Partitioning
for
Parallel Finite Difference Computations
in
Coastal Water Simulation

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus
Prof.dr.ir. J. Blaauwendraad,
in het openbaar te verdedigen
ten overstaan van een commissie,
door het College van Dekanen aangewezen,
op maandag 8 september 1997 te 13.30 uur
donk

Mark Roy Taco ROEST,
natuurkundig ingenieur,
geboren te Zwijndrecht.
Dit proefschrift is goedgekeurd door de promotoren:

Prof.dr.ir. L. Dekker
Prof.dr.ir. A.W. Heemink

Samenstelling promotiecommissie:

Rector Magnificus, voorzitter,
Prof.dr.ir. L. Dekker, promotor,
Prof.dr.ir. A.W. Heemink, promotor,
Dr.ir. H.X. Lin,
Prof. R.K. Price,

Prof.dr.ir. H.J. Sips,
Prof.dr.ir. G.S. Stelling,
Dr.ir. Th.L. van Stijn,

Technische Universiteit Delft
Technische Universiteit Delft
Technische Universiteit Delft
International Institute for Infrastructural,
Hydraulic and Environmental Engineering
Technische Universiteit Delft
Technische Universiteit Delft
Rijkswaterstaat/RIKZ

Copyright ©1997 by M.R.T. Roest

Roest, Mark Roy Taco

Partitioning for Parallel Finite Difference Computations in Coastal Water Simulation /
Thesis Technische Universiteit Delft. - With index, ref. -
With summary in Dutch and English.
ISBN 90-9010869-6
NUGI 811
Subject headings: parallel computing / shallow water simulation / partitioning.
Preface

This thesis presents the results of a PhD-study on partitioning for parallel finite difference computations in coastal water simulation. The research was initiated by Rijkswaterstaat/RIKZ (the Dutch National Institute for Coastal and Marine Management), ICIM B.V. and Delft University of Technology. The aim of the research was to study the potential of parallel computing for large scale simulation models of coastal areas.

Such models are used in the Netherlands (and abroad) for a large variety of applications, including storm surge forecasting and the assessment of the consequences of proposed construction works along the coast. When very high resolution results have to be obtained from such models, the computing time becomes unacceptably long. Parallel computing has the potential of reducing this computing time, but unfortunately, most of the implementations of simulation models that are in use today are not suitable for parallel computing. The results that are presented in this thesis provide indications for the solution of several of the problems that arise in modifying the implementations such that they can effectively run on parallel computing platforms.

The research that is presented in this thesis has greatly benefited from the continuous support of Rijkswaterstaat/RIKZ and ICIM B.V. They provided the feedback that was needed to indicate the direction of research, and the software and models to validate the results. The regular meetings with Theo van Stijn, Johan Dijkzeul, Arnold Heemink, Len Dekker and my colleagues from the Parallel Algorithms Group have been a source of inspiration throughout the past four years.

Apart from this, I am much indebted for the pleasant atmosphere at work that was due to a large extend to my colleagues. In the first place, I would like to thank Edwin Vollebregt for putting up with me all these years and hopefully some years to come. The discussions that we had, were indispensable for shaping my thoughts. Similarly, I will retain pleasant memories of the discussions with Eric ten Cate, which led us more than once into the vast blue universes of abstraction. Finally, I thank HaiXiang Lin for his valuable input for my research (and his help in getting me through China) and Kees Lemmens for our interesting talks on history and physics and for keeping my computer running.

Apart from these people, I have received a lot of support from other people as well, for which I am truly grateful. First of all, there is of course my wife, Jorien Burger, who had to miss me much too often. I thank her for being there and for putting my feet back on the ground once in a while. Furthermore, I would like to thank my parents for giving me the possibilities to do what I wanted to do.
Acknowledgements

The research that is presented in this thesis has been supported by the North-West European Shelf Project of the EC MAST-II programme and has benefited from the MMarie concerted action in the same EC programme.

I gratefully thank the institutes that provided access to parallel computing systems: the Interdisciplinary Center for Complex Computer facilities IC^3^A in Amsterdam, the Dutch National foundation for Computer Facilities NCF, the numerical mathematics group of Delft University of Technology and the High Performance Applied Computing center Hpac of Delft University of Technology.
List of Symbols

The symbols that are given below are those that are used at several places in the thesis. Symbols that are only used once or in a small part of the thesis are not included. The symbols have been chosen in accordance with common practice. This has made it necessary to sometimes use rather similar symbols for very different purposes in cases where the thesis uses results from several fields of research that all use the same symbol for a different purpose.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B$</td>
<td>workload distribution balance (see page 91)</td>
</tr>
<tr>
<td>$C$</td>
<td>a set of computations; a set of nodes in a graph</td>
</tr>
<tr>
<td>$C_s$</td>
<td>a separator of a graph (see page 48)</td>
</tr>
<tr>
<td>$Cost(.)$</td>
<td>communication costs associated with a partitioning (see page 27)</td>
</tr>
<tr>
<td>$dB$</td>
<td>improvement of workload distribution balance (see Equation 4.10 on page 91)</td>
</tr>
<tr>
<td>$D$</td>
<td>dimension</td>
</tr>
<tr>
<td>$Dl$</td>
<td>Dilation (see Definition 9 on page 55)</td>
</tr>
<tr>
<td>$\mathcal{E}$</td>
<td>a set of edges</td>
</tr>
<tr>
<td>$\mathcal{E}_c$</td>
<td>the cutset of a partitioning (see page 48)</td>
</tr>
<tr>
<td>$f$</td>
<td>the number of a cycle in cyclic blockwise distribution</td>
</tr>
<tr>
<td>$f(.)$</td>
<td>a continuous function (see Equation 5.1 on page 111)</td>
</tr>
<tr>
<td>$F$</td>
<td>the cycle frequency (see page 30), i.e. the number of blocks per processor under cyclic distribution</td>
</tr>
<tr>
<td>$F(.)$</td>
<td>a represented discrete function (see page 125)</td>
</tr>
<tr>
<td>$F(.)$</td>
<td>a discrete function (see page 113)</td>
</tr>
<tr>
<td>$\mathcal{F}(.)$</td>
<td>Fourier transform of a function</td>
</tr>
<tr>
<td>$g(.)$</td>
<td>a continuous function (see Equation 5.2 on page 111)</td>
</tr>
<tr>
<td>$g(.)$</td>
<td>a represented discrete function (see page 128)</td>
</tr>
<tr>
<td>$G(.)$</td>
<td>a discrete function (see page 116)</td>
</tr>
<tr>
<td>$G$</td>
<td>a graph</td>
</tr>
<tr>
<td>$GlobalNr_p(.)$</td>
<td>function that associates a local item-number with global number of that same item (see Equation 5.6 on page 5.6)</td>
</tr>
<tr>
<td>Symbol</td>
<td>Definition</td>
</tr>
<tr>
<td>--------</td>
<td>------------</td>
</tr>
<tr>
<td>$h(.)$</td>
<td>topography (see page 81)</td>
</tr>
<tr>
<td>$H$</td>
<td>the parameter in the Weierstrass-Mandelbrot fractal function (see Equation 4.6 on page 83)</td>
</tr>
<tr>
<td>$I$</td>
<td>an index set (see page 23)</td>
</tr>
<tr>
<td>$I_{p_1,p_2}$</td>
<td>the interface from $p_1$ to $p_2$ (see Definition 6 on page 26)</td>
</tr>
<tr>
<td>$L_p$</td>
<td>the workload on a processor (see page 25)</td>
</tr>
<tr>
<td>$\overline{L}$</td>
<td>the mean workload per processor (see page 25)</td>
</tr>
<tr>
<td>$N$</td>
<td>the size of a workload function domain, number of vertices in a graph</td>
</tr>
<tr>
<td>$Nb$</td>
<td>number of neighbours of a part (see page 54)</td>
</tr>
<tr>
<td>$p$</td>
<td>a processor number</td>
</tr>
<tr>
<td>$P$</td>
<td>number of processors</td>
</tr>
<tr>
<td>$\mathcal{P}(.)$</td>
<td>a partitioning of a set (see e.g. page 25)</td>
</tr>
<tr>
<td>$q$</td>
<td>an offset vector (see Definition 5 on page 26)</td>
</tr>
<tr>
<td>$Q$</td>
<td>a stencil (see Definition 5 on page 26)</td>
</tr>
<tr>
<td>$s$</td>
<td>a difference vector</td>
</tr>
<tr>
<td>$Sk$</td>
<td>skewness (see Definition 7 on page 47)</td>
</tr>
<tr>
<td>$SL$</td>
<td>sea level (see page 89)</td>
</tr>
<tr>
<td>$V$</td>
<td>amount of variation in a workload function (see Theorem 14 on page 73)</td>
</tr>
<tr>
<td>$W(.)$</td>
<td>a workload function (see Definition 4 on page 23)</td>
</tr>
<tr>
<td>$Z(.)$</td>
<td>a random process (see page 81)</td>
</tr>
<tr>
<td>$\gamma(.)$</td>
<td>semivariogram (see Equation 4.3 on page 82)</td>
</tr>
<tr>
<td>$\partial\Omega$</td>
<td>the boundary of a continuous spatial domain</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>a zero-mean second-order stationary random process</td>
</tr>
<tr>
<td>$\phi$</td>
<td>phase of a Fourier coefficient, also: a phase of a program (see Definition 3 on page 21)</td>
</tr>
<tr>
<td>$\iota$</td>
<td>the imaginary unit</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>an eigenvalue of a matrix</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>standard deviation of a stationary random process</td>
</tr>
<tr>
<td>$\nu$</td>
<td>amplitude of a Fourier coefficient</td>
</tr>
<tr>
<td>$\omega$</td>
<td>(spatial) frequency</td>
</tr>
<tr>
<td>$\Omega$</td>
<td>a continuous spatial domain</td>
</tr>
</tbody>
</table>
Contents

Preface ................................................. 3
List of Symbols ....................................... 5

1 Introduction ....................................... 11
   1.1 Parallel Processing versus Sequential Processing .................................. 11
   1.2 The Partitioning Problem in the Field of Parallel Processing .................. 13
   1.3 Parallel Coastal Water Simulation ....................................................... 13
   1.4 This Thesis ........................................ 14

2 Partitioning for Parallel Processing; The State of the Technology ............. 17
   2.1 Definitions ........................................................................ 18
      2.1.1 Definition of the Workload Function ........................................ 19
      2.1.2 Stencils and Interfaces ....................................................... 25
      2.1.3 The Distribution Problem .................................................... 27
   2.2 Heuristics for Solving the Distribution Problem .................................. 30
      2.2.1 Regular Distribution Methods ................................................ 30
      2.2.2 Variable Block Methods ....................................................... 32
      2.2.3 Clustering Methods ............................................................ 35
      2.2.4 Optimisation Methods ......................................................... 35
      2.2.5 Comparing distribution methods ............................................ 37
   2.3 The Effectiveness of Distribution Methods .......................................... 38
      2.3.1 Regular Methods ............................................................... 38
      2.3.2 Variable Block Methods ....................................................... 47
      2.3.3 Clustering Methods ............................................................ 57
   2.4 Implementation Aspects ................................................................. 57
      2.4.1 Software Support for Regular Partitionings ................................ 58
      2.4.2 Software Support for Variable Block Partitionings ...................... 59
      2.4.3 Software Support for Cluster Partitionings ................................ 60

3 Analysis of Partitioning Methods ......................................................... 61
   3.1 Overview ........................................................................... 61
   3.2 Analysis of (Cyclic) Blockwise Distribution ........................................ 62
      3.2.1 Cyclic Blockwise Distribution ................................................ 62
      3.2.2 Maximum Workload for trigonometric workload functions ............ 63
Contents

Summary 163
Samenvatting 165
Curriculum Vitae 167
1.1 Parallel Processing versus Sequential Processing

In the few decades since the first computers were built, computer technology has gone through a tremendous development. The first computers were large machines, that could hardly compute reliably for more than a few seconds and that could be programmed only by a team of specially trained people who had to connect the proper contacts by plugging wires. Nowadays, computers are small enough to fit in children’s toys and satellites and they can be programmed by almost everyone. But in spite of their significant differences, almost all computers that were ever built were similar in that they consisted of a single processing unit connected to a single memory in which both the program and the data are stored. This type of computer is usually referred to as a sequential computer, because the single processing unit performs its tasks sequentially, one after the other.

Somewhat apart from this line of development, another type of computer has existed ever since the first computers were built. This other type of computer, called a parallel computer, is characterised by the fact that it consists of multiple processing units. When the processors all use a single memory, the computer is called a shared memory parallel computer; if each processor has its own, private memory, the computer is called a distributed memory parallel computer. The fact that parallel computers have never been anywhere near as successful as their sequential counterparts is probably best explained by the inherent difficulty of programming them. Whereas a programmer can easily learn to program a single processing unit, it proves to be difficult for people to effectively program a collection of processing units.

In recent years though, the balance seems to be shifting from sequential computers to parallel computers. It becomes more and more difficult to further increase the speed of sequential computers whereas an increase of speed can readily be achieved by exploiting parallelism. This can be done by using multiple processors that cooperate on a task, or by using processors that can perform several computations simultaneously. In fact, all present-day supercomputers apply some form of parallelism to achieve their high computing speed.

But the increasing use of parallelism comes not only from developments in computer architecture: the use of computer networks gives a similar impulse to parallel computing. Such networks of computers are effectively parallel computers, that are capable of serving many users at moderate computing speed or providing high computing speed for a single
user when needed. When a company needs high computing speed only occasionally, it may be worthwhile to use the existing network of computers to deliver the computing speed instead of buying an expensive high-speed sequential computer.

But even when it is acknowledged that parallel computers have significant benefits for a particular application, the fact remains that virtually all presently existing applications have been programmed for sequential computers and it is usually difficult to make these programs suitable for a parallel computer. Tools exist that can automatically transform a sequential program into a parallel program, but these tools are not always effective. They fail, for example, when the problem formulation that is implemented in the program is inherently sequential, or when the data structures that are used in a code can not easily be distributed across distributed memory, or when the program is too complex to be handled by the tool. For all these reasons, it is usually necessary to re-program an application or at least modify its implementation to make it run on a parallel computer.

The best way of reprogramming is probably to start right from the beginning again, by considering the problem formulation and trying to reformulate it so that an effective parallel method can be used to solve it. But even when such an improved formulation can be found, the task of implementing the application still remains to be done. Noting that present-day codes for industrial applications usually consist of several 100,000 lines of code and have required many man-years to become as reliable as they are today, it is immediately obvious that reprogramming an application is often not a viable option. The existence of legacy codes is probably one of the main inhibitors for the acceptance of parallel computing as a standard technique.

To make the potentially large computing power of parallel computers more readily available for practical applications, it should become easier to program this kind of computers and to port legacy codes to them. This thesis tries to contribute to such a development by studying the partitioning problem, which is one of the fundamental problems in parallel processing. This problem occurs, for example, in distributing data over the local memories of the processors, or when allocating computations to processors.

Because the partitioning problem plays such a central role in parallel computing, it has received a lot of attention in the past. Many solution methods have been proposed, some of which have stood the test of time and are now widely used, whereas others have hardly been used at all. Since so many solution methods are already known, this thesis will not try to find new ones. Instead, the emphasis will be on exploring how effective the known methods are, and under which circumstances they fail. Furthermore, a basic technology will be proposed to use the existing partitioning methods also in the context of legacy codes in which data structures prohibit a straightforward application of partitioning methods.

In the next section, the partitioning problem will be discussed to an extend that is sufficient for anyone who is new to the subject to understand the main aspects of the problem. Then, an introduction will be given to the application of parallel computing that will be considered in this thesis: coastal water simulation. This chapter will be concluded with a short preview of the contents of this thesis.
1.2 The Partitioning Problem in the Field of Parallel Processing

One aspect of programming for a parallel computer is that the operations that constitute an application algorithm must be properly assigned to the processors. This is a crucial step; when done incorrectly the parallel program will fail to utilise the parallel computer effectively and the processing will proceed much slower than potentially possible. There are two main conditions that must be fulfilled by the assignment: every processor must get an equal share of the work to be done and the amount of data that is produced by one processor but needed on another should be kept to a minimum. If not all processors get an equal share of the work, then some processors will remain idle during part of the processing and their potential computing capability will be wasted. When too much data must be transferred from one processor to another, this will keep the processing from proceeding, again wasting potential computing power.

This assignment problem can be formulated as a partitioning problem: given the set of computations to be performed, and the relations between them, a partitioning of the set of computations must be found such that a minimum of computing power is wasted when each part is handled by a different processor. This problem can usually not be solved exactly within an acceptable times-span. Therefore, in practice, one has to settle for less than the optimal partitioning and just try to find one that comes close. Over the years, many different methods have been developed to find near-optimal solutions in an acceptable time. (Many of these partitioning methods are actually referred to as distribution methods. Although there is admittedly a conceptual difference between these two terms, they will be used interchangeably in this thesis to avoid unnatural formulations.)

At this moment, choosing the correct partitioning method to solve a given problem is still much of a craft. Theoretical knowledge on the effectiveness of partitioning methods is only sparsely available and hardly useful in practical situations. Empirical results have been widely published, but their applicability is limited by the fact that there is only a superficial understanding of the characteristics of partitioning problems that determine the success or failure of a partitioning method. Therefore, even in the rare cases where authors use the same test-case to evaluate their partitioning method, there is no reason to assume that this test-case is representative of all practical partitioning problems as it is impossible to tell what the relevant characteristics are that should be included in the test-case. This thesis makes an effort at improving this situation through extensive theoretical and empirical study.

1.3 Parallel Coastal Water Simulation

The work that will be presented in this thesis has been performed in the context of an effort to parallelise a major package for the simulation of water flow and transport processes in coastal areas. The package is the TRIWAQ package, which is extensively used for coastal and marine management in the Netherlands. It has been developed over the past twenty years and has become an indispensable tool for such diverse applications as predicting storm surges, controlling pollution at sea and assessing the consequences of proposed construction works along the coast.
Chapter 1. Introduction

For most of these applications, it is desirable to have high resolution simulations of flow and transport in coastal areas. Because TRIWAQ is based on a finite difference discretization of the shallow water equations, such a high resolution requires a relatively fine discretization in the horizontal plane and in the order of 10 layers for depth resolution. At the same time, it is sometimes necessary to simulate long time-periods with a rather small time-step. This makes TRIWAQ in many cases a very compute intensive application.

Unfortunately, its long history and the nature of the application have also given TRIWAQ many of the qualities that make it difficult to port to a parallel computing environment: it is large, it uses inherently sequential numerical methods, it relies on data structures that defy a simple distribution of data across distributed memory, and it poses a non-trivial partitioning problem.

The way to deal with the sequential methods is extensively discussed by Vollebregt [205]. The data distribution and partitioning problems are the subject of this thesis. TRIWAQ is used on many different grids, and these grids often take quite a complex shape. Also, most of the users of TRIWAQ are not willing to determine the partitioning themselves: they are merely interested in the results they obtain with TRIWAQ. Furthermore, the number of processors on which the parallel version of TRIWAQ is to be run, is not known a priori, but may be determined on the fly. All this makes it necessary to devise automatic support for partitioning the grids that are used with TRIWAQ.

A further complication in the parallelisation of TRIWAQ is that the amount of computational work in each grid point is not necessarily fixed: in coastal areas, large parts may become dry at low tide and be flooded at high tide. Since no computations have to be done in 'dry' grid points, there may be large parts of the grid that are not actually used in a part of the simulation. Adding to this the complexity of TRIWAQ's data structures and the wish to run parallel TRIWAQ also on networks of workstations, where communication is relatively slow, completes the picture of a complicated partitioning problem.

1.4 This Thesis

This thesis presents the results of a study on the effectiveness of partitioning methods and their use in practical situations, where also indirect addressing can play a significant role.

First of all, in Chapter 2, the partitioning problem will be formalised and the current volume of literature on the partitioning problem will be surveyed to get a clear view of the current state of knowledge. This will reveal that in some respects, the knowledge is quite extensive and complete. At other points, there are theoretical issues that are not yet resolved. But mostly, the survey of literature will show that a major part of the field of partitioning for parallel processing is still unexplored: there is little knowledge on the issue of which partitioning method is best suited for a particular problem.

Having noted this, the thesis then proceeds to extend the theoretical and practical understanding of partitioning methods in the subsequent two chapters. In Chapter 3 a new and more complete theory on the effectiveness of one widely-used partitioning method, Cyclic Blockwise Distribution, will be developed. Then, in the same chapter, an analytical study will be made of the effectiveness of another common partitioning method, Recursive
1.4. This Thesis

Coordinate Bisection. Such a study has not been made before and is therefore the first to provide any formal understanding of the effectiveness of the method.

The theoretical analyses from Chapter 3 will be validated in Chapter 4 with an empirical study of the effectiveness of the two methods for one particular application. Using the theoretical results, it is possible to understand the results of the empirical study and to see how they might carry over to other applications. The application that is used for this study is the simulation of flow and transport processes in coastal water, which was introduced briefly in the previous section. This particular application will prove to be a very interesting one because it contains several features that make it rather complex from the partitioning point of view.

Apart from the lack of insight in the effectiveness of partitioning methods, programming for a parallel computer is made difficult by the fact that even when a programmer can make an motivated choice for a method to distribute the computations, then it may still not be easy to distribute all the necessary data so that each computation can readily access the data that it needs. This is mainly caused by the fact that the data may be structured in a way that does not correspond neatly with the way in which the data is used by the computations. This holds in particular when indirect addressing schemes are used, and precisely this use of indirect addressing is currently seen as a major problem for automatic distribution of data.

Chapter 5 will discuss one way of dealing with this issue. In many cases, the distribution of the variables in the abstract specification of a program is relatively easy to determine. Problems arise when these abstract variables are not directly represented in the program code but are instead molded into a form that does no longer immediately reflect the original properties of the variable. For example, a sparse matrix is never stored completely but only its non-zero entries are stored and indirect addressing is used to mark the place of the entries in the original matrix. Even when the distribution of the matrix is readily specified, the distribution of its sparse representation can still pose problems.

The basic approach of the work that is presented in Chapter 5 is to formally describe the representation of abstract variables and then, using a specification of the distribution of the abstract variables, deriving the distribution of their representation in the code. This formal approach has made it possible to develop an automatic tool that restructures the data in a code so that it can easily and optimally be distributed. But even apart from this immediate application, the study in Chapter 5 provides an improved insight in the nature of indirect addressing which may prove to be very useful in developing other support for automatic parallelisation of existing codes.

The thesis will be concluded with a brief overview of the major results and an outline of how further research can provide assistance for programmers in solving the partitioning problem and related issues. All this will be the subject of Chapter 6.
Chapter 2
Partitioning for Parallel Processing; The State of the Technology

The ultimate reason for parallel computing is to get results quicker than with sequential computing. This is done by distributing the computational work over multiple processors. If the computations are completely independent of each other, then it is easy to find a distribution of computations over the processors such that the speedup over sequential processing is equal to the number of processors that are applied.

However, computations are rarely completely independent. Computations may need data that is computed by other computations. If a computation that requires a certain data element is performed on a different processor than the one that computes it, then the data element must be communicated from one processor to another. Such communication constitutes an overhead which must be kept to a minimum to obtain optimal speedup from parallelisation. Thus, the programmer is faced with an optimisation problem: distributing the computations as evenly as possible over the processors and at the same time keeping the number of dependencies between computations on different processors to a minimum. This problem will be denoted as the distribution problem.

In reality, the distribution problem is more complex than just described: some communications may be better supported by the parallel hardware than others and the dependencies between computations may depend on the distribution that is chosen. In general, there is no way to solve the distribution problem exactly. Formulated as a graph partitioning problem, it can be shown to be NP-complete (see e.g. Garey et al. [75]). As the number of computations that must be distributed, i.e. the number of vertices in the graph to be partitioned, is almost always very large, traditional optimisation methods are not practical to find an optimal distribution of computations.

Therefore, the problem is handled through heuristics. Choosing the proper heuristic for a problem and setting the correct parameters is as yet still much of an art rather than a technology. A lot of empirical results have been published, but the usefulness of these results is restricted by the fact that there is little understanding of the reasons for the observed behaviour of partitioning heuristics. Furthermore, most reports on experimental studies of distribution methods use their own set of test problems which makes it impossible to compare results from different authors.

However, there is, scattered over literature, quite a volume of theoretical results that...
can make at least some of the experimental results understandable. But as these results have never been compiled into a single overview together with empirical results, it is rather difficult to get a complete picture of the state of knowledge in this field. This chapter will be an effort to give such an overview, and therewith show that the theoretical knowledge on the effectiveness of distribution methods is indeed extensive. At the same time, it will reveal where the gaps in this knowledge are and may indicate lines of research that can be pursued to fill them in. The aim here is to provide an overview and not a detailed discussion of theory and practice of partitioning for parallel processing. Therefore, the reader will be referred to literature for more details, except for the cases where those details are relevant for the current discussion.

Before giving the overview of knowledge on partitioning methods, first the main concepts that will be used in the overview and in the remainder of this thesis will be discussed in Section 2.1. Then, in Section 2.2, a brief introduction to existing partitioning heuristics will be given which mainly serves to prepare for the Sections 2.3.1-2.3.3, where the theoretical and empirical results that are known for each of the partitioning methods will be reviewed. The last section of this chapter will deal with implementation aspects to show how the heuristics are used in practice.

2.1 Definitions

As already mentioned in the introduction, the distribution problem for parallel processing is the problem of assigning computations to processors such that every processor gets an equal share of the work and such that computations that need data from each other are mostly assigned to the same processor. Thus, the main entities in this problem are a specification of the amount of work associated with each of the computations and a specification of the relations between computations. This section will provide a formal definition of these entities and introduce a notation to work with them.

At the basis of the discussion in this section is the notion of a program as a set of computations. Each computation involves an amount of computational work, is possibly dependent on other computations and produces results that are used by yet other computations.

The computations are numbered in a particular way, so that their distribution can be represented by the distribution of their numbers. Also, this numbering makes it possible to represent the computational work that is associated with each computation by a function from the set of computation numbers to the set of amounts of computational work. This function will be called the workload function and will play a central role in this thesis. A complete discussion of the concept and the necessary definitions will be given in Section 2.1.1.

The set of computations on which a given computation is dependent will be defined as its stencil. The collection of all computations on a given processor that depend on computations on other processors, will be called the interface of the set of computations on that processor. Section 2.1.2 will formalise these concepts.

After the main entities (i.e. the workload function, stencils and interfaces) in the formal
2.1. Definitions

All definitions in this section will be illustrated with a simple example, which is the code fragment in Figure 2.1. It is a simple form of the Gaussian elimination algorithm for solving a system of linear equations of the form:

\[ A \tilde{x} = \tilde{b} \]

where \( \tilde{x} \in \mathbb{R}^N \) is the vector of unknowns, \( A \) is an \( N \times N \) matrix and \( \tilde{b} \in \mathbb{R}^N \).

The code fragment in Figure 2.1 is written in HPF version 1.1. The main constructs used in the code will be briefly explained below; a complete description of the language can be found in the HPF language specification [100].

The matrix \( A \) is stored in the \( N \times N \) array \( A \) and \( \tilde{b} \) is stored in the one-dimensional array \( b \) (see line 2 of the code). Line 3 defines the (abstract) parallel computer as a one-dimensional array of \( P \) processors, where \( P \) is assumed to be a constant. For the sake of argument it will be assumed that \( P \) divides \( N \). The array \( b \) is distributed in line 4 over these processors in a block-wise fashion, i.e. blocks of contiguous elements are allocated to each processor, each block consisting of \( N/P \) elements. In line 5, the rows of array \( A \) are aligned with the elements of \( b \), i.e. the processor that owns element \( i \) of \( b \) also owns the entire row \( i \) of array \( A \). Because array \( b \) is distributed block-wise, the alignment causes a distribution of array \( A \) in which blocks of contiguous rows are assigned to each processor.

The actual computations start in line 6, which opens a loop on variable \( k \), i.e. over the rows of the matrix \( A \). In each pass over the loop, first the \( k \)-th column of \( A \) is divided by the pivot (the element \( A(k,k) \)). These operations can be done in parallel, which is indicated by the \texttt{forall} construct that is used here instead of a \texttt{do} construct. When the \( k \)-th column of \( A \) has been updated in this way, a loop is performed over all rows of \( A \) after the \( k \)-th row, in which the last \( N-k \) elements of each row are modified (line 9) and in which the element of \( b \) that corresponds to that row is updated (line 10). The rows can be processed in parallel and the same is true for the elements within each row. Again, this is indicated by the use of the \texttt{forall} construct.

2.1.1 Definition of the Workload Function

1) Programs, Computations and Dependencies

Workload functions will be used to quantitatively characterise the computational work in a program. To properly define the concept of workload functions, it is necessary to first consider what is meant by a \textit{program}. This will be done in terms of what will be considered the elementary parts of every program: the \textit{computations}. Whatever constitutes a computation in a program is not really important: it can be a single statement in a single assignment program or a group of statements or even entire subroutines. The only requirement on a computation as used in this thesis is that it must be atomic, i.e. it can only interact with other computations at its start and at its end:
subroutine Gauss(A,b)

real A(N,N),b(N)

processors procs(P)

distribute (block) onto procs :: b

align A(i,j) with b(i)

do k = 1,N-1

forall (i = k+1:N) A(i,k) = A(i,k)/A(k,k)

forall (i = k+1:N)

forall(j = k+1:N) A(i,j) = A(i,j) - A(i,k)*A(k,j)

b(i) = b(i) - A(i,k)*b(k)

end forall

do

end subroutine

Figure 2.1: The Gaussian Elimination subroutine. See the text for a discussion.

Definition 1 A computation is a collection of operations on a set of operands such that the operands are not externally accessible during the execution of the computation and such that the set of operations is fully determined at the start of the computation. Furthermore, a computation has a computational weight that represents the computational effort that is needed to perform the operations.

This definition leaves some freedom in how one would represent a given program in terms of computations. For example, one might consider the entire subroutine in Figure 2.1 to be a single computation (a coarse grain view), or one might view each pass through the k-loop (lines 6 through 12) as a computation (medium grain view) or one can consider every assignment to be a computation (fine grain view). Which view is taken will depend on the context: when talking about a large program in which the subroutine of Figure 2.1 is only a minor part, a coarse grain view may be appropriate, in other cases a more fine grain view will be better.

Although Definition 1 may seem quite formal and precise, it is in fact still rather loose. It does not specify what an operation is or what constitutes an operand, nor does it specify how the computational weight must be determined. All these terms will be assumed to have an obvious meaning. Also, it will be assumed that each computation is executed only once; e.g.: each pass through a loop can be considered to be new computation.

In general, a program consists of a set of computations. Computations in a program are usually dependent on other computations. For example, in Figure 2.1, the statement in line 7 for a certain value of k must have been executed prior to the execution of the loop in lines 8 through 11 for that value of k. Thus the set of computations that constitute a program is at least partially ordered. This partial ordering induces a graph on the set of computations, which will be called the precedence graph. The precedence graph of the
2.1. Definitions

Gaussian elimination program in Figure 2.1 is shown in Figure 2.2 for the case that \( N = 3 \). Thus, a program can be represented by its precedence graph. De Keyser and Roose [115] have given the following definition of a program, which will be used in this thesis:

**Definition 2** A program is a set, \( C \), of computations equipped with a partial ordering (denoted by \( E \)) that represents the data dependencies between them. The acyclic directed graph \( (C, E) \) is known as the precedence graph. It can be assumed that it has a unique minimum (a begin computation).

Note that if the precedence graph does not have a unique begin computation, then a unique begin computation can be created by extending the set of computations with a dummy computation that is ordered before the initial computations of the original precedence graph.

II) Phases

Most of the work that is presented in this thesis deals with the study of the distribution of workload across processors when using a particular way of distributing the computations. The dependencies between computations in a program are quite troublesome in this context. Consider for example two computations, both representing the same amount of computational work. Let the second computation depend on the first. When both are assigned to a separate processor, the computational work is optimally distributed but still only one processor can process at a time due to the dependencies. So, the processing time will be no smaller than if both computations had been assigned to the same processor and the statement about optimally distributed work is not very useful.

Therefore, to allow for a meaningful study of distribution methods, only a single phase of a program will be considered, where a phase consists of computations that are not dependent on each other and can therefore be processed in parallel. For instance, in the example code of Figure 2.1, line 7 constitutes a phase: all computations in the forall-loop on that line can be processed independently of each other. Likewise, the forall-loop in lines 8 through 11 is a phase. The following definition gives a more precise account of what constitutes a phase:

**Definition 3** The set of computations in a precedence graph consists of non-overlapping subsets, called phases, each with a unique number in \( N \). A phase with number \( \phi \) consists of computations that are only dependent on computations in phases with a number \( \psi \), such that \( \psi < \phi \).

Since the computations in a phase are independent of each other, the way in which they are distributed determines the time that is needed to process all computations in a phase. By restricting the study of distribution methods to their effect on computations within a phase, it is possible to make meaningful statements about how effectively distribution methods distribute computational work without being bothered by dependencies.

A division of a program into phases can usually be done in different ways; any division of the computations into phases that meets Definition 3 is valid. For instance, in the Gaussian elimination example (Figure 2.1), the loop in lines 8 through 11 is a phase, but it
can also be considered to be a sequence of $N-k$ phases each consisting of the computations in lines 9 and 10. A possible phase decomposition of the entire program of this example is indicated in Figure 2.2. Phase zero is the begin computation, which can be understood here to comprise everything before line 6. Phase 1 consists of the execution of all computations in line 7 in the first pass through the $k$-loop (lines 6 through 12). Phase 2 consists of the execution of all computations in the loop on lines 8 through 11, again for $k=1$. After this phase, a new pass through the $k$-loop starts, again with executing all computations on line 7, this time for $k=2$ (phase 3), etc.

The indeterminism in the definition of phases here differentiates it from the concept of phases as are used by other authors. For instance, a rather common way of dividing a program into phases is by collecting into phase $\phi$ all the computations that are located at a distance $\phi$ from the begin computation in the ordering induced by the data dependencies. This is the approach that is taken by e.g. De Keyser and Roose [115] who use it to
2.1. Definitions

delimit sets of operations in a program to which they apply their dynamic load balancing algorithm. The same division into phases is used by Feautrier [66] (who calls his type of phases “fronts”) and Saltz [179]. Here, the definition of phases is kept somewhat looser: it is unnecessary in the current context to put more restrictions on phases than is done by Definition 3.

III) Computation Numbers and the Workload Function

Now, consider the set of computations in a given phase $\phi$ and let these computations be numbered. This numbering can be done in any convenient way. For example, in the loop in line 7 of Figure 2.1 it seems natural to number the computations $A(i,k) = A(i,k)/A(k,k)$ by their loop index $i$, running from $k+1$ through $N$. This numbering can be seen as assigning a coordinate to each computation. Often, the coordinate (i.e. the number) will be taken from $\mathbb{N}$, but it may be useful in some cases to take the coordinate from $\mathbb{N}^2$, $\mathbb{N}^3$ or, generally, from $\mathbb{N}^n$. A few more remarks on this will be made below. If a numbering is done for each phase in a program, then this results in a numbering of all the computations. The set of numbers of all computations will be denoted by $I$, the set of numbers of computations within phase $\phi$ will be denoted by $I_\phi$.

The sets $I$ and $I_\phi$ are sets of numbers that index the set of computations. Therefore, these sets will be called index sets in this thesis.

When defining computations above, (see Definition 1) it was stated that each computation has associated with it an amount of computational work. With the numbering of computations just described, the computational work of computations in a phase can be represented by a function, which will be called the Workload Function. This function is defined as follows:

**Definition 4** For phase $\phi$ of a program, and a given numbering of the computations within that phase, let $I_\phi$ be the set of numbers each associated with a computation in $\phi$ and let $i$ be the number (coordinate) of a computation in $\phi$, then the **workload function** for phase $\phi$ is the function

$$W_\phi : I_\phi \rightarrow \mathbb{R}$$

$$W_\phi(i) = \text{computational weight of computation } i \text{ in phase } \phi.$$  

Note that the workload function is defined per phase and not for an entire program. Consequently, in the remainder of this thesis, when speaking about a workload function this will imply a workload function for a single phase, where the subscript $\phi$ will be often omitted. Also, in many cases, the domain of the workload function, $I_\phi$, will be taken somewhat larger than just the set of numbers associated with computations. For example, in the loop in line 7 of Figure 2.1, the computations can numbered from $k+1$ through $N$, but it will seen to be convenient below to have the domain of the workload function run from 1 through $N$. Points in the domain that are not associated with computations will have a workload function value of zero. Taking $I_\phi$ somewhat larger implies that also $I$, the set of all computation numbers in the precedence graph, is larger than the set of computations itself.
For the example of the Gaussian elimination program in Figure 2.1 with the phase-decomposition as shown in Figure 2.2, the workload functions for the first three phases is shown in Figure 2.3. Here it is assumed that a computation is numbered by number \((\phi, i)\) if it assigns a value to either \(A(i,j)\) or \(B(i)\) during phase \(\phi\). Also, it is assumed that additions, multiplications and divisions all represent a unit amount of computational work. Thus, in phase 1, where \(k=1\) and \(A(i,k) = A(i,k)/A(k,k)\) for \(i=k+1:N\), the workload in point 2 (i.e. \(i=2\): the assignment to \(A(2,1)\)) equals one because one division is made in that computation.

The work in this thesis will mostly deal with workload functions on one-dimensional domains. This is because they are the easiest to handle analytically, although the methods of analysis can be extended to workload functions on higher-dimensional domains. Alternatively, the results for one-dimensional domains can be applied to workload functions on higher-dimensional domains by considering the function along one dimension only. In case of a workload function \(W(.)\) on a two-dimensional domain, the workload function along the first dimension, (denoted by \(W'(.)\)) can be constructed by taking the sum along the second dimension:

\[
W'_\phi(i_1) = \sum_{i_2=1}^{N} W_\phi(i_1, i_2).
\]

where \(N\) is the size of the domain in the second dimension. If the dimension of the domain of \(W(.)\) is higher than 2, then a workload function \(W'(.)\) along one dimension can likewise be obtained by taking the sum over all other dimensions.

IV) Partitionings

For parallel computing, the set of computations in a program, \(C\), is divided, or \textit{partitioned} into a number of subsets. Let these subsets be numbered by \(1, \ldots, P\), where \(P\) is the number of subsets. This induces a partitioning of a part of the set of computation numbers, \(I\). When every point in the domain of the workload function is associated with a computation, then it makes no difference whether either the set of computations or the domain of the workload function is partitioned. Otherwise, the partitioning of the domain implies a partitioning...
2.1. Definitions

of the computations but the reverse does not hold. It will prove convenient to define a partitioning on the domain of the workload function rather than on the computations themselves.

Each part of $\mathcal{I}$ under a given partitioning will be denoted by $\mathcal{I}_p$. The partitioning will be denoted by $\mathcal{P}(\cdot) : \mathcal{I} \rightarrow [1, P]$. Thus:

$$\mathcal{I}_p = \{i \in \mathcal{I} \mid \mathcal{P}(i) = p\}$$

The partitioning also leads to a division of the numbers of the computations in each phase $\phi$ into subsets $\mathcal{I}_{p,\phi}$:

$$\mathcal{I}_{p,\phi} = \{i \in \mathcal{I}_\phi \mid \mathcal{P}(i) = p\}$$

The total amount of computational work in $\mathcal{I}_{p,\phi}$ is:

$$L_{p,\phi} = \sum_{i \in \mathcal{I}_{p,\phi}} W_\phi(i)$$

The average over $L_{p,\phi}$ over all $p$ will be denoted by $\overline{L_\phi}$.

V) Concluding Remarks

The workload function is usually only characteristic for a (very) small part of a program, like a single pass through a loop. The number of workload functions in a program can be enormous because of the large number of phases. The aim of the studies in the next chapters is therefore not to find the perfect distribution method for each workload function, but rather to find characteristics of workload functions that make it unsuitable for a particular way of distributing. This will make it possible to quickly identify the locations in a program where performance will be lost due to an unsuitable choice of distribution method.

One issue that needs to be addressed here is whether the computational weight is always determinable in a location of the workload function domain. Obviously, if there are conditional statements in a phase, the weight will depend on whether the condition is true or false, which is usually unknown. In this case the branching probabilities can be used to approximate the workload function value. Alternatively, one may wish to consider the workload function for each branch of the conditional construct separately.

In any case, the exact value of the workload function will prove to be of little interest in the remainder of this thesis; overall characteristics of the workload function, like its smoothness or its average value, will be seen to be much more interesting.

2.1.2 Stencils and Interfaces

The second basic element of the distribution problem, apart from the workload function, is the connectivity between the computations. A connection between two computations arises from the fact that one computation needs data that is computed by another computation, i.e. if one computation is dependent on another. Such dependencies lead to communication between processors and hence to a less effective use of the computing resources. This is
why the dependencies between computations are of major interest when considering ways to distribute computations over the available processors.

This section will introduce the notation and give definitions of such dependencies. Note that, where the workload function is defined for computations within a phase, the dependencies that are considered here always lead from one phase to another. Therefore, the complete precedence graph will be considered and not just one phase, as was done in the discussion of the concept of workload functions.

Let $G = (C, E)$ be the precedence graph of a program according to Definition 2 in Section 2.1.1, where $C$ is the set of computations and $E$ is the set of dependencies. Each element in $E$ can be denoted by $(c_1, c_2)$ where $c_1, c_2 \in C$. Computation $c_1$ will be called the source computation and computation $c_2$ the sink computation. When the computations of $G$ have been numbered, as in Section 2.1.1, then each element in $E$ can also be denoted by the number of its source and sink computations. Data dependencies with identical source numbers and also identical sink numbers will be assumed to be collected into a single data dependency.

The set of dependencies that have a particular computation $c_i$ for a sink, will be called the stencil of the computation $c_i$ and be denoted by $Q_i$:

**Definition 5** Let $G = (C, E)$ be the precedence graph of a program and the elements of $C$ be numbered by numbers in $I \subseteq \mathbb{N}^n$ for some $n$. Then the stencil $Q_i$ of the computation at $I = (i_1, \ldots, i_n)$ is defined by:

$$Q_i = \{\vec{q} \in \mathbb{N}^n \times \mathbb{N}^n | (i - \vec{q}, i) \in E\}$$

where each of the vectors $\vec{q}$ is called an offset vector.

This definition is in keeping with that of similar concepts by other authors (e.g. Abraham and Hudak [1], Feautrier [66], Ramanujam and Sadayappan [163] and Reed et al. [164]). When the stencil is independent from $i$, then the index $i$ will be dropped and the stencil will be simply denoted by $Q$.

Given the partitioning $\mathcal{P}(\cdot)$, there will be dependencies in $G$ that cross subset boundaries. These specify the interfaces between the subsets:

**Definition 6** The interface $I_{p_1,p_2}$ from the set of computations $I_{p_1}$ to the set of computations $I_{p_2}$, given the set of dependencies between the computations, $E$, is the subset of computations in $I_{p_1}$ that depend on computations in $I_{p_2}$:

$$I_{p_1,p_2} = \{i \in I | (\mathcal{P}(i) = p_1) \land (\exists j \in I : \mathcal{P}(j) = p_2 \neq p_1 \land (i, j) \in E)\}$$

Usually, a subset does not have interfaces to all other subsets, but only to a limited number of neighbours. Two subsets $p$ and $q$ will be called neighbours if either $I_{p_1,p_2}(E)$ or $I_{p_2,p_1}(E)$ is non-empty.

When solving partial differential equations using a finite difference method, computations are often associated with grid points and then the interface can be visualised as the set of grid points on one processor from which values are needed on another processor. In this case, the definition above coincides with what is commonly meant by an interface. However, the definition is also applicable in cases where the computations are less regular.
2.1. Definitions

On all distributed memory parallel computers and also on shared memory parallel
computers with non-uniform memory access time, there is a time-cost to pay for obtaining
data from another processor. The way in which this cost depends on the number of items to
be transferred and the distance between the communicating processors, is highly dependent
on the specific parallel architecture. Assuming any particular form of cost function is
therefore not desirable at this point. Therefore, the non-specific function $Cost : [1, P] \to \mathbb{R}$
will be used to denote the cost of communication for a given processor $p$. Examples of
possible specifications of the cost function are:

$Cost(p_1) = \max_{p_2 \in \text{Neighbour}(p_1)} \beta_{p_2,p_1} |I_{p_1,p_2}|$
(used by a.o. Roose et al. [174]) where $\beta_{p_2,p_1}$ is the time per data item for transmitting over
the connection between processors $p_2$ and $p_1$, or

$Cost(p_1) = \sum_{p_2 \in \text{Neighbour}(p_1)} \alpha_{p_2,p_1} + \beta_{p_2,p_1} |I_{p_1,p_2}|$
(used by a.o. Thuné [196]) where $\alpha_{p_2,p_1}$ is the latency of the connection between processors
$p_2$ and $p_1$. The first assumes that communication to all neighbours of a given processor
can be performed in parallel and that communication between processors $p_2$ and $p_1$ does
not have any latency. These assumptions are more or less valid in transputer systems
when large messages are transferred between the processors: transputers are capable of
performing multiple communications simultaneously and if the length of the messages is
large enough, the latency can be neglected. The second costfunction above assumes that a
processor communicates to each of its neighbours in turn and that latency is not negligible.

These costfunctions are just a few examples. In fact, finding a suitable costfunction to
describe the communication costs between processors in a particular interconnection net-
work is often not a trivial task. A notable example is the communication on on clusters of
workstations connected by Ethernet. There the communication between one pair of pro-
cessors may hamper communication between another pair (see also Section 4.3 and Roest
et al. [171] for experimental results; see Cap and Strumpen [32] for theoretical details).

2.1.3 The Distribution Problem

With the definitions of the workload function, stencils and interfaces, as given in the pre-
vious sections, the distribution problem for a given program and a given parallel computer
can be stated as follows:

Let $P$ be the number of processors in the parallel computer, let $G = (C, E)$ be the preced-
ence graph of the program, and let the elements of $C$ be indexed by elements of $I \subseteq \mathbb{N}^D$
for some $D$, then determine a partitioning $P : I \to [1, P]$ that meets the following criteria:

I Minimum Workload: Each processor should get a minimum of work to do, i.e. for
each phase $\phi$ of a program

\[
\text{minimise } \max_{p \in [1, P]} L_{p,\phi}
\]
Chapter 2. Partitioning for Parallel Processing: The State of the Technology

If the total workload \( \sum_{p=1}^{P} L_{p,\phi} \) is independent of the partitioning \( \mathcal{P}(.) \), then this is equivalent to:

\[
\text{minimise } \max_{p \in [1,P]} \left( L_{p,\phi} - \frac{\overline{L}_{\phi}}{L_{\phi}} \right)
\]

where \( \overline{L}_{\phi} = \frac{1}{P} \sum_{p=1}^{P} L_{p,\phi} \), i.e. the average amount of work per processor in phase \( \phi \).

II Minimum Communication: The cost that is paid by any processor for fetching its data should be minimised.

\[
\text{minimise } \max_{p} \text{Cost}(p)
\]

III Minimum Partitioning Time: the time spent in determining a suitable partitioning must be kept to a minimum.

It is usually impossible to meet all criteria simultaneously, and therefore, some tradeoff must be made. For example, one may wish to minimise the sum of the workload and communication per processor (see e.g. [43, 152, 174, 177, 196]). Alternatively, one could weigh the first two criteria to put emphasis on one or the other (see e.g. [59, 60]) or use still other ways of combining the criteria (see e.g. [85, 86, 104, 211, 214, 217]). Chapter 4 will illustrate the fact that all the criteria must indeed be considered when determining the effectiveness of distribution methods (see Section 4.3).

Although this formulation of the distribution problem seems obvious enough, there are a number of points to notice about it. First of all, the load balancing problem is not just a matter of minimising the difference between maximum and average workload per processor. Instead, the maximum workload itself must be considered because a partitioning can have effect on the the speed with which computations are performed. An example of this is found in the fact that different partitionings may lead to different cache-use and thus to different computing speeds (see Section 4.3). Requiring the maximum computation time of a processor to be minimal, includes both the desire for an evenly balanced workload distribution (the optimum is always achieved if the maximum equals the average) and the wish for minimum throughput time. In this form, criterion I has been used before by e.g. Farhat and Lesoinne [63] and Zone et al. [217]. Other authors (e.g. [149]) have considered the variance or the root mean square of the processor workloads. Although this usually does give an indication of the load balance, a low variance still allows a high maximum workload and thus a badly balanced computation.

Secondly, the minimum workload requirement is formulated per phase. This assumes that the total throughput time is minimum if the throughput time of each phase is a minimum. This is true for a series-parallel computation, but may not hold for general dependence graphs. Also, it may not hold for programs in which the number of phases depends on the distribution as in some iterative methods (see e.g. Ciarlet et al. [44], Chan et al. [37], Farhat [64], Roose and Van Driessche [176] or Vanderstraeten et al. [199]). In those cases, the minimum workload requirement should be augmented with a requirement on the number of phases.
2.1. Definitions

Furthermore, it will be assumed throughout this thesis that the dependency graph is static and known from an analysis of the program. This is certainly not always true. For example, when applying local mesh-refinement (see e.g. Berger and Oliger [19] or Bell et al. [15]), then the dependency graph may take wildly different shapes during program execution (see e.g. Williams [211] or De Keyser and Roose [114]). In those cases, it may be necessary to resort to dynamic load balancing methods. These methods usually assess the load balance situation after each phase and then decide whether it is useful to adjust the partitioning and/or allocation of processes to processors. This is one more reason for restricting the distribution problem to one phase: it remains valid also in the context of dynamic load balancing. The remainder of this thesis will not address the issue of dynamic load balancing. The interested reader can find more information on this subject in e.g. Bixby et al. [23], Fonlupt et al. [69], Von Hanxleden and Scott [93], Choudhary and Ponnusamy [41], Streng et al. [190] or Tantawi and Towsley [193].

Thirdly, the distribution problem is often formulated for mesh-based algorithms as they occur in e.g. finite difference, finite volume of finite element computations. In those cases, the distribution is usually based on the mesh and it is assumed that each mesh point represents the same amount of work. This is not generally true, however: a balanced partitioning of the mesh points does not necessarily lead to a balanced distribution of workload (in particular, mesh points on the boundary may represent a computational load that is different from that of interior points, as noted by e.g. Thuné [194, 195] and Fox et al. [74], Chapter 11). That is an important reason for formulating the distribution problem in terms of computations rather than mesh points, even though this thesis will primarily deal with mesh-based computations.

Also, a choice is made here to consider the distribution of computations rather than the distribution of data. In most data parallel languages, the computations follow the data on which they act (under the owner-computes rule). Then it would seem more obvious to associate a computational weight with each data item and consider the distribution of data. Here, the possibility is left open that a computation is performed on a processor that holds none of the data on which the computation acts. This formulation is therefore more general. It does, however, require the assumption that any data is always the outcome of a computation (be it a true computation, an input/output operation or a defining statement). In a data parallel language, the location of computations can be steered by the distribution of the data, but in the end, it is still the location of the operations that determines the workload distribution. In this respect, the current work takes the same view as e.g. Feautrier [66].

Furthermore, the communication requirement is formulated in terms of a cost function to account for differences in parallel architectures. So, also for mesh-based computations, the requirement is not simply to minimise the number of mesh-edges across a partition boundary, but rather to minimise the cost associated with those edges.

Finally, the requirement to minimise the time spent in determining a partitioning of the set of computations is usually not explicitly seen as an essential criterion. But in particular for application of partitioning heuristics in dynamic distribution methods, this criterion can be of paramount importance (see e.g. Hanxleden and Scott [93] or Fox et al. [74], Chapter 11).
Note in passing that the distribution problem is actually a partitioning problem on the computations in a precedence graph. The terms ‘distribution’ and ‘partitioning’ will be used interchangeably below. Using any of these words exclusively would lead to very unnatural formulations in some cases and is therefore not desirable.

2.2 Heuristics for Solving the Distribution Problem

The terminology introduced in the previous sections allows for a uniform description of any of the wealth of partitioning heuristics that have been invented and used. That will be the subject of remaining sections of this chapter. In this section, an overview of such methods will be given. The next section will discuss the effectiveness of each of these methods. This chapter will be concluded with a discussion of implementation aspects of distribution methods.

Surveys on the subject have been written before by Chrisochoides et al. [43], Fox [74, 72], Pothen [161] and Roose [174, 175]. As stated in the introduction of this chapter, none of these surveys gives the necessary account of partitioning methods both from an empirical and a theoretical point of view. That is not to say that earlier reviews are not extremely useful: their limited scope is always made up for by the more extensive treatment of their subject.

The classification of partitioning methods that is used here is loosely based on the discussion by Chrisochoides et al. [43]. This classification is a suitable one: the class to which a method belongs reflects the kind of run-time support that is needed by a parallel program that exploits it. Hence, the classification indicates the practical applicability of a method, given the availability of a form of run-time support.

There are three different classes: regular distribution methods, variable block methods and clustering methods. Each of these will be discussed in turn. This section will be concluded with a short review of optimisation methods which are not actually partitioning methods, but are frequently used in combination with partitioning methods and may have a strong effect on the result.

2.2.1 Regular Distribution Methods

The distribution methods that are probably most widely used in parallel processing are the stripwise and blockwise methods and their cyclic variants. These methods are all different forms of what is actually a single method: Cyclic Blockwise Distribution, also referred to as scattered distribution (for example by Morrison [146]). Given a multi-dimensional index set \( I \in \mathbb{N}^{D_I} \) and a set of processors that are numbered by a multi-dimensional set of processor numbers, \( P \in \mathbb{N}^{D_P} \), assuming \( D_I \geq D_P \), and a set of constants \( F_1, \ldots, F_{D_I} \) (the cycle frequencies), then, for cyclic distribution:

\[
\forall i \in I : P(i) = (p_1, \ldots, p_k, \ldots, p_n), \quad p_k = \left[\left(\frac{i_k}{N_k/(P_k F_k)}\right) \mod(P_k)\right] k = 1, \ldots, D_I
\]

where \( i_k \) is the component of \( i \) in the \( k \)-th dimension, \( N_k \) is the size of \( I \) in the \( k \)-th dimension and \( P_k \) is the size of the processor set in the \( k \)-th dimension (\( P_k = 1 \) if \( k > D_P \)).
2.2. Heuristics for Solving the Distribution Problem

Figure 2.4: Cyclic Distribution with $P_1 = P_2 = 2$ and $F_1 = 1$, $F_2 = 2$ (left) and with $P_1 = 1$, $P_2 = 2$ and the $F_1 = 1$, $F_2 = 2$ (right).

$N_k$ is assumed to be a multiple of $P_k F_k$; if it is not then the formulation changes slightly but not essentially (Roose and Van Driessche [175] show that the effect is usually negligible). The cycle frequencies $F_k$, $k = 1, \ldots, n$ are free to choose, subject to $F_k \leq N_k / P_k$.

The effect of cyclic blockwise distribution is most easily explained by considering its effect for a one-dimensional set of computations (i.e. $D_1 = 1$ in the above definition) and a one-dimensional set of processors (i.e. $D_P = 1$). In that case, cyclic blockwise distribution divides the set of computation numbers, $[1, N]$ into $F_P$ equally sized blocks. These blocks are then assigned in a round-robin way to the processors. Processor 1 receives blocks 1, $F + 1$, $2F + 1$, etc. Processor 2 receives blocks 2, $F + 2$, $2F + 2$ etc. and so on. Hence, each processor gets $F$ blocks assigned to it. The reason for calling $F$ a cycle frequency is because it denotes the frequency with which the assignment algorithm returns to a given processor.

Examples of cyclic distribution for two-dimensional sets of computations are shown in Figure 2.4. In the left of the figure, $F_1 = 1$ in the vertical direction and this amounts to a normal blockwise distribution in that direction. Horizontally, $F_2 = 2$ and this produces blocks that are alternately allocated to one of the two processors in that direction. In the right of the figure, the number of processors in the vertical direction is 1 and this produces a stripwise distribution (in this particular case, the value of $F_1$ has no effect).

In many practical cases, non-cyclic (or normal) blockwise distribution (i.e. $F_k = 1$ for any $k$) has the attractive property of producing rather small interfaces (see Reed et al. [164]). On the other hand, when not all data elements represent the same amount of computational work, then a blockwise distribution will not be balanced. This imbalance can be reduced by increasing $F_k$ (see e.g. Cannataro et al. [31], Feautrier [66], Fox et al. [73], Gupta and Banerjee [89, 90], Von Hanxleden and Scott [93], Mierendorff [142] or Migliardi et al. [143]). But increasing $F_k$ usually also increases the total interface size and should therefore be done with care. A further discussion of these aspects will be given in Section 2.3.1 and Chapter 4.

A different form of the above method is the hexagonal partitioning, discussed by Abraham and Hudak [1] and by Reed et al. [164], where the partitions are not multi-dimensional intervals but (in two dimensions) hexagons. The hyperplane partitioning method, described
Chapter 2. Partitioning for Parallel Processing; The State of the Technology

Figure 2.5: Variable blockwise partitioning into $P_1 \times P_2 = 4 \times 2$ parts.

by Ramanujam and Sadayappan [163] and later by Feautrier [66] and Huang and Sadayappan [103], is also a form of blockwise distribution. It rotates a given index set so that all (or most of the) edges in the stencils are orthogonal to the second dimension and then performs a one-dimensional cyclic block partitioning on the rotated index set with the partitioning planes orthogonal to the second dimension. In this way, a minimum number of edges will cross the partitioning planes and consequently, the communication volume is small. See also the discussion in Section 2.4.

The cyclic blockwise distribution method and hexagonal partitioning have in common that the part to which any index belongs can be computed explicitly from just a few parameters ($F$ and $P$ for ordinary cyclic blockwise distribution). This property is due to the fact that the width of each part is the same, which makes the partitioning regular. Therefore, these methods will be referred to as regular partitioning methods. The popularity of these methods is mainly due to the fact that all the necessary communication operations in a program can be inserted automatically by a compiler, so that the programmer needs only specify the partitioning itself.

2.2.2 Variable Block Methods

For all other distribution methods that are currently known, the partitioning can not be expressed in closed form. In many cases, the actual distribution depends on quantities that are not known until runtime. Therefore, a compiler can not insert the necessary communication operations and hence a far more intricate run-time support is required, like the inspector-executor mechanism (see Section 2.4). When such support is available, there is no real restriction to the partitioning, other than that it must be a good (approximate) solution to the distribution problem.

The most obvious extension of cyclic distribution to more general partitionings is the variable block method where each partition is still a multi-dimensional interval, but the intervals are no longer equally sized (see Figure 2.5 for examples). When the size of partitions is no longer fixed, it is possible to accommodate variations of workload on the index space by choosing the size of each interval so that it contains a fixed amount of workload. In one dimension, the selection of interval sizes is trivial (see e.g. Choudhary and Ponnusamy [41] and Von Hanxleden and Scott [93]). In higher dimensions, the situation is more complicated. Choosing the size of the intervals in each direction independently
2.2. Heuristics for Solving the Distribution Problem

does usually not produce a balanced partitioning. Consequently, the partitioning in each
direction must be done in turn. The most direct implementation of this is the *jagged*
method, described by Saltz et al. [182] (for an example, see Figure 2.5). This method
starts by performing a partitioning into \( P_1 \) parts in the first dimension. Then each of these
parts is partitioned into \( P_2 \) parts in the second dimension, etc.

A different way of constructing a variable block partitioning is the *Orthogonal Recursive
Bisection (ORB)* or *Recursive Coordinate Bisection (RCB)* method, first described by Fox
[72] and later analysed thoroughly by Berger and Bokhari [18]. It has been widely studied
and used; theoretical and empirical results are reported by a.o. Baden [7], Boris [27],
Cybenko and Allen [46], Simon [186], De Keyser and Roose [115], Williams [211] and Zone
et al. [217]. The RCB method starts with a bisection (i.e. a partition into two parts) in
the first dimension, after which both parts are bisected in the second dimension, followed
by a bisection of each of the resulting four parts in the third dimension, etc. (see Figure
2.5 for an example). After bisection in the highest dimension, the next splitting is done
in the first dimension again. In this form, the method can only be used if the number of
processors in each dimension is a power of two, but variants exist that can accommodate
any number of processors (see Hu and Blake [102] for details). A parallel implementation
of this partitioning method has been described by Heath and Raghavan [95]; such an
implementation can be useful when handling extremely large partitioning problems.

Instead of bisecting each time, it is also possible to perform a partitioning into more
parts, like quadrisection or octasection (as implemented in the Chaco partitioning package
[97]), which is something of a compromise between jagged partitioning and pure ORB.
Chan et al. [36] and Gilbert et al. [79] have developed a variant of this method to make it
applicable to graphs in which vertices are originally not associated with coordinates.

In the previous section, hexagonal partitioning was introduced in which the index set
is rotated prior to partitioning so that a minimum of edges will be cut by the hyperplane.
The basic idea behind this approach is that a proper embedding of the dependence graph
in the index set can make it easier to find a good partitioning of the graph. Many variable
block methods employ the same approach; they use a relatively simple partitioning scheme
in combination with some more or less advanced scheme for numbering the vertices in a
dependence graph.

Several examples of such methods are presented by Ou et al. [153, 152], Kaddoura et
al. [109] and Oden et al. [150]; they use a bit interleaving operation to combine the two
indices of a two-dimensional index set into a single sequence number (see Figure 2.6, left)
or an ordering along a Hilbert curve (see Figure 2.6, middle). These orderings have the
advantage that points that are spatially close together are usually also close in the ordering.
They are particularly attractive for use in dynamic load balancing: the interval sizes in
the variable block distribution can be easily modified while preserving the proximity in the
distribution.

Other useful orderings are the bandwidth-reducing orderings that have been developed
in the field of finite element analysis. Malone [135] shows how these can be used to limit
communication in a parallel processing environment. Examples of such ordering methods
are the *(Reverse)* Cuthill-McKee algorithm (see e.g. Liu and Sherman [132] for details
about the algorithm and Farhat [65] or Venkatakrishnan et al. [202] for applications in
partitioning) and the one-way dissection method introduced by George [77]. The one-way dissection method orders the vertices in a graph (e.g. a dependence graph) according to their distance from one particular vertex, called the pseudoperipheral vertex (see Grimes et al. [87] for a way of selecting such a vertex). The distance is measured for example by the minimum number of edges between a given vertex and the pseudoperipheral vertex. After all vertices have been embedded in an interval in N in this way, cyclic distribution can be applied to this interval. The nested dissection method, introduced by Lipton et al. [128] and also described by Sadayappan et al. [177] and Simon [186], can be viewed as a one-way dissection method combined with RGB: in each RCB recursion, the vertices are embedded in an interval of N using one-way dissection and then the resulting interval is bisected.

Inertial bisection, developed by Farhat [63, 65], is also a form of RCB but in each recursion, it rotates a given index set such that it has its highest moment of inertia in the direction orthogonal to the next partitioning plane. The rationale behind this method is that the inertial axes give an indication of the main elongation of the index set, and a plane that is orthogonal to the principal inertial axis will hopefully cut the index set in a direction in which the index set is small. A parallel version of this algorithm has been presented by Diniz et al. [52]. Chрисoхoids et al. [43] present yet another numbering method, where the vertices are ordered according to their distance from the one of the boundaries of the mesh.

The spectral bisection method, which is widely recognised as one of the most reliable partitioning methods, is an RCB-based method in which the vertices in a graph are ordered in each recursion according to their value in the eigenvector belonging to the second-smallest eigenvalue (called the Fiedler vector) of a particular form of adjacency matrix (the so called Laplacian matrix; details are given in Section 2.3.2). The ordering that is determined by the Fiedler vector is usually very good at keeping the number of edges that cross the bisection plane low. The method is based on the work of Pothen et al. [162] and has aroused a tremendous interest (see e.g. [11, 56, 57, 58, 88, 99, 102, 106, 186, 211] to mention just a few). Initially, a major drawback of the method was the fact that computation of the Fiedler vector has a high time-complexity. The introduction of multilevel versions of the algorithm (see e.g. Barnard and Simon, [11], or Van Driessche and Roose, [57]), where
2.2. Heuristics for Solving the Distribution Problem

the graph is first coarsened into a smaller graph, then bisected, and then expanded again, has largely reduced this problem. Another method to reduce the time complexity of RSB, introduced by Walshaw [208], is to start from a given partitioning and applying RSB only to vertices near the boundaries of the parts. Finally, parallel processing can be applied to reduce the time taken by the partitioning, as discussed by e.g. Barnard and Simon [12], Johan et al. [106], Williams [211] and by Allen [4].

2.2.3 Clustering Methods

The last class of partitioning methods is usually referred to as clustering methods. These methods do not produce equidistant separators and do not even assume an ordering of the vertices. Instead, these algorithms search clusters of vertices in the graph that are compact in some sense. A well-known representative of this class is the greedy algorithm that was first used by Farhat [62, 63, 64, 65] and later improved by Al-Nasra and Nguyen [3] and in a different way by Lin [126]. Several other clustering algorithms are discussed by Ercal et al. [60], Liu [130], De Keyser and Roose [115], Roose et al. [174], Sadayappan et al. [177], Vanderstraeten et al. [200] and by Zhou [214]. Some of these clustering algorithms are also in use to coarsen meshes prior to partitioning to reduce the costs of partitioning as is common practice for Recursive Spectral Bisction. Partitioning packages like Chaco [97] and Metis [110, 111] offer such a coarsening facility.

2.2.4 Optimisation Methods

In many cases, the partitioning that is determined by one of the algorithms that has been mentioned so far is not optimal. Since each of these methods is heuristic in nature, it can be expected to give at best an approximately good partitioning. Therefore, an optimisation method is often applied to the result to improve the quality of the partitioning. In virtually every case, the optimisation results in a cluster partitioning: no deterministic, one-shot method can be devised that will produce the same final partitioning. Sometimes, the original partitioning that is to be optimised is just a random assignment of vertices to clusters and the optimisation method is then used to find a more suitable assignment. In those cases, the optimisation method effectively takes over the role of the partitioning method. Formulating the partitioning problem as an optimisation problem makes it amenable for application by optimisation methods like linear programming. Early work in this direction was reported by Gyllys and Edwards [91], but their results readily showed that the usual optimisation methods fail for problems of sizes as they occur in parallel processing.

By far the oldest optimisation method that is still in use in the field of partitioning is the Kernighan-Lin method, which was proposed as early as 1969 by Kernighan and Lin [113]. In worst-case, the method has an $O(N^3)$ time complexity, where $N$ is the number of vertices in the graph, and is therefore extremely time-consuming. In practice however, the method needs far less time. Van Driessche and Piesens [55] report a complexity of $O(N^{2.45})$ and Johnson et al. [107] even state that the complexity is not much worse than $O(N)$ if the method is implemented in a particular way. Variants of the method exist that have a worst-case complexity much lower than $O(N^3)$, like the one that was developed by Fiduccia
Chapter 2. Partitioning for Parallel Processing; The State of the Technology

and Mattheyes [67] which is claimed to be a linear-time procedure, but also gives poorer results according to Johnson et al. [107]. Another way of reducing the time that is spent by the method is through parallelisation; parallel variants have been reported by e.g. Van Driessche and Roose [55], Gilbert and Zmijewski [81] and by Diniz et al. [52]. An important property of the Kernighan-Lin method and its variants is that it possesses a hill-climbing capability, that is: it does not necessarily search for the closest, local, optimum solution, but is capable of “climbing out” of a local optimum towards a better one. Generalisations of the Kernighan-Lin method have been reported by Goto [84] and Lee et al. [122]. Extensions that were independently developed by Chrisochoides et al. [43, 42] and by Walshaw et al. [209] aim not only at minimising the size of the interface between parts but also at making the parts as compact as possible. The Kernighan-Lin method has also been extended to optimise a quadrisection rather than a bisection as in the original variant (see Suaris and Kedem [191]).

Some other methods that have been reported to optimise a given partitioning do not have this hill-climbing property, but can still be very worthwhile. One such method has been introduced by Liu [131] and has later been modified by Lin [126]. This method starts from a given partitioning and tries to reduce the size of the interface by a method that is based on bipartite graph matching.

Two methods that have received quite a lot of attention in recent times are simulated annealing and genetic algorithms. Both are stochastic in nature and theoretically very robust, but in practice very time-consuming.

Simulated annealing [116] performs a sequence of random moves, accepting those moves that are beneficial and some of the moves that are not. The probability of accepting a bad move is lowered as the process continues and if this lowering is done slowly enough, the final solution will be optimal (see van Laarhoven and Aarts [118] or Ingber [105] for good introductions to the method). Johnson et al. [107] discuss the application of simulated annealing to graph partitioning; they find that the time complexity of the method is somewhat balanced by the fact that the results are often better that those of Kernighan-Lin optimisation. However, when there are constraints on the time that can be spent on partitioning, simulated annealing becomes infeasible (see e.g. Williams [211]). Efforts have been made to reduce the time complexity of the method through parallelisation (see Darema et al. [47], Natarajan [148], Williams [211] and Xue [213]), but these variants do not seem to be widely used.

A genetic partitioning algorithm starts from a set of arbitrary initial partitionings, which are probably not optimal. The best of these initial partitionings are taken and combined to produce new solutions, from which again a number of good candidates is selected etc. (see Whitley [210] for a good introduction). Many experiments have been done with genetic algorithms in the field of partitioning (see e.g. [54, 94, 104, 120, 174, 192] to name just a few of the most interesting) but, like the original Kernighan-Lin method and the simulated annealing method, genetic algorithms are usually prohibitively time-consuming. They can be successfully parallelised (see e.g. [68, 83, 184, 189, 192]) but even then their results seldom justify the costs of the algorithm. An extensive comparative study of Kernighan-Lin, simulated annealing and genetic algorithms has been performed by Laszewski [121].
2.2. Heuristics for Solving the Distribution Problem

Table 2.1: Order of practical time-complexity of partitioning methods, where $N$ is the number of vertices in the graph, $D$ is the dimension of vertex-number space, $P$ is the number of parts. The orders apply to practical time complexity of optimal implementations of the original algorithms. In cases where more than one complexity is given, the lower complexity applies in cases where the vertices are already properly sorted.

<table>
<thead>
<tr>
<th>method</th>
<th>order of time complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Regular methods</strong></td>
<td></td>
</tr>
<tr>
<td>Cyclic blockwise distribution</td>
<td>$DN$</td>
</tr>
<tr>
<td><strong>Variable block methods</strong></td>
<td></td>
</tr>
<tr>
<td>Jagged</td>
<td>$DN, DN \log(N)$</td>
</tr>
<tr>
<td>RCB, RIB</td>
<td>$N \log(P), N \log(N) \log(P) [153]$</td>
</tr>
<tr>
<td>Nested Dissection</td>
<td>$N \log(P), N \log(N) \log(P)$</td>
</tr>
<tr>
<td>HI, BI</td>
<td>$N \log(N)$</td>
</tr>
<tr>
<td>RSB</td>
<td>$N \sqrt{N} [153, 208]$</td>
</tr>
<tr>
<td><strong>Clustering methods</strong></td>
<td></td>
</tr>
<tr>
<td>Greedy, Cuthill-McKee</td>
<td>$N$</td>
</tr>
<tr>
<td><strong>Optimisation methods</strong></td>
<td></td>
</tr>
<tr>
<td>Kernighan-Lin</td>
<td>$N^{1.25}$</td>
</tr>
<tr>
<td>Simulated Annealing</td>
<td>$&gt; N^3$</td>
</tr>
<tr>
<td>Genetic Algorithms</td>
<td>$&gt; N^3$</td>
</tr>
</tbody>
</table>

2.2.5 Comparing distribution methods

In Section 2.1.3, the distribution problem has been formulated in terms of three criteria. A distribution method can be said to be effective in solving the distribution problem if it performs well on all three criteria: it must properly distribute the workload, it must lead to a minimum of communication and it must determine the distribution quickly. With respect to the first two criteria, a more extensive discussion of each method will be given below. With respect to the time needed to determine the distribution, a few remarks will be made here.

In general, the regular methods have the smallest time-complexity, i.e. they require the least time to determine a partitioning. On the other end of the complexity-spectrum are the optimisation methods, which usually require much computational work. A comparison of the order of time-complexity of distribution methods is given in Table 2.1

Note that the order of time complexity is listed, rather than the time-complexity itself. This is because the time-complexity of methods such as IRB, RSB and in particular the optimisation methods depends on the problem that needs to be solved and can not be as easily listed.
Chapter 2. Partitioning for Parallel Processing: The State of the Technology

2.3 The Effectiveness of Distribution Methods

In this section, a survey will be given of the reports in literature with regard to the effectiveness of partitioning methods. The effectiveness will be studied with respect to the interface size (see Definition 6) and the degree of balance achieved (see Section 2.1.3). Both theoretical and empirical studies will be used to evaluate partitioning methods. A major problem of this kind of study is that almost every author in the field of partitioning for parallel processing uses his own set of assumptions, his own set of test problems and his own quality indicators. Therefore, it is difficult to compare results from different authors. This should be kept in mind throughout the discussion below.

It is the aim of this section to compile an overview of existing results, rather than to fill in the gaps and therefore, no attempt will be made to do so even if this will leave the results sometimes rather scarce. With respect to theoretical results, there will occasionally be a need for corrections or for a proof of theorems for which no proof has been given. This will be the only exception to the general rule that has been followed in compiling this section: to restrict the discussion to results from literature only.

The line of discussion will follow the classification that was introduced above. First, the methods will be discussed that work on a fully regular index set and give equidistant splitting lines. In the next two sections, results for the variable block methods and clustering methods will be discussed.

2.3.1 Regular Methods

The discussion here will start with results for blockwise partitionings with cycle frequency equal to one in all dimensions. Then, the effect of raising the cycle frequency will be addressed.

I) Cycle frequency equal to one

Distribution of Computational Work

In a series of articles Thuné [194, 195, 196] has studied the effect of the stripwise and blockwise partitioning methods for a particular workload function. The workload function is the following. Let \( I = [1, N] \times [1, N] \) be a two-dimensional index set. Let \( \delta I \) be the border of the index set, i.e. the collection of points \((i_1, i_2)\) in \( I \) for which at least one of the four points \((i_1 \pm \tilde{q}, i_2 \pm \tilde{q})\) is outside \( I \), where \( \tilde{q} \in \mathbb{N} \) is the width of the boundary. Now the workload function \( W(.) \) that is considered by Thuné is given by:

\[
W(i_1, i_2) = \begin{cases} 
  w_B & (i_1, i_2) \in \delta I \\
  w_I & (i_1, i_2) \in I - \delta I 
\end{cases}
\]

This workload function models explicit finite difference computations on a rectangular grid: the number of operations is constant in all gridpoints except those on the boundary, where the forcing of boundary values may require a different number of operations.
The load imbalance for one-dimensional blockwise distribution (i.e. \( D_p = 1 \) in the expression in Section 2.2.1) of workload function (2.1) over \( P \) processors, is, according to Thuné [194]:

\[
\frac{\max(L_p) - \overline{L}}{L} \leq \begin{cases} 
2 \frac{w_l - w_B}{w_l} \frac{\tilde{N}}{N} + \mathcal{O}\left(\frac{1}{N^T}\right) & w_B \leq w_l \\
(P - 2) \frac{w_l - w_B}{w_l} \frac{\tilde{N}}{N} + \mathcal{O}\left(\frac{1}{N^T}\right) & w_B > w_l
\end{cases}
\] (2.2)

For a two-dimensional blockwise distribution, Thuné [194] finds:

\[
\frac{\max(L_p) - \overline{L}}{L} < \begin{cases} 
4 \frac{w_l - w_B \tilde{N}}{w_l} + \mathcal{O}\left(\frac{1}{N^T}\right) & w_B \leq w_l \\
2(\sqrt{P} - 2) \frac{w_l - w_B}{w_l} \frac{\tilde{N}}{N} + \mathcal{O}\left(\frac{1}{N^T}\right) & w_B > w_l
\end{cases}
\] (2.3)

These expressions confirm the intuitive expectation that, if the size of the grid grows with respect to the size of the boundary (i.e. \( N \) grows with respect to \( \tilde{N} \)), then the variations in load per block decrease with respect to the load itself and hence the load imbalance for stripwise or blockwise distribution decreases. Also, if the difference between the boundary workload and the internal workload becomes larger, then so does the load imbalance. Finally, when the workload on the boundary is larger than the internal workload, then an increase in \( P \) will increase the load imbalance, because the average size of each part will decrease and therefore the effect of the load differences will become more pronounced.

Some interesting experimental results regarding the inability of regular partitionings to evenly distribute computational work are given by Choudhary and Ponnusamy [41]. They compare several methods for distributing a digital image to be analysed by a parallel computer vision system. For one part of the analysis (the time-match algorithm), the regular distribution produces a speedup of only four on sixteen processors, whereas variable block methods come close to a speedup of 14. The difference in speedup is entirely due to the imbalanced computation. Less extreme but similar results are presented by Hanxleden and Scott [93]: for their Monte Carlo algorithm the differences in speedup between a regular non-cyclic distribution and other distribution methods amounts to a factor of 1.5.

The Size of the Interface

In this and following sections, the interface that will be considered is the interface around the computations in a single phase of a program, i.e. \( I_{p_1} \) in Definition 6 is the set of computations on a given processor \( p_1 \) in a single phase.

With respect to the interface size under regular partitionings, a very general result is due to Ramanujam and Sadayappan. In its original form, their result contains a few flaws; here it will be discussed in its correct form. Abraham and Hudak [1] came to similar results before Ramanujam and Sadayappan, but the expression given by the latter authors is more compact. Therefore, the result of Ramanujam and Sadayappan will be discussed here in stead of that of Abraham and Hudak.

Consider a two-dimensional index set (e.g. \( I \) above) with stencil \( Q \) independent from the index to which the stencil is applied (see Definition 5). Assume that the stencil consists of \( m \) offset vectors, denoted by \( \tilde{q}_j \in \mathbb{R}^2, j = 1, \ldots, m \). Then, define the matrix \( Q \) as the \( 2 \times m \) matrix that has \( \tilde{q}_j \) for its \( j \)-th column. Let the index set be partitioned by two sets...
of equidistant parallel splitting lines, normal to $s_1$ and $s_2$ respectively, where $s_1$ and $s_2$ are independent and have unit length (see Figure 2.7 for an illustration). The parts that are defined in this way are parallelograms. Then, let $S$ be a $2 \times 2$ matrix that has $s_1$ and $s_2$ for its rows. Let the ratio between the lengths of the sides of the parallelogram be $r$. Then the size of the total interface around any part $p$ is (based on the result of Ramanujam and Sadayappan [163])

$$
\sum_{p \in [0,P]} |I_{p_1,p_2}| \leq \frac{N}{\sqrt{P} r \sqrt{|\det(S)|}} (Q'(1)r + Q'(2))
$$

(2.4)

where $|\det(S)|$ denotes the absolute value of the determinant of $S$ and where

$$
Q'(k) = \sum_{j=1}^{m} |(SQ)_{kj}| \quad k = 1, 2.
$$

(2.5)

In Expression (2.4) as originally given by Ramanujam and Sadayappan, an equality symbol is used instead of the '$\leq$' symbol. This is not correct though, because the contributions to the interface from each of the offset vectors can overlap. This was already noted by Abraham and Hudak who use a different expression for $Q'(k)$ to compensate for this overlapping:

$$
Q'(k) = \max_{j \neq (SQ)_{kj} < 0} |(SQ)_{kj}| + \max_{j \neq (SQ)_{kj} > 0} (SQ)_{kj}.
$$

With $Q'(k)$ given by this expression, the '$\leq$' symbol in Expression (2.4) can be replaced by a '$\approx$' symbol.

Furthermore, Ramanujam and Sadayappan place a factor 2 in front of the expression for the total interface size but this is again not right. To see this, consider the special case where $Q$ is a five-point star stencil and $s_1$ and $s_2$ point along the $x-$ and $y-$axis respectively. Then $Q$ is the matrix

$$
\begin{pmatrix}
1 & -1 & 0 & 0 \\
0 & 0 & 1 & -1
\end{pmatrix}
$$
2.3. The Effectiveness of Distribution Methods

Let $S$ be the identity matrix (i.e. the splitting lines are along the first and second dimension), so that $Q'(1) = Q'(2) = 2$ and Expression (2.4) reduces to:

$$\sum_{p_2 \in [1,P]} |I_{p_1,p_2}| = \frac{2N}{\sqrt{P}} (r + 1).$$

where an equality sign can be used in this particular case because the contributions to the interface due to the different offset vectors do not overlap. The right-hand side of this expression attains its minimum for $r = 1$, i.e. when the parts are square. The size of the interface is then $4N / \sqrt{P}$ which is easily seen to be correct. Thus, there should not be a factor 2 in front of the right-hand side of Expression 2.4.

Reed et al. [164] have compared several special cases of combinations of stencils and partitionings. For a five-point star stencil and a partitioning into triangles they find a total interface size of $3\sqrt{2N} / \sqrt{P} \approx 4.2N / \sqrt{P}$ and for a partitioning into hexagons they find a total interface size of $3N / \sqrt{P} - 2$. Thus, for a five-point star stencil, the hexagonal partitioning has a smaller interface than a partitioning into triangles or squares. For several other common stencils, the hexagonal partitioning is also best. But when taking the influence of the parallel architecture into account, the balance shifts in favour of the partitioning into squares, because the hexagonal partitioning has a higher number of neighbours which can not be accommodated in many interconnection topologies.

Noting that the size of $J_0$ is proportional to $1/P$ and the size of the total interface, $\sum_{p_2 \in [1,P]} |I_{p_1,p_2}|$, is proportional to $1/\sqrt{P}$ in case of rectangular parts (see above), it can be seen that the ratio between the number of points in a part and the size of the interface is:

$$\frac{\sum_{p_2 \in [1,P]} |I_{p_1,p_2}|}{|I_p|} \propto \sqrt{P},$$

thus: when the number of processors grows then the size of the interface will grow with respect to the size of the parts themselves. Therefore, a high number of parts will usually lead to a less efficient parallel computation. As noted by Roose and Van Den Driessche [175], this effect implies an upper bound on the number of processors that can be usefully applied for a given problem size $N$. Solchenbach [188] gives a more extensive account of this effect, including the effect from machine-parameters and also considering the case of a three-dimensional index set, coming to essentially the same conclusion.

In some cases, it is possible to find a partitioning such that the size of the interface around any partition is zero (see e.g. Ramanujam and Sadayappan [163] and Huang and Sadayappan [103]). For stripwise partitionings of index sets that are intervals in $\mathbb{Z} \times \mathbb{Z}$, a sufficient condition for the existence of such a communication-free partitioning has been obtained by Ramanujam and Sadayappan [163]. In cases where the existence of a communication-free partitioning is not immediately obvious, this condition can be used to determine whether it exists and, if so, how it can be constructed. Therefore, it can be a powerful tool in searching for an optimal partitioning. This warrants a short discussion here, even though a communication-free partitioning will not often be possible in practical situations.

Consider a program consisting of two phases, $\phi_1$ and $\phi_2$ (see also the definition of phases in Definition 3). Let the computations in $\phi_1$ be numbered as $(\phi_1, i_{11}, i_{12})$ and those in $\phi_2$
Chapter 2. Partitioning for Parallel Processing; The State of the Technology

by \( (\phi_2, i_{11}, i_{22}) \). Let the stencil \( Q_r \) consist of a single offset vector of the form:

\[
\tilde{g}_r = (\phi_2 - \phi_1, (1 - a_{11})i_{11} - a_{12}i_{12} - a_{13}, -a_{21}i_{11} + (1 - a_{22})i_{12} - a_{23})
\]

for integer values of \( a_{11}, \ldots, a_{23} \). Then according to Ramanujam and Sadayappan [163], it is possible to partition the computations in phase \( \phi_1 \) and those in phase \( \phi_2 \) such that the interface around any part in the partitioning is empty, if the system

\[
\begin{pmatrix}
  a_{11} & a_{12} & 0 \\
  a_{21} & a_{22} & 0 \\
  -a_{13} & -a_{23} & 1
\end{pmatrix}
\begin{pmatrix}
  \alpha' \\
  \beta' \\
  \gamma'
\end{pmatrix}
= \begin{pmatrix}
  \alpha \\
  \beta \\
  \gamma
\end{pmatrix} \tag{2.7}
\]

has a solution for which \( \alpha', \beta', \gamma', \alpha, \beta, \gamma \in \mathbb{N} \) and at most one of \( \alpha', \beta' \) is zero and at most one of \( \alpha, \beta \) is zero.

If the number of offset vectors in the stencil is larger than one, a condition of the form of Equation (2.7) must be fulfilled for all offset vectors simultaneously for the same vectors \((\alpha', \beta', \gamma')\) and \((\alpha, \beta, \gamma)\).

II) The effect of raising the cycle frequency

Distribution of Computational Work

Results regarding the balancing effect of making cycle frequencies larger than one all consider cyclic blockwise distribution applied to a one-dimensional index set for a one-dimensional set of processors. The earliest analysis of the effect has been given by Salmon [178]. He assumes that, under cyclic distribution, the workload per block of all the blocks on a processor are independent. With this assumption, he derives an expression for the load imbalance that, with 99% certainty, will not be exceeded:

\[
\left( \frac{\max(L - \bar{L})}{\bar{L}} \right)_{99\%} = \frac{\sigma_{\text{block}}}{\sqrt{PF \cdot W_{\text{block}}}} \sqrt{\ln \left( \frac{10^4P}{2\pi} \right)} \tag{2.8}
\]

where \( W_{\text{block}} \) and \( \sigma_{\text{block}} \) are the mean and standard deviation of the workload per block respectively. Usually, \( \sigma_{\text{block}} / W_{\text{block}} \) will be approximately independent from \( F \), so that the load imbalance will be proportional to \( 1/\sqrt{F} \). Exactly the same result was obtained before by Kruskal and Weiss [117], who performed a much more extensive study of the effect of placing several blocks with independent workloads on one processor.

But Salmon’s assumption that workloads of the blocks on each processor are independent when using cyclic blockwise distribution, is debatable. If workload is correlated over a major part of the domain, then the execution time of the blocks on a given processor will not be completely independent.

In contrast to Salmon, Nicol and Saltz [149] argue that the process behind cyclic distribution is to make the total workloads on all processors correlated: because blocks from a certain region of the domain will be scattered over all processors, each processor will get an equal share of the compute intensive blocks. This makes the variation of workloads among the processors small and consequently improves the load balance.

Nicol and Saltz study the effect of cyclic blockwise distribution for several increasingly strict assumptions on the covariance function of one-dimensional workload functions. From
2.3. The Effectiveness of Distribution Methods

these studies, they derive three theorems, that will be discussed here, because they provide quite some insight in the effect of raising the cycle frequency.

The first theorem is the most general:

**Theorem 1 (Nicol and Saltz [149])** Suppose the workload function \( W(i) \), \( i = 1, \ldots, N \) is a second-order stationary process with a convex covariance function. Then increasing the cycle frequency of a cyclic distribution does not increase the processor workload variance.

which shows that, if reduction of the variance of processor load is the only issue, increasing the degree of a cyclic distribution is never wrong provided the workload function has a convex covariance function. However, the variance is not the most suitable quantity to consider, as the execution time for a parallel program is dominated by the maximum processor workload, rather than by the variance of the processor workload. This maximum is the subject of the following theorem:

**Theorem 2 (Nicol and Saltz [149])** Let \( W(i), i = 1, \ldots, N \) be a stationary Gaussian process with a covariance function \( \text{Cov}(i) = \sigma^2 \max(0, 1 - \frac{i}{N}) \), where \( \alpha \geq 1 \) is such that \( N/\alpha \) is a multiple of the blocksize \( N/PF \). Let there be \( P = 2^a \) processors and let the cycle frequency \( F = 2^b \) where \( a \) and \( b \) are integers. Let cycle frequency \( F_0 \) be the smallest power of two such that \( F_0/\alpha \) is integer. If \( F_2 > F_1 > F_0 \), then the expected maximum processor workload of a cyclic distribution with cycle frequency \( F_2 \) is no greater than that of a cyclic distribution with cycle frequency \( F_1 \).

The requirement that \( N/\alpha \) is a multiple of the blocksize guarantees that the point where the covariance becomes zero will coincide with the boundary of a block, which eases the proof of the theorem. Similarly, the requirement that \( F_0/\alpha \) must be integer states that there will be an integer number of cycles on the range in which the covariance is non-zero, again to simplify the proof of the theorem. Both assumptions are not essential for the basic result of the theorem to hold.

Like Theorem 1, this second theorem does not guarantee that there will be any substantial decrease of the maximum processor workload; it merely states that there will be no increase. However, in proving the theorem, Nicol and Saltz derive an explicit expression for the processor workload variance, which is:

\[
\text{Var}[L(P, F)] = \frac{N^2 \sigma^2}{\alpha P^2} \left( 1 - \frac{1}{3\alpha} + \frac{\alpha(1 - 1/P)}{3F^2} \right)
\]  

(2.9)

This equation gives an impression of how fast the variance decreases with \( F \). Now assume that \( N \gg \alpha \), so that a wide range of values for \( P \) and \( F \) can be found that meet the criteria of Theorem 2, and that furthermore \( \alpha/F^2 \gg 1 \), then Equation (2.9) can be approximated by:

\[
\text{Var}[L(P, F)] \approx \frac{N^2 \sigma^2}{3(PF)^2} \left( 1 - \frac{1}{P} \right) = O\left( \frac{N^2}{(PF)^2} \right)
\]  

(2.10)

This shows that the decrease in variance that is obtained by cyclic distribution can be quite significant. Assuming that a small variance of the processor workloads \( L \) implies a small load imbalance, Equation (2.10) indicates that increasing the cycle frequency may strongly
improve the load balance. The reduction in processor workload variance according to $1/P^2$ is in agreement with results from this thesis (see Chapter 3).

Note in passing that the variance according to Equation (2.9) decreases with $1/P^2$, which implies that the standard deviation (the square root of the variance) decreases equally fast with $P$ as the mean workload per processor (which obviously decreases proportional to $1/P$). So, the ratio between the standard deviation of the workload and the mean workload is insensitive to the number of processors: adding more processors to the system does not seem to improve or worsen the load imbalance. On the other hand, the result of Thuné, as given in Equation 2.2, does show an effect from the value of $P$ when $w_B > w_I$. The results in Chapters 3 and 4 of this thesis will also show a limited influence of $P$ on the load imbalance. However, it is difficult to compare the results of Thuné with those of Nicol and Saltz in this thesis, because they apply to different kinds of workload functions (deterministic vs. stochastic) and to different quantities (the maximum workload vs. the standard deviation of the workload).

The last theorem from the work of Nicol and Saltz [149] states that the cyclic distribution is the best of any distribution that assigns $F$ blocks to each of $P$ processors. It uses the so-called balanced assignment matrix. An assignment matrix is defined by the authors as a $P \times PF$ matrix, in which entry $i, j$ equals one if block $j \in [1, PF]$ is assigned to processor $i \in [1, P]$ and zero otherwise. An assignment matrix is balanced if it assigns the same number of blocks to each processor, i.e. if the sum over the elements in each row of the matrix is equal. The theorem states that the balanced assignment matrix of a cyclic distribution is the best conceivable one for a workload function with linearly decreasing covariance:

**Theorem 3 (Nicol and Saltz [149])** Let $W(i), i = 1, \ldots, N$ be a second-order stationary process, with a covariance function $\text{Cov}(i) = \sigma^2(i, t; i)$, where $0 \leq \alpha \leq 2$. Let $A_M$ be the balanced assignment matrix describing the cyclic distribution. Then for any $P \times PF$ balanced assignment matrix $A$, the average variance of the workload on a processor is at least equal to that for assignment matrix $A_M$.

Although these results make a fairly general impression, their proofs rely completely on the assumed form of the covariance function, and consequently, it is not obvious how the results carry over to other workload functions. There is some reason to assume that the covariance of workload functions that occur in practice are often similar to those used in the theorems, but still there is no indication of what deviations from the covariance assumptions of the theorems can be accepted for the results to remain valid.

The main contribution of Nicol and Saltz's paper is their method of proof. It can be extended to other workload functions, although the algebra will probably become much more complicated and perhaps in some cases it will even be impossible to get closed-form expressions for the workload variance.

One of the few explicit results on the effect of raising the cycle frequency for a particular workload function is given by Gupta and Banerjee [90]. They state without proof a formula for the workload of the most heavily loaded processor for a workload function of the form:

$$W(i_1, i_2) = \begin{cases} 1 & i_2 \leq i_1 \\ 0 & i_2 > i_1 \end{cases} i_1, i_2 \in [1, N].$$
2.3. The Effectiveness of Distribution Methods

The formula they give is:

$$\max(L_p) = N \left( \frac{N}{PF} \left[ \frac{N}{2} \right] / \left( \frac{N}{F} \right) + \min\left( \frac{N}{PF}, \left[ \frac{N}{2} \right] \mod \left( \frac{N}{F} \right) \right) \right).$$

but this does not seem to be correct: supposing $N$ and $F$ are multiples of two, such that $F > 2$, then the formula gives $\max(L_p) = N^2/(2P)$, irrespective of the value of $F$. In fact, a bit of elementary algebra (see e.g. Saltz [179]) will show that a good approximation to $\max(L_p)$ is:

$$\max(L_p) \approx \frac{N^2}{2P} \left( 1 + \frac{1}{F} \left( 1 - \frac{1}{F} \right) \right)$$

from which it can be seen that (noting that $N^2/(2P)$ is the average workload per processor)

$$\frac{\max(L_p - \bar{L})}{\bar{L}} = O\left( \frac{1}{F} \right).$$

Summarising the above discussion, it can be stated that theoretical considerations have not yet come to a satisfactory answer regarding the effect of raising the cycle frequency. The fact that such an effect exists is, however, well known from empirical studies.

One such study has been performed by Hanxleden and Scott [93]. The have compared different partitioning methods for the WaTor application, which is essentially a 2D Monte Carlo dynamical simulation. They also consider a two-dimensional cyclic blockwise distribution, finding a dependence of the load imbalance on the cycle frequency that can be approximated by $1/\sqrt{F_1 F_2}$, where $F_1$ and $F_2$ are the cycle frequencies in the first and second dimension respectively. This would indicate that the load imbalance is inversely proportional to the square root of the total number of blocks per processor, which is consistent with the result of Salmon [178] (Equation (2.8)). In this particular case, Salmon’s assumption that the workloads of the blocks are independent may seem to apply, but, as Fox et al. [73] note in discussing the WaTor application, high workload tends to be restricted to a limited part of the domain and therefore it is not likely that Salmon’s assumption is valid. Consequently, the apparent validity of Equation (2.8) may well be accidental.

Chrisochoides et al. [43] find that, when disregarding the large communication volume, cyclic distribution is among the best methods to get a balanced workload distribution for an irregular mesh, but they do not give details regarding the way in which they employ cyclic distribution for irregular problems. Most probably, they employ a scheme like the one presented by Chan et al. [38], where cyclic distribution is applied to the elements of the adjacency matrix of the irregular mesh, which does not directly translate into a spatial cyclic distribution of the vertices and furthermore generates indeed a tremendous communication volume. Cannataro et al. [31] report a reduction in execution time of 30%, obtained from raising the cycle frequency in a cyclic blockwise distribution of a grid for the simulation of lava flow, but they do not give details regarding the value of $F$ which they use.

The Size of the Interface

With regard to the interface size under cyclic blockwise distribution, and hence the communication requirements, it should be noted that each block on a given processor has its own
interface, so that the communication for each processor will be determined by the number of blocks on the processor and the size of the interface per block. For a one-dimensional index set, the size of the interface is proportional to $F$, as noted by Gupta and Banerjee [90]. Hence: increasing the cycle frequency will usually increase the size of the interface linearly with $F$ and increase the volume of data that must be communicated accordingly.

For two-dimensional index sets, Expression (2.4) can be used when replacing the number of blocks under normal blockwise distribution, $P$, with that under cyclic blockwise distribution, $PF$, and multiplying the entire expression by $F$ to account for the fact that every processor has to handle the communication for all the $F$ blocks that are assigned to it. This makes the interface for the 2D case in general proportional to $\sqrt{F}$, i.e. the number of blocks per processor in one dimension. The ratio of total interface size, $\sum_{p_1 \in [1,P]} |I_{p_1,p_2}|$, to the volume of each part, $|I_p|$, becomes:

$$\frac{\sum_{p_1 \in [1,P]} |I_{p_1,p_2}|}{|I_p|} \propto \sqrt{PF},$$

(2.11)

which is a generalisation of Equation (2.6).

Hanxleden and Scott [93] express the expectation that the product of $PF$ should be kept constant to maintain a constant ratio of communication to computation. But this would mean that, if the number of processors is increased, $F$ should decrease, which will make the load imbalance worse and lead to a less efficient computation. This is confirmed by the performance results of Hanxleden and Scott, which indicate that it is better to let $PF$ grow slightly with increasing $P$ to retain the benefits of a balanced computation.

**Concluding Remarks on Regular Distribution Methods**

To conclude this discussion of regular distribution methods, it can be stated that there is a rather extensive understanding of the effects of regular partitioning methods with respect to the size of the total interface. However, there is little generally applicable knowledge regarding the load imbalances that can occur when using such regular distribution methods and there is no conclusive theory on the balancing effect of raising the cycle frequency, even though such an effect has often been reported in literature. As long as there is no conclusive theory on this matter, it is unclear whether the balancing effect is in general strong enough to compensate for the rise in total interface size that is cause by raising the cycle frequency.

However, the theoretical results on regular distribution methods indicate that they are less suitable when the number of processors grows. On the one hand, the load imbalances tend to become more pronounced when the number of processors becomes larger, and on the other hand the size of the interfaces grows with respect to the number of computations per processor. But the latter effect is not a unique feature of regular distribution methods: in the next section it will be seen that the total interface size under variable block distribution is roughly proportional to $P^{(D-1)/D}$ ($D$ being the dimension of the workload function domain), whereas the average number of computations per processor is obviously proportional to $1/P$. Thus, the ratio of the interface per processor to the number of computations per processor under variable block distribution tends to behave like $P^{(D-1)/D}$ and consequently grows with $P$. 
2.3. The Effectiveness of Distribution Methods

Finally, the results that will be presented in Section 4.3 indicate that some load imbalance may at times be beneficial as it spreads the loading of the communication network over time. Hence, the load imbalances that remain under regular distribution methods can also be advantageous under certain conditions.

2.3.2 Variable Block Methods

By construction, variable block methods produce parts of approximately equal computational load and therefore load balancing is not really an issue in these methods, although a few more words will be spent on this below. Instead, the effectiveness of variable block methods is best quantified in terms of the communication they induce. Primarily, this means considering the size of the interface. But there are two more relevant indicators to be studied. The first of these is the number of neighbours of each part, the second is the effective distance between two processors that have to communicate according to the partitioning of the problem and the mapping of parts onto the processors. The discussion here will start with results on the size of the interface. The number of neighbours and the logical distance to those neighbours will be addressed towards the end of this section.

1) The Interface Size

Skewness

If the index set is an interval in \( \mathbb{N}^D \) for some \( D \), and if the divisions between parts are parallel hyperplanes in \( \mathbb{N}^D \), then the size of the interface can be studied relative to the size of an interface in a regular partitioning. This relative size is defined by Berger and Bohari [18] as Skewness:

Definition 7 The Skewness of a partitioning of an interval of \( \mathbb{N}^D \) into \( P_1 \times P_2 \times \ldots \times P_D \) blocks is the maximum size of a part in any dimension relative to the size in the same dimension of the same part in a blockwise partitioning into \( P_1 \times P_2 \times \ldots \times P_D \) blocks.

For example, consider the \( 2 \times 2 \) variable block partitioning given on the left of Figure 2.8. The corresponding \( 2 \times 2 \) regular partitioning is shown on the right of that figure. The width of all blocks in the variable block partitioning is identical in this example to that in the regular partitioning, but the height of part \( P_{2,1} \) is significantly larger than that of the corresponding part in the regular partitioning. The Skewness is the maximum ratio of the size in any dimension of a part in the variable block partitioning to that of the same part in the regular partitioning, which is 3/2 (for block \( P_{2,1} \)). This Skewness gives an indication of the relative size of the interface in the variable block partitioning in comparison to that in a regular partitioning. In the previous section, it was shown that extensive theoretical results exists on the sizes of interfaces under regular distributions and with additional information on Skewness, these results can be also used for variable block distributions.

For a variable block partitioning into \( P \) blocks (in one dimension), the Skewness \( Sk \) is bounded by:

\[
1 \leq Sk \leq P(1 - \frac{P-1}{N})
\]
Chapter 2. Partitioning for Parallel Processing; The State of the Technology

Figure 2.8: A variable block partitioning (left) and the corresponding regular partitioning (right). The variable block partitioning has a Skewness of 3/2.

(according to Berger and Bokhari [18]). The upper bound is reached when all blocks but one contain only one point in a particular direction and the remaining block contains the other points. Because the workload must be balanced, this would mean that the workload associated with the points in the large block must be very low compared to that in the small blocks. In practical cases, therefore, the Skewness will usually be lower. For the workload function defined by Expression (2.1), Thuné [194] finds:

\[ 1 < Sk < \left\{ \begin{array}{ll} 1 + \frac{(P-2) \tilde{w}_I - \tilde{w}_B}{\tilde{w}_I} \frac{2 \tilde{q}}{N} + O\left(\frac{1}{N^2}\right) & w_B \leq w_I \\ 1 + \frac{\tilde{w}_B - \tilde{w}_I}{\tilde{w}_I} \frac{2 \tilde{q}}{N} + O\left(\frac{1}{N^2}\right) & w_B > w_I \end{array} \right. \]

when distributing in one direction over \( P \) processors, where, as in Expression (2.1), \( \tilde{q} \) is the width of the domain-boundary, \( w_B \) is the workload in boundary points and \( w_I \) is the workload in interior points. This shows that, if the difference \( w_I - w_B \) is not too large and if the width of the domain-boundary is not too large, then the Skewness will not be too serious for relatively small numbers of processors. In Chapter 4, a more general result regarding the Skewness of a partitioning of a subset of \( \mathbb{N}^3 \) will be obtained.

Separators and Cutsets

In the remainder of this section on the interface size under variable block partitionings, the fact that the dependence graph is directed will be ignored and results will be given that hold for undirected graphs. Because the definition of interfaces (see Definition 6) is only valid for directed graphs, no results will be given on the actual interface size itself. Instead, indirect measures for the size of the interface will be given in terms of

1. the size of a vertex separator (or separator for short) denoted by \( |\mathcal{S}_v| \) below, i.e. the set of vertices that, when removed, splits the graph into two unconnected parts, and
2. the size of the cutset (also called the edge separator) denoted below by \( |\mathcal{E}_c| \) i.e. the number of edges between vertices in different parts.
The results on the sizes of separators and cutsets provide lower bounds on the interface size under variable block partitionings. This is because the partitioning that attains the smallest separator is not necessarily feasible from parallel processing point of view.

For example, consider a dependence graph in which a phase $\phi$ exists that consists of a single computation. Let there be no dependencies of computations after $\phi$ on computations before $\phi$. Then $\phi$ is a separator of this graph and it is indeed a very small one. But the partitioning that is induced by choosing this separator is not useful for parallel processing because the parts can not be processed in parallel. Therefore, the size of the smallest separator of this graph is an underestimate of the actual interface size in a feasible partitioning.

In a different way, the separator and cutset results also give a more immediate indication of interface sizes in finite difference computations (and also finite element computations). That is because in such applications, computations and their connections are usually associated with mesh points and their connections. Thus, a partitioning of the mesh, which can be represented by an undirected graph, can be interpreted as a partitioning of the computations. In this way, information about the separators and cutsets of grids gives an indication about the sizes of interfaces in finite difference computations.

The size of the separator also provides an indication of the load imbalance that may remain in a variable block decomposition. The fact that some load imbalance may occur even in a variable block decomposition comes from the fact that it may be impossible to choose a splitting plane with exactly the same number of vertices on either side. For example, consider a one-dimensional line of points with an odd total number of points; the load imbalance $\max(L_p - L)$ that remains in this case can not be smaller than 1/2 as the difference between the sizes of the parts is at least 1. A similar situation can arise in multiple dimensions, for instance in making a cut of an RCB partitioning. Then it may be necessary to allocate an entire row of points either to the part to the left or to that to the right of the cut.

In multiple dimensions, the size of the separator gives some indication of the size of load imbalances that arise when no cut plane exists that splits a set of computations exactly in half. For example, consider a graph of $N$ vertices, partitioned into two parts and a separator, such that the first part contains $N_1$ vertices, the second part contains $N_2$ vertices and the separator contains $N_s$ vertices. Suppose $N_1 < N_2 < N/2$. Then this partitioning can be turned into a bisection by coalescing the separator with the smallest of the two parts, which is the first part in this case. Then the load imbalance would be $\max(L_p - L) = N_1 + N_s - N/2$. In this way, the size of the separator can be used to obtain an impression of the load imbalance in a variable block partitioning.

Generally speaking, the separator theorems usually give bounds that are easily evaluated but very wide, whereas the cutset theorems give more accurate bounds that are difficult to evaluate. This makes the cutset theorems less applicable to obtain a quick estimate of the interface size that is to be expected when partitioning a given graph. Results on separator sizes will be discussed first, followed by a brief overview of results on cutset size.

Separator Theorems

Vavasis [201] finds an upper and a lower bound on the separator size. These limits use the
concept of locality in the graph, where locality is defined by the following definition:

**Definition 8 (Vavasis [201])** Let $\|\|$ denote some $l_p$ norm on $\mathbb{R}^3$. Let graph $G$ be embedded in $\mathbb{R}^3$, i.e. each vertex has a three-dimensional coordinate associated with it. Let $d$ be the minimum distance between any two connected vertices of $G$ with respect to $\|\|$ and let $d'$ be the maximum distance between any pair of vertices that are connected by an edge. Then $G$ is $r$-local if $d'/d < r$.

For example, consider a graph in which the vertices are the points in $[1, N] \times [1, N]$ for some $N$ and where points are connected to their left, right, upper and lower neighbour. Such a graph has locality one, since all edges have the same length under the usual 2-norm.

According to Vavasis [201], the separator in a variable block decomposition of an $r$-local graph has a lower bound given by

**Theorem 4 (Vavasis [201], Theorem 6.4)** Let $G$ be an $N$-vertex, $r$-local graph such that $N^{4/3}$ is an integer, $N^{4/3} > 4r$, and the vertices of $G$ embedded in $\mathbb{R}^3$ are the points $[1, N^{4/3}] \times [1, N^{4/3}] \times [1, N^{4/3}]$. Then, in order to partition $G$ into $(G_1, G_2, \ldots, G_p, G_s)$ such that $N/(2P) \leq |G_p| \leq N/P$ for $p = 1, \ldots, P$, the number of vertices in $G_s$ must be at least $0.025rN^{4/3}P^{1/3}$ assuming that $N/P > 16$, and $P > 8$.

and an upper bound given by

**Theorem 5 (Vavasis [201], Theorem 4.1)** Let $G$ be an $N$-vertex, $r$-local graph embedded in $\mathbb{R}^3$. Then there exists a variable block partitioning that partitions $G$ into $(G_1, G_2, \ldots, G_p, G_s)$ such that $|G_p| \leq N/P$ for $p = 1, \ldots, P$ and such that the size of $G_s$ is at most $39.4\beta N^{4/3}(P^{1/3} - 1)$ when $P$ is a power of 2, where $\beta$ is 1.71 when $r = 1$ and proportional to $r$ otherwise.

The range of separator sizes that are allowed by these theorems is still very large. The ratio of the largest necessary separator size (see Theorem 4) over the smallest one (see Theorem 5) is more than $2.5 \times 10^3$ for any value of $N$. Also, the assumptions that are used in Theorem 4 are quite strict and will usually not hold.

But both theorems assert that the size of the separator is $O(N^{4/3}P^{1/3})$ for a graph embedded in $\mathbb{R}^3$ which provides at least some indication of the interface size that is to be expected from a variable block decomposition. Table 2.2 shows that the value of $N^{(D-1)/D}P^{1/D}$, where $D$ is the dimension of the space in which the graph is embedded, is indeed a useful estimator for the size of the separator.

In $\mathbb{R}^2$ a better upper limit on the necessary separator size has been obtained by Lipton and Tarjan [129] for planar graphs, i.e. graphs that can be embedded in $\mathbb{R}^2$ such that edges do not cross:

**Theorem 6 (Lipton and Tarjan [129], Corollary 3)** Let $G$ be any $N$-vertex planar graph having nonnegative vertex costs summing to no more than one. Then a variable block partitioning exists that partitions the vertices of $G$ into three sets $G_1$, $G_2$ and $G_s$ such that no edge joins a vertex in $G_1$ with a vertex in $G_2$, neither $G_1$ nor $G_2$ has total cost exceeding $1/2$ and $G_s$ contains no more than $2\sqrt{2}\sqrt{N}/(1 - \sqrt{2}/3) \approx 15.4\sqrt{N}$ vertices.
2.3. The Effectiveness of Distribution Methods

This theorem could be extended to a partitioning into any number of parts that is a power of two by applying it recursively. In any case, the limit provided by Theorem 6 can be wide, as can be seen by considering a square grid of $\sqrt{N} \times \sqrt{N}$ points of equal cost, which can clearly be partitioned into two parts by a separator of exactly $\sqrt{N}$ points, which is only about 6% of the limit predicted by Theorem 6. Similar theorems for other types of graphs have been developed by Alon et al. [5], Gilbert et al. [78, 80] and by Miller et al. [144].

Note that the order of the separator size as expressed in terms of $N$ is the same as that of the interface in regular partitionings (see Section 2.3.1). This seems to indicate that, when it comes to minimising the interface size, the gain from using a regular partitioning rather than a variable block partitioning will be largely independent from the problem size.

Some notation is required before stating the last of the separator theorems. This notation will also be used in the discussion of the cutset theorems below. First of all, let $G$ be an undirected graph, such that $G = (C, E)$, where $C = \{c_1, c_2, \ldots, c_N\}$ is the set of vertices and $E \subset C \times C$ is the set of edges. Let the adjacency matrix $A \in \mathcal{M}^{n \times n}$ of $G$ be defined by:

$$a_{ij} = \begin{cases} 1 & i \neq j \land (c_i, c_j) \in E \\ 0 & \text{otherwise} \end{cases}$$

and let $\lambda_i(M)$, $i = 1, \ldots, n$ denote the $i$-th smallest eigenvalue of some matrix $M \in \mathcal{M}^{n \times n}$, where $\mathcal{M}^{n \times n}$ is the set of all $n \times n$ matrices. Let $D \in \mathcal{M}^{n \times n}$ be a diagonal matrix in which entry $d_{ii}$ is the sum of the elements in row $i$ in the adjacency matrix $A$. Then the matrix $L \in \mathcal{M}^{n \times n} : L = D - A$ is called the Laplacian matrix. It is positive semi-definite, having a smallest eigenvalue $\lambda_1(L) = 0$.

The Laplacian matrix of a graph provides information about the minimum size of the separator, as is shown by a.o. Pothen et al. [162]. For binary splittings, they prove the following theorem:

**Theorem 7 (Pothen et al. [162], Theorem 3.3)** Let $G$ be an $N$-vertex graph with degree $\Delta$ and let $L$ be the Laplacian matrix of $G$. Then, in order to partition $G$ into $G_1$, $G_2$ and $G_3$, with vertex sets $C_1$, $C_2$, $C_3$ respectively, by a variable block partitioning, such that $|C_1|/|C_2| = a < 1$, assuming $|C_1| \geq |C_2|$, the size of $C_3$ must be at least:

$$\frac{|C_3|}{N} \geq \frac{(1 - a)\lambda_2(L)}{2\Delta - (\lambda_3(L) - \lambda_2(L))}$$

To consider one example of an application of this theorem, consider a rectangular grid of $N_1 \times N_2$ vertices. Pothen et al. [162] give an expression for the eigenvalues of the Laplacian of such a grid graph:

$$\lambda_{kl} = 4 \left\{ \sin^2 \left( \frac{(k - 1)\pi}{2N_1} \right) + \sin^2 \left( \frac{(l - 1)\pi}{2N_2} \right) \right\}, \quad k = 1, \ldots, N_1, \quad l = 1, \ldots, N_2$$

If $N_1 > N_2$ and $n \gg 1$ then the first two eigenvalues are 0 and $\approx \frac{\pi^2}{N_1^2}$. Furthermore, $\lambda_3 - \lambda_2$ is small with respect to $\Delta$ ($= 4$). Then, for a bisection such that $a = 1/2$, Theorem 7 evaluates to $|C_3| \geq \frac{\pi^2N_2}{16N_1} = O(1)$ if $N_2$ and $N_1$ are of the same order, ignoring higher order terms. Pothen et al. remark that indeed the bounds are not very tight except for particular
Table 2.2: The maximum number of edges that are cut or the maximum number of internal boundary vertices and the execution time needed for partitioning for several graphs as reported in literature.

| Graph No | Description                                                                 | $|E|$ | $|E_c|$ | $|E_c|$ | $N^{(D-1)/D}$ | $P^{1/D}$ | $lb$ | $ub$ |
|----------|------------------------------------------------------------------------------|------|--------|--------|----------------|-----------|-------|-------|
| 1        | 2D JAGMESH1 from Harwell-Boeing collection, Pothen et al. [162]              | 2680 | 13853  | 13853  | 85             | 73        |       |       |
| 2        | 2D True-3564, Hu and Blake [102]                                            | 3564 | 6320   | 6320   | 288            | 238       |       |       |
| 3        | 2D GRID80.80.5, Pothen et al. [162]                                         | 6400 | 6320   | 6320   | 8              | 113       |       |       |
| 4,5,6    | 2D airfoil problem, Simon [186]                                             | 11451| 16880  | 16880  | 91             | 151       | 2     | 59    |
| 7        | 2D airfoil barth5                                                           | 11451| 16880  | 16880  | 1154           | 856       |       |       |
| 8,9,10   | 2D airfoil barth5, results on RCB from from Ou et al. [153],                 | 15606| 45878  | 45878  | 245            | 200       | 177   | 6     |
| 11       | results on RIB and RSB from Leland and Hendrickson [123],                   | 15606| 45878  | 45878  | 245            | 200       | 177   | 6     |
| 12,13,14 | 3D carbody problem, Simon [186]                                             | 15606| 45878  | 45878  | 245            | 200       | 177   | 6     |

Cutset Theorems

The results on the size of the cutset are presently more accurate and general, although the evaluation of the lower and upper bounds is usually difficult. This is because, like the last separator theorem discussed above, they are often formulated in terms of eigenvalues of the adjacency matrix or similar matrices and such eigenvalues are not explicitly known for any classes of graphs.
but the most trivial graphs. But the results do give accurate estimates of the size of the optimal cutset and can therefore serve as a reference point when discussing the effectiveness of partitioning methods.

The first to give lower bounds on cutset sizes using the eigenvalues of the Laplacian were Donath and Hoffman [53], building on previous work by Hall [92]. They give a compactly formulated lower bound on the cutset when partitioning $G$ into $P$ disjoint subsets, each containing $N_p$ vertices:

**Theorem 8 (Donath and Hoffman [53], Theorem 1)**

$$|E_c| \geq \frac{1}{2} \sum_{p=1}^{P} N_p \lambda_p(L)$$

For a splitting of the $N_1 \times N_2$ grid graph mentioned above into two equally sized parts, Theorem 8 yields a lower limit for a binary splitting of approximately $\frac{N_1 N_2}{4N_1}$, which is no better than $O(1)$ if $N_1$ and $N_2$ are of the same order. This means that the cutset contains at least a few edges, which is not much of a surprise. Thus, even though this lower bound has the advantage of being relatively easy to evaluate, it is of little use in practice.

Recently, a more intricate but also more accurate lower bound was given by Falkner et al. [61]:

**Theorem 9 (Falkner et al. [61] Bound (3.5))** Let $A$ be the adjacency matrix of an undirected graph on $N$ vertices and let this graph be partitioned into $P$ equally-sized parts. Let $\bar{u}$ denote the unit vector in $\mathbb{R}^N$ along $(1, 1, ..., 1)^T$ and $sm(A)$ denote the sum of all entries in the matrix $A$, then

$$|E_c| \geq \frac{1}{2} sm(A) - \left[ \frac{sm(A)}{2P} + \min_{d \in \mathbb{R}^P} \frac{N}{2P} f(\bar{d}) \right]$$

where

$$f(\bar{d}) = \sum_{j=1}^{P-1} \lambda_j(V^T_N(A + diag(\bar{d}))V_N) - \frac{P-1}{N} \bar{d}^T \bar{u}$$

and $V_N$ is an orthogonal $N \times (N - 1)$ matrix in which the column vectors span the space $u^\perp$.

As $f(\bar{d})$ is convex, the minimum operation in the theorem can be evaluated by an iterative procedure, the details of which are given by Falkner et al. They also show how a partitioning can be obtained that is close to the optimum partitioning and in this way they are able to tightly limit the range of values that $|E_c|$ can attain. In Table 2.2, the columns $lb$ and $ub$ give, respectively, the lower bound as predicted by Theorem 9 and the cutset of a close-to-optimum partitioning as constructed by Falkner. Note that the cutset for the close-to-optimum partitioning is far superior even to the results of the RSB method, which are generally thought to be quite good.

Although Falkner’s procedure to find a close-to-optimum partitioning can be used to find a very good partitioning (comparable to the best results of simulated annealing as
found by Johnson et al. [107]), it is extremely time consuming and therefore not yet practically applicable. However, the partitionings that are generated in this way can be a good reference point to evaluate the effectiveness of more practical heuristics.

Concluding Remarks on the Interface Size

Table 2.2 gives a comparison of theoretical and practical results on the partitioning of finite element graphs. The separator theorems stated above are usually very wide, but they properly indicate the order of magnitude of the separator (i.e. $N^{(D-1)/D} P^{1/D}$). On the other hand, the edge separator theorem from Falkner et al. is very tight and most partitioning methods do not even come close to the optimal partitioning defined by that theorem.

The size of the interface that is produced by a partitioning of the computations of a program is not directly given by either the separator size or the cutset size. However, the separator and cutset theorems provide the best indication for the interface size that is available. Still, it seems fair to state that no more than the order of magnitude of the interface can be predicted beforehand because the separator and cutset theorems themselves are usually not very accurate for practical graphs.

II) The Number of Neighbours of a Part

At the beginning of this section on the variable block methods, it was already stated that the size of the interface is not the only relevant measure of effectiveness for these methods. Often equally important are the number of parts that are the neighbour of a given part and the distance between two parts that are connected by an interface as mapped onto a parallel processing system. With regard to the number of neighbours of a given part, there seem to be no theoretical results.

However, the impact of the number of neighbours can be significant, as was noted e.g. by Roest et al. [172] (see also Section 4.3 of this thesis). Table 2.3 gives some experimental results taken from literature. In a regular partitioning, the average number of neighbours of a part, $N_b$, is about four. Table 2.3 shows that the connectivity between parts in a variable block partitioning is usually not much higher than in a regular partitioning. These results will be confirmed by the studies in Chapter 4 of this thesis when studying the effectiveness of the RCB method.

III) The Distance between Communicating Processors

The final issue regarding the effectiveness of variable block partitionings is the distance between neighbouring parts as mapped onto a parallel computer. This issue is becoming less relevant with current developments in computer architecture, in which communication time is not so strongly related to the logical distance between the communicating processors, but may still be relevant on many parallel computers systems that are now in use. The distance between processors that hold neighbouring parts is best quantified in terms of the dilation of the partitioning. The concept of dilation is defined by Berger and Bokhari [18] for an RCB partitioning of a subset of $N^D$ on a mesh-connected computer. It can be defined more generally by the following definition:
2.3. The Effectiveness of Distribution Methods

Table 2.3: Empirical results regarding the average number of neighbours per part.

<table>
<thead>
<tr>
<th>Nb</th>
<th>average number of neighbours of a part</th>
<th>P</th>
<th>number of parts</th>
<th>RCB</th>
<th>Recursive Coordinate Bisection</th>
<th>RGB</th>
<th>Recursive Graph Bisection, with Cuthill-McKee ordering</th>
<th>RSB</th>
<th>Recursive Spectral Bisection</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>5</td>
<td>3.2</td>
<td>1.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>3.8</td>
<td>1.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>20</td>
<td>4.3</td>
<td>2.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>2.4</td>
<td>0.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Definition 9

Given a partitioning of an index set and a mapping of parts onto a set of processors, let $d_{\text{max}}$ be the maximum distance between two processors holding neighbouring parts then the dilation, $Dl$, of the partitioning is $d_{\text{max}} - 1$.

For partitionings of a subset of $\mathbb{N}^2$ using RCB, an expression of the worst case dilation on a mesh connected parallel computer has been given by Berger and Bokhari [18]:

$$Dl \leq \sqrt{P} - \left\lfloor \frac{\sqrt{P}}{Sk} \right\rfloor$$

Berger and Bokhari do not give a proof of this expression, but it can be seen to hold as follows.

Consider a square area of $N_1 \times N_1$ points that is partitioned into parts by a number of horizontal and vertical cuts, made according to the RCB method. Assume that there are $P = 2^{2k}$ parts for some integer $k$. Then there are $\sqrt{P}$ parts along any vertical line. Now it is clearly possible to shift the horizontal cuts along the vertical direction so as to create $\sqrt{P} - 1$ distinct lines that run straight across the entire domain in the horizontal direction. In the same way, the cutlines in the vertical direction can be aligned. This then gives a regular partitioning, that can be easily mapped onto a mesh-connected computer. In this way, a natural mapping of parts onto processors can be established.

With this mapping, shift the vertical cutlines back to their original position. It can be seen that a communication between two neighbouring parts can be done by one transmission in the vertical direction, followed by a number of hops in the horizontal direction. There are $\sqrt{P}$ parts along any horizontal line. There can be at most $\lfloor N_1/(Sk N_1/\sqrt{P}) \rfloor$ parts with horizontal size $(Sk N_1/\sqrt{P})$, i.e. the maximum horizontal size of any part in the partitioning. The remaining $\sqrt{P} - \lfloor N_1/(Sk N_1/\sqrt{P}) \rfloor$ parts are together horizontally
smaller than one part with maximum horizontal length and will, in worst case, be neighbours to one such maximum-sized block. Thus, the maximum horizontal distance in the mesh between two connected processors will be \(\sqrt{P - [N_1/(Sk N_1/\sqrt{P})]}\), which proves the above equation.

Cybenko and Allen give a similar theorem for mapping onto a hypercube. They prove the following Theorem:

**Theorem 10 (Cybenko and Allen [46], Theorem 2)** Two parts that share a cutting plane created in recursion depth \(d\) in a partitioning created by \(D\) recursions can be mapped onto a \(D\)-dimensional hypercube such that the distance between processors holding these parts is no more than

\[
1 + |\{k : d + 1 \leq k \leq D \land \text{direction}_{\text{cutplane}} d \neq \text{direction}_{\text{cutplane}} k\}|
\]

This distance is by definition \(1 + Dl\), so that this theorem provides an upper bound on the dilation, \(Dl\), of an RCB partitioning on a hypercube. Moreover, the proof that is given by Cybenko and Allen makes no use of any assumptions regarding the structure of the graph that is partitioned are therefore seems to hold also for irregular graphs. Note though that for arbitrary graphs the distance between parts that share a cutplane is not necessarily a good measure of the distance between communicating parts, since vertices at any distance may be connected.

Apart from these theorems, there are no general results on the dilation known to the author. Of course, dilation is only partly determined by the partitioning; the mapping of parts onto processors and the processor interconnection topology are equally important. Obviously, the quality of the partitioning constrains the quality of the possible mappings, but given a partitioning, finding a good mapping is mostly a non-trivial problem. A discussion of the mapping problem is beyond the scope of this work; more details on this subject can be found in papers by e.g. Bokhari [24, 25, 26], Chen and Shin [40], De Keyser and Roose [115] and Shen [185].

IV) Concluding Remarks on Variable Block Methods

In this section, the main results have been given regarding the interface size, number of neighbours and distance to the neighbours of any part in a variable block partitioning. The results on the interface size are either very crude or difficult to evaluate, so that their practical usefulness is limited. What remains to be done is to develop methods that will quickly give a good indication of the interface size. For example, Table 2.2 shows that the quantity \(N^{(D-1)/D}P^{1/D}\) often gives a surprisingly good indication of the size of the interface for the problems that were considered. An indication of the problem properties that can make this quantity a less reliable predictor of the interface size would help to identify which problems produce particularly large or particularly small interfaces. Such an understanding would be quite helpful in assessing the quality of the result of a partitioning. Also, it would help to identify situations where finding a good partitioning is extremely difficult, so that extensive optimisation of the partitioning may be required.

With regard to the number of neighbours of a part in a variable block partitioning and the distance to those neighbours, the results are scarce. It seems justified to state that there is no practically applicable knowledge in this respect.
2.4 Implementation Aspects

Table 2.4: Empirical results of partitioning graphs with clustering methods.

| N   | \( \overline{N_b} \) | \( |E| \) | P   | \( |E_c| \) | \( N_b \) | \( |E_c| \) | \( Nb \) | \( |E| \) | \( Nb \) | \( N^{(D-1)/D} \) | \( P^{1/D} \) |
|-----|-------------------|------|-----|--------|--------|--------|------|--------|------|--------|-----------|---------|
| 1   | 3564              | 16   | 16  | 263    | 250    | 250    | 32   | 64     | 125  | 110    | 238       | 56      |
| 2   | 250               | 312  | 2   | 35     | 32     | 32     | 999  | 238    | 56   | 141    | 999       | 999     |
| 3   | 500               | 625  | 2   | 64     | 57     | 57     | 141  | 999    | 56   | 141    | 999       | 999     |
| 4   | 1000              | 1250 | 2   | 125    | 110    | 110    | 110  | 110    | 110  | 110    | 110       | 110     |
| 5   | 15606             | 45878| 64  | 4046   | 5.34   | 5.34   | 110  | 110    | 110  | 110    | 110       | 110     |

2.3.3 Clustering Methods

There seems to be little theory regarding the effects of clustering methods. Therefore, only empirical results can be given, whose value is clearly limited by the fact that there is no reason to assume that these results are somehow representative of what could happen in other cases. Moreover, many clustering methods have a stochastic component to them, so that results are only meaningful when studying the ensemble of a sufficiently large number of runs as is done e.g. by Johnson et al. [107].

Table 2.4 gives a short list of some empirical results with the most well-known clustering algorithms. The scarcity of results is immediately obvious from this table and therefore it is impossible to make any definite statements on the effectiveness of these methods. Some of the results from this table can be compared to results for the same problems from variable block methods (see Table 2.2), showing that the clustering methods perform well when compared to the variable block methods.

2.4 Implementation Aspects

Obviously, a very good partitioning method can still be of little use if there is no software support for it. Therefore, it is important to consider the possibilities for devising software support for each of the partitioning methods that have been described above. The kind of support that is needed is closely related to the class of methods to which a particular partitioning method belongs. Regular partitionings can be easily implemented in data parallel languages and are indeed often found to be supported in such languages. Variable block methods require a more intelligent support but still this support can make use of the fairly regular properties of the partitionings. The final class, the so-called clustering algorithms, allow a vertex in a graph to be assigned to any part and therefore require the
most extensive run-time support. However, most of the support that is used for variable block partitionings (i.e. the inspector-executor mechanism and low-level communication libraries) can also be applied for cluster partitionings and therefore, the discussion of support for cluster partitionings will be very brief.

2.4.1 Software Support for Regular Partitionings

Regular partitionings are supported in most data parallel programming languages, such as HPF [100], Vienna Fortran [29, 39, 215], Fortran D [2, 101], Pandore [6, 133] and Paragon [165]. Because of the regularity of the distribution, the transformations between global indices (i.e. indices as they are stated in the program) and local indices (i.e. indices that are used by the processor to address data in its local memory) can be generated automatically (see e.g. Bozkus et al. [28]). Zima and Chapman [216] give a good account of the required compiler technology.

For regular partitionings, the necessary communications are relatively easy to determine from a program code. Also, it is relatively easy to determine from a program whether load imbalances can be expected to occur. Therefore, it seems possible to choose a regular partitioning method automatically based on an analysis of the program that can also be done automatically. Procedures to do this have been described by Bixby et al. [23], Feautrier [66], Gupta and Banerjee [89, 90] or Palermo and Banerjee [154]. Their work shows clearly that the partitioning problem for parallel processing usually involves a trade-off between benefits and disadvantages of partitioning methods. This can be illustrated by a short discussion of the work of Gupta and Banerjee [89, 90].

They have developed methods to analyse a program and define constraints on the distribution of each of the arrays in the program. A constraint indicates a desired property of a distribution. Examples of constraints are: an indication that an array should be distributed to exploit parallelism or an indication that one array should be aligned to another array (more on the topic of alignment can be found in the work of Li and Chen [125]).

A cost is associated with each of these constraints, indicating the cost in performance that will be paid if the constraint is not fulfilled. The selection of distribution methods is then done by searching a selection of mutually non-conflicting constraints to comply with so that the total cost of those constraints that are not fulfilled is a minimum.

The costs that are associated with each constraint are, obviously, dependent on the architecture. Balasundaram et al. [10] describe a tool to obtain good cost estimates for constraints by studying the costs of basic communication operations on the target parallel system. For balancing the workload, the usual approach in procedures like those that were just described, is to choose a blocksize of one (i.e. \( F = N/P \) for a distribution of \( N \) points over \( P \) processors) when load imbalances are expected and to choose a cycle frequency of one otherwise. As was discussed in Section 2.2.1, choosing the cycle frequency as high as \( N/P \) may lead to a very high communication volume and is therefore often not a good choice. The results that will be presented in Chapter 3 allow for a more subtle choice of \( F \).
2.4. Implementation Aspects

2.4.2 Software Support for Variable Block Partitionings

Variable block partitionings usually make use of run-time information and therefore, a compiler can not explicitly insert communication routines as it lacks the necessary information. However, a successful way of solving this problem has emerged in recent years. The basic technique is due to Das et al. [48, 49], who developed the Inspector/Executor mechanism that is now at the heart of most parallel computing language support for irregular partitionings (see e.g. Brezany and Sipkova [29], Choudhary et al. [41], Mehrotra and Van Rosendale [140], Kennedy [112], Müller and Rühl [147], Ponnusamy et al. [160] and Saltz et al. [181]).

Each loop over the index set of a distributed array is preceded by a call to an inspector routine, that is inserted by the compiler. This inspector routine uses run-time information about the distribution of the arrays in the program to determine which communications must be performed to properly execute the loop. This information is then passed on to an executor routine that executes the loop and performs the necessary communications. Crucial to the success of the inspector/executor scheme is the intelligence of the inspector routine to recognise possible ways of reducing the overhead involved in both the inspector itself and in the executor. Examples of such optimisations are the reuse of existing copies of remote data, the combination of several communications into a single one or the reuse of results from earlier calls to inspector routines (see e.g. Das et al. [48] or Müller and Rühl [147]).

It is presently unclear how efficient this kind of support can be. Pommerell and Rühl [159] have reported an evaluation of the Oxygen compiler, which makes use of a version of the Inspector/Executor mechanism, and found a speedup of around 7 on a 64 processor CM-5 machine for several applications with an irregular structure. On an AP1000 parallel computer, they found a speedup of around 30 on 128 processors for the same applications. These results are obviously not too impressive, but may still be due to the nature of the code rather than to the overhead from the run-time system. Mukherjee et al. report speedups of around 20 on a 32 processor CM-5 system for several irregular applications using the CHAOS system, which is probably quite good.

In spite of the developments in data parallel programming language support for variable block partitionings, many present-day codes for large-scale parallel computations make use of communication libraries instead. Some of these libraries offer high-level support in which most of the details of communication are hidden.

In some cases, this is done by offering standard implementations for particular distributed data structures. This limits the freedom of the user to define his own distributed data structures, but when the library is used for a particular application domain, this is not necessarily a problem. Among such libraries are SOLEQS [14], the GMD communications library [96], Babbage [108], LOCO [114, 174], CLIC [151], PUL-SM [197] and DIME [212].

Other libraries offer more freedom in the definition of distributed data structures (e.g. genMP [7], the general routines in the GMD communications library [96], the SUPRENUM mapping and communications library [188], and ComLib [206]).

Finally, there is a class of libraries offering low-level communication operations, leaving it to the programmer to handle the details of collecting data into messages, sending
the messages, receiving them and unpacking the messages on the receiving side. Among such libraries are ICC [145], CCL [9], PVM [76] and MPI [141]. The advantage of these libraries is that they allow the programmer to fine-tune communication and hence to get optimal performance from a parallel processing system. However, for practical application, low-level libraries are mainly used as a basis for constructing support on a higher level (see e.g. Skjellum et al. [187]).

Apart from run-time support for variable block distributions, there are also implementations of the partitioning methods themselves available in partitioning tools. Among the best known tools are Chaco [97] and TOP/DOMDEC [65], which implement a wide range of partitioning methods. Limited in scope but not less useful are Metis [111] and Jostle [209]. Scotch [157] is one of the latest additions to this collection of partitioning tools. Tools like these perform the actual partitioning, but leave the actual use of the resulting partitioning to the programmer.

The PARPRE tool that is described in Chapter 5 of this thesis is also a partitioning tool, but in addition, it can restructure data structures in a program to allow for an efficient distribution of data across distributed memory.

2.4.3 Software Support for Cluster Partitionings

Most of the parallel programming language support that has been described above for variable block distributions is also suited to implement cluster partitionings. The inspector/executor mechanism assumes no structure in the partitioning at all and is therefore equally applicable to variable block distributions and to cluster distributions. Of the libraries that were mentioned in the previous section, some can also be used for arbitrary distributions. This holds in particular for the libraries described by Hempel and Ritzdorf [96], Solchenbach et al. [188] and Vollebregt et al. [206]. Obviously, the low-level communication libraries are also suited for use with cluster partitionings, but, as stated above, they put most of the burden of coordinating the communications on the user.
Chapter 3

Analysis of Partitioning Methods

3.1 Overview

In the second chapter of this thesis, a description has been given of a large number of distribution methods. In this chapter and the next, a closer look will be taken at two methods that seem to be particularly useful in the context of parallel coastal water simulation: Cyclic Blockwise Distribution and Recursive Coordinate Bisection.

Chapter 1 already introduced the particular problems of partitioning for finite difference computations in the context of coastal water simulation. Among these are the fact that the workload in each grid point may vary during the simulation (because of drying and flooding) and the fact that the grids for coastal regions usually have a complex shape. In the particular case of a package like TRIWAQ, which is used daily for a wide range of applications, the users can not be expected to determine a proper partitioning of each grid themselves. Thus, there is a need for robust partitioning methods, which can be expected to automatically produce good partitionings for any of the grids that are used with TRIWAQ.

In this respect, Cyclic Blockwise Distribution is a very interesting method: if the cycle frequency is set high enough, then the distribution of computational work over the processors remains more or less balanced, even though the workload in each gridpoint may vary in the course of the simulation. However, setting the cycle frequency too high will lead to unacceptable amounts of communication among the processors and thus to a poor parallel performance. Thus, it must be established whether it is possible to find a value for the cycle frequency that sufficiently balances the computational workload in coastal water simulation and at the same time keeps the amount of communication at an acceptable level.

In Section 3.2, an analysis of Cyclic Blockwise Distribution will be presented that shows that the effect of the cycle frequency is quite strong. Hence, relatively small values for the cycle frequency may be sufficient to balance the workload. The results that will be presented, have been published before by the author in [166] and [167].

Cyclic Blockwise Distribution has one more quality that makes it interesting for parallel coastal water simulation in general: it is supported by most current high performance computing languages like HPF, Vienna Fortran, Pandore and others. Thus, language support is already widely available for this method, so that no specific partitioning tools would have to be developed.

However, in the next chapter, it will be shown that the speedup that can be obtained
Chapter 3. Analysis of Partitioning Methods

from parallel processing for coastal water simulation is largely determined by interprocessor communication. This holds in particular on networks of workstations, where communication is relatively slow. Therefore, it is essential that communication is avoided as much as possible, and consequently, even small values of the cycle frequency are hardly acceptable.

Other partitioning methods do not have the property that they automatically balance workload variations that arise in the course of a simulation. However, for many coastal water simulations the amount of computations in each gridpoint remains more or less constant during the simulation. In that case, there is no real need to use Cyclic Blockwise Distribution and it is probably better to use a variable block method to handle the complex shape of the grid and at the same time keep the communication within limits. But the communication between processors can still become large if the parts have large interfaces. In Section 3.3, it will be shown that the size of the interface between parts that are determined by Recursive Coordinate Bisection (RCB) is not likely to become very large. Consequently, RCB can be expected to perform well as a robust, automatical partitioning method.

A disadvantage of variable block methods in general is that they are not yet supported in high performance computing languages. However, ideas exist on how to design a general language construct for such distribution methods (see e.g. Chapman et al. [39]) and significant advances have been made to implement support for them (see e.g. Das et al. [48]). Therefore, it is likely that variable block distribution methods will be supported in high performance computing languages in the near future.

After the analysis of Cyclic Blockwise Distribution and RCB in this chapter, some examples will be given to illustrate the results. A more complete account of the consequences of the results for coastal water simulation will be given in Chapter 4.

3.2 Analysis of (Cyclic) Blockwise Distribution

This section will start by defining Cyclic Blockwise Distribution (CBD) to introduce the notation that will be used. This definition differs from the one that was given in the previous chapter only in the way it is presented. Next, a theorem will be proved that states the effect of CBD on a sinusoidal workload function. This theorem then makes it possible to understand the effectiveness of Cyclic Blockwise Distribution for general workload functions, as is discussed in Section 3.2.3. It will be shown that the effectiveness of CBD is determined to a large extent by the workload function of the distribution problem to which it is applied.

3.2.1 Cyclic Blockwise Distribution

Consider a one-dimensional workload function $W(.)$ with domain $I = [1, N] \subseteq \mathbb{N}$. Applying Cyclic Blockwise Distribution with cycle frequency (i.e. the number of blocks per processor) $F$ over $P$ processors means that the set $I$ is partitioned into $F \cdot P$ subsets (blocks) $I_{p,f}$, where

$$I_{p,f} = \{i \in I \mid ((p - 1) + (f - 1) \cdot P) \frac{N}{PF} + 1 \leq i \leq (p + (f - 1) \cdot P) \frac{N}{PF}\}.$$
3.2. Analysis of (Cyclic) Blockwise Distribution

\[ p \in \{1, \ldots, P\}, \ f \in \{1, \ldots, \frac{N}{P}\} \]

These blocks are allocated onto the processors in such a way that the set of indices in \( I \) that are allocated onto a given processor \( p \) is:

\[ I_p = \bigcup_{f=1}^{F} I_{p,f} \]

The workload for processor \( p \) is now simply:

\[ L_p = \sum_{i \in I_p} W(i) \]

In the following, the term processor workload will be used to refer to \( L_p \). It will be assumed that \( PF \) is a divisor of \( N \). If not, the bounds of \( I_{p,f} \) as given in the definition above are not in \( \mathbb{N} \) and consequently are not properly defined. But the results also hold approximately if \( PF \) is not a divisor of \( N \), as long as \( N \gg PF \).

### 3.2.2 Maximum Workload for trigonometric workload functions

Consider the application of Cyclic Blockwise Distribution to a domain \([1, N]\) on which a workload function is defined that is given by:

\[ W(i) = \sin(2\pi i \frac{\omega}{N} + \phi) \]

Then the following lemma holds:

**Lemma 1** If Cyclic Blockwise Distribution is applied to a workload function given by \( W(i) = \sin(2\pi i \frac{\omega}{N} + \phi) \), \( \omega \in \{0, N - 1\} \subset \mathbb{N} \), then the amount of work on any processor can only be nonzero if \( \omega \) is a multiple of \( F \) but not of \( PF \)

**Proof**

Using simple algebra it can be shown that:

\[
L_p = \sum_{f=1}^{F} \sum_{i \in I_p} W(i) = \frac{\sin(\pi \omega) \sin(\pi \frac{PF}{F})}{\sin(\pi \frac{PF}{F})} \sin(2\pi \omega \left[ \frac{1}{2} - \frac{1}{2F} + \frac{p - \frac{1}{2}}{PF} + \frac{1}{2N} \right] + \phi)
\]

\[
= \begin{cases} 
(-1)^{k(F-1)}\frac{\sin(\pi \frac{PF}{F})}{\sin(\pi \frac{PF}{F})} \sin(2\pi \omega \left[ \frac{1}{2} - \frac{1}{2F} + \frac{p - k}{PF} + \frac{1}{2N} \right] + \phi) & \text{for } \omega = kF \text{ for some } k \in \mathbb{N} \\
0 & \text{elsewhere}
\end{cases}
\]

which shows that \( L_p \) equals zero for any \( \omega \) that is not a multiple of \( F \). If \( \omega \) is a multiple of both \( F \) and \( PF \) then the expression also evaluates to zero.
This lemma provides a first glimpse at how Cyclic Blockwise Distribution actually works. Almost every workload function can be expressed as a Fourier series, i.e. as a sum of a number of sinusoidal components. The variation of $L_p$ over the processors is determined by the sinusoidal components in $W(.)$ for which $\omega$ is a multiple of $F$. When $F$ is increased, the components in $W(.)$ that contribute to the variation in $L_p$ are components at higher $\omega$. When the high-frequency components in $W(.)$ are smaller than the low-frequency terms, then the variation in $L_p$ will decrease when $F$ is increased. In other words: the processor workloads will become better balanced.

The factor $F$ in the expression for $L_p$ above might suggest that $L_p$ grows linear with $F$, which would largely undo any decreasing effect on $L_p$ from the decrease of the amplitude of $W(.)$’s Fourier coefficients towards higher $\omega$. But this is not the case, as can be seen by noting that

$$\sin(\frac{\pi \omega}{PF}) = \frac{\pi \omega}{PF} + \mathcal{O}(\frac{1}{P^3F^3}) \quad \text{and} \quad \sin(\frac{\pi \omega}{N}) = \frac{\pi \omega}{N} + \mathcal{O}(\frac{1}{N^3})$$

so that, in the expression for $L_p$ above:

$$|L_p| \approx F\frac{\pi \omega}{PF} \frac{\sin(2\pi \omega \left[ \frac{1}{2} - \frac{1}{2F} + \frac{p - \frac{1}{2}}{PF} + \frac{1}{2N} \right])}{\pi \frac{N}{P} \sin(2\pi \omega \left[ \frac{1}{2} - \frac{1}{2F} + \frac{p - \frac{1}{2}}{PF} + \frac{1}{2N} \right])}$$

which is largely independent from $F$. Thus, for a general workload function $W(.)$, the balancing effect on $L_p$ from an increase in $F$ is a direct reflection of the decrease in Fourier coefficients of $W(.)$ towards higher $\omega$.

### 3.2.3 Maximum Workload for general workload functions

In this section, a theorem will be stated and proved that gives an indication of the effectiveness of Cyclic Blockwise Distribution when applied to a general workload function $W(.)$ defined on a one-dimensional interval $[1, N]$. The workload function $W(.)$ is assumed to be sufficiently well behaved to allow its representation as a Fourier series:

$$W(i) = \sum_{\omega=0}^{N-1} v(\omega) \exp(2\pi i \frac{\omega}{N} i + i\phi(\omega))$$

where $v(\omega) \in \mathbb{R}^+$ are the amplitudes of the Fourier coefficients, $\phi(\omega) \in [0, 2\pi)$ are their phases, $i$ is the imaginary unit and $\omega$ is the frequency. Because $W(.)$ is a real function, $v(\omega) = v(N - \omega)$, and $\phi(\omega) = -\phi(N - \omega)$. Furthermore, it will be assumed that the amplitudes of the Fourier coefficients are bounded from above by

$$v(\omega) < \frac{C_1}{\omega^\alpha}$$

where $\alpha$ is a parameter that characterises the workload function. For many of the workload functions that do not comply to this assumption, it is possible to prove theorems similar
3.2. Analysis of (Cyclic) Blockwise Distribution

to those in this chapter. Thus, the work that is presented here is applicable also to other kinds of workload functions. This issue will be addressed again after the proof of Theorem 11.

In this section, the load imbalance that results from Cyclic Blockwise Distribution will be considered. The load imbalance is expressed as the difference between the maximum processor workload and the average processor workload for a single phase in a program (for the concept of phases, see Section 2.1). This quantity will be denoted by $\max_{p \in 1 \ldots P} (L_p - \overline{L}_p)$, where $\overline{L}_p$ denotes the average processor workload (see also Section 2.1.3 for a discussion of the concept of load imbalance).

The theorem below will provide an upper bound on the load imbalance. This upper bound is inversely proportional to $F^\alpha$, which means that if $\alpha$ is sufficiently large, then the load imbalance will be quickly reduced by increasing $F$. Note that $\alpha$ is a property of the workload function. In general, a high value of $\alpha$ occurs when the workload function varies smoothly and a low value of $\alpha$ occurs for workload functions with rapid fluctuations. Thus, for very smooth workload functions, CBD will be a very effective method.

The upper bound on $\max_{p \in 1 \ldots P} (L_p - \overline{L}_p)$ is established by the following theorem:

**Theorem 11** Let $W(.) : [1, N] \rightarrow \mathbb{R}$ be a real deterministic one-dimensional workload function and let $a(\omega) \omega = 0, \ldots, N - 1$ be its Fourier coefficients, where $a(\omega) = v(\omega) \exp(i\phi(\omega))$, with $v(\omega) \in \mathbb{R}^+$, $\phi \in [0, 2\pi)$. Let $v(\omega) < C_1/\omega^\alpha$ for some $C_1$ and $\alpha \geq 0$ and $\omega > 0$. Then

$$\max_{p \in 1 \ldots P} (L_p - \overline{L}_p) < \frac{C_2}{F^\alpha}, \quad F \geq 1$$

where $C_2$ is only marginally dependent on $F$.

**Proof:**

Let

$$\kappa(F, k, p, P, N) = (-1)^{k(F-1)} F \sin(\pi \frac{k}{F}) \sin(2\pi k F \left[ \frac{1}{2} - \frac{1}{2F} + \frac{p - \frac{1}{2}}{PF} + \frac{1}{2N} \right] + \phi(kF)).$$

Then, using Lemma 1,

$$\max_{p} (L_p - \overline{L}_p) < \max_{p} \left( \sum_{k=1}^{\frac{N}{F}} \frac{C_1}{(kF)^\alpha} \kappa(F, k, p, P, N, \phi(kF)) \right)$$

$$= \frac{1}{F^\alpha} \max_{p} \left( \sum_{k=1}^{\frac{N}{F}} \frac{C_1}{k^\alpha} \kappa(F, k, p, P, N, \phi(kF)) \right).$$

The fact that $\kappa(F, k, p, P, N, \phi(kF))$ has only a very weak dependence on $F$ (as discussed above), concludes the proof of the theorem.
Note that even for a workload function that complies with the assumptions, the theorem does not predict the effect of Cyclic Blockwise Distribution exactly. Increasing $F$ may still lead to an increase in $\max(L_p - \bar{L}_p)$ for particular values of $F$ and, on the other hand, a decrease of the load imbalance may be faster than $1/F^\alpha$. This is because the theorem ignores the exact form of the Fourier transform (it only uses an upper bound on the amplitude of the Fourier coefficients). However, for workload functions with a Fourier transform for which the upper bound somehow reflects the behaviour of the Fourier transform itself, the observed behaviour will be quite accurately predicted by the theorem. Examples will be encountered in Section 3.2.5.

The class of workload functions to which the theorem applies directly is rather wide: it includes all workload functions that consist of superpositions of block functions with sufficiently wide blocks. Moreover, the line of proof can be applied to any workload function with decreasing Fourier coefficients, leading to similar results.

The relevance of this theorem lies in the fact that it shows that the workload function has a significant impact on the effect of the Cyclic Blockwise Distribution method and gives an indication as to what kind of effect can be expected. If $\alpha$ is large, i.e. if the workload function is somehow smooth, then Cyclic Blockwise Distribution will be highly effective. If, on the other hand, the workload function is fluctuating rapidly, then $\alpha$ is probably small and Cyclic Blockwise Distribution will not be effective.

Furthermore, the line of argument shows that the maximum workload per processor is determined by the Fourier coefficients at frequencies that are multiples of $F$. This means that Cyclic Blockwise Distribution is sensitive to periodicity in the workload function. This fact was noted before by Nicol and Saltz [149], who suggested that periodicity in the workload function can make Cyclic Blockwise Distribution ineffective. But periodicity alone does not cause Cyclic Blockwise Distribution to fail. Consider a one-dimensional workload function on $N$ points, where $N$ is any power of 2. Let the workload function be given by:

$$W(i) = 1 + \sin(\phi + 2\pi \frac{8i}{N})$$

for arbitrary $\phi \in [0, 2\pi)$. Then this workload function displays a strong periodic behaviour, but still the workload will be perfectly balanced over the processors under a Cyclic Distribution for $F = 2^k$, $k > 3$. Thus, on top of being periodic, the workload function must also have slowly decreasing or non-decreasing Fourier coefficients for Cyclic Blockwise Distribution to fail in balancing the load.

### 3.2.4 Mean Square Deviation for General Workload Functions

The previous section has discussed an upper bound on the maximum workload per processor when using Cyclic Blockwise Distribution. Although that is indeed the most useful quantity to study when considering the effectiveness of CBD, it is also interesting to look at the mean square deviation of the processor loads. That gives an impression of the overall balance in a Cyclic Blockwise Distribution.

In this section, an upper bound on the mean square deviation will be given that is very similar to the upper bound on the maximum workload that was discussed in the previous
3.2. Analysis of (Cyclic) Blockwise Distribution

The theorem states that the mean square deviation of the processor workloads goes down inversely proportional to $F^2$ with increasing $F$ if the Fourier coefficients of the workload function decrease in amplitude faster than $1/\omega^2$.

The proof of the upper bound is somewhat more involved than that of Theorem 11 above and therefore it will be done in two steps. First a lemma will be stated, which will then be used in the proof of the theorem about the upper bound of the mean square deviation. The lemma states a property of a quantity that plays a central role in the proof of the theorem.

**Lemma 2** Let

$$\mathcal{K}_{v_a,\omega_a,\phi_a, v_b,\omega_b,\phi_b,F,N} = \frac{1}{P} \sum_{p=1}^{P} \sum_{f_1=1}^{F} \sum_{f_2=1}^{F} \sum_{j_1 \in I_p} \sum_{j_2 \in I_p} v_a v_b \sin(2\pi \omega_{a} j_{1} + \phi_{a}) \sin(2\pi \omega_{b} j_{2} + \phi_{b})$$

then

$$|\mathcal{K}| < v_a v_b F^2 \left| \frac{\sin\left(\frac{\pi \omega_a}{P}\right) \sin\left(\frac{\pi \omega_b}{N}\right)}{\sin\left(\frac{\pi \omega_a}{P}\right) \sin\left(\frac{\pi \omega_b}{N}\right)} \right|$$

if \( \omega_a = k_a F, \omega_b = k_b F \land k_a + k_b = mP \lor k_a - k_b = nP \)

$$\mathcal{K} = 0 \text{ elsewhere}$$

Furthermore, $\mathcal{K}$ is almost independent from $F$ for $\omega_a, \omega_b \ll N$.

**Proof:**

Using just simple algebra, the five-fold sum in the definition of $\mathcal{K}$ can be evaluated to

$$\mathcal{K} =$$

$$(-1)^{\omega_a + \omega_b + 1} v_a v_b \left\{ \cos(\pi \frac{\omega_a + \omega_b}{N} + \phi_a + \phi_b) \frac{\sin\left(\frac{\pi (\omega_a + \omega_b)}{P}\right)}{\sin\left(\frac{\pi (\omega_a + \omega_b)}{F}\right)} + \cos(\pi \frac{\omega_a - \omega_b}{N} + \phi_a - \phi_b) \frac{\sin\left(\frac{\pi (\omega_a - \omega_b)}{P}\right)}{\sin\left(\frac{\pi (\omega_a - \omega_b)}{F}\right)} \right\}$$

Using the fact that $\forall x: \cos(x) < 1$, this can be bounded by

$$|\mathcal{K}| < v_a v_b \frac{F^2}{2P} \left\{ \sin\left(\frac{\pi (\omega_a + \omega_b)}{P}\right) \frac{\sin\left(\frac{\pi (\omega_a - \omega_b)}{P}\right)}{\sin\left(\frac{\pi (\omega_a + \omega_b)}{F}\right)} + \sin\left(\frac{\pi (\omega_a - \omega_b)}{P}\right) \frac{\sin\left(\frac{\pi (\omega_a + \omega_b)}{P}\right)}{\sin\left(\frac{\pi (\omega_a - \omega_b)}{F}\right)} \right\}$$

which is zero everywhere except when $\omega_a, \omega_b$ are multiples of $F$. Then $\frac{\sin\left(\frac{\pi \omega}{F}\right)}{\sin\left(\frac{\pi \omega}{N}\right)}$ is $F$ for $\omega = \omega_a, \omega_b$, so that

$$|\mathcal{K}| < v_a v_b \frac{F^2}{2P} \left\{ \sin\left(\frac{\pi (\omega_a + \omega_b)}{F}\right) \frac{\sin\left(\frac{\pi (\omega_a - \omega_b)}{F}\right)}{\sin\left(\frac{\pi (\omega_a + \omega_b)}{F}\right)} + \sin\left(\frac{\pi (\omega_a - \omega_b)}{F}\right) \frac{\sin\left(\frac{\pi (\omega_a + \omega_b)}{F}\right)}{\sin\left(\frac{\pi (\omega_a - \omega_b)}{F}\right)} \right\}$$

But then also, $\omega_a + \omega_b, \omega_a - \omega_b$ are multiples of $F$, and $\mathcal{K}$ is still zero, except when $\omega_a + \omega_b$ or $\omega_a - \omega_b$ is also a multiple of $P$, so that

$$\frac{\sin\left(\frac{\pi (\omega_a + \omega_b)}{F}\right)}{\sin\left(\frac{\pi (\omega_a + \omega_b)}{P}\right)} = \frac{\sin\left(\frac{\pi (\omega_a - \omega_b)}{F}\right)}{\sin\left(\frac{\pi (\omega_a - \omega_b)}{P}\right)}$$

The fact that $\mathcal{K}$ is independent of $F$ except when $\omega_a = k_1 F$ or $\omega_b = k_2 F$ close to $N$, is easily seen by noting that, then

$$|\mathcal{K}| \lesssim v_a v_b \frac{F^2}{P} \frac{\sin\left(\frac{\pi \omega_a}{F}\right) \sin\left(\frac{\pi \omega_b}{F}\right)}{N^2} = v_a v_b \frac{N^2}{P} \frac{x^2}{k_1 k_2}$$
With this lemma, it is relatively easy to prove the following theorem about the mean square deviation of the processor workloads:

**Theorem 12** Let \( W(.) : [1, N] \to \mathbb{R} \) be a real deterministic one-dimensional workload function and let \( w(\omega) = \omega = 0, \ldots, N - 1 \) be its Fourier coefficients, where \( w(\omega) = v(\omega) \exp(i\phi(\omega)), \) with \( v \in \mathbb{R}^+, \phi \in [0, 2\pi) \). Let \( v(\omega) < C_1/\omega^\alpha \) for some \( C_1 \) and \( \alpha \geq 0 \) and \( \omega > 0 \). Then

\[
\frac{(L_p - \bar{L})^2}{F^2 \alpha} < C_2, \quad F \geq 1
\]

where \( C_2 \) is non-increasing with \( F \).

**Proof:**

The mean square deviation is given by

\[
\frac{1}{P} \sum_{p=1}^{P} (L_p - \bar{L})^2
\]

\[
= \frac{1}{P} \sum_{p=1}^{P} \sum_{f=1 \text{ or } f=2} \sum_{j=1}^{F/2} \sum_{j'=1}^{F/2} \sum_{\omega_a=0}^{N/2-1} \sum_{\omega_b=0}^{N/2-1} v_a v_b \sin(2\pi \omega_j j_1 + \theta_a) \sin(2\pi \omega_{j'} j_2 + \theta_b) - \frac{v_0^2}{P}
\]

using the fact that the summation over all \( \omega_a \) for \( \omega_b = 0 \) and over all \( \omega_b \) for \( \omega_a = 0 \) yields \( v_0^2 \). Using now Lemma 2, it can be seen that

\[
\frac{1}{P} \sum_{p=1}^{P} (L_p - \bar{L})^2 < \sum_{D(\omega_a, \omega_b)} v_a v_b F^2 \left| \frac{\sin\left(\frac{\tau a}{P}\right)}{\sin\left(\frac{\tau a}{N}\right)} \sin\left(\frac{\tau b}{P}\right) \sin\left(\frac{\tau b}{N}\right) \right| < \frac{1}{F^2 \alpha} \sum_{D(\omega_a, \omega_b)} (C_1 F)^2 \left| \frac{\sin\left(\frac{\tau a}{P}\right)}{\sin\left(\frac{\tau a}{N}\right)} \sin\left(\frac{\tau b}{P}\right) \sin\left(\frac{\tau b}{N}\right) \right|
\]

where \( D(\omega_a, \omega_b) \) is the set of all \( \omega_a, \omega_b \), where \( \omega_a = k_a F, \omega_b = k_b F \), and \( k_1 + k_2 = mP \) or \( k_1 - k_2 = nP \), with \( k_1, k_2, m, n \in \mathbb{N} \) and using the upper bound on \( v_\omega \). Using further the fact that the term in the summation is nearly independent of \( F \) (see the proof of Lemma 2) and that the size of \( D(\omega_a, \omega_b) \) decreases with \( F \), it follows that

\[
\frac{1}{P} \sum_{p=1}^{P} (L_p - \bar{L})^2 < \frac{C_2}{F^2 \alpha}
\]

as is stated by the theorem.
3.2. Analysis of (Cyclic) Blockwise Distribution

Figure 3.1: Left: $\max_p(L_p - \bar{L}_p)$ versus $F$ for phase $2k$ as defined in Section 2, for several values of $k$. The scaling has been done to make $C_1$ in the proof of Theorem 1 equal for all $k$. Right: $\text{var}(L_p)$ versus $F$ for phase $2k$ as defined in Section 1, for several values of $k$, and the upper bound predicted by Theorem 2. The scaling has been done in order to better compare the lines for different $k$, just like in the left part of the figure.

3.2.5 Examples

Example A: Gaussian Elimination

In Section 2.1.1 the Gaussian Elimination algorithm of Figure 2.1 was used to illustrate concepts such as precedence graphs, phases and workload functions. Consider the phase decomposition of this algorithm as given in Figure 2.2. Then phase $2k$ for $k \in \mathbb{N}$ (i.e. phase $2, 4, \ldots$) consists of the statements in lines 8 through 11 of Figure 2.1.

When considering the workload function of phase $2k$ of the Gaussian elimination algorithm along the $i$-direction, then it can be seen to be:

$$W_{2k}(i) = \begin{cases} 2(N - k) + 2 & \text{where } i \in [k + 1, N] \\ 0 & \text{elsewhere} \end{cases}$$

where the $2(N - k)$ term represents the additions and multiplications in line 9 (addition and multiplication each counting for a single unit of work) and the additional term 2 represents the addition and multiplication in line 10.

For $W_{2k}(\cdot)$ the amplitude of the coefficients of the Fourier transform is given by

$$|v(\omega)| = \left| 2 \cdot (N - k + 1) \frac{1}{N} \frac{\sin(\frac{\pi (N - k)}{N})}{\sin(\frac{\pi \omega}{N})} \right| < \frac{N - k + 1}{\omega}, \quad 0 \leq \omega \leq \frac{N}{2}$$

where the upper bound on the amplitude is found by noting that $\sin(\frac{\pi (N - k)}{N}) \leq 1$ and $\sin(\frac{\pi \omega}{N}) \geq 2\frac{\pi}{N}$ for $\omega \leq \frac{N}{2}$. Thus, the workload function matches the assumptions of
Chapter 3. Analysis of Partitioning Methods

Figure 3.2: Left: an artificial workload function; Right: the load imbalance versus the cycle frequency under Cyclic Blockwise Distribution.

Theorem 1 for \( a = 1 \) and \( C_1 = (N - k + 1) \). The fact that \( C_1 \) still depends on \( N \) and \( k \) and is therefore not a constant, is not relevant when considering the effect of \( F \) on the load imbalance. In Figure 3.1 the maximum deviation of an \( L_p \) in phase \( 2k \) from the mean \( \overline{L_p} \) is shown for several values of \( k \). It can be clearly seen that the deviation is indeed bounded from above in the way that is predicted by Theorem 11. Also shown is the variance of \( L_p \) for phase \( 2k \) for several values of \( k \) versus \( F \), which agrees with Theorem 12. However, the bound on the variance is not very tight because of the absolute values used in the summation in the proof of Theorem 12.

Example B: Constructed workload.

From Lemma 1 it is known that the load imbalance under Cyclic Blockwise Distribution with cycle frequency \( F \) is determined only by the frequency components in the Fourier transform of the workload function that are multiples of \( F \). This can be illustrated by considering the load imbalance under Cyclic Blockwise Distribution when applied to a workload function with a limited number of frequency components.

Figure 3.2 shows the results of applying Cyclic Blockwise Distribution to a workload function on 1024 points that is constructed from a Fourier representation of the following form:

\[
|\psi(\omega)| = \begin{cases} 
1 & \omega = 3, 6, 9, 12, 15, 18 \\
0 & \text{elsewhere}
\end{cases}
\]

The phase of each Fourier coefficient is chosen at random. As expected, the load imbalance over 8 processors using Cyclic Blockwise Distribution of the resulting workload function (shown in the left graph of Figure 3.2) is large whenever multiples of \( F \) coincide with one of the frequencies in the workload function (see right-hand graph in Figure 3.2). When \( F \) is larger than 18, the load imbalance is very small, which is to be expected since there are
3.3 Analysis of Recursive Coordinate Bisection

In Chapter 2, the Recursive Coordinate Bisection (RCB) method has been described in some detail. For a one-dimensional workload function $W(.)$ on the interval $[1, N]$ for some $N$, Recursive Coordinate Bisection starts by bisecting the domain of the workload function at location $b$ so that:

$$
\sum_{i=1}^{b} W(i) = \sum_{i=b+1}^{N} W(i).
$$

Next, bisection is applied recursively to the domains $[1, b]$ and $[b + 1, N]$, possibly with some reordering of the elements in the two sub domains. The recursion ends when the desired number of parts has been created. In the pure form of recursive bisection, the target number of parts must be a power of two due to the recursive nature of the algorithm.

The best known representative of Recursive Coordinate Bisection methods for domains in two or more dimensions is Orthogonal Recursive Bisection (ORB). When $W(.)$ is defined on a two-dimensional domain, then the first cut is made in the $y$-direction, the next in the $x$-direction, the third again in the $y$-direction and so on. For such a two-dimensional workload function, the location of the first cut in the $y$-direction is such that:

$$
\sum_{i_1=1}^{b} \sum_{i_2=1}^{N} W(i_1, i_2) = \sum_{i_1=b+1}^{N} \sum_{i_2=1}^{N} W(i_1, i_2)
$$

which is effectively a bisection of the domain of the one-dimensional workload function $W'(.)$ that is given by:

$$
W'(i) = \sum_{i_1=1}^{N} W(i, i_2)
$$

This shows that it is sufficient to consider workload functions on a one-dimensional domain when studying the effectiveness of RCB.

RCB gives by construction an (almost) balanced distribution of the workload. Therefore, the workload imbalance under RCB is not of interest. Instead, the quality of an RCB partitioning is best expressed in terms of the communication that is required under the partitioning. For this, the so called Skewness is a useful quantifier (see Section 2.3.2). The Skewness $Sk$ of an RCB partitioning is defined as the maximum size in any direction of any part divided by the size of each block in a normal, blockwise distribution. So, in the bisection just described, the Skewness would be:

$$
Sk = \frac{\max(b, N - b)}{N/2}.
$$
Chapter 3. Analysis of Partitioning Methods

The Skewness gives an indication of the maximum size of a message that is to be exchanged between two processors. Also, it puts an upper bound on the distance between processors that must communicate (see page 54ff).

In this section, two theorems about Skewness will be proved. The first is a rather obvious theorem that states that when the number of processors is increased, the Skewness will never decrease. The next theorem then provides an upper bound on the increase in Skewness when going from one cutting recursion to the next. This upper bound is formulated in terms of quantities that are characteristic of a workload function. The two theorems together give an upper and lower bound on the Skewness for a given workload function.

3.3.1 A Lower bound on the Skewness

The following theorem states that, for a given workload function, Skewness is non-decreasing as a function of the number of parts that it creates.

**Theorem 13** Let \( Sk(P) \) be the Skewness of an \( P \)-way RCB partitioning of the domain of a workload function \( W(.) \). Then

\[
Sk(2P) - Sk(P) \geq 0
\]

**Proof**

Let \( N \) be the size of a part under normal blockwise distribution of the domain of the workload function \( W(.) \). Then, a \( P \)-way RCB partitioning of the domain of the workload function has, by definition, at least one part with size \( Sk(P)N \). Let this part be bisected into two parts. The larger of these two parts has a size that is at least \( Sk(P)N/2 \), so that \( Sk(2P) \geq Sk(P)N/2/(N/2) = Sk(P) \), because \( N/2 \) is the size of a part under normal blockwise distribution. Consequently, \( Sk(2P) - Sk(P) \geq 0 \).

Since the Skewness in the first bisection is always at least 1, the consequence of this theorem is that the Skewness is at least equal to 1 for any number of parts and that, once it is larger than one, it will never become smaller again if RCB is applied to create more parts.

3.3.2 An Upper bound on the Skewness

In their discussion of RCB, Berger and Bokhari [18] stated that the Skewness in two dimensions can only attain values in the range \([1, \sqrt{p}(1 - (\sqrt{p} - 1)/N)]\), when \( p \) is an even power of two. In one dimension the same can be stated when replacing \( \sqrt{p} \) by \( p \). The upper bound is reached when all parts but one have a size of only 1, so that the remaining part has size \( N - (p - 1) \). However, it is obvious that it takes a very particular form of workload function to reach this upper bound. The following theorem gives a upper bound that will, in general, be much tighter than the one mentioned by Berger and Bokhari. It characterises a one-dimensional workload function on domain \([1, N]\) by:

- its average value \( a \)
3.3. Analysis of Recursive Coordinate Bisection

Figure 3.3: A workload function as used in the proof of Theorem 14

- the function \( v(.) \) which is the deviation of the workload function from its average value,
- the sum of the positive part of \( v(.) \) over the entire domain, denoted by \( V/2 \), and
- the maximum and average value of the positive part of \( v(.) \).

All these quantities together determine the upper bound on the Skewness that arises from a bisection. Primarily, the theorem states that the Skewness is bounded by the ratio \( V/aN \). Thus: a workload function with a high average value and small variations will have a small Skewness. The maximum and average of the positive part of \( v(.) \) serve to tighten the upper bound in particular cases.

**Theorem 14** Let \( W(.) \) be a one-dimensional workload function on \([1,N]\) with average value \( a \) and let \( W(i) = a + v(i) \). Let

\[
 v_+(i) = \begin{cases} v(i) & \text{where } v(i) > 0 \\ 0 & \text{elsewhere} \end{cases} \quad v_-(i) = \begin{cases} v(i) & \text{where } v(i) < 0 \\ 0 & \text{elsewhere} \end{cases}
\]

Finally, let \( V = \sum_{i=1}^{N} |v(i)| \) and \( m = \max v_+(i)/\text{average}(v_+(.)) \), then the Skewness \( Sk \) of the bisection of the \( W(.) \) is bounded by:

\[
 Sk < \begin{cases} 1 + \frac{V}{aN} & \text{if } \frac{V}{aN} < 1, m > \frac{2}{1-\frac{V}{aN}} \\ 1 + \frac{m}{m+2/\frac{V}{aN}} & \text{elsewhere} \end{cases}
\]

Furthermore, \( 0 \leq \frac{V}{aN} < 2 \) and \( 1 \leq m \leq N \).

**Proof**

First, it will be shown that (1): \( 0 \leq \frac{V}{aN} < 2 \) and (2): \( 1 \leq m \leq N \). Then, it will be shown that (3): not all combinations of \( \frac{V}{aN} \) and \( m \) can occur. With these facts, it can then be shown that (4): for a workload function that gives the highest possible Skewness, the Skewness is bounded as indicated by the theorem.

(1) Note that, \( \sum_{i=1}^{N} v_+(i) = -\sum_{i=1}^{N} v_-(i) = V/2 \). Now, because \( W(.) \) is a workload function, it is positive everywhere. Therefore, \( |v_-(i)| < a \) for all \( i \). Hence, \( V = 2\sum_{i=1}^{N} |v_-(i)| < 2aN \), so that \( \frac{V}{aN} < 2 \). As \( V \) can be zero, and \( V, a \) and \( N \) are positive, \( \frac{V}{aN} > 0 \).
(2) Let $\max$ be $\max_{i \in [1, N]} v_+(i)$ and let $\text{mean}$ be $\text{average}(v_+())$. Then $m = \max/\text{mean}$. Obviously, $\max \geq \text{mean}$, so that $1 \leq m$. The largest value of $\max$ is obtained if $v_+(k) = \max$ for only one $k$ in $[1, N]$ and $v_+(i) = 0$ for $i \neq k$. Then $\text{mean} = \max/N$ and so it must hold that $m \leq N$.

(3) Not all combinations of $\frac{\max}{\text{mean}}$ and $m$ are possible, which can be seen as follows. Let $z = \frac{\max}{aN}$, then

$$\sum_{i=1}^{N} |v_-(i)| = V/2 = aNz/2$$

Since $|v_-(i)| < a$, $v_-(i)$ is nonzero in at least $Nz/2$ locations, leaving at most $N - Nz/2$ locations for $v_+(i)$, to be nonzero. This means that $\max$ must be at least $\frac{V}{2(N - Nz/2)}$ in order to have $\sum_{i=1}^{N} v_+(i) = V/2$. Noting that $\text{mean} = V/2$, it must hold that

$$m = \frac{\max}{\text{mean}} \geq \frac{1}{1 - z/2} = \frac{1}{1 - \frac{V}{aN}}$$

(4) Now, the highest possible skewness is attained for a workload function like the one depicted in Figure 3.3. In this function, all nonzero elements of $v_+(.)$ have the value $\max$ and are all located to the far left of the domain. Likewise, all nonzero elements of $v_-(.)$ have their extreme value of $-a$ and are all located to the far right of the domain. In that case, $v(.)$ is such that

$$v_+(i) = \begin{cases} 
\max & i \leq k = \frac{N - b}{2a} \\
0 & \text{elsewhere}
\end{cases}$$
$$v_-(i) = \begin{cases} 
-a & i \geq s
\\
0 & \text{elsewhere}
\end{cases}$$

where $1 \leq k < s \leq N$. Let $b \in \mathbb{R}$ be the location of the separator as determined by Orthogonal Recursive Bisection. Then two possibilities have to be considered:

1. $k < b < s$. This means that

$$b = \frac{N}{2} - \frac{V}{2a} > k = \frac{\max}{2a} \Rightarrow \max > \frac{V}{2} \Rightarrow \frac{1}{N - \frac{V}{a}} \Rightarrow \frac{\max}{\text{mean}} > \frac{1}{N - \frac{V}{a}}$$

$$\Rightarrow m > \frac{1}{1 - \frac{V}{aN}}$$

Also, because $b > 0$, it must hold that $\frac{V}{aN} < 1$. Now, because the total surface under $W(.)$ to the left of $b$ must be equal to that to the right of $b$, it must hold that

$$ba + V/2 = (N - b)a - V/2 \Rightarrow b = \frac{N}{2} - \frac{V}{2a}$$

so that

$$Sk = \frac{N - b}{N/2} = 1 + \frac{V}{aN}$$

2. $b < k$ which is when $\frac{V}{aN} > 1$ or $m < \frac{2}{1 - \frac{V}{aN}}$. In that case

$$ba + b\max = (N - b)a + (k - b)\max - V/2 \Rightarrow 2b(a + \max) = N\max - V/2$$

$$\Rightarrow 2b(a + \max) = N\max \Rightarrow b = \frac{Na}{2(a + \max)}$$

$$Sk = \frac{N - b}{N/2} = \frac{N/2 + N/2\frac{\max}{a + \max}}{N/2} = 1 + \frac{m}{\text{mean}} + m = 1 + \frac{m}{2aN} + m = 1 + \frac{m}{m + 2aNN}$$

Note that $b > s$ cannot occur, since then the surface under $W(.)$ to the left of $b$ will always be larger than that to the right of $b$, which is contrary to the way in which $b$ is constructed using Orthogonal Recursive Bisection.

From (1)...(4), the theorem follows immediately.
3.3. Analysis of Recursive Coordinate Bisection

In Figure 3.4, the upper bound on the skewness $Sk$ is plotted versus $m$ and $\frac{V}{aN}$. Several obvious properties of the skewness can be seen: if $V = 0$, then $Sk = 1$ and furthermore, $Sk < 2$ everywhere. In the right part of the figure, the lines of the upper bound for several, low values of $m$ are plotted. Note that $m$ indicates whether the high workload values are very localised or not: if $m$ is large then the workload function contains small regions where the workload is much higher than elsewhere. From Figure 3.4, it can be seen clearly that the value of $\frac{V}{aN}$ is the most important in determining the maximum skewness when $m > 1$. For $1 \leq m < 2$, the value of $m$ is equally important, but will never lead to an upper bound that is wider than $1 + \frac{V}{aN}$.

The upper bound is attained for workload functions of the sort that is sketched in Figure 3.3. If the variations in the workload are spread more evenly over the domain, then the Skewness will be lower and can even attain the lower bound. Thus, for a workload function characterised by $V$, $m$ and $aN$, all Skewnesses between 1 and the upper bound given by Theorem 14 are attainable.

3.3.3 Examples

**Example A: Gaussian Elimination**

Consider once more the Workload function $W_{2k}(.)$ of phase $2k$ in the Gaussian Elimination Algorithm (see Section 2.1, and Section 3.2.5).

The left graph of Figure 3.5 shows the Skewness that arises from bisection of the domain of $W_{2k}(.)$ for different $k$. Also shown are the absolute limit on the Skewness as given by Berger and Bokhari and the upper bound given by Theorem 14. The upper bound predicted by the theorem coincides with the Skewness that is actually measured. This is not surprising since the form of the workload function is precisely the form that leads to the most extreme Skewness (i.e. all values above average are equal to the maximum value
Chapter 3. Analysis of Partitioning Methods

Figure 3.5: The Skewness for bisection of workload function \( W_{2k}(.) \) compared to the upper bound from Theorem 14 and some other upper bounds (left). The Skewness for \( P \)-way distribution of the domain of workload function \( W_{2k}(.) \) for several values of \( P \) (right). The domain of \( W_{2k}(.) \) in these figures is \([1,64]\).

and they all occur to one side of the domain). In fact, the measured Skewness is a little bit larger than the predicted upper bound due to round-off errors. Note that the coarser upper bound of \( 1 + V/aN \), also given by the theorem, gives already a fair indication of the Skewness.

Consider next the two-dimensional workload function \( W'_{2k}(.) \), that describes the workload in the \( k \)-th pass through lines 8 and 9 of the algorithm in Figure 2.1. This workload function is given by

\[
W'_{2k}(i_1, i_2) = \begin{cases} 
2 & \text{where } k < i_1 \leq N, \; k \leq i_2 < N \\
0 & \text{elsewhere}
\end{cases}
\]

The right-hand graph of Figure 3.5 shows the Skewness that occurs when two-dimensional RCB is applied to the domain of \( W'_{2k}(.) \). Due to the diagonal-symmetry of the workload function, the Skewness for \( 2P \) equals that for \( P \) for each of these values of \( P \), which is why the Skewness for \( P = 4, 16, 64 \) is not plotted.

Note that for \( k = N - 1 \), the Skewness is approximately \( \sqrt{2P} \). This is readily explained using Theorem 14. In this case, the value of \( V/aN \) is close to two in each cutting recursion: there is only one point where \( W_{2k}(.) \) is non-zero and thus \( V \) is \( 2(N-1)/N \) and \( aN \) equals one. Furthermore, \( m \) is much larger than one. Consequently, the upper bound predicted by Theorem 14 is

\[
Sk < 1 + \frac{m}{m + 2aN/V} \approx 2.
\]

Repeated bisection thus produces first a Skewness of two (\( P = 2, 4 \)), then four (\( P = 8, 16 \)), then eight (\( P = 32, 64 \)), etc.
3.3. Analysis of Recursive Coordinate Bisection

If \( k = 1 \) then \( V/aN \) is almost zero and the Skewness in each bisection is close to one. In that case, repeated bisection does not produce a Skewness that deviates significantly from one.

Example B: An Arbitrary Workload

Theorem 14 predicts that, if the average value of a workload function is high with respect to the range of variations, then the Skewness is low. This effect can be illustrated by showing the effect of raising the average value of a workload function with respect to the range of variations. Figure 3.6 shows on the left an arbitrary workload function and on the right the effect on the Skewness of bisection of raising the average value by adding a positive constant to the workload in each point of the domain.

Consider the left graph in Figure 3.6. Clearly, the upper bound given by Theorem 14 is close to the maximum possible value when the variations are large with respect to the average value. But if the range of variations reduces to the order of 10% of the average value, then the upper bound quickly decreases, coming close to the minimum possible Skewness when the variations are in the order of 1% of the average. So, when the variations are small with respect to the average, the upper bound given by Theorem 14 clearly indicates that the Skewness can only be small.

It is clear though, that the upper bound is rather far from the Skewness that is actually observed for this particular workload function. This shows that the upper bound does not necessarily reflect the actual behaviour of the Skewness. Being an upper bound, it can only be used to determine the range of values that the Skewness can take in a particular situation, not to estimate the value that it actually will take.
Chapter 3. Analysis of Partitioning Methods
Chapter 4

Partitioning Shallow Water Computations for Coastal Areas

In the previous chapters, the concept of workload functions has been defined and this concept has been used to obtain theoretical results on the effectiveness of Cyclic Blockwise Distribution (CBD) and Orthogonal Recursive Bisection (ORB). This chapter will discuss the effectiveness of CBD and ORB for a particular application to show the practical implications of the theoretical results.

Along the way, the discussion will illustrate a general procedure for studying the effectiveness of partitioning methods for a given application. The procedure is roughly the following: first the application is studied to determine what kind of workload functions are typical for it. If the effectiveness of candidate partitioning methods for these kinds of workload functions is known, then the applicability of those partitioning methods for the application can be quickly assessed. If not, then the effectiveness of the partitioning methods for the application's workload functions must be studied first. The advantages of this procedure will be discussed below.

The application that will be discussed in this chapter is the simulation of flow and transport processes in coastal waters. This application is interesting as a case study because it poses a difficult partitioning problem whose proper solution has significant practical relevance. This relevance comes from the fact that coastal water flow simulation itself has important practical applications. As parallel processing is required to perform the most accurate simulations of coastal waters, an efficient parallel implementation is of great interest.

For example, coastal water flow simulation is relevant for countries that rely on an extensive system of dams and dikes to prevent parts of the country from flooding. The construction of such water works requires accurate predictions of the effects of proposed designs on flow patterns along the coast. These predictions make it possible to assess beforehand whether the water works will offer an effective protection and whether side-effects will occur, such as unwanted sand depletion. Apart from this, simulation of coastal water flow has a wide range of other applications, such as e.g. environmental protection in areas with intensive shipping.

Some of the more accurate models of flow and transport processes along coasts require very high computing power. To reach a high resolution, the mesh size must be small and
this leads to a high total number of mesh-points to cover a given area.

It has been shown that parallel computers can deliver the required performance for such large scale simulations, but implementing the models on parallel architectures is far from trivial. The simulation models that are used are often based on a finite-difference discretization of the shallow water equations. The obvious way to parallelise such models is by using mesh-partitioning. The mesh is partitioned into submeshes which are handled in parallel. But due to the irregular shape that the meshes often have, it is usually difficult to find a proper partitioning of the mesh that minimises the impact of interprocessor communication and at the same time leads to an acceptable load balance. The effectiveness of CBD and ORB for this problem is the topic of this chapter. It should be noted though that the partitioning problem is only one aspect of the parallelisation of coastal water simulation; more details regarding simulation algorithms and implementation aspects can be found in the thesis of Vollebregt [205].

As indicated above, the study of the effectiveness of CBD and ORB for coastal water simulation starts by identifying the workload functions that are typical for this application. In this case, the workload function is largely determined by the shape of the coastal region that is simulated. Therefore, a general description of coastal regions will be given first. With this description, a specification can be given of workload functions that are typical for coastal water simulation. All this is the topic of Section 4.1.

Then, Section 4.2 presents the results of a study into the effectiveness of CBD and ORB when applied to workload functions as they occur in coastal water flow simulation. The first part of this study consists of experiments to assess the average behaviour of the partitioning methods for these workload functions. This average behaviour is obtained by generating workload functions of the type that occur in coastal water simulation and determining how effective CBD and ORB are for each generated workload function. In the second part of the study, the effectiveness of CBD and ORB is studied for a few workload functions that are taken from actual coastal water simulations.

Finally, Section 4.3 will discuss practical experiences with Blockwise Distribution and ORB for simulation of flow in two different coastal areas on several different parallel computing platforms. These practical results serve to place the theoretical results into perspective: they confirm the basic conclusions from the theoretical study but at the same time emphasise the fact that theoretical considerations can only give a rough indication of the actual parallel performance that is observed in practice.

4.1 The Workload Function for Coastal Water Flow Simulation

The aim of this chapter is to try and make generally applicable statements regarding the effectiveness of CBD and ORB in the simulation of coastal water flow on parallel computers. This could be done by applying these partitioning methods in a few practical situations and then trying to infer the general applicability from the results. The problem with that approach is that it is difficult to tell whether the chosen practical situations are sufficiently typical to allow general conclusions to be drawn. Therefore, a somewhat different approach will be used here. First, a model will be developed that describes the workload functions that occur in coastal water flow simulation. The model is stochastic
4.1. The Workload Function for Coastal Water Flow Simulation

and has two parameters: one that is related to the roughness of the sea-floor in the coastal region and another that is related to the water level in the coastal area. Any real workload function for this type of application is considered to be a realisation of the model for a particular value of the parameters.

With this model, it is possible to study average case, worst-case and best-case behaviour of partitioning methods for workload functions of the kinds that occur in coastal water simulation. By varying the value of the parameter, the properties of the workload function can be varied and the impact of that on the CBD and ORB partitionings can be studied. Provided that the model suitably describes practical workload functions for coastal water simulation, it is a powerful tool in studying the effectiveness of the two partitioning methods for workload functions as they occur in coastal water simulation.

The discussion here will focus on coastal water simulations that use a regular discretization of the coastal region, such as finite difference discretizations or rectangular finite element discretizations. Also, it will be assumed that natural coordinates are used rather than some form of boundary conforming coordinatization. Under these assumptions, the domain of the workload function will take the same shape as the coastal water that is being simulated. The shape of the coastal water is, in turn, determined by the topography of the coastal area. Thus, the definition of a model of workload functions for coastal water flow simulation will start in Section 4.1.1 with finding a description of the topography of coastal areas. Efficient algorithms to generate artificial topographies that conform to to this description are discussed in Section 4.1.2. Such artificial topographies will be needed in the experiments that are described later in this Chapter (Section 4.2).

Section 4.1.3 then discusses how two types of workload functions can be constructed from a given topography. These two types of workload functions are each characteristic for the one of the two kinds of workload functions that occur in coastal water simulation.

4.1.1 A Model for the Topography of Coastal Areas

As mentioned above, finding a way to describe coastal topographies (i.e. depth profiles) is the first step towards the specification of a model for the main workload functions in coastal water flow simulation. This section will discuss a way of describing coastal topographies that is not only fairly accurate, but is also very convenient in the current context. In the following, \( h(x) \) will denote the topographies of a coastal region. Two-dimensional coordinates will be denoted by \( x \) and one-dimensional coordinates by \( x \). The letter \( s^* \) will denote difference-vectors. The length of a vector \( x \) will be denoted by \( |x| \).

A very general model to describe semi-stochastic spatial functions, such as topographies, is (see e.g. Cressie [45]):

\[
h(x) = Z(x) + \sum_{k=1}^{N_k} \beta_k f_k(x)
\]

(4.1)

The summation term describes the deterministic part of \( h(x) \) and the \( Z(x) \)-term describes the stochastic part. \( Z(.) \) is assumed to be a random process. It is assumed to have zero mean and to be intrinsically stationary. The latter states that the increments of \( Z(.) \) from
one point to another have zero-mean, i.e.

\[ E[Z(\tilde{x} + \tilde{s}) - Z(\tilde{x})] = 0, \]  

(4.2)

where \( E[.] \) denotes the expected value of its argument, and that it has a semivariogram \( \gamma(\tilde{s}) \), defined as:

\[ \gamma(\tilde{s}) = \frac{1}{2} \text{var}(Z(\tilde{x} + \tilde{s}) - Z(\tilde{x})) \]  

(4.3)

The \( f_k(\cdot), k = 1, N_k \) in Equation (4.1) are a set of basic functions, often polynomials, and the \( \beta_k \) \( k = 1, N \) are the weights associated with each of the functions \( f_k(\cdot) \).

The general model that is given by Equation (4.1) is often used for spatial prediction, or kriging as it is also called. When applied in kriging, the details of the model (i.e. the specification of the process \( Z(\cdot) \) and the values of the \( \beta_k \)) are filled in by fitting the model with available measured data. Here, the general model is used in a somewhat different way: the aim is not to describe a single, given spatial data set, but rather to describe coastal topographies in general. Therefore, it must be determined which processes \( Z(\cdot) \) and which values of \( \beta_k \) can occur, given a set of \( f_k(\cdot) \).

This information can be obtained from the reports of the many authors that have worked on models for earth-surface topography. In the following, it will be assumed that under-water topographies are in essence identical to land topographies; any difference between land-surfaces and sea-floor surfaces that might exist, is ignored.

First, consider the values that can occur for the \( \beta_k \) in Equation (4.1) to describe topographies that are observed in practice. A very interesting study in this respect has been done by Basset and Chorley [13]. They have fitted a polynomial surface to a given topography. Up to a certain degree of matching, the polynomial surface is quite appropriate, but even the introduction of high-order terms does not seem to lead to a very good match. This is not too surprising: most topographies have variations on all scales, including very small ones, which can only be described by very high-order polynomials.

Most other authors have ignored the deterministic term in Equation (4.1) altogether or have removed it from the model by detrending observations of topographic height. In these cases, the model consists of the stochastic term \( Z(\cdot) \) only and therefore \( h(\cdot) \) itself is a stochastic process. Most reports regarding suitable processes \( h(\cdot) \) for describing topographies have restricted attention to one-dimensional cross-sections of topographies. Therefore, the current discussion will also limit itself to one-dimensional topography profiles. But the process \( h(\cdot) \) that will eventually be selected as the most suitable one can be extended to full two-dimensional topographies.

Processes that have been used in extensive studies of topographies are:

\[ h(x) = h(x - 1) + \epsilon_x + \phi \epsilon_{x-1} \]  

(4.4)

where \( x \in \mathbb{N} \), \( \epsilon_x \) is a stationary random process, and where \( \phi \) is a constant (see Bennet [17]).

\[ h(x) = h(x - 1) + \theta[h(x - 1) - h(x - 2)] + \epsilon_x \]  

(4.5)

where \( x \in \mathbb{N} \), \( \epsilon_x \) is a stationary random process, and where \( \theta \) is a constant (see Malinverno
4.1. The Workload Function for Coastal Water Flow Simulation

The Weierstrass-Mandelbrot fractal function:

\[ h(x) = \text{Re} \left[ \sum_{n=-\infty}^{+\infty} \frac{(1 - e^{i\mu_n x})e^{i\psi_n}}{\mu_n^H} \right] \]  

(4.6)

where \( \mu, H, \phi_n \in \mathbb{R} \), \( \mu > 1 \), \( 0 < H < 1 \) and where \( \psi_n \) are random numbers (see Burrough [30] and Berry and Lewis [22]) and

d) the fractional Brownian motion (fBm) model, introduced by Mandelbrot and Van Ness [138], i.e. a generalisation of the classical Brownian motion, which has stationary increments \( (h(x + s) - h(x)) \) with semivariogram \( \gamma(s) \propto |s|^{2H} \), where \( \propto \) denotes proportionality, and with \( 0 < H < 1, H \in \mathbb{R} \).

The fBm model has been found to be a better model to describe topographies than the model given by Equation (4.5) (see Malinverno [134]). But it is a rather complicated model from a mathematical point of view, and topography properties that are usually considered to indicate an fBm model (i.e. the semivariogram and the power spectrum) are equally well described by the much simpler Weierstrass-Mandelbrot model with \( \mu \) close to one. The model given by Equation (4.4) is essentially a classical Brownian motion model and is therefore covered by the fBm model and the Weierstrass-Mandelbrot fractal model with \( \mu \) close to one. Thus, the Weierstrass-Mandelbrot fractal model seems to be a fair choice for a generic model of topographies. In the following discussion, it will be briefly referred to as the fractal model or fractal function.

There is extensive evidence that the fractal model gives a fairly accurate description of topographies. It neatly describes many of the features that have often been observed in topographic data sets. The most important of these properties is the power spectral density \( S(\omega) \), i.e. the Fourier transform of the autocorrelation \( C(s) \), where

\[ C(s) = \lim_{X \to \infty} \frac{1}{2X} \int_{-X}^{X} h(x + s)h(x)dx. \]

Berry and Lewis [22] have shown that the Weierstrass-Mandelbrot function has a power spectral density \( S(\omega) \) that is approximately proportional to \( 1/\omega^{2H+1} \). Such a power spectrum has been observed for real topographies by many authors [16, 20, 21, 51, 71, 134, 158, 183, 198]. Crudely stated, it means that there are height variations on all spatial scales (i.e. all \( \omega \)), but that the rapid variations represent a relatively small portion of the total height variations. This is consistent with the findings of Basset and Chorley (see above).

A second feature of topographies that is correctly modelled by the fractal function, is the fact that the semivariogram \( \gamma(s) \) (see Equation (4.3)) of \( h(x) \) is approximately

\[ \gamma(s) \propto |s|^{2H}. \]

For \( H \) around 0.5, this would lead to \( \gamma(s) \propto |s| \), i.e. \( \gamma(s) \) is then approximately linear in \( s \). Figure 4.1 gives the semivariogram for two coastal topographies, showing that the observed semivariogram can indeed be approximated by a curve of the form \( |s|^{2H} \), at least when the
Figure 4.1: The semivariograms of the topography between 0 and 200 m depth of the West-European Continental Shelf (left) and that between 0 and 50 m depth of the Waddensea along the Dutch North-coast (right). The semivariograms were estimated using Cressie’s robust estimator (see Cressie, Eq. (2.2.8)).

The spatial scale is not too large. For the Continental Shelf topography, \( H \) is approximately 0.5, but for the Waddensea topography, \( H \) is somewhat larger.

The values of \( H \) that have been determined from the power spectra of real topographies are mostly around 0.5 [16, 30, 51, 134, 183, 198], but the variation can be quite significant (0 < \( H < 1 \) [21], 0.5 < \( H < 1 \) [158], or 0 < \( H < 2 \) [20]). The value of \( H \) has been shown to be related to the sea-floor roughness (see Berkson and Matthews [20] and Fox and Hayes [71] and also Figure 4.2 for an illustration of the effect of \( H \)).

The fractal model is conceptually appealing (see e.g. the discussion by Malinverno [134] or that by Turcotte [198]) because it has no intrinsic scale: it looks the same at any distance. This can be illustrated by the fact that,

\[
\gamma(c.s) = c^{2H} \gamma(s)
\]

for an arbitrary positive constant \( c \). Scale invariance is also a property of topographies (see e.g. Malinverno [134] or Turcotte [198]). Adding this to the fact that the Weierstrass-Mandelbrot fractal function correctly describes the power spectrum and the semivariogram, it seems fair to state that the fractal model is quite successful in describing topographies. This alone could motivate a choice for this model for the current purposes, but it has some additional properties that make it particularly attractive for use in this thesis.

First of all, the power-law behaviour of the spectrum fits neatly with the assumptions of the theorems in Chapter 3. This will make it easy later on to apply the theorems to workload functions that are based on the Weierstrass-Mandelbrot function. Secondly, it contains a parameter that can be varied to model different sea-floor roughnesses. In this way, the impact of different roughnesses can be studied. Finally, several very efficient algorithms exist to generate approximate Weierstrass-Mandelbrot functions (see Section 4.1.2), which makes it possible to validate theoretical predictions extensively through simulation.
Thus, apart from being an acceptable model in its own right, the Weierstrass-Mandelbrot function is also an attractive one for the current purposes. Therefore, it will be used in the remainder of this chapter. Nevertheless, a few words of caution are in place here. First of all, the fractal model is not unanimously accepted as a good model for topographic data. For instance, Lewin [124] states that the power spectrum of topographic data sets is approximately

\[ S(\omega) \propto 1/(c + \omega)^{2H+1} \]

where \( c \) is a constant. For large \( \omega \), this form is indiscernible from \( S(\omega) = 1/\omega^{2H+1} \), but for smaller \( \omega \) the difference can be significant. This would be consistent with the notion that the processes that determine long-range variations are different from those that determine short-range variations (see e.g. Parsons and Sclater [155]), so that the power spectrum for long wavelengths should differ from that for short wavelengths. Also, actual observations of long-wavelength topographic spectra (see Gilbert [82]) seem to confirm the fact that the spectrum is indeed not as simple as \( S(\omega) = 1/\omega^{2H+1} \). However, the fractal model is probably reasonably valid for wavelengths shorter than about 100 km (see e.g. Turcotte[198] or Gilbert [82]), provided the area under study does not possess special features such as rifts.

Also, the model is probably less appropriate when considering topographies that are largely determined by human constructions (for instance near major harbours). So, when using the fractal model below for generating artificial sea-floor topographies, it should be understood to apply only on scales below 100 km but above 1 km (which is assumed to be the range at which human activities and other small-scale processes become noticeable).

In the next section, algorithms will be discussed for generating approximate Weierstrass-Mandelbrot surfaces. The way in which the workload functions that are typical for coastal water simulation are constructed from a Weierstrass-Mandelbrot function is discussed in Section 4.1.3.

4.1.2 Algorithms for generating fractal surfaces

The fractal function that was introduced in the previous section as a one dimensional model, can easily be extended to higher dimensions. In two dimensions, its semivariogram \( \gamma(s_1, s_2) \) behaves like

\[ \gamma(s_1, s_2) \propto (s_1^2 + s_2^2)^{H/2} \tag{4.7} \]

and its power spectrum index \( S(\omega_1, \omega_2) \) behaves like

\[ S(\omega_1, \omega_2) \propto 1/(\omega_1^2 + \omega_2^2)^{(H+1)} \tag{4.8} \]

where \( s_1 \) and \( s_2 \) are the \( x \) and \( y \) components of a difference vector \( s \) and where \( \omega_1 \) and \( \omega_2 \) are the spatial frequencies in the \( x \) and \( y \) direction respectively. It is possible to generate an approximation to the two-dimensional fractal model by evaluating the two-dimensional variant of the Weierstrass-Mandelbrot function. Such a procedure has been described by Voss [207], but it is computationally very expensive. The problem is that the properties of
the Weierstrass-Mandelbrot function that were discussed in the previous section only hold when \( \mu \) is close to one. In that case the terms in the summation of Equation 4.6 decay only slowly with \( n \) and thus a large number of terms must be retained to find the value of the function with acceptable accuracy. Therefore, other algorithms will be reviewed here that produce approximations to the fractal model with \( \mu \) close to one. Some approximations are substantially better than others but usually the better approximations take more time to generate than less sophisticated ones. In this section, several algorithms for generating approximate fractal surfaces will be reviewed in order to select the most appropriate one for generating artificial topographies for the experiments that are described in Section 4.2.

Most algorithms to generate approximate fractal surfaces have been developed in the field of computer graphics applications (see e.g. Fournier et al. [70], Lewis [124], Mastin et al. [139], Peitgen and Saupe [156] and Voss [207]). Not all of them generate true fractal surfaces: some algorithms produce surfaces which look like fractal surfaces but in fact have substantially different characteristics. An example is the MidPointFM2D algorithm, originally published by Fournier et al. [70] and later by Peitgen and Saupe [156]. It is very attractive because of its low time-complexity of \( O(N^2) \) for generating a surface of \( N \times N \) points. But unfortunately, the generated surfaces are not fractal surfaces. For the triangular patches version of the algorithm, this was noted by Mandelbrot [136] immediately after the algorithm was first published. The square patches version (i.e. the version published by Peitgen and Saupe) suffers from the same drawbacks which will be discussed briefly here to demonstrate that this algorithm cannot be used and that more time-consuming algorithms are indeed necessary.

Let \( \text{Gauss}(\sigma^2) \) be a random number generator, that returns normally distributed random numbers with zero mean and variance \( \sigma^2 \). Then the MidPointFM2D algorithm starts as follows:

\[
\text{set the four corner points to } \text{Gauss}(\sigma^2) \\
\quad h(N/2,N/2) = \frac{1}{4}(h(1,1) + h(N,1) + h(1,N) + h(N,N) + \text{Gauss}(\sigma^2/2^H)) \\
\quad h(N/2,1) = \frac{1}{3}(h(1,1) + h(N/2,N/2) + h(N,1) + \text{Gauss}(\sigma^2/2^H))
\]

and proceeds likewise for the middle points of the three other sides (i.e. \( h(1,N/2), h(N,N/2) \) and \( h(N/2,N) \)). Then, essentially the same procedure is applied to each of the four quadrants of the square: first the value of the centre point is determined from the average of the four corner points, then the value of the midpoints of the four sides are determined. In this way, the algorithm proceeds recursively until all points have been assigned a value, each time adjusting the variance of the random number generator to match the scale of the sub-square. For details, the reader is referred to Peitgen and Saupe [156].

This brief description is sufficient to show why the algorithm is incorrect. Let \( H = 0.5 \), then the semivariogram \( \frac{1}{2} \text{var}(h(i_{11},i_{12}) - h(i_{21},i_{22})) \) should be proportional to \( \cdot \sqrt{(i_{21} - i_{11})^2 + (i_{22} - i_{12})^2} \) (see Equation (4.7)). But it is easily seen that:

\[
\text{var}(h(N,N) - h(1,1)) = 2\sigma^2 \\
\text{var}(h(N/2,N/2) - h(1,1)) = \frac{3 + 2\sqrt(2)}{4} \sigma^2 \approx 1.45\sigma^2 \\
\text{var}(h(N/2,1) - h(1,N/2)) = \frac{11}{9} \sigma^2 \approx 1.22\sigma^2
\]
which clearly demonstrates that the semivariogram is not that of an fractal surface and is also non-isotropic. Lewis [124] has published an improved version of the MidPointFM2D algorithm, called Generalised Stochastic Subdivision or GSS for short. It reduces the flaws of the MidPointFM2D algorithm at the cost of a much higher time complexity (although the time complexity remains $O(N^2)$). But the improvement in approximation quality requires a very careful treatment of points near the boundary, which makes the algorithm rather difficult to handle when really good approximations are required.

The most reliable methods that are known to generate fractal surfaces are two algorithms that were introduced by Mandelbrot: the Repeated Step Fault procedure (see Mandelbrot, [137], Chapter 28) and the SpectralSynthesis algorithm (see e.g. Voss [207]). The first has an unacceptably high time-complexity ($O(N^3)$), according to Fournier et al. [70], probably referring to a 1D version) and can not be used to generate fractal with $H \neq 1/2$.

The SpectralSynthesis, on the other hand, is a very powerful and relatively fast method. The algorithm starts by generating a Fourier representation of a fractal surface, $\mathcal{F}(\omega_1, \omega_2)$ with the power-law behaviour of a fractal spectrum (see Equation (4.8)) and then takes the inverse Fourier transform of $\mathcal{F}(\omega_1, \omega_2)$ to construct the fractal itself. The Fourier transform is generated by

$$
\mathcal{F}(\omega_1, \omega_2) = \frac{1}{(\omega_1^2 + \omega_2^2)^{(H+1)/2}} \epsilon_{\omega_1, \omega_2} e^{i\phi_{\omega_1, \omega_2}}
$$

where $\epsilon_{\omega_1, \omega_2}$ is a normally distributed random variable with zero mean and unit variance and $\phi_{\omega_1, \omega_2}$ is a random variable with a uniform distribution on $[0, 2\pi)$. Symmetry properties in $\mathcal{F}(\omega_1, \omega_2)$ are enforced to assure that the inverse transform is real. The time complexity of the algorithm for generating a fractal surface of $N \times N$ points is $O(N^2 \log(N))$. But the actual time complexity can still be quite high, because only a relatively small part of the generated surface can be used. This is because the generated surface is periodic with a period equal to the size of the surface. Consequently, if the surface has size $N \times N$, then $\text{var}(h(i_1, N-n) - h(i_1, 1))$ (which should be proportional to $(N-n-1)^{2H}$) is the same as $\text{var}(h(i_1, n) - h(i_1, 1))$ (which should be proportional to $(n-1)^{2H}$) for any $i_1 \in [1, N]$. When retaining only a fraction (1/2 or 1/4 in each dimension according to Voss [207]) of the generated surface, this effect becomes less noticeable. But obviously, this means that a fractal surface of a specified size requires that first a fractal surface of up to 16 times that size (in 2D) is generated, which is not really efficient. Also, it will prove to have serious consequences for the Fourier transform of the fractal surface (see Section 4.2).

An important point to note in selecting an algorithm to generate artificial topographies is the following. In the previous chapter, it has been shown that the Fourier transform of the topography largely determines the effectiveness of Cyclic Blockwise Distribution. Consequently, it seems important to have a correct Fourier transform for the artificial topographies that will be used for studying the effectiveness of CBD for coastal water simulations. This suggest the use of the SpectralSynthesis algorithm for generating the artificial topographies. Also, the time-complexity of $O(N^2 \log(N))$ is not too serious when $N$ is not taken extremely large. It has been found necessary to retain only 1/4 of the surface in each dimension to diminish the effect of the periodicity in the fractal surface. Figure 4.2 shows two examples of fractal surfaces, generated with the SpectralSynthesis algorithm.
4.1.3 Constructing a Workload Function from a Fractal Surface

Above, it has been shown that the topography of coastal waters can be described by the fractal model. This section will discuss the way in which the workload functions that occur in coastal water simulation can be derived from the topography of the coastal water that is simulated.

It has been stated before that the discretization is assumed to be rectilinear. Therefore, if $A' \times A'$ is the domain of the workload function, then the points in this domain represent $N \times N$ points of the coastal topography. For points in which the topography is negative with respect to some reference level, no computations have to be done since these points are effectively dry points (i.e. point situated on the land).

Points where the topography is positive (again with respect to some reference level) are wet points, i.e. points in which the water level is non-zero. Some computational work has to be done in these points. The amount of computational work depends on the way in which the depth of the water is treated in the simulation. The two main possibilities are:

a) the computational work is proportional to the depth or
b) the computational work is constant for all points where the depth is non-zero.

The first situation occurs when the water is modelled vertically (i.e. in the depth direction) by layers that have constant thickness across the domain. The second situation is typical of simulation in which depth-averaged values are computed or in which the water is modelled vertically by a constant number of layers whose thickness varies across the domain. In the first case, the workload function is:

$$W_1(i_1, i_2) = \begin{cases} 
    h(i, j) & \text{where } h(i_1, i_2) < \text{SeaLevel} \\
    0 & \text{where } h(i_1, i_2) \geq \text{SeaLevel} 
\end{cases} \quad i_1, i_2 \in [1, N]$$
4.2. The Effectiveness of CBD and ORB.

where SeaLevel is a constant that indicates the water level in the coastal area. For the case of a constant number of layers, the workload function is:

\[
W_2(i_1, i_2) = \begin{cases} 
1 & \text{where } h(i_1, i_2) < \text{SeaLevel} \\
0 & \text{where } h(i_1, i_2) \geq \text{SeaLevel}
\end{cases}
\]

These two workload functions will be used below for assessing the effectiveness of Cyclic Blockwise Distribution and Recursive Coordinate Bisection.

4.2 The Effectiveness of CBD and ORB.

In this section, the effectiveness of Cyclic Blockwise Distribution (CBD) and Orthogonal Recursive Bisection (ORB) will be discussed for the workload functions that are typical for coastal water simulation. The discussion consists of two parts. First, in Section 4.2.1 a study will be presented in which CBD and ORB are applied to artificial workload functions of the kinds that occur in coastal water simulation. Secondly, Section 4.2.2 will show the effectiveness of CBD and ORB for workload functions that are obtained from an actual coastal water simulation.

It will be shown that the CBD method produces on average a well balanced partitioning if the cycle frequency \( F \) is chosen large enough (see Section 3.2.1 for the meaning of \( F \)). However, if \( F \) is not taken sufficiently large, then CBD leads to significant load imbalances. Thus, the method works only when \( F \) is not too small.

The ORB method by construction always gives a balanced partitioning, but it may lead to a high connectivity between the parts and consequently to a high communication overhead during parallel processing. The results that are presented in this section indicate that this effect is not too serious. Although the parts can become very elongated in some cases, this usually does not cause an unacceptably high connectivity. ORB will be seen to be a better method than CBD in the context of coastal water simulation.

4.2.1 A Study using Simulation.

To assess the average effectiveness of CBD and ORB, the methods have been applied to a large number of generated workload functions of the sorts that are typical for coastal water simulation. The details of these workload functions and the way they are generated, have been described in Section 4.1. The description of the workload functions that is given there contains two parameters: one that determines the roughness of the coastal topography (i.e., \( H \), see Section 4.1.1) and one that determines the water level in the coastal area (i.e., SeaLevel, see Section 4.1.3). In the following discussion, SeaLevel will be denoted by \( SL \) and it will be quantified as the number of points in the \( N \times N \) domain that are below sea-level. Different settings of \( H \) and \( SL \) will be considered to obtain an indication of their effect on the effectiveness of CBD and ORB.

All results that will be shown are averages over 100 generated workload functions of \( 128 \times 128 \) points. This size is large enough to get realistic results and at the same time not too large to prohibit a large number of experiments.
Chapter 4. Partitioning Shallow Water Computations for Coastal Areas

I) Simulation Results for (Cyclic) Blockwise Distribution

Quantification of the Effectiveness of CBD

CBD is known as a method that is potentially capable of producing a balanced partitioning when the cycle frequency $F$ is chosen large enough. However, increasing the cycle frequency generally also increases the communication volume, so that the cycle frequency can not be set arbitrarily high. Therefore, the effectiveness of CBD can be expressed in terms of the reducing effect of the cycle frequency on the load imbalance: if this effect is strong, then CBD can be qualified as an effective method.

The effect of the cycle frequency will be quantified by the relative decrease in load imbalance: $\frac{B(F) - B(2F)}{B(F)}$. This is plotted in Figure 4.3 for different values of $F$. The graphs show the load imbalance $B(F)$ versus $F$ for different values of $P$, scaled to minimise the effect of $F$. The graphs also show the relative decrease in load imbalance for different values of $H$. The graphs further show the effect of the sea level on the relative decrease in load imbalance.

Figure 4.3: The effectiveness of CBD for workload function $W_1(.)$. Top Left: the load imbalance $B(F)$ versus $F$ for different values of $P$. Top Right: the load imbalance $B(F)$ versus $P$ for different values of $F$, scaled to minimise the effect of $F$. Bottom Left: the relative decrease in load imbalance $\frac{B(F) - B(2F)}{B(F)}$ versus $F$ for different values of $H$. Bottom Right: the effect of the sea level on the relative decrease in load imbalance.

- $F = 1$, scale $= 1:1.56$
- $F = 2$, scale $= 1:0.61$
- $F = 4$, scale $= 1:0.25$
- $F = 8$, scale $= 1:0.12$
4.2. The Effectiveness of CBD and ORB.

imbalance that is obtained by doubling the cycle frequency. This relative decrease will be denoted by $dB$:

$$dB \equiv \frac{B(F) - B(2F)}{B(F)}$$  \hspace{1cm} (4.10)

where $B(F)$ is the load imbalance in a CBD partitioning with cycle frequency $F$:

$$B(F) = \max_{p \in [1, P]} \left( \frac{L_p(F) - \bar{L}}{\bar{L}} \right)$$

A value of $dB$ close to 1 indicates that the load imbalance decreases very fast with $F$ and thus indicates a high effectiveness of CBD. If $dB$ is much smaller than 1 (perhaps even negative), then this indicates a low effectiveness of CBD.

The reason for considering this particular form of relative decrease is motivated by the fact that, from Chapter 3, it is expected that a workload function with a Fourier transform decreasing like $1/\omega^a$ (as is the case for the workload functions that will be considered here) should give $B(F) \propto 1/F^a$. Then:

$$\frac{B(F) - B(2F)}{B(F)} = \left( \frac{c}{F^a} - \frac{c}{(2F)^a} \right) \cdot \frac{F^a}{c} = \left( 1 - \frac{1}{2^a} \right)$$

which depends only on $a$. So, the relative decrease $dB$ should be a constant for a particular value of $a$.

Results of the Simulation Experiments

The results of the simulation study are shown in Figure 4.3 (for workload function $W_1(.)$ as defined in Section 4.1.3) and Figure 4.8 (for workload function $W_2(.)$, as defined in Section 4.1.3). In all cases, Cyclic Blockwise Distribution has been applied only along one dimension so that the theory from Chapter 3 can be used to interpret the results.

The top two graphs in Figure 4.3 show the influence of the cycle frequency, $F$, and the number of processors, $P$ on the load imbalance for workload function $W_1(.)$. From the top left graph it can be seen, first of all, that the load imbalance is significant when $F = 1$. For large numbers of processors it is over 100%, which effectively means that the processor that gets most work to do, has a workload that is twice as large as the average workload. But increasing the cycle frequency has a very strong balancing effect. In fact, $B(F)$ behaves approximately like $1/F$ when $P$ is large enough. This is confirmed by the lower left graph, which shows that $dB(F)$ is indeed a constant, as would be expected for a $1/F$ behaviour of $B(F)$ (see above). What is not shown in the graph, but is certainly worth noting, is that the standard deviation of $B(F)$ is high for small $F$ (about 50%) but decreases as $F$ is increased (to about 10% for $F = 32$). So, not only does $F$ decrease the load imbalance, but it also decreases the variation in load imbalances across different workload functions.

The number of processors, $P$, has no significant impact: the top right graph shows that once the number of processors is higher than about 16, a further increase in $P$ hardly makes the load imbalance any worse. Thus, to summarise the results: the load imbalance,
averaged over many workload functions $W_i(.)$ is large for small $F$, is strongly attenuated by an increase in $F$ and does hardly depend on $P$ except when $P$ is small.

The bottom two graphs in Figure 4.3 show the effect of the two parameters of the workload function, i.e. the roughness of the sea-floor, $H$, and the water level, $SL$. Neither seem to have any impact on the effect of $F$. The theory in Chapter 3 suggests that the effect of $F$ is determined by the form of the Fourier transform of the workload function. So, if $H$ and $SL$ have no impact on this effect, then they should also have no impact on the form of the Fourier transform. This is indeed the case, as is illustrated in Figure 4.4: for any value of $H$ the Fourier transform is approximately given by $1/\omega$ when $\omega$ is not too small.

It may seem surprising at first glance, that the parameter $H$ does not have any influence on the Fourier transform of the workload function: it does, after all, determine the Fourier transform of the fractal surface from which the workload function is constructed.

The reason for this is not that a one-dimensional CBD is performed on a two-dimensional surface, as might be suspected. Performing a one-dimensional CBD on a two-dimensional workload function means that the workload function that is actually distributed by the CBD is in fact the sum of $W_i(.)$ over all points of the domain along the direction that is not distributed. But this summed workload function has the same type of Fourier transform as $W_i(.)$ itself, which can be seen as follows. Suppose the workload function $W_i(.)$ has been generated from a Fourier representation of the form given in Equation (4.9). Then

$$W_i(i_1, i_2) = \sum_{\omega_1=0}^{N-1} \sum_{\omega_2=0}^{N-1} \mathcal{F}(\omega_1, \omega_2) \exp(2\pi i (\omega_1 i_1 + \omega_2 i_2) / N)$$
4.2. The Effectiveness of CBD and ORB.

Figure 4.6: The relative decrease in load imbalance $dB(F)$ versus $F$ for different values of $H$ using the full fractal surface as the workload function.

Figure 4.7: The changes in Fourier transform when constructing a workload function $W_1(.)$ from a fractal surface with $H = 0.9$.

so that the sum of $W_1(.)$ in the $y$-direction is

$$\sum_{y=0}^{N-1} W(i_1, i_2) = \sum_{i_2=0}^{N-1} \sum_{\omega_1=0}^{N-1} \sum_{\omega_2=0}^{N-1} F(\omega_1, \omega_2) \exp\left(2\pi i (\omega_1 i_1 + \omega_2 i_2)/N\right)$$

Thus: the Fourier transform of the summation of $W_1(.)$ in the $y$-direction is $F(\omega_1, 0) = N/\omega_1^{H+1}$, which is similar in shape to the original form of the Fourier transform of the workload function. This is visible in Figure 4.5, which shows the modifications to the Fourier transform that are caused by the different steps in the construction of $W_1(.)$ from the fractal surface. The line that represents the summation of $W_1(.)$ in the $y$-direction is the same as the line $1/\omega_1^{H+1}$.

The fact that the Fourier transform is independent from $H$ and $SL$ is instead caused by the fact that only $1/16$th of the fractal surface is actually used for the construction of the workload function. This is done to attenuate the periodicity in the fractal surface that results from the way in which it is generated (see Section 4.1.2). Figure 4.5 shows the impact of this on the form of the Fourier transform: the rate of decay is reduced from $H + 1$ to nearly 1.

The last step in the construction of a workload function of type $W_1(.)$ from a fractal surface, i.e. setting the workload to zero in all points where the fractal surface is above sea-level, has hardly any consequences. It does have some effect, in particular when the SeaLevel is set low, but this effect is unnoticeable after the surface has been cut back to $1/16$th of it.

Figure 4.6 shows that, if the complete fractal surface itself is used as a workload function
Figure 4.8: The effectiveness of CBD for workload function $W_2(.)$. Left: the relative decrease in load imbalance $dB(F)$ versus $F$ for different values of $H$. Right: the effect of the sea level on the relative decrease in load imbalance, $dB(F)$.

(on the full $N \times N$ domain and using the height directly as a measure of workload), then the expected effect of $H$ is clearly observed. This confirms that Theorem 11 indeed holds.

The results for workload function $W_2(.)$ are not essentially different from those for workload function $W_1(.)$ (see Figure 4.8). Again, there is roughly a $1/F$ behaviour of the load imbalance and hardly any effect from the number of processors $P$. The relative decrease in load imbalance, $dB$, is constant at about $1/2$ for a wide range of $F$. However, for $W_2(.)$ the independence of $dB$ from $H$ is less of an artifact than in case of workload function $W_1(.)$. In $W_2(.)$, the properties of the the Fourier transform of the fractal surface are hidden largely by the fact that values below sea-level are all set to one and those above sea-level are all set to zero. Figure 4.7 shows that the Fourier transform of the workload function is affected strongly both by the fact that only $1/16$th of the surface is retained and by the fact that the workload values are either one or zero. The net effect of these two aspects of the construction of workload $W_2(.)$ is to make the form of the Fourier transform hardly dependent on $H$.

The simulations show that CBD is indeed effective at balancing the workloads across the available processors. But if $F$ is chosen small, then the load imbalances can still be quite significant. Also, the variation in the balancing effect across different workload functions of either $W_1(.)$ or $W_2(.)$ are large, again in particular when $F$ is small. This means that CBD becomes a reliable method only when $F$ is taken large enough. But this in turn leads to a relatively large communication volume and is therefore not always feasible. In the following discussion, simulation results for the ORB method will be presented that show that ORB is a much more robust method than CBD.
4.2. The Effectiveness of CBD and ORB.

Figure 4.9: The effectiveness of ORB partitioning for workload function $W_2(\cdot)$. Top: the Skewness versus the number of processors for different sea-levels and different values of $H$. Bottom Left: Dilation versus the number of processors for different sea-levels. Bottom Right: average number of connections per processor for different SeaLevels.

II) Simulation Results for Recursive Coordinate Bisection

Quantification of the Effectiveness of ORB

ORB gives a balanced workload by definition. Therefore, the load balance is not an issue for this method. Instead, the method’s effectiveness is determined by the communication that it induces. In some cases, ORB may create very elongated parts that are bordered by a relatively large number of neighbouring parts. This may lead to both a large communication volume and to congestion in the communication network so that the communication overhead during the parallel processing will be significant.

Because the time that is spent in communication is determined by more than just the communication volume, there is not one single quantity that can express the communication
overhead that is induced by an ORB partitioning. Therefore, three different quantifiers will be considered, that together give an impression of the effectiveness of ORB:

- **Skewness**, i.e. the maximum size of a side of a part divided by the average size, which expresses the maximum communication volume in any direction.

- **Dilation**, i.e. the maximum number of parts bordering the same neighbour in a particular direction. This expresses the distance that messages may have to travel in networks that do not provide full connectivity (see Chapter 2),

- **Connectivity**, i.e. the total number of connections between the parts, where a connection exists between two parts if one of them has a point next to a point in the other. This expresses the number of messages that have to be sent by a processor. Note that the Connectivity divided by the number of processors gives the average number of neighbours per processor.

ORB will be said to be an effective partitioning method if the Skewness and Dilation remain close to one, and if the Connectivity stays close to four (i.e. the connectivity under regular blockwise partitioning). The relative importance of the three quantifiers depends on the kind of network that is used for the communication. On networks where processors are connected only sparingly, the Connectivity and Dilation will be very important, whereas the Skewness is particularly important on networks in which the processors are connected by low-speed links.

**Results of the Simulation Experiments**

The results of the simulation experiments are shown in Figure 4.9 (for workload function $W_1(.)$) and Figure 4.11 (for workload function $W_2(.)$).

First, consider the results for workload function $W_1(.)$ in Figure 4.9. From the top-left graph, it can be seen that the Skewness increases rapidly with increasing number of processors. On 256 processors with 25% of the gridpoints below sea-level, there are subdomains that are up to 13 times longer than they would have been under regular blockwise distribution. But this high Skewness does not lead to a very high Dilation (see bottom-left graph in Figure 4.9). It turns out that elongated parts are usually bordered by other elongated parts, so that the number of neighbours of a part in any direction remains relatively small.

This is also visible in the average total number of connections per part, i.e. the Connectivity, which is never far from the connectivity of a regular blockwise partitioning (see bottom-right graph in Figure 4.9). Altogether, the communication pattern that arises from an ORB partitioning is not too much different from that of a regular blockwise partitioning and should therefore be efficiently supported by most interconnection networks.

But the Skewness can be large, in which case there are parts with a relatively long perimeter. Such parts will need to transfer a large volume of data when communicating values on the perimeter to other processors. This can limit the effectiveness of ORB partitioning.
4.2. The Effectiveness of CBD and ORB.

It should be noted though, that the value of the Skewness is strongly dependent on the parameters of the workload function, i.e. on the sea-level SL and on the sea-floor roughness \( H \). The Skewness remains small if the sea-level is high. Thus, ORB is much more effective for workload functions with a high sea-level than for those with a low sea-level.

Partly, this can be understood from Theorem 14. If the sea-level is high, then the average workload in each part is high and the amount of depth variation is relatively small. In terms of Theorem 14: a high average workload gives a high value for \( aN \) and a small amount of variation gives a small value of \( V \). So, for a high sea-level, \( V/aN \) is small, and because the Skewness is bounded from above by \( 1 + V/aN \) (ignoring the influence of the peakedness of the variation as expressed by \( m \) in Theorem 14), the Skewness itself must be close to one.

If, on the other hand, the sea-level is low, then \( V/aN \) is high and the upper bound on the Skewness increases so that the Skewness can become higher. The results of the simulations show that the Skewness does indeed become larger when the sea-level decreases.

The same effect is clearly visible in Figure 4.10, which shows the parts of the domain where the Skewness is highest. In places near the coastline, where the depth is relatively low, there are several very elongated parts, i.e. parts that have a high Skewness. Far from the coastline, in the lower-left section of the graphs, the depth is high and the partitioning becomes much like a regular block partitioning.

The other workload function parameter, i.e. the sea-floor roughness \( H \), also has considerable impact on the Skewness (see top-left graph in Figure 4.9). When \( H \) is small, the Skewness is significantly smaller than when \( H \) is large. This can be understood from the fact that for small \( H \) the variations in depth are local (see Figure 4.2 for an example) and the variations average out over each part. When \( H \) is high, on the other hand, there are parts with shallow depth which must have a large horizontal size to represent the same total workload as parts where the water is very deep.

For workload function \( W_2(.) \), the skewness is again strongly dependent on the SeaLevel

Figure 4.10: Left: the ORB partitioning of a generated workload function of type \( W_1(.) \) with \( H = 0.8, SL = 50\% \). Right: the number of connections of each part.
Chapter 4. Partitioning Shallow Water Computations for Coastal Areas

Figure 4.11: The effectiveness of ORB for workload function $W_2(.)$. Left: the Skewness versus the number of processors for different SeaLevels. Right: the Dilation versus the number of processors for different SeaLevels.

(see Figure 4.11). Theorem 14 offers an elegant explanation of this.

For ease of presentation, consider a one-dimensional workload function $W_2(.)$ with domain $[1, N]$. For such a workload function it is possible to explicitly determine the value of $V/aN$ in Theorem 14. Suppose there are $k$ points in the domain where the $W_2(.) = 1$. Then there are $N - k$ points where the workload is zero. So:

$$aN = (k \cdot 1 + (N - k) \cdot 0) = k$$

Also,

$$V = k \cdot (1 - k/N) + (N - k) \cdot k/N = 2k(N - k)/N$$

Therefore, $V/aN = 2(1 - k/N)$ which is linear in $k$, i.e. the number of points in the domain where the workload is non-zero. Now, a rise in sea-level increases $k$ and therefore decreases $V/aN$. According to Theorem 14 this reduces the upper bound on the Skewness, so that the Skewness can take only smaller values. Eventually, when the sea-level is so high that $W_2(.)$ is nonzero in all points, then the upper bound on the Skewness becomes one, and since the Skewness can not be smaller than one, it must always equal one. This is indeed observed (see the curve for SeaLevel at 100%).

With regard to the effect of the workload function parameter $H$, the same argument as for workload function $W_1(.)$ still holds. When $H$ is small the Skewness will be small and if $H$ is large, then so is the Skewness.

To summarise the results, it can be stated that ORB seems to be a very effective partitioning method for both workload functions $W_1(.)$ and $W_2(.)$ if the sea-level is high, if $H$ is small and if the communication network efficiently supports a communication pattern that is similar to that of a regular block partitioning. If the sea-level is low and $H$ is large,
4.2. The Effectiveness of CBD and ORB.

then a high Skewness may occur so that some parts will have to communicate a relatively large volume of data.

4.2.2 Results of CBD and ORB for Real Workload Functions

To validate the results from the simulation study that was discussed above, the effectiveness of Cyclic Blockwise Distribution (CBD) and Orthogonal Recursive Bisection (ORB) has been studied for workload functions that are based on two real topographies.

The topographies have been taken from two coastal water models that are in daily use at the Dutch Institute for Coastal and Marine Management (RIKZ). The first is the so-called Continental Shelf Model (CSM), which covers a region of the North-West European continental shelf between 48°N, 12°W and 62°N, 12°E. The topographic height is available in gridpoints that are 8 km apart. Only heights between 0 and -200 m are taken into account, which means that only 19,796 gridpoints of the 201 x 173 are used. These points are shown in Figure 4.12.

The second topography has been taken from the Waddensea model, which represents an area along the north-coast of the Netherlands, approximately between Petten in the South-West and the German Emden in the North-East. This topography is represented on a grid with a meshsize of 500 m. All heights below 0 meter are used, which gives 44,845 gridpoints out of the total grid of 400 x 270 points. In all results that are presented in this section, the sea-level has been set at 0 m.

Figure 4.13 shows the relative decrease in load imbalance $dB$ for workload function $W_1(.)$ on the CSM topography.

\begin{figure}[h]
\centering
\includegraphics[width=0.4\textwidth]{continental_shelf_grid}
\caption{The Continental Shelf Grid with meshsize of 8 km (CSM).}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=0.4\textwidth]{load_imbalance}
\caption{The relative decrease in load imbalance $dB$ for workload function $W_1(.)$ on the CSM topography.}
\end{figure}

Figure 4.13 shows the relative decrease in load imbalance, $dB(F)$, when CBD is applied in one direction to a workload function of type $W_1(.)$ based on the CSM topography (for a definition of $dB(F)$, see Section 4.2). Increasing the cycle frequency does indeed have a significant balancing effect: the load imbalance is reduced by roughly 50% each time the
cycle frequency is doubled. But the fluctuations in $dB(F)$ are significant.

This behaviour is similar to the behaviour that was observed in the simulation experiments (see Figure 4.3 and the discussion of that figure in Section 4.2). There, it was noted that $dB(F)$ has an average of 0.5 when applied to many different workload functions of type $W_1(.)$, but the variation of $dB(F)$ across different workload functions was found to be large. The line of $dB(F)$ in Figure 4.13 shows that fluctuations also occur between different values of $F$ for the same workload function.

The effect of CBD for workload functions that are based on the Waddensea topography are shown in Figure 4.14. Again, CBD is applied only in one direction. For this topography, CBD seems to be a somewhat more reliable method in case of workload function $W_1(.)$: the load imbalance decreases steadily with $F$. But for workload function $W_2(.)$, the imbalance reduction strongly fluctuates with $F$, just like for $W_1(.)$ on the CSM topography.

These results confirm that CBD is not a very reliable method: the reduction in load imbalance may strongly fluctuate from one topography to another and from one value of $F$ to another. Nevertheless, the balancing effect that is usually attributed to Cyclic Blockwise Distribution is clearly observed.

The effect of ORB partitioning for workload functions that are based on the two real topographies is shown in Figure 4.15. With respect to Skewness, the Waddensea topography gives significantly worse results than the CSM topography. This is consistent with the results of the simulations. The Waddensea grid represents a very shallow area and the simulation study indicated that a small average depth (with respect to the size of the depth variations) leads to a high Skewness.

For both workload functions, the dependence of the Skewness on the number of processors is similar to what was found in the simulation studies. This indicates that those studies have provided realistic results. The same remark can be made for the Dilation.
4.3 Run-time effects

Just like the results from the simulations suggested, the Dilation does not become very large, even though the Skewness is high for large numbers of processors.

Figure 4.15: The Skewness (left) and Dilation (right) for ORB partitioning of workload functions $W_1(.)$ and $W_2(.)$ on the CSM topography and on the Waddensea topography.

4.3 Run-time effects

Until now, the discussion has been theoretical in nature and little attention has been paid to the actual performance that is observed for different partitioning methods on parallel computers. There are two reasons for this emphasis on theoretical aspects. Firstly, parallel computers differ widely in the parameters that determine performance and thus practical performance results on one or a few parallel computers are of little use by themselves. Secondly, parallel computers develop quickly and any performance behaviour that is observed today will be outdated in a few years at best.

Nevertheless, all the work on partitioning methods that is presented in this thesis serves to obtain a high performance in practical situations. Therefore, it seems appropriate to devote some discussion to the more practical aspects of partitioning. Hence, in this section, the parallel performance will be discussed that has been observed for Blockwise and ORB partitioning of the CSM grid (see Section 4.2.2) on several different parallel computers. Along the way, the aspects that determine the performance will be discussed. This will illustrate and complement much of the discussion in the previous chapters and the previous sections of this Chapter.

Among other things, it will be shown that the load imbalances that occur in blockwise partitioning do not necessarily lead to serious performance degradation with respect to partitioning methods that give a more balanced partitioning. Also, the higher inter-part connectivity of ORB can be a problem when the connectivity of the communication network is low. Therefore, the relative benefits of ORB with respect to CBD, suggested by
the simulation studies that were discussed in the previous two sections, is not clear from their relative merits on present-day parallel architectures.

The performance results for simulation of flow and transport processes on the CSM grid have been obtained with the parallel version of a program called TRIWAQ. The sequential version of this program has been developed by the Dutch Institute for Coastal and Marine Management (RIKZ) and has long been in use to do simulations for predictive purposes and for decision support. It is based on the three-dimensional shallow water equations. An extensive discussion of it has been published elsewhere (see Lander [119] or the brief introduction in Vollebregt [204]).

To allow parallel processing with TRIWAQ, a provision has been implemented in the program that allows it to exchange information with other programs at run-time. With these provisions, several instances of the TRIWAQ program can be coupled, which effectively gives a parallel version of the program.

The structure of the parallel version of TRIWAQ is sketched in Figure 5.4. Before the simulation is started, the input file containing the specification of the simulation is divided into several files, each containing the specification of the simulation on a part of the grid. Then several instances of the extended version of TRIWAQ are started, each dedicated to performing the simulation for one of the parts of the grid. Through the exchange mechanism, the TRIWAQ processes can communicate intermediary results in order to couple the simulations on the different parts. The results of the parallel simulation are collected into a single output file after the simulation has ended. More details on parallel TRIWAQ can be found in the next chapter and in a number of publications (Lin et al. [127], Roest and Vollebregt [168], Roest et al. [170, 172], Vollebregt [203, 205] and Vollebregt et al. [206]).

The parallel version of TRIWAQ has been run on several different architectures. Ex-
4.3. Run-time effects

tensive studies of its performance for the CSM model have been done for a cluster of
workstations and for a Parsytec PowerXplorer.

The cluster of workstations that has been used, consists of eight HP9000/735 worksta­
tions , each with a 125 MHz PA-RISC 7100 processor. The workstations are connected
by Ethernet (10 Mb/s). During the experiments, there was no interference from other
users. The Parsytec PowerXplorer has 32 processors, of which only 16 could be used.
Each processor is an 80 MHz PowerPC 601 processor with 32Kbyte cache. The processors
are connected to a T800 transputer for communication over four links that each sustain a
communication rate of 8.8 Mbyte/s.

Most of the results that are presented here use a version of the CSM model in which
the vertical (i.e. depth-) structure is resolved by five layers, whose thickness varies with the
local depth. Incidentally, results will be presented for a CSM model with a single layer but
with the same horizontal grid as the one with five layers.

Five different partitioning methods have been used. One of these is a regular (i.e. non-
cyclic) blockwise partitioning, which will further be denoted by BLOCK. Cyclic Blockwise
Distribution could not be studied with the current implementation of parallel TRIWAQ.
That is because each TRIWAQ process can only handle a single part and thus each block
that is mapped onto a processor must be handled by a separate TRIWAQ process. Therefore,
process-switching overhead is bound to obscure any performance results regarding the
use of Cyclic Blockwise Distribution.

The other four partitioning methods that have been used are all versions of Recursive
Coordinate Bisection (ORB):

Figure 4.17: Examples of ORB partitioning of the CSM model. Left: STRIP partitioning.
Right: SORB partitioning.
Figure 4.18: The expected and measured communication times for the CSM model with different partitioning methods assuming full bandwidth available to all processors and no delays due to load imbalances (left), the same, but now including synchronisation delays (middle) and assuming bandwidth is shared by processors that communicate simultaneously (right).

ORB splits the grid into subgrids with equal numbers of gridpoints, allowing a difference of at most one gridpoint in each recursive splitting. The interfaces between subgrids will usually not be straight lines, but will contain a step, which is caused by the fact that a row of gridpoints must be divided over two parts in order to obtain the required equal number of gridpoints. Interfaces that are created in subsequent recursions are orthogonal.

SORB splits the grid such that the interfaces between subgrids are straight lines (i.e. without a step). In general, this second version of ORB will not give exactly equal amounts of gridpoints in each subgrid, but the differences are usually not too serious. Again, interfaces that are created in subsequent recursions are orthogonal.

STRIP splits in one direction only, that is: the splitting lines are all parallel. In this version of ORB, the parts are effectively strips of the grid, hence the name. The interfaces in the STRIP method may contain a step of one gridpoint wide. Note that quantities such as Skewness, Dilation and Connectivity are not meaningful for this version of ORB. The Skewness and Dilation have no impact on the communication volume or on the number of communications. The connectivity is always constant: each part has exactly two neighbours (except for those on the edge of the domain).

SSTRIP the same as STRIP, but now the interfaces between the parts are straight lines, which usually requires a slight imbalance in the number of gridpoints per part. Again, Skewness, Dilation and Connectivity are not meaningful for this version of ORB.

An example of SORB and STRIP partitioning of the CSM model is shown in Figure 4.17. The speedup curves on the different machines are shown in Figure 4.16. These speedup curves do not include file I/O nor the time needed for initialisation and termination. The reason for excluding I/O time is that this time is highly dependent on the particular I/O requirements of a run. The initialisation and termination times have been excluded because
they are relatively unimportant for long runs.

First, consider the results on the cluster of workstations (left in Figure 4.16). The speedup is almost linear in the number of processors when up to four workstations are applied. But when more than four processors are used, there is hardly any further increase in speedup.

This is largely due to contention in the communication network. Figure 4.18, left, shows the time that is expected to be spent in communication, based on latency, transfer speed and number and size of the messages. The expected times are far less than what is actually observed and the deviation grows fast with the number of workstations. Also, delays due to synchronisation can not account for the long communication times (see middle plot of Figure 4.18).

Instead, the best explanation for the long communication times is that the Ethernet acts as a bus, whose bandwidth must be shared by all processors. In the leftmost plot of Figure 4.18, the communication times are given that are to be expected when assuming that all processors that communicate simultaneously must share the available bandwidth. These expected times at least follow the observed trend in communication time. The fact that the observed times still differ significantly from the predicted times is due to the fact that it is difficult to properly model the contention in a program with a complex communication and computation pattern such as TRIWAQ.

From Figure 4.16, it can be seen that on a cluster of workstations the straight interfaces in the SSTRIP and SORB methods give a noticeable performance improvement with respect to the STRIP STRIP and ORB methods. This benefit is due to the nature of one of the algorithms that are used in TRIWAQ, the SUW algorithm. It consists of two parts for each time-step. In the first part the computation of new values in each gridpoint requires the old values in neighboring points in the x-direction. When the interface between two parts is along the y-direction, then this algorithm requires an exchange of values across this interface. The second part of the SUW-algorithm likewise causes an exchange of values across boundaries along the x-direction.

Now, in case of a partitioning in which the boundaries of the parts contain a step, each interface has a section in the x-direction as well as in the y-direction, one of which is only a single gridpoint long. Therefore communication must occur in both parts of the algorithm. When the interfaces are straight lines, each interface is either along the x- or y-direction, but never in both directions. Hence, communication across a particular interface happens only in one of the two parts of the algorithm. Thus the number of communications is reduced by straightening the interfaces and this gives the notable performance benefit. The reduction in the number of messages that is obtained from straightening the interfaces can be seen in Figure 4.19.

That figure also shows another interesting feature. The size of the interface under the BLOCK, ORB and SORB methods should approximately be inversely proportional to the square root of the number of processors (see Chapter 2). The fact that total message volume increases also for the (S)ORB and BLOCK methods is due to the fact that in the range from one through eight workstations, the average number of neighbours of a part is steadily increasing. And with the number of neighbours, the number and total size of the interfaces also grow.
Chapter 4. Partitioning Shallow Water Computations for Coastal Areas

Figure 4.19: The maximum total message volume over all processors (left) and the maximum total number of messages over all processors (right) for the CSM model with 5 layers.

For BLOCK partitioning, the average number of neighbours of a block on two workstations is one, on four workstations it is two and on eight it is 2.5 (not counting neighbours in diagonal direction). So, even though the size of each interface decreases when the number of processors increases, the growing number of neighbours of each block still leads to a growing total interface size. This illustrates that the results on interface sizes for regular block methods that were discussed in Chapter 2 only hold for large numbers of processors.

Note further from Figure 4.19 that the total message volume for the (S)STRIP methods is significantly higher than for the BLOCK and (S)ORB methods. But nevertheless, the (S)STRIP methods give the highest speedup on the cluster of workstations when less than six workstations are used. In that case, the communication volume is not a problem. Only on eight workstations does the total message volume start to determine the performance and there the (S)STRIP methods perform worse than the (S)ORB methods (see Figure 4.16).

The BLOCK method performs surprisingly well, given the fact that it suffers from significant load imbalances. On eight workstations, it is even the best method of all those that were tested. There, it gives a significantly lower communication volume than the (S)STRIP methods and the load imbalances probably cause the communication to be spread in time, so that less contention occurs in the network. Thus, the load imbalances are even somewhat advantageous in this case.

The speedup curves for the Parsytec PowerXplorer (Figure 4.16, right) show a pleasantly surprising feature: on 16 processors the speedup for the (S)STRIP methods is well over 17. This superlinear speedup is due to a strong cache-effect. When the parts that are computed per processor become smaller, the data for those parts fits better in cache and consequently the computations can be done faster.

That such a cache effect indeed occurs, is illustrated in Figure 4.20, which shows a break-
4.3. Run-time effects

Figure 4.20: Left: The total time spent by all processors in communication and computation on the Parsytec PowerXplorer for a SSTRIP partitioning of the single layer CSM model. Right: the total computing time for the single layer CSM model for different partitioning methods.

down of the execution time for a the CSM model (with a single layer) into communication and computation. It can be seen that the total time that all processors together spend in computations becomes smaller whereas is should slightly increase due to the fact the the convergence of the iterative methods slightly deteriorates. Note though that this effect would disappear if the computations for the sequential program would be reorganised to improve cache-usage. In that case, no superlinear speedup would be observed.

The cache-effect also complicates the discussion of the speedup curves on the PowerXplorer shown in Figure 4.16. Apart from the total message volume and number of messages also the computing speed differs from one partitioning method to another (see Figure 4.20). Furthermore, the communication network is not as simple as in the case of a cluster of workstations. Altogether, only a very detailed performance model can precisely reveal the reasons for the observed speedup curves. Any discussion of such a model is beyond the scope of this section.

It does show however, why the partitioning problem has been formulated in the way it was done in Chapter 2: apart from the requirement for a minimum partitioning time, all aspects of the problem formulation play a significant role in determining the performance of parallel TRIWAQ on the PowerXplorer.

This last section has discussed some practical aspects of applying partitioning methods for the parallelisation of coastal water simulation. Based on the theoretical studies in Section 4.2, ORB would seem to be the best method for this application. But this conclusion only holds when the network is capable to handle the somewhat higher connectivity of ORB partitioning.

Furthermore, the number of processors must be high enough. When only a few processors are used, the average number of neighbours increases when more processors are
added, so that the average total perimeter size increases.

In the experiments that were described in this last section, these conditions do not hold and hence the advantages of ORB partitioning have not become apparent. This illustrates the notion, which was expressed at the start of this chapter, that looking at present-day performance results does not give a complete impression of the relative merits of partitioning methods.
Chapter 5

Data partitioning and Data structures

Until now, this thesis has concentrated on the effectiveness of partitioning methods. The discussion in the last three chapters has provided an improved theoretical and practical understanding of two well-known partitioning methods. This will make it easier to effectively apply these methods in the future. But the selection of a partitioning method is just one of the problems that a programmer encounters when he or she has to specify how data and computations must be distributed over processors in a parallel computer.

A second issue is that each data element should preferably be stored on the processor on which it is needed most. Thus, once the the distribution of the computations has been specified, the distribution of the data should be done accordingly. But it is not always easy to see which processor needs a data element most. Some data elements are used by many different computations so that the relative importance of all those computations will have to be considered. Also, the use of indirect addressing schemes can make it difficult to determine which data element is used by a certain computation.

The problem becomes much easier if it is considered on a higher level. This holds in particular for many scientific computing codes that constitute large computing problems when the domain of computation is large. An obvious way of parallelising such codes is by partitioning the domain of computation into subdomains and computing each subdomain on a separate processor. Then the problem of distributing data can be solved by giving each processor all the data that is associated with the subdomain that is computed there.

Although this approach is quite straightforward, it will always lead to a high performance on parallel computers provided that the subdomains can be solved largely independently from each other. But the actual implementation of the data distribution can still cause problems. The input data for a computing problem is usually supplied through input files and command-line parameters. The data that is supplied, applies to the computational domain as a whole and it may not be a trivial task to extract the data for each of the subdomains from it. Fortunately, though, much of this can be done automatically.

In this chapter, a tool will be discussed that can automatically partition the domain of a computation and reorganise the input data into input data for each of the subdomains. Basically, the tool first reads the data into the data structures that are used in the computing code, then reorganises it into similar data structures for each of the subdomains and finally writes the subdomain data back to file.

The reorganising of the data in the data structures is steered by a high level specification
of the data structures that are used in the computing code. The design of this high-level description is based on a study of data structures that occur in scientific computing codes. The results of this study will be presented.

Apart from showing how input data for a problem can be reorganised into input data for its subproblems, the discussion in this chapter will also provide valuable insight in the nature of indirect addressing schemes. This understanding will be useful for designing support for such schemes in data parallel languages. It will become clear, among other things, that different forms of indirect addressing exist, each posing particular problems for distribution and requiring particular solutions.

Section 5.1 will sketch the basic approach of the subsequent theoretical discussion and will introduce the most important concepts. In particular, the concept of abstraction functions will be briefly discussed for those that are not familiar with it. Then, Section 5.2 will discuss how data structures emerge in the design of a scientific computing code. This will lead to a formal description of these data structures, which is then used in 5.3 to determine how the data for each subdomain can be extracted from data structures that apply for the complete domain. Section 5.4, finally, gives an outline of the PARPRE/PARPOS tool, which has been constructed on the basis of the theoretical considerations of Sections 5.2 and 5.3.

Most of the work that will be presented in this chapter has been published in parts before (see [33, 34, 169, 170, 173]). The most complete account of the work so far has been published by Ten Cate, Roest and Vollebregt [35].

5.1 Introduction

This chapter will give a formal description of data structures which are used in computing codes that are characterised by a large computational domain. Once such a formal description has been given, it can be used to show, again formally, how the data in those data structures can be reorganised into similar data structures for each of the subdomains that are defined by a partitioning of the computational domain.

This section will first discuss the type of computing codes that will be considered in this chapter. Then, an outline of the discussion in this chapter will be given to introduce the basic approach. Finally, some concepts will be introduced that will be used in the discussion in the next sections.

5.1.1 An Abstract Specification of a Class of Computing Codes

The class of computing codes to which the discussion of this chapter applies is that of codes for the numerical solution of a system of partial differential equations on a large domain. This class encompasses the major part of all scientific and technical computing codes. Examples of this type of code are the finite element codes that are used in construction analysis. Some other examples are codes that are used for the simulation of flow processes in oceans and coastal waters and codes that are used for crash-worthiness simulation.

All these codes have a significant practical relevance, and they usually constitute a large computational effort. Therefore, the codes of this type are very likely candidates
5.1. Introduction

for parallelisation. This can be done by decomposing the domain into subdomains and computing each subdomain on a separate processor. As this is also the basis of the the discussion in this chapter, the approach that will be sketched here applies to all codes of the specified class.

The problem of solving a set of partial differential equations (PDE’s) on a given domain can be generally formulated as:

\[
\begin{align*}
\text{solve for } u(\vec{x}, t) \text{ where } t \in [0, T_{\text{end}}]: & \quad \mathcal{D}u(\vec{x}, t) = f(\vec{x}, t) \quad \text{where } \vec{x} \in \Omega \\
\mathcal{B}u(\vec{x}, t) = g(\vec{x}, t) \quad \text{where } \vec{x} \in \partial\Omega \\
u(\vec{x}, 0) = u_0(\vec{x}) \quad \text{where } \vec{x} \in \Omega + \partial\Omega
\end{align*}
\]

where \(\mathcal{D}\) is some, possibly non-linear, partial differential operator, \(\mathcal{B}\) is the boundary condition operator, usually also given in terms of partial derivatives, and \(u_0\) is the initial state. \(\Omega\) is the domain of the PDE, and \(\partial\Omega\) is its boundary. In this chapter, \(\Omega\) will be assumed to be a subset of \(\mathbb{R}^2\). This is mainly done to ease the discussion; all concepts apply equally when \(\Omega \subset \mathbb{R}\) or \(\Omega \subset \mathbb{R}^3\).

The formulation in Equations (5.1)-(5.3) is a very general one. The form of \(\mathcal{D}, \mathcal{B}\) and the expressions for \(f(.)\) and \(g(.)\) differ from one application to another. For the field of coastal water simulation, which is the subject of most of the examples in this chapter, details of the PDE’s are given a.o. by Vollebregt [205].

5.1.2 Outline of the Discussion

The formal description of data structures in this chapter will be derived from the abstract problem specification that was given above. In this abstract specification, the usual mathematical notation has been used to denote the data. In this case, the data entities of the problem are the functions \(u(\vec{x}, t), f(\vec{x}, t)\) and \(g(\vec{x}, t)\) and the domain of computation \(\Omega + \partial\Omega\). Section 5.2 will show how the abstract data entities are typically transformed into instances of data structures in a computer program.

The transformation will be given as a sequence of reformulation steps, that each make the specification slightly less abstract and a bit more amenable for implementation in a programming language. Each reformulation step will be formally specified, so that the entire transformation of abstract data entities into the program’s data structures can also be expressed formally. This then gives the formal description of data structures that is needed. All this is the topic of Section 5.2.

The next step will be to consider how a partitioning of the computational domain leads to a reorganisation of the data in the data structures. For that, the computational domain in the abstract specification will first be expressed as a collection of subdomains. This leads to a formulation of the abstract specification in terms of subdomains. That subdomain-based specification is then transformed into program code through the same sequence of reformulation steps that were applied before to the original specification. The result of this is that the abstract data entities in the subdomain-based specification are represented by subdomain-oriented data structures.

The entire operation of reorganising data structures for the complete domain into data structures for each of the subdomains can then be expressed in terms of three sub-operations
Chapter 5. Data partitioning and Data structures

Figure 5.1: Sketch of the outline of the way in which the reformulation of data structures is obtained

based on the previous discussion. The first sub-operation is to find the abstract data entities that correspond to the data structures in the code for the complete domain. The next sub-operation is to transform those abstract data entities for the complete domain into abstract data entities for the subdomains. The last sub-operation is to transform the data entities on the partitioned domain into partitioned data structures.

This manipulation can be illustrated with Figure 5.1. The operation that leads from the data structures in the code (top right) to the data structures in the code for the partitioned problem (bottom right) is the sequence of transformations that is encountered when going along the top of the figure back to the abstract problem formulation, and then along the bottom of the figure to the code level again.

Now, given the formal expression of the reformulation steps, and also given the formal rewriting of the abstract specification in terms of subdomains, the entire operation of reorganising data structures for the complete domain into similar ones for the subdomains can be expressed formally. This will be shown in Section 5.3. The formal expression of the operation makes it possible to implement it in a computer program. A short discussion of such a program will be given in Section 5.4.

Figure 5.2 shows all the entities that will be used in this chapter at their location in the scheme of Figure 5.1. The figure is given here so that it may help the reader to place the entities that will be encountered in their proper context. The meaning of the entities will become clear in course of the discussion below.

5.1.3 Abstraction Functions

It was mentioned above that each of the reformulation steps that transform an abstract code specification into a program will be expressed formally in this chapter. This will be done through abstraction functions. An abstraction function is the function that gives the relation between an entity on a certain abstraction level and the entity on the next higher abstraction level of which it is the representation.

For example, let \( \Omega \) in the abstract code specification above be a rectangular area \( [x_1, x_1'] \times [x_2, x_2'] \subset \mathbb{R} \times \mathbb{R} \). A common way of making the abstract specification ready for numerical analysis is by discretizing the domain of computation. Thus, \( \Omega \) will
be represented by a set of discrete points \([1, N_1] \times [1, N_2] \subseteq \mathbb{N} \times \mathbb{N}\). The set of discrete points is usually called the *Mesh*. This reformulation of the abstract specification into a discrete version can be represented by an abstraction function that gives for each *Mesh*-point its counterpart in the continuous formulation. Let this abstraction function be called \(\text{Mesh2Domain}(\cdot)\).

A possible form of \(\text{Mesh2Domain}(\cdot)\) is the following:

\[
\text{Mesh2Domain}: \text{Mesh} \to \Omega, \\
\text{Mesh2Domain}(n_1, n_2) = (x_{1,0} + (n_1 - 1) \Delta x_1, x_{2,0} + (n_2 - 1) \Delta x_2) \in \Omega \\
\Delta x_1 = (x_{1,e} - x_{1,o})/(N_1 - 1), \Delta x_2 = (x_{2,e} - x_{2,o})/(N_2 - 1)
\]

where *Mesh* is a closed subset of \(\mathbb{N}^2\). This abstraction function shows the exact meaning of the discrete point \((n_1, n_2)\) as a representation of a point in the continuous domain. Note that the abstraction function is not surjective: not all points in \(\Omega\) are the image of a point in *Mesh*. This is a common characteristic of abstraction functions which bring functions and operations on infinite sets back to the finite proportions that can be handled with a computer.

The discretization of the domain has consequences for the functions that are defined on it. For example, the function \(f(\vec{x}, t)\) in Equation (5.1) will have to be replaced by one that is defined on *Mesh*. In the following, it will be assumed that \(f(\vec{x}, t)\) is not dependent on \(t\), so that the last argument can be dropped. Then \(f(.)\) can be represented by a semi-discrete version \(F(.) : \text{Mesh} \to \mathbb{R}\) according to the abstraction function:

\[
\text{MeshFunction2DomainFunction} : \{\text{Mesh} \to \mathbb{R}\} \to \{\Omega \times \mathbb{R} \to \mathbb{R}\}, \\
\forall t : \text{MeshFunction2DomainFunction}(F(.))(n_1, n_2) = f(\text{Mesh2Domain}(n_1, n_2), t)
\]

where \(\{\text{Mesh} \to \mathbb{R}\}\) denotes the set of all functions with domain *Mesh* and range \(\mathbb{R}\). Likewise, derivatives of \(f(.)\), like \(\partial f(.)/\partial x_1\), will have to be replaced by semi-discrete versions. This can be done by noting that, under certain conditions,

\[
f(x + \Delta x_1, x_2, t) = f(x_1, x_2, t) + \frac{\partial f(x_1, x_2, t)}{\partial x_1} \Delta x_1 + \mathcal{O}(\Delta x^2)
\]

\[
\Rightarrow \frac{\partial f(x_1, x_2, t)}{\partial x_1} \approx f(x_1 + \Delta x_1, x_2, t) - f(x_1, x_2, t)
\]

so that, with the representation \(F(.)\) for \(f(.)\), \(\partial f(.)/\partial x_1\) can be represented by:

\[
\frac{\Delta F(n_1, n_2)}{\Delta x_1} = \frac{F(n_1 + 1, n_2) - F(n_1, n_2)}{\Delta x_1} = \frac{\partial f(\text{Mesh2Domain}(n_1, n_2), t)}{\partial x_1}
\]

which can again be expressed by an abstraction function.

The discretization of the function \(f(\vec{x}, t)\) shows that a reformulation step (in this case the discretization) is usually specified by a number of abstraction functions, one for each kind of data entity. In this case, the discretization required separate abstraction functions for the domain, for functions on the domain and for derivatives of functions. The further discussion below will focus on the representation of the domain.
Chapter 5. Data partitioning and Data structures

5.2 The Domain and its Representation

The first part of the discussion will be dealing with the representation of the data entities in the abstract code specification in Equations (5.1)-(5.3) by data structures in the program code. This representation will be done in two reformulation steps. The first of these is the discretization as given in Section 5.1.3. The result of that step, i.e. the representation of the domain by a collection of Mesh-points, will be the starting point for the discussion in this section.

The discrete version of the domain as obtained in Section 5.1.3 has been called the Mesh, in keeping with the usual nomenclature. But in further reformulations, the set of points that represents the domain can not be usefully referred to by the name Mesh. Therefore, the more generic term Index Set will be used. An index set is an interval in \( \mathbb{N} \) (see also Section 2.1.1). In fact, the Mesh can also be the Cartesian product of two index sets according to this definition if it has a rectangular shape.

The use of index sets is not limited to the representation of the spatial domain. The set of time-points for which the solution of Equations (5.1)-(5.3) is computed can also be represented by an index set. The same holds for instance for the set of chemical components in a furnace simulation: it can be represented by an index set in which each point represents a single chemical component.

In the following, two different index set representations of collections of Mesh-points and functions thereon will be considered. In addition, a discussion will be given of two common ways of combining index sets. The index set representations can be directly implemented in array-based languages such as Fortran77. Therefore, these representations are in fact identical to the data structures in a program code.

5.2.1 One-to-One Representations

The first type of representation of Mesh-points that will be considered, is typically used when one wants to represent data that is only defined in an arbitrary subset of all Mesh-
5.2. The Domain and its Representation

points. For example, the function \( f(x,t) \) in Equation (5.1) is a function that represents some sort of input to the domain. In a coastal water simulation, \( f(x,t) \) may represent the influx of pollution into the water. In that case \( f(x,t) \) will be zero almost everywhere, except at locations where for instance drilling platforms are located. Thus, there is no need to represent \( f(x,t) \) for all \( x \) in the domain: only the points where it is non-zero are of interest.

Consider the Mesh-version of \( f(x,t) \), i.e. \( F(n_1,n_2) \). The points of the Mesh where \( F(n_1,n_2) \) is non-zero can be represented by an index set, \( I_1 \) by numbering the points in an arbitrary way. This constitutes a reformulation of \( F(n_1,n_2) \) into a representation of the form \( F(i) \), where \( i \in I_1 \).

The reformulation of the Mesh-points into the index set \( I_1 \) can be formally expressed by the abstraction function \( \text{Interval2Subset}(.) : \)

\[
\text{Interval2Subset} : I_1 \rightarrow \text{Mesh},
\]

\[
\text{Interval2Subset}(i) = (n_{1,i}, n_{2,i})
\]

Note that each point in \( I_1 \) is associated with exactly one point in \( \text{Mesh} \). Therefore, this type of abstraction function will be called a One-to-One representation. A Many-to-One representation will be encountered below.

With this representation of the Mesh-points, the representation of the discrete function \( F(n_1,n_2) \) can be expressed by:

\[
\forall (n_1,n_2) \in \text{Mesh} : F(n_1,n_2) = \begin{cases} 
  f \in \text{Real2IR}(F(i)) & \text{where } (n_1,n_2) = \text{Interval2Subset}(i) \\
  \text{zero elsewhere}
\end{cases}
\]

where \( \text{Real2IR}(.) \) is the function that transforms a real value as represented on a computer to the range of values in \( \mathbb{R} \) that are all represented by that same real value. It will be assumed below that different real numbers on a computer represent disjoint intervals in \( \mathbb{R} \), i.e. that

\[
\text{Real2IR}(a) \cap \text{Real2IR}(b) \neq \emptyset \iff a = b
\]

It should be emphasized that the function value of \( F(n_1,n_2) \) in points that are not represented in \( I_1 \) does not have to be zero: it can take any constant value. The value zero is used here only for explanatory purposes.

5.2.2 Many-to-One Representations

Consider the function \( g(.) : \partial \Omega \times \mathbb{R} \rightarrow \mathbb{R} \) in the specification of the abstract problem given by Equations (5.1)-(5.3), which is the function that defines the right-hand side of the boundary condition equation. This function is only defined on \( \partial \Omega \), the boundary of the domain; like \( f(.) \), it will be assumed to be independent of \( t \). In discretizing the domain \( \Omega \) (see Section 5.1.3), \( g(.) \) will have been represented by a semi-discrete version of it, \( G(.) : d\text{Mesh} \rightarrow \mathbb{R} \), where \( d\text{Mesh} \) denotes the set of boundary points of \( \text{Mesh} \).

In case of a finite difference discretization, the boundary will often consist of a number of straight sections, where \( G(.) \) is a constant on each section. In that case, one could
number the sections with numbers from an index set $I_3$ and represent $G(.)$ by a new function $G(.) : I_3 \rightarrow \mathbb{R}$. The function $G(.)$ is lateron easily represented by an instance of a data structure in the computer program.

In this case, the set of boundary points ($dMesh$) will be represented through a set of boundary sections, that are each referred to by an index from the index set $I_3$. This index set is in fact a compact representation of the boundary: each point in $I_3$ represents an entire section in $dMesh$. The relation between boundary points and the points in $I_3$ is an example of what will be called a Many-to-One representation.

The relation between points in $I_3$ and points in $dMesh$ (which is a subset of $Mesh$) is given by the abstraction function $Interval2SetofSubsets(.)$:

$$Interval2SetofSubsets : I_3 \rightarrow Mesh^\wedge,$$

$$Interval2SetofSubsets(r) =$$

$$\{(n_1, n_2) \in Mesh \mid (n_1, n_2) \text{ in straight line from } (n_{1,0,r}, n_{2,0,r}) \text{ to } (n_{1,e,r}, n_{2,e,r})\}$$

where the symbol $Mesh^\wedge$ denotes the powerset of $Mesh$ (i.e. the set of all subsets of $Mesh$), and where $(n_{1,0,r}, n_{2,0,r})$ and $(n_{1,e,r}, n_{2,e,r})$ denote the start and end of the line segment. The function $G(.)$ can then be represented on the index set $I_3$ by:

$$\forall (n_1, n_2) \in Mesh :$$

$$G(n_1, n_2) = G(r) \text{ where } (n_1, n_2) \in Interval2SetofSubsets(r), \ r \in I_3$$

The PARPRE/PARPOS tool that will be discussed in Section 5.4, supports only situations where each boundary section is a straight line along the border of $Mesh$. The main reason for this will be discussed at a later stage (see Section 5.3.3), but at this point it may suffice to say that this particular type of abstraction function is all that is needed to describe Many-to-One representations in the TRIWAQ program for which PARPRE/PARPOS was constructed.

### 5.2.3 Index Set Combinations

Many programs use combinations of index sets. The two kinds of combinations that will be considered here are a sort of union construct and a Cartesian product construct.

The first kind of index set combination is used in cases where one wants to store different kinds of data or data from different subsets of the domain in one array. For example, in a finite difference code for 2D-flow simulation, one may wish to store flow velocities at several cross-sections of the domain. Assume for the moment that there are two cross-sections and that $I_1$ is the index set that represents the points in the first cross-section and $I_2$ is the index set that represents the points in the second cross-section. Suppose furthermore that one uses a single array to store the information about both cross-sections, where the first $|I_1|$ positions are used for storing data from the first cross-section and the remaining $|I_2|$ positions are used for data from the second cross-section. Then the set of array indices is effectively the concatenation of index sets $I_1$ and $I_2$.

The abstraction function $Union2Set(.)$ that links such a concatenation of index sets to a set of $Mesh$-points is very similar to those that were discussed above. When $I_3$ is the
5.2. The Domain and its Representation

concatenation of index sets \( I_1 \) and \( I_2 \), both of which are One-to-One representations of sets of Mesh-points, then

\[
\text{Union2Set} : I_3 \rightarrow \text{Mesh},
\]

\[
\text{Union2Set}(i) = \begin{cases} 
\text{Interval2Subset}_1(i) & \text{if } i \leq \text{card}(I_1) \\
\text{Interval2Subset}_2(i - \text{card}(I_1)) & \text{if } i > \text{card}(I_1)
\end{cases}
\]

where \( \text{Interval2Subset}_1(.) \) and \( \text{Interval2Subset}_2(.) \) are the abstraction functions of the index set \( I_1 \) and \( I_2 \) respectively.

A product construct is typically used to store data that relates to the interaction between two points in the domain. For example, in a molecular dynamics simulation, one may wish to store the strength of interaction between the pairs of molecules. In that case, each pair of molecules is best represented by a number \((i_1, i_2)\), where \(i_1\) is the number of the first molecule and \(i_2\) is that of the second molecule. Now, let \( I_1 \) be the set of molecules, then the set of molecule-pair numbers, \( I_4 \), is the product of \( I_1 \) with itself:

\[
I_4 = I_1 \times I_1
\]

As a second example, suppose that one is interested in the positions of the molecules at a number of times \( t \), where \( t \in T \) and where \( T \) is the set of times at which the positions must be determined. Then the position of molecule \( i_1 \) at time \( t_1 \) can be represented by the tuple \((i_1, t_1)\). If \( I_4 \) denotes the set of all tuples \((i_1, t_1)\), then

\[
I_4 = I_1 \times T
\]

In the two examples above, each element of \( I_1 \) represented a molecule. However, in finite difference computations, \( I_1 \) may be used to represent a set of Mesh points, so that each element of \( I_1 \) represents a point in Mesh. Now let index sets \( I_1 \) and \( I_2 \) both represent sets of Mesh points through One-to-One representations \( \text{Interval2Subset}_1(.) \) and \( \text{Interval2Subset}_2(.) \) respectively. Then each element in the product of \( I_4 = I_1 \times I_2 \) represents a pair of Mesh points. The relation between an element in \( I_4 \) and the corresponding pair of Mesh points can be expressed through an abstraction function \( \text{Product2Set}(.) \):

\[
\text{Product2Set} : I_4 \rightarrow \text{Mesh}^2,
\]

\[
\text{Product2Set}(i, j) = (\text{Interval2Subset}_1(i), \text{Interval2Subset}_2(j))
\]

The representation of functions that are defined on combinations of Mesh-points will not be considered separately here: it follows immediately from the way in which the sets of Mesh-points are represented.

5.2.4 Concluding Remarks

In this section, two essentially different types of representations of sets of Mesh-points have been discussed: the One-to-One representation and the Many-to-One representation. In addition, two forms of representations have been discussed that are in fact combinations of One-to-One or Many-to-One representations.
Chapter 5. Data partitioning and Data structures

For each representation, the abstraction function has been given that links data arrays in computer programs to functions on the \( \text{Mesh} \). Together with the discretization that is given in Section 5.1.3, these abstraction functions define the transformation that lead from the data arrays back to the abstract entities which they represent. In this way, a formal description of data structures has been given.

However limited the collection of abstraction functions that is discussed here may seem to be, it is sufficient to describe the data structures in already a very wide class of real-life Fortran programs. For example, the abstraction functions of the type \( \text{Interval2Subset}(.) \) and \( \text{Interval2SetofSubsets}(.) \) both occur often in sparse storage schemes which are used in simulation codes that are based on the solution of sparse matrices.

Still, it is not claimed that the collection of abstraction functions that has been discussed is sufficient in general. However, the line of working that will be presented here does carry over to other types of abstraction functions.

5.3 The Effect of Partitioning

One way to parallelise a code that solves the problem defined in Equations (5.1)-(5.3) is by decomposing the domain \( \Omega \) into a number of subdomains. Each of these subdomains can then be discretized, just like the original domain \( \Omega \) is represented by \( \text{Mesh} \) (see Section 5.1.3). In this way, the domain is effectively represented by a collection of SubMeshes (i.e. the Meshes for the subdomains).

In many cases, the Mesh is decomposed (partitioned) directly without first considering the corresponding decomposition of the domain. This line of working also gives a collection of SubMeshes that together represent \( \Omega \). In this chapter, this second approach will be followed. In terms of Figure 5.2: not the left downward arrow will be followed, but the middle one.

The previous section has discussed several different representations of sets of points in the Mesh: a One-to-One representation, a Many-to-One representation and combinations of these two. When the Mesh is decomposed, this will usually also lead to a decomposition of the index sets that result from such representations (e.g. \( I_1, I_2, I_3 \) and \( I_4 \) in the previous section). The relation between the partitioned form of such index sets and their original form will be the basis for distributing data from a full Mesh representation to a SubMesh representation.

This section will first discuss formally the decomposition of the Mesh into SubMeshes. Then, for each of the abstraction functions from the previous Section, it will be shown whether and how the SubMesh data can be retrieved from the data for the full Mesh. The mechanism to do this is based on a general expression for the relation between index sets that are related to the full Mesh and their partitioned counterparts. This relation will be called \( \text{GlobalNr}(.) \). The potential and limitations of this relation will be discussed. A practical application of the theory of this section will be discussed in Section 5.4.
5.3. The Effect of Partitioning

5.3.1 Partitioning of the Mesh

Let $P_{\text{Mesh}}(.)$ be a $P$-way partitioning of the Mesh, i.e. a function that assigns an integer value in the range $[1, P]$ to each Mesh-point. This partitioning produces a set of submeshes $\{\text{SubMesh}_p \mid p = 1, \ldots, P\}$ defined by:

$$\text{SubMesh}_p = \{(n_1, n_2) \in \text{Mesh} \mid P_{\text{Mesh}}(n_1, n_2) = p\}$$

Note that $\bigcup_p \text{SubMesh}_p = \text{Mesh}$ and that $\text{SubMesh}_{p_1} \cap \text{SubMesh}_{p_2} = \emptyset$ when $p_1 \neq p_2$, i.e. the submeshes do not overlap.

It is often useful to add a guardband to each SubMesh. A guardband is an extension around a submesh that does not belong to the submesh itself but is used e.g. to define Neumann-type boundary conditions on the SubMesh. An extended SubMesh will be denoted as SubMesh$^G$, where the $G$ indicates the presence of the guardband. In holds that, $\text{SubMesh}_p \subseteq \text{SubMesh}^G_p$. The guardband points can be points of Mesh, but that is not strictly necessary. It might be useful in certain cases to add a guardband point to a SubMesh at a location that has no counterpart in the original Mesh.

The representation of the Mesh by a collection of SubMesh$^G$'s can be formally expressed through an abstraction function that relates points in the SubMesh$^G$'s to points in Mesh. This abstraction function will be called $\text{SubMesh}^G \times \text{Mesh}(\cdot)$, and is defined by:

$$\text{SubMesh}^G \times \text{Mesh} : \bigcup_p \text{SubMesh}^G_p \rightarrow \mathbb{N}^2,$$

$$\text{SubMesh}^G \times \text{Mesh}(n_1, n_2, p) = \begin{cases} (n_1, p, n_2) & \text{if } (n_1, p, n_2) \in \text{SubMesh}_p \\ \text{arbitrary} & \text{elsewhere} \end{cases}$$

where "arbitrary" means that the image of $(n_1, n_2, p)$ for points outside SubMesh$^G_p$ is not a priori defined: it can be chosen freely for any particular application of the abstraction function.

Note that the range of this abstraction function is not the Mesh, but the wider set $\mathbb{N}^2$ (remember that Mesh $\subseteq \mathbb{N}^2$). This is because the guardband points can be mapped onto points outside the original Mesh. This fact will be encountered below when discussing the distribution of data from the Mesh to the Submeshes.

In the definition of $\text{SubMesh}^G \times \text{Mesh}(\cdot)$ above, it has been assumed that the coordinates of a point in one of the SubMeshes are identical to those of the corresponding point in Mesh. However, the coordinates can be chosen more or less freely within each SubMesh. An obvious example of this is the fact that coordinates of the lower-left point of a (Sub)Mesh are usually taken to be $(1, 1)$.

If such a free choice of coordinates per SubMesh is used, then the definition of the abstraction function $\text{SubMesh}^G \times \text{Mesh}(\cdot)$ changes slightly to take the coordinate transform from SubMesh points to Mesh points into account:

$$\text{SubMesh}^G \times \text{Mesh} : \bigcup_p \text{SubMesh}^G_p \rightarrow \mathbb{N}^2,$$

$$\text{SubMesh}^G \times \text{Mesh}(n_1, p, n_2, p) = \begin{cases} \text{CoordTrans}_p(n_1, p, n_2, p) & \text{if } \text{CoordTrans}_p(n_1, p, n_2, p) \in \text{SubMesh}_p \\ \text{arbitrary} & \text{elsewhere} \end{cases}$$
where $\text{CoordTrans}_p(.)$ is some coordinate transformation that relates coordinates in a submesh $\text{SubMesh}_p$ to coordinates in the original $\text{Mesh}$. Again, the word "arbitrary" implies that the image of points outside $\text{SubMesh}_p$ can be chosen to suit the situation in which the abstraction function is applied.

The partitioning of the $\text{Mesh}$ into $\text{SubMeshes}$ also leads to a reformulation of the functions that are defined on the $\text{Mesh}$, like $F(.)$ and $G(.)$ (see Sections 5.1.3 and 5.2).

If the function value of a function, $F(.)$, is independent of the size and shape of the $\text{Mesh}$ and of the coordinate system used in it, then the relation between a function $F(.)$ that is defined on the $\text{Mesh}$ and its representation $F_p(.)$ on each of the $\text{SubMesh}_p$'s is easily defined:

$$F_p : \text{SubMesh}_p \rightarrow \mathbb{R},$$

$$F_p(n_{1,p}, n_{2,p}) = \begin{cases} F(\text{SubMeshes2Mesh}(n_{1,p}, n_{2,p})) & \text{where } (n_{1,p}, n_{2,p}) \in \text{SubMesh}_p \\ \text{arbitrary} & \text{elsewhere} \end{cases}$$

Note that this abstraction function does not impose a function value in points outside $\text{SubMesh}_p$; this function value can be chosen to be anything that is appropriate in a given situation.

But if the function value of $F(.)$ is somehow dependent on the size and shape of the $\text{Mesh}$ or on the coordinate system that is used in it, then the relation of the $F_p(.)$'s to $F(.)$ is not as obvious. Consider for instance a function $F(n_1, n_2)$ defined as

$$F(.) : \text{Mesh} \rightarrow [1, N_1], F(n_1, n_2) = n_1.$$

Suppose furthermore that the partitioning of $\text{Mesh}$ puts the point $(n_1, n_2)$ in part $p$, and take for instance

$$\text{SubMeshes2Mesh}(n_{1,p}, n_{2,p}) = (n_{1,p} + 5, n_{2,p}) \text{ if } (n_{1,p} + 5, n_{2,p}) \in \text{SubMesh}_p.$$  

Then, if the original definition of $F(.)$ is also used to define $F_p(.)$, then

$$F_p(n_{1,p}, n_{2,p}) = n_{1,p} \neq F(\text{SubMeshes2Mesh}(n_{1,p}, n_{2,p})) = n_{1,p} + 5$$

or, if Relation (5.5) is imposed, then it must hold that

$$F_p(n_{1,p}, n_{2,p}) = n_{1,p} - 5$$

so that the definition of $F_p(.)$ differs from that of $F(.)$. Although this example may seem to be a bit contrived, it does show that, if $F_p(.)$ is to have exactly the same definition on $\text{SubMesh}_p$ as $F(.)$ has on $\text{Mesh}$ then Relation (5.5) does not hold when the function value of $F(.)$ is somehow defined in terms of the size, shape or coordinate system of $\text{Mesh}$.

The following will discuss the consequences of the $\text{Mesh}$ partitioning for index sets that are related to the $\text{Mesh}$ through the different kinds of representations that have been discussed, and for the functions that are defined on those index sets.
5.3. The Effect of Partitioning

5.3.2 One-to-One Representations

In order to extract SubMesh data from the Mesh data, it must be known what the relation is between an index set that represents a set of points in a SubMesh and the index set that represents the same set of points in the full Mesh. This section will first discuss that relation in case the representation is of a One-to-One type. Then, a general procedure will be sketched to extract SubMesh data from Mesh data, again for the case of a One-to-One representation. The power and the limitations of that procedure will be discussed. The cases of Many-to-One representations and index set combinations will be discussed later.

A. The Partitioning of the Index Set

Consider an index set \( I_1 \) that is related to the Mesh by an abstraction function of the form \( \text{Interval2Subset}(.) \) as defined above in Section 5.2.1. A partitioning of the Mesh obviously induces a partitioning on \( I_1 \). Let this partitioning be denoted by \( \mathcal{P}_{I_1}(.) \). Then \( \mathcal{P}_{I_1}(.) \) is given by:

\[
\forall i \in I_1 : \mathcal{P}_{I_1}(i) = \mathcal{P}_{\text{Mesh}} \circ \text{Interval2Subset}(i)
\]

where \( \circ \) denotes function composition as usual.

Just like the partitioning of the Mesh leads to the representation of the Mesh by a collection of SubMeshes, so does the partitioning of \( I_1 \) lead to a representation of that index set by a collection of sub-index sets, \( I_{1,p} \). Now, the relation between a point in one of the \( I_{1,p} \)’s and the corresponding point in \( I_1 \) can be derived as follows:

\[
\forall i_1 \in I_1 : \text{Interval2Subset}(i_1) = (n_1, n_2) \in \text{Mesh} = (n_1, n_2) \in \text{SubMesh}_p \text{ for } p = \mathcal{P}_{\text{Mesh}}(n_1, n_2) = \text{SubMeshes}_2\text{Mesh}(n_{1,p}, n_{2,p}) \text{ for some } (n_{1,p}, n_{2,p}) \in \text{SubMesh}_{G_p} = \text{SubMeshes}_2\text{Mesh} \circ \text{Interval2Subset}_p(i_{1,p}) \text{ for some } i_{1,p} \in I_{1,p}
\]

assuming that an \( \text{Interval2Subset}_p(.) \) exists such that the collection of all \( (n_{1,p}, n_{2,p}) \) can be represented. This issue will be addressed further below. Now, note that \( \text{Interval2Subset}^{-1}(.) \) must exist because \( \text{Interval2Subset}(.) \) itself is a One-to-One relation. Therefore, the relation between \( i_1 \) and \( i_{1,p} \) can be written as:

\[
i_1 = \text{Interval2Subset}^{-1} \circ \text{SubMeshes}_2\text{Mesh} \circ \text{Interval2Subset}_p(i_{1,p})
\]

where \( p = \mathcal{P}_{I_1}(i_1) \)

This entire expression can be denoted more briefly by introducing a new name for the relation between \( i_1 \) and \( i_{1,p} \). This relation will be called \( \text{GlobalNr}_p(.) \), defined as:

\[
\text{GlobalNr}_p(.) = \text{Interval2Subset}^{-1} \circ \text{SubMeshes}_2\text{Mesh} \circ \text{Interval2Subset}_p(.) \quad (5.6)
\]

This function forms the basis of the procedure that will be sketched below for extracting SubMesh data from full Mesh data. First, some peculiarities of the \( \text{GlobalNr}_p(.) \) function will be discussed and an efficient algorithm will be given to determine this function. Then it will be shown how it can be applied.
B. A further Discussion of the \( \text{GlobalNrp}(.) \) Function

The \( \text{GlobalNrp}(.) \) function that is defined by Equation 5.6 will form the basis of the discussion below. Therefore, it is worthwhile to take a closer look at that function before continuing the discussion. The most important issue with regard to the \( \text{GlobalNrp}(.) \) function is that its definition in Equation 5.6 implies that it can only handle points in \( I_{1,p} \) that have a direct counterpart in the original \( I_1 \). In practical situations, however, there are likely to be points in \( I_{1,p} \) that do not have a counterpart in \( I_1 \), as will be shown.

The second issue that will be considered here is that, although the \( \text{GlobalNrp}(.) \) function can in principle be evaluated by evaluating the left-hand side expression of Equation 5.6, this is not always the most efficient procedure. An algorithm will be presented that efficiently constructs a lookup table to represent the function. This lookup table can then be used in the actual data distribution.

First, consider the fact that \( \text{GlobalNrp}(.) \) can only be applied to points in \( I_{1,p} \) that have a counterpart in \( I_1 \), or, stated differently, that the domain of \( \text{GlobalNrp}(.) \) can actually be smaller than \( I_{1,p} \). This can be seen by noting that the domain of \( \text{GlobalNrp}(.) \) consists only of those points in \( I_{1,p} \) for which \( \text{Interval2Subset}(.) \) gives a point in \( \text{SubMesh}_G(.) \), i.e. in the domain of \( \text{SubMeshes}_G(.) \), and for which \( \text{SubMeshes}_G \circ \text{Interval2Subset}(.) \) gives a point in the set that is represented by \( I_1 \), i.e. in the domain of \( \text{Interval2Subset}^{-1}(.) \).

Now, in general, there will be points in \( I_{1,p} \) which are outside the domain of \( \text{GlobalNrp}(.) \). First of all, it is sometimes useful to reserve extra locations in the index set \( I_{1,p} \) to temporarily hold copies of data that actually belongs to another \( \text{SubMesh} \) but is needed for the computations on \( \text{SubMesh}_p \). In that case, the range of \( \text{Interval2Subset}(.) \) may be larger than \( \text{SubMesh}_G(.) \), so that it is no longer a proper subset of the domain of \( \text{SubMeshes}_G(.) \). Such points are therefore not in the domain of \( \text{GlobalNrp}(.) \).

Secondly, there may be elements in \( I_{1,p} \) that represent guardband points in \( \text{SubMesh}_G(.) \). According to the definition of \( \text{SubMeshes}_G(.) \) above (see Section 5.3.1), guardband-points do not have to be mapped onto Mesh-points, and thus are also not necessarily mapped onto points that are represented in \( I_1 \). Consequently points in \( I_{1,p} \) that represent guardband-points may be outside the domain of \( \text{GlobalNrp}(.) \), depending on the actual definition of \( \text{SubMeshes}_G(.) \) for guardband-points.

Thirdly, it may occur that the set that is represented by \( I_1 \) is defined implicitly, e.g. as the set consisting of the four cornerpoints of the \( \text{Mesh} \). Then, if one wants to have a set \( I_{1,p} \) in \( \text{SubMesh}_p \) with the same meaning as \( I_1 \) there may be points in \( I_{1,p} \) corresponding to points in \( \text{SubMesh}_p \) that do not have a counterpart in the set that is represented by \( I_1 \). In the example of the set consisting of the four cornerpoints of \( \text{Mesh} \), the set \( I_{1,p} \) should represent the four cornerpoints of \( \text{SubMesh}_p \), but these points are not, in general, cornerpoints of \( \text{Mesh} \) and thus are not represented in \( I_1 \). Therefore, the domain of \( \text{GlobalNrp}(.) \) does not include these points.

In a different form, this issue was already addressed above in the discussion that led to the definition of of \( \text{GlobalNrp}(.) \). There, it was mentioned that the range of \( \text{SubMeshes}_G^{-1} \circ \text{Interval2Subset}(.) \) must be representable through a function \( \text{Interval2Subset}(.) \). Now, it may not be possible to define the latter function completely analogous to \( \text{Interval2Subset}(.) \), as can again be seen from the same example of the
5.3. The Effect of Partitioning

corner-point index set that was just discussed. Suppose that the abstraction function \( \text{Interval2Subset}(.) \) is defined such that its domain is assumed to consist of exactly four points, which a perfectly valid assumption for the original \( \text{Mesh} \). Now, the four corner-points of \( \text{Mesh} \) will, after application of \( \text{SubMeshes2Mesh}^{-1}(.) \), usually each be located in different \( \text{SubMeshes} \). Then, within one particular \( \text{SubMesh}_p \), there will be at most one point in the domain of \( \text{Interval2Subset}_p(.) \) instead of the four that are needed to define \( \text{Interval2Subset}_p(.) \) analogous to \( \text{Interval2Subset}(.) \). In this case, one can either add the additional points to make the meaning of the index set on the \( \text{SubMeshes} \) consistent with that in the original \( \text{Mesh} \) (but accepting the fact that the \( \text{GlobalNr}_p(.) \) function does not apply to these additional points, see above), or one must use a different representation instead of the \( \text{Interval2Subset}_p(.) \) that requires exactly four points in \( I_{1,p} \).

The fact that the domain of \( \text{GlobalNr}_p(.) \) can be smaller than \( I_{1,p} \) is not too surprising. In all of the possible causes that were just discussed, the essential point is that there are points in \( I_{1,p} \) that have no counterpart in the original \( I_1 \): the \( \text{SubMesh} \) representation of the problem contains entities that are not present in the original \( \text{Mesh} \) representation. This means that it is not necessarily possible to construct the \( \text{SubMesh} \) representation of the problem exclusively from its \( \text{Mesh} \) representation. This is a very important point to note, since it implies that rewriting a \( \text{Mesh} \) problem into an equivalent set of coupled \( \text{SubMesh} \) problems is not easily automated: the rewriting process requires at least some intelligence.

In the discussion below, it will be convenient to extend the domain of \( \text{GlobalNr}_p(.) \) to the full \( I_{1,p} \). For that purpose, its definition must be extended to define also a function value for points in \( I_{1,p} \) that do not have a counterpart in \( I_1 \). In those points, the function value of \( \text{GlobalNr}_p(.) \) will be defined to be -1:

\[
\text{GlobalNr}_p : I_p \rightarrow I \cup \{-1\},
\]

\[
\text{GlobalNr}_p(i_p) = \begin{cases} 
\text{Interval2Subset}^{-1} \circ \text{SubMeshes2Mesh} \circ \text{Interval2Subset}_p(i_p) \\
\text{if } \text{Interval2Subset}_p(i_p) \in \text{SubMesh}_p \\
\land \exists i \in I : \text{SubMeshes2Mesh} \circ \text{Interval2Subset}_p(i_p) = \text{Interval2Subset}(i) \\
-1 \text{ elsewhere}
\end{cases}
\]

This definition forms the basis for the algorithm that is shown in Figure 5.3. This algorithm constructs a lookup table for the function \( \text{GlobalNr}_p(.) \) for all \( p \).

It starts by creating, for each \( \text{SubMesh} \), the set that will be represented by \( I_{1,p} \) (using \( \text{Mesh} \) coordinates). This is done either by explicitly constructing the set (cf. identifying the four corner-points of a \( \text{SubMesh} \) in the example above) or by determining the intersection of the original set that is represented by \( I_1 \) with each of the \( \text{SubMeshes} \). The construction of the \( \text{SubMesh} \) set is briefly denoted in Figure 5.3 by the call to \( \text{CreateSet}(.) \).

The set that is thus created can then be extended with some extra points, e.g. points to temporarily store data from other \( \text{SubMeshes} \) (see discussion above). In the presentation of the algorithm, it is assumed that these points are known explicitly (i.e. the set \( \text{ExtraPoints} \)) but they may also be defined implicitly, for instance through some operation on \( \text{Set}_p \).
procedure Create_GlobalNr(.)
input: sets $I_1$, $SubMesh_p$, $ExtraPoints$
integer $p$
functions $Interval2Subset(.)$, $CoordTrans(.)$
output: set $I_{1,p}$
functions $GlobalNr_p(.)$, $Interval2Subset_p(.)$
algorithm:
$Set_p = CreateSet(Interval2Subset, I_1, SubMesh_p)$
$Set_p = Set_p \cup ExtraPoints$
$> I_{1,p} = [1, \text{card}(Set_p)]$
for $ip=1, \text{max}(I_{1,p})$ do
let $(n_1, n_2) \in Set_p$
$Set_p = Set_p \setminus (n_1, n_2)$
$Interval2Subset_p(ip) = CoordTrans(n_1, n_2)$
if $\exists i \in I_1 : (n_1, n_2) = Interval2Subset(i)$ then
$GlobalNr(ip) = i$
else
$GlobalNr(ip) = -1$
endif
endfor
$Permutation = \text{permutation of } I_p \text{ s.t. } GlobalNr_p \text{ is non-decreasing}$
$GlobalNr_p = GlobalNr_p \circ Permutation$
$Interval2Subset_p = Interval2Subset_p \circ Permutation$
endprocedure

Figure 5.3: Algorithm to construct the $GlobalNr$ function

Then, the index set $I_{1,p}$ and its abstraction function $Interval2Subset_p(.)$ are created by numbering the elements of the subsets with contiguous numbers starting at one. Each time an element of $Set_p$ is numbered, the corresponding point in $Mesh$ is determined (effectively the operation $SubMeshes2Mesh(.)$). If that point does also have a number in the range of $Interval2Subset(.)$, then that number is recorded in $GlobalNr_p(.)$. If not, then $GlobalNr_p(.)$ is set to $-1$.

Finally, a permutation is applied to the numbers of the elements of $Set_p$ to make $GlobalNr$ non-decreasing. This is done to make sure that the order in which points occur in the local $I_{1,p}$ is the same as the order in which they occur in $I_1$. It may not be necessary, but it probably will not harm either. Also, this permutation will put all $ExtraPoints$ together at the start of $I_{1,p}$. 
5.3. The Effect of Partitioning

C. Functions on One-to-One Representations

Once the $GlobalNr_p(.)$ function is available, it is a relatively easy task to extract data for $SubMesh_p$ from data that is available for $Mesh$. Consider for example a function $F(.)$ that is non-zero only in a subset of $Mesh$-points. This subset can be represented by an index set $I_1$ through an abstraction function of the form $Interval2Subset(.)$. When such a representation is used, this leads to a representation of $F(.)$ by a function $F(.)$ whose domain is the index set $I_1$, as was discussed in Section 5.2.1. The representation is defined by:

$$ F(n_1, n_2) \in \text{Real2IR}(F(Interval2Subset^{-1}(n_1, n_2))) \tag{5.8} $$

In Section 5.3.1 above, it was shown that the partitioning of $Mesh$ into $SubMeshes$ leads to a decomposition of $F(.)$ into a number of functions $F_p(.)$, where

$$ F_p(n_{1,p}, n_{2,p}) = F(SubMeshes2Mesh(n_{1,p}, n_{2,p})) \text{ where } (n_{1,p}, n_{2,p}) \in SubMesh_p \tag{5.9} $$

under the conditions that were discussed in Section 5.3.1. If the functions $F_p(.)$ are now represented analogously to the function $F(.)$, then

$$ F_p(Interval2Subset_p(i_{1,p})) \in \text{Real2IR}(F_p(i_{1,p})) \tag{5.10} $$

Equations (5.8)-(5.9) hold simultaneously only when

$$ F(GlobalNr_p(i_{1,p})) = F_p(i_{1,p}) \tag{5.11} $$

This can be seen as follows. Let $f = F_p(n_{1,p}, n_{2,p})$ and let $(n_{1,p}, n_{2,p}) = Interval2Subset_p(i_{1,p})$. Then, from Equation (5.10):

$$ f \in \text{Real2IR}(F_p(i_{1,p})) $$

Also, from Equation (5.9) and Equation (5.8):

$$ f \in \text{Real2IR}(F(Interval2Subset^{-1} \circ SubMeshes2Mesh \circ Interval2Subset_p(i_{1,p}))) $$

so that

$$ f \in \text{Real2IR}(F_p(i_{1,p})) \cap \text{Real2IR}(F(Interval2Subset^{-1} \circ SubMeshes2Mesh \circ Interval2Subset_p(i_{1,p}))) $$

Using now the fact that $\text{Real2IR}(a) \cap \text{Real2IR}(b) \neq \emptyset \iff a = b$ (see Section 5.2.1), it must hold that

$$ F_p(i_{1,p}) = F(Interval2Subset^{-1} \circ SubMeshes2Mesh \circ Interval2Subset_p(i_{1,p})) $$

which gives Equation (5.11) when substituting the definition of $GlobalNr_p(.)$ (see Expression 5.7) in the argument of the left-handside, and assuming that $SubMeshes2Mesh(n_{1,p}, n_{2,p}) \in \text{Domain}(Interval2Subset^{-1}(\cdot))$.

In this proof of Equation (5.11), it has been assumed that Equation (5.9) holds, which is not the case if $F(.)$ is defined in terms of the size or shape of the $Mesh$ or in terms of coordinates of $Mesh$-points (see Section 5.3.1). If Equation (5.9) does not hold then the
value of \( F_p(.) \) will, in general, have to be determined from the definition of \( F_p(.) \), since the value of \( F_p(.) \) in a point \((n_1,p,n_2,p)\) is not necessarily a simple copy of the value of \( F(.) \) in the Mesh point that corresponds to \((n_1,p,n_2,p)\).

Also, it has been assumed that \( SubMeshes2Mesh(n_1,p,n_2,p) \in \text{Domain}(Interval2Subset^{-1}(.)) \) and that \((n_1,p,n_2,p) \in SubMesh_p\), or, stated differently, that \( GlobalN_{rp}(i_1,p) \neq -1 \). If these assumptions do not hold, then \( i_1,p \) does not have a counterpart in \( I_1 \) and consequently the function value of \( F_p(i_1,p) \) can not possibly be determined from the value of \( F(i_1) \) for some \( i_1 \), since there is no point \( i_1 \) in the domain of \( F(.) \) that corresponds to \( i_1,p \). In this case, again, the value of \( F_p(i_1,p) \) must be determined explicitly from the definition of \( F_p(.) \).

### 5.3.3 Many-to-One Representations

The outline of the discussion in this section about Many-to-One representations will be similar to that of the previous one. First, the consequences of mesh-partitioning will be discussed with regard to index sets that result from a Many-to-One representation. It will be shown that the procedure for extracting \( SubMesh_p \) data from the original Mesh data is similar to that for One-to-One representations. After that, a short discussion will be given of how this extracting of data is actually done, and which pitfalls should be kept in mind.

#### A. The Partitioning of the Index Set

Let \( I_2 \) be an index set that is related to the Mesh through an abstraction function of the form \( Interval2SetofSubsets(.) \) as described above. This means that each element of \( I_2 \) is a representation of a set of Mesh-points (see Section 5.2.2). When the Mesh is partitioned, then the set of Mesh-points that is associated with a particular element of \( I_2 \) is possibly scattered over several \( SubMesh_p \)'s. Formally, when \( SubSet_{i_2} \) denotes the set of Mesh-points associated with an element \( i_2 \) of \( I_2 \), then:

\[
Interval2SetofSubsets(i_2) = SubSet_{i_2} = \bigcup_{p}(SubSet_{i_2} \cap SubMesh_p) = \bigcup_{p}\{SubMeshes2Mesh(n_1,p,n_2,p) | (n_1,p,n_2,p) \in Interval2SetofSubsets_p(i_2,p)\}
\]

where \( Interval2SetofSubsets_p(.) \) is the \( SubMesh_p \) analogue of the full-Mesh abstraction function \( Interval2SetofSubsets(.) \).

There are two important points to note about this derivation. Firstly, the function \( Interval2SetofSubsets_p(.) \) may have to be taken differently from the function \( Interval2SetofSubsets(.) \), as will be explained shortly. Secondly, the derivation assumes that the set of points that is associated with \( i_2 \) is fixed, i.e. does not depend on things like the size or coordinate system of the Mesh. A few more words will be spend on this below.

Usually, one would like to use an abstraction function \( Interval2SetofSubsets_p(.) \) that is similar to the function \( Interval2SetofSubsets(.) \) that is used on the non-partitioned Mesh. Then, the same data structures can be used to represent data that is defined on \( I_2 \), both
in the original code and in the partitioned version. Consequently, a minimum of changes to the program code have to be made in that case.

But this is only possible if the subset that is to be represented by \(i_{2,p}\) has the correct properties for the function \(\text{Interval2SetofSubsets}(\cdot)\) to be applicable. Above, when defining \(\text{Interval2SetofSubset}(\cdot)\), it was stated that the Many-to-One abstraction function that is supported by the PARPRE/PARPOS tool (see Section 5.4) assumes that the subset points lie on a straight line. This means that the parts of the subset within each \(\text{SubMesh}_p\) also lie on a straight line (assuming that \(\text{SubMeshes2Mesh}(\cdot)\) does not distort it) and can thus be represented in the same way as the original line of points in \(\text{Mesh}\).

This is one of the reasons for only supporting abstraction functions of this particular form in PARPRE/PARPOS. Other forms of abstraction function may also be supportable, but if the intersection of a subset with an arbitrarily shaped \(\text{SubMesh}_p\) can not be represented by the abstraction function, then matters will obviously become far more complex.

For example, consider a set of square areas of the \(\text{Mesh}\), and let the area be numbered with numbers in some index set \(I_2\). Then the abstraction function \(\text{Interval2SetofSubsets}(\cdot)\) that defines the square area for each area-number can be defined by giving for each area-number the coordinates of its lower-left point and the length of one of its sides. But if the part of this set of points that falls in a particular \(\text{SubMesh}_p\) is not square, then it is not possible to use an analogous definition of \(\text{Interval2SetofSubsets}_p(\cdot)\), since the part of the area in \(\text{SubMesh}_p\) is not completely defined by the coordinates of the lower-left point and the length of one of its sides. Thus, the definition of \(\text{Interval2SetofSubsets}_p(\cdot)\) must be different from that of \(\text{Interval2SetofSubsets}(\cdot)\).

The derivation that is sketched above assumes that the subsets in the \(\text{SubMeshes}\) are in fact representations of the subsets in the original \(\text{Mesh}\). This is not necessarily the case when the subsets are defined in terms of properties of the \(\text{Mesh}\), like e.g. the set of points that are on the extreme left of the \(\text{Mesh}\). In this case, the \(\text{SubMesh}\) representation must be constructed from the local mesh properties. Again, as in the case of One-to-One representations, this occurs when the set represented by \(I_2\) or the elements of that set are defined in terms that relate to the size or form of the \(\text{Mesh}\) or the coordinate system that is used in it.

Just like for One-to-One representations, it is possible to introduce a compact notation for the relation between points \(i_{2,p}\) in the index sets \(I_{2,p}\) on the \(\text{SubMesh}_p\)'s and the corresponding point \(i_2\) in \(I_2\). This relation will also be denoted by \(\text{GlobalNr}_p(\cdot)\) but in this case the definition of the \(\text{GlobalNr}_p(\cdot)\) relation is somewhat different from that for One-to-One representations. Here, the definition is:

\[
\text{GlobalNr}_p(i_{2,p}) = i_2 \iff \\
\forall (n_{1,p}, n_{2,p}) \in \text{Interval2SetofSubsets}_p(i_{2,p}) : \\
\text{SubMeshes2Mesh}(n_{1,p}, n_{2,p}) \in \text{Interval2SetofSubsets}(i_2) \land \\
\mathcal{P}_{\text{Mesh}}(\text{SubMeshes2Mesh}(n_{1,p}, n_{2,p})) = p
\]

The procedure to determine the \(\text{GlobalNr}\) function in for Many-to-One representations is a very straightforward extension of that for One-to-One representations; a detailed
Chapter 5. Data partitioning and Data structures

discussion will be omitted here.

Note, finally, that in the case of Many-to-One representations, each point in the index set \( I_2 \) can have counterparts in several different \( SubMesh_p \)'s. Thus, the \( Mesh \)-partitioning does not simply lead to a partitioning of \( I_2 \). In fact, what comes closest to a kind of partitioning function for \( I_2 \) is the function \( P_{I_2}(.) \) defined by:

\[
P_{I_2} : I_2 \rightarrow \{1, P\}^2,
\]

\[
P_{I_2}(i_2) = \{p \mid \exists(n_1, n_2) \in Interval2SetofSubsets(i_2) : P_{Mesh}(n_1, n_2) = p\}
\]

where the symbol '\(^2\)' denotes a powerset.

B. Functions on Many-to-One representations

Functions on index sets that are formed by Many-to-One representations are in fact functions that give a value for a set of \( Mesh \)-points at once. For instance, if \( G(.) \) is such a function, then \( G(i_2) \) could specify a function value for the set of \( Mesh \)-points that is represented by \( i_2 \). Functions on Many-to-One representations fall apart into two classes.

The first class is that of functions for which the function value is the same for all elements of the set of \( Mesh \)-points that is represented by an index. An example from finite difference analysis is a boundary type number associated with a section of the boundary. Such a boundary type number can specify that the section is e.g. a Neumann boundary or a Dirichlet boundary. This implicitly defines the boundary condition in each point of the boundary section.

The second class is that of functions for which the function value for an index \( i_2 \) indirectly gives the function value in all \( Mesh \)-points that are associated with \( i_2 \). For example, let the set of points that is associated with \( i_2 \) be a line of \( Mesh \)-points. Let the function value in these points be linear in the distance of each point to the first point of the line, e.g.:

\[
\forall(n_1, n_2) \in \text{line} : G(n_1, n_2) = G(n_{1,0}, n_{2,0}) + a \cdot (n_1 - n_{1,0})
\]

where \((n_{1,0}, n_{2,0})\) are the coordinates of the first point of the line. Then it is sufficient to store just \( G(n_{1,0}, n_{2,0}) \) and \( a \) rather than to explicitly store the function value in each point.

For the first kind of function, it is relatively easy to extract data for \( SubMesh_p \) from the data for \( Mesh \). In this case, it holds for any particular \( p \) that:

\[
\forall (n_1, n_2) \in Interval2SetofSubsets(i_2) :
\]

\[
\text{Real2IR}(G(i_2)) = G(n_1, n_2) \quad \text{(see Expression 5.4)}
\]

\[
\Rightarrow \forall (n_1, n_2) \in Interval2SetofSubsets(i_2) \cap SubMesh_p :
\]

\[
\text{Real2IR}(G(i_2)) = G(n_1, n_2)
\]

\[
\Rightarrow \forall(n_{1,p}, n_{2,p}),
\]

\[
SubMeshes2Mesh(n_{1,p}, n_{2,p}) \in Interval2SetofSubsets(i_2) \cap SubMesh_p :
\]

\[
\text{Real2IR}(G(i_2)) = G_p(n_{1,p}, n_{2,p}) \quad \text{(see Expression 5.5)}
\]

\[
\Rightarrow \text{for } i_{2,p}, \text{GlobalNr}_p(i_{2,p}) = i_2 :
\]

\[
\text{Real2IR}(G(\text{GlobalNr}_p(i_{2,p}))) = \text{Real2IR}(G_p(i_{2,p}))
\]
5.3. The Effect of Partitioning

using the definition of $GlobalN_{p}(.)$ for Many-to-One representations. This equality holds when

$$g(GlobalN_{p}(i_{2,p})) = g_{p}(i_{2,p})$$

Thus, for functions that are represented through a Many-to-One representation, the procedure for extracting $SubMesh_{p}$ data from $Mesh$ data is identical to the one that was discussed above for One-to-One representations.

The second class of functions, i.e. functions for which the function value is indirectly represented through a Many-to-One representation, can generally not be distributed in this way. Instead, the function values have to be evaluated from the local subsets. For instance, consider the example that was given above in which the function value on a line of points is given by a linear expression. Then the first point of the line within a $SubMesh_{p}$ is not necessarily the point $(m_{0}, n_{0})$, so the function value in that first point is not necessarily equal to $G(m_{0}, n_{0})$. Instead, it will have to be computed from the original $G(m_{0}, n_{0})$ and the value of $a$.

5.3.4 Union and Product Representations

After having discussed the consequences of $Mesh$ partitioning for data that is stored through One-to-One or Many-to-One representations, the theoretical discussion will be finished now with a brief consideration of the consequences for functions on index set combinations. The effects of partitioning on the index-set combinations themselves will be discussed first, followed by a few remarks regarding functions that are defined on such index-set combinations.

A. The Partitioning of the Index Set

When $I_{3}$ is the union of two index sets $I_{1}$ and $I_{2}$ (see Section 5.2.3), then the partitioning of $I_{3}$ is completely specified by the partitioning of $I_{1}$ and $I_{2}$. Also, it is easily seen that

$$GlobalN_{3,p}(i_{3,p}) = \begin{cases} GlobalN_{1,p}(i_{3,p}) & \text{where } i_{3,p} \leq |I_{1,p}| \\ |I_{1}| + GlobalN_{2,p}(i_{3,p} - |I_{1,p}|) & \text{where } i_{3,p} > |I_{1,p}| \end{cases}$$

(5.12)

where $GlobalN_{x,p}(.)$ is the $GlobalN_{p}(.)$ function for index set $I_{x}$, $x = 1, 2, 3$, and where $I_{x,p}$ is the domain of $GlobalN_{x,p}(.)$. Thus, unions of index sets pose no special problems.

For an index set $I_{4}$ that is formed by the product of two index sets $I_{1}$ and $I_{2}$, the matter is more problematic. Let $GlobalN_{x,p}(.)$ and $I_{x,p}$ be defined as above. Then it might seem at first glance that $I_{1,p} \times I_{2,p}$ is the representation of the part of $I_{4}$ within $SubMesh_{p}$. But this is not correct. Note that

$$\forall (i_{1}, i_{2}) \in I_{4} :$$

$$(i_{1}, i_{2}) = (GlobalN_{1,p_{1}}(i_{1,p_{1}}), GlobalN_{2,p_{2}}(i_{1,p_{2}})) \text{ for some } p_{1}, p_{2} \in [1, P].$$

where in general $p_{1} \neq p_{2}$. Thus, the product construct can associate a point in $I_{1,p_{1}}$ for some $p$ with a point in $I_{2,p_{2}}$ for some $p_{2} \neq p_{1}$, i.e. with an element of an index set in a different $SubMesh$. 


A possible solution is to extend \( I_2 \) with a guardband which essentially extends the size of the local index set \( I_{2,p} \) to that of the full index set \( I_2 \) of the original \textit{Mesh}. Then the product \( I_{1,p} \) also has a guardband which is sufficiently wide to store information for points \( i_{2,p} \) for all \( p \) in \([1,P]\).

In many cases, though, only one of the index sets \( I_1 \) and \( I_2 \) will be a representation of \textit{Mesh}-points, whereas the other can be associated with points in time or in some other dimension which is replicated across the processors. In those cases, the product of \( I_1 \) and \( I_2 \) does not lead to problems. E.g. the product of set the local \textit{Mesh}-points with the set of time frames is readily constructed on each processor once the \( \text{GlobalNr}_{p}(.) \) function is known for the index set that represents \textit{Mesh}-points.

### B. Functions on Combinations of Index Sets

Functions that are defined on combinations of index sets, i.e. on the union or product of index sets, can be handled using the same mechanisms as described above for One-to-One representations and for Many-to-One Representations.

Consider first the case of a function defined on a union of index sets. Let \( F(.) \) be a function defined on an index set \( I_3 \) which is the union of index sets \( I_1 \) and \( I_2 \). Let \( \text{GlobalNr}_{3,p}(.) \) be given by Equation 5.12. Then

\[
\forall i_{3,p} \in I_{3,p} : F_p(i_{3,p}) = F(\text{GlobalNr}_{3,p}(i_{3,p}))
\]

again under the restrictions that were discussed above with regard to the existence of the \( \text{GlobalNr}_{p}(.) \) functions that define \( \text{GlobalNr}_{3,p}(.) \) and on the definition of the function \( F(.) \).

The case of functions defined on the product of two index sets is similarly simple. Let \( F(.) \) be defined on an index set \( I_4 \) which is the product of index sets \( I_1 \) and \( I_2 \). Let \( \text{GlobalNr}_{x,p}(.) \) and \( I_{x,p} \) be defined as above. Then

\[
\forall (i_{1,p}, i_{2,p}) \in I_{4,p} : F_p(i_{1,p}, i_{2,p}) = F(\text{GlobalNr}_{1,p}(i_{1,p}), \text{GlobalNr}_{2,p}(i_{2,p}))
\]

assuming \( I_{4,p} \) is sufficiently large, as discussed above. Again, the conditions under which the expression holds are similar to those that were discussed for One-to-One and Many-to-One representations.

### 5.3.5 Concluding Remarks

This section has shown how a formal treatment of the way in which abstract functions are represented by data structures allows for the definition of a general mechanism (i.e. the \( \text{GlobalNr}_{p}(.) \) function and its application) to extract the data for each \( \text{SubMesh}_{p} \) from the \textit{Mesh} data.

It has also been shown that, in some cases, data for a \( \text{SubMesh}_{p} \) can not directly be extracted from the \textit{Mesh} data, and sometimes can not be extracted at all. This happens in particular when the set of \textit{Mesh}-points in which a function has meaningful values is not fixed, but depends on the size or form of the \textit{Mesh} or on coordinates of the \textit{Mesh}-points.
The discussion that has been given is not meant to be exhaustive. Even though the representations that were discussed cover a wide range of practical situations, many other kinds of representations exist. The consequences of partitioning for those other representations can be investigated along the same line of reasoning as has been applied in this section.

5.4 Practical Application of the Theory

To illustrate the discussion above, and to show the practical applicability of the results, this section will describe the PARPRE/PARPOS tool, which implements many of the ideas that have been presented in Sections 5.2 and 5.3. Because PARPRE/PARPOS is a dedicated tool, it needs not, and in fact does not, offer a full implementation of all that has been described in this chapter. Even so, it has proved to be a flexible and reliable tool.

5.4.1 The PARPRE/PARPOS Tool

The PARPRE/PARPOS tool has been developed to distribute input data for the TRIWAQ program which is used for the simulation of flow and transport processes in coastal waters. TRIWAQ is applied in a.o. water quality management, tidal prediction and the assessment of the consequences for coastal flow of new dams, dikes and dredging works.

For some applications the meshes that discretize the problem domain can become large; current 2D problems have a size that is typically around 10,000 meshpoints, but there is a growing interest in simulating 3D problems, which readily become larger than 100,000 meshpoints. Problems of this size take an impractically long time to simulate and this has motivated an effort to turn to parallel computing to reduce the throughput time.

TRIWAQ takes the specification of the problem to be simulated from an input file that contains a description of the Mesh, the model parameters, the initial state and the boundary conditions. The basic approach to the parallelisation of TRIWAQ has been to split this input file into a number of smaller files, each specifying the problem to be simulated on a part of the total Mesh. These subproblems are then simulated in parallel using a slightly modified TRIWAQ code for each of the subproblem.

The parallel run produces a separate output file for each of the subproblems. After the run, these output files are collected into a single output file, which is (practically) identical to the file that would have been produced by a sequential TRIWAQ run. The entire procedure is illustrated in Figure 5.4.

PARPRE/PARPOS is the tool that determines the partitioning of the Mesh, splits the input file into files for each of the SubMeshes and, after termination of the simulation, collects the results into a single output file. It consists of a preprocessing phase (PARPRE, i.e. the determination of the partitioning and the splitting of input files) and a post processing phase (PARPOS, i.e. the collection of the output files into a single file).

To split the input file into SubMesh files, PARPRE first reads the contents of the input file into similar data structures as those that are used within TRIWAQ, then it splits these
Figure 5.4: Schematic overview of the structure of the parallel version of TRIWAQ, showing the place of the PARPRE/PARPOS tool therein.

The data structures used by TRIWAQ are far from trivial. There are about 140 arrays, of which only a few have the same index domain. Also, indirect addressing schemes are extensively used in TRIWAQ. This is necessary because the domain of the simulation, i.e. the wet part of a coastal area, can take a very irregular shape (see e.g. Figure 4.12). Such an irregular shape can not easily be represented on a single rectangular finite difference mesh, not even when using boundary-fitted coordinates. Consequently, the boundaries of the Mesh are usually not straight lines, and this makes the specification of boundary conditions problematic. The current way to do this is by using an indirect addressing
5.4. Practical Application of the Theory

INDEXSETS

INDSET( NAME='MMAX' , CARD = 'IDIMEN(2)' , DISTRIBUTETE, MAP=1 )
INDSET( NAME='NMAX' , CARD = 'IDIMEN(3)' , DISTRIBUTETE, MAP=2 )
INDSET( NAME='MNMAXK' , CARD = 'IDIMEN(5)' , DISTRIBUTETE, MAP=3 )
INDSET( NAME='LDAM' , CARD = 'IDIMEN(7)' , DISTRIBUTETE, MAP=13 )
INDSET( NAME='NOROCO' , CARD = 'IDIMEN(9)' , DISTRIBUTETE, MAP=16 )
INDSET( NAME='NSLUV' , CARD = 'IDIMEN(12)' , DISTRIBUTETE, MAP=9 )
INDSET( NAME='NT0' , CARD = 'IDIMEN(14)' , DISTRIBUTETE, MAP=19 )
INDSET( NAME='KMAX' , CARD = 'IDIMEN(18)' , REPLICATE )

ARRAYS

ARR( NAME='GRID' ,TYPE='INT' ,INDSET='MMAX*MMAX' )
ARR( NAME='IDIMEN' ,TYPE='INT' ,INDSET='30' )
ARR( NAME='COMMSH' ,TYPE='REAL' ,INDSET='10' )
ARR( NAME='IOOPEN' ,TYPE='INT' ,INDSET='NT0*4' )
ARR( NAME='BARPOS' ,TYPE='INT' ,INDSET='NSLUV*3' )
ARR( NAME='IRGEO' ,TYPE='INT' ,INDSET='3*NOROCO' )
ARR( NAME='IDAM' ,TYPE='INT' ,INDSET='LDAM*2' )
ARR( NAME='INTERF' ,TYPE='REAL' ,INDSET='MNMAXK*(1+KMAX)' )

Figure 5.5: An index domain description file for PARPRE/PARPOS scheme.

Apart from this, indirect addressing is needed because simulation results are usually collected at particular points in the Mesh (the so called stations). The fact that these points can be located at arbitrary locations also calls for an indirect addressing scheme.

Analysis of the index domains of the arrays has shown that they can all be constructed from a set of about 50 basic index sets. Of these, about 20 represent (sets of) sets of Mesh-points, and the remaining 30 represent points in non-spatial dimensions, like the time-dimension or the dimension of chemical entities. The 50 basic index sets are combined by union and product constructs to form the index domains of each of the 140 arrays.

The index sets that are representations of spatial points can all be expressed by abstraction functions of the types that were discussed above: Interval2Subset(), and Interval2SetofSubsets(). Except for one case, the set of Mesh-points that is represented by each index set is independent of the form, size and coordinatization of the Mesh. Thus, for all but one of the index sets a GlobalNrP() function can be created using an algorithm like that in Figure 5.3, in which the operation CreateSet is a simple evaluation of the abstraction function Interval2Subset(). The one exception is an index set in which each element represents a part of the contour around the Mesh, but it is only used for visualisation and is not needed in the actual simulation itself.

After having determined a partitioning of the Mesh, the PARPRE/PARPOS tool
determines a \( GlobalN_{r_p}(\cdot) \) function for each of the spatial index sets through a call to one of the two routines that implement \( GlobalN_{r_p}(\cdot) \) creation (one for abstraction function \( Interval2Subset(\cdot) \) and one for abstraction function \( Interval2SetofSubsets(\cdot) \)). The \( GlobalN_{r_p}(\cdot) \) function for each of the remaining index sets is simply the identity function, as each element of a non-spatial index set is replicated on each \( SubMesh \). Guardbands are not added to index sets.

Next, PARPRE/PARPOS distributes the arrays using the \( GlobalN_{r_p}(\cdot) \) functions. The information about the way in which the basic index sets are combined into the index domain of each of the arrays is supplied to the tool through an input file, the so-called index domain description file. PARPRE/PARPOS only handles arrays that are described in this file.

The use of this file has made it easy to do the parallelisation incrementally. Each time a new part of TRIWAQ was ready to be used for parallel computing, the distribution of the necessary arrays could be performed by simply adding the arrays to the index domain description file. In this way, the full complexity of distributing TRIWAQs data remained manageable. Apart from this, the use of the description file has also proved to be advantageous, because some errors in the documentation regarding the data structures could be easily corrected for once they were discovered during the parallelisation effort.

Figure 5.5 shows a part of an index domain description file. It first contains lines that define the relation between the name of an index set (which is free to choose) and the number of the corresponding \( GlobalN_{r_p}(\cdot) \) function (called a MAP in the file). This specification also gives the array locations where cardinalities of index sets are stored, so that these cardinalities can be automatically adjusted in the \( SubMesh \) arrays to give correct values for the \( SubMesh \). This is the only kind of array value correction for which PARPRE/PARPOS has a general facility.

After the specification of index set names, the next lines in the index domain description file contain the specification of the index domain of the arrays. This specification must be done in terms of index set names or constants, which can be combined using a union construct (the '+' operator) or a product construct (the '*' operator).

For those arrays for which the value of one or more elements has to be corrected or reevaluated (because the value represents e.g. a coordinate or a quantity derived from coordinates) a correction is performed on the arrays for the \( SubMeshes \) to make their contents consistent again with properties of the \( SubMesh \). These correction routines are specific for TRIWAQ: no effort has been made to devise a general scheme for these corrections.

Summarising, PARPRE/PARPOS distributes arrays using build-in \( GlobalN_{r_p}(\cdot) \) tables and build-in correction routines. The flexibility in PARPRE/PARPOS is that it allows for the external specification of the index domain of arrays in terms of index sets for which a \( GlobalN_{r_p}(\cdot) \) function is known or which are to be replicated. This flexibility proved to be convenient in the process of parallelising TRIWAQ.

The applicability of PARPRE/PARPOS is limited by the fact that only build-in \( GlobalN_{r_p}(\cdot) \) functions can be used and that most of the corrections have explicitly been coded into the tool. Also, PARPRE/PARPOS can only handle index sets that are linked to sets of Mesh points by an abstraction function of one of the kinds that were discussed in this chapter. However, these limitations are not essential. The work on PARPRE/PARPOS
has suggested that much more of its functionality can be generalised to make it applicable to other codes than TRIWAQ as well.

5.5 Conclusions

This chapter has shown that some of the data structures in a code for the solution of a PDE can be understood as representations of sets of points in the Mesh which represents the domain of the PDE. The decomposition of the Mesh implies a decomposition of the data structures. The way in which this decomposition must be done is defined by the way in which the Mesh-points are represented by the data structure. For two common forms of representation, a procedure has been sketched to decompose the data structures and it has been discussed under what conditions these procedures are correct. The implementation of the theoretical considerations in the PARPRE/PARPOS tool has been described.

The discussion in this chapter has made it clear that an optimal distribution of data is not easily reached without considering the meaning of the data. The distribution and alignment statements that are presently supported by data parallel languages such as HPF are incapable of doing a distribution that takes the meaning of data into account. Therefore, these constructs are likely to fail for achieving high performance when communication between processors is not fast enough to allow for a sloppy distribution of data.

On the other hand, if the structure of data is taken into account, then the distribution becomes a much more knowledge-intensive procedure for which a general, fool-proof solution probably does not exist. The most promising way out of this dilemma is to formally describe the development steps that lead to a code and use code generation techniques to generate distribution software. A more flexible version of PARPRE/PARPOS could be a first effort in that direction.
Chapter 5. Data partitioning and Data structures
Chapter 6

Conclusion

This chapter will conclude the thesis by giving an overview of the main results that have been presented. It is in the nature of research that it tends to raise at least as many new questions as it answers old ones. This certainly holds for the work that has been described here. Section 6.2 will sketch briefly some topics related to the present work that could not be addressed in this research, but that should be considered in the future.

6.1 The work that has been done...

In a distributed memory parallel computer, the efficiency of computation is largely determined by the way in which data is located in the distributed memory and by the way in which the computations are assigned to the processors. This has been common knowledge in the field of parallel computing right from the time that the first distributed memory parallel computers were built. The reason is that communication between processors is usually slow, certainly in comparison with computation. In spite of the great advances in electronics, the situation is not much different today than it was a decade ago.

To cope with this problem, many researchers have worked on methods to quickly determine a close-to-optimal distribution of computations and data over a set of processors. In this context, optimal means that a minimum of data has to be sent from one processor to another and, at the same time, all processors get an equal share of the computational work.

The problem of finding such a distribution can be formulated as a partitioning problem. Given a set of computations, each representing an amount of computational work, and a set of data dependencies between them, a partitioning of the set of computations must be found in which all parts represent roughly the same total amount of work and in which there is a minimum of dependencies between computations in different parts. In this thesis, the problem has been referred to both as a partitioning problem and as a distribution problem, dependent on which of the two terms was most appropriate in the context.

In general, partitioning problems in parallel processing can not be solved exactly within an acceptable time-span. Therefore, the methods that are used to deal with partitioning problems in parallel processing only give approximate solutions. Many of such methods are known, but unfortunately there is little understanding of which one gives the best
solution for a particular problem. This makes it difficult to choose the proper partitioning method for a given partitioning problem.

This issue has been one of the main topics of this thesis. A start has been made at formalising knowledge about the effectiveness of partitioning methods in relation to properties of the problem to which they are applied.

In this thesis, properties of partitioning problems have been expressed in terms of \textit{workload functions} and dependencies between computations.

The concept of workload functions has been discussed extensively in Chapter 2. Basically, a workload function is the function that gives the amount of computational work that is associated with each of the elements of a set of computations. The set of computations can be represented by a set of numbers (an index set) by associating a number with each of the computations. In terms of index sets, the workload function takes the form \( W(.) : I \rightarrow \mathbb{R} \), where \( I \subset \mathbb{N} \) is the index set.

The concept of a workload function, as it is used in this thesis, applies only to sets of computations of which the elements are mutually independent. This is done to be able to use the workload function for studying how computational work is distributed over processors without being bothered by dependencies between computations.

The second main characteristic of a set of computations for the study of partitioning methods is the set of dependencies between the computations. Concepts and notations to work with this set of dependencies have already been developed by other authors and this thesis gratefully uses the familiar concepts. They have been introduced only briefly in Chapter 2.

Having thus defined a characterisation of partitioning problems in terms of workload functions and dependencies, the next aim in this thesis has been to study how the properties of a partitioning problem affect the outcome of partitioning methods that are applied to it. In Chapter 2 an overview of the existing knowledge in this field has been given. This survey has clearly shown that a lot of work remains to be done. Even for widely used methods such as Cyclic Blockwise Distribution and Recursive Coordinate Bisection, there is not a good theory to predict how effective they are for a given distribution problem. Because these two methods are so widely used, a further study of their effectiveness has been performed. The results of that study have been presented in Chapter 3.

Cyclic Blockwise Distribution (CBD) is a rather straightforward distribution method with one parameter (the \textit{cycle frequency}) whose value can be chosen at will. For a low value of the cycle frequency, the distribution is characterised by a low communication volume but a possibly very uneven distribution of computational work over the processors (i.e. a strong load imbalance). Increasing the value of the cycle frequency usually improves the balance in the workload distribution but also increases the communication volume. Hence, the parameter must be chosen such that the load balance is acceptable and the communication volume is not yet too large.

Chapter 3 has shown that the balancing effect of raising the cycle frequency in CBD is largely determined by characteristics of the workload function of the problem to which it is applied. The most important characteristic of a workload function that determines
the effect of the cycle frequency, is the rate at which it’s Fourier coefficients decay toward higher Fourier-frequencies.

An indication of this rate can be obtained from the workload function itself: if the workload function has sharp peaks, it has significant high-frequency Fourier coefficients. In that case increasing the cycle frequency in CBD can be expected to have little effect on the load imbalance. If there are no sharp peaks in the workload function, the Fourier coefficients are likely to be small at higher frequencies, and then the cycle frequency in CBD needs not be chosen very high in order to obtain a good load balance.

This knowledge, combined with the existing knowledge that the communication volume under CBD generally increases linearly with the value of the cycle frequency, makes it possible to estimate a priori the effect of raising the cycle frequency and thus to estimate a proper setting for it.

Recursive Coordinate Bisection is a method that gives, by construction, a balanced distribution of computations over the processors. But it has the disadvantage that some parts may become very elongated (i.e. have a high skewness), which can cause a high communication volume and possibly a high inter-part connectivity. Chapter 3 has shown, through a theoretical consideration, that this effect is only significant for somewhat pathological workload functions. In short: problems that are not properly handled by RCB are characterised by workload functions in which large regions have a value that is much smaller than the average value of the workload function.

The validity of these results and their practical implications have been illustrated in Chapter 4 by an extensive case study. The topic of the case study has been the partitioning problem that occurs in the parallelisation of a finite difference code for the solution of the shallow water equations for a coastal region. In this application, the computations are related directly to the grid that is used as a discretization of the coastal region, which in turn directly reflects the form of the actual coastal region. Therefore, the workload function is related to the topography of the region that is being simulated. Chapter 4 has shown how a model of the topography of coastal regions can be used to formulate a general model for the workload function of coastal water simulation.

Although the topography model, on which the workload function model is based, applies only on relatively small spatial scales (up to 100 km), it has several features that make it convenient for the purposes of this thesis. First of all, the resulting workload function model matches the assumptions of the theorems in Chapter 3, so that those theorems can be easily applied. Secondly, the topography model is parameterised with respect to the roughness of the coastal area. Hence, by varying the roughness-parameter in the model, the effect of CBD and RCB can be studied for simulations of different types of coastal areas.

Chapter 4 has discussed the effect of CBD and RCB for the model of the workload function of coastal water simulation. It has clearly illustrated the validity and usefulness of the theoretical results of the previous chapter. Increasing the cycle frequency in CBD has indeed been seen to effectively decrease load imbalances, but not without causing a problematically large communication volume. RCB, on the other hand, has been identified as a suitable partitioning method for this problem.
The case study has been concluded with some practical performance results to illustrate the effectiveness of RCB for a real parallel simulation of two existing coastal waters. These results have shown that, even though the connectivity of parts in an RCB distribution is relatively low, it is still too high to be efficiently handled by today’s communication networks. Also, the number of processors in the parallel computers that were used is rather small, so that many of the theoretical results of Chapters 2 and 3 of this thesis do not apply. The practical performance results have complemented the theoretical discussion in this thesis by showing the practical implications and the limitations of its results.

After having studied the effectiveness of several common partitioning methods, the thesis has turned to consider problems related to the implementation of support for data distribution. An effective distribution of data may be hampered by the structure of the data that must be distributed. In those cases, it may be necessary to restructure the data to allow its proper distribution. This restructuring can be done automatically when the data structures can be formally described. Chapter 5 has introduced a formal way of describing data structures, and has presented a tool that can use such a description for restructuring data to make an efficient distribution possible.

Altogether, this thesis has provided an improved understanding of the effectiveness of partitioning/distribution methods through an overview of current knowledge, through a theoretical study of two well-known partitioning methods and through an extensive case study. It has furthermore given a formal account of how data structures can be restructured to ease the distribution of data. This account will hopefully help in the development of automatic support for parallel programming and for the restructuring of existing codes to make them run effectively on a parallel computer.

6.2 ...and the work that remains to be done

This thesis has studied, among other things, the effect of characteristics of a distribution problem on the effectiveness of methods to solve it. The study has concentrated on two well-known distribution methods, but it should be extended to include other methods as well. Also, a lot of work remains to be done in collecting empirical data to show the practical relevance of theoretical results. An extensive empirical study of the effectiveness of partitioning methods in relation to properties of the partitioning problems to which they are applied is needed to provide a solid basis for further study.

With a more complete understanding of the pros and cons of partitioning methods in relation to problem characteristics, it should be possible to develop expert-systems that can make a well-founded choice for a partitioning method in cases where the human programmer can not determine it, or does not want to determine it.

Along the way, a further study of the effectiveness of partitioning methods will show where improvements to those methods are still to be made. This will make the work on partitioning methods more productive, as it makes it possible to concentrate effort in directions in which it is really needed.
6.2. ...and the work that remains to be done

An issue that has not been addressed in this thesis, but that is certainly relevant for the evaluation of partitioning methods, is the question of how a partitioning can influence the efficiency of parallel algorithms. For example, it is known that for some parallel iterative solution methods, the partitioning of the domain of computation can influence the number of iterations that are needed to find the solution. Thus, it is well possible that a partitioning that gives a perfect load balance and little communication for a certain program, must still be discarded because it leads to a bad performance of one of the algorithms. Very little formalised knowledge seems to exist on this topic.

The last chapter of this thesis has sketched a tool that can be used to reorganise data structures to allow an effective distribution of data. This tool uses a formal description of the data structures in a program, which it needs to correctly interpret the data in the data structures. In developing the formal description of data structures, an improved understanding has been gained of how the structure of data may complicate its distribution. Considering more different data structures will further enhance this understanding and can serve to extend the capabilities of the tool that has been described in this thesis. Because structure in the data of a program is presently considered to be one of the major obstacles for successful parallelisation of legacy code, an improved understanding of this issue will certainly be beneficial. It will make it possible to more effectively use parallel computers for existing codes and this will ease the acceptance of parallel computers as the very powerful computing engines they are.
Bibliography


[29] P. Brezany and V. Sipkova. Coupling parallel data and work partitioners to VFCS. In Dekker et al. [50].


Bibliography


Bibliography


Bibliography


Index

Abstract
- data entities, 111, 112
- problem specification, 111

Abstraction Functions, 110, 112–113
- Interval2SetofSubsets(.), see Interval2SetofSubsets(.)
- Interval2Subset(.), see Interval2Subset(.)
- Mesh2Domain(.), 113
- Product2Set(.), 117
- Union2Set(.), 116

Adjacency Matrix, 34, 51–53

Algorithm
- for generating fractal functions, 84–87
- Gaussian Elimination, 19–23, 69, 75
- MidPointFM2D, 86
- Repeated Step Fault, 87
- SpectralSynthesis, 87, 93

BLOCK, see Blockwise Distribution

Blockwise Distribution, 103, 105, 106

Cache Effect, 106

Clustering Method, 30, 35, 37, 57, 60

Coastal Water Simulation, 13, 79, 80, 89, 99–108, 110, 131

Communication, 27, 137
- free partitioning, 41
- costfunction, 27, 29, 96
- libraries, 58, 59
- minimisation, 28
- under Cyclic Blockwise Distribution, 45
- under Orthogonal Recursive Bisection, 47–56

Computation, 18, 19, 21
- dependency, 17, 26, 137

Computational Work, 17–19, 23–25, 137

Computer
- distributed memory parallel, 11, 137
- parallel, 11, 27, 101, 102
- sequential, 11
- shared memory parallel, 11

Connectivity
- of parts, 89, 95, 96, 101, 104, 107, 139
- of processors, 101

Contention, 105

Continental Shelf Model, 84, 99, 99, 104, 107

Costfunction, 27, 29

CSM, see Continental Shelf Model

Cutset, 48
- theorems, 52–54

Cycle Frequency, 30

Cyclic Blockwise Distribution, 30–32, 34, 37–47, 138
- analysis, 62–68, 139
- effect of cycle frequency, 31, 42, 87, 90–94, 99, 138, 139

Data Distribution, 109–135, 140
- automatical, 109, 131, 140

Data Structures, 110, 111, 114, 131
- formal description, 110, 111, 118, 140
- partitioned, 112
- reorganisation, 109, 111, 112

Differential Equations
- partial, 110, 111, 135
- Dilation, 54, 95, 96, 104

Distribution, see Partitioning
- of data, see Data Distribution

Domain Decomposition, 111

Dynamic Load Balancing, 29

Effectiveness
of Orthogonal Recursive Bisection, 77, 89–108
of partitioning methods, 18, 28, 30, 37, 38, 46, 47, 54, 57, 79, 89, 138
of Recursive Coordinate Bisection, 47–56, 71–139
Empirical Results, 27, 39, 45, 52, 54, 55, 57, 59, 99–108, 140
Fourier Transform, 64, 66, 83, 87, 91–94
Fractal Function, 83, 83–85
Fractional Brownian Motion, 83
Genetic Algorithms, 36, 37
GlobalNr(.), 118, 121, 122–124, 127, 129, 130, 133, 134
Graph
cutset, see Cutset
dependency, 29, 33
locality, 50
precedence, 20, 21, 21, 26, 27, 30, 48
separator, see Separator
Greedy Algorithm, 35, 37, 57
Hexagonal Partitioning, 31, 41
Hyperplane Partitioning, 31
Index Sets, 23, 39, 114, 116, 118, 121, 124–126, 128, 133, 134
combinations of, 116–117, 129–130
partitioning, 121, 126, 129
Indirect Addressing, 109
Inspector/Executor Mechanism, 59
Interface, 18, 26, 42
size, 39–42, 45–54
see also Skewness
Interval2SetofSubsets(.), 116, 118, 126, 129
Interval2Subset(.), 115, 117, 118, 121, 124, 125, 133, 134
Jagged Partitioning, 33, 37
Kernighan-Lin Algorithm, 35, 37, 57
Kriging, 82
Libraries
communication, 59
for communication, 58, 59
of partitioning methods, 60
with distributed data structures, 59
Load Balance, 28, 29, 38, 42, 48
Load Imbalance, 31, 33, 39, 42, 45, 49, 58, 91
relative decrease, 91, 94, 99
Many-to-One Representations, 115–116, 126–129
functions on, 128
Mapping, 47, 55, 56, 59
Mesh, 113–116, 118–121, 123, 127, 130, 131, 135
boundary of, 116, 132
coordinates, 126
data on, 118, 119, 125, 126, 128
functions on, 113, 118, 120
partitioning, 118–120, 125, 126, 129, 133
Neighbours, 26
distance between, see Dilation
number of, 54
Nested Dissection, 34, 37
Octasection, 33
Offset Vector, 26, 39, 42
One-to-One Representations, 114–115, 117, 121–126
functions on, 125
One-Way Dissection, 34
Optimisation
genetic algorithms, see Genetic Algorithms
in automatic parallelisation, 59
Kernighan-Lin Algorithm, see Kernighan-Lin Algorithm
linear programming, 35
methods, 35–37
simulated annealing, see Simulated Annealing
ORB, see Orthogonal Recursive Bisection
Ordering
Index

bit interleaving, 33, 37
Cuthill-McKee, 33, 37, 52, 55
Hilbert curve, 33, 37
methods, 33
one-way dissection, see One-Way Dissection
Orthogonal Recursive Bisection, 33, 89, 103–107
see also Recursive Coordinate Bisection
Connectivity, see Connectivity of parts
dilation, see Dilation
effectiveness, 71–77, 95–99
skewness, see Skewness

Parallel computing, 11, 17, 24, 137
programming language, 29, 57–61, 110
PARPRE/PARPOS, 110, 116, 127, 131–135
Parystec PowerXplorer, 102, 107
Partitioning, 24, 26, 27
automatic, 58
communication free, 41
heuristics, see Partitioning methods
hexagonal, see Hexagonal Partitioning
hyperplane, 31
jagged, see Jagged Partitioning
methods, 12, 30–37, 137
comparing, 37
of computations, 29
of domain, 109
of Mesh, see Mesh partitioning
package, 33, 35, 60
parallel, 33–36
problem, 12, 13, 17, 27, 37, 137
time, 28
Parystec PowerXplorer, 103, 106, 107
PDE, see Differential Equations, partial
Phase, 21, 28, 42
Power Spectral Density, 83, 84, 85
Product Representations, 116–117, 129–130, 133
functions on, 130
Program, 18, 19, 21
Programming Language, 111, 114
parallel, see Parallel programming language
Quadrissection, 33, 36
RCB, see Recursive Coordinate Bisection
Recursive Coordinate Bisection, 33, 34, 37, 49, 52, 54, 55
analysis, 71–75, 139
effectiveness, 139
Recursive Inertial Bisection, 34, 37, 52
Recursive Spectral Bisection, 34, 37, 52, 53, 55
Reformulation, 115, 120
steps, 111, 112, 114
Regular Distribution Method, 32, 30–32, 37–47, 51, 54, 55, 58
RIB, see Recursive Inertial Bisection
Roughness, 84, 89, 92, 97
RSB, see Recursive Spectral Bisection
Run-Time Support, 30, 32, 58, 59
Scattered Distribution, see Cyclic Blockwise Distribution
Sealevel, 89, 99, 92, 93, 95, 97
Semivariogram, 82, 83–86
Separator, 48
theorems, 49–52
Simulated Annealing, 36, 37, 53, 57
Skewness, 47, 71, 95, 96, 100, 104, 139
lower bound, 72
upper bound, 72
SORB, 103–106
Speedup, 104
SSTRIP, 104–107
Stencil, 18, 26, 39, 42
SSTRIP, 103, 104, 106
Time Complexity
of fractal generating methods, 86, 87
of partitioning methods, 37
Topography, 81–83, 87, 88
CSM, 84, 99–101
model, 81–85, 139
roughness, see Roughness
Waddensea, 84, 99–101
TRIWAQ, 13, 61, 102, 103, 105, 107, 116, 131, 131–135
parallel, 102, 131–135

Union Representations, 116–117, 129–130, 133
functions on, 130

Variable Block Method, 30, 32–35, 37, 39, 47, 54, 56, 58, 59

Waddensea Model, 84, 99
Weierstrass-Mandelbrot fractal function, see Fractal Function

Workload
distribution, 42
maximum, 28, 43, 44, 63, 64, 66
mean square deviation, 43, 66
minimisation, 27

for Coastal Water Simulation, 80–89, 139
general, 64–68
in higher dimensions, 24, 92
model, 80, 81, 88–89, 93
one-dimensional, 24, 98
trigonometric, 63–64

Workstation Cluster, 102, 103, 105–107
Summary

Large scale simulations of the flow of water and the transport of constituents in coastal areas are being performed in the Netherlands on a regular basis. Such simulations are needed for instance to predict the water levels along the coast and for environmental management. They are usually based on finite difference discretizations of the Shallow Water Equations.

To obtain a high resolution with finite difference discretizations, it is necessary to use a fine mesh, which in turn leads to a large number of grid points to cover a given area. When such a high resolution simulation has to be performed for a long simulation period, then the computing time becomes unacceptably long. One of the ways to reduce this computing time is to use parallel computers.

Such computers consist of several processing units that jointly execute a set of computations. To make best use of these processors, the computations have to be assigned to the processors in such a way that each processor gets approximately the same amount of work and such that the processors have to communicate with each other as little as possible. This can be achieved by a proper partitioning of the set of computations. Unfortunately, the task of finding such a partitioning is very complex.

In practice, partitionings are determined using heuristics, i.e. methods that do not necessarily give the optimum solution, but generally come close within an acceptable time. A great variety of such heuristics exists and all of them are suited for at least a portion of the partitioning problems that arise in parallel computing. However, there is presently too little understanding of which partitioning heuristic is most suitable for a particular distribution problem. This makes it difficult to tell a priori which heuristic(s) would be most useful for partitioning the computations in a parallel coastal water simulation. This thesis is an effort to collect additional knowledge on the effectiveness and applicability of partitioning methods for coastal water simulation.

In Chapter 2, the current volume of literature is surveyed to determine the current state of knowledge on the effectiveness of partitioning heuristics, both from a theoretical and from an empirical point of view. From this literature survey, two methods arise as particularly promising for coastal water simulation: Cyclic Blockwise Distribution and Recursive Coordinate Bisection. But the available results on these methods are insufficient to assess their applicability for parallel coastal water simulation.

Therefore, an analysis of these methods has been performed, the results of which are presented in Chapter 3. It is shown there that Cyclic Blockwise Distribution can give a well balanced partitioning (i.e. all processors get approximately the same amount of work to do), also when the amount of computational work per grid point varies during the simulation. But this balanced partitioning requires in general a prohibitively strong com-
munication between the processors. Recursive Coordinate Bisection, on the other hand, gives a balanced partitioning by construction that does not require excessive interprocessor communication. However, this method is not capable to deal with variations in the computational work per grid-point during a simulation.

In Chapter 4, the effectiveness of Blockwise Distribution (a special case of Cyclic Blockwise Distribution) and Orthogonal Recursive Bisection (ORB, a special case of Recursive Coordinate Bisection) for coastal water simulation are studied empirically. First, a model is developed that describes the grids and the workload per grid-point in coastal water simulation. Then, the effectiveness of Blockwise Distribution and ORB is assessed for generated instances of this model. The results confirm those of the theoretical analysis. Chapter 4 concludes with a study of the performance of two different parallel computing platforms for coastal water simulation using Blockwise Distribution and several different forms of ORB. This shows that Blockwise Distribution is in general not very suitable for this application and that ORB can give the required balanced, low-communication partitioning.

After the study of the effectiveness of partitioning methods, the thesis turns to another aspect of partitioning for parallel processing. One way of parallelising a sequential simulation program is to run several instances of the program in parallel, each computing a part of the grid. This has the advantage that the sequential code can be used almost in its original form. But it does require that each program gets its input presented as though it were input for a normal, sequential run. Thus, the data from the grid for which the simulation must be done, must be restructured into input data for each of the parallel instances of the code. This restructuring requires knowledge about the way in which the code expects its input, which in turn is reflected by its data structures. Chapter 5 of this thesis shows how the data restructuring can be done by considering how data structures for a mesh-based computation (such as finite difference coastal water simulation) can be decomposed into data structures for each of the sub-meshes. A general relation between mesh-data and sub-mesh-data is derived and it is shown under which conditions it holds. Finally, a tool is described which, using this relation, performs the restructuring automatically.
Samenvatting

Grootschalige simulaties van de stroming van water in kustgebieden en van het transport van in het water opgeloste stoffen worden in Nederland regelmatig uitgevoerd. Zulke simulaties zijn bijvoorbeeld nodig voor het voorspellen van de waterhoogten langs de kust en om na te gaan of voorgestelde constructies in kustwateren geen nadelige neven-effecten hebben. Dit soort simulaties is gewoonlijk gebaseerd op een eindige-differentie discretisatie van de ondiep-watervergelijkingen.

Om met een dergelijke discretisatie een hoge resolutie te bereiken, moet de roosterwijdte betrekkelijk klein gekozen worden. Dit vereist dan weer een groot aantal roosterpunten om een gegeven gebied te discretiseren. Als het dan bovendien nodig is om een lange simulatie-tijd door te rekenen, dan wordt de rekentijd die daarvoor nodig is al snel te lang voor praktische doeleinden. Een manier om deze rekentijd te reduceren, is door gebruik te maken van parallelle computers.

Zulke computers bestaan uit meerdere processoren die gezamenlijk berekeningen uitvoeren. Om de processoren optimaal te benutten, moeten de berekeningen zodanig verdeeld worden, dat elke processor ongeveer evenveel werk heeft en de processoren zo min mogelijk met elkaar hoeven te communiceren. Helaas is het bepalen van een dergelijke verdeling veelal zeer moeilijk.

In de praktijk wordt dan ook gebruik gemaakt van heuristieken: methoden die weliswaar niet gegarandeerd de optimale oplossing geven maar die snel een acceptabel resultaat opleveren. Er bestaan veel verschillende heuristieken, maar het is nog vrijwel niet bekend welke heuristiek de beste resultaten geeft voor een gegeven distributie probleem. Dat maakt het moeilijk om één of meerdere van dergelijke heuristieken te kiezen om daarmee de berekeningen in kustwater-simulaties te distribueren over parallelle processoren.

Hoofdstuk 2 van dit proefschrift geeft een overzicht van de bestaande literatuur betreffende de effectiviteit van distributie heuristieken. Daarbij wordt zowel aandacht besteed aan theoretische resultaten als aan gerapporteerde praktische ervaringen. Uit het overzicht blijkt dat er twee methoden zijn die mogelijk zeer bruikbaar kunnen zijn voor kustwater-simulaties: Cyclisch Blokgewijze Verdeling en Recursieve Coordinaten Bisectie. De kennis over deze methoden is echter niet voldoende om vast te kunnen stellen of ze inderdaad bruikbaar zijn voor deze toepassing.

Daarom is een analyse van deze methoden uitgevoerd, waarvan de resultaten werden gepresenteerd in Hoofdstuk 3. Daar wordt aangetoond dat Cyclisch Blokgewijze Verdeling weliswaar een zeer gelijkmatige verdeling van het rekenwerk over de processoren geeft (ook wanneer het rekenwerk per roosterpunt gedurende de simulatie varieert) maar daarbij wel leidt tot een intensieve communicatie tussen de processoren. Recursieve Coordinaten
Samenvatting

Bisectie geeft wel de gewenste gelijkmatige verdeling met weinig communicatie, maar kan weer niet inspelen op variaties in de hoeveelheid rekenwerk per roosterpunt gedurende de simulatie.

In Hoofdstuk 4 wordt dan langs empirische weg nagegaan of Blokgewijze Verdeling (een speciaal geval van Cyclisch Blokgewijze Verdeling) en Orthogonale Recursieve Bisectie (ORB, een speciaal geval van Recursieve Coördinaten Bisectie) voor parallelle kustwater-simulaties bruikbaar zijn. Allereerst is een model ontwikkeld dat een beschrijving geeft van de roosters die in kustwater-simulaties voorkomen en van de hoeveelheid rekenwerk per roosterpunt. De effectiviteit van Blokgewijze Verdeling en Orthogonale Recursieve Bisectie wordt bestudeerd door deze methoden toe te passen op een reeks instantiaties van dit model. De resultaten bevestigen de theoretische resultaten uit Hoofdstuk 3. Tenslotte worden in Hoofdstuk 4 nog de resultaten beschreven van parallelle kustwater simulaties op een tweetal computer systemen, waarbij gebruik is gemaakt van Blokgewijze verdeling en diverse varianten van Orthogonale Recursive Bisectie. Deze resultaten tonen eens te meer dat Blokgewijze Verdeling geen geschikte methode is, en dat met ORB redelijk goede resultaten te behalen zijn.

Nadat in de eerste hoofdstukken van dit proefschrift aandacht is besteed aan de effectiviteit van distributie methoden, wordt in Hoofdstuk 5 gekeken naar een ander aspect van distributie ten behoeve van parallel rekenen. De structuren waarmee gegevens in een programma opgeslagen worden, kan het namelijk moeilijk maken om de gegevens op de juiste manier over de processoren te verdelen. Voor eindige differentie discretisaties zijn de berekeningen veelal gekoppeld aan roosterpunten. Men kan dus de verdeling van berekeningen doen door de roosterpunten te verdelen. Dit kan op zijn beurt weer gebeuren door het rekenrooster op te delen in subroosters. Wanneer men nu de gegevens voor het originele rooster omzet in gegevens voor elk van de subroosters, en die gegevens op dezelfde manier structureert als die voor het globale rooster, kan voor het doorrekenen van elk van de subroosters gebruik worden gemaakt van vrijwel dezelfde programmatuur als die voor het globale rooster gebruikt wordt. Dit verbetert de onderhoudbaarheid van de programmatuur. In Hoofdstuk 5 wordt getoond hoe de omzetting van gegevens op het globale rooster naar gegevens op elk van de subroosters kan gebeuren. Er wordt een programma besproken dat deze omzetting automatisch kan doen. Dit programma is gebaseerd op een relatie tussen gegevens op de subroosters en die op het globale rooster, die wordt afgeleid door een beschouwing over hoe data structuren ontstaan in het ontwerp van een programma. Bovendien wordt getoond onder welke voorwaarden de relatie geldig is.
