this technique requires the utilisation of *fast good hash functions* as they are called in [65]. These are hash functions that are quickly computable at run time, and successfully approximate a random distribution. As for the routing of remote memory requests, *two-phase randomised routing* [66] or even *greedy routing* can be utilised since the PRAM memory references are randomly distributed among the local memories of the target machine.

Using the above mentioned techniques, and exploiting a certain degree of parallel slackness, Valiant has derived several randomised PRAM simulations that are optimal with high probability [65, 66]; these simulations are targeted at the BSP model and

[1] $v = p$ is assumed here

its instance, the XPRAM, respectively. In all these approaches, supra-unitary parallel slackness is required as a mean to hide the network latencies specific to distributed-memory parallel computers. Thus, each step of the simulated PRAM program is executed in a BSP superstep in which each processor simulates approximately v/p virtual processor steps. If the parallel slackness is large enough, then the communication network will have to realise a $\mathrm{O}(v/p)$-relation for the considered superstep with high probability. Further assuming that this relation can be implemented in time $\mathrm{O}(v/p)$ (i.e., assuming that $g = \mathrm{O}(1)$), PRAM simulations are obtained that are optimally efficient with high probability. Applying this strategy, a p-processor BSP computer has been shown [66] to be able to simulate a v-processor EREW (CRCW) PRAM with expected optimal efficiency if $v/p \geq \log p$ (respectively if $v/p \geq p^{\epsilon}$ for some $\epsilon > 0$). Assuming that the address translation takes place at run time, the corresponding memory blowup is $\mathrm{O}(1)$.

To conclude, PRAM simulation represents a very promising approach to parallel code derivation: it is simple and can be easily automated once the memory management and routing schemes are specified. However, even the optimally-efficient PRAM simulations have a number of serious limitations. Thus, they require the use of fast good hash functions, and assume that a sufficient degree of parallel slackness is available in the simulated code. Furthermore, PRAM simulation places restrictive demands on the target parallel architecture, requiring a fast context-switching mechanism and powerful address translation capabilities.

Although suitable hash functions have been proven to exist (e.g., the class of polynomials of degree[2] $\log p$ in arithmetic modulo m), and parallel architectures providing adequate multithreading support might be manufactured in the future, a final limitation exists which renders the universal usage of simulation for portable parallel code design very unlikely. Indeed, as shown in [58], simulation may hide unnecessarily large multiplicative constants behind the optimal complexity notations when locality exists in the source virtual parallel code. To illustrate this, let us consider for instance the BSP simulation of a $T(n)$-step EREW PRAM algorithm written for $v \geq p \log p$ virtual processors, where p is the number of processors of the BSP computer. Using the randomised simulation technique devised in [66], one obtains an optimally efficient BSP program with the cost

$$T(n)\left[L + \frac{v}{p} + g\frac{v}{p}\right] \leq T(n)\frac{v}{p}(g+2) = \mathrm{O}\left(T(n)\frac{v}{p}\right)$$

if $L \leq v/p$. However, if data locality can be exploited, a second optimal BSP program may be obtained through a "careful", explicit mapping of the memory references to local memories (i.e., through scheduling). This mapping is very likely to result in only a constant number of the $T(n)$ supersteps involving communication among processors, and hence to an algorithm whose cost is

$$T(n)\left[L + \frac{v}{p}\right] + cg\frac{v}{p} \leq T(n)\frac{v}{p}(2 + \mathrm{o}(1)) = \mathrm{O}\left(T(n)\frac{v}{p}\right)$$

[2] in fact, even polynomials of constant degree (including linear transformations) have been shown to produce good practical results

where c is a constant, if $L \leq v/p$ and (for instance) $T(n) = \log n$. Thus, while both the simulation and the "careful" mapping result in optimally-efficient BSP programs, the simulation leads to $(g+2)$-optimality, whereas the *scheduling* yields two-optimality. The explanation for this result is that the randomised distribution of the PRAM memory references, while ensuring an expected uniform access to individual memory modules, destroys any data locality that might have existed in the original program. As a consequence, the usage of simulation leads to guaranteed best possible results only when the simulated virtual parallel program deals with sparse data structures, i.e., when the code to be mapped onto the real parallel computer presents no locality whatsoever.

4.2 Potential parallelism scheduling

The second method that can be used to map potential parallelism on a real parallel computer is scheduling. Unlike simulation, the scheduling of potential parallelism consists of grouping the operations (or statements) of the virtual parallel program into tasks, and assigning these tasks to the real processors together with the "accompanying" data[3]. In doing so, certain *precedence relations* among tasks (i.e., precedence constraints due to data dependences) must be respected in order to obtain a correct schedule. Therefore, it is customary for scheduling algorithms to use as input a task graph representation of the virtual parallelism, namely a directed graph whose vertices represent tasks, and whose edges stand for precedence relations between pairs of tasks.

A formal definition of scheduling is provided by El-Rewini et al. in [20]: given a *parallel program task model* (i.e., a set T of n partially ordered tasks with durations and communications) and a *target machine model* (i.e., a set of parameters encapsulating the computation and communication characteristics of the target parallel machine), the scheduling problem consists of finding a function $s : T \rightarrow (1..p) \times \mathbb{N}$ such that

- for any $v \in T$, $s(v) = (i, t)$ means that task v is to be executed by processor i starting at time step t;
- all precedence relations among tasks are respected;
- the *schedule length*

  $$length(s) = \max\{v \in T \mid s(v) = (i,t) \bullet t + \tau_i(v)\},$$

  where $\tau_i(v)$ is the execution time of task v on processor i, is minimum among all possible schedules.

Unfortunately, with the exception of a few restrictive instances (e.g., when the communication costs are ignored and the task graph is a tree, or when $p = 2$ and the communication costs are again neglected), the scheduling problem has been proven to be NP-complete [20, 46, 60]. Hence, in most cases, a "near optimal" solution is sought instead of an optimal one, typically by means of a greedy heuristic with polynomial complexity in the size of the input task graph.

[3] hence, data locality in the original code is preserved

Scheduling techniques can be broadly classified into *static* and *dynamic*. Static scheduling techniques are applied at compile time, based on a priori knowledge of the program to be scheduled and of the target machine. Dynamic scheduling techniques, on the other hand, postpone scheduling decisions until run time, when more information such as input data or parallel system status is available. Hybrid approaches that combine methods specific to static and dynamic scheduling also exist. A finer categorisation of scheduling strategies is possible, but its inclusion here is beyond our purpose; for a thorough taxonomy of scheduling techniques the reader is referred to the work of Casavant and Kuhl [15].

Two types of static scheduling strategies are most common, list scheduling heuristics, and clustering heuristics. The approaches belonging to the former class comprise two steps. In the first step, each task is assigned a priority, and a list of tasks is constructed in decreasing order of priorities. Typical task priorities include [1, 36] the *task level*—the length of the longest path from the considered task to an exit task in the task graph, the *task co-level*—the length of the longest path from a starting point to the considered task, etc. Then, in the second step, a greedy procedure is used to iteratively select for execution the highest priority ready task whenever one of the processors becomes idle. Asynchronous calls to communication primitives may also be generated in this second step to solve the data dependences among tasks.

As concerns the clustering approaches, they work by identifying and grouping into *clusters* (i.e., sets of tasks to be executed on the same processor) the tasks that communicate heavily [15] or belong to long paths in the task graph [20]. The algorithms in this class work as follows. Initially, each task is assigned to a separate cluster. Then, in a sequence of "refinement" steps, pairs of clusters whose unification is most desirable according to an optimising criterion are identified and merged into a single cluster. This procedure is repeatedly applied until no further pair of clusters satisfying the optimising criterion exists. Examples of clustering algorithms can be found in [25, 46, 60, 61].

Without neglecting the usefulness of compile-time scheduling of virtual parallelism, it is worth pointing out that cases exist when static approaches are less effective. Such cases include lack of a priori knowledge of task sizes, uneven grain size of tasks, task graphs with structure dependent on the input data, etc., and can only be dealt with at run time, namely by using dynamic scheduling techniques. Dynamic scheduling (also called *load balancing* as it typically attempts to balance processor workload across the parallel machine) can be applied at application level—when the scheduler is part of the application, or at operating system level—when the scheduler is implemented as a function of the operating system. We are here concerned with application-level scheduling; for details on load balancing at operating system level see [13, 14, 32].

The most common technique in application-level dynamic scheduling is to organise the tasks as a FIFO list or as a pool of tasks; in each case, either one processor is employed as *task manager*, being in charge to dispatch tasks to the other processors (this is called *centralised scheduling*), or each processor has to access the task list or the task pool to get an unprocessed task whenever it becomes idle (*distributed scheduling*). Task migration among (neighbour) processors has also been considered as a way of improving the scheduling of the potential parallelism. As a common characteristic for all dynamic approaches, the overheads due to the delay of the scheduling decisions until run time could be significant, meaning that dynamic scheduling must be cautiously applied.

4.3 Previous work on portable parallel code derivation through scheduling

The overwhelming part of the scheduling techniques developed so far have been targeted at a parallel machine model incorporating details on the hardware of the computer on which the resulting parallel code is to be executed. As a consequence, these techniques yield architecture-specific parallel code. At the same time, most scheduling approaches have adopted an asynchronous parallel programming model, i.e., a programming model in which the processors operate asynchronously and synchronise pairwise through communication. The immediate effect of using such a model is that the resulting programs cannot be costed; the best one can do is to assess their effectiveness through experiments.

The reason that determined the parallel computing community to consistently avoid focusing on the design of scheduling techniques resulting in portable and costable parallel code has been a false belief that efficient such code is not achievable. However, the recent emergence of the bulk-synchronous parallel and related models made possible the development of parallel algorithms that can be implemented on any general purpose parallel computer, and whose cost can be accurately analysed [65, 47, 48, 8, 26, 10, 11]. As a result, the first attempts have been made to devise scheduling techniques for the automatic generation of portable and costable parallel code. The two such approaches we are aware of—both based on the BSP model—are briefly presented in the remainder of this section.

The former approach [48] tackles the scheduling of regular-structure two- and three-dimensional dags of computations in the context of triangular linear system solving and LU decomposition, respectively. A tiling of the two regular-structure dags into subdags isomorphic with the initial dags is proposed, followed by the distribution of the subdag computations among the p processors of the BSP computer. Both patterns of computation can be represented as (two-, respectively three-level) uniform-dependence loop nests, and scheduled using the basic technique presented in Section 5.4, which is a generalisation of the scheduling method proposed in [48]. A preliminary version of this generalisation is briefly presented in [12].

The second approach is due to Ding and Stefanescu [18]. This approach addresses the BSP scheduling of a very particular case of loop nest, namely of a perfect loop nest consisting of k outermost parallel loops and $K-k$ innermost sequential loops (where K is the depth of the loop nest), and having a reduction operation as the loop body. Two possible decompositions of the iteration space (i.e., two BSP schedules) are proposed in [18], block^k-serial^{K-k} decomposition, and block^K decomposition (the terminology is from [4]), and the one yielding the lowest cost is eventually preferred. Since both decompositions are chosen to be computationally optimal, the selection of the best schedule is actually done by taking into account the associated communication overheads. This scheduling method was tested against a couple of manually coded BSP programs on matrix multiplication, producing identical schedules.

5 Scheduling regular patterns of computation

It is a well-known fact that, excepting the I/O operations, an imperative program spends most of its execution time performing loop computations. As a consequence, the BSP scheduling techniques devised in this chapter are all targeted at the mapping of loop computations on a parallel computer. The chapter starts with the assessment of the communication costs incurred by the partitioning of the iteration space of a loop nest among the processors of a BSP computer. Then, in the remainder of the chapter, the BSP scheduling of various regular patterns of computation is described, and the efficiency of the resulting schedules is analysed in terms of the BSP cost model.

5.1 Computing the communication cost for a rectangular tile of an iteration space

One of the most important tasks in the analysis of a BSP schedule is the accurate assessment of its communication overhead. Indeed, the communication cost will be used throughout this chapter, either in choosing the best scheduling strategy for a given problem or in establishing the parameters of a certain scheduling technique.

Therefore, this section is entirely dedicated to the assessment of the communication overheads involved by several basic operations characteristic to the parallel execution of a loop nest on a distributed memory computer. In analysing the communication cost of these operations, we shall consider a generic K-level loop nest, $K \geq 1$; the j-th loop of the loop nest, $1 \leq j \leq K$, is indexed by i_j, and iterates from 0 to $n_j - 1$. The K-dimensional iteration space $\mathcal{I} = 0..(n_1 - 1) \times 0..(n_2 - 1) \times \cdots \times 0..(n_K - 1)$ is partitioned into rectangular tiles of size $x_1 \times x_2 \times \cdots \times x_K$. Each such tile T comprises the iteration points satisfying

$$c_j x_j \leq i_j \leq (c_j + 1)x_j - 1, \qquad 1 \leq j \leq K,$$

where $(c_1, c_2, \ldots, c_K)$ represent the coordinates of the tile in *tile space*, $0 \leq c_j < \lceil n_j / x_j \rceil$; some of the boundary tiles may be incomplete. We shall also assume that this partitioning is done in such a way that the tiles can be ordered (and computed) in a manner which is consistent with the data dependences of the loop, with typically more than one tile scheduled for execution at any time step. For each basic operation considered in this section, we shall evaluate the communication overhead required for the computation of a tile of the iteration space on a single processor of the parallel computer.

5.1.1 Communication overheads due to input data

Definition 1 Let $\mathcal{L}$ be a loop nest, and a an array referenced within the loop body of $\mathcal{L}$. Then, if during the execution of $\mathcal{L}$ the value of at least one element from a is read without being previously written, a is said to be an *input array* for $\mathcal{L}$. Furthermore, if no element of a is written as a result of executing $\mathcal{L}$, a is called a *pure-input array*; otherwise, a is an *input/output array.*

Any schedule of a loop nest on a distributed memory architecture must comprise an initialisation phase in which data corresponding to the input arrays of the loop is fetched from the appropriate locations, and stored in the local memory of the processors that will use it. In assessing the communication overhead due to this initialisation phase, we shall first introduce the notion of *trace* of an input array[4].

Definition 2 Let $\mathcal{L}$ be a loop nest, a an input array of $\mathcal{L}$, and T a tile of the iteration space of $\mathcal{L}$. Then, the *trace* of a across T (denoted $\mathrm{Tr}_T(a)$) represents the set of elements from a required for the computation of the loop body for the iteration points in T, and whose values have been assigned outside the loop nest.

It is not difficult to see that the trace of a pure-input array is exactly the set of array elements accessed during the computation of the considered tile. The situation is more complex for an input/output array, since some of the array values used within the loop body are not external, but are obtained while computing a previous tile, or even the considered tile. In the following, the trace size of an input array is assessed, first for a pure-input array, then for an input/output array.

The trace size of a pure-input array Consider a generic K-level loop nest $\mathcal{L}$, and let a be a single-referenced pure-input array of $\mathcal{L}$, with the array subscripts depending on exactly $k \leq K$ of the loop nest indices. Let also T be a rectangular tile of size $x_1 \times x_2 \times \cdots \times x_K$ of the iteration space of $\mathcal{L}$. Then, if the k loop indices that appear in the subscripts of a are $i_{j_1}, i_{j_2}, \ldots, i_{j_k}$,

$$\#\mathrm{Tr}_T(a) \leq \prod_{j \in \{j_1, j_2, \ldots, j_k\}} x_j. \tag{2}$$

Indeed, in the worst case, for any value taken by $(i_{j_1}, i_{j_2}, \ldots, i_{j_k})$ across T, a different element from a is used. When the subscripts of a are arbitrary expressions of the k loop indices, the trivial upper bound in (2) is the best we can do. However, in almost all practical programs the subscripts are linear combinations of the loop indices. A better upper bound can often be obtained in this case by using the rules provided by the following theorem.

Theorem 1 *Let $\mathcal{L}$ be a K-dimensional loop nest, a a single-referenced pure-input array of $\mathcal{L}$ whose subscripts are linear combinations of the loop nest indices, and S the set of subscripts of the array a. Let also the iteration space of $\mathcal{L}$ be partitioned into rectangular tiles of size $x_1 \times x_2 \times \cdots \times x_K$. Then, for any such tile T, the following rules can be used to compute an upper bound for $\#\mathrm{Tr}_T(a)$, the size of the trace of a across T:*

Rule 1: *If S contains any constant, then $\#\mathrm{Tr}_T(a) = \#\mathrm{Tr}_T(a')$, where a' is a pure-input array whose set of subscripts comprises only the non-constant subscripts from S.*

Rule 1': *Any constant term can be eliminated from a subscript without changing the value of the trace size.*

[4] a similar measure, called footprint, was introduced in [18] for pure-input arrays

Rule 2: *Let $s = s'$ be two subscripts from S; then, $\#\mathrm{Tr}_T(a) = \#\mathrm{Tr}_T(a')$, where a' is a pure-input array whose set of subscripts is $S \setminus \{s'\}$.*

Rule 3: *Let i_j be a loop index such that S comprises at least one term which depends only on i_j; then, $\#\mathrm{Tr}_T(a) = x_j\#\mathrm{Tr}_T(a')$, where a' is a pure-input array whose set of subscripts is obtained from S by eliminating all the subscripts depending only on i_j, and the terms depending on i_j from the other subscripts.*

Rule 3': *Let i_j be a loop index occurring in at least one subscript of a; then, $\#\mathrm{Tr}_T(a) \leq x_j\#\mathrm{Tr}_T(a')$, where a' is a pure-input array whose set of subscripts is obtained from S by eliminating the terms depending on i_j from all the subscripts.*

Rule 4: *Let $s = \alpha_1 i_{j_1} + \alpha_2 i_{j_2} + \cdots + \alpha_r i_{j_r}$, with $1 \leq j_1, j_2, \ldots, j_r \leq K$ and $\alpha_1, \alpha_2, \ldots, \alpha_r$ integer constants be a subscript of a, and $g = \gcd(\alpha_1, \alpha_2, \ldots, \alpha_r)$. Then, the subscript s can be replaced with $\alpha_1/g i_{j_1} + \alpha_2/g i_{j_2} + \cdots + \alpha_r/g i_{j_r}$ without changing the size of the trace of a.*

Rule 5: *Let $s = \alpha_1 i_{j_1} + \alpha_2 i_{j_2} + \cdots + \alpha_r i_{j_r}$, with $1 \leq j_1, j_2, \ldots, j_r \leq K$ and $\alpha_1, \alpha_2, \ldots \alpha_r$ integer constants be a subscript of a; then $\#\mathrm{Tr}_T(a) \leq (|\alpha_1| x_{j_1} + |\alpha_2| x_{j_2} + \cdots + |\alpha_r| x_{j_r})\#\mathrm{Tr}_T(a')$, where a' is a pure-input array whose set of subscripts is $S \setminus \{s\}$.*

Proof Let m be the dimensionality of the pure-input array a (i.e., $\#S = m$). For the purpose of this proof we shall view the elements of a as disposed in an m-dimensional Euclidean space, with the array subscripts giving the coordinates of the element required for the computation of the loop body at a certain iteration point.

Since a constant subscript has only the role of selecting the elements from a fixed $(m-1)$-dimensional hyperplane of the array, and the presence of a constant term in a subscript only introduces a displacement into that subscript, rules 1 and 1' are obviously correct. It is worth emphasising that any variable whose value does not change across the iteration space should also be considered a constant from the point of view of these rules.

If two subscripts are identical, they have the same value for any iteration point in T. Accordingly, the presence of a subset of identical elements among the subscripts of a brings the same contribution to its trace size as a single element of the subset; hence rule 2.

To prove the correctness of rule 3, consider a generic subscript $\alpha i_j \in S$, $1 \leq j \leq K$. Let us also consider one of the x_j values taken by i_j across T. Replacing this value in all the subscripts in S, and applying rules 1 and 1', we obtain that the number of elements from a referenced for this single value of i_j is exactly $\#\mathrm{Tr}_T(a')$, with a' defined as in rule 3. Since the subscript αi_j has a different value for any of the x_j choices of i_j, the $\#\mathrm{Tr}_T(a')$ elements of a are in a different $(m-1)$-dimensional hyperplane for each such choice, so $\#\mathrm{Tr}_T(a) = x_j\#\mathrm{Tr}_T(a')$. Rule 3' can be proved in a similar way: for each of the x_j values taken by an index i_j, $1 \leq j \leq K$ across T, $\#\mathrm{Tr}_T(a')$ elements from a are used within the loop body. However, in the general case the x_j sets of $\#\mathrm{Tr}_T(a')$ elements from a are not necessarily disjoint, so $\#\mathrm{Tr}_T(a) \leq x_j\#\mathrm{Tr}_T(a')$.

Rule 4 expresses the obvious fact that by multiplying a subscript with an arbitrary integer factor the number of different values taken by the resulting subscript across a rectangular tile does not change.

Finally, to prove rule 5, it is enough to show that a subscript $s = \alpha_1 i_{j_1} + \alpha_2 i_{j_2} + \cdots + \alpha_r i_{j_r}$ can take at most $|\alpha_1| x_{j_1} + |\alpha_2| x_{j_2} + \cdots + |\alpha_r| x_{j_r}$ distinct values across T. As T is a rectangular tile, for all the iteration points in T we have

$$c_j x_j \leq i_j \leq (c_j+1)x_j - 1, \quad 1 \leq j \leq K,$$

where $(c_1, c_2, \ldots, c_K)$ represent the coordinates of tile T. By multiplying the inequality corresponding to i_{j_l}, $1 \leq l \leq r$ with α_l, we obtain

$$\begin{array}{rcccll} |\alpha_l| c_{j_l} x_{j_l} & \leq & \alpha_j i_{j_l} & \leq & |\alpha_l| ((c_{j_l}+1)x_{j_l} - 1), & \text{if } \alpha_l > 0 \\ -|\alpha_l| ((c_{j_l}+1)x_{j_l} - 1) & \leq & \alpha_j i_{j_l} & \leq & -|\alpha_l| c_{j_l} x_{j_l}, & \text{if } \alpha_l < 0 \end{array}$$

Summing the above inequality for all the r indices that occur in s, we get:

$$\sum_{\substack{l=1\\ \alpha_l>0}}^{r} |\alpha_l| c_{j_l} x_{j_l} - \sum_{\substack{l=1\\ \alpha_l<0}}^{r} |\alpha_l| ((c_{j_l}+1)x_{j_l} - 1) \leq s \leq \sum_{\substack{l=1\\ \alpha_l>0}}^{r} |\alpha_l| ((c_{j_l}+1)x_{j_l} - 1) - \sum_{\substack{l=1\\ \alpha_l<0}}^{r} |\alpha_l| c_{j_l} x_{j_l}.$$

Consequently, the length of the interval in which s may take only integer values is

$$\sum_{\substack{l=1\\ \alpha_l>0}}^{r} |\alpha_l| ((c_{j_l}+1)x_{j_l} - 1) - \sum_{\substack{l=1\\ \alpha_l<0}}^{r} |\alpha_l| c_{j_l} x_{j_l} - \sum_{\substack{l=1\\ \alpha_l>0}}^{r} |\alpha_l| c_{j_l} x_{j_l} + \sum_{\substack{l=1\\ \alpha_l<0}}^{r} |\alpha_l| ((c_{j_l}+1)x_{j_l} - 1) + 1 =$$

$$= \sum_{l=1}^{r} |\alpha_l| (x_{j_l} - 1) + 1 \leq |\alpha_1| x_{j_1} + |\alpha_2| x_{j_2} + \cdots + |\alpha_r| x_{j_r},$$

so s can take at most $|\alpha_1| x_{j_1} + |\alpha_2| x_{j_2} + \cdots + |\alpha_r| x_{j_r}$ values, and rule 5 is correct. $\square$

Example 1 To illustrate the usage of the results provided by Theorem 1, consider the following example of a loop nest:

```
for i1=0,n1-1 do
  for i2=0,n2-1 do
    for i3=0,n3-1 do
      ...=...a[i1+i2+i3-1,i2+i3+1,0,i1]...
```

Assuming that a is a single-referenced pure-input array, the size of its trace across an $x_1 \times x_2 \times x_3$ rectangular tile T is

$$\begin{array}{ll} \#\mathrm{Tr}_T(a[i_1+i_2+i_3-1, i_2+i_3+1, 0, i_1]) = & \\ = \#\mathrm{Tr}_T(a^1[i_1+i_2+i_3-1, i_2+i_3+1, i_1]) & \text{(Rule 1)} \\ = \#\mathrm{Tr}_T(a^2[i_1+i_2+i_3, i_2+i_3, i_1]) & \text{(Rule 1}') \\ = x_1 \#\mathrm{Tr}_T(a^3[i_2+i_3, i_2+i_3]) & \text{(Rule 3)} \\ = x_1 \#\mathrm{Tr}_T(a^4[i_2+i_3]) & \text{(Rule 2)} \\ \leq x_1(x_2+x_3) & \text{(Rule 5)} \end{array}$$

When a pure-input array is referenced more than once within the loop body, computing an accurate upper bound for the size of its overall trace can be very difficult. Therefore, in many situations one has to conservatively assume that traces corresponding to different references are disjoint, and to compute the overall trace size as the sum of the individual trace sizes. Since even for very simple examples of multiple references such as $a[i_1]$ and $a[2i_1]$, or $a[2i_1]$ and $a[2i_1+1]$ the overall trace size is indeed equal to the sum of the individual trace sizes, this method of computing the overall size of a trace is appropriate in most cases.

Nevertheless, there is a particular case which occurs very often in practice, and therefore deserves a special consideration, namely the case when:

a. all the subscripts are linear combinations of the loop nest indices;
b. the correspondent subscripts from different references differ only in their constant terms;
c. the individual traces of the input array occupy contiguous rectangular blocks in the array.

Indeed, in this case the overlaps of individual traces can be identified, and their size subtracted from the above mentioned approximation of the overall trace size.

Let us first identify in which circumstances the trace of an array is a contiguous rectangular block. For this, let $s = \alpha_1 i_{j_1} + \alpha_2 i_{j_2} + \cdots + \alpha_r i_{j_r} + \beta$, $1 \leq j_l \leq K$, $1 \leq l \leq r$ be a generic subscript, and T a tile whose coordinates in the tile space are $(c_1, c_2, \ldots, c_K)$. As shown in the proof of Theorem 1, for the iteration points in T, s may take (integer) values in the interval

$$\left[\beta + \sum_{\substack{l=1\\ \alpha_l>0}}^{r} |\,\alpha_l\,|\, c_{j_l} x_{j_l} - \sum_{\substack{l=1\\ \alpha_l<0}}^{r} |\,\alpha_j\,|\,((c_{j_l}+1)x_{j_l}-1),\ \beta + \sum_{\substack{l=1\\ \alpha_l>0}}^{r} |\,\alpha_j\,|\,((c_{j_l}+1)x_{j_l}-1) - \sum_{\substack{l=1\\ \alpha_l<0}}^{r} |\,\alpha_l\,|\, c_{j_l} x_{j_l}\right]$$

It is immediate to see that a sufficient condition to have the subscript s taking each value in the interval, is to have $|\,\alpha_l\,| = 1$ for any $1 \leq l \leq r$. Clearly, having all the subscripts taking a contiguous set of values is not enough to ensure that an individual trace is a contiguous rectangular block, as proven for instance by the trace of $a[i_1, i_1+i_2]$. However, if in addition to the above condition any loop nest index appears in at most one subscript, the subscripts vary independently across the tile, and the resulting individual traces satisfy the desired requirement. When this requirement is fulfilled, the intersection of several individual traces can be exactly assessed, and the overall trace size computed as

$$\begin{aligned} \#\mathrm{Tr}_T(a) &= \# \bigcup_{k=1}^{q} \mathrm{Tr}_T^k(a) = \\ &= \sum_{k=1}^{q} \#\mathrm{Tr}_T^k(a) - \sum_{\{k_1,k_2\}\in\mathbb{P}1..q} \#\left(\mathrm{Tr}_T^{k_1}(a) \cap \mathrm{Tr}_T^{k_2}(a)\right) + \cdots + (-1)^{q-1} \# \bigcap_{k=1}^{q} \mathrm{Tr}_T^k(a) \end{aligned} \tag{3}$$

where a is a pure-input array referenced q times within the loop body, and $\mathrm{Tr}_T^k(a)$ represents the individual trace for the k-th reference to array a, $1 \leq k \leq q$. We are now ready to give the following theorem.

Theorem 2 *Let $\mathcal{L}$ be a K-level loop nest, and $a[s_1+\beta_1^k, s_2+\beta_2^k, \ldots, s_m+\beta_m^k]$, $1 \le k \le q$ be the q references of a pure-input array a of $\mathcal{L}$, with:*

- β_i^k, $1 \le i \le m$, $1 \le k \le q$ *integer constants;*
- $(s_1 \;\; s_2 \;\; \ldots \;\; s_m)^{\mathrm{T}} = \mathsf{A}\ (i_1 \;\; i_2 \;\; \ldots \;\; i_K)^{\mathrm{T}}$, *where* $\mathsf{A} = (\alpha_{ij})_{1\le j\le K}^{1\le i\le m}$ *is an $m \times K$ matrix with elements from the set $\{-1, 0, 1\}$;*
- *for any $1 \le j \le K$, $\sum_{i=1}^{m} \mid \alpha_{ij} \mid\le 1$.*

Let also T be a rectangular tile of size $x_1 \times x_2 \times \cdots \times x_K$ of the iteration space of $\mathcal{L}$, $(c_1, c_2, \ldots, c_K)$ be the coordinates of T in the tile space, and $\mathrm{Tr}_T^{k_1}(a)$, $\mathrm{Tr}_T^{k_2}(a)$, $\ldots$, $\mathrm{Tr}_T^{k_r}(a)$, $1 \le k_1, k_2, \ldots, k_r \le q$ be the traces of $r \ge 1$ arbitrary references of a. Then,

$$\# \bigcap_{x=1}^{r} \mathrm{Tr}_T^{k_x}(a) = \begin{cases} \prod_{i=1}^{m} \big(\min\limits_{1\le x\le r} L_i^{k_x} - \max\limits_{1\le x\le r} l_i^{k_x} + 1\big), & \textit{if } \forall i : 1..m \bullet \min\limits_{1\le x\le r} L_i^{k_x} \ge \max\limits_{1\le x\le r} l_i^{k_x} \\ 0, & \textit{otherwise} \end{cases} \tag{4}$$

where

$$l_i^k = \beta_i^k + \sum_{\substack{j=1 \\ \alpha_{ij}=1}}^{K} c_j x_j - \sum_{\substack{j=1 \\ \alpha_{ij}=-1}}^{K} ((c_j+1)x_j - 1) \;\; \textit{and}$$

$$L_i^k = \beta_i^k + \sum_{\substack{j=1 \\ \alpha_{ij}=1}}^{K} ((c_j+1)x_j - 1) - \sum_{\substack{j=1 \\ \alpha_{ij}=-1}}^{K} c_j x_j$$

for any $1 \le i \le m$, $1 \le k \le q$.

Proof According to the analysis of the circumstances in which the trace of an input array is a contiguous rectangular block, all the q references of the array a in the theorem correspond to such a block. Moreover, the i-th subscript from the k-th reference of a (i.e., $s_i+\beta_i^k$) takes all the integer values in the interval $[l_i^k,\ L_i^k]$, and does so independently of all the other subscripts. Consequently, the k-th individual trace of a (i.e., $\mathrm{Tr}_T^k(a)$) is a rectangular block of a defined by the subscript limits $l_1^k..L_1^k \times l_2^k..L_2^k \times \cdots \times l_m^k..L_m^k$. The intersection of $r \ge 1$ such arbitrary blocks $\mathrm{Tr}_T^{k_1}(a)$, $\mathrm{Tr}_T^{k_2}(a)$, $\ldots$, $\mathrm{Tr}_T^{k_r}(a)$, $1 \le k_1, k_2, \ldots, k_r \le q$ is obviously the block

$$\max_{1\le x\le r} l_1^{k_x} .. \min_{1\le x\le r} L_1^{k_x} \times \max_{1\le x\le r} l_2^{k_x} .. \min_{1\le x\le r} L_2^{k_x} \times \cdots \times \max_{1\le x\le r} l_m^{k_x} .. \min_{1\le x\le r} L_m^{k_x},$$

if $\forall i : 1..m \bullet \min\limits_{1\le x\le r} L_i^{k_x} \ge \max\limits_{1\le x\le r} l_i^{k_x}$, and is the empty set otherwise. Since equation (4) gives exactly the size of the above described intersection, the proof is complete.
□

Example 2 Consider for instance a 3-level loop nest $\mathcal{L}$, and a pure-input array a of $\mathcal{L}$ whose references within the loop body are $a[i_1+3, i_2-i_3-1]$ and $a[i_1, i_2-i_3+4]$. Then, the trace size of a across an $x_1 \times x_2 \times x_3$ rectangular tile T is

$$\#\mathrm{Tr}_T(a) = \#\mathrm{Tr}_T^1(a) + \#\mathrm{Tr}_T^2(a) - \#\left(\mathrm{Tr}_T^1(a) \cap \mathrm{Tr}_T^2(a)\right),$$

where $\mathrm{Tr}_T^1(a)$ and $\mathrm{Tr}_T^2(a)$ are the traces of a corresponding to the two references. As the assumptions in Theorem 2 are satisfied, the overall trace size can be computed as indicated in the theorem. For a tile T whose coordinates in the tile space are (c_1, c_2, c_3), we obtain

$$\begin{array}{ll} l_1^1 = 3 + c_1 x_1, & L_1^1 = 2 + (c_1+1)x_1; \\ l_2^1 = c_2 x_2 - (c_3+1)x_3, & L_2^1 = -2 + (c_2+1)x_2 - c_3 x_3; \\ l_1^2 = c_1 x_1, & L_1^2 = -1 + (c_1+1)x_1; \\ l_2^2 = 5 + c_2 x_2 - (c_3+1)x_3, & L_2^2 = 3 + (c_2+1)x_2 - c_3 x_3 \end{array}$$

Accordingly,

$$\begin{array}{l} \#\mathrm{Tr}_T^1(a) = (L_1^1 - l_1^1 + 1)(L_2^1 - l_2^1 + 1) = x_1(x_2 + x_3 - 1), \\ \#\mathrm{Tr}_T^2(a) = (L_1^2 - l_1^2 + 1)(L_2^2 - l_2^2 + 1) = x_1(x_2 + x_3 - 1), \quad \text{and} \\ \#\left(\mathrm{Tr}_T^1(a) \cap \mathrm{Tr}_T^2(a)\right) = (\min\{L_1^1, L_1^2\} - \max\{l_1^1, l_1^2\})(\min\{L_2^1, L_2^2\} - \max\{l_2^1, l_2^2\}) = \\ = (x_1 - 3)(x_2 + x_3 - 6), \end{array}$$

so

$$\#\mathrm{Tr}_T(a) = (x_1 + 3)(x_2 + x_3 + 4) - 30.$$

The trace size of an input/output array So far we only discussed the communication overheads induced by pure-input arrays. In the remainder of this section, we shall concentrate on the other class of loop-external communication costs, namely on the communication costs due to input/output arrays.

One can distinguish several cases of input/output references to an array. First, as an extreme case, the input and output references to the input/output array can be totally independent, as illustrated by the following example of a loop body:

```
a[2*i₁,i₂,i₃]=f₁(...)
b[i₁,i₂,i₃]=f₂(a[2*i₁+1,i₂,i₃])
```

Obviously, the contribution of the input/output array to the total communication cost must be computed in this case as though the array were a pure-input array.

Second, the input and output references to the same array can be related through a loop-independent data dependence. Two subcases can be identified here: when the data dependence between the statements containing the two references is a true dependence— as in the following example of a loop body

```
a[i₁,i₂,i₃]=...
b[i₁,i₂,i₃]=f(a[i₁,i₂,i₃])
```

and when this dependence is an antidependence:

```
b[i₁,i₂,i₃]=f(a[i₁,i₂,i₃])
a[i₁,i₂,i₃]=...
```

It is immediate to see that in the first subcase all read references to an array element use values computed by a previous statement in the loop body, whereas in the second subcase all read references address values from outside the loop body. Consequently, the input/output array adds no contribution to the communication cost in the former subcase, and a contribution equivalent to that corresponding to a pure-input array in the latter subcase.

Finally, the input and output references to the same array can be related through a loop-carried dependence. Again, one can distinguish two subcases, namely when the loop-carried dependence is a true dependence—as in the following example

```
a[i₁,i₂,i₃]=f(a[i₁,i₂-1,i₃],...)
```

and when the loop-carried dependence is an antidependence. Here is an example for the second subcase:

```
a[i₁,i₂,i₃]=f(a[i₁,i₂+1,i₃],...)
```

Since for the antidependence subcase any usage of an element of the input/output array refers a value not previously modified within the loop, the contribution of such a reference to the amount of communication is identical with the contribution of a similar reference to a pure-input array. As for the subcase of a true dependence, an upper bound for this amount is given by the following theorem.

Theorem 3 *Let a be an input/output array of a K-level loop nest $\mathcal{L}$, and assume that a induces a loop-carried true dependence encoded by the distance vector $d = (d_1, d_2, \ldots, d_K)$. Let also T be a rectangular tile of size $x_1 \times x_2 \times \cdots \times x_K$ of the iteration space of $\mathcal{L}$. Then, if $x_j \geq \mid d_j \mid$ for any $1 \leq j \leq K$,*

$$\#\mathrm{Tr}_T(a) \leq \prod_{j=1}^{K} x_j - \prod_{j=1}^{K} (x_j - \mid d_j \mid). \tag{5}$$

Proof Let $(c_1, c_2, \ldots, c_K)$ be the coordinates of T in the tile space. Then, if $(i_1, i_2, \ldots, i_K)$ is a generic iteration point in T,

$$c_j x_j \leq i_j \leq (c_j + 1)x_j - 1, \quad 1 \leq j \leq K,$$

and the computation of the loop body for this iteration point uses the element of a corresponding to the subscripts given by $i_1, i_2, \ldots, i_K$. According to the hypothesis of the theorem, if $(i_1 - d_1, i_2 - d_2, \ldots, i_K - d_K)$ is an iteration point from T, this element from a is modified in a previous iteration also belonging to T, and the reference to the input/output array implies no use of loop-external data. The problem now is to compute the maximum number of elements from a which are needed for the computation of the iteration points in T without being previously modified during the computation of T. To determine this quantity, we will consider the worst case, i.e., the case when a different element from a is used for any value taken by $(i_1, i_2, \ldots, i_K)$ across T. Then,

$$\begin{aligned}\#\mathrm{Tr}_T(a) \leq \#(&\{(i_1, i_2, \ldots, i_K) \mid \forall j : 1..K \bullet c_j x_j \leq i_j \leq (c_j + 1)x_j - 1\} \setminus \\ &\{(i_1, i_2, \ldots, i_K) \mid \forall j : 1..K \bullet c_j x_j \leq i_j, i_j - d_j \leq (c_j + 1)x_j - 1\}).\end{aligned}$$

Since the second set is a subset of the first one, we obtain

$$\#\mathrm{Tr}_T(a) \leq \#\{(i_1, i_2, \ldots, i_K) \mid \forall j : 1..K \bullet c_j x_j \leq i_j \leq (c_j+1)x_j - 1\} -$$
$$\#\{(i_1, i_2, \ldots, i_K) \mid \forall j : 1..K \bullet c_j x_j \leq i_j, i_j - d_j \leq (c_j + 1)x_j - 1\} =$$
$$= \prod_{j=1}^{K} x_j - \prod_{j=1}^{K} (x_j - \mid d_j \mid).$$

□

Example 3 To illustrate the application of the result in Theorem 3, consider the following loop nest:

```
          for i1=0,n1-1 do
            for i2=0,n2-1 do
              for i3=0,n3-1 do
S1:             a[i1,i2,i3]=...
S2:             ...=...a[i1,i2-1,i3+2]...
```

A flow data dependence S1δS2 encoded by the distance vector $d = (0, 1, -2)$ due to the common usage of the input/output array a exists between the two statements of the loop body. Accordingly, the trace size of array a across an $x_1 \times x_2 \times x_3$ rectangular tile T is

$$\mathrm{Tr}_T(a) \leq x_1 x_2 x_3 - x_1(x_2 - 1)(x_3 - 2) = x_1(2x_2 + x_3 - 2).$$

5.1.2 Communication overheads due to loop-carried dependences

Two types of external data are required for the computation of a tile and may need to be satisfied through inter-processor communication: data whose values are assigned outside the loop nest, and data whose values are assigned by preceding tile computations. In the previous section, we examined the communication overheads incurred by the first type of external data. As for the communication costs due to the loop-carried data dependences, they are assessed in this section.

Once more, it is impossible to obtain a non-trivial value (or upper bound) for the communication overhead in the general case. Therefore, taking into account that in this chapter we are dealing with regular patterns of computation, we will only consider the case when all the data dependences in the loop nest are uniform and can be represented by distance vectors. Under these circumstances, the amount of data to be sent by a processor after computing a tile of the iteration space is given by the following theorem.

Theorem 4 *Let $\mathcal{L}$ be a K-level loop nest and D the set of distance vectors encoding the loop-carried data dependences of $\mathcal{L}$. Then, the amount of data to be sent by a processor after computing a rectangular tile of size $x_1 \times x_2 \times \cdots \times x_K$ of the iteration space of $\mathcal{L}$, with $x_j > d_j$ for any $1 \leq j \leq K$ and $d \in D$, is at most*

$$\sum_{d \in D'} \left[\prod_{j=1}^{K} x_j - \prod_{j=1}^{K} (x_j - \mid d_j \mid) \right], \tag{6}$$

where $D' \subseteq D$ is the set of distance vectors standing for flow data dependences.

Proof We will first show that non-flow data dependences involve no inter-processor communication. There are two types of non-flow data dependences, namely antidependences and output dependences. Since the former type stands for dependences in which the value of a variable is first used, then modified during a subsequent iteration, the value that is used comes from outside the loop nest, so the antidependence itself involves no data movement from one iteration to another. As for an output dependence, it stands for the modification of the same variable in two different iterations; again, no data needs to be passed from one iteration to another, so no communication overhead is involved. However, it is worth mentioning that both types of non-flow data dependences make possible the existence of non-coherent copies of the same variable in different local memories. Of course, this is not a problem as long as the out-of-date copies are known to be so.

We will now consider a single distance vector $d \in D$ that stands for a flow data dependence, and a rectangular tile T of the iteration space, and we will compute $Comm(d)$, the amount of data that must be sent after computing the tile due to the data dependences encoded by d. Similarly to the situation in Theorem 3, this amount of data is upper bounded by the number of iteration points in T for which the data dependence due to d points to an iteration point outside T. If the coordinates of T in the tile space are $(c_1, c_2, \ldots, c_K)$, this result can be expressed formally as

$$\begin{aligned}
&Comm(d) \leq \#\{i \mid i \in T \wedge i + d \notin T\} = \\
&\quad = \#\{(i_1, i_2, \ldots, i_K) \mid (\forall j : 1..K \bullet i_j \in [c_j x_j, (c_j+1)x_j - 1]) \wedge \\
&\qquad\qquad (\exists j : 1..K \bullet i_j + d_j \notin [c_j x_j, (c_j+1)x_j - 1])\} = \\
&\quad = \#(\{(i_1, i_2, \ldots, i_K) \mid \forall j : 1..K \bullet c_j x_j \leq i_j \leq (c_j+1)x_j - 1\} \setminus \\
&\qquad \{(i_1, i_2, \ldots, i_K) \mid \forall j : 1..K \bullet c_j x_j \leq i_j, i_j + d_j \leq (c_j+1)x_j - 1\}) = \\
&\quad = \#\{(i_1, i_2, \ldots, i_K) \mid \forall j : 1..K \bullet c_j x_j \leq i_j \leq (c_j+1)x_j - 1\} - \\
&\qquad \#\{(i_1, i_2, \ldots, i_K) \mid \forall j : 1..K \bullet c_j x_j \leq i_j, i_j + d_j \leq (c_j+1)x_j - 1\} \\
&\quad = \prod_{j=1}^{K} x_j - \prod_{j=1}^{K} (x_j - \mid d_j \mid).
\end{aligned}$$

As in the worst case the communication overheads due to different flow-dependence distance vectors are totally independent, we obtain the upper bound in (6) by summing this quantity over the whole set of flow-dependence distance vectors.

□

Example 4 To illustrate the usage of this theorem, consider the following 2-level loop nest:

```
        for i₁=0,n₁-1 do
          for i₂=0,n₂-1 do
S1:         a[i₁,i₂]=f₁(b[i₁-1,i₂+1])
S2:         b[i₁,i₂]=f₂(a[i₁,i₂-2])
```

```
forall i₁=0,n₁-1 do in parallel
  forall i₂=0,n₂-1 do in parallel
    ..................
      forall i_K=0,n_K-1 do in parallel
        loop_body
```

Figure 3: A fully parallel loop nest.

It is immediate to see that the set of flow-dependence distance vectors of this loop nest is $D' = D = \{(1,-1),(0,2)\}$. Consequently, the amount of data to be sent by a processor after computing an $x_1 \times x_2$ rectangular tile T of the iteration space of this loop is

$$[x_1x_2 - (x_1-1)(x_2-1)] + [x_1x_2 - x_1(x_2-2)] = 3x_1 + x_2 - 1.$$

5.2 Scheduling fully parallel loop nests

Fully parallel loop nests (Figure 3) represent the loop structure most suitable for parallelisation. Indeed, since all the iterations of the loop are independent, its iteration space can be partitioned into p totally independent tiles of equal volume, and executed in a single computation superstep on a p-processor BSP computer. This computation superstep must be preceded by an "initialisation" superstep in which external data required for the tile computations is fetched from the appropriate local memories.

Any partitioning of the iteration space into equally sized tiles yields a perfect load balancing of the computation. However, we will only consider rectangular tiles of size $x_1 \times x_2 \times \cdots \times x_K$, $\prod_{j=1}^{K} x_j = (\prod_{j=1}^{K} n_j)/p$ because this partitioning induces a very simple parallel schedule (Figure 4), which avoids costly index manipulations. The cost of this BSP schedule is

$$2L + \frac{\prod_{j=1}^{K} n_j c}{p} + gComm, \tag{7}$$

where c is the cost of computing the loop body for a single point of the iteration space, and $Comm$ is the maximum amount of data fetched by or from a processor during the initialisation superstep. Since the schedule is computationally-optimal, and the synchronisation cost is also optimal, the size of the tiles must be chosen such as to minimise $Comm$.

To compute $Comm$ for the schedule in Figure 4, we will assume an uniform distribution of data prior to executing the loop[5]. Accordingly, the amount of data exchanged by any processor during the first superstep can be approximated to the amount of data fetched by that processor.

Techniques for the computation of an upper bound for the amount of input data to be fetched by a processor before computing a rectangular tile of an iteration space are

[5]since input data is often uniformly distributed after the initialisation phase of a parallel program, and the execution of the BSP schedules devised in this report always leaves the data uniformly distributed, this is a justified assumption

```
forall t₁=0,n₁/x₁-1 do in parallel
  forall t₂=0,n₂/x₂-1 do in parallel
    .................
      forall t_K=0,n_K/x_K-1 do in parallel
        Processor t₁*n₂/x₂*...*n_K/x_K+t₂*n₃/x₃*...n_K/x_K+...+t_K:
          BSP_start_superstep
            fetch external data for the computation of tile T_{t₁,t₂,...,t_K}
          BSP_end_superstep
          BSP_start_superstep
            for i₁=t₁*x₁,(t₁+1)*x₁-1 do
              for i₂=t₂*x₂,(t₂+1)*x₂-1 do
                ..............
                  for i_K=t_K*x_K,(t_K+1)*x_K-1 do
                    loop_body
          BSP_end_superstep
```

Figure 4: The BSP schedule of a fully parallel loop nest. Each processor is assigned the computation of a rectangular tile of size $x_1 \times x_2 \times \cdots \times x_K$, $\prod_{j=1}^{K} x_j = (\prod_{j=1}^{K} n_j)/p$; some boundary tiles may be incomplete.

presented in Section 5.1.1. These techniques can now be used to compute an upper bound for $Comm$ as a polynomial of degree at most K in the variables x_1, x_2, ... , x_K. Therefore, the optimal schedule corresponds to the values of x_1, x_2, ... , x_K which minimise $Comm(x_1, x_2, \ldots, x_K)$, subject to the load balancing constraint $\prod_{j=1}^{K} x_j = (\prod_{j=1}^{K} n_j)/p$.

One method to find the minimum of the scalar field $Comm = Comm(x_1, x_2, \ldots, x_K)$, under the (load balancing) constraint $g(x_1, x_2, \ldots, x_K) = \prod_{j=1}^{K} x_j - (\prod_{j=1}^{K} n_j)/p = 0$, $1 \le x_j \le n_j$, $1 \le j \le K$ is [68] to apply the Lagrange multiplier rule, which specifies that the constrained minima must be among the solutions of

$$\bigtriangledown Comm(x_1, x_2, \ldots, x_K) = \lambda \bigtriangledown g(x_1, x_2, \ldots, x_K)$$

with $\lambda \in \mathbb{R}$. The mean inequality has also proved to be useful in many of our tests.

Example 5 Consider for instance the following example of a fully parallel loop nest:

```
forall i₁=0,n₁-1 do in parallel
  forall i₂=0,n₂-1 do in parallel
    forall i₃=0,n₃-1 do in parallel
      a[i₁,i₂,i₃]=f(b[i₁,i₂+i₃],c[i₁,i₂])
```

Assume that the BSP schedule in Figure 4 is used to map this loop nest onto a BSP computer. Then, according to the results in Section 5.1.1, the communication cost corresponding to the resulting BSP schedule is

$$Comm(x_1, x_2, x_3) = g\left(\mathrm{Tr}_T(b) + \mathrm{Tr}_T(c)\right) = g\left(x_1(x_2 + x_3) + x_1 x_2\right),$$

```
for i₁=0,n₁-1 do
  for i₂=0,n₂-1 do
    .............
      for iK=0,nK-1 do
        s=s⊕f(i₁,i₂,...,iK)
```

Figure 5: A reduction loop nest; $\oplus$ stands for an associative and commutative operator (or only associative if $K = 1$).

with T a rectangular tile of size $x_1 \times x_2 \times x_3$, $x_1x_2x_3 = n_1n_2n_3/p$. Since $x_1x_2 \geq n_1n_2/p$, and $x_1x_3 \geq n_1n_3/p$, this communication overhead is minimised when $(x_1, x_2, x_3) = (n_1/p, n_2, n_3)$. This choice of the tile size yields a BSP schedule whose total cost is

$$2L + \frac{n_1n_2n_3c}{p} + g\frac{n_1(2n_2 + n_3)}{p},$$

where c is the cost of computing the loop body for a single point of the iteration space.

5.3 Reduction scheduling

A K-level reduction loop nest is a perfect loop nest comprising K loops, and whose loop body consists of a *reduction* statement; the general pattern of a reduction loop nest is depicted in Figure 5.

Since the loop body is a reduction operation, the iterations can be executed in any convenient order without affecting the final result. Moreover, it is possible to partition the iteration space of the loop into tiles, to execute the reduction statement for the iteration points in these tiles independently (starting each time with a local $s = 1_\oplus$, where $1_\oplus$ is the identity element for $\oplus$), and, finally, to obtain the global result by combining the partial results using the reduction operator $\oplus$.

Consequently, the scheduling of a reduction loop nest onto a bulk-synchronous parallel computer is very similar to that of a fully parallel loop nest. Thus, in a first phase, each processor is assigned the computation of a partial reduction over a rectangular tile of size $x_1 \times x_2 \times \cdots \times x_K$, $\prod_{j=1}^{k} x_j = (\prod_{j=1}^{k} n_j)/p$ of the iteration space. As in the case of a fully parallel loop nest, this phase takes two supersteps, an initialisation superstep in which the processors fetch external data, and a pure-computation superstep, in which the partial reductions are actually performed. Similarly, the values chosen for x_1, x_2, ..., x_K must be those which minimise the amount of data fetched for the computation of the tiles.

Once the first phase is completed, the partial results are combined using a variant of the standard broadcasting algorithm introduced in [65]. The p processors are organised in a logical d-ary tree, and in each superstep t, $0 \leq t < \log_d p$ of the algorithm, the processors at distance t from the leaves send their data to the parent, which combines this data in the following superstep. The global result is computed by the root processor in superstep $t = \log_d p$. To minimise the cost

$$(\log_d p\ + 1)L + \log_d p\ gd + \log_d p\ d \approx \log_d p(L + d(g + 1))$$

```
for i1=0,n-1 do
  for i2=0,n-1 do
    ..............
      for iK=0,n-1 do
        loop_body
```

Figure 6: A uniform-dependence loop nest: the data dependences of the loop are representable as distance vectors.

of this phase, we choose $d = L/(g+1)$. Accordingly, the cost of the entire schedule is

$$2L(1 + \log_{L/(g+1)} p) + \frac{\prod_{j=1}^{K} n_j c}{p} + gComm(x_1, x_2, \ldots, x_K), \tag{8}$$

where c is the cost of computing the reduction statement for a single point of the iteration space, and $Comm(x_1, x_2, \ldots, x_K)$ is the maximum amount of data fetched by any processor during the initialisation superstep.

5.4 Scheduling uniform-dependence iteration spaces

One of the loop structures which is very often used to express regular patterns of computation in imperative programs is the *uniform-dependence loop nest.* A K-level uniform-dependence loop nest ($K > 1$) is a tightly-nested loop comprising K loops, and whose data dependences are representable as distance vectors (Figure 6).

In this section we will devise a BSP scheduling technique for uniform-dependence loop nests that respect certain constraints, then we will discuss the modifications required for the scheduling of generic such loops onto a general purpose parallel computer. Thus, we will first assume that all the distance vectors of the loop have only non-negative components (i.e., that we are dealing with a *fully permutable loop nest* as defined in [71]). If the loop nest to be scheduled does not respect this constraint, an affine transformation applied to its iteration space can bring it to the desired form; several ways of finding an appropriate affine transformation for this are presented at the end of the section. We will also assume that the set of distance vectors span the entire iteration space of the loop. This is not a restrictive assumption, but an assumption which puts aside a simpler, particular case of uniform-dependence loop nest. Indeed, when the distance vectors do not span the entire iteration space, the iteration space can be partitioned into totally independent tiles and the scheduling becomes trivial—this particular case is discussed in section 5.8.2. Finally, this time for the sake of simplicity, we will consider that all the loops in the loop nest iterate from 0 to $n-1$, where n is the problem size; it is worth emphasising that no major change in the BSP schedule presented in this section is required when the loop bounds are different.

To derive a p-processor BSP program for the computation of a uniform-dependence loop nest which respects the above assumptions, the iteration space of the loop is first partitioned into hypercubic tiles of size $(n/p^{1/(K-1)}) \times (n/p^{1/(K-1)}) \times \cdots \times (n/p^{1/(K-1)})$.

A generic tile T comprises the iteration points $(i_1, i_2, \ldots, i_K)$ given by

$$c_j \frac{n}{p^{1/(K-1)}} \leq i_j < (c_j + 1) \frac{n}{p^{1/(K-1)}}, \quad 1 \leq j \leq K,$$

where $(c_1, c_2, \ldots, c_K)$ are the coordinates of T in the *tile space*, $0 \leq c_1, c_2, \ldots, c_K < p^{1/(K-1)}$. It is worth noticing here that, since the computation of any tile will be assigned to a single processor, most data dependences are *internalised* by this partition, and only a few of them will have to be satisfied through inter-processor communication.

The resulting tiles are then scheduled for parallel execution using the hyperplane method of Lamport [40]. This widely applied method (also known as *wavefront* scheduling [4, 72]) consists of successively scheduling for concurrent execution the intersections of the iteration space (the $(0..p^{1/(K-1)} - 1)^K$ tile space in our case) with a family of $(K-1)$-dimensional parallel hyperplanes. The number of hyperplanes in this family gives the number of *supersteps* required to accomplish the computation (i.e., the *schedule length*). It is therefore necessary to find a family of hyperplanes that satisfies the following two conditions:

(C.1) the successive intersections generate a schedule which respects the data dependences of the loop;

(C.2) the number of supersteps required by the resulting schedule is minimum among all schedules that satisfy condition (C.1).

Theorem 5 *Let $\mathcal{L}$ be a K-level fully permutable loop nest whose iteration space is partitioned into hypercubic tiles of size $\left(n/p^{1/(K-1)}\right)^K$, $K > 1$. Assume that the tile size is large enough (i.e., larger than the distance vector sizes) for the dependences not internalised by the tiling to occur only between neighbour tiles, and for any tile $T_{c_1,c_2,\ldots,c_K}$ to depend (at least) on each of the tiles $T_{c_1-1,c_2,\ldots,c_K}$, $T_{c_1,c_2-1,\ldots,c_K}$, $\ldots$, $T_{c_1,c_2,\ldots,c_K-1}$. Then, the set of $(K-1)$-dimensional hyperplanes given by*

$$\left\{ c_1 + c_2 + \cdots + c_K = t \;\middle|\; 0 \leq t \leq Kp^{1/(K-1)} - K \right\} \tag{9}$$

defines a minimum-length legal schedule for $\mathcal{L}$.

Proof It is a well-known result (see for instance [35]) that when a family of hyperplanes $\alpha_1 i_1 + \alpha_2 i_2 + \cdots + \alpha_K i_K = t$, $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_K) : \mathbb{N}^K$ is used for the (linear) scheduling of a size x^K hypercubic iteration space whose distance vector set is D, the schedule is legal if and only if $\forall d \in D \bullet \alpha \cdot d > 0$, and requires a number of $(\alpha_1 + \alpha_2 + \cdots + \alpha_K)x + 1$ supersteps. In our case, D includes the distance vectors $(1, 0, \ldots, 0)$, $(0, 1, \ldots, 0), \ldots, (0, 0, \ldots, 1)$ (and possibly other distance vectors $d : \{0, 1\}^K$), so the schedule is correct if and only if $\alpha_j \geq 1$, $1 \leq j \leq K$. Therefore, the schedule in (9), which corresponds to $\alpha_j = 1$, $1 \leq j \leq K$, is legal and has minimum length.
□

To detail the p-processor BSP schedule of a K-level fully permutable loop nest, we will consider the p processors of the parallel computer indexed from 0 to $p-1$. The BSP schedule requires $Kp^{1/(K-1)} - K + 1$ supersteps, with the following operations taking place in each superstep t, $0 \leq t \leq Kp^{1/(K-1)} - K$:

```
x=p^(1/(K-1))
for t=0,K*x-K do
  for all c1,c2,...,cK=0,x-1 such that c1+c2+...+cK=t do in parallel
    Processor c1*x^(K-2)+c2*x^(K-3)+...+cK-1:
      BSP_begin_superstep
        compute tile Tc1,c2,...,cK:
          for i1=c1*x,(c1+1)*x-1 do
            for i2=c2*x,(c2+1)*x-1 do
              ................
                for iK=cK*x,(cK+1)*x-1 do
                  loop_body
        send data required by subsequent tile computations
      BSP_end_superstep
```

Figure 7: The pseudocode for the p-processor BSP schedule of a K-level fully permutable loop nest.

(1) each tile $T_{c_1,c_2,\ldots,c_K}$ with $c_1+c_2+\cdots+c_K=t$ is computed by processor $c_1p^{(K-2)/(K-1)}+c_2p^{(K-3)/(K-1)}+\cdots+c_{K-1}$;

(2) data computed in the current superstep and required by subsequent tile computations is sent to the appropriate processor(s).

The pseudocode for this schedule is presented in Figure 7. Since the tile coordinates satisfy $0 \le c_1, c_2, \ldots, c_K < p^{1/(K-1)}$, any tile is scheduled for execution on one of the p processors (i.e., $0 \le c_1p^{(K-2)/(K-1)} + c_2p^{(K-3)/(K-1)} + \cdots + c_{K-1} < p$). This approach is equivalent to arranging the p processors in a $(K-1)$-dimensional array of size $p^{1/(K-1)} \times p^{1/(K-1)} \times \cdots \times p^{1/(K-1)}$, with processor $(c_1, c_2, \ldots, c_{K-1})$ being assigned the computation of tiles $T_{c_1,c_2,\ldots,c_K}$, $0 \le c_K < p^{1/(K-1)}$. Moreover, the equation $c_1 + c_2 + \cdots + c_K = t$ has at most one solution when c_1, c_2, ... , c_{K-1} are fixed, so any processor is assigned at most one tile per superstep.

In order to evaluate the cost of a superstep of the BSP schedule, we need to know the amount of data sent/received by a processor during a superstep. This result is provided by the following theorem.

Theorem 6 *Let $\mathcal{L}$ be a K-level fully permutable loop nest whose loops iterate from 0 to $n-1$, and D the set of distance vectors encoding the flow data dependences of $\mathcal{L}$. Then, the maximum amount of data sent or received by a processor in any superstep of a p-processor BSP schedule of $\mathcal{L}$ is given by*

$$\begin{aligned} Comm &= \sum_{d\in D}\left[\left(\frac{n}{p^{1/(K-1)}}\right)^K - \frac{n}{p^{1/(K-1)}}\prod_{j=1}^{K-1}\left(\frac{n}{p^{1/(K-1)}} - d_j\right)\right] = \\ &= \frac{n^{K-1}}{p}\left(\sum_{d\in D}\sum_{j=1}^{K-1} d_j + \mathrm{o}(1)\right). \end{aligned} \tag{10}$$

Proof The first part of the theorem is a corollary of Theorem 4. Indeed, all we need to do to prove that $Comm$ is an upper bound for the amount of data sent by a processor after computing a tile is to make $x_1 = x_2 = \cdots = x_K = n/p^{1/(K-1)}$ in equation (6), and to take into account the fact that the data dependences along the K-th direction of the iteration space are internalised by our schedule, and $d_j \geq 0$ for any $1 \leq j \leq K$.

For the second part of the theorem, we will consider the worst case, i.e., the case of a processor P_x which is assigned a tile in each of the following K supersteps that may depend on data sent during the current superstep t (we still assume only neighbour dependences among tiles). Let us compute the maximum amount of data P_x may receive during superstep t due to a single distance vector $d = (d_1, d_2, \ldots, d_K) \in D$. For this, we will consider all the iteration points in the K isomorphic tiles assigned to P_x during the supersteps $t+1, t+2, \ldots, t+K$; let $(x_1, x_2, \ldots, x_K)$ be one of these iteration points such that

$$(x_1 - d_1, x_2 - d_2, \ldots, x_K - d_K) \in T_{c_1, c_2, \ldots, c_K} \text{ with } c_1 + c_2 + \cdots + c_K = t.$$

Then, the distance vector d does not imply dependences on data computed during the superstep t for any of the other $K-1$ iteration points isomorphic with $(x_1, x_2, \ldots, x_K)$. Indeed, assume that such a vertex

$$(y_1, y_2, \ldots, y_K) = \left(x_1 + \alpha_1 \frac{n}{p^{1/(K-1)}}, x_2 + \alpha_2 \frac{n}{p^{1/(K-1)}}, \ldots, x_K + \alpha_K \frac{n}{p^{1/(K-1)}}\right)$$

existed with $\sum_{j=0}^{K} \alpha_j \neq 0$ (the two vertices must belong to tiles computed in different supersteps). Then, we obtain:

$$\begin{array}{l}
(x_1 - d_1, x_2 - d_2, \ldots, x_K - d_K) \in T_{c_1, c_2, \ldots, c_K} \\
\Longrightarrow \\
\forall j : 1..K \bullet c_j \frac{n}{p^{1/(K-1)}} \leq x_j - d_j < (c_j + 1) \frac{n}{p^{1/(K-1)}} \\
\Longrightarrow \\
\forall j : 1..K \bullet (c_j + \alpha_j) \frac{n}{p^{1/(K-1)}} \leq x_j + \alpha_j \frac{n}{p^{1/(K-1)}} - d_j < (c_j + \alpha_j + 1) \frac{n}{p^{1/(K-1)}} \\
\Longrightarrow \\
\forall j : 1..K \bullet (c_j + \alpha_j) \frac{n}{p^{1/(K-1)}} \leq y_j - d_j < (c_j + \alpha_j + 1) \frac{n}{p^{1/(K-1)}} \\
\Longrightarrow \\
(y_1 - d_1, y_2 - d_2, \ldots, y_K - d_K) \in T_{c_1 + \alpha_1, c_2 + \alpha_2, \ldots, c_K + \alpha_K}
\end{array}$$

Since both $(x_1 - d_1, x_2 - d_2, \ldots, x_K - d_K)$ and $(y_1 - d_1, y_2 - d_2, \ldots, y_K - d_K)$ are supposed to be computed in the same superstep $t = \sum_{j=1}^{K} c_j = \sum_{j=1}^{K} (c_j + \alpha_j)$, the assumption that $\sum_{j=0}^{K} \alpha_j \neq 0$ is contradicted. We can therefore "superpose" the K isomorphic tiles, and for any iteration point for which the distance vector d involves an external data dependence, at most one iteration point from the superposed set of tiles will receive this data from a tile computed in superstep t. Hence, the maximum amount of data received by P_x during superstep t due to the distance vector d can be computed as in the first part of the proof, and has the value

$$\left(\frac{n}{p^{1/(K-1)}}\right)^K - \frac{n}{p^{1/(K-1)}} \prod_{j=1}^{K-1} \left(\frac{n}{p^{1/(K-1)}} - d_j\right).$$

The desired result is obtained by summing this amount of data over all distance vectors in D; this summation corresponds to the worst case, i.e., to the case in which the communication overheads due to different distance vectors in D are fully independent.
□

According to the results obtained so far, the cost of a superstep of the BSP schedule is

$$L + \left(\frac{n}{p^{1/(K-1)}}\right)^K c + gComm,$$

where $Comm$ is given by equation (10), and c represents the computation cost for a single point of the iteration space. Then, the cost of the entire schedule is given by

$$(Kp^{1/(K-1)} - K + 1)\left[L + \left(\frac{n}{p^{1/(K-1)}}\right)^K c + gComm\right]$$

or, after simplifications,

$$Kp^{1/(K-1)}L + (K + \mathrm{o}(1))\frac{n^K c}{p} + gK\frac{n^{K-1}}{p^{(K-2)/(K-1)}}\left(\sum_{d\in D}\sum_{j=1}^{K-1} d_j + \mathrm{o}(1)\right). \qquad (11)$$

Consequently, the cost of the BSP schedule is K-optimal[6] if the computation cost dominates both the synchronisation and communication overheads in (11):

$$K\frac{n^K c}{p} \gg Kp^{1/(K-1)}L,\ gK\frac{n^{K-1}}{p^{(K-2)/(K-1)}}\sum_{d\in D}\sum_{j=1}^{K-1} d_j.$$

This requirement is fulfilled if the problem size is large enough for the BSP computer on which the schedule is executed, i.e., if

$$n \gg \max\left\{p^{1/(K-1)}\left(\frac{L}{c}\right)^{1/K}, \frac{gp^{1/(K-1)}\sum_{d\in D}\sum_{j=1}^{K-1} d_j}{c}\right\}. \qquad (12)$$

Example 6 Consider the following 2-level loop nest

```
for i1=0,n-1 do
  for i2=0,n-1 do
    a[i1,i2]=f1(a[i1-1,i2])
    b[i1,i2]=f2(a[i1,i2],b[i1-1,i2-2])
```

The two data dependences of this loop nest are encoded by the distance vectors $(1,0)$ and $(1,2)$, respectively, so it is a fully permutable loop nest whose iteration space is entirely spanned by its distance vectors. Consequently, the schedule in Figure 7 can be used

[6] remember that K is rarely larger than 2 or 3

to map this loop nest onto a BSP computer. The resulting BSP schedule takes $2p-1$ supersteps, and its total cost is

$$(2p-1)L+\frac{(2+\mathrm{o}(1))n^2c}{p}+2gn\left(\sum_{d\in\{(1,0),(1,2)\}}\sum_{j=1}^{1}d_j+\mathrm{o}(1)\right)=(2p-1)L+\frac{2n^2c}{p}+4gn,$$

where $c=cost(f_1)+cost(f_2)$ represents the cost of computing the loop body for a single point of the iteration space. Accordingly, the schedule attains 2-optimality if the problem size n satisfies the constraint

$$n\gg\max\left\{p\left(\frac{L}{c}\right)^{1/2},\frac{2gp}{c}\right\}.$$

For an 8-processor BSP computer with, say, $L=600$ and $g=6$ (i.e., for a typical BSP computer), and for $c=10$, this condition translates into $n\gg\max\{62,10\}=62$, meaning that a speedup of $p/2=4$ will be obtained for the implementation of any problem whose size is larger than about 300.

5.4.1 Improving the load balancing of the BSP schedule

The BSP schedule proposed so far only attains K-optimality for a K-level loop nest; this is due to the fact that in most supersteps, some of the processors are idle. Although K has a very small value for most loop nests that occur in real imperative programs, it is worth trying to improve this result. To do so, the load balancing of the schedule needs improvement. Two methods are available for this, both working by decreasing the granularity of parallelism in order to improve the work distribution of the schedule.

The first method is to partition the iteration space of the loop nest into rectangular tiles of size $n/p^{1/(K-1)}\times n/p^{1/(K-1)}\times\cdots\times n/p^{1/(K-1)}\times x$ (i.e., into tiles whose size along the K-th direction of the iteration space is an unknown parameter $x>0$). The resulting tiles are scheduled for execution in the typical fashion, so tile $T_{c_1,c_2,\ldots,c_K}$, $0\le c_1,c_2,\ldots,c_{K-1}<p^{1/(K-1)}$, $0\le c_K<n/x$, is computed by processor $c_1p^{(K-2)/(K-1)}+c_2p^{(K-3)/(K-1)}+\cdots+c_{K-1}$ in superstep $t=c_1+c_2+\cdots+c_K$. The schedule induced by this strategy takes $(K-1)(p^{1/(K-1)}-1)+n/x$ supersteps. In each such superstep, any processor is assigned at most one tile, so the computation cost of a superstep is:

$$\left(\frac{n}{p^{1/(K-1)}}\right)^{K-1}xc=\frac{n^{K-1}x}{p}c,$$

where c is again the cost of computing the loop body for a single point of the iteration space. We can obtain an upper bound for the amount of data sent or received by any processor during a superstep by using the result provided by Theorem 6, and taking into account that the tile size along the K-th direction of the iteration space is x instead of

$n/p^{1/(K-1)}$:

$$Comm = \sum_{d\in D}\left[\left(\frac{n}{p^{1/(K-1)}}\right)^{K-1} x - x\prod_{j=1}^{K-1}\left(\frac{n}{p^{1/(K-1)}} - d_j\right)\right] =$$
$$= \frac{n^{K-2}x}{p^{(K-2)/(K-1)}}\left(\sum_{d\in D}\sum_{j=1}^{K-1} d_j + \mathrm{o}(1)\right).$$

According to these results, the cost of a superstep of the modified BSP schedule is

$$L + \frac{n^{K-1}x}{p}c + gComm.$$

Hence, the cost of the entire schedule is

$$\left((K-1)(p^{1/(K-1)} - 1) + n/x\right)\left[L + \frac{n^{K-1}x}{p}c + gComm\right]$$

or, after simplifications,

$$\left((K-1)p^{\frac{1}{K-1}} + \frac{n}{x}\right)L + \left(1 + \frac{(K-1)p^{\frac{1}{K-1}}x}{n}\right)\frac{n^K c}{p} + \left(1 + \frac{(K-1)p^{\frac{1}{K-1}}x}{n}\right)\frac{gnComm}{x}. \tag{13}$$

The three terms of the total cost stand for the synchronisation cost, computation cost, and communication cost, respectively. It is immediate to see that the computation cost approaches 1-optimality when x decreases; on the other hand, any decrease in the value of x increases the number of supersteps and therefore the synchronisation overhead. The value of x which minimises the total cost in (13) is the solution of

$$\frac{\mathrm{d}}{\mathrm{d}x}(cost) = -\frac{nL}{x^2} + \frac{(K-1)p^{1/(K-1)}}{n}\left(\frac{n^K c}{p} + \frac{gn^{K-1}\sum_{d\in D}\sum_{j=1}^{K-1} d_j}{p^{(K-2)/(K-1)}}\right) = 0,$$

so the size of the tiles along the K-th direction of the iteration space must be

$$x = \sqrt{\frac{p^{(K-2)/(K-1)}L}{(K-1)n^{K-3}(nc + gp^{1/(K-1)}\sum_{d\in D}\sum_{j=1}^{K-1} d_j)}}. \tag{14}$$

This value depends on both the problem size and the parameters of the BSP computer on which the loop nest is to be executed, and can be obtained at run time.

The second method that can be used to improve the load balancing of the BSP schedule is to partition the iteration space of the loop into hypercubic tiles of size $x\times x\times\cdots\times x$, $x > 0$. A similar scheduling is used, i.e., tile $T_{c_1,c_2,\ldots,c_K}$, $0 \leq c_1, c_2, \ldots, c_K < n/x$ is scheduled to be executed in superstep $t = c_1 + c_2 + \cdots + c_K$ on processor $proc(c_1, c_2, \ldots, c_{K-1})$, where $proc : (0..n/x - 1)^{K-1} \to 0..p-1$. This function must be chosen in such a way that an equal number of points in $(0..n/x - 1)^{K-1}$ is mapped to each value in $0..p-1$, and tiles are assigned to processors in an interleaved fashion. In fact, if the number of points in $(0..n/x-1)^{K-1}$ is much larger than the number of processors p, even a random mapping would do.

To analyse the cost of the modified schedule, we must take into account the fact that this time more than one tile may be assigned to a processor in any superstep. Finding the precise number of tiles per processor is very difficult in the general K-dimensional case, so we will only consider the upper bound for the number of tiles assigned to a single processor in any superstep , i.e., $(n/x)^{K-1}/p$. Accordingly, the communication cost per superstep is bounded by the product between this number of tiles and the communication cost per tile:

$$\frac{1}{p}\left(\frac{n}{x}\right)^{K-1} gx^{K-1}\sum_{d\in D}\sum_{j=1}^{K-1} d_j = g\frac{n^{K-1}}{p}\sum_{d\in D}\sum_{j=1}^{K-1} d_j.$$

Since the modified schedule comprises $K(n/x-1)+1$ supersteps, an upper bound for the total communication overhead of the schedule is

$$g\frac{Kn^K}{px}\sum_{d\in D}\sum_{j=1}^{K-1} d_j.$$

Similarly, the total synchronisation overhead is given by

$$\frac{Kn}{x}L.$$

Hence, both the communication and the synchronisation overheads increase when the size of the tile (i.e., x) decreases. However, the computation cost—a non-trivial upper bound of which is difficult to compute in the general case—approaches 1-optimality when $n \gg x$. The solution for getting a good schedule is therefore to choose x as small as possible such that the computation cost still dominates both the communication and the synchronisation overheads. Since the optimal computation cost (i.e., n^Kc/p) is a lower bound for the actual computation cost, this condition is satisfied when

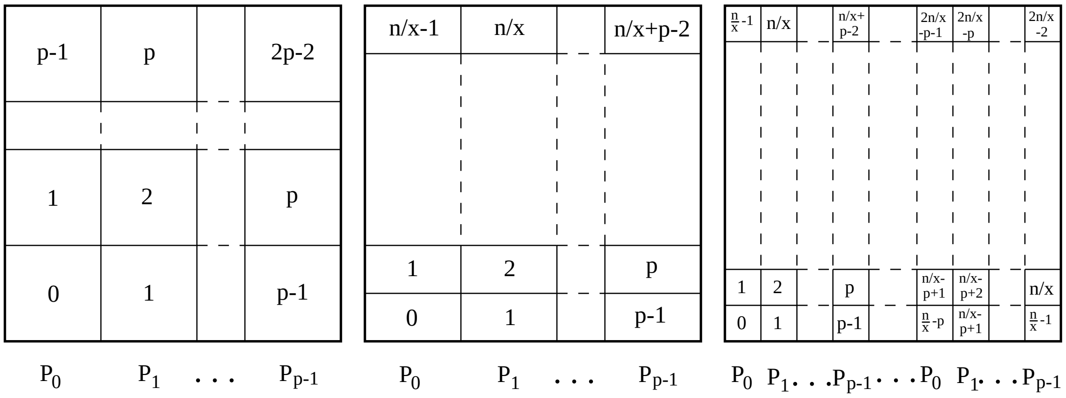
$$\frac{n^Kc}{p} \gg \frac{Kn}{x}L,\ \ g\frac{Kn^K}{px}\sum_{d\in D}\sum_{j=1}^{K-1} d_j.$$

A good solution in practice is to take

$$x = \alpha \max\left\{\frac{KpL}{n^{K-1}c}, \frac{Kg\sum_{d\in D}\sum_{j=1}^{K-1} d_j}{c}\right\},$$

with $\alpha \geq 5$. It is also worth emphasising that for many typical values taken by K in real imperative programs, the number of tiles in any of the $K(n/x-1)+1$ supersteps can be accurately computed, and an optimal value can be obtained for x.

The scheduling of a two-dimensional iteration space using the two methods described in this section is compared with the basic BSP schedule of the same iteration space in Figure 8. As expected, both techniques introduced in this section improve the workload



(a) basic technique (b) method 1 (c) method 2

Figure 8: A comparison between the schedule derived using the basic technique and the schedules induced by the two load balancing improving methods. The same two-dimensional iteration space is scheduled for parallel execution using the basic technique, and the enhanced techniques, respectively. Each tile is marked with the number of the superstep in which it is executed, and the processor that is assigned the execution of each column of tiles is marked under that column.

distribution, at the expense of increased synchronisation and (especially for the second method) communication overheads. The first method yields simpler schedules, but the second one has the advantage of generating balanced schedules even when the iteration space is no longer rectangular. Indeed, while scheduling a triangular two-dimensional iteration space using either the basic technique or the first enhanced technique results in a highly unbalanced schedule, the schedule induced by second method is still well balanced.

5.4.2 Scheduling uniform-dependence loop nests that are not fully permutable

So far we only discussed the scheduling of uniform-dependence loop nests that are also fully permutable. It is not unusual though for a distance vector to have negative components, so the case of loop nests that are not fully permutable must be taken into account. Since fully permutable loop nests are very easy to manipulate, they have been regarded as a normalised form of uniform-dependence loop nests [71], and techniques for the transformation of a generic loop nest into a fully permutable one have been largely studied. All these techniques achieve their goal by applying an affine transformation to the original loop nest, such that the distance vectors have only non-negative components in the transformed iteration space.

Here is a formal description of the method. Let $\mathcal{L}$ be a K-level uniform-dependence loop nest and $D = \{d_1, d_2, \ldots, d_q\}$ its set of distance vectors, some of which have negative elements. An affine transformation T is sought such that

$$\forall j : 1..q \bullet \mathsf{T} d_j^{\mathrm{T}} \geq \mathbf{0}^{\mathrm{T}},$$

where T is the $K \times K$ matrix of the affine transformation, and $\mathbf{0}$ is the q-dimensional vector with all elements 0. If we let D be the $K \times q$ matrix whose columns are the q distance vectors, the requirement may be rewritten [57] as

$$\mathsf{D}^{+} = \mathsf{T} \cdot \mathsf{D} \geq \mathsf{0}, \tag{15}$$

where D^{+} is the $K \times q$ matrix whose columns are the distance vectors in the transformed iteration space, $\mathsf{0}$ stands for the $K \times q$ matrix with all elements 0, and '$\geq$' is applied to all corresponding pairs of elements from its two matrix operands. It is important to mention here that the affine transformation of a loop nest followed by the partitioning of the resulting iteration space into rectangular tiles is equivalent to tiling the iteration space with hyperplanes normal to the rows of the affine transformation matrix. Accordingly, in many cases strategies for finding an appropriate affine transformation of the loop nests are reported as tiling techniques; no real difference exists between the two ways of solving the problem.

Since data dependences occur from a given iteration point to an iteration point in its "future", distance vectors are lexicographically positive, and an affine transformation T satisfying (15) always exists. In fact, the solution of (15) is not even unique, and a carefully chosen criterion must be used to select an appropriate transformation. Two different such criteria have been advocated so far: the simplicity of the transformation, and the minimisation of the communication to computation ratio for the tiles of the transformed iteration space.

Thus, Wolf and Lam [71] propose an algorithm which computes a lower triangular transformation T which satisfies (15) in time $\mathrm{O}(qK^2)$. As the authors do not consider the scheduling of the transformed iteration space onto a distributed-memory computer, the criterion they use is the simplicity of the resulting transformation. Other strategies though consider and attempt to reduce or minimise the communication overheads induced by such a scheduling. The following corollary gives the communication overhead for the BSP scheduling of a transformed iteration loop.

Corollary 7 *Let $\mathcal{L}$ be a K-level uniform-dependence loop nest, and $\mathsf{T} = (t_{ij})_{1 \leq i,j \leq K}$ an affine transformation which applied to $\mathcal{L}$ converts it into a fully permutable loop nest. Then, the maximum amount of data to be fetched or sent by a processor after computing a hypercubic tile of size $x \times x \times \cdots \times x$ of the transformed iteration space is*

$$x^{K-1} \left(\sum_{j=1}^{q} \sum_{i=1}^{K-1} \sum_{k=1}^{K} t_{ik} d_{kj} + \mathrm{o}(1) \right), \tag{16}$$

where $\mathsf{D} = (d_{ij})_{1 \leq j \leq q}^{1 \leq i \leq K}$ is the $K \times q$ matrix whose columns are the distance vectors of $\mathcal{L}$.

Proof Since the distance vectors of the transformed loop are $(d_{1j}^{+}, d_{2j}^{+}, \ldots, d_{Kj}^{+})$, $1 \leq j \leq q$, with $d_{ij}^{+} = \sum_{k=1}^{K} t_{ik} d_{kj}$, (16) directly follows from Theorem 6.
□

Therefore, the quantity to be minimised by an optimal transformation T is the one in (16). Several techniques for the identification of an affine transformation yielding a low communication overhead are briefly presented in the remainder of this section. First, based on results introduced in [35], Ramanujam and Sadayappan [57] formulate the following linear programming problem whose solution is an optimal unimodular lower-triangular transformation T (i.e., $t_{ij} = 0$ for $1 \leq i < j \leq K$, and $t_{ii} = 1$ for $1 \leq i \leq K$):

Find

$$\mathsf{T} = (t_{ij})_{1 \leq j < i \leq K}$$

that minimises

$$\sum_{j=1}^{q} \sum_{i=1}^{K-1} \left(\sum_{k=1}^{i-1} t_{ik} d_{kj} + d_{ij} \right)$$

subject to

$$\sum_{k=1}^{i-1} t_{ik} d_{kj} + d_{ij} \geq 0, \qquad 1 \leq j \leq q,\ 2 \leq i \leq K.$$

The resulting transformation is simple, yet the communication overheads are typically lower than in the case when the algorithm in [71] is used to obtain an affine transformation.

A different strategy has been adopted in [63], where a heuristic is employed to search the solution space for a *blocking* (i.e., tiling) that maximises the computation to communication ratio. As previously emphasised, this kind of tiling is equivalent to an affine transformation of the loop nest, followed by its partitioning into rectangular tiles. Consequently, the transformation induced by the blocking method described in [63] may be used to "normalise" a generic uniform-dependence loop nest before its scheduling on a BSP computer.

Finally, in [9], Boulet et al. argue that minimising the total communication overhead or the communication to computation ratio per tile is very difficult, and devise a method that yields, for any fixed amount of computation, a legal tiling of the iteration space which minimises the communication overhead. If the dependence matrix D is square, the matrix of the affine transformation generated by this method is D^{-1} (recall that D is non-singular), which is optimal under all circumstances.

To conclude, various techniques exist for the conversion of a generic loop nest with uniform dependences to fully permutable form, including techniques that lead to very simple transformations of the loop, and techniques which aim at optimal communication. Although apparently preferable, the latter techniques have an important disadvantage and must be used with a grain of caution. Indeed, the transformed iteration spaces induced by these techniques often comprise iteration points with non-integer coordinates, fact which translates into increased computation costs and a complication of the associated parallel code.

5.5 Iterative scheduling of regular computations

The scheduling technique developed in this section can be applied to uniform-dependence loop nests (Figure 6), i.e., to tightly-nested loops whose data dependences can be represented as a set of distance vectors. Informally, this technique leaves the outermost loop of the loop nest sequential, and parallelise the innermost loops by uniformly distributing the computation of their iterations among the p processors of the parallel machine. For this strategy to produce a correct schedule, any data dependence must relate computations belonging to different iterations of the outermost loop, i.e., the first component of any distance vector must be positive. Therefore, before approaching the scheduling of a generic such loop nest, an affine transformation of its iteration space may be needed to ensure that the first component of any distance vector is positive. Since distance vectors are lexicographically positive, an appropriate affine transformation T,

$$\mathsf{T} = \left(\begin{array}{c|ccc} \alpha_0 & \alpha_1 & \dots & \alpha_K \\ \hline 0 & & & \\ \vdots & & \mathsf{I} & \\ 0 & & & \end{array}\right)$$

with I the $(K-1) \times (K-1)$ identity matrix always exists and can be straightforwardly computed in $\mathrm{O}(K^2 q)$ time, where $K > 1$ is the level of the loop nest to be scheduled, and $q \geq K$ is the number of distance vectors encoding the data dependences of the loop.

The iterative schedule of a uniform-dependence loop nest which obeys the above mentioned constraint (possible after an affine transformation) is presented in Figure 9. The $(K-1)$-dimensional iteration space of the innermost loops of the loop nest is partitioned into p hypercubic tiles of size $n/p^{1/(K-1)} \times n/p^{1/(K-1)} \times \cdots \times n/p^{1/(K-1)}$, and in any iteration of the outermost loop, processor i, $0 \leq i < p$ is assigned the computation of tile $T_{t_2,t_3,\ldots,t_K}$, $t_2 p^{(K-2)/(K-1)} + t_3 p^{(K-3)/(K-1)} + \cdots + t_K = i$. After performing the computation of the loop body for the iteration points in the assigned tile, each processor sends the data involved in external data dependences (i.e., the *halo data*) to the appropriate neighbour processor(s). Each iteration of the outermost loop takes one superstep, so the whole schedule requires n computation supersteps (as well as an initial "start-up" superstep in which loop-external input data is fetched by the processors).

In each of the n supersteps of the schedule, the computation is perfectly distributed among the p processors, so the amount of computation per superstep is $(n^{K-1}c)/p$, where c is the cost of computing the loop body for a single point of the iteration space. The maximum amount of data received or sent by any processor during a superstep is given by the following corollary to Theorem 4.

Corollary 8 *The maximum amount of data sent or received by a processor in any superstep of the BSP schedule in Figure 9 is*

$$\begin{aligned} Comm = \sum_{d \in D} \left[\frac{n^{K-1}}{p} - \prod_{j=2}^{K} \left(\frac{n}{p^{1/(K-1)}} - \mid d_j \mid \right) \right] = \\ = \frac{n^{K-2}}{p^{(K-2)/(K-1)}} \left(\sum_{d \in D} \sum_{j=2}^{K} \mid d_j \mid + \mathrm{o}(1) \right), \end{aligned} \tag{17}$$

```
x=p^(1/(K-1))
for i1=0,n-1 do
  forall t2=0,x-1 do in parallel
    ...........
      forall tK=0,x-1 do in parallel
        Processor t2*x^(K-2)+t3*x^(K-3)+...+tK:
          BSP_begin_superstep
            compute tile Tt2,t3,...,tK:
              for i2=t2*n/x,(t2+1)*n/x-1 do
                ...........
                  for iK=tK*n/x,(tK+1)*n/x-1 do
                    loop_body
            send halo data
          BSP_end_superstep
```

Figure 9: The iterative schedule of a uniform-dependence loop nest.

where D is the set of distance vectors encoding the flow data dependences of the scheduled loop.

Proof Theorem 4 gives the maximum amount of data to be sent by a processor after computing a K-dimensional rectangular tile of size $x_1 \times x_2 \times \cdots \times x_K$. Considering this result for $x_1 = 1$ and $x_2 = x_3 = \cdots = x_K = n/p^{1/(K-1)}$, and taking into account that tiles with different t_1 coordinates, but identical t_2, t_3, $\ldots$, t_K coordinates are computed by the same processor, the quantity in (17) is obtained as an upper bound for the amount of data sent by a processor in any superstep of the BSP schedule.

For the second part of the theorem, consider a distance vector $d \in D$, and a generic processor P_x that is assigned the computation of tile $T_{t_2,t_3,\ldots,t_K}$ in each of the n supersteps of the schedule. Then, for any superstep i_1, $0 \leq i_1 < n$, only the tile computed by processor P_x in superstep $i_1 + d_1$ will depend on data computed in superstep i_1 and linked by data dependences encoded by d. The argument in Theorem 4 can be used to prove that the amount of data received by processor P_x due to these data dependences is again

$$\frac{n^{K-1}}{p} - \prod_{j=2}^{K} \left(\frac{n}{p^{1/(K-1)}} - \mid d_j \mid \right),$$

where the first term represents the volume of tile $T_{t_2,t_3,\ldots,t_K}$, and the second is the volume of that part of the tile for which the data dependences encoded by d are solved internally. Summing this quantity over all distance vectors in D, and recalling that P_x is a generic processor, *Comm* is indeed an upper bound for the amount of data received by any processor during a superstep of the schedule.

□

The results obtained so far show that the cost of a single outermost loop iteration of the schedule in Figure 9 is

$$L + \frac{n^{K-1}c}{p} + g\frac{n^{K-2}}{p^{(K-2)/(K-1)}}\left(\sum_{d\in D}\sum_{j=2}^{K} \mid d_j \mid + \mathrm{o}(1)\right),$$

where D is the set of distance vectors representing the flow data dependences of the loop. Accordingly, the cost of the whole schedule is

$$nL + \frac{n^{K}c}{p} + g\frac{n^{K-1}}{p^{(K-2)/(K-1)}}\left(\sum_{d\in D}\sum_{j=2}^{K} \mid d_j \mid + \mathrm{o}(1)\right). \tag{18}$$

This cost is 1-optimal if it is dominated by the computation cost, i.e., if

$$n \gg \max\left\{\left(\frac{pL}{c}\right)^{1/(K-1)}, \frac{gp^{1/(K-1)}}{c}\sum_{d\in D}\sum_{j=2}^{K} \mid d_j \mid\right\}. \tag{19}$$

Comparing this result with the cost of the uniform-dependence loop nest schedules developed in Section 5.4 (equations (11), (13)), it is immediate to see that the schedule devised in this section is preferable whenever its higher synchronisation overhead is not significant. As shown by relation (19), this condition is likely to be fulfilled when $K > 2$ and the problem size is large, or even when $K = 2$ if the BSP parameter L is low, and the problem size is very large.

Example 7 Let us consider again the 2-level uniform-dependence loop nest in Example 6

```
for i1=0,n-1 do
  for i2=0,n-1 do
    a[i1,i2]=f1(a[i1-1,i2])
    b[i1,i2]=f2(a[i1,i2],b[i1-1,i2-2])
```

and use the schedule in Figure 9 for its mapping onto a BSP computer. Here is the resulting BSP schedule:

```
for i1=0,n-1 do
  forall t2=0,p-1 do in parallel
    Processor t2:
      BSP_begin_superstep
        for i2=t2*n/p,(t2+1)*n/p-1 do
          a[i1,i2]=f1(a[i1-1,i2])
          b[i1,i2]=f2(a[i1,i2],b[i1-1,i2-2])
        if t2<p-1 then
          send b[i1,(t2+1)*n/p-2], b[i1,(t2+1)*n/p-1] to processor t2+1
      BSP_end_superstep
```

```
for i=k,n-1 do
  a[i]=f(a[i-1],a[i-2],...,a[i-k])
```

Figure 10: The generic form of a recurrence loop.

The cost of this schedule is

$$nL + \frac{n^2c}{p} + gn \sum_{d\in\{(1,0),(1,2)} \sum_{j=2}^{2} d_j = nL + \frac{n^2c}{p} + 2gn,$$

where c is the cost of computing the loop body for a single point of the iteration space. This cost is 1-optimal for $n \gg \max\{pL/c, 2gp/c\}$. Under the same circumstances as in Example 6 (i.e., for an 8-processor BSP computer with $L = 600$ and $g = 6$, and for $c = 10$), this requirement translates into $n \gg \max\{480, 10\} = 480$, indicating that the iterative schedule of a 2-level loop nest is feasible only for very large problem sizes.

5.6 Recurrence scheduling

Recurrences represent one of the most common regular patterns of computation appearing in imperative programs. The general form of a recurrence is illustrated in Figure 10, where $0 < k \ll n$ and f is a generic function of k variables. The sequential cost of computing the recurrence loop is cn, where c is the cost of evaluating f at a single point.

To parallelise the execution of the recurrence, one needs to be able to concurrently start the computation at p equally distanced points of the iteration space, i.e., to evaluate $a[i]$ for p equidistant sets of k points without computing the value of the intermediate points. Furthermore, this preliminary evaluation must require sublinear time.

Two cases exist when the above mentioned requirements are satisfied, namely when $k = 1$ and f^x, $x > 1$ can be evaluated to a function no more complex than f, and when $f : \mathbb{R}^k \to \mathbb{R}$ is a linear transformation. In the former case, an 1-optimal BSP schedule can be obtained as follows. First, a_0 is broadcasted to the p processors organised in a d-ary logic tree, $d = L/g$ at a cost of $2L\log_d p$ [65]. Then, in a single computation superstep, each processor j, $0 \le j < p$ computes $f^{jn/p}$ in $\mathrm{O}(\log(jn/p))$ time, evaluates $a[jn/p] = f^{jn/p}(a_0)$ in $\mathrm{O}(1)$ time, and, finally, computes $a[i]$, $jn/p < i < (j+1)n/p$ using the basic definition of the recurrence in cn/p time. The total cost of this computation superstep is

$$\mathrm{O}(\log n) + \mathrm{O}(1) + \frac{nc}{p} = (1 + o(1))\frac{nc}{p},$$

giving a total cost of the entire schedule of

$$L(2\log_{L/g} p + 1) + (1 + o(1))\frac{nc}{p}. \tag{20}$$

This cost is 1-optimal if $n \gg 2pL/c$.

To obtain a BSP schedule for the evaluation of a k-level linear recurrence, we will use the strategy proposed in [31]. Thus, if $f(a[i-1], a[i-2], \ldots, a[i-k]) = \sum_{j=1}^{k} x[k-j]a[i-j]$, we build the $k \times k$ (recurrence) matrix

$$\mathsf{M} = \left(\begin{array}{ccc|c} 0 & \ldots & 0 & x[0] \\ \hline & & & x[1] \\ & \mathsf{I} & & \vdots \\ & & & x[k-1] \end{array}\right) \tag{21}$$

with I the $(k-1) \times (k-1)$ identity matrix, and compute $a[i]$, $jn/p \leq i < jn/p + k$ on processor j, $0 \leq j < p$ using the alternative definition

$$(a[jn/p]\ a[jn/p+1]\ \ldots\ a[jn/p+k-1]) = (a[0]\ a[1]\ \ldots\ a[k-1]) \cdot \mathsf{M}^{jn/p}.$$

The BSP schedule comprises two stages. In the first stage, the values of $a[i]$, $x[i]$, $0 \leq i < k$ are broadcasted to all processors. Using the standard broadcasting strategy, the cost of this stage is $\theta(L+2kg)$. Then, in the second stage, each processor j, $0 \leq j < p$ computes $\mathsf{M}^{jn/p}$ in $\theta(k^3 \log(jn/p))$ time, evaluates $a[i]$, $jn/p \leq i < jn/p + k$ using the alternative definition in (21) in $\theta(k^2)$ time, then computes $a[i]$, $jn/p + k \leq i < (j+1)n/p$ in kn/p time. Accordingly, the cost of the entire schedule is

$$\theta(L+2kg) + \theta(k^3 \log n) + \theta(k^2) + \frac{nk}{p} = \theta(L+2kg) + (1+\mathrm{o}(1))\frac{nk}{p}, \tag{22}$$

which is 1-optimal for

$$n \gg (L+2kg)p/k.$$

5.7 Scheduling regular computations involving broadcasts

There is a very important loop structure which, although stands for a regular pattern of computation, has data dependences that cannot be expressed as distance vectors. This is because certain items of loop-computed data are repeatedly used for the computation of the loop body at many iteration points. Since its mapping on a distributed-memory computer typically requires broadcasts of the multiple-used items of data, we will call this loop structure a *broadcast loop nest.*

As shown in Figure 11, a generic K-level broadcast loop nest, $K > 1$ comprises a number of so-called *broadcast initialisation loops*, and a main computation loop which iteratively computes the elements of a $(K-1)$-dimensional output array a. Each of the $s > 0$ broadcast initialisation loops has the structure in Figure 12, and computes broadcast data used by the following execution of the main computation loop. Some of the broadcasts within the main computation loop may be based on data computed by its previous execution (i.e., $a_k[i_2, \ldots, i_{j-1}, f_k(i_1), i_{j+1}, \ldots, i_K] = a[i_2, \ldots, i_{j-1}, f_k(i_1), i_{j+1}, \ldots, i_K]$ for some $1 \leq k \leq s$), and the corresponding broadcast initialisation loops are then missing. In such a case, an additional auxiliary array a_k and a broadcast initialisation loop can be added to bring the broadcast loop nest to the "normalised" form in Figure 11.

```
for i1=0,n-1 do
  broadcast_init_loop_1
  broadcast_init_loop_2
  ...............
  broadcast_init_loop_s
  for i2=0,n-1 do
    ..............
      for iK=0,n-1 do
        a[i2,...,iK]=f(a[i2,...,iK],...,ak[i2,...,ij-1,fk(i1),ij+1,...,iK],...)
```

Figure 11: A K-level broadcast loop nest.

```
for i2=0,n-1 do
  ...............
    for ij-1=0,n-1 do
      for ij+1=0,n-1 do
        ...............
          for iK=0,n-1 do
            ak[i2,...,ij-1,fk(i1),ij+1,...,iK]=...
```

Figure 12: The k-th broadcast initialisation loop in Figure 11, $1 \leq k \leq s$.

Although in Figure 11 all the loops iterate from 0 to $n-1$, this is not essential for the results devised in this section; the only purpose of this initial assumption was to simplify the description of a broadcast loop nest. In fact, the BSP schedules presented in the remainder of this section do not even require that the iteration space be rectangular.

Before approaching the scheduling of a generic broadcast loop nest, let us analyse several well-known examples of this structure. First, in Figure 13, the pseudocode of triangular linear system solving is arranged to match the notations used for a generic 2-level broadcast loop nest. The triangular linear system to be solved is $\mathsf{C}\mathbf{a_1} = \mathbf{a}$, $\mathsf{C} = (c_{i_1 i_2})_{0 \leq i_1 \leq i_2 < n}$. The loop nest comprises a single broadcast initialisation loop, which computes the elements of the solution, and has level $K-2=0$ (i.e., it is a simple statement). The main computation loop is a $K-1=1$-level loop which uses the broadcast data to iteratively update the elements of the output array a. The elements of the coefficient matrix C of the linear system represent in this case only pure-input data for the loop, and must be fetched by the processors that use them before the actual computation of the loop. Besides the data dependences due to broadcasts, the main computation loop exhibits a flow data dependence encoded by the distance vector $(1,0)$, due to the usage of $a[i_2]$ on both sides of its assignment statement. Finally, it is worth noticing that the iteration space of the broadcast loop nest is triangular.

A second example of broadcast loop nest is Gaussian elimination—Figure 14. The augmented matrix of the linear system is $\mathsf{A} = (a_{i_2 i_3})^{0 \leq i_2 < n}_{0 \leq i_3 \leq n}$. The loop nest comprises two broadcast initialisation loops, each of level $K-2=1$, and a $K-1=2$-level main computation loop. The first broadcast initialisation loop appears in the standard

```
for i₁=0,n-1 do
  a₁[i₁]=a[i₁]/c[i₁,i₁]
    for i₂=i₁+1,n-1 do
      a[i₂]=a[i₂]-c[i₂,i₁]*a₁[i₁]
```

Figure 13: Triangular linear system solving.

```
for i₁=0,n-1 do
  for i₃=i₁,n do
    a₁[i₁,i₃]=a[i₁,i₃]/a[i₁,i₁]
  for i₂=i₁,n-1 do
    a₂[i₂,i₁]=a[i₂,i₁]
  for i₂=i₁,n-1 do
    for i₃=i₁,n do
      if i₂=i₁ then
        a[i₂,i₃]=a₁[i₁,i₃]
      else
        a[i₂,i₃]=a[i₂,i₃]-a₁[i₁,i₃]*a₂[i₂,i₁]
```

Figure 14: Gaussian elimination.

Gaussian elimination algorithm, typically with the output array a on the left-hand side of the assignment statement. The second broadcast initialisation loop, on the other hand, was introduced for conformity with the generic pattern of a broadcast loop nest; its only effect is a renaming of the broadcast data computed by the previous execution of the main computation loop. It is interesting that the first broadcast initialisation loop is itself a broadcast loop nest; however, since its cost is dominated by the cost of performing the main computation loop, we will pay no special attention to its scheduling. The main computation loop has a flow data dependence that can be expressed by the distance vector $(1, 0, 0)$, and the iteration space of the loop is not rectangular, but has the shape of a tetrahedron.

Many other examples of broadcast loop nests exist; they include LU decomposition (without pivoting), Gauss-Jordan elimination, transitive closure, shortest paths, etc.

Two strategies can be used to map a broadcast loop nest onto a BSP computer. The former strategy is the straightforward one: the access of processors to commonly used data is realised through broadcasts. The second strategy on the other hand, is to first convert the broadcast loop nest to an equivalent form which comprises no commonly used data, then to schedule the resulting loop nest as a uniform-dependence loop nest. Both strategies are described in the remainder of this section.

5.7.1 Broadcast loop nest scheduling through broadcast implementation

The straightforward way of scheduling a K-level broadcast loop nest is to sequentialise the execution of its outermost loop, to partition the iteration space of its $(K-1)$-level

```
fetch pure-input data
for i₁=0,n-1 do
  (1) compute the broadcast initialisation loops
  (2) broadcast the data used in common by several processors
  (3) perform the main computation loop:
      forall t₂=0,n/x₂-1 do in parallel
        ..............
          forall tK=0,n/xK-1 do in parallel
            Processor t₂*n/x₃*...*n/xK+...+tK computes tile T_{t₂,t₃,...,tK}
```

Figure 15: The straightforward BSP schedule of a K-level broadcast loop nest.

main computation loop into rectangular tiles of size $x_2 \times x_3 \times \cdots \times x_K$, $\prod_{j=2}^{K} x_j = n^{K-1}/p$, and to execute these tiles concurrently on the p processors of the target BSP machine. Indeed, provided that the outermost loop is executed sequentially and the broadcast data is fetched before the execution of the main computation loop, the iterations of this loop are fully independent, and can be executed concurrently. The distribution of the broadcast data computations (i.e., the partitioning of the broadcast initialisation loops) is induced by the partitioning of the main computation loop. Thus, iteration $(i_2, \ldots, i_{j-1}, i_{j+1}, \ldots, i_K)$ of the k-th broadcast initialisation loop, $1 \leq k \leq s$ is computed by the processor which computes iteration $(i_2, \ldots, i_{j-1}, f_k(i_1), i_{j+1}, \ldots, i_K)$ of the main computation loop.

The scheduling strategy described so far yields the BSP schedule in Figure 15. The outermost loop of the loop nest is left unchanged, so the whole computation is performed in n sequential iterations. Each such iteration comprises three stages. In the first stage, the broadcast initialisation loops are performed, typically in a single superstep; as each broadcast initialisation loop is a $(K-2)$-level loop nest, the cost of this stage is

$$L + \theta(n^{K-2}).$$

The resulting data is then broadcasted during the second stage of any iteration to the processors that will use it for the computation of the main computation loop. The cost of this stage will be assessed later. Finally, in a third stage, which also takes a single superstep, the p tiles of the main computation loop are executed in parallel. Assuming that the iteration space is indeed a hypercube, the cost of this last stage is

$$L + \frac{n^{K-1}c}{p},$$

where c is the cost of computing the body of the main computation loop for a single point of its iteration space. In order to compute the cost of the second stage of the schedule, we need the result provided by the following theorem.

Theorem 9 *Let $\mathcal{L}$ be a K-level broadcast loop nest whose main computation loop is partitioned into p rectangular tiles of size $x_2 \times x_3 \times \cdots \times x_K$, $\prod_{j=2}^{K} x_j = n^{K-1}/p$, $K \geq 2$.*

Let also i_j, $2 \leq j \leq K$ be the index of the missing loop in the k-th broadcast initialisation loop of $\mathcal{L}$, $1 \leq k \leq s$. Then, the cost of broadcasting the data computed by the k-th broadcast initialisation loop is

$$broadcast\_cost = \begin{cases} 0, & \text{if } x_j = n \\ 2L \log_{L/g} p, & \text{if } x_j < n \text{ and } K = 2 \\ 2\left(L + g\frac{n^{K-1}}{px_j}\right), & \text{otherwise} \end{cases} \tag{23}$$

for any iteration of the outermost loop of $\mathcal{L}$.

Proof For any fixed value taken by $(i_2, \ldots, i_{j-1}, i_{j+1}, \ldots, i_K)$ within the iteration space of the main computation loop of $\mathcal{L}$, the value of $a_k[i_2, \ldots, i_{j-1}, f_k(i_1), i_{j+1}, \ldots, i_K]$ is required for the computation of the n iteration points on the segment determined by this fixed value and by $0 \leq i_j < n$. Since the iteration space is partitioned into tiles whose size along the j-th direction is x_j, the iteration points on this segment are assigned to n/x_j of the p processors. Accordingly, for any fixed value taken by $(i_2, \ldots, i_{j-1}, i_{j+1}, \ldots, i_K)$, n/x_j processors need the value of $a_k[i_2, \ldots, i_{j-1}, f_k(i_1), i_{j+1}, \ldots, i_K]$ for the computation of their tiles. In fact, our straightforward schedule ensures that $a_k[i_2, \ldots, i_{j-1}, f_k(i_1), i_{j+1}, \ldots, i_K]$ is computed by one of these n/x_j processors (namely by that processor whose tile comprises the iteration point $(i_2, \ldots, i_{j-1}, f_k(i_1), i_{j+1}, \ldots, i_K)$). As a consequence, for any fixed value taken by $(i_2, \ldots, i_{j-1}, i_{j+1}, \ldots, i_K)$, an item of data must be broadcasted to exactly $n/x_j - 1$ processors, if $x_j < n$, or to no processor at all otherwise. As this result proves the theorem for the case when $x_j = n$, we shall consider that $x_j < n$ in the remainder of this proof.

To compute the maximum number of data items to be broadcasted by any processor during an iteration of the outermost loop of $\mathcal{L}$, we need to find the maximum number of different values taken by $(i_2, \ldots, i_{j-1}, i_{j+1}, \ldots, i_K)$ across a tile. It is immediate to see that this number is a constant equal to $x_2 \ldots x_{j-1} x_{j+1} \ldots x_K$. Using the load balancing constraint $\prod_{l=2}^{K} x_l = n^{K-1}/p$, the number of different items of data to be sent by any processor involved in the computation of the k-th broadcast initialisation loop is $(n^{K-1}/p)/x_j = n^{K-1}/(px_j)$. Finally, since the iteration space of the main computation loop of $\mathcal{L}$ is partitioned along the directions given by $i_2, \ldots, i_{j-1}, i_{j+1}, \ldots, i_K$ with $K-2$ hyperplanes parallel to the broadcast direction, the number of items of data to be received by any processor which is not a sender is again $n^{K-1}/(px_j)$.

To summarise, the p processors can be viewed as partitioned into $n^{K-2}/(x_2 \ldots x_{j-1} x_{j+1} \ldots x_K)$ groups of n/x_j processors, with one of the processors in each group having to broadcast exactly $n^{K-1}/(px_j)$ items of data to all other processors in its group. The most efficient way of implementing the broadcast depends on the value of K. Thus, for $K = 2$, there is only one item of data to be broadcasted by one of the processors to the other $p-1$ processors, and the algorithm in [65] is to be used. This algorithm realises the broadcast by sending the data to the $p-1$ processors in a d-ary tree fashion, with $d = L/g$; the total cost of the broadcast is $2L \log_d p$.

For $K > 2$ on the other hand, $n^{K-1}/(px_j) \gg n/x_j$ for any practical values of n and p, and the standard two-superstep broadcast procedure is used independently for each group of processors. In the first superstep of this procedure, the sender partitions the

data to be broadcasted into $n/x_j - 1$ chunks of equal size, and sends a different chunk of data to each of the other processors in its group. Then, in the second superstep, each processor sends its chunk of data to all the other processors in the group. Both supersteps have the same cost $L + gn^{K-1}/(px_j)$, and the result in (23) is proved.
□

The result provided by the previous theorem allows us to choose the appropriate tiling, and to compute the cost of the second stage of an iteration of the BSP schedule. Thus, if there is any direction l, $2 \leq l \leq K$ such that no broadcast data is used along direction l, then the best partition is that with $x_l = n/p$, and $x_j = n$ if $j \neq l$ because this tiling implies no communication costs at all; the cost of an iteration of the schedule is in this case

$$L + \frac{n^{K-1}c}{p} + \theta(n^{K-2}),$$

because stage 2 does not exist, and the other two stages of each iteration can be executed in a single superstep. Furthermore, since the computations performed by different processors are fully independent, the whole computation can be executed in a single superstep, and the total cost of the schedule is

$$L + \frac{n^K c}{p}(1 + \mathrm{o}(1)).$$

This cost is 1-optimal, provided that the problem size is large enough for the computation cost to dominate the synchronisation overhead, i.e., that

$$n \gg \left(\frac{pL}{c}\right)^{1/K}.$$

If there are several directions that correspond to no broadcast, any tiling which has $x_j = n$ for all broadcast directions j, and respects the load balancing constraint $\prod_{j=2}^{K} x_j = n^{K-1}/p$ has the above cost. The optimal tiling is then the one which minimises the cost of the initialisation superstep, i.e., of the superstep in which the pure-input data is fetched.

Nevertheless, in most practical situations, broadcast loop nests comprise data broadcasts along each of the $K-1$ directions of the main computation loop. As this is by far the most common case, we will assume that a single *broadcast stream* exists for each of these $K-1$ directions. Since the $s = K-1$ broadcasts are independent, the communication cost of the second stage of an iteration of the BSP schedule is given by the sum of the communication costs of the $K-1$ broadcasts. As for the synchronisation cost, it is the same as for a single broadcast, because the $K-1$ broadcasts can be performed concurrently, in two supersteps. Consequently, if $K > 2$, the cost of the second stage is

$$2L + 2\frac{gn^{K-1}}{p}\sum_{j=2}^{K}\frac{1}{x_j}.$$

According to the mean inequality, this cost is minimised for $x_2 = x_3 = \cdots = x_K = n/p^{1/(K-1)}$; its minimum value is

$$2L + 2g\frac{n^{K-2}(K-1)}{p^{(K-2)/(K-1)}}.$$

When this tiling is used, and the first superstep of stage 2 is merged with the only superstep of stage 1, the cost of a whole iteration of the schedule becomes

$$3L + \frac{n^{K-1}c}{p} + \theta(n^{K-2}) + 2g\frac{n^{K-2}(K-1)}{p^{(K-2)/(K-1)}},$$

giving a total cost of the schedule of

$$3nL + \frac{n^{K}c}{p}(1 + \mathrm{o}(1)) + 2g\frac{n^{K-1}(K-1)}{p^{(K-2)/(K-1)}}. \tag{24}$$

Accordingly, 1-optimality is obtained in the general case if the computation cost dominates equation (24), i.e., if the problem size is large enough:

$$n \gg \max\left\{\left(\frac{3pL}{c}\right)^{1/(K-1)}, \frac{2g(K-1)p^{1/(K-1)}}{c}\right\}.$$

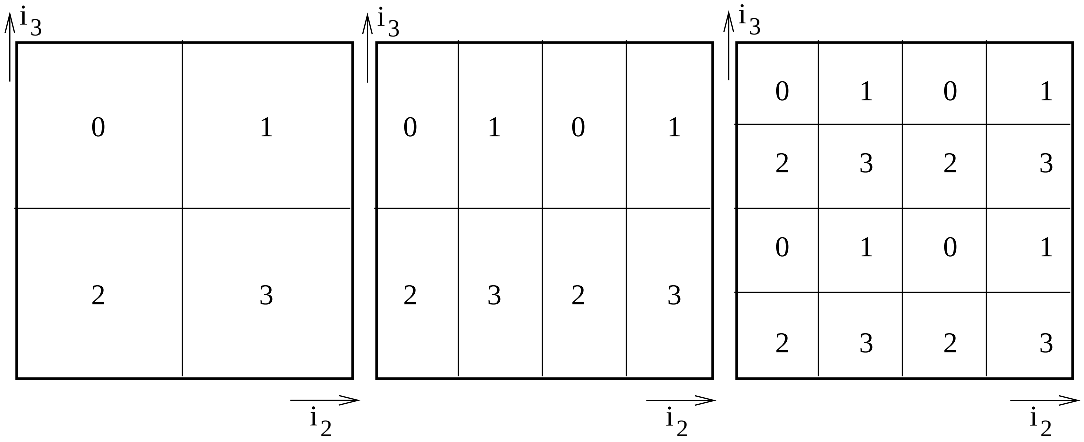
So far, we considered that the iteration space of the main computation loop is rectangular, and were able to achieve 1-optimality due to a perfect load balancing. Unfortunately, the basic schedule does not preserve this perfect load balancing when the iteration space of the main computation loop is not rectangular. To overcome this limitation, the partitioning of the iteration space of the main computation loop must be modified as follows. For any direction of the iteration space along which the number of iteration points varies during the computation, the tiles are *interleaved* along that direction. Informally, interleaving the tiles along the j-th direction of the iteration space, $2 \leq j \leq K$ means changing the size of the tiles along the j-th direction from x_j to x_j/α_j, where $\alpha_j > 1$ is the interleaving factor. More than p tiles result after all interleavings, and tile $\mathrm{T}_{c_2,c_3,\ldots,c_K}$ is assigned to the processor which would execute tile $\mathrm{T}_{c_2 \bmod n/x_2, c_3 \bmod n/x_3, \ldots, c_K \bmod n/x_K}$ in the basic schedule. An example of interleaving for a 3-level broadcast loop nest is presented in Figure 16.

As shown in Figure 16, the number of processors along any direction of the iteration space does not change after the interleaving. Consequently, the interleaving does not modify the broadcast costs. The load balancing, however, is greatly improved: roughly speaking, an interleaving factor of α_j along a direction j reduces the imbalance[7] along that direction α_j times. Ideally, one would choose $\alpha_j = n/x_j$ to minimise the imbalance. In practice however, it is better to choose a smaller value for the interleaving factors (e.g., $\alpha_j = 10..20$) because such a value yields an acceptable load balancing, while still enabling the exploitation of data locality (by the caching system, for instance).

5.7.2 Broadcast loop nest scheduling through broadcast elimination

Let us consider again the generic form of a K-level broadcast loop nest in Figure 11, and assume that $f_k(i_1) = 0$ (or, in the general case, $f_k(i_1) = l_j$, where the loop indexed by i_j

[7]the imbalance of a computation is defined as the ratio between the maximum deviation from the average processor workload and the average processor workload



(a) initial tiling   (b) interleaving along direction i_2 (c) interleaving along both direction i_2 and direction i_3

Figure 16: Tile interleaving for a 3-level broadcast loop nest. The tiles obtained after the standard partitioning (a) are interleaved along direction i_2 (b), then along direction i_3 (c); the index of the processor assigned to each tile is marked on the tile. The interleaving factors $\alpha_2 = \alpha_3 = 2$ were used. The tiling in (b) can be used to schedule Gauss-Jordan elimination, as the iteration space shrinks along direction i_2 in this case, while the tiling in (c) is appropriate for Gauss elimination, whose iteration space shrinks both in direction i_2 and in direction i_3.

is the missing loop from the k-th broadcast initialisation loop, and l_j is the lower bound of the iteration space along direction i_j) for any $1 \leq k \leq s$. Then, the k-th broadcast initialisation loop can be rewritten as in Figure 17, and the s broadcast initialisation loops can be fusioned with the main computation loop. Assuming that the initial broadcast loop nest comprises broadcasts along all directions of the iteration space, the equivalent loop nest which results is a fully permutable loop nest whose data dependences are encoded by the set of distance vectors $D = \{(d_1, d_2, \ldots, d_K) \mid (\exists j : 1..K \bullet d_j = 1) \wedge (\forall j' : 1..K \bullet (j' \neq j) \Rightarrow (d_{j'} = 0))\}$.

This fully permutable loop nest can then be scheduled as shown in Section 5.4; if the first enhanced schedule in this section is used, the total cost of the resulting BSP code is

$$\frac{n}{x}L + (1 + \mathrm{o}(1))\frac{n^K c}{p} + (1 + \mathrm{o}(1))\frac{g(K-1)n^{K-1}}{p^{(K-2)/(K-1)}}, \tag{25}$$

since in this particular case $\sum_{d \in D} \sum_{j=1}^{K-1} d_j = K - 1$. Comparing this cost with the one in (24), it is immediate to notice that both schedules are computationally optimal, but have different synchronisation and communication costs. Thus, the communication cost of the schedule based on broadcast elimination is only half the communication cost of the schedule using broadcast implementation. Although typically negligible for both schedules, it

```
for i2=0,n-1 do
  ...............
    for ij-1=0,n-1 do
      for ij=0,n-1 do
        for ij+1=0,n-1 do
          ...............
            for iK=0,n-1 do
              if ij=0 then
                ak[i2,...,ij-1,ij,ij+1,...,iK]=...
              else
                ak[i2,...,ij-1,ij,ij+1,...,iK]=ak[i2,...,ij-1,ij - 1,ij+1,...,iK]
```

Figure 17: Rewriting a broadcast initialisation loop to eliminate broadcasts when $f_k(i_1)=0$.

is worth pointing out that the synchronisation cost is also smaller for the broadcast elimination schedule. Consequently, the schedule devised in this section is preferable to the broadcast implementation schedule, especially when the iteration space to be scheduled is rectangular. For non-rectangular iteration spaces, however, the application of the schedule based on broadcast implementation may result in better balanced parallel programs, so the usage of the technique devised in Section 5.7.1 is worth considering.

If there is any broadcast initialisation loop k, $1 \leq k \leq s$ for which $f_k(i_1) \neq 0$ (or, in the general case, $f_k(i_1) \neq l_j$, where the loop indexed by i_j is the missing loop from the broadcast initialisation loop k, and l_j is the lower bound of the iteration space along direction i_j), it is still possible to eliminate the broadcasts. This can be achieved by rewriting the k-th broadcast initialisation loop as shown in Figure 18, with two distinct loop nests replacing the original broadcast initialisation loop. Furthermore, each of the other broadcast initialisation loops as well as the main computation loop must be splitted into two loop nests whose iteration spaces coincide with those of the loop nests in Figure 18.

After all the broadcast initialisation loops are converted into $(K-1)$-level loop nests (those with $f_k(i_1) = 0$ into a single loop nest, and those with $f_k(i_1) \neq 0$ into two loop nests), all the loop nests with identical iteration spaces are fusioned. Finally, the outermost loop nest is distributed to the resulting loop nests yielding $2^{s'}$ K-level fully permutable loop nests, where s', $0 \leq s' \leq s$ represents the number of broadcast initialisation loops for which $f_k(i_1) \neq 0$. The set of iteration spaces corresponding to the resulting fully permutable loop nests represents a partitioning of the iteration space of the original loop nest into equally sized chunks. These loop nests can then be executed in the appropriate order, using one of the scheduling techniques in Section 5.4. If the first load balancing improving technique is used, the total cost of the resulting schedule is

$$2^{s'}\left[\frac{n}{x}L + (1+\mathrm{o}(1))\frac{n^K c}{p} + (1+\mathrm{o}(1))\frac{g(K-1)n^{K-1}}{p^{(K-2)/(K-1)}}\right]. \qquad (26)$$

The total cost is $2^{s'}$ times higher than that in (25) because the $2^{s'}$ fully permutable loop nests must be executed in a well-defined order, one after another. As a result, the

```
for i₂=0,n-1 do
  ...............
    for i_{j-1}=0,n-1 do
      for i_j=f_k[i_1],n-1 do
        for i_{j+1}=0,n-1 do
          ...............
            for i_K=0,n-1 do
              if i_j=0 then
                a_k[i_2,...,i_{j-1},i_j,i_{j+1},...,i_K]=...
              else
                a_k[i_2,...,i_{j-1},i_j,i_{j+1},...,i_K]=a_k[i_2,...,i_{j-1},i_j - 1,i_{j+1},...,i_K]
for i_2=0,n-1 do
  ...............
    for i_{j-1}=0,n-1 do
      for i_j=f_k[i_1]+1,0,-1 do
        for i_{j+1}=0,n-1 do
          ...............
            for i_K=0,n-1 do
              a_k[i_2,...,i_{j-1},i_j,i_{j+1},...,i_K]=a_k[i_2,...,i_{j-1},i_j + 1,i_{j+1},...,i_K]
```

Figure 18: Rewriting a broadcast initialisation loop to eliminate broadcasts when $f_k(i_1) \neq 0$.

schedule which implements the broadcasts instead of eliminating them is recommended in this case.

5.8 BSP scheduling of hybrid skeletons

It is often the case that the loop structures which occur in real programs comprise features characteristic to two or more of the regular computation patterns whose BSP scheduling was approached in this chapter. When this happens, the best schedule which exploits these features—possibly a schedule which combines elements belonging to several of the techniques described so far—has to be used. This section presents the most common hybrid loop structures which can be encountered in imperative programs, and discusses their scheduling onto a BSP computer.

5.8.1 Fully parallel outermost loops with reduction innermost loops

Two of the most basic building blocks in imperative programs dealing with dense data structures are matrix-vector multiplication and matrix-matrix multiplication. While optimal BSP algorithms for these basic operations have already been developed (see for instance [48]), the existence of many other instances of the same pattern of computation makes the study of its BSP scheduling worthwhile.

It is not difficult to see that the main computation of the standard matrix-matrix multiplication algorithm (Figure 19) is a 3-level tightly-nested loop comprising two fully-parallel outermost loops and a reduction innermost loop. Accordingly, the scheduling

```
for i1=0,n-1 do
  for i2=0,n-1 do
    for i3=0,n-1 do
      c[i1,i2]=c[i1,i2]+a[i1,i3]*b[i3,i2]
```

Figure 19: Matrix-matrix multiplication: $\mathsf{C} = \mathsf{A} \cdot \mathsf{B}$.

techniques corresponding to both loop structures can be employed to schedule this loop nest. Thus, the iteration space of the two outermost loops can be partitioned as shown in Section 5.2, with the resulting tiles being computed in a single computation superstep; this strategy yields the straightforward BSP schedule described in [48, 65]. Second, one can leave the outermost loops sequential and schedule the n^2 reductions for parallel execution as shown in Section 5.3. Although computation-optimal, this strategy is highly inefficient, mainly due to the very large synchronisation overhead.

Nevertheless, the best schedule for matrix-matrix multiplication is a combination of the two techniques mentioned so far. This schedule, described in [48], requires 2 computation supersteps. In the first computation superstep, each processor executes the loop body for the iteration points in an equally-sized hypercubic tile of the loop nest. The partial results obtained in this superstep are then combined in the second computation superstep to generate the global result. Generalising this schedule, which mixes elements characteristic to both fully-parallel loop nest scheduling and reduction scheduling, Ding and Stefanescu [18] have devised a technique for the efficient scheduling of generic loop nests with the same structure. The scheduling technique proposed in [18] is briefly described in Section 4.3.

5.8.2 Iteration spaces partially spanned by the distance vector set

Another common hybrid loop structure is the loop nest with fully parallelisable outermost loops, and the innermost loops forming a uniform-dependence loop nest whose distance vectors span its entire iteration space. Typically, such a loop structure is induced by a uniform-dependence loop nests whose distance vectors span the iteration space of the loop only partially. Indeed, if a K-level uniform-dependence loop nest has only k, $0 < k < K$ linearly independent distance vectors, then an affine transformation exists that maps the iteration space of the loop to an iteration space in which the distance vectors have null elements along k directions of the iteration space. Consequently, the k loops of the transformed loop nest that correspond to these directions can be made outermost loops and executed in parallel [4].

Once again, techniques corresponding to two loop structures can be employed to schedule such a loop nest. First, one can consider only the outermost k loops of the loop nest, and schedule the whole loop as a fully parallel loop nest. Also, one may schedule the innermost $K - k$ loops as indicated in Sections 5.4 and 5.5, leaving the outermost loops sequential, or attempting to parallelise their execution at the same time. Among these various choices, the former strategy is by far the simplest. Since it also leads to optimal schedules for common problem sizes (see Section 5.2) it is recommended in all circumstances.

6 BSP scheduling of generic loop nests

This chapter discusses the scheduling of loop nests that match none of the regular patterns of computation whose BSP scheduling has been approached in Chapter 5. The results presented in this chapter build on the work of Kuck [38, 39], and of Allen and Kennedy [3]. However, in approaching the scheduling of a generic loop nest, we add new theoretical results that generalise those in [3, 38, 39], and permit the identification of that kind of parallelism which is best suited for the BSP setting.

The BSP scheduling of a generic, untightly-nested loop comprises three stages. In the first stage, a data-dependence graph which is an enhanced version of that proposed in [3] and briefly described in Section 3.1 is built. Then, in a second stage, the information in this dependence graph is used to identify the potential parallelism of the loop nest. Finally, in a third stage, the resulting potential parallelism is scheduled for parallel execution on a BSP computer.

6.1 The enhanced data-dependence graph of a generic loop nest

In this section, we describe the structure of the enhanced data-dependence graph of a generic loop nest, and provide the theoretical background for the identification of the potential parallelism of the loop nest using the information in this graph. Each of the theoretical results introduced in this section will be illustrated with several simple examples. Sometimes, keeping these examples as simple as possible will make them match one of the regular patterns of computation analysed in Chapter 5. Clearly, this resemblance does not invalidate the generality of the new results, or their applicability to the parallelisation of generic loop nests.

Throughout the section we will consider a generic loop nest $\mathcal{L}$ whose statements are S_i, $1 \leq i \leq N$, and whose loop indices i_j, $1 \leq j \leq K$, iterate from 0 to $n_j - 1$. However, $\mathcal{L}$ need not be a perfect loop nest, i.e., we allow the body of any loop to comprise both other loops and/or statements. The loops of $\mathcal{L}$ are labeled with natural numbers between 1 and K. This labeling is done in such a way that any set of perfect nested loops are assigned consecutive numbers, and, if the labels of two loops are $1 \leq x < y \leq K$, then loop x cannot belong to the body of loop y.

With the above defined notations, the enhanced data-dependence graph of $\mathcal{L}$ is a directed graph $\mathcal{G}$ comprising a vertex for each of the N statements of $\mathcal{L}$, and an edge between each pair of statements S_x, S_y, $1 \leq x, y \leq N$ for which the non-existence of any dependence $\mathrm{S}_x\delta^*\mathrm{S}_y$ cannot be proved. Each vertex of $\mathcal{G}$ is assigned a label $\mathrm{S}_j(L)$, where S_j is the statement represented by that vertex, and L represents the ordered set of loops surrounding statement S_j. Each edge of $\mathcal{G}$ is also assigned a label $\delta^*(S, P)$, where $\delta^* \in \{\delta, \overline{\delta}, \delta^o\}$ represents the type of the dependence, and the significance of S and P is explained later in this section. We will start with a theorem which summarises results from [3].

Theorem 10 (Allen and Kennedy [3]) *Let* S_x *and* S_y *be two (not necessarily distinct) statements of a generic loop nest* $\mathcal{L}$ *which are surrounded by* $k \geq 1$ *common loops* *indexed by $i_1, i_2, \ldots, i_k$. Suppose that a data dependence $\mathrm{S}_x\delta^*\mathrm{S}_y$ must be assumed between the two statements due to the fact that both statements use the m-dimensional array a (or, if $\mathrm{S}_x = \mathrm{S}_y$, due to the fact that a is both read and modified by the statement):*

```
Sx:   ...a[f(i1,i2,...,ik)]...
Sy:   ...a[g(i1,i2,...,ik)]...
```

with $f, g : \mathbb{Z}^k \to \mathbb{Z}^m$. Then, if for some value $\beta \in 1..k$, and for any legal values $(x_1, x_2, \ldots, x_k) < (y_1, y_2, \ldots, y_k)$ assigned to $(i_1, i_2, \ldots, i_k)$,

$$(x_1, x_2, \ldots, x_\beta) = (y_1, y_2, \ldots, y_\beta) \Rightarrow f(x_1, x_2, \ldots, x_k) \neq g(y_1, y_2, \ldots, y_k) \tag{27}$$

and, if $\beta > 1$, it cannot be proved that

$$(x_1, x_2, \ldots, x_{\beta-1}) = (y_1, y_2, \ldots, y_{\beta-1}) \Rightarrow f(x_1, x_2, \ldots, x_k) \neq g(y_1, y_2, \ldots, y_k), \tag{27'}$$

the sequential execution of the outermost β loops surrounding S_x and S_y satisfies all the restrictions imposed by this data dependence on the execution order of the iteration space of $\mathcal{L}$, and β is called the upper nesting level *of the data dependence*[8].

Proof Consider two generic iteration points $(x_1, x_2, \ldots, x_k) < (y_1, y_2, \ldots, y_k)$ such that

$$f(x_1, x_2, \ldots, x_k) = g(y_1, y_2, \ldots, y_k).$$

The original loop nest executes these two iterations in lexicographical order, i.e., first $(x_1, x_2, \ldots, x_k)$, then $(y_1, y_2, \ldots, y_k)$. Consequently, any rewriting of the loop which preserves this execution order satisfies the restrictions imposed by the considered data dependence.

Assume now that the outermost β loops surrounding S_x and S_y are executed sequentially, and (some of) the other loops are executed concurrently. Then, according to the hypothesis of the theorem, $(x_1, x_2, \ldots, x_\beta)$ must be lexicographically smaller than $(y_1, y_2, \ldots, y_\beta)$. Hence, iteration $(x_1, x_2, \ldots, x_k)$ is executed before iteration $(y_1, y_2, \ldots, y_k)$, and the new execution order of the iteration space of $\mathcal{L}$ satisfies the restrictions imposed by the data dependence $\mathrm{S}_x\delta^*\mathrm{S}_y$.
□

The name of upper nesting level given to β from (27) is justified by the fact that the associated data dependence can be viewed as "eliminated" when the outermost β loops surrounding the two statements involved in the dependence are left sequential. Theorem 10 provides a strategy for the parallelisation of the innermost loops of a loop nest.

Example 8 For the untightly-nested loop

[8] β is called the *nesting level* of the data dependence in [3]

```
     for i₁=0,n₁-1 do
S1:    a[i₁]=f(i₁)
       for i₂=0,n₂-1 do
S2:      b[i₁,i₂]=g(a[i₁],i₂)
```

a

```
     forall i₁=0,n₁-1 do in parallel
S1:    a[i₁]=f(i₁)
       forall i₂=0,n₂-1 do in parallel
S2:      b[i₁,i₂]=g(a[i₁],i₂)
```

b

Figure 20: A loop nest comprising a loop-independent flow dependence $S_1 \delta S_2$ (a), and the potential parallelism of the loop (b).

```
     for i₁=0,n₁-1 do
       for i₂=0,n₂-1 do
S1:      a[i₁,i₂]=f(i₁,i₂)
         for i₃=0,n₃-1 do
S2:        b[i₁,i₂,i₃]=g(a[i₁-1,i₂],i₃)
```

a flow data dependence $S_1 \delta S_2$ of upper nesting level 1 exists because

$$\forall (x_1, x_2), (y_1, y_2) : 0..n_1 - 1 \times 0..n_2 - 1 \bullet x_1 = y_1 \Rightarrow (x_1, x_2) \neq (y_1 - 1, y_2),$$

and

$$\forall (x_1, x_2), (y_1, y_2) : 0..n_1 - 1 \times 0..n_2 - 1 \bullet (x_1, x_2) \neq (y_1 - 1, y_2)$$

cannot be proved to be true. Accordingly, the loop can be rewritten as

```
     for i₁=0,n₁-1 do
       forall i₂=0,n₂-1 do in parallel
S1:      a[i₁,i₂]=f(i₁,i₂)
         forall i₃=0,n₃-1 do in parallel
S2:        b[i₁,i₂,i₃]=g(a[i₁-1,i₂],i₃)
```

with the potential parallelism of the loop revealed.

An extreme case which must be addressed separately is that of a loop-independent data dependence. Indeed, for a loop-independent dependence, relation (27) does not hold for any value $\beta \in 1..k$. Therefore, in this extreme case we will take $\beta = \infty$, indicating that the dependence is not eliminated however many loops surrounding the two statements are left sequential. It is immediate to see, however, that all the loops surrounding two statements involved in a loop-independent dependence can be executed in parallel without affecting the correctness of the transformed code. An example of a loop nest comprising a loop-independent dependence is presented in Figure 20.

The following theorem is complementary to Theorem 10: it defines the *lower nesting level* of a data dependence, and provides a way of using this new parameter of a dependence for the extraction of the coarse-grained potential parallelism of a generic loop nest.

Theorem 11 *Let* S_x *and* S_y *be two (not necessarily distinct) statements of a generic loop nest* $\mathcal{L}$ *which are surrounded by* $k \geq 1$ *common loops indexed by* $i_1, i_2, \ldots, i_k$*. Suppose that a data dependence* $\mathrm{S}_x \delta^* \mathrm{S}_y$ *must be assumed between the two statements due to the fact that both statements use the* m*-dimensional array* a *(or, if* $\mathrm{S}_x = \mathrm{S}_y$*, due to the fact that* a *is both read and modified by the statement):*

```
Sx:   ...a[f(i1,i2,...,ik)]...
Sy:   ...a[g(i1,i2,...,ik)]...
```

with $f, g : \mathbb{Z}^k \to \mathbb{Z}^m$*. Then, if for some value* $\alpha \in 1..k$*, and for any legal values* $(x_1, x_2, \ldots, x_k) < (y_1, y_2, \ldots, y_k)$ *assigned to* $(i_1, i_2, \ldots, i_k)$ *it cannot be proved that*

$$(x_1, x_2, \ldots, x_\alpha) \neq (y_1, y_2, \ldots, y_\alpha) \Rightarrow f(x_1, x_2, \ldots, x_k) \neq g(y_1, y_2, \ldots, y_k), \tag{28}$$

and, if $\alpha > 1$*,*

$$(x_1, x_2, \ldots, x_{\alpha-1}) \neq (y_1, y_2, \ldots, y_{\alpha-1}) \Rightarrow f(x_1, x_2, \ldots, x_k) \neq g(y_1, y_2, \ldots, y_k), \tag{28$'$}$$

the sequential execution of the innermost $k - \alpha + 1$ *loops surrounding* S_x *and* S_y *satisfies all the restrictions imposed by this data dependence on the execution order of the iteration space of* $\mathcal{L}$*, and* α *is called the* lower nesting level *of the data dependence.*

Proof As in the proof of Theorem 10, consider two generic iteration points $(x_1, x_2, \ldots, x_k) < (y_1, y_2, \ldots, y_k)$ such that

$$f(x_1, x_2, \ldots, x_k) = g(y_1, y_2, \ldots, y_k).$$

Again, any rewriting of the loop which executes $(x_1, x_2, \ldots, x_k)$ and $(y_1, y_2, \ldots, y_k)$ in lexicographical order satisfies the restrictions imposed by the considered data dependence.

Assume now that the outermost $\alpha - 1$ loops surrounding S_x and S_y are executed concurrently, and all the other $k - \alpha + 1$ loops are left sequential. Then, according to the definition of α (equation (28)), $(x_1, x_2, \ldots, x_{\alpha-1}) = (y_1, y_2, \ldots, y_{\alpha-1})$, and both iterations of interest are executed in the same iteration of outermost parallel loops. The order in which they are executed is imposed by the lexicographical order of $(x_\alpha, x_{\alpha+1}, \ldots, x_k)$ and $(y_\alpha, y_{\alpha+1}, \ldots, y_k)$. Obviously, since $(x_1, x_2, \ldots, x_k) < (y_1, y_2, \ldots, y_k)$ and $(x_1, x_2, \ldots, x_{\alpha-1}) = (y_1, y_2, \ldots, y_{\alpha-1})$, we must have $(x_\alpha, x_{\alpha+1}, \ldots, x_k) < (y_\alpha, y_{\alpha+1}, \ldots, y_k)$, so the two iteration points are executed in the correct order.
□

Example 9 Here is an example of a 2-level loop nest comprising a flow data dependence $\mathrm{S}_1 \delta \mathrm{S}_1$ of lower nesting level $\alpha = 2$:

```
      for i1=0,n1-1 do
        for i2=0,n2-1 do
S1:       a[i1,i2]=f(a[i1,i2 - 1])
```

Then, according to Theorem 11, the outermost loop of the nest can be executed in parallel:

```
        forall i₁=0,n₁-1 do in parallel
          for i₂=0,n₂-1 do
S1:         a[i₁,i₂]=f(a[i₁,i₂ − 1])
```

The potential parallelism of the transformed loop can be readily scheduled for execution on a BSP computer, yielding a 1-superstep BSP program. It is worth emphasising that, since the upper nesting level of this loop is $\beta = 2$, the method described in [3] and summarised by Theorem 10 is not able to identify any parallelism in the loop. Furthermore, although a loop interchange would enable the application of the parallelisation strategy in [3], it will only expose the fine-grained parallelism in the loop, leading to a n_2-superstep BSP program.

The lower nesting level of a data dependence can be regarded as the maximum value α such that the sequentialisation of loop α and of the innerer loops "breaks" (i.e., eliminates) the data dependence. Accordingly, the lower nesting level of a loop-independent dependence is considered to be $\alpha = 0$. Indeed, not even the sequentialisation of all loops surrounding two statements involved in a loop-independent data dependence breaks the dependence.

The last theorem presented in this section provides a way of increasing the granularity of the potential parallelism extracted from a generic loop nest, as well as a method for the identification of parallel loops situated between the lower and the upper nesting level of a data dependence.

Theorem 12 *Let* S_x *and* S_y *be two (not necessarily distinct) statements of a generic loop nest* $\mathcal{L}$ *which are surrounded by* $k \geq 1$ *common loops indexed by* $i_1, i_2, \ldots, i_k$*. Assume that a data dependence* $\mathrm{S}_x\delta^*\mathrm{S}_y$ *exists between the two statements due to the fact that both statements use the* m*-dimensional array* a *(or, if* $\mathrm{S}_x = \mathrm{S}_y$*, due to the fact that* a *is both read and modified by the statement):*

```
Sx:  ...a[f(i₁,i₂,...,iₖ)]...
Sy:  ...a[g(i₁,i₂,...,iₖ)]...
```

with $f, g : \mathbb{Z}^k \to \mathbb{Z}^m$*. Let* $P \subseteq 1..k$ *be the set of all loops* j*,* $1 \leq j \leq k$*, with the property that, for any legal values* $(x_1, x_2, \ldots, x_k) < (y_1, y_2, \ldots, y_k)$ *assigned to* $(i_1, i_2, \ldots, i_k)$*,*

$$x_j \neq y_j \Rightarrow f(x_1, x_2, \ldots, x_k) \neq g(y_1, y_2, \ldots, y_k). \tag{29}$$

Then, some or all the loops in P *(which is called the* set of fully parallel loops of the data dependence*) can be made outermost loops and executed in parallel without violating the restrictions imposed by the considered data dependence on the execution order of the iteration space of* $\mathcal{L}$*.*

Proof Assume that the loop nest $\mathcal{L}$ is rewritten such that some (or all) the loops in P are made outermost loops and executed in parallel. We will prove the theorem by showing that any pair of iteration points $(x_1, x_2, \ldots, x_k) < (y_1, y_2, \ldots, y_k)$ for which $f(x_1, x_2, \ldots, x_k) = g(y_1, y_2, \ldots, y_k)$, is executed in the correct, lexicographical order by

the transformed loop. To do so, it is enough to notice that, for any loop $j \in P$, $x_j = y_j$ for any pair of interdependent iteration points. Consequently, any two interdependent iteration points are executed in the same iteration of the outermost parallel loops, and in a order imposed by the lexicographical order of the elements of the two index vectors that correspond to the other loops. Since only equal elements are "eliminated" from the two index vectors, this is exactly the lexicographical order of the iteration points, and the proof is complete.
□

It is worth noticing that, with the notations in Theorem 12, the set of fully parallel loops of a loop-independent data dependence is $P = 1..k$. Indeed, all the loops surrounding two statements involved in a loop-independent data dependence can be parallelised as long as the order of the two statements within the loop body is preserved.

The three parameters of a data dependence introduced so far are not redundant. In order to prove this, we will present several examples of simple loop nests whose parallelisation cannot be achieved without the usage of the two new parameters, α and P. First, let us compare the parameters α and β. From their definitions in (28) and (27), respectively, it is immediate to see that, for any data dependence, $\alpha \leq \beta$. The flow data dependence $S_1 \delta S_1$ in the following loop nest is an example of a data dependence whose lower nesting level is actually lower than its upper nesting level:

```
        for i₁=0,n₁-1 do
          for i₂=0,n₂-1 do
            for i₃=0,n₃-1 do
S1:           a[i₁,i₂,i₃]=f(a[i₁,i₃,i₃ − 1])
```

In this case, the lower nesting level of the data dependence is $\alpha = 2$ because $x_1 \neq y_1 \Rightarrow (x_1, x_2, x_3) \neq (y_1, y_3, y_3 - 1)$ for any values assigned to (x_1, x_2, x_3), (y_1, y_2, y_3), and it cannot be proved that $(x_1, x_2) \neq (y_1, y_2) \Rightarrow (x_1, x_2, x_3) \neq (y_1, y_3, y_3 - 1)$. The upper nesting level of the data dependence, on the other hand, is $\beta = 3$, since $(x_1, x_2, x_3) = (y_1, y_2, x_3) \Rightarrow (x_1, x_2, x_3) \neq (y_1, y_3, y_3 - 1)$, and it cannot be proved that $(x_1, x_2) = (y_1, y_2) \Rightarrow (x_1, x_2, x_3) \neq (y_1, y_3, y_3 - 1)$ for any legal value assigned to (x_1, x_2, x_3), (y_1, y_2, y_3). Hence, we have $2 = \alpha < \beta = 3$, and the outermost loop of the loop nest can be executed in parallel:

```
        forall i₁=0,n₁-1 do in parallel
          for i₂=0,n₂-1 do
            for i₃=0,n₃-1 do
S1:           a[i₁,i₂,i₃]=f(a[i₁,i₃,i₃ − 1])
```

Finally, P, the set of fully parallel loops of a data dependence, adds new useful information to that conveyed by α and β. Indeed, although $1..(\alpha - 1) \subseteq P$, P can also comprise other loops, as it does for the flow data dependence $S_1 \delta S_1$ in the following loop

```
        for i₁=0,n₁-1 do
          for i₂=0,n₂-1 do
S1:         a[i₁,i₂]=f(a[i₁-1,i₂])
```

In this case, $\alpha = \beta = 1$, and $P = \{2\}$, showing that the inner loop can be brought into the outermost position and parallelised:

```
        forall i₂=0,n₂-1 do in parallel
          for i₁=0,n₁-1 do
S1:           a[i₁,i₂]=f(a[i₁-1,i₂])
```

Since the usage of α and β alone would only permit the parallelisation of the loop indexed by i_2 as the innermost loop, the introduction of P is justified. At a first glance, it might seem that $P = 1..(\alpha - 1) \cup (\beta + 1)..k$. This is not true, as illustrated by the following example:

```
        for i₁=0,n₁-1 do
          for i₂=0,n₂-1 do
            for i₃=0,n₃-1 do
              for i₄=0,n₄-1 do
S1:               a[i₁,i₂,i₃,i₄]=f(a[i₂,i₂,i₃-1,i₄+1])
```

The flow data dependence $S_1 \delta S_1$ in this example has $\alpha = 1$, $\beta = 3$, and $P = \{2\}$. Consequently, the loop between the lower and the upper nesting levels of the dependence (i.e., the loop indexed by i_2) can be brought to the outermost position and parallelised, while the loop indexed by $i_{\beta+1}$ can only be parallelised in the innermost position. In conclusion, although the three parameters of a data dependence are related, each of them conveys different dependence information. Furthermore, the parameters β and P must be determined independently, whereas the parameter α can be deduced from the value of P:

$$\alpha = \begin{cases} 0, & \text{if } P = 1..k \\ 1, & \text{if } 1 \notin P \\ \gamma + 1, & \text{if } 1..\gamma \subseteq P \wedge \gamma + 1 \notin P \end{cases}$$

So far, we have introduced three parameters characterising a data dependence between two statements that access the elements of the same array within the body of a generic loop nest. In doing so, we assumed that various comparisons of the subscripts selecting the array elements used by the two statements can be performed. However, we have provided no clue about how these comparisons can actually be realised. This is because we are not proposing any new method for the assessment of the truth of propositions (27)-(27′), (28)-(28′), or (29). Nevertheless, several ways of doing these comparisons, such as the greatest common divisor (GCD) test [3] or the Banerjee inequality test [5], do exist, and can be used in conjunction with the new results devised in this section.

The remainder of this section describes how the information provided by the three parameters of a data dependence is incorporated into the enhanced data-dependence graph $\mathcal{G}$ of a generic loop nest. The straightforward way of integrating the parameters of a data dependence into $\mathcal{G}$ would be to simply make them part of the label assigned to the edge corresponding to the considered data dependence. However, this is only possible for the set of fully parallel loops of a data dependence P. This is because no labeling of the loops of a generic loop nest $\mathcal{L}$ exists which would permit the interpretation of the lower

```
        for i₁=0,n₁-1 do
          for i₂=0,n₂-1 do
S1:         a[i₁,i₂]=f₁(i₁,i₂)
            for i₃=0,n₃-1 do
S2:           b[i₁,i₂,i₃]=f₂(a[i₁,i₂],b[i₁+1,i₂+1,i₃],c[i₂,i₂-1,i₃])
S3:           c[i₁,i₂,i₃]=f₃(b[i₁-1,i₂-1,i₃+2])
```

Figure 21: Example of a untightly-nested loop.

and upper nesting levels of all the data dependences of $\mathcal{L}$ in a consistent manner. Indeed, in the general case, it is impossible to label the loops of a generic loop nest in such a way that any pair of statements is surrounded by loops which are assigned consecutive numbers. In fact, as illustrated by the following example, this is impossible to achieve even for very simple loop nests:

```
for i₁=0,n₁-1 do
  for i₂=0,n₂-1 do
    S1
    S2
  for i₃=0,n₃-1 do
    S3
    S4
```

In this example, the three loops cannot be labeled in such a way that both statements $\mathrm{S}_1, \mathrm{S}_2$, and statements $\mathrm{S}_3, \mathrm{S}_4$ are surrounded by consecutively numbered loops. As a result, when assessing the lower and upper nesting levels of the dependences between S_1 and S_2, and between S_3 and S_4, one has to assume a different labeling of the three loops. Since this requirement renders the resulting lower and upper nesting levels of a dependence dependent on the dependence itself, a way of coherently expressing the information encapsulated in the two parameters is needed.

The solution adopted by the enhanced dependence graph introduced in this section is to represent this information as the set of loops that must be kept unchanged for the considered data dependence to be satisfied. This set, which is the complementary of P, is denoted S, and comprises all the loops situated between the lower and the upper nesting levels of the considered dependence, and which are not in P.

Example 10 An example of a more complex generic loop nest, and its associated enhanced dependence graph are depicted in Figure 21 and Figure 22, respectively. As illustrated by the dependence $\mathrm{S}_1\delta\mathrm{S}_2$ in this example, the set S of a loop-independent data dependence comprises no loop at all. Also, some of the loops surrounding the statements involved in a data dependence may belong neither to S, nor to P; for the flow dependence $\mathrm{S}_2\delta\mathrm{S}_3$ for instance, $2, 3 \notin S \cup P$.

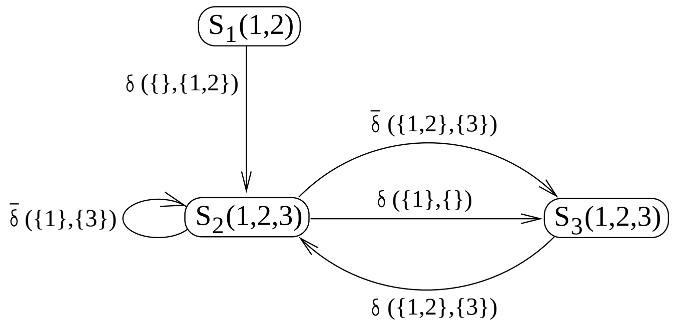


Figure 22: The enhanced dependence graph of the loop nest in Figure 21.

6.2 Potential parallelism identification in a generic loop nest

In this section, an algorithm for the identification of the potential parallelism available in a generic loop nest is introduced. This algorithm is an extension of the loop distribution algorithm of Kuck [38], and of the parallel code generation algorithm of Allen and Kennedy [3]. The main advantages of the new algorithm are the usage of the enhanced data-dependence graph of the loop nest for the extraction of more parallelism, and the fact that it exposes the coarse-grained parallelism in the loop nest rather than the fine-grained one. The latter advantage, crucial in the BSP setting where fine-grained parallelism may translate into prohibitive synchronisation overheads, is, to our knowledge, unique among the parallelisation algorithms proposed so far. The algorithm is based on the following corollary to Theorems 10-12.

Corollary 13 *Let $\mathcal{L}$ be a generic loop nest and $\mathcal{G}$ its associated enhanced data-dependence graph. Assume that the $n \geq 1$ edges of $\mathcal{G}$ are labeled $\delta^*_j(S_j, P_j)$, with $1 \leq j \leq n$, and that $S = \cup_{j=1}^{n} S_j$ and $P = \cap_{j=1}^{n} P_j$. Then, if*

- *the loops indexed by i_j, $j \in P$ are brought into the outermost position and executed in parallel;*
- *the loops indexed by i_j, $j \in S$ are left unchanged;*
- *the loops indexed by i_j, $j \notin S \cup P$ are executed in parallel,*

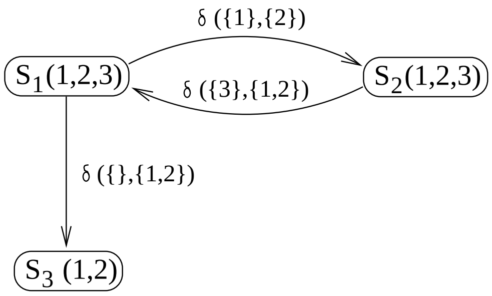
the restrictions imposed by all the data dependences of $\mathcal{L}$ are not violated.

Proof For any data dependence j, $1 \leq j \leq n$, the parallel version of the loop nest proposed by the corollary keeps all the loops in S_j sequential, and brings to the outermost position only loops in P_j. Hence, according to the results in Theorems 10, 11 and 12, the restrictions imposed by all the n data dependences of $\mathcal{L}$ are satisfied.
□

Example 11 Figure 23 illustrates the usage of the corollary. The loop nest depicted in Figure 23(a) comprises three flow data dependences. First, statement S_3 uses a value computed by statement S_1 in the same iteration of the common surrounding loops, so a loop-independent dependence $S_1 \delta S_3$ exists; this dependence is labeled $\delta(S_1, P_1)$, with $S_1 = \{\}$ and $P_1 = \{1, 2\}$ (Figure 23(b)). Second, statement S_2 uses a value computed by statement S_1 in a previous iteration, so a loop-carried data dependence $S_1 \delta S_2$ with

```
       for i1=0,n1-1 do
         for i2=0,n2-1 do
           for i3=0,n3-1 do
S1:          a[i1,i2,i3]=f1(b[i1,i2,i3-1])
S2:          b[i1,i2,i3]=f2(a[i1-1,i2,i3-1])
S3:        c[i1,i2]=f3(a[i1,i2,n3-1])
```

(a)



(b)

```
       forall i2=0,n2-1 do in parallel
         for i1=0,n1-1 do
           for i3=0,n3-1 do
S1:          a[i1,i2,i3]=f1(b[i1,i2,i3-1])
S2:          b[i1,i2,i3]=f2(a[i1-1,i2,i3-1])
S3:        c[i1,i2]=f3(a[i1,i2,n3-1])
```

(c)

Figure 23: A generic loop nest (a), its associated enhanced data-dependence graph (b), and the potential parallelism exposed by Corollary 13 (c).

$\alpha = \beta = 1$ (and hence with $S_2 = \{1\}$) and $P_2 = \{2\}$ exists. Finally, the elements of array b modified by statement S_2 are used in a later iteration by statement S_1; a flow data dependence with $\alpha = \beta = 3$ (i.e., with $S_3 = \{3\}$) and $P = \{1, 2\}$ exists between the two statements.

Consider now $S = S_1 \cup S_2 \cup S_3 = \{1, 3\}$ and $P = P_1 \cap P_2 \cap P_3 = \{2\}$. Then, according to Corollary 13, the three dependences are satisfied when the loop indexed by i_2 is brought into the outermost position and executed in parallel, and the other two loops are left unchanged, so the parallel version in Figure 23(c) is correct. This parallel version reveals enough coarse-grained parallelism for the loop nest to be executed in a single superstep on a BSP computer.

However, it is worth emphasising that Corollary 13 does reveal insufficient or, indeed, no parallelism at all in some cases. Consider for instance the following example of a loop nest:

```
          for i1=0,n1-1 do
            for i2=0,n2-1 do
S1:           a[i1,i2]=f1(a[i1-1,i2])
S2:           b[i1,i2]=f2(a[i1,i2-1])
```

This loop nest comprises two data dependences, $S_1\delta S_1$ with $\alpha = \beta = 1$ and $P_1 = \{2\}$, and $S_1\delta S_2$ with $\alpha = \beta = 2$ and $P_2 = \{1\}$. Obviously, since $S = S_1 \cup S_2 = \{1,2\}$ and $P = \{\}$, the application of Corollary 13 identifies no potential parallelism in the loop nest. However, the following parallel version of the loop nest is correct:

```
          forall i2=0,n2-1 do in parallel
            for i1=0,n1-1 do
S1:           a[i1,i2]=f1(a[i1-1,i2])
          forall i1=0,n1-1 do in parallel
            forall i2=0,n2-1 do in parallel
S2:           b[i1,i2]=f2(a[i1,i2-1])
```

When such cases occur, the loop nest cannot be parallelised as a whole, but must be broken into separate parts which are parallelised independently.

Before presenting an algorithm which, in addition to applying Corollary 13, realises this partitioning of the loop nest whenever necessary, let us make very clear what the parallel program designer expects from such an algorithm. Consider the following example of a loop nest, in which the problem size n is much greater than p, the number of processors of the BSP computer.

```
for i1=0,n-1 do
  S1
  for i2=0,n-1 do
    S2
    for i3=0,n-1 do
      S3
```

The sequential cost of executing the loop nest is $O(n^3)$, so what one expects from a p-processor BSP version of this loop nest is an execution cost of $O(n^3/p)$. Obviously, this can only be obtained if the execution of statement S_3—which is called the *main statement* of the loop nest—is parallelised. Furthermore, the fact that statements S_1 and S_2 are executed in parallel or not is irrelevant for the successful parallelisation of the loop nest as a whole. Consequently, a parallelisation algorithm must concentrate on the identification of potential parallelism corresponding to the main statements of a loop nest, possibly neglecting the less significant parts of the loop.

A second feature that must characterise a good parallelisation algorithm is illustrated by the following loop nest:

```
for i1=0,n-1 do
  for i2=0,2 do
    for i3=0,n-1 do
      S1
```

Assuming again that the problem size n is much greater than the number of processors p, any successful BSP version of this loop nest must be able to concurrently execute the loop body for different iteration points along direction(s) i_1 or/and i_3 of the iteration space. Indeed, a parallelisation of the loop indexed by i_2 is of little use in this case. Therefore, a parallelisation algorithm must focus on those loops whose iteration space size depends on the problem size—these are called the *main loops* of the loop nest—rather than on the loops whose index bounds are (small) constants.

Finally, assume that a main statement S of a loop nest is surrounded by several main loops which iterate from 0 to $n-1$, where n is the problem size, and suppose that one of these main loops can be parallelised. As emphasised throughout this section, the *relative level* of the parallel loop among the other main loops is of crucial importance. If the parallel loop is in the innermost position (i.e., has the relative level 1), a synchronisation is required after every n executions of the statement S. As a result, the BSP program induced by this parallelisation could be efficient only if the problem size is much greater than the synchronisation cost L, which is a very restrictive constraint. If, on the other hand, the parallel loop has the relative level 2 (i.e., it is in the second innermost position), barrier synchronisations are only required after each n^2 (parallel) computation steps. Hence, an efficient BSP program can be obtained for $n^2 \gg L$, which is a much more relaxed constraint. Accordingly, when attempting to identify the potential parallelism of a loop nest, one must also pay attention to the relative level of the loops that are parallelised.

The observations made so far are taken into account by the parallelisation algorithm in Figure 24. This algorithm identifies the potential parallelism of a generic loop nest using the information in its enhanced data-dependence graph $\mathcal{G}$. In doing so, the algorithm attempts to parallelise the execution of all statements in the main statement set `Stmts`, which is provided as a second parameter to the algorithm. A statement in `Stmts` is considered to be successfully parallelised when (at least) one of its surrounding main loops can be executed in parallel, and the relative level of this loop is greater than or equal to a threshold level `th_level` provided as input to the algorithm. The set of the main loops of the loop nest (denoted `Loops`) is also a parameter of the algorithm.

The algorithm works as follows. In a first stage, the algorithm attempts to parallelise the loop nest as a whole by applying Corollary 13 to the data-dependence graph $\mathcal{G}$ received as the first parameter. The resulting parallel version of the loop nest is accepted—and the algorithm terminates—if each statement from `Stmts` is successfully parallelised. The algorithm also terminates when the loop nest comprises a single statement, since in this case no partitioning of the loop is possible.

When none of the two termination conditions is fulfilled, the algorithm enters a second stage, in which the partitioning of the loop nest is attempted. First, the π-blocks (i.e., the maximal cycles or individual vertices belonging to no cycle [3, 39]) of $\mathcal{G}$ are identified. Then, if $\mathcal{G}$ comprises a single π-block, the cyclic dependence of this π-block is broken by applying Theorem 10 to a data dependence belonging to the cycle and whose elimination requires that a minimum number of loops are kept sequential. Clearly, one should always try to eliminate a data dependence that requires the sequentialisation of loops not in `Loops`. Once the cyclic dependence is broken, the edge corresponding to the eliminated

```
proc parallelise(𝒢, Stmts, Loops, th_level)
  Stage 1.Parallelise the loop nest as a whole
    apply Corollary 13 to 𝒢
    if 𝒢 comprises a single vertex then
      STOP
    if any statement in Stmts is surrounded by at least a parallel loop
    of relative level greater than or equal to th_level from Loops then
      STOP
  Stage 2.Partition the loop nest and parallelise the resulting parts
    partition 𝒢 into π-blocks
    if 𝒢 comprises a single π-block then
      break the cyclic dependence by sequentialising the minimum number of loops
      adjust 𝒢 and Loops to account for the changes
      call parallelise(𝒢, Stmts, Loops, th_level)
    else
      lexicographically sort the π-blocks
      in lexicographical order, call parallelise recursively for each π-block
```

Figure 24: Potential parallelism identification algorithm.

data dependence is deleted from $\mathcal{G}$, and the parallelisation algorithm is applied recursively to the resulting data-dependence graph.

Finally, when $\mathcal{G}$ comprises many π-blocks, these π-blocks are lexicographically ordered, and the parallelisation procedure is recursively called for each π-block. In order to apply the algorithm to an individual π-block, the reduction of $\mathcal{G}$ to the set of statements belonging to the block, and the relevant sets of statements `Stmts` and main loops `Loops` are computed in the straightforward way. Then, a call to the parallelisation procedure with the new set of parameters is issued.

Example 12 To illustrate the usage of the parallelisation algorithm, let us identify the potential parallelism of the loop nest in Figure 21 by calling

$$\mathtt{parallelise}(\mathcal{G}, \mathtt{Stmts}, \mathtt{Loops}, \mathtt{th\_level})$$

with $\mathcal{G}$ the enhanced data-dependence graph in Figure 22, $\mathtt{Stmts} = \{\mathtt{S}_2, \mathtt{S}_3\}$, $\mathtt{Loops} = \{1, 2, 3\}$, and $\mathtt{th\_level} = 2$. In the first stage of the algorithm, the application of Corollary 13 yields $S = \cup_{j=1}^{3} S_j = \{1, 2\}$ and $P = \cap_{j=1}^{3} P_j = \{\}$, so only the loop indexed by i_3 can be parallelised in the innermost position:

```
         for i₁=0,n₁-1 do
           for i₂=0,n₂-1 do
S1:          a[i₁,i₂]=f₁(i₁,i₂)
             forall i₃=0,n₃-1 do in parallel
S2:            b[i₁,i₂,i₃]=f₂(a[i₁,i₂],b[i₁+1,i₂+1,i₃],c[i₂,i₂-1,i₃])
S3:            c[i₁,i₂,i₃]=f₃(b[i₁-1,i₂-1,i₃+2])
```

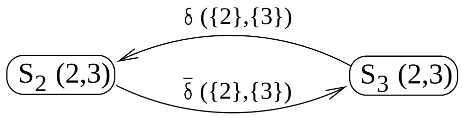
Since for all the statements in `Stmts` the relative level of the parallel loop is $1 <$ `th_level` $= 2$, the main statements of the loop nest are not successfully parallelised, and the algorithm proceeds to Stage 2. In this stage, the data-dependence graph $\mathcal{G}$ is partitioned into two π-blocks, one comprising the vertex S_1 and no edge, and the other comprising the cycle formed by vertices S_2 and S_3. Next, the parallelisation procedure is called recursively for the two π-blocks:

$$\texttt{parallelise}(\mathcal{G}_1, \{\}, \{1,2\}, 2)$$
$$\texttt{parallelise}(\mathcal{G}_2, \{\mathrm{S}_2, \mathrm{S}_3\}, \{1,2,3\}, 2)$$

where $\mathcal{G}_1$ and $\mathcal{G}_2$ represent the reductions of the data-dependence graph $\mathcal{G}$ to the two π-blocks. The parallelisation of the first π-block succeeds in Stage 1, because this π-block comprises a single vertex; the parallel version of this part of the loop nest is:

```
           forall i1=0,n1-1 do in parallel
             forall i2=0,n2-1 do in parallel
S1:            a[i1,i2]=f1(i1,i2)
```

For the second π-block, however, Stage 1 is again unable to reveal enough parallelism: only the loop indexed by i_3 is parallelised in the innermost position. As a result, the re-partitioning of $\mathcal{G}_2$ is attempted in Stage 2 of the algorithm. Since $\mathcal{G}_2$ comprises a single π-block, a new partition is not possible, so the algorithm attempts to break the cyclic dependence. Obviously, the data dependence whose elimination requires the sequentialisation of a minimum number of loops is $\mathrm{S}_2\delta\mathrm{S}_3$. Hence, this data dependence is eliminated by keeping the loop indexed by i_1 sequential, and the parallelisation procedure is called recursively for the data-dependence graph



This time, the application of Corollary 13 parallelises the loop indexed by i_3 in the outermost position, succeeding to reveal enough coarse-grained parallelism for the algorithm to terminate. The final parallel version of the entire loop nest is:

```
           forall i1=0,n1-1 do in parallel
             forall i2=0,n2-1 do in parallel
S1:            a[i1,i2]=f1(i1,i2)
           for i1=0,n1-1 do
             forall i3=0,n3-1 do in parallel
               for i2=0,n2-1 do
S2:              b[i1,i2,i3]=f2(a[i1,i2],b[i1+1,i2+1,i3],c[i2,i2-1,i3])
S3:              c[i1,i2,i3]=f3(b[i1-1,i2-1,i3+2])
```

6.3 Scheduling the potential parallelism of a generic loop nest

Once the coarse-grained potential parallelism of a generic loop nest is identified, it must be scheduled for parallel execution on a BSP computer. This section discusses the underlying principles of a successful mapping of this potential parallelism on a BSP machine.

First, the principal target of an efficient scheduling strategy must be the balanced distribution of the main statement executions among the p processors of the parallel computer. This goal can be achieved by considering the parallel loops surrounding each main statement, partitioning their iteration space into p equally sized tiles, and computing each such tile on a different processor.

Second, since the communication overheads may grow high if the scheduling of each main statement of a loop nest is done independently, one has to correlate the scheduling of various main statements. This can be done by tiling only the iteration space of those parallel loops which induce an identical or similar distribution of the data used within the loop body. Consider the following example of a parallel version of a generic loop nest:

```
        forall i1=0,n-1 do in parallel
          forall i2=0,n-1 do in parallel
            for i3=0,n-1 do
S1:           ...a[i1,i2,i3]...
        forall i2=0,n-1 do in parallel
          forall i3=0,n-1 do in parallel
            for i1=0,n-1 do
S2:           ...a[i1-1,i2,i3]...
```

If the iteration space of the first loop nest are tiled along directions i_1 and i_2, and the iteration space of the second loop nest along directions i_2 and i_3, the elements of array a need to be redistributed after the computation of the first loop nest. Since in this case each processor has to exchange its block of n^3/p elements of a for another block of the same size, the communication cost of this data redistribution is gn^3/p. Clearly, this cost may well dominate the computation cost $\mathrm{O}(n^3/p)$, rendering the considered BSP schedule inefficient. If, on the other hand, the iteration space of both loop nests is tiled along direction i_2 alone, a similar distribution of array a results for the two loop nests. As this reduces the communication cost to gn^2, an efficient BSP schedule can thus be obtained.

It is possible, however, that no tiling direction common to all the main statements of the considered loop nest exists. When this is the case, it is necessary to choose between the concurrent scheduling of all the main statements and the concurrent scheduling of only some of them. The decision must be made in accordance with the (communication) cost of the data redistribution required by the first choice.

As concerns the secondary statements of a loop nest, they should be scheduled in a way which preserves as much as possible the data distribution induced by the scheduling of the main statements. Consider for instance the following parallel version of a loop nest:

```
        forall i₁=0,n-1 do in parallel
          forall i₂=0,n-1 do in parallel
S1:         ...a[i₁,i₂]...
        forall i₂=0,n-1 do in parallel
          for i₁=0,n-1 do
            for i₃=0,n-1 do
S2:           ...a[i₁,i₂]...
```

In this case, statement S_2 must be scheduled by tiling the iteration space of the surrounding loops along direction i_2. This tiling induces a data distribution in which each of the p processors is assigned n/p columns of array a. To preserve this data distribution (thus avoiding additional communication costs), the iteration space of the first loop must also be tiled along direction i_2 of its iteration space. The extent to which the data distribution preservation must take precedence over other scheduling criteria is difficult to assess in the general case. Here is an example of a loop nest whose secondary statement scheduling must be based on the actual values of the BSP parameters for the target computer:

```
        forall i₁=0,n-1 do in parallel
          for i₂=0,n-1 do
S1:         a[i₁,i₂]=f(a[i₁,i₂-1])
        forall i₂=0,n-1 do in parallel
          for i₁=0,n-1 do
            for i₃=0,n-1 do
S2:           ...a[i₁,i₂]...
```

As in the previous example, the iteration space of the second loop nest must be tiled along direction i_2. This tiling yields a data distribution in which each processor is assigned n/p columns of the array a. This distribution can only be maintained for the first 2-level loop nest if this loop nest is sequentialised, and the processors compute a $n \times n/p$ tile of its iteration space in p consecutive supersteps. The total cost of executing the first loop nest in this way is $pL+cn^2+gpn$. An alternative to this strategy is to execute the first loop nest concurrently, in the straightforward way, and then to redistribute the elements of array a. The cost of the resulting BSP schedule of the first loop is $L + cn^2/p + gn^2/p$. Obviously, the less expensive strategy can only be decided at run time, when the parameters of the target BSP machine are known.

7 Conclusions and future work

One of the major aims currently focusing the efforts of the parallel computing community is the development of a realistic basis for the design and programming of general purpose parallel computers. The advent of the bulk-synchronous parallel model has offered a solution to this problem, providing a possible underlying framework for the devising of both scalable parallel architectures and portable parallel software. Consequently, a huge amount of research work is now targeted at the standardisation of the newly emerged model, and especially at the development of a unanimously accepted set of BSP programming primitives [30]. Taking into account the present pace of progress in the latter pursuit, it is very likely that a universal, BSP-based platform for portable parallel application design will be agreed upon in the very near future.

Notwithstanding the advantages of the direct, manual design of BSP algorithms and applications, one cannot neglect the existence of a broad basis of efficient sequential solutions for the most various classes of problems. If the huge amount of work invested into the development of this basis of sequential code is to be taken advantage of, then reliable techniques for the automatic mapping of sequential code onto a BSP computer are needed. A first, important step in this direction has been made with the recent devising of effective techniques for the identification of the potential parallelism of a sequential program. As a result of this step, a virtual parallel version of a sequential program can be automatically generated. How to actually map this (PRAM-like) virtual parallelism onto a BSP computer is a question whose definite answer is still to be found.

A first solution to this problem was proposed in [65, 66], where optimally-efficient BSP simulations of PRAMs were devised by exploiting parallel slackness and by using shared-memory address hashing. Simulation, however, places restrictive demands on the target BSP architecture, requiring [47] powerful multithreading support and fast context switching and address translation capabilities. Furthermore, even if parallel architectures providing these features were built, simulation could attain the lower bounds established for the execution of a virtual parallel program on a real parallel computer only when no locality whatsoever exists in the simulated code [58]. Since the lack of locality is a reality only for the computations involving complex, sparse data structures, simulation is very unlikely to ever represent a feasible generic solution for the mapping of potential parallelism on BSP computers. An alternative solution for this mapping is therefore worth seeking.

It is such an alternative to PRAM simulation that the project whose first steps are presented in this report proposes by considering the BSP scheduling of potential parallelism. Whereas hitherto scheduling has merely been used to derive architecture-specific parallel programs or special-purpose parallel designs, no real impediment exists to its usage in the context of portable parallel code generation. Indeed, as shown in Chapter 5, regular patterns of computation can be successfully identified and efficiently scheduled within the framework of the BSP model. Furthermore, the kind of potential parallelism required for the BSP scheduling of generic loop nests can be identified using the techniques presented in Chapter 6.

Nevertheless, the work carried out so far only covers the initial phase of our project, which aims at developing a coherent framework for bulk-synchronous parallel scheduling. Thus, further efforts will be dedicated to the integration of the observations in Section 6.3 into an algorithm for the the BSP scheduling of generic loop nests. Also, we intend to use the scheduling techniques devised so far for the development of a generic strategy for the BSP scheduling of virtual parallel programs, and, finally, to implement this strategy as a useful tool for the design of portable parallel software.

Acknowledgements

The author would like to thank Prof. W.F. McColl for his advice and encouragement. I am also highly indebted to Dr. M.B. Giles for suggesting the first load balancing improving technique described in Section 5.4.1 of the current report.

References

[1] T.L. Adam et al. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17:685–690, 1974.

[2] J.R. Allen et al. Conversion of control dependence to data dependence. In *Conference Record of the 10th ACM Symposium on Principles of Programming Languages*, pages 177–189. ACM Press, 1983.

[3] J.R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[4] D F Bacon et al. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):346–420, December 1994.

[5] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[6] U. Banerjee. A theory of loop permutation. In *Proceedings of the 2nd Workshop on Programming Languages and Compilers for Parallel Computing*, 1989.

[7] U. Banerjee. Unimodular transformations of double loops. In A. Nicolau, editor, *Advances in Languages and Compilers for Parallel Processing*, Research Monographs in Parallel and Distributed Computing. MIT Press, 1991.

[8] R.H. Bisseling and W.F. McColl. Scientific computing on bulk synchronous parallel architectures. Technical Report 836, Department of Mathematics, Utrecht University, December 1993.

[9] P. Boulet et al. (Pen)-ultimate tiling? *Integration, the VLSI Journal*, 17(1):33–51, August 1994.

[10] R. Calinescu. Bulk synchronous parallel algorithms for conservative discrete event simulation. *Parallel Algorithms and Applications*, 9:15–38, 1996.

[11] R. Calinescu. Bulk synchronous parallel algorithms for optimistic discrete event simulation. Technical Report TR-8-96, Programming Research Group, Oxford University Computing Laboratory, April 1996.

[12] R. Calinescu. Bulk synchronous parallel scheduling of uniform dags. In Luc Bouge et al., editors, *Euro-Par'96. Parallel Processing*, volume 2 of *Lecture Notes in Computer Science 1124*, pages 555–562. Springer–Verlag, 1996.

[13] R. Calinescu and D.J. Evans. A parallel model for dynamic load balancing in clustered distributed systems. *Parallel Computing*, 20(1):77–91, January 1994.

[14] R. Calinescu and D. Grigoras. A neural self-organizing scheme for dynamic load allocation. In A. De Gloria et al., editors, *Transputer Applications and Systems '94*, pages 860–868. IOS Press, 1994.

[15] T.L. Casavant and J.G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, February 1988.

[16] T. Cheatham et al. Bulk synchronuous parallel computing – a paradigm for transportable software. Technical Report TR-36-94, Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts, December 1994.

[17] D. Culler et al. LogP: Towards a realistic model of parallel computation. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 19–22. ACM Press, 1993.

[18] Y.-Z. Ding and D. Stefanescu. Efficient scheduling of loop nests for BSP programs. Technical Report TR-04-95, Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts, 1995.

[19] J. Dongarra and A.R. Hind. Unrolling loops in FORTRAN. *Software Practice and Experience*, 9(3):219–226, March 1979.

[20] H. El-Rewini et al. *Task Scheduling in Parallel and Distributed Systems.* PTR Prentice Hall, 1994.

[21] A.P. Ershov. ALPHA—an automatic programming system of high efficiency. *Journal of the ACM*, 13(1):17–24, January 1966.

[22] P. Feautrier. Some efficient solutions to the affine scheduling problem. part I, one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, October 1992.

[23] P. Feautrier. Some efficient solutions to the affine scheduling problem. part II, multi-dimensional time. *International Journal of Parallel Programming*, 21(6):389–420, December 1992.

[24] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[25] A. Gerasoulis and T. Yang. A comparison of clustering heuristics for scheduling DAGs on multiprocessors. *Journal of Parallel and Distributed Computing*, 14(4):276–291, December 1992.

[26] A.V. Gerbessiotis and L.G Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22(2):251–267, August 1994.

[27] A. Gibbons and P Spirakis, editors. *Lectures on Parallel Computation*. Cambridge International Series on Parallel Computation: 4. Cambridge University Press, 1993.

[28] L.A. Goldberg et al. An optical simulation of shared memory. In *SPAA '94. 6th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 257–267. ACM, 1994.

[29] M. W. Goudreau et al. The Green BSP library. Technical Report CS-TR-95-11, Department of Computer Science, University of Central Florida, Orlando, 1995.

[30] M. W. Goudreau et al. A proposal for the BSP Worlwide standard library. Technical report, BSP Worldwide Technical Report, April 1996. Available from http://www.bsp-worldwide.org/standard/.

[31] A. C. Greenberg et al. Efficient parallel algorithms for linear recurrence computation. *Information Processing Letters*, 15(1):31–35, August 1982.

[32] D. Grigoras and R. Calinescu. μkernel service for dynamic load balancing. In B.M. Cook et al., editors, *Transputer Applications and Systems '95*, pages 269–279. IOS Press, 1995.

[33] T.J. Harris. A survey of PRAM simulation techniques. *ACM Computing Surveys*, 26(2):187–206, June 1994.

[34] K.T. Herley and G. Bilardi. Deterministic simulations of PRAMs on bounded degree networks. *SIAM Journal of Computing*, 23(2):276–292, April 1994.

[35] F. Irigoin and F. Triolet. Supernode partitioning. In *Conference Record of the 15th ACM Symposium on Principles of Programming Languages*, pages 319–329. ACM Press, 1988.

[36] H. Kasahara and S. Narita. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Transactions on Computers*, C-33(11):1023–1029, November 1984.

[37] W. Kelly and W. Pugh. Finding legal reordering transformations using mappings. In K. Pingali et al., editors, *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science 892, pages 107–124. Springer–Verlag, 1995.

[38] D.J. Kuck. A survey of parallel machine organization and programming. *ACM Computing Surveys*, 9(1):29–59, March 1977.

[39] D.J. Kuck. *The Structure of Computers and Computations.* John Wiley and Sons, 1978.

[40] L Lamport. The parallel execution of do loops. *Communications of the ACM*, 17(2):83–93, February 1974.

[41] C. Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR'93*, Lecture Notes in Computer Science 715, pages 398–416. Springer–Verlag, 1993.

[42] V. Leppanen and M. Penttonen. Work–optimal simulation of PRAM models on meshes. *Nordic Journal of Computing*, 2(1):51–69, Spring 1995.

[43] W. Li and K. Pingali. A singular loop transformation framework based on nonsingular matrices. In *Proceedings of the 5th Workshop on Languages and Compilers for Parallel Computers*, pages 249–260, 1992.

[44] A.W. Lim and M.S. Lam. Communication-free parallelization via affine transformations. In K. Pingali et al., editors, *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science 892, pages 92–106. Springer-Verlag, 1995.

[45] D.B. Loveman. Program improvement by source-to-source transformation. *Journal of the ACM*, 24(1):121–145, January 1977.

[46] B.A. Malloy et al. Scheduling DAG's for asynchronous multiprocessor execution. *IEEE Transactions on Parallel and Distributed Systems*, 5(5):498–508, May 1994.

[47] W F McColl. General purpose parallel computing. In A. Gibbons and P. Spirakis, editors, *Lectures on Parallel Computation*, Cambridge International Series on Parallel Computation: 4, pages 337–391. Cambridge University Press, 1993.

[48] W.F. McColl. Scalable computing. In J. Van Leeuwen, editor, *Computer science today : recent trends and developments*, Lecture Notes in Computer Science 1000, pages 46–61. Springer-Verlag, 1995.

[49] R. Miller. *Two approaches to architecture-independent parallel computation.* PhD thesis, Oxford University Computing Laboratory, 1994.

[50] R. Miller and J.L. Reed. The Oxford BSP library: Users guide version 1.0. Technical report, Oxford Parallel Technical Report, Oxford University Computing Laboratory, 1994.

[51] J.M. Nash et al. Parallel algorithm design on the WPRAM model. Technical Report 94.24, Division of Computer Science, University of Leeds, July 1994.

[52] M.T. O'Keefe and H.G. Dietz. Loop coalescing and scheduling for barrier MIMD architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):1060–1063, September 1993.

[53] D.A. Padua and M.J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[54] A. Pietracaprina et al. Constructive deterministic PRAM simulation on a mesh-connected computer. In *SPAA '94. 6th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 248–256. ACM, 1994.

[55] C.D. Polychronopoulos. Loop coalescing: A compiler transformation for parallel machines. In *Proceedings of the International Conference on Parallel Processing*, pages 235–242. Pennsylvania State Univ. Press, 1987.

[56] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[57] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–120, October 1992.

[58] A Ranade. A framework for analyzing locality and portability issues in parallel computing. In F. Meyer auf der Heide et al., editors, *Parallel Architectures and Their Efficient Use*, Lecture Notes in Computer Science 678, pages 185–194. Springer-Verlag, 1993.

[59] B. Rau and J.A. Ficher. Instruction-level parallel processing: History, overview and perspective. *Journal of Supercomputing*, 7(1–2):9–50, May 1993.

[60] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Research Monographs in Parallel and Distributed Computing. MIT Press, 1989.

[61] V. Sarkar and J. Hennessy. Compile time partitioning and scheduling of parallel programs. In *SIGPLAN Symposium on Compiler Construction*, pages 17–26, 1986.

[62] V. Sarkar and R. Thekkath. A general framework for iteration-reordering transformations. *SIGPLAN Notes*, 27(7):175–187, July 1992.

[63] R. Schreiber and J.J. Dongarra. Automatic blocking of nested loops. Technical Report 90-38, Univ. of Tennessee at Knoxville, May 1990.

[64] R. Subramonian. Designing synchronous algorithms for asynchronous processors. In *SPAA '92. 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 189–198. ACM, 1992.

[65] L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[66] L.G. Valiant. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science. Volume A: Algorithms and Complexity*, pages 943–971. North Holland, 1990.

[67] U. Vishkin. Structural parallel algorithms. In A. Gibbons and P Spirakis, editors, *Lectures on Parallel Computation*, Cambridge International Series on Parallel Computation: 4, pages 1–18. Cambridge University Press, 1993.

[68] J R L Webb. *Functions of several real variables.* Ellis Horwood series in mathematics and its applications. Statistics and operational research. Ellis Horwood, 1991.

[69] D. Wedel. FORTRAN for the Texas Instruments ASC system. *SIGPLAN Notes*, 10(3):119–132, March 1975.

[70] M.E. Wolf and M.S. Lam. Maximizing parallelism via loop transformations. In *Programming Languages and Compilers for Parallel Computing*, 1990.

[71] M.E. Wolf and M.S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–470, October 1991.

[72] M J Wolfe. *Optimizing Supercompilers for Supercomputers.* Research Monographs in Parallel and Distributed Computing. MIT Press, 1989.

[73] M.J. Wolfe. More iteration space tiling. In *Proceedings Supercomputing '89*, pages 655–664. ACM Press, 1989.

[74] Y.-Q. Yang et al. Minimal data dependence abstractions for loop transformations. In K. Pingali et al., editors, *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science 892, pages 201–216. Springer-Verlag, 1995.