DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

A review of data redistribution algorithms using block-cyclic distributions

Leonard Evers

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

A review of data redistribution algorithms using block-cyclic distributions

Ein Überblick über Datenumverteilungsalgorithmen unter Verwendung blockzyklischer Verteilungen

Author:	Leonard Evers
Supervisor:	Prof. Dr. Hans-Joachim Bungartz
Advisor:	Santiago Narvaez
Submission Date:	13.02.2025

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 13.02.2025 Leonard Evers

Abstract

Load balancing is a topic highly relevant to parallel computing. Distributing arrays across processors in a block-cyclic manner is one way to achieve it in a distributed memory framework. The ability to redistribute between different block-cyclic distributions allows for a more optimal load balancing that minimizes processor communication. This thesis presents several algorithms from the related literature for efficiently performing such redistributions. In doing so, we consider the optimization of both the computation of the message contents and the scheduling of communication between processors. Methods for handling multi-dimensional arrays as well as different source and target processor sets are also discussed. Furthermore, the results of the papers are used to compare the performance of the algorithms. We conclude that some algorithms can result in a several fold reduction in redistribution time.

Kurzfassung

Lastausgleich ist ein hoch relevantes Thema im Bereich des Parallelrechnens. Die Verteilung von Arrays über Prozessoren in block-zyklischer Weise ist eine Möglichkeit, dies in einem verteilten Speicher Modell zu erreichen. Die Fähigkeit, zwischen verschiedenen block-zyklischen Verteilungen umzuverteilen, ermöglicht einen optimierten Lastausgleich, der die Prozessor-Kommunikation möglichst gering hält. Diese Abschlussarbeit stellt mehrere Algorithmen aus der Literatur vor, die eine effiziente Umverteilung ermöglichen. Dabei betrachten wir sowohl die Optimierung der Berechnung der Nachrichteninhalte als auch das Scheduling der Kommunikation zwischen den Prozessoren. Zudem werden Methoden zur Verarbeitung mehrdimensionaler Arrays sowie verschiedene Quell- und Zielprozessor-Sets diskutiert. Darüber hinaus werden die Ergebnisse der betrachteten Arbeiten genutzt, um die Leistung der Algorithmen zu vergleichen. Wir kommen zu dem Schluss, dass einige Algorithmen die Umverteilungszeit um ein Vielfaches reduzieren können.

Contents

Abstract iii

Kurzfassung iv

1 Introduction 1

1.1 Motivation 1

1.2 Contributions 2

2 Definitions and Notations 3

2.1 Basic terminology 3

2.2 Extension to the multi-dimensional 4

2.3 Types of block-cyclic redistribution 5

2.3.1 Source and distribution factor 5

2.3.2 Shape retaining vs shape changing redistributions 6

3 Index Computation 7

3.1 Definition and Notation 7

3.2 Algorithm Analysis 8

3.2.1 1994: Runtime Array Redistribution in HPF Programs 8

3.2.2 1995: Automatic Generation of Efficient Array Redistribution Routines 12

3.2.3 1996: Optimizations for efficient Array Redistribution on Distributed Memory Multicomputers 15

3.2.4 1996: Efficient Algorithms for Array Redistribution 16

3.2.5 1997: Fast Runtime Data Redistribution - Indexing 17

3.2.6 1998: A Basic-Cycle Calculation Technique For Efficient Dynamic Data Redistribution 17

3.2.7 1998: Efficient Methods for Multidimensional Array Redistribution . . 20

3.2.8 2000: A Generalized Basic-Cycle Calculation Method for Efficient Array Redistribution 22

3.3 Performance Comparison 24

3.3.1 The general case 24

3.3.2 The one-divides-the-other case 26

3.3.3 The block-to-cyclic case 26

4 Communication **27**
4.1 Definition and Notation 27
4.1.1 Synchronous Communication 27
4.1.2 Asynchronous Communication 28
4.2 Algorithm Analysis 29
4.2.1 1994: An Approach to Communication Efficient Data Redistribution . . 29
4.2.2 1995: Multiphase Array Redistribution - Modeling and Evaluation . . . 33
4.2.3 1996: Redistribution of Block-cyclic Data Distributions using MPI . . . 37
4.2.4 1997: Fast Runtime Data Redistribution - Communication 43
4.2.5 1998: Scheduling Block-Cyclic Array Redistribution 45
4.2.6 1999: Efficient Algorithms for Block-Cyclic Array Redistribution Between Processor Sets 51
4.3 Performance Comparison 56
4.3.1 Scheduled vs non-scheduled 56
4.3.2 Schedule time comparison 58
4.3.3 Single-phase vs multi-phase 61

5 Discussion **63**
5.1 Conclusion 63
5.2 Future work 64

List of Figures **65**

List of Tables **66**

Acronyms **67**

Bibliography **68**

1 Introduction

1.1 Motivation

In high-performance computing, where efficiency, scalability, and performance are key, parallelism is of utmost importance. It allows for the simultaneous execution of multiple computations and the maximization of resource utilization, thereby reducing execution time. Using multiple processors yields the best results, but necessitates use of distributed memory at large numbers due to hardware limitations. Within such a distributed memory framework, splitting the data among the processors is necessary, so that each processor working on a different section of the data can efficiently access its part. Given this fact, the choice of how to distribute the data becomes important. Block-cyclic distributions are a popular choice because, given the correct block-size, they are able to achieve good load balancing and computational efficiency [1]. However, the correct block size is not fixed and changes depending on the operations being performed on the array elements. As such, the optimal block-cyclic distribution can change from one step of a data processing algorithm to the next.

Examples for algorithms that benefit from redistribution include the Alternating-direction Implicit (ADI) algorithms used to iteratively solve Slyvester matrix equations, which have the form $A \times X + B \times X = C$, with X being unknown. Because this method involves switching between updating the solution based on row-wise and column-wise calculations, redistributing the array accordingly is sensible. Not doing so would result in a large amount of communication overhead for at least half of the steps. This also applies to the two-dimensional Fast-Fourier Transform (FFT), which involves performing FFT on each dimension individually, e.g. row-wise and column-wise, when interpreting the data set as a matrix [2].

It is also possible that the amount of processors available for any given step of an algorithm changes: in this case too, the previous distribution would no longer be optimal. These cases represent a problem when attempting to achieve efficient computation at every point in an algorithm.

One possible solution for these problems is redistribution: changing the block-cyclic distribution across the processor sets. That means changing the block-size or adjusting the distribution for a different number of available processors. In some cases, both is necessary. In the case of the ADI method or the two-dimensional FFT this could involve switching from a row-wise to a column-wise distribution of the two-dimensional array. This process can be roughly divided into two steps: first, the computation of the processor number each block in the current distribution has to be sent to. Then secondly, the communication of this data:

from the message creation (packing) to the communication scheduling and the storage of the new blocks (unpacking). Both of these steps must be efficient: in order to make use of the improved data layout, the overhead of redistribution must be minimized as much as possible.

1.2 Contributions

This paper aims to explore the evolution of block-cyclic redistribution over time. In doing so, the primary focus will be on the aforementioned index computation and communication. In the respective sections, this thesis will explain and compare the different algorithms proposed in different papers with regards to functionality, general applicability and performance. Chapter 2 explains how we define block-cyclic distributions for different cases and what notations we use to describe them. Chapter 3 will discuss the process of index computation and how to perform it most efficiently. Chapter 4 will do the same for the scheduling of the inter-processor communication. Finally, chapter 5 will discuss the ramifications of this work, its limits as well as future work that can build on the results presented in this thesis. All algorithms and methods presented in this thesis stem from the papers the respective subsection of the thesis is named after.

2 Definitions and Notations

This section will define the redistribution problem more formally and introduce the notation to be used in the rest of the paper.

2.1 Basic terminology

We define the to be redistributed one-dimensional array as A, with size N and indexes 0 through $N-1$. We define the source processor set as P with size P_N and the destination processor set as Q with size Q_N. These sets are one-dimensional. Whenever we speak of a single processor set not in the context of source or destination distribution, we use P. p_i, then, describes the processor with rank i in the processor set P and takes the value of said rank when used in a mathematical context. In other words, $p_i = i$. In all cases, $0 \leq i < P_N$. q_i is defined analogously for the destination processor set Q.

We describe a given processor p_i's local array that contains the elements belonging to it in the distribution as that processors Source Local Array or Destination Local Array (SLA_i/DLA_i) depending on whether we are pre or post-redistribution [3].

Regular data distributions take one of three forms: cyclic, block, or block-cyclic. Cyclic and block distributions are defined as follows:

- Cyclic: Each array element is assigned to each processor in a round robin fashion
- Block: The array is split into P_N blocks of consecutive elements and each processor receives one in order of its rank

Block-cyclic distributions are a generalized form of cyclic and block distributions that are much more powerful. We can define all regular, one-dimensional distributions in the following manner: cyclic(x). This is defined as a block-cyclic distribution with block-size x. Then, in turn, a block-cyclic distribution of block size x given an Array A of size N with P_N processors can be defined as a mapping of global index n of array A to index tuple (p, b, i), where p is an element of the processor set P, b is the block number and i is the index that specifies the array elements storage location in the processors block described by b. This mapping is described by Equation 2.1 [4].

$$n \rightarrow (\lfloor (n \bmod (P_N \times x)) \div x \rfloor, \lfloor n \div (P_N \times x) \rfloor, n \bmod x) \tag{2.1}$$

Colloquially, this translates to the following: blocks of x elements of array A are assigned in a round robin fashion to each processor in P, looping around back to the start once the final one is reached. This is done until all elements are distributed. Both the block and cyclic distributions we mentioned earlier can be expressed using this definition: a cyclic distribution is equivalent to cyclic(1) and a block distribution is equivalent to cyclic($\lceil \frac{N}{P_N} \rceil$).

Let us consider a situation where $N = 18$ and $P_N = 3$. Figure 2.1 shows the array first in a cyclic distribution, then in a block distribution and finally, in a block-cyclic(3) distribution.

0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2

0	0	0	0	0	0	1	1	1	1	1	1	2	2	2	2	2	2

0	0	0	1	1	1	2	2	2	0	0	0	1	1	1	2	2	2

Figure 2.1: An array of size 18 in a cyclic, block and block-cyclic(3) distribution

In this case, each box stands for an array element and the number inside describes the processor it is assigned to.

2.2 Extension to the multi-dimensional

The extension to the multi-dimensional is fairly trivial: instead of having a block-cyclic(x) distribution we now have a block-cyclic($x_0, x_1, \ldots, x_{m-1}$) distribution, one parameter for each dimension and an array that is $N_0 \times N_1 \times \ldots \times N_{m-1}$. Typically, this is also accompanied by a processor grid that is $P_0 \times P_1 \times \ldots \times P_{m-1}$. In such a grid a given processor can be described in one of two ways: $p_{d_0,d_1,\ldots,d_{m-1}}$ with $0 \leq d_i \leq P_i$ or, alternatively, via p_i, defined by Equation 2.2 [5]. We make use of both notations depending on the context.

$$i = \sum_{k=0}^{m-1} (d_k \times \prod_{l=k+1}^{m-1} P_l) \tag{2.2}$$

Then, each array block of the distribution $B_{v_0,v_1,\ldots,v_{m-1}}$ with $v_h = \lceil \frac{N_h}{x_h} \rceil$, $0 \leq h < m-1$ is distributed to $p_{(v_0 \bmod P_0),(v_1 \bmod P_1),\ldots,(v_{m-1} \bmod P_{m-1})}$. This is inferred from the definitions and examples shown in the Scalapack pages [6]. In the common two-dimensional case, a simple example could be an 8×8 array A and a distribution of block-cyclic(2, 2) with $P_N = 4$ in a 2×2 grid. Using this example, we would see the distribution shown in Figure 2.2.

0	0	1	1	0	0	1	1
0	0	1	1	0	0	1	1
2	2	3	3	2	2	3	3
2	2	3	3	2	2	3	3
0	0	1	1	0	0	1	1
0	0	1	1	0	0	1	1
2	2	3	3	2	2	3	3
2	2	3	3	2	2	3	3

Figure 2.2: An array of size 8×8 distributed in block-cyclic(2, 2)

As we can see, the 2×2 blocks B_{ij} are distributed to processor $p_{(i \bmod 2),(j \bmod 2)}$, i.e $p_{i \times 2+j}$. Incidentally, the processors of P are distributed as shown in Figure 2.3.

p_0	p_1
p_2	p_3

Figure 2.3: A 2×2 grid of processors

2.3 Types of block-cyclic redistribution

2.3.1 Source and distribution factor

There are generally three types of block-cyclic redistribution most commonly discussed in the literature. The general case, a redistribution from block-cyclic(x) to block-cyclic(y) where x and y are arbitrary and not assumed to have any particular relation. Then, the one-divides-the-other case, wherein a redistribution from cyclic(x) to cyclic(kx) occurs, or vice-versa. Finally, there is the block-to-cyclic case, which constitutes a redistribution from cyclic($\lceil \frac{N}{P_N} \rceil$) to cyclic(1) or vice versa (cyclic-to-block), for a processor set P and array A. In every case, the first distribution is referred to as the source distribution and the second one as the destination distribution. Furthermore, the size of the blocks of the respective distributions are described as source and destination distribution factors. Henceforth, these names will be used when talking about the above cases.

2.3.2 Shape retaining vs shape changing redistributions

Furthermore, it is important to differentiate between different types of redistributions not only regarding the source and target distribution but also the "shape" of the resulting distribution.

A shape retaining redistribution is one in which all dimensions of the array remain intact and do not change. Furthermore, the processor set is also fixed, so $P = Q$. All that changes is the distributions along one or more of the dimensions, e.g. going from cyclic(2, 2) to cyclic(3, 2), or any other arbitrary change within the parameter of a normal two-dimensional block-cyclic redistribution. This change can also be one-dimensional, such as going from cyclic(1, 2) to cyclic(1, 1) which is essentially the same as a one-dimensional redistribution [7].

A shape changing redistribution, on the contrary, implies a change in the shape of either the array or the processor set. A change in the shape of a processor set means a change in terms of size or topology, such as an increase or decrease in the number of processors or a change in the structure of a processor grid[8]. For example, changing a layout in the form of 2×2 such as the one shown in Figure 2.3 to one in the form of 1×4, shown in Figure 2.4. This type of change can result in a significantly different array distribution since the distribution of the array blocks are dependent on the row and column sizes of the processor grid.

p_0	p_1	p_3	p_4

Figure 2.4: A 1×4 grid of processors

This thesis deals with redistributions that are either shape retaining, or only change the shape of the processor set. Redistributions that involve changing the shape of the array require specialized algorithms for individual cases [7].

3 Index Computation

3.1 Definition and Notation

For a general redistribution from cyclic(x) to cyclic(y), we define index computation as the following problem: For every element in the source local array of processor p_s in the source processor set P, determine the destination processor q_d of the destination processor set Q as well as its location in the destination local array. It is important to note that often, not all of this is done at once. In other words, it could be the case that initially, before communication, only the destination processor q_s is determined and the destination local address is only computed after communication. This depends on the algorithm in question. For almost all of these algorithms, the computation on the destination and source processor is almost identical. This section will focus heavily on the sending as opposed to the receiving, bearing in mind that the corresponding algorithm is essentially the same, only in reverse.

Important to note is also the concept of packing and unpacking. While this is not directly relevant to index computation, it occurs extremely close to it and is also distinctly different from communication scheduling. This means it is often mentioned and analyzed alongside indexing in the papers we cover.

Packing and unpacking describes the process of "packing" the data from memory into a message that can be sent to another processor and then, once the message arrives, taking that data and "unpacking" it back into local memory. In other words, how to efficiently transfer all the local data that needs to be sent to one processor into a message and vice versa [3]. Packing something into a message often refers to what is essentially a send buffer. This can be done either synchronously or asynchronously. Synchronous unpacking is when each destination processor waits to unpack until it has received all messages from its corresponding source processors. This is highly memory intensive, as all messages must be buffered in the meantime and also inefficient, as the processor remains idle during the communication. Because of these problems, which are exacerbated with a greater amount of processors, it is common to use asynchronous unpacking. This is when unpacking occurs immediately after a message has been received from a given processor, thereby overlapping communication time and computation and reducing processor idle time in the process [7].

3.2 Algorithm Analysis

Index computation is the most fundamental part of redistribution, without which the whole process is impossible. Hence, the very earliest papers addressing the topic of array redistribution primarily deal with this topic.

3.2.1 1994: Runtime Array Redistribution in HPF Programs

The first paper of note is called "Runtime Array Redistribution in HPF Programs" published in May of 1994[7]. This paper considers specific cases of redistribution, namely the one-divides-the-other case and the block-to-cyclic case. However it also formulates a general algorithm for cyclic(x) to cyclic(y) with arbitrary x and y. In doing so, it considers redistribution between a fixed set of processors and primarily deals with the case of one-dimensional arrays, though an extension to the multi-dimensional case is described. As the processor set is fixed, there is no need to differentiate between source and destination processor sets and the paper simply considers processor set P with size P_N. The formulas used in this paper assume all arrays are indexed starting from 1 as opposed to the usual 0. Processors are still indexed from 0.

To begin with, it is important to mention that this paper assumes any data sent from processor p_s to processor p_d is first collected in a packet, so it can be sent in one operation. This is done in order to not unnecessarily incur communication startup cost multiple times.

The idea for the block-to-cyclic case presented in the paper is very simple: knowing that cyclic(1) means that every element is distributed in a round robin fashion, each processor need only calculate the destination processor of the first element in its source local array. From there on, it is trivial to distribute the others in a round robin fashion.

The calculation of the destination processor is similarly trivial: given p_s is equal to rank of source processor and m is equal to size of block, simply calculate the formula $p_s \times m \bmod P_N$. The part $p_s \times m$ calculates the global index of the first element of the source local array of the processor, as each previous processor also owns m array elements. Then, calculating the modulo with P_N ensures we arrive at the correct destination processor since our destination distribution is cyclic(1). The receiving side is also trivial, as the data is ordered already by order of the processor indexes, making the unpacking logic simple.

The other way around, so cyclic-to-block, is slightly more complicated. To illustrate this redistribution process, consider the following example. A cyclic-to-block redistribution on an array of size 20 with 5 processors. In other words, a cyclic(1) to cyclic($\frac{20}{4}$) = cyclic(5) redistribution, i.e $m = 4$. The source and destination distribution of the array can be seen in Figure 3.1.

Similarly to earlier, we once again calculate the destination processor p_d of the first element for each source processor. We do this using the formula shown in Equation 3.2, computing the ceiling division CD (Equation 3.1) of $(p_s + 1)$ and m, then subtracting 1, which is equivalent to $\frac{p_s}{m}$. This is done because the paper uses $CD(g, m) - 1$ to transform a given global array index

0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3

0	0	0	0	0	1	1	1	1	1	2	2	2	2	2	3	3	3	3	3

Figure 3.1: An array of size 16 distributed in cyclic(1), then cyclic(4)

g to the respective processor number p on a block distribution. That equation determines the processor the gth element of the array is assigned to in the source distribution. Since this paper indexes arrays starting from 1 and we have a cyclic(1) distribution, $p_s + 1 = g$ for the given array element. Here, consider source processor $p_s = p_3$ of our example. $3 \div 5 = 0$, hence, obviously, processor p_3 sends its first element to processor p_0, which our figure confirms.

$$CD(j,k) = \frac{j+k-1}{k} \tag{3.1}$$

$$\begin{aligned} p_d = CD(p_s+1, m) &= (((p_s+1)+m-1) \div m) - 1 \\ &= ((p_s+m) \div m) - 1 \\ &= p_s \div m \end{aligned} \tag{3.2}$$

This time, we also make sure to calculate the destination local address l_d. This is calculated using the formula shown in Equation 3.4, the ceiling remainder CR (Equation 3.3) of $p_s + 1$ and m added to m. Similarly, this uses the papers formula for converting a global array index to a local array index on a block distribution: $m + CR(g, m)$. This is equivalent to $1 + p_s \bmod m$ on account of division in this paper being equivalent to floor division, i.e cutting of any decimals and array indexes starting from 1. $p_s + 1 = g$ as explained earlier. This is trivially correct as we can see from our example that processor p_3's 1st element is stored in index 4 of the array, if we start from 1 and using the formula $3 \bmod 5 + 1 = 4$, this is corroborated.

$$CR(j,k) = j - k \times CD(j,k) \tag{3.3}$$

$$\begin{aligned} l_d = m + CR(p_s+1, m) &= m + (p_s+1) - m \times (((p_s+1)+m-1) \div m) \\ &= m + (p_s+1) - m \times ((p_s+m) \div m) \\ &= m + (p_s+1) - m \times ((p_s \div m) + 1) \\ &= m + (p_s+1) - (p_s + m - p_s \bmod m) \\ &= 1 + p_s \bmod m \end{aligned} \tag{3.4}$$

From there, we have to send $\frac{m-l_d}{P_N}+1$ elements to the destination processor. The logic employed here is that the local destination address tells us how many elements out of m possible ones are already occupied, meaning $(m-l_d)$ remain. Using our example, $l_d=4$. This means the first element belonging to processor p_3 in the source distribution is placed into the 4th slot of the $5=m$ available on p_0 in the destination distribution. Then, logically, there is one slot left unfilled. Then we must divide by P_N as any given processor only sends one in P_N elements given the round robin nature of the cyclic(1) source distribution. In our example, before p_3 sends its next local block, p_0, p_1, p_2 all need to go next. And since there is only one element remaining for p_0 in the destination distribution it is obvious this has to come from p_0, not p_3. At the end we add 1 to account for the fact that l_d is calculating the address of the 1st element. Using our example, obviously p_3 still does need to send the first element which we computed the destination processor p_0 and local address 4 for.

We must then perform similar calculations for address $((m-l_d)\times P_N)+2$ the starting address of the first block not belonging to the initial destination processor. In the case of our example array and processor p_3, we can see this would be sent to p_1 at $l_d=3$. This is repeated until the end of the source local array is reached. On the receiving end, it is important to take into account that all data received from any processor must be stored with stride P_N as the source distribution is cyclic(1). For efficiency reasons, it is smart to use an asynchronous communication scheme here.

The other big case covered by this paper is the one-divides-the-other case, so cyclic(x) to cyclic(kx) and vice versa. We will only be covering the latter here (cyclic(kx) to cyclic(x)) as the two are very similar with the send and receive phases only switching around, essentially. In the send phase, we once again calculate the destination processor p_d of the first element belonging to the processor p_s using the formula $k\times p_s \bmod P_N$. This works simply because the k segments of size x per processor with a lower index than s are allocated in a round robin manner before the first element belonging to the current processor, i.e. it is the $(k\times p_s)$th segment to be allocated in a round robin manner. From there we send the first x elements to p_d, the next x to $p_d+1 \bmod P_N$ and so on, until we have sent k segments, at which point we reset the destination processor back to p_d, as the sending order repeats itself. This means we essentially only need one proper destination processor calculation per sending processor. The receiving phase is a little more complicated, and gets split up into two cases depending on k: if $k\leq P_N$ and $P_N \bmod k=0$, we follow case 1, otherwise case 2.

Case 1: We calculate the source processor p_s of first block of size x received using $p_d \div k$, where p_d is the rank of the receiving processor. After that, the next k blocks are received from the source processor p_{next} described by Equation 3.5 with i being the index of current block being received. The concept is that, since the destination distribution is cyclic(x), the P_N other processors must receive the next blocks before it is the turn of the current processor again. This receive set then repeats for each k blocks until the redistribution is complete.

$$p_{next}=(p_s+i\times\frac{P_N}{k}) \bmod P_N \quad i\in\{0,...,k-1\} \tag{3.5}$$

Case 2: Here we must explicitly calculate the source processor p_s for each block i as shown in Equation 3.6, as P_N is no longer a multiple of k, meaning the previous properties do not apply. Also, the sequence of processors does not follow a clearly identifiable pattern as it did in case 1.

$$p_s = \frac{i \times P_N + p_d}{k} \bmod P_N \tag{3.6}$$

Let us use the following two distributions of an array shown in Figure 3.2 with $N = 20$ and $P_N = 3$ as an example, the first being cyclic(4), the second cyclic(2). Meaning, $k = 2$ and $P_N \bmod k \neq 0$.

0	0	0	0	1	1	1	1	2	2	2	2	0	0	0	0	1	1	1	1

0	0	1	1	2	2	0	0	1	1	2	2	0	0	1	1	2	2	0	0

Figure 3.2: An array of size 20 distributed in cyclic(4), then cyclic(2)

Now, we will take $p_d = 2$ as an example. Using the formulas described above we calculate the first element in its local array is received from $((i \times P_N + p_s) \div k) \bmod P_N) = (0 \times 3 + 2) \div 2 = 1$. And when we look into the first element owned by 2 in the destination distribution we indeed see that in the source distribution, it was owned by 1. Moving on, we then apply the formula again: $((i \times P_N + p_s) \div k) \bmod P_N = (1 \times 3 + 2) \div 2 = 2$. It is like a counter starting from 0 counting the amount of distributions of block size x that have happened, which then must be divided by k because each block in the source distribution consists of k blocks of x.

Finally, the method for the general case of cyclic(x) to cyclic(y) where x is not a multiple of y and vice versa. Here, this paper proposes what is more or less a brute force algorithm which calculates the send and destination processor individually for each element for each source and destination local array in each processor using longer formulas for each individual calculation. This kind of runtime resolution is obviously inefficient.

In terms of extending any of this to a multi-dimensional case, the paper suggests a simple tactic. The recommended approach is simply to apply the aforementioned algorithms dimension by dimension. So, if you want to redistribute an array from (block, block) to (cyclic, cyclic), first go to to (block, cyclic). Naturally these approaches only work for shape retaining redistributions. Shape changing ones require a separate approach not discussed in this paper at all.

In summary, it can be concluded that this paper represents a good start. It presents important formulas for conversion from processor local to global indexes of distributed arrays, as well as how to convert these into calculations for destination and source processors within a given redistribution context. Especially noteworthy are the algorithms regarding conversion between cyclic(1) and block(m) distributions as well as from cyclic(x) to cyclic(kx) and vice versa, that attempt to minimize unnecessary address calculation by making use of the characteristics of these distributions.

On the other hand, it is noticeable that there is a lack of efficiency related to the more general cases. The algorithm for redistributing between arbitrary cyclic distributions is tantamount to a brute force algorithm that simply individually calculates every address for each local element of a processor, making it inefficient. The extension of the multidimensional case is also lackluster, as no new algorithms are provided that address the needs of the case and only the consecutive use of the original algorithm for one-dimensional cases is recommended.

3.2.2 1995: Automatic Generation of Efficient Array Redistribution Routines

This paper [9] addresses the same index computation problem, but by different means. By creating a new representation for regular distributions called Processor Index Tagged Family of Line Segments (PITFALLS), it presents algorithms for redistribution that apply to regular distributions in general. In other words, covering the case of cyclic(x) to cyclic(y) with no restraints on x or y. PITFALLS is designed to be a method that easily determines which processors communicate which data with each other.

The basic concept behind PITFALLS is that of the Line Segments (LS). These are represented by a pair (l, r) which stands for a block of elements within an array starting at l and ending at r. Finding the intersection of a pair of such Line Segments L_1: (l_1, r_1) and L_2: (l_2, r_2) is easily achieved using another line segment: $(max(l_1, l_2), min(r_1, r_2))$ where $max(l_1, l_2) \leq min(r_1, r_2)$, otherwise the intersection is an empty set.

This concept is then extended to a 4-tuple (l, r, s, n) called Family of Line Segments (FALLS). Similarly to the Line Segments, the first block starts at l and ends at r. s then measures the stride between l and the start of the next block and the n describes the total number of blocks. Individual members i of the FALLS are described by the following LS $(l + i \times s, r + i \times s)$. Now, to represent a given regular distribution, no more than two FALLS are necessary for each processor and in many cases, one is enough [10]. This is because the FALLS structure precisely imitates that of a block-cyclic distribution. By definition, a block-cyclic(x) distribution on P_N processors for a given processor i assigns every P_Nth block of x elements to a given processor from the given starting point $i \times x$. The amount of blocks b_n is described by Equation 3.7. So, the FALLS $(i \times x, (i+1) \times x - 1, P_N \times x, b_n)$ necessarily describes the distribution for that processor. However, in some cases, it may not represent all of it: if N mod $x \neq 0$ some processor will necessarily end up with an incomplete block at the end. This then needs to be described with another FALLS. Hence, the necessity for two FALLS sometimes arises.

$$b_n = \begin{cases} \lceil \frac{N}{P_N \times x} \rceil, & \text{if } N \bmod (P_N \times x) > (i+1) \times x \\ \lfloor \frac{N}{P_N \times x} \rfloor, & \text{otherwise} \end{cases} \tag{3.7}$$

An example of a case in which one FALLS suffices is a simple cyclic(2) distribution across 3 processors on an array of size 16. Given processor p_1, this results in the FALLS (2, 3, 6, 3). Visually, this is represented by Figure 3.3. The underlined elements are the ones distributed onto p_1. Each line represents a block.

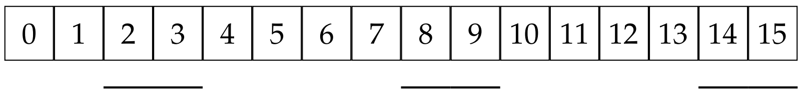

Figure 3.3: A visual representation of a FALLS (2, 3, 6, 3), each line representing a block

Then, an example in which one FALLS does not suffice is given by a cyclic(3) distribution on 2 processors for an array of size 17. Given processor p_1 we must use the following two FALLS to represent the distribution: (3, 5, 6, 2) and (15, 16, 0, 1). These FALLS can be seen in Figure 3.4, the first one in black above the second one in red.

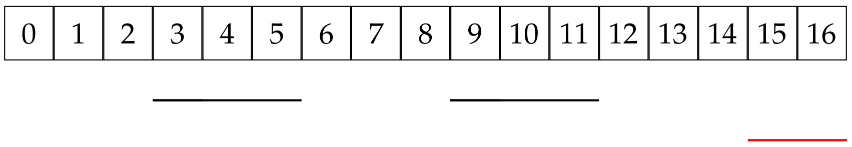

Figure 3.4: A visual representation of a FALLS (3, 5, 6, 2), followed by another FALLS (15, 16, 0, 1) each line representing a block

It now becomes interesting to develop an efficient intersection algorithm for FALLS, as it translates to an efficient way to determine which data must be sent from one processor to the other to get from a source to a target distribution. The best idea involves testing if 2 given members i_1 and i_2 of two different FALLS F_1, (l_1, r_1, s_1, n_1), and F_2, (l_2, r_2, s_2, n_2) intersect. This can be done by simply testing if the 2 conditions described in equations 3.8 and 3.9 are met, which are simply extensions of the intersection conditions for Line Segments, taking into account the number of the respective members and the strides s1 and s2 of the two FALLS. Member i_1 of F_1 is given by $(l_1 + i_1 \times s_1, r_1 + i_1 \times s_1)$ and member i_2 of F_2 is given by $(l_2 + i_2 \times s_2, r_2 + i_2 \times s_2)$. According to the intersection criteria for Line Segments defined earlier, these two arbitrary members intersect only if $max(l_1 + i_1 \times s_1, l_2 + i_2 \times s_2) \leq min(r_1 + i_1 \times s_1, r_2 + i_2 \times s_2)$. This is equivalent to Equation 3.8 ∧ Equation 3.9. Equation 3.8 ensure that i_2 ends after i_1 begins and Equation 3.8 ensures that i_2 begins before i_1 ends. Self-evidently, this is equivalent to intersection knowing that $l_i \leq r_i$.

$$i_2 \geq i_1 \times \frac{s_1}{s_2} + \frac{l_1 - r_2}{s_2} \tag{3.8}$$

$$i_2 \leq i_1 \times \frac{s_1}{s_2} + \frac{r_1 - l_2}{s_2} \tag{3.9}$$

The intersection itself can then be calculated using the algorithm for Line Segments. These two conditions also do not need to be evaluated for all possible combinations of pairs of the two FALLS: it is enough to do so for one intersection period, defined as the Least Common Multiple (LCM) of s_1 and s_2. This is because within 2 FALLS that intersect, the intersecting members have a periodical relationship.

Given our earlier FALLS (2, 3, 6, 3), which we will call F_1 and a new FALLS F_2 (0, 2, 4, 3) on the same array of size 16, Figure 3.5 will demonstrate the intersection. The first FALLS in red is F_1, the second in blue is F_2 and the third in black is the intersection of the two $F_{1\cap 2}$.

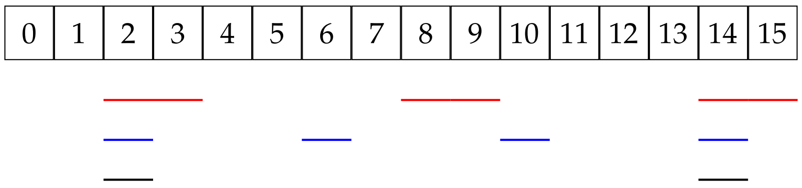

Figure 3.5: A visual representation of two FALLS and their intersection

This representation is then further extended to PITFALLS, a six-tuple (l, r, s, n, d, p). It stands for a set of evenly spaced FALLS across a set of p processors. The spacing between the successive l's of different processors is d. The ith processors FALLS then takes the following form: $(l + i \times d, r + i \times d, s, n)$. Given a source and a target distribution, then, it is possible to check if any FALLS members intersect by iterating through their intersection periods and checking their member pairs for the 2 conditions above. If there is an intersection, we can construct the respective LS and FALLS immediately. If we do this for all PITFALLS of the source and target distributions, so usually just one but up to 3 depending on the distribution we have completed the index computation.

For the multi-dimensional case, the extension works similarly to the one used in the previous paper. Simply compute the FALLS and their intersection for each dimension and then combine the results to figure out the final results for sending and receiving sets.

Using the algorithms constructed from the PITFALLS representation, the paper shows a significant speed-up (2-3x) over brute-force runtime resolution methods, such as the one presented for the general cyclic(x) to cyclic(y) redistribution problem in the previous paper. When it comes to the multi-dimensional case, there is also a large difference. Unlike the previous papers suggestion which results in a non-linear increase of redistribution time, PITFALLS allows for only a linear increase when applying the technique to each dimension in succession.

All-in-all it can be observed that this paper improves upon the results of the work of Thakur et al. [7] (subsection 3.2.1) when it comes to performance in the general case, if not the one-divides-the-other or the block-to-cyclic case. Furthermore, it also works for different processor subsets, and so has the added advantage of versatility.

Nevertheless, there are some weaknesses to be found in the PITFALLS algorithm as well. Noteworthy is that performance depends heavily on the number of processors involved in the redistribution, as you may have to compute the intersection between every single pair.

This means that when we have large processor sets, the performance of PITFALLS declines drastically. This lines up with the runtime complexity of the algorithm: $O(P_N \times \frac{|x-y|}{gcd(x,y)})$ [3]. Furthermore, the paper does not take explicitly into account anything regarding shape changing redistributions, although this is a trivial extension with regards to processor structure.

3.2.3 1996: Optimizations for efficient Array Redistribution on Distributed Memory Multicomputers

This paper [11] is the follow-up work to the one covered in the previous section, again covering the FALLS algorithm, with several extensions:

1. Generation of local addresses for the elements of distributed arrays as opposed to global ones
2. Integration of the library with MPI
3. Exploitation of memory locality during packing/unpacking
4. Use of a communication scheduling scheme

For the purposes of this section particularly points one and three are of interest as they provide benefits not having to do with communication scheduling and potentially change the algorithm itself.

The paper demonstrates a simple formula for generating the local addresses to save any single element of a member of a local FALLS, given by Equation 3.10. In this case, l and r have their typical meanings as in FALLS/PITFALLS, j is the global address of the element being saved and i is the number of the member of the FALLS the element is a part of. This generation of address assures that all elements are saved consecutively and by simply switching the i, r and l depending on whether we are considering the source or destination FALLS we can adjust to be correct for both the sending and receiving processor.

$$l(address) = i \times (r - l + 1) + j \bmod (r - l + 1) \tag{3.10}$$

The paper makes use of a figure to represent this, which we will replicate using our original FALLS F_1 = (2, 3, 6, 3) over an array with $N = 16$ from section 3.2.2, as shown in Figure 3.6.

Improving memory locality is also important as it allows for better performance since modern computers are typically optimized for accessing memory in contiguous blocks. Here, this can be achieved by generating the minimum amount of different FALLS possible for a given intersection between two FALLS. Since all elements of each generated FALLS are packed into a continuous buffer for transport purposes, a small set of FALLS means a greater amount of consecutive accesses (as opposed to strided), which is, per definition, better memory locality. The paper does not specify further on how to achieve this algorithmically, however.

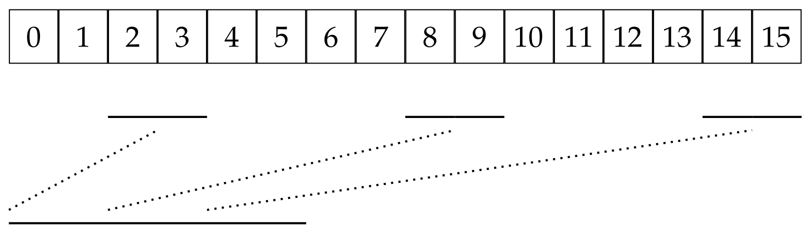

Figure 3.6: A visual representation of the translation into a local address

All-in-all, the actual indexing algorithm does not significantly change. However, the improvements made with regards to memory access and local memory address computation guarantees increased efficiency, nonetheless.

3.2.4 1996: Efficient Algorithms for Array Redistribution

This paper[12] serves as an extension and an update to [7], which is covered in subsection 3.2.1. Primarily, it adds two new methods to perform a general redistribution from cyclic(x) to cyclic(y) that are more efficient than the one discussed in the original paper, which requires the explicit calculation of the destination location for every single array element. Instead, here, the methods make use of the highly efficient technique developed for the cases that y is a multiple of x or vice versa. This technique is often called the multi-phase method by other works.

The first method is called the GCD method, as it makes use of the greatest common divisor. The idea behind this method is that two consecutive redistributions using the specialized technique for the one-divides-the-other can be better than a direct brute force redistribution. It is important to take into acccount, however, that this constitutes a trade-off: the improved indexing time must offset the greater communication time resulting from the two redistributions. This is generally the case for larger arrays where the indexing time dominates.

Given this, the tactic is to redistribute from cyclic(x) to cyclic(g), where $g = gcd(x, y)$ and then from there to cyclic(y). However, doing it this way comes with a major disadvantage: the block size in the intermediate distribution is smaller than both pre and post-redistribution. Smaller blocks require more calculations as we can see looking back at the algorithm description of the one-divides-the-other-case in the 1994 paper. In the worst case scenario, $gcd(x, y)$ is equal to 1 and we revert back to the brute force case where we must calculate everything.

An alternative is to use the LCM method instead, which makes use of the least common multiple instead of the greatest common denominator, guaranteeing the intermediate distribution uses a larger block size than both source and target distribution. This way, we bypass the aforementioned problem.

Due to this, the LCM method also outperforms the GCD method in all cases, unsurprisingly. In comparison to the general, brute-force method, it does better for large array sizes where the indexing-communication trade-off becomes worthwhile.

This kind of redistribution, which is done in multiple phases shows demonstrable improvements over the original in some cases, although it is unclear whether it can outperform a technique like PITFALLS even for large array sizes.

3.2.5 1997: Fast Runtime Data Redistribution - Indexing

This paper[13] presents a redistribution algorithm implemented for the ScaLAPACK library and the High-Performance Fortran (HPF) language. Henceforth it is also referred to as the "ScaLAPACK method" or "Pyrill's algorithm". This method deals with the general redistribution problem, allowing for different source and destination processor sets. It can be easily extended to multidimensional arrays and processor grids in turn, a process also described in the paper.

The indexing algorithm introduced in this paper is very similar to PITFALLS, as it also uses intervals to describe the overlap between the elements of a processor in the source distribution and a processor in the target distribution. It also notes the same periodicity as PITFALLS, the least common multiple of the strides between the respective blocks of the two processors. The only noticeable difference is the lack of use of a compact notation like FALLS. The list of intervals of overlap is not summarized into anything and is simply used as is.

This paper also has a segment on communication in which it makes use of a caterpillar algorithm, a popular communication algorithm for this type of work which we will revisit in subsection 4.2.4. This is noteworthy here only in so far as it could affect any comparisons with other algorithms here that focus more exclusively on indexing.

This paper is primarily discussed here because it is often used as a point of comparison and is cited often by others.

3.2.6 1998: A Basic-Cycle Calculation Technique For Efficient Dynamic Data Redistribution

This paper [3] presents a more efficient method for the general redistribution problem of cyclic(x) to cyclic(y) redistribution, the Basic-Cycle Calculation (BCC) technique. In doing so, it limits itself to primarily the one-dimensional case and makes the assumption that the source and target processor sets are the same. The primary idea of the paper is to develop methods for computing the source and destination processors of certain array elements in a so-called "basic-cycle". These can then be used to easily compute the source and destination processors of the other array elements, making for an efficient redistribution.

A basic-cycle is defined as the least common multiple of the two redistribution parameters divided by their greatest common divisor. So, for a general redistribution from cyclic(x) to cyclic(y), the basic-cycle BC would be calculated using Equation 3.11.

$$BC(x,y) = \frac{lcm(x,y)}{gcd(x,y)} \tag{3.11}$$

The concept behind the BCC technique is that the communication pattern of the first basic-cycle of a given processors Source Local Array (SLA) is the same as any other. However, this is only the case if $gcd(x,y) = 1$ and otherwise requires transforming the to be distributed array in such a way that this is effectively the case. This works by transforming array A with size N into array B with size $N \div gcd(x,y)$. Each entry of B then consists of a sub-array of $gcd(x,y)$ consecutive elements of A. Each entry c of B is given by Equation 3.12 .

$$B[c] = \{A[(c-1) \times gcd(x,y) + 1, \ldots, A[(c) \times gcd(x,y)]\} \tag{3.12}$$

Within this new array B, the communication once again repeats every basic-cycle, e.g. all elements in the sub-array $B[c]$ are sent to the same processor as all elements of the sub-array $B[c + BC(x,y)]$.

To simplify, from here on, the paper assumes $gcd(x,y) = 1$. Otherwise, we must use $x \div gcd(x,y)$ and $y \div gcd(x,y)$ as the source and destination distribution factor instead, which results in a Greatest Common Divisor (GCD) of 1.

Explaining the BCC technique requires the introduction of two new terms: the Source Distribution Pattern Position (SDPP) and the Destination Distribution Pattern Position (DDPP) (of an element, respectively). They are defined by Equation 3.13 and Equation 3.14.

$$SDPP(A[c]) = \left(\left\lceil \frac{c}{gcd(x,y)} \right\rceil - 1 \right) \bmod \frac{P_N \times x}{gcd(x,y)} \tag{3.13}$$

$$DDPP(A[c]) = \left(\left\lceil \frac{c}{gcd(x,y)} \right\rceil - 1 \right) \bmod \frac{P_N \times t}{gcd(x,y)} \tag{3.14}$$

These terms define the position of a given array element over one iteration through the processors, so $y \times P_N$ elements for the destination distribution. Given an array with $N = 10$ and a redistribution of cyclic(2) to cyclic(1) with $P_N = 2$, we would see the results shown in Table 3.1.

Knowing this, we can move into explaining the send phase of the technique. Given a source processor p_s and its source local array, we want to determine the destination processor p_d of every element within the first basic-cycle, so SLA_s[0:BC-1]. The paper presents a formula that computes p_d for any given element with local array index c, given by Equation 3.15.

$$p_d = \begin{cases} P_N - 1, & \text{if } \alpha = 0 \\ \left\lfloor \frac{\alpha - 1}{y} \right\rfloor, & \text{otherwise} \end{cases} \tag{3.15}$$

index	0	1	2	3	4	5	6	7	8	9
SLA_0	0	1	/	/	2	3	/	/	4	5
SLA_1	/	/	0	1	/	/	2	3	/	/
SDPP	0	1	2	3	0	1	2	3	0	1
DLA_0	0	/	1	/	2	/	3	/	4	/
DLA_1	/	0	/	1	/	2	/	3	/	4
DDPP	0	1	0	1	0	1	0	1	0	1

Table 3.1: Table that describes SLA's, DLA's, SDPP and DDPP for a given problem

$$\alpha = \left(x \times \left(\left\lfloor \frac{c-1}{x} \right\rfloor \times (P_N - 1) + p_s + c\right)\right) \bmod (P_N \times y) \tag{3.16}$$

One problem, however, is that a basic-cycle can be large depending on the values of x and y. In that case, we have to do a lot of calculations to determine all the destination processors, which is impractical. To take care of this problem, the paper suggests making use of the consecutive nature of elements in blocks. That is to say, the fact that in a cyclic(x) distribution, the x elements are always taken consecutively from the global array. This allows us to simply calculate the DDPP of the first element of each block in a given source local array which then allows for trivially determining the destination processors of the following $x - 1$ elements. $\lfloor DDPP \div y \rfloor$ determines the destination processor and the DDPP of each block element $c \in$ [0, x - 1] is simply $DDPP[0] + c \bmod P_N$.

The DDPP of the first element u of a respective block with the number $v \in \{1, \ldots, \frac{BC}{x}\}$ can be easily calculated using Equation 3.17 and Equation 3.18.

$$DDPP[SLA_s(u)] = ((v-1) \times P_N + p_s \times x) \bmod (P_N \times y) \tag{3.17}$$

$$u = (v-1) \times x + 1 \tag{3.18}$$

The calculations for the receiving phase are essentially identical, except SDPP is used instead of DDPP and the x and y parameters switch roles.

The above algorithm results in an indexing cost of $O\left(\frac{lcm(x,y)}{gcd(x,y)}\right)$. This is the primary achievement of this paper compared to the previous ones. The development of an indexing algorithm, the runtime of which is independent of number of processors and array size, is hugely important in facilitating scalable index computation. This work achieves this by recognizing the repetitive pattern in the spacing of the data in the send sets of a given processor relative to the original distribution. This same principle is used in several other papers in this field as well, and is one of the quintessential aspects of effective data redistribution algorithms. Its importance can be demonstrated by the marked improvements in performance the BCC technique shows in almost all cases compared to the algorithms previously discussed in this section.

3.2.7 1998: Efficient Methods for Multidimensional Array Redistribution

This paper[5] is based on the BCC method discussed in the previous subsection 3.2.6. Using it as a basis, two techniques are presented specifically for multi-dimensional array redistribution. These are, on one hand, the Basic-Block Calculation (BBC) method, and on the other, the Complete-Dimension Calculation (CDC) technique.

The paper posits that, much like in the one-dimensional case, it is only necessary to construct the communication sets (i.e source and destination processor) for a certain section of the multi-dimensional array, and that the rest repeats itself. In this case, that unit is one basic-block. In the case of a general multi-dimensional redistribution from cyclic($x_0, x_1, ..., x_{m-1}$) to cyclic($y_0, y_1, ..., y_{m-1}$) that basic-block consists of the basic-cycles in each dimension, so $BC_0, BC_1, ..., BC_{m-1}$ for a total size of $BC_0 \times BC_1 \times ... \times BC_{m-1}$. Each basic-cycle BC_i is then defined as in the original paper (Equation 3.11), except that one must use x_i and y_i as the distribution parameters.

To explain the algorithm used in this paper we must first slightly extend the notation of SLA used up until now. $SLA_{i,l}$ will be defined as the set of array elements of processor i in the first row of the lth dimension, in the source distribution. The DLA is extended analogically.

In other words, assuming SLA_i is the following 4×4 two-dimensional array, $SLA_{i,0}$ consists of the elements coloured in blue, and $SLA_{i,1}$ of the elements coloured in red. The element (0, 0) that is in both arrays is coloured purple, as can be seen in Figure 3.7.

(0, 0)	(0, 1)	(0, 2)	(0,3)
(1, 0)	(1, 1)	(1, 2)	(1, 3)
(2, 0)	(2, 1)	(2, 2)	(2, 3)
(3, 0)	(3, 1)	(3, 2)	(3, 3)

Figure 3.7: An array of size 4×4 colored to depict $SLA_{i,0}$ and $SLA_{i,1}$

The paper presents two vital new lemmas that build the basis for the redistribution algorithm to come, which we must first discuss.

The first defines a translation between redistribution in one-dimension and redistribution in m dimensions. This lemma starts by observing the array element $SLA_{i,k}[r_k]$, that is to say, the r_kth element of the first row of the kth dimension of the source local array of processor p_i. Now, let us say the destination processor of this element is $p_{0,...,0,j_k,0,...,0}$ where $0 \leq j_k \leq P_k$. Then, given the array element $SLA_i[r_0, r_1, ..., r_{m-1}]$ its destination processor would be, in turn, $p_{j_0,j_1,...,j_{m-1}}$. In other words, the destination processor of a given array element in a multi-dimensional array can be determined if one knows the destination processors of each index in the first row of its respective dimension. Knowledge of the destination processors of $SLA_{i,k}[r_k]$ $\forall k \in \{0, ..., m-1\}$ can thus be used to determine the destination processor of $SLA_i[r_0, r_1, ..., r_{m-1}]$.

Under the consideration that each basic-cycle (of a given dimension) always has the same communication set if $gcd(x_i, y_i) = 1, i, y \in \{0, ..., m-1\}$, or if the array is adjusted according to the algorithm described in subsection 3.2.6, the second lemma is derived. To calculate the destination processor of any given element in a multi-dimensional array it is enough to calculate the destination processors for the first basic-block, i.e. $SLA_i[1 : BC_0, 1 : BC_1, ..., 1 : BC_{m-1}]$. And in turn, to calculate these, we simply need to calculate the destination processors for each of the individual basic-cycles, as lemma 1 implies. This means that for any multi-dimensional array, Equation 3.19 holds true, with $0 \leq k_i \leq \lfloor N_i \div (lcm(x_i, y_i) \times P_i) \rfloor$. As a reminder, N_i and P_i are the size of the respective ith dimensions of the array and processor sets respectively. In other words, k_i is defined so that the index does not exceed the size of the arrays dimension.

$$\begin{aligned} SLA_i[r_0, r_1, ..., r_{m-1}] &= SLA_i[r_0 + k_0 \times BC_0, r_1, ..., r_{m-1}] \\ &= SLA_i[r_0, r_1 + k_1 \times BC_1, ..., r_{m-1}] \\ &= ... \\ &= SLA_i[r_0, r_1, ..., r_{m-1} + k_{m-1} \times BC_{m-1}] \\ &= SLA_i[r_0 + k_0 \times BC_0, r_1 + k_1 \times BC_1, ..., r_{m-1} + k_{m-1} \times BC_{m-1}] \end{aligned} \tag{3.19}$$

This also further extends to the case where any arbitrary subset of the dimensions changes by the respective factors of k and BC. Given this knowledge, we can now think of an efficient algorithm.

At first glance, it seems simple: we just need to compute all destination processors of array elements in $SLA_i[1 : BC_0, 1 : BC_1, ..., 1 : BC_{m-1}]$. However, in case of a large basic-block size this can take a long time even with the simple formula to enumerate the processor described in chapter 2, as we need to calculate each destination processor. As such, we instead make use of send tables. A table is created for each dimension of the array, gathering the indexes of the respective $SLA_{i,l}[1 : BC_l]$ array elements that have the same destination processor into one entry. Then, for any given processor p_i we wish to send to, we can simply look up the entries in the tables. This can be done simply by reversing the formula described in chapter 2 which converts the dimension wise representation of a processors index to the absolute one using the appropriate *mod* operations. This concludes the basic-block method.

When it comes to the complete-dimension-calculation method, things change only slightly. Instead of calculating only the destination processors of the first basic-cycles of each dimension, we also calculate the destination processors of the first row of every dimension (all else being 0), so: $SLA_{i,l}[0 : L_l]$, with $l \in \{0, ..., m-1\}$ and $L_l = \frac{N_l}{P_l}$, the size of the array and the processor set in the dimension l. This information is then used to construct the send tables, similarly to the basic-block calculation technique. The idea behind this is that we trade in higher indexing cost due to having to calculate destination processors for more elements of source local arrays for more packing/unpacking information that does not require any deciphering via modulo operations like in the case of the previous method.

In terms of performance relative to one another, the paper also gives a clear answer by determining the exact conditions under which the CDC technique outperforms the BBC technique. What is being compared here is the total computation time, so not only indexing where the basic-block method would be trivially superior, but also packing and unpacking, i.e: $T_{comp} = T_{indexing} + T_{(un)packing}$. The conclusion at which the paper arrives is summarized by the Equation 3.20 in combination with Equation 3.21.

$$T_{comp}(CDC) < T_{comp}(BBC) \iff O(L_0 + L_1 + ... + L_{m-1}) < O(\frac{L_{m-1}}{BC_{m-1}} \times L_{m-2} \times ... \times L_0) \tag{3.20}$$

$$L_l = \frac{N_l}{P_l} \tag{3.21}$$

Using this equation we can then judge which method is superior depending on the situation.

This papers description of an extension to the multi-dimensional covers one of the two aspects missing from the original Basic-Cycle Calculation method paper to make it a more universally applicable indexing technique. The other one is discussed by another paper extending the original technique, namely:

3.2.8 2000: A Generalized Basic-Cycle Calculation Method for Efficient Array Redistribution

As the name suggests, this paper [14] is another extension of the BCC technique described in [3] (subsection 3.2.6). The generalized version of the basic-cycle method covers the case of cyclic(x) to cyclic(y) redistribution from P_N to Q_N processors. In other words, the primary difference compared to the original version is the consideration of different source and target processor sets, P and Q. In doing so, a new version of a basic-cycle involving these new variables is introduced and the same principles of communication pattern repetition are applied to minimize index computation overhead.

This papers generalized basic-cycle changes for source and destination distribution. For the source distribution, it is defined by Equation 3.22 . For the destination distribution, by Equation 3.23

$$GBC(P) = \frac{lcm(x \times P_N, y \times Q_N)}{gcd(x, y) \times P_N} \tag{3.22}$$

$$GBC(Q) = \frac{lcm(x \times P_N, y \times Q_N)}{gcd(x,y) \times Q_N} \tag{3.23}$$

The basic logic used for the generalized basic cycle calculation technique is the same as for the normal BCC technique: calculating the source and destination processors for the first generalized basic cycle is enough to determine the destination processors for the rest. As is the case with the original technique, this requires $gcd(x,y)$ to be equal to 1 or otherwise a reorganisation of the array as discussed in the earlier paper. Similarly, the consecutive nature of blocks is again used to only calculate the destination processor of the first element of each block explicitly, after which the rest can be inferred the rest via addition and modulo.

Where things change are the formulas used to calculate the destination processors. For a given index c which is the first in a given block of the source local array of a processor p_s, we use Equation 3.24 to determine the corresponding global array index.

$$sgindex_s(c) = c \times P_N + i \times x \tag{3.24}$$

Then, using this global array index we determine the destination processor q_d with Equation 3.25.

$$q_d = \left\lfloor \frac{sgindex_s(c)}{y} \right\rfloor \bmod Q_N \tag{3.25}$$

From here, we must now calculate how many elements must still be sent to q_d. We will call this value r and calculate it using Equation 3.26.

$$r = \left(\left\lfloor \frac{sgindex_s(c)}{y} \right\rfloor + 1 \right) \times y - sgindex_s(c) \tag{3.26}$$

With knowledge of x, y, r and q_d we can now easily distribute the rest of the block. We do this by sending the next r elements to q_d, or as many as we can until the block ends. Having done that, we must now send segments of y elements at a time until we reach the end of the block. The first block of y elements, $SLA_i[r : r + y]$ would be sent to $(q_{d+1}) \bmod Q_N$, the second block, $SLA_i[r + y : r + (2y)]$ to $(q_{d+2}) \bmod Q_N$ and so on. This process must be performed for every block in a generalized basic-cycle, so, a total of $GBC(P) \div x$ times.

The generalized form of the basic cycle calculation method yields similar results to the original, applied to the broader scope of different source and target processor sets. It outperforms methods from previous papers such as the ScaLAPACK method and PITFALLS and offers an efficient way to deal with indexing as well as packing and unpacking.

On the other hand, it does nothing to address the topic of efficient communication, much like its predecessor, even though the paper itself addresses that this is the most time consuming aspect of redistribution in the examples given. Ensuring efficient communication is the other half of efficient data redistribution which will be addressed in chapter 4.

3.3 Performance Comparison

In this section, the topic of discussion will be the performance of the algorithms we discussed earlier. Almost all of the papers include some kind of performance comparison to previous algorithms, which allows for a fairly grounded view of how they compare. We will start by looking at the general case, as in a lot of instances the results of the comparisons there translate to the other two cases.

3.3.1 The general case

In terms of performance with regard to the general case we see a marked progression over the course of the papers discussed. The baseline for the discussion will be a "brute-force" runtime resolution method such as the one from [7] briefly discussed in subsection 3.2.1, wherein the destination processor is calculated from scratch for every element of the array.

[9] (subsection 3.2.2), which presents the PITFALLS representation, as well as the associated redistribution algorithm, also includes some performance tests. Performance comparisons were made using 3 different source/target array distribution pairs, 2 different processor source/target distribution pairs per array distribution and 2 different array sizes, 128×128 and 512×512. PITFALLS came out ahead consistently, the difference being larger the larger the size of the array. The improvements ranged from a 1.7x to a 3.8x speed-up.

Next, [13] (subsection 3.2.5) shows a further improvement. Here it is important to note that communication scheduling is being used that is not mentioned at all in PITFALLS, so whether these results are actually because of the indexing algorithm is unclear. Nevertheless it is important to mention because this algorithm in turn is subject to more comparisons down the line. The comparisons made similarly used 128×128 and 512×512 sized arrays as well as a variety of source/target array and processor distributions. The results were speed-ups ranging from 1.16 to 3.5 times, growing with larger block and array sizes.

[3] (subsection 3.2.6) also measures performance across a variety of cases, specifically comparing its own BCC method to both PITFALLS and the multi-phase method. The multi-phase method in this case refers to what is presented in [12] (subsection 3.2.4) which was thus far not mentioned in this section as result of not containing performance comparisons to other algorithms (excluding its previous iteration).

For the sake of this comparison, we will observe only the indexing and packing/unpacking times (i.e computation cost) of the respective algorithms, which are recorded separately, as anything to do with communication is not relevant to this section. Incidentally, it does not change the result significantly.

Upon analyzing the indexing costs of the respective methods, the primary realization is the differences in dependencies: the multi-phase method is dependent on array size and number of processors (inversely). PITFALLS is dependent on number of processors and distribution factors while the basic-cycle calculation technique is dependent only on the parameters of the source and target distribution x and y. The packing costs are all almost identical, differing only in so far as that the multi-phase method needs to pack and unpack twice as much due to the two-stage redistribution process.

The performance tests utilized in this paper compare the algorithms using various different block-cyclic redistributions on a one-dimensional array. For each one, they perform two types of tests: one where the amount of processors P_N is constant and the array size N is gradually increased, and vice versa. The amount of processors varies between 10 and 72, where as the size of the array size is between 360.000 and 1.800.000. In other words, the focus is on the testing of performance regarding larger arrays, which is beneficial to the basic-cycle calculation method as it is the only one of the three the execution time of which does not increase with the array size. Nevertheless, this can be considered reasonable, as redistribution is often times relevant precisely to larger arrays.

When testing using an $N = 1.800.000$ and performing a cyclic(5) to cyclic(8) redistribution and gradually increasing P_N in steps of 10, we notice that, when it comes to indexing, the basic-cycle calculation method outperforms the multi-phase method by a factor of about 10 to 20x, with that factor shrinking the higher the number of processors. When it comes to PITFALLS, the basic-cycle calculation method is also ahead, though the speed-up is only roughly 1.3x in the case where $P_N = 10$. This then grows to a speed-up of roughly 4.3x when the P_N is increased to 72. In other words, when the number of processors is low, PITFALLS and the basic-cycle calculation method are not massively apart in terms of performance. That being said, this gap also depends on the distribution parameters used, as a large difference in x and y bodes poorly for the performance of PITFALLS, while a small LCM to GCD ratio is good for the BCC method. In general, both methods benefit from a large GCD. In the case of a cyclic(300) to cyclic(200) redistribution we see the minimum speed-up of the BCC technique over PITFALLS rise to roughly 2.3x, and for cyclic(60) to cyclic(3) it even gets to about 3.5x.

On the other hand, when testing with $P_N = 72$ and N incremented from 340.000 to 1.800.000 in steps of 340.000, we see that the PITFALLS and the BCC method retain roughly the same performance throughout, with a ratio equivalent to the above case where we also had $P_N = 72$. On the contrary, the multi-phase method performs worse and worse, and the initial 6x speed-up BCC had over it increases to 12x.

In total it can be said that this papers performance analysis demonstrates that, for the general case, when it comes to indexing large arrays, the hierarchy of the algorithms is as follows: BCC > PITFALLS > Multi-phase. However, something also worth noting is that in practically all the examples, the overall redistribution time was dominated by packing/unpacking costs and communication time. This means that, depending on the potential for improvement in these two areas, indexing algorithms may not be very important to the total redistribution optimization process.

3.3.2 The one-divides-the-other case

This case is only explicitly discussed in the paper [7] (subsection 3.2.1) and its extension [12] (subsection 3.2.4), though obviously any algorithm that can carry about the general case is also capable of this. There are little explicit performance comparisons regarding this case, though it is notable that the distribution parameters being multiples of one another is not particularly relevant to the runtime of any of the general algorithms. This means that the relationships from the previous discussion should hold. The only question is where [7]'s algorithm fits in, which specializes in the case. However, this is also answerable, as both [13] (subsection 3.2.5) and [3] (subsection 3.2.6) address this topic, albeit with somewhat different results.

[13] (subsection 3.2.5) mentions the multi-phase method only in passing and does not provide a detailed analysis. However, it does note that in the one-divides-the-other case, the multi-phase method slightly outperforms its own methods with regards to packing, which, as explained earlier, dominates indexing itself in terms of total cost.

The 1998 paper [3] (subsection 3.2.6) has an actual test case measuring the performance of the one-step multi-phase method compared to PITFALLS and the BCC technique. In this case, a redistribution of cyclic(10) to cyclic(500) is performed and the results are measured over the same ranges of N and P_N described earlier. In these cases, the multi-phase method is still outperformed by PITFALLS and the BCC technique for almost all cases. However, in two cases it beats out PITFALLS with a speed-up of roughly 1.25x. That is, in the case where the N is $\in$ {360.000, 720.000} and $P_N = 72$. In other words, the smaller the N to P_N ratio is, the better the multi-phase method performs against PITFALLS. So, for small arrays and large processor sets, there will be cases where multi-phase can win out over PITFALLS, though not against the BCC technique which is at minimum about 3x faster even for the best case for multi-phase ($N = 360.000$ and $P_N = 72$).

3.3.3 The block-to-cyclic case

Regarding this last case there are not any specific tests that deal with it. However, precisely because this case is so specific we can say that the importance is small: it would almost always be handled by an algorithm that deals with either of the above cases as opposed to anything specific. Interestingly, however, depending on the size of the array, this could be a troublesome case for something like the BBC technique because the LCM to GCD ratio is very lopsided: the LCM is always equal to m and the GCD is always 1. Therefore, if we have a large array, we are in rough shape. This is similarly true for PITFALLS due to the large gap in size between the distribution factors. Thus, as long as the N to P_N ratio is smaller than m, it is probably an edge case where the multi-phase method can perform better than either of the other methods.

4 Communication

4.1 Definition and Notation

Communication refers to the process of sending messages buffered with the elements computed by the indexing algorithm from the source processors to the destination processors receiving them. This means that each source processor creates a message for each destination processor it must send to, and fills said message with the array data (i.e packing). Then the destination processor receives said message and stores the data locally according to the indexes it computed (i.e unpacking).

The question now, is how to structure this process so that it is maximally efficient. Setting aside packing and unpacking, this means optimizing the sending and receiving process. In principle, when it comes to such communication, there are two possible approaches: synchronous and asynchronous communication algorithms. Each have their own advantages and disadvantages, which we will go over.

4.1.1 Synchronous Communication

Communication is usually called synchronous, or otherwise blocking, when, after the source processor sends the message, it must wait until the corresponding destination processor has received it to continue with performing other tasks, such as sending the next message. The same goes for the destination processor, which must wait until the message has arrived after posting its receive operation. Communication algorithms operating using this principle of communication have destination processors post one receive at a time, and source processors post a corresponding send, explicitly pairing them up in each iteration of communication. It is worth noting, however, that such algorithms as the one above, that are described, in general, as synchronous, often use asynchronous receives. This is not particularly relevant, however, as each "round" of communication is synchronous nonetheless [1].

The advantage of these types of algorithms is that they are very forgiving in terms of memory consumption. Both source and destination processors can reuse buffers used to store messages after each round of communication. On the other hand, it is inefficient when it comes to performance, due to the need to wait on the completion of the specific pairwise exchange before continuing with other work. This means that, if two or more source processors are paired up with the same destination processor, one may have to wait a significant amount of time until it can continue sending to other processors. This is called node contention.

4.1.2 Asynchronous Communication

Communication is called asynchronous or otherwise non-blocking, when the operations for sending and receiving messages return immediately without waiting for completion of the communication. The rest of the communication then happens in the background while the processors are freed up to continue their work, which in this case usually means sending and receiving more messages. Alternatively, it can also refer to index computation, in case we overlap it with communication as opposed to completing it all beforehand. The completion of the individual communications can be checked on later. Communication algorithms that make heavy use of this concept may simply have destination processors post all of their receives at once, asynchronously, and then wait for the messages from the source processors, knowing they can immediately receive them [1].

These kinds of algorithms are the opposite of the synchronous ones in terms of their advantages and disadvantages. We must use separate buffers for the messages, assuming we post multiple asynchronous receives before checking if others are completed, meaning memory requirements are large, but we can continue working without waiting on the completion of individual communications, which guarantees efficiency. Explanations for a possible implementation of both a synchronous and asynchronous algorithm are provided in [1], which we discuss in subsection 4.2.3.

Generally, if memory requirements are not an issue, asynchronous communication will be used. However, if we are sending a large amount of data or otherwise do not have a large amount of memory available, we must resort to synchronous communication. Furthermore, there may be some cases where asynchronous receives are not be available, such as was the case at the time for the BLACS message passing library used for ScaLAPACK [13]. In that case, we optimize the communication so that we lose as little performance as possible compared to the asynchronous version. As the end result of this optimization process is similar for all papers discussed, it shall be explained here.

The goal of scheduling is to minimize node contention such that we do not have hot spots in communication. The optimal way to do this is to divide the communication into steps denoted by k_i for the ith communication step, indexed from 0. In each step, each sending processor, if possible, is paired up with a distinct receiving processor it sends to, that is to say, no two sending processors are paired up with the same receiving processors and vice versa. Then, after all communication steps are completed, all data must have been sent. Such a schedule can be represented as a matrix, the exact size depending on the case: each column representing a source processor and each row a communication step. An entry of x in column i and row j then means that the processor p_{i-1} sends a message to processor q_x in communication step k_{j-1}. As an example, Table 4.1 shows such a communication matrix, otherwise also called process send schedule. Here we assume $P = Q$.

Here we can see that there are 3 sending and receiving processors, as the length of each row is three and there are three communication steps. We also note that each communication step consists of a permutation of all destination processors: indeed, this is a necessity if we

k / p	$\mathbf{p_0}$	$\mathbf{p_1}$	$\mathbf{p_2}$
$\mathbf{k_0}$	1	2	0
$\mathbf{k_1}$	2	0	1
$\mathbf{k_2}$	0	1	2

Table 4.1: Example of a communication matrix

want a minimum number of contention-free steps while the source processor set is equal to or larger than the destination processor set, i.e $P_N \geq Q_N$, in a redistribution involving all-to-all communication. Finally, the entries of each row i consist of a permutation of the processors p_i must send to. At first glance this appears simple, but if $P \neq Q$ and/or we do not have all-to-all communication, constructing a contention free schedule can be significantly more complex.

The following section will revolve largely around the different algorithms for obtaining such a schedule. Some algorithms are indeed more efficient than others or cover a broader case. For instance, many of the algorithms presented below apply only to the one-divides-the-other case for a single processor set P. Furthermore, while all of the algorithms presented will result in a contention-free schedule, some contention-free schedules are superior to others, as we will discuss. We also consider other communication based optimizations outside of scheduling, primarily the idea of a multi-phase redistribution that was already presented for indexing purposes in the previous chapter.

4.2 Algorithm Analysis

4.2.1 1994: An Approach to Communication Efficient Data Redistribution

The 1994 paper "An Approach to Communication Efficient Data Redistribution"[15] continues to explore the idea of a multi-phase redistribution, this time with respect to its advantages in terms of communication. In previous papers such as [12], which we covered in subsection 3.2.4, it was assumed as a given that this approach would result in an increase in communication time. However, this paper challenges that idea: depending on how large the message set-up time is compared to the actual data transfer, it can be more efficient in terms of communication time to perform a multi-phase redistribution. This paper develops a model for analysing the communication costs of a given redistribution as well as a method of communication efficient redistribution, making use of the model.

This paper makes heavy use of a tensor product representation to create its analysis model and efficient methods. Given two matrices A and B, with A as an $m \times n$ matrix and B as a $p \times q$ matrix, it is defined by Equation 4.1.

$$A \otimes B = \begin{bmatrix} a_{0,0}B & \dots & a_{0,n-1}B \\ \vdots & \ddots & \vdots \\ a_{m-1,0}B & \dots & a_{m-1,n-1}B \end{bmatrix} \tag{4.1}$$

Specifically relevant are the tensor products of vector bases, e_i^m, a column vector of length m with a 1 at position i and 0 everywhere else. Tensor products of two such bases are called a tensor basis, demonstrated in Equation 4.2.

$$e_i^m \otimes e_j^n = e_{in+j}^{mn} \tag{4.2}$$

In this paper, they key is the ability to factorize a given vector basis e_i^M. This means representing it as a tensor product of smaller vector bases such that the above conditions apply. For example, it would be possible to factorize e_9^{16} in the alternative ways shown in Equation 4.3.

$$\begin{aligned} e_9^{16} &= e_1^2 \otimes e_1^8 \\ e_9^{16} &= e_2^4 \otimes e_1^4 \\ e_9^{16} &= e_1^2 \otimes e_0^2 \otimes e_1^4 \end{aligned} \tag{4.3}$$

This notation can be used to express regular distributions. A distribution cyclic(x) on P_N processors for an array A of size N can be represented by factorizing a vector basis e_i^N, where i is the index of a given element in array A and $M = N \div P_N$. The exact way to do this is given by Equation 4.4.

$$e_i^N = e_{i \div (P_N \times x)}^{M \div B} \otimes p_{i \div x \bmod P_N}^{P_N} \otimes e_{i \bmod x}^{x} \tag{4.4}$$

The index in the second vector basis corresponds to the rank s of the processor p_s i is assigned to, because of which p is used instead of e in the notation to signify the vector basis that determines the processor index, so, the processor basis. Meanwhile, the vector basis identified by e is called the data basis. Then, the tensor product of the first and third vector base produces the local index of the element i on p_s. Such a factorized tensor basis in which the vector basis determining the processor index is identified with the p is called a distribution basis.

This representation thus neatly encompasses all relevant information: the global array index, the local array index and the assigned processor. This notation is also easily extended to the multi-dimensional case by simply taking the tensor product of the distribution representations in each dimension.

Using this notation, it is now possible to develop a cost model that then allows for an optimal multi-phase redistribution. With respect to the communication cost of a given redistribution, several factors are of importance: the start-up time, t_s, which consists of the packing and unpacking process we have discussed. This is used as a constant in this paper, which is not necessarily accurate depending on the communication in question, since, if one processor has to send significantly more data than another, its packing and unpacking time could be considerably greater. Of primary consideration is the transfer time t_e, the time it takes for data to be transmitted, which is dependent on the network bandwidth as well as the path length, though the latter is negligible. This time is assumed to scale linearly with the length of the message, such that the communication cost for a message of length n is $t_s + t_e \times n$.

The last aspect to be considered is network contention: communication can be slowed down significantly when several sending processors communicate simultaneously with the same destination processor. As such, it is logical to schedule communication into several steps, such that each step involves a permutation of destination processors: as no processor sends to the same destination in each step, we avoid node contention, assuming a sufficiently large topology. This is equivalent to the common idea presented in section 4.1.

Knowing this, we can deduce that two factors are of great importance in the communication cost of a redistribution: the amount of communication steps and the cost of each step. If it is possible to determine these factors, it is also possible to develop a coherent model for communication cost. This is easily possible using two qualities that can be applied to the distribution basis of the source and destination distribution: the distribution base difference and union.

The distribution base difference is defined as $Q(\beta_1, \beta_2)$ and is given by Equation 4.5, with β_1 as the source distribution basis and β_2 as the destination distribution basis. β_1 and β_2 can be factorized such that $\beta_1 = \sigma_{j_t}^{n_t} \otimes \ldots \otimes \sigma_{j_0}^{n_0}$ and $\beta_2 = \delta_{k_t}^{n_t} \otimes \ldots \otimes \delta_{k_0}^{n_0}$. Being able to factorize the distribution bases of the source and destination distributions in this manner means they are compatible, a pre-requisite for redistributing between them as far as this paper is concerned.

$$Q(\beta_1, \beta_2) = \prod_l n_l \quad \text{with } 0 \le l \le t \quad \text{s.t. } (\sigma_{j_l}^{n_l} = e_{j_l}^{n_l}) \wedge (\delta_{j_l}^{n_l} = p_{j_l}^{n_l}) \tag{4.5}$$

The distribution base union $U(\beta_1, \beta_2)$, then, is defined by Equation 4.6.

$$U(\beta_1, \beta_2) = \prod_l n_l \quad \text{with } 0 \le l \le t \quad \text{s.t. } (\sigma_{j_l}^{n_l} = e_{j_l}^{n_l}) \vee (\delta_{j_l}^{n_l} = p_{j_l}^{n_l}) \tag{4.6}$$

This is, in turn, equivalent to $P_N \times Q(\beta_1, \beta_2)$.

These two qualities correspond to important aspects of the communication process. The distribution base difference $Q(\beta_1, \beta_2)$ is equivalent to the maximum number of messages a processor must send during redistribution. The fraction of the total data that constitutes the largest amount of data to be sent in a single message is determined by the distribution base union. This means that the size of the largest message to be sent is $\frac{N}{U(\beta_s, \beta_d)}$. Furthermore, knowing that each processor sends at most $Q(\beta_1, \beta_2)$ messages is vital, as this implies that we can construct exactly that many permutations to perfectly represent the communication, as is proven in Theorem 3.1 on page 369 in [15]. In turn, this also means that this is the minimum amount of communication steps we can undertake. All of this results in the formula for the communication cost $C(\beta_s, \beta_d)$ of a redistribution from β_s to β_d given by Equation 4.7, in which case the constant k is the fixed number of communication steps per permutation. This value depends on the underlying architecture and will not be further considered.

$$C(\beta_s, \beta_d) = Q(\beta_s, \beta_d) \times k \times (t_s + \frac{N \div P_N}{Q(\beta_s, \beta_d)} \times t_e) \tag{4.7}$$

Immediately noticeable is the fact that the cost of a given communication step is equivalent to the cost of the single largest message: this observation stems from the fact that all messages are sent in parallel: hence the cost of the largest one dominates the step. This is a characteristic that generally applies to communication steps, regardless of the scheduling algorithm. This same formula can also be applied to each dimension of a given multi-dimensional array if need be, in order to facilitate the optimization of communication for arrays that are not one-dimensional.

Considering this cost model and the representation of distributions via distribution bases, it is possible to determine the optimal intermediate distributions for an optimal multi-phase redistribution. Considering the source distribution β_s and the destination distribution β_d, the condition of compatibility means both can be factorized into equally many vector bases, each corresponding one being of the same size. This means that redistribution is essentially just a mapping of the respective data bases to processor bases and vice versa. These mappings can be performed either all at once or in multiple steps. How many steps exactly, depends on the factorization performed on the respective distributions. $Q(\beta_s, \beta_d) = r_0 \times \ldots \times r_{g-1} \implies g - 1$ intermediate distributions, one for every vector basis in the factorization β_s that is mapped into a processor basis in the factorization of β_d. Each intermediate distribution then performs another mapping. Pursuing this method for a given redistribution leads to the cost structure given by Equation 4.8.

$$C(\beta_s, \beta_d) = \sum_{i=0}^{g-1} (r_i \times t_s + \frac{N}{P_N} \times t_e) \tag{4.8}$$

Comparing this to the single-phase redistribution, we can see the trade-offs demonstrated in Table 4.2.

	single-phase	multi-phase
start-up($\times t_s$)	$\prod_{i=0}^{g-1} r_i$	$\sum_{i=0}^{g-1} r_i$
transmission($\times t_e$)	$\frac{N}{P_N}$	$g \times \frac{N}{P_N}$

Table 4.2: The trade-offs between single-phase and multi-phase redistribution

In other words, if $((\prod_{i=0}^{g-1} r_i) - (\sum_{i=0}^{g-1} r_i)) > (g-1) \times \frac{N}{P_N}$, the multi-phase redistribution delivers better performance than the single-phase variation. Of course, it is also possible to choose a lesser amount of intermediate distributions for a multi-phase approach. The precise partition of $Q(\beta_s, \beta_d)$ necessary for the best result is in of itself an optimization problem worth discussing. A subset of this problem is presented in the paper "Optimal phase barrier synchronization in k-ary n-cube wormhole-routed systems using multirendezvous primitives" [16], for the case that $Q(\beta_s, \beta_d) = r^p, r \in \mathbb{Z}$. Comparing this with the single-phase approach would then yield the optimal approach under those constraints.

It also noteworthy that the distribution basis notation can also be used for index computation, which is also described in the paper and can thus be used for the whole redistribution process. This will not be covered further here, as it falls outside the scope of this section.

Overall, the paper presents an interesting approach to efficient communication that falls somewhat outside of most of the literature. Instead of presenting an optimized scheduling algorithm for the communication in a single-phase redistribution as the following papers largely do, it presents a redistribution strategy that can result in better performance. In practice, whether it is worth pursuing this strategy depends on factors like the specific redistribution parameters and the size of the array, which we cover in the performance analysis section, section 4.3.

4.2.2 1995: Multiphase Array Redistribution - Modeling and Evaluation

This paper [17] acts as an extension and evaluation of the previously discussed [15] (subsection 4.2.1). It translates the results of the previous paper, based heavily in the tensor notation, into a more standard notation, conforming roughly to the one this thesis establishes in chapter 2. Furthermore, it goes much more in-depth in terms of the actual scheduling process that takes place in a given communication step, which was only a small footnote in the previous paper. Finally, it provides an analysis of the multi-phase redistribution techniques in terms of performance which the original paper does not do. The communication cost model used mirrors that of the original paper, so the focus of this analysis will be on the scheduling.

The focus of this paper is on the one-divides-the-other case regarding a single set of P_N processors, as the intermediate distributions of a given multi-phase distribution always follow this pattern, hence scheduling this case is what is relevant to the topic. Specifically, the paper devises a scheduling algorithm to optimally schedule, in terms of both number of communication steps and contention, a cyclic(kx) to cyclic(x) redistribution.

For a cyclic(kx) to cyclic(x) redistribution, there are two cases to cover: one in which the communication is all-to-all, and another where it is not. The case in which all-to-all communication is present is one in which $k > P_N$, assuming that $P_N \times kx \leq N$. This assumption is a non-issue because the array can simply be padded accordingly. In this case scheduling communication is trivial and the paper does not focus on it. This is because it is simple: if every processor communicates with every other processor, a contention-free schedule can be obtained with the following algorithm 1.

Algorithm 1 Processor Communication Scheduling

```
1: for $i = 0$ to $P_N - 1$ do
2:     for $j = 0$ to $P_N - 1$ do
3:         Schedule message of processor $p_i$ to processor $p_{(j+i) \bmod P_N}$ in communication step $j$
4:     end for
5: end for
```

By simply spacing out the start of the iteration through all processors by one, contention free scheduling in P_N steps is guaranteed. Note that this algorithm no longer works in the same way if we have separate processor sets P and Q. If this is the case and $Q_N > P_N$ we either have to accept contention or accept that some processors cannot communicate in some communication steps as all the destination processors are already paired up with source processors. This then means that we must schedule $P_N - Q_N$ processors after the first Q_N.

This paper covers the less trivial second case: all-to-many communication which occurs when $P_N > k$. This in turn is split into two further sub-cases: first is the case in which the source processors each communicate with distinct first destination processors. The second case occurs when the first destination processors are not distinct. The first destination processors refer to the destination processors source processors must send the very first block of the global array they are assigned in the source distribution. In other words, for a given processor p_j, $(j \times k) \bmod P_N = (p_j \times k) \bmod P_N$.

The first sub-case is once again trivial: in the case where each first destination processor is distinct, the respective next destination processors in order are necessarily also distinct. This means simply scheduling messages in order of the blocks in the local array and their respective destination processor will produce a contention-free schedule. In the other sub-case, in step i each processor sends its ith local block. Consider, for example, a redistribution from cyclic(4) to cyclic(2) with $P_N = 5$ and array size $N = 20$. Figure 4.1 shows the array distribution pre and post-redistribution.

0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4

0	0	1	1	2	2	3	3	4	4	0	0	1	1	2	2	3	3	4	4

Figure 4.1: An array of size 20 distributed across 5 processors, first in cyclic(4), then in cyclic(2)

The first destination processors of processors p_0 through p_4 are as follows, in order: p_0, p_2, p_4, p_1, p_3. This then allows for the following communication table, shown in Table 4.3 which, when comparing it side by side to the two distributions shown earlier, is clearly correct and contention-free.

k / p	$\mathbf{p_0}$	$\mathbf{p_1}$	$\mathbf{p_2}$	$\mathbf{p_3}$	$\mathbf{p_4}$
$\mathbf{k_0}$	0	2	4	1	3
$\mathbf{k_1}$	1	3	5	2	4

Table 4.3: Communication matrix for the problem described above

The second sub-case is significantly more complex. Essentially, the goal is to structure the communication in such a way that we mimic the first sub-case: we want to manually alter the destination processor order that source processors with the same first destination processor send to, such that the new first destination processors of every source processor are different. In order to do this, we must first identify which source processors have the same first destination processor. To do this, we must partition the processors into $\frac{lcm(P_N,k)}{k}$ groups, with each processor p_j being sorted into group i with, i given by Equation 4.9.

$$i = p_j \bmod \frac{lcm(P_N,k)}{k} \tag{4.9}$$

All processors within the same group have the same first destination processor p_d, as $(p_j \times k) \bmod P_N$ determines the first destination processor for processor p_j and thus $\frac{lcm(P_N,k))}{k}$ limits the amount of times p_j can be incremented until we return to the same value. The size of each group is given by $gcd(P_N,k)$, because $P_N = \frac{lcm(P_N,k)}{k} \times gcd(P_N,k)$. What's noticeable here is that this means there are only $\frac{lcm(P_N,k)}{k}$ possible first destination processors, all of which are multiples of $gcd(P_N,k)$, according to Lemma 4.2. This means that there are $gcd(P_N,k) - 1$ non-first destination processors following any p_d, all of which every processor in group i obviously has to send to. Hence, a simple approach is to change the first destination processor of processor p_h with $h = p_j + \frac{k \times lcm(P_N,k)}{k}$ with $0 \leq j < \frac{lcm(P_N,k)}{k}$ from $(j \times k) \bmod P_N$ to $(p_j \times k) \bmod P_N + k$. Doing this for every group guarantees unique first destination processors for all source processors and therefore a contention-free schedule when following the method of sub-case 1. Though we must remember to loop back around to the destination processors of the earlier blocks we skipped over when determining the new first destination processors.

Let us consider the example redistribution of cyclic(3) to cyclic(1) with $P_N = 6$ and $N = 18$. In this redistribution, $k = 3$. Since $P_N > k$ we do not have all-to-all communication. Immediately, it is noticeable that the first destination processors are not all distinct. They form two groups: group one consists of processors p_0 , p_2 and p_4 with the first destination processor p_0 and group two consists of processors p_1, p_3 and p_5 with the first destination processor p_3. This means we are in the second sub-case. This means we must iterate through each group i, $0 \leq i < \frac{lcm(6,3)}{3}$, such that the kth element of the group has its first destination processor altered to p_g with $g = (i \times 3) \bmod 6 + k$. This then produces the communication matrix seen in Table 4.4, which is clearly contention free and also complete.

k / p	**p0**	**p1**	**p2**	**p3**	**p4**	**p5**
$\mathbf{k_0}$	0	3	1	4	2	5
$\mathbf{k_1}$	1	4	2	5	3	0
$\mathbf{k_2}$	2	5	3	0	4	1

Table 4.4: Communication matrix for the problem described above using the algorithm from the multi-phase method

Having covered the scheduling method, we now dive into the cost model presented, which is the same as in the previous paper, except translated from tensor notation and applied specifically to the cyclic(kx) to cyclic(x) case. Just as it was before, the cost of a single communication cost is dominated by that of the largest message, which is $t_s + n \times t_e$ if that size is n. Then, the amount of permutations, previously described by the distribution base difference Q, is now simply $min(k, P_N)$. This is because, using the schedule, each processor can send a message in each step and that amount of messages is either k in the case $k < P_N$ or P_N in the case $k > P_N$ as we discussed earlier. Then, the size of each message must be $\frac{N}{P_N \times min(P_N,k)}$. This is simply because we know each processor sends $min(P_N, k)$ messages and there are P_N processors, hence each message size is the one described above. Here the paper implicitly assumes the array is padded accordingly, preferably such that its size is some multiple of $P_N \times kx$. All of this results in the redistribution cost C given by Equation 4.10.

$$C = min(P_N,k) \times t_s + \frac{N}{P_N} \times t_e \tag{4.10}$$

For a g-phase redistribution as described in the previous paper, which consists of a redistribution from cyclic(kx) to cyclic($\frac{k}{k_1}x$) and so on to cyclic($\frac{k}{k_1 \times \ldots \times k_{g-1}}x$) and finally, to cyclic(x), the paper presents the following cost C given by Equation 4.11.

$$C = \sum_{i=0}^{g-1} (k_i \times t_s + \frac{N}{P_N} \times t_e) \tag{4.11}$$

Once again, the structure of the equation is identical to the one from the previous paper.

This paper now goes one step further and dives into a discussion of the two-phase approach for the general case redistribution, cyclic(x) to cyclic(y). Specifically, the optimal intermediate distribution for this case is determined. Naturally, the distribution should have a distribution factor that either divides or can be divided by both x and y, the source and destination distribution factors. This is so the previously discussed models and schedules can be applied, which work only for the one-divides-the-other case.

And so, for a cyclic(x) to cyclic(f) to cyclic(y) redistribution where f is divided by x and y, Equation 4.12 is constructed to determine the total cost of such a redistribution.

$$C = \frac{f}{x} \times t_s + \frac{N}{P_N} \times t_e + \frac{f}{y} \times t_s + \frac{N}{P_N} \times t_e = f \times (\frac{1}{x} + \frac{1}{y}) \times t_s + 2\frac{N}{P_N} \times t_e \tag{4.12}$$

This equation assumes $f > x \wedge f > y$. We can determine the best f within those parameters by minimizing $f \times (\frac{1}{x} + \frac{1}{y})$, which in turn means simply picking the smallest possible f such that $f \mod x = 0 \wedge f \mod y = 0$. This is the least common multiple of x and y. In the other case, where $f < x \wedge f < y$ and thus f divides x and y, we must instead minimize $\frac{1}{f} \times (x + y)$. This means choosing the largest possible f such that $x \mod f = 0 \wedge y \mod f = 0$. In others words, the greatest common divisor of x and y. These being the two most optimal intermediate distribution choices, equally good, aligns with the results discussed in the indexing section. Then, given the advantages the least common multiple has over the greatest common divisor when it comes to indexing, the most optimal choice for f should be $lcm(x, y)$.

All-in-all, this paper does not provide a new method, as it is merely an extension and evaluation of an approach already discussed in the previous section, however, the clarity it provides with regard to the previous paper is very helpful nonetheless. In terms of the scheduling algorithm presented, its primary weakness lies within the fact that it applies only to redistributions within the same processor set, which is a big limitation. Furthermore, in the exact form explained here, the algorithm only applies to the sending part of a cyclic(kx) to cyclic(x) redistribution or the receiving part of a cyclic(k) to cyclic(kx) redistribution. The next paper discussed presents an algorithm for the other half.

4.2.3 1996: Redistribution of Block-cyclic Data Distributions using MPI

This paper [1], much like the previous case, deals with scheduling redistribution primarily for the one-divides-the-other case, specifically cyclic(x) to cyclic(kx). It also does not consider different processor sets, simply using a uniform set P for both the source and destination distribution. This work presents both an asynchronous and a synchronous communication algorithm in the form of MPI code: then, later, it develops an algorithm to schedule the synchronous communication in order to make it more efficient.

Both of these algorithms work much the way they were described in the first section of this chapter. Here, we will cover them in a bit more detail. This paper makes the same observation several others in the indexing section also did, that is, that there is a repeating communication pattern in the redistribution: in what this paper calls a superblock, that is, every $P_N \times k$ blocks or every $P_N \times k \times x$ elements, communication patterns repeat themselves. Hence, for both algorithms, we have only k communication steps: an MPI derived vector data type is used to send all blocks occupying the same position within the superblock at once. Then the P_N processors need only send k blocks each. Some perhaps less, if the last block is incomplete. Such cases must be handled separately, but as this is a trivial aspect, neither this paper nor most of the others discussed explicitly deal with it.

The synchronized algorithm has each processor, within a single communication step, first post a non-blocking, wildcard receive. This means that it is not determined from which other processor a message will be received and it is essentially done on a first-come, first-serve basis. The receive is non-blocking so as to allow the process to pack the necessary data and perform a send call in parallel to waiting on the incoming data. Next, the processors use a blocking operation to send out messages to the respective destination processors, iterating through the local blocks belonging to them from front to back, so in the *n*th step k_n, each process sends its *n*th block. Here, it is important that they also prepend the global index of the block being sent to the beginning of the message: otherwise it would be impossible to tell where the block must be placed locally, given that a wildcard receive is used as opposed to any specific one and we thus have no information about what data we are receiving. This changes when scheduling is introduced, which makes it definitive from which processor is received in which communication. Next, `MPI_wait` is called to complete the receive operation that was started earlier. Finally, the received data is unpacked. Then all processors move into the next step and repeat this process until communication is complete. The problem this method faces is that some processors may have to wait a long time for their send operation to complete, which in turn also means they must wait a long time for their receive. Additionally, some processes may receive their data later in general, due to other processes only sending to them in the very last steps.

Take, for example, a redistribution from cyclic(1) to cyclic(3) with $P_N = 4$ and $N = 12$. Using the above algorithm without any sort of additional scheduling would result in the communication matrix seen in Table 4.5.

k / p	$\mathbf{p_0}$	$\mathbf{p_1}$	$\mathbf{p_2}$	$\mathbf{p_3}$
$\mathbf{k_0}$	0	0	0	1
$\mathbf{k_1}$	1	1	2	2
$\mathbf{k_2}$	2	3	3	3

Table 4.5: Communication matrix for the problem described above using the synchronized algorithm without scheduling

Here we immediately see the problems described earlier. In the first communication step, processor p_0 is clearly a hotspot: it must receive data both from itself and 2 other processors before the communication step can conclude. At least one of the processors sending to processor 0 will have to wait a significant amount of time. We also see that processor p_3 receives all of its data in the very last communication step, when processors p_0 and p_1 have already received all of theirs. The schedule produced by this algorithms basic form is very sub-optimal.

The asynchronous version is very similar in nature, also using the same non-blocking receives and blocking send operations. The difference is that, in the unsynchronized version, all *k* receives are posted consecutively, and only after they have all been posted, are the *k* send operations performed, after which `MPI_waitany` is called *k* times and the received data is unpacked. This sidesteps the problems described above, as we can be sure all send operations complete very quickly, since the corresponding receives are posted before any waiting is done for the send operations. On the other hand, posting k receives at once, all of which require a separate buffer into which to receive data, is very memory expensive.

In order to allow the synchronous algorithm to compete with the non-synchronous one in terms of performance, contention must be reduced. One way to do this is by randomizing the block-sending order for each process involved: this reduces communication hotspots significantly. This can best be explained using the earlier example. Assuming we randomize the sending order with a random number generator, we might obtain a schedule like the one shown in Table 4.6.

k / p	$\mathbf{p_0}$	$\mathbf{p_1}$	$\mathbf{p_2}$	$\mathbf{p_3}$
$\mathbf{k_0}$	1	3	2	1
$\mathbf{k_1}$	0	1	0	3
$\mathbf{k_2}$	2	0	3	2

Table 4.6: Communication matrix for the problem described above using the synchronized algorithm with random sending order

Clearly, this is an improvement on the previous case: only 2 processors send to the same processor in each step. No processor has to wait until the last step to begin receiving data either. Nonetheless, this is an imperfect tactic, as it fails to eliminate node contention in most cases. As such, this paper returns to devising an optimal communication schedule in the form of a $k \times P_N$ matrix, called a communication matrix or a process send schedule here. This is done according to criteria very similar to those presented at the beginning of this chapter, which are formulated here as follows:

1. The rows of the process send schedule are permutations of the process ranks, so, permutations of the numbers $0, 1, \dots, P_N - 1$
2. The ith column of the send schedule is a permutation of the processes to which process p_i must send data

If these two criteria are fulfilled in the schedule, it is possible for the synchronous communication method to rival the asynchronous one in terms of performance. Determining the solution that fulfills these conditions numerically is somewhat complex. First Equation 4.13 is constructed for the global block B from both the source and destination distribution, given the respective local block indexes b and d, as well as the source and destination processor ranks p_i and p_j, taking into account that this paper is strictly focused on the case of a cyclic(x) to cyclic(kx) redistribution.

$$B = P_N \times b + p_i = k(P_N \times d + p_j) + t \tag{4.13}$$

This is the primary equation this paper works with. Here, $0 \leq t < k$. t is representative of which place inside of the k slots of size x in a block the local block sent from p_i is placed into. This equation can be simplified, because we know communication repeats itself each superblock: and since a superblock is merely $k \times P_N$ elements, we know the first superblock is completely covered by elements within local block $d = 0$ of the destination distribution. Hence, Equation 4.13 can be rewritten as Equation 4.14.

$$B = P_N \times b + p_i = k \times p_j + t \tag{4.14}$$

The goal is to use this equation to determine in which step k_n each processor sends a given local block. An intuitive thought may be to simply send in step k_n, from each processor, the block that is placed into slot $k_n = n$ in the corresponding destination processors block. So, essentially, for a given value of k_n setting $t = k_n$ and, for each processor p_i, determining the corresponding block b (or B) and destination processor p_j that fulfills this condition. Formally, this is given by Equation 4.15.

$$B = P_N \times b + p_i = k \times p_j + k_n \tag{4.15}$$

This carries with it a fatal flaw, however: it does not necessarily produce a unique solution. We can easily envision a redistribution where processor p_i sends a different block to a different destination processor that happens to be slotted into the same fixed index k_n in the larger block of the destination distribution. Take, for example, a redistribution from cyclic(1) to cyclic(4) with $P_N = 6$ and $N = 48$. Processor p_1 would send its first block with global block index 1 to processor p_0, which would be placed into the second slot in the new, larger block. Then, processor p_1's third block would be redistributed to processor p_d with $d = (6 \times 2 + 1) \div 4 = 3$. Here, too, it would be placed into the second slot of the block, with index 1, since $(6 \times 2 + 1) \bmod 4 = 1$.

Because we are left with (potentially) more than one destination processor to which a given source processor can send in any given step the problem may have been narrowed down somewhat, but it is not yet solved.

Nonetheless, this attempt at a solution is on the correct path: if the redistribution problem is minimized in such a way that the solutions become unique while still being able to expand back into the original case, we will have a viable method. This can be achieved simply by factoring out the greatest common denominator g of P_N and k, such that $k = g \times k'$ and $P_N = g \times P'_N$. Then the following formulas shown in Equation 4.16 are derived for the other variables.

$$\begin{aligned} b = k' \times \beta + b' \qquad p_j = P'_N \times \beta + p'_{j'} \\ p_i = g \times p'_{i'} + \alpha \qquad t = g \times t' + \alpha \end{aligned} \tag{4.16}$$

When inserted into the original equation, we receive a reduced version of the initial equation, perfectly suited for deriving unique solutions, which is shown below. The reason factoring out $gcd(k, P_N)$ results in unique solutions is because relatively prime values for k and P_N guarantee that if a processor p_i sends to a processor $p'_{i'}$ a block into slot i, it sends blocks into slot i only to processor $p'_{i'}$. This uniqueness theorem is proven in the appendix of the paper, A.1.

$$B' = P'_N \times b' + p'_{i'} = k' \times p'_{j'} + t' \tag{4.17}$$

Solving this equation by setting $t' = k_{n'}$ gives an optimal schedule. This is because the earlier problem has been eliminated: since k and P_N have been reduced down to k' and P'_N, there are no longer multiple destination processor, block number combinations that fulfil the equation $B' = P'_N \times b' + p'_{i'} = k' \times p'_{j'} + k_{n'}$. Therefore, per $(k_{n'}, p'_{i'})$ pair, we have a unique solution to the equation, $(p'_{j'}, b')$, as, obviously, there must be at least one solution assuming we have a valid global index B. Furthermore, we know that for a given k_n, for each p_i, the result contains a different p_j, as each global block is only assigned to one processor. The same argument applies for a given k_n and each p_i. Therefore, we know this method fulfils the two criteria for the communication matrix explained at the beginning of the section. The proofs for these attributes can be found in the appendix of the paper (uniqueness, existence and permutation theorem).

The solutions for this equation are given by the formulas shown in Equation 4.18.

$$\begin{aligned} q'_{j'} &= \lambda \times x + P'_N \times z \\ b' &= -\lambda \times y + k' \times z \end{aligned} \tag{4.18}$$

Here, x and y are the solutions for $g = x \times k + y \times P_N$ that can be found with the extended Euclid algorithm. $\lambda = p'_{i'} - k_{n'}$ and $z = \lceil \lambda \times y \div k' \rceil = \lceil \lambda \times x \div P'_N \rceil$. As such, we now know that $p'_{i'}$ sends local block b' to destination processor $p'_{j'}$ in step $k_{n'}$ and that this is optimal. It is worth noting that this calculation doubles as an index computation, even though that is not its primary purpose.

The send schedule corresponding to the equations solution can be acquired by constructing the respective matrix of size $k' \times P'_N$. This is done by starting at position (0, 0) in the matrix, which is filled by the value 0 and moving along diagonally, incrementing the value along the way. When moving off the edge or bottom of the matrix, we simply wrap-around. It is worth noting that this manner of constructing the schedule means the entries of the schedule are global block indexes B. However, these can easily be converted to the corresponding processors by simply dividing by k'. Taking the example from earlier where $k = 4$, $P_N = 6$, we would have $k' = 2$ and $P'_N = 3$, hence, the following, obviously contention free global block/processor send schedules seen in Table 4.7 and Table 4.8.

The last step is to expand the solution we obtain via these equations from the reduced equation where g is factored out to the original one. This is fairly easy: we merely need to convert each of the $k' \times P'_N$ entries we obtain by calculating the results of the reduced

k / p	$\mathbf{p_0}$	$\mathbf{p_1}$	$\mathbf{p_2}$
$\mathbf{k_0}$	0	4	2
$\mathbf{k_1}$	3	1	5

Table 4.7: Block send schedule for the problem described above, using the synchronized algorithm with scheduling in the reduced system

k / p	$\mathbf{p_0}$	$\mathbf{p_1}$	$\mathbf{p_2}$
$\mathbf{k_0}$	0	2	1
$\mathbf{k_1}$	1	0	2

Table 4.8: Process send schedule for the problem described above, using the synchronized algorithm with scheduling in the reduced system

equation for all k' and $p'_{i'}$ into a $g \times g$ block, called a permutation block. This is done by making use of the equations for b and p_i that were presented earlier. By inserting them into the basic equation $B = P_N \times b + p_i$ which we recognize as the first part of the primary equation, Equation 4.13 and summarizing, Equation 4.19 is obtained.

$$B = g \times B' + P_N \times k' \times \beta + \alpha \tag{4.19}$$

This is immediately recognizable as an equation that allows us to construct the global block values for the full communication matrix out of the ones calculated for the reduced system. B', which is a direct consequence of the chosen calculated values $p'_{j'}/b'$ (as well as $p'_{i'}/k'$), is what differentiates the $g \times g$ blocks from each other. Inside of the block, the differentiating factors are the variables β and α which take the values of 0 to $g - 1$. Each entry in the $g \times g$ block has a different pair of (α, β), which adhere to the following rules.

1. α = column number of permutation block: if $g = 4$, all entries in second column of the block use $\alpha = 2$
2. Each row of a permutation block must have a permutation of all β values: if $g = 3$, the second row must have an entry where $\beta = 0$, $\beta = 1$ and $\beta = 2$

The first rule guarantees that the second criteria of the process send schedule is upheld: since we know $p_i = g \times p'_{i'} + \alpha$, having different α values inside of the same column would violate it. The second rule guarantees the first criteria is upheld: if two identical β values were allowed within the same row, two different processors would be sending to the same destination, as $p_j = P'_N \times \beta + p'_{j'}$. Having generated all of the blocks, assembling the schedule is a simple matter of placing them adjacent to each other in the correct order. Using $\alpha = p_i \bmod g$ and $\beta = (\alpha - y + g) \bmod g$, with $y = k_n \bmod g$, for example, the above criteria are fulfilled.

Let us return to the example from earlier, where $k = 4$ and $P_N = 6$ and, therefore, $gcd(k, P_N) = 2$. Using α and β as described above, we get the block/process send schedules shown in

Table 4.9 and Table 4.10.

k / p	p_0	p_1	p_2	p_3	p_4	p_5
k_0	0	13	8	21	4	17
k_1	12	1	20	9	16	5
k_2	6	19	2	15	10	23
k_3	18	7	14	3	22	11

Table 4.9: Block send schedule for the problem described above, using the synchronized algorithm with scheduling in the full system

k / p	p_0	p_1	p_2	p_3	p_4	p_5
k_0	0	3	2	5	1	4
k_1	3	0	5	2	4	1
k_2	1	4	0	3	2	5
k_3	4	1	3	0	5	2

Table 4.10: Process send schedule for the problem described above, using the synchronized algorithm with scheduling in the full system

This total procedure is similar to the one described in the paper that was covered in subsection 4.2.2, [17]. Much like in that paper, the processor set is first reduced and then the results computed for that reduced set are used to deduce the destination processors of the expanded set accordingly. The main difference is that this paper considers the redistribution from cyclic(k) to cyclic(kx) while the previously discussed paper covers primarily cyclic(kx) to cyclic(x).

In total, this paper presents an important intermediate step. While it only offers a solution to general case redistribution in the form of a multi-phase method, much like earlier iterations and does not account for cases in which the source and destination processor sets are different, it is this paper that later becomes the basis for more complete algorithms that can solve for the general case directly. Additionally, actually providing the MPI code for the synchronized and unsynchronized versions of the redistribution clarifies the process significantly.

4.2.4 1997: Fast Runtime Data Redistribution - Communication

The 1997 paper "Fast Runtime Block Cyclic Data Redistribution on Multiprocessors" [13], which we had previously discussed with regards to indexing in subsection 3.2.5, also provides an algorithm for scheduling communication. The so-called caterpillar algorithm is what is implemented in ScaLAPACK and referenced in several other highly relevant papers, such as [2], which we will cover in subsection 4.2.6. The caterpillar algorithm as presented in this paper works for the general case redistribution from cyclic(x) to cyclic(y). It only considers one processor set *P*, however, due to the nature of the algorithm, *P* can contain both the source and destination processor set - effectively allowing for a different processor sets between the source and destination distribution. This also means that in this paper, it is not assumed that all processors in P are considered when assigning elements in the source or destination distribution. In the other papers, this is usually the case.

The idea behind the caterpillar algorithm is almost the same as the simple scheduling algorithm for all-to-all communication we discussed covering the multi-phase method in the subsection 4.2.2. Essentially, each processor starts off with a different first destination processor and iterates through all the others in a static order. In this case, it works via a given processor p_i exchanging data with $p_{(P_N-i-d) \bmod P_N}$ in a given step, with d being the current communication step and $1 \leq d < P_N - 1$.

The only real difference is the ability to account for non all-to-all communication by mapping processors that are only sending and ones that are only receiving to the same index slot. This way, the algorithm does not have to iterate through all P_N processors in all cases, but only through the max(sending processors, receiving processors), which can, if both sets are disjoint, halve the communication step amount.

As an example, let us consider a case in which we have $P_N = 6$, with 3 sending processors and 3 receiving processors, 0 through 2 and 3 through 5 respectively. In this case, we can safely map p_i and p_{i+3} to the same index i, where $0 \leq i < 3$. Let us call this new set involving these mappings G, which we will use in place of P for the algorithm. This means that g_i is treated as p_i when sending data, but as p_{i+3} when receiving data. In other words, when the algorithm determines g_s must send to g_d, this means processor p_s must send to processor p_{d+3}. To illustrate this point, let us look at the third step of the communication $d = 3$, considering $G_N = 3$ as described above:

- g_0, in other words, p_0 communicates with $g_{(3-0-3) \bmod 3} = g_0$, so, p_3
- g_1, in other words, p_1 communicates with $g_{(3-1-3) \bmod 3} = g_2$, so, p_4
- g_2, in other words, p_2 communicates with $g_{(3-2-3) \bmod 3} = g_1$, so, p_5

Here, we can quite clearly see how g_0 sends to g_0, again, but these take on different values depending on whether they are on the sending or receiving side. The sending g_0 is p_0 but the receiving g_0 is $p_{0+3} = p_3$. The total communication matrix of the given example would be the one that can be seen in Table 4.11.

k / p	**p₀**	**p₁**	**p₂**
$\mathbf{k_0}$	5	4	3
$\mathbf{k_1}$	4	3	5
$\mathbf{k_2}$	3	5	4

Table 4.11: Communication matrix for the problem described above using the caterpillar algorithm

Evidently, this is contention-free and optimal, in so far as we have all-to-all communication.

Aside from this scheduling algorithm, the paper also presents the idea of communication pipelining. The idea is that data sent in the form of a message is split into much smaller packets. Then, the unpacking and communication process is overlapped: while the small data packets are being sent and the processors receiving them would usually wait, they instead unpack the small packet that was received previously. This method is used when applying the caterpillar algorithm, if the total data sent is large enough to be split into small packets.

This scheduling algorithm is very important, as the one used in ScaLAPACK for array redistribution, and it gets the job done by ensuring all sending processors communicate with all receiving ones. However, this could also be seen as a problem: this algorithm does not take into consideration that there are pairs of source and destination processors that may not exchange messages at all, as we will see when analyzing the following papers, in which non all-to-all communication is an important topic. As such, some processors may not be sending or receiving data at all during some steps of communication, which is a clear inefficiency, as it also leads to needing more total communication steps than a schedule that takes this into account [18]. Furthermore, we do not take any care to schedule messages into a communication step according to their size and since communication step costs are dominated by the size of the largest message being sent, this can lead to inefficiencies.

4.2.5 1998: Scheduling Block-Cyclic Array Redistribution

This paper [18] builds on the foundations of Walker and Otto's paper from 1996 [1], which we have previously discussed in subsection 4.2.3. It builds on said paper by creating an algorithm that can arrive at an optimal schedule for the general case of a cyclic(x) to cyclic(y) redistribution over arbitrary processor sets P and Q, compared to the original paper which focuses only on the one-divides-the-other case. This is done by modelling the communication involved as a bipartite graph and then using a matching algorithm to derive the best schedule. This paper also focuses on the one-dimensional case exclusively, as an extension to the multi-dimensional is merely the tensor product of the individual dimensions, essentially, a series of one-dimensional redistributions due to the limitations of HPF, an extension to the Fortran language this paper is motivated by. As this paper is more complex than most of the others in terms of the mathematics involved, we will reference the propositions and their proofs made in the paper when we feel it is appropriate. This is to avoid covering every mathematical process the paper engages in, as doing this would unnecessarily bloat this sub-section.

In principle, this paper sticks with the same idea as its predecessor: communication is still scheduled into steps in which contention is avoided by guaranteeing that each sender only sends at most one message and each receiver only receives at most one message. The concept of a superblock, a certain number of elements after which communication repeats, is also recycled. So, for a superblock of size L and an array of size N, $N \geq 2L$, element i, $0 \leq i < L$, elements i and $L + i$ are distributed onto the same processor in both source and destination distribution. Here the superblock size $L = lcm(P_N \times x, Q_N \times y)$. In addition, message size is now considered when constructing the schedule: in the previous paper the goal was simply contention-free communication steps. As a consequence of the two criteria set up for the process send schedule, the number of communication steps was guaranteed to be minimum, but the cost of each step was not taken into account. This paper, on the other hand, aims to create a schedule with a minimal cost, which is evaluated in one of two ways:

1. by the number of communication steps H
2. by the total cost, i.e. $\sum_{k_n=0}^{H-1} cost(k_n)$, whereby k_n is a given communication step and $cost(k_n)$ is the longest message sent in communication step k_n

The first option is the simpler one, as it implicitly assumes the cost per communication step is roughly equal, thereby avoiding explicitly calculating it. The second option is more precise, though it does not take into account other aspects such as link contention.

Before diving into the actual algorithms for computing an optimal schedule, the the concept of classes is introduced, the purpose of which is to group together pairs of processors sharing certain characteristics. Classes are sets of pairs (p_i, q_j) and are defined as shown in Equation 4.20, assuming $gcd(x, y) = 1$, in other words, that x and y are relatively prime. This is a safe assumption, because translation between a redistribution of cyclic(x) to cyclic(y) and cyclic(x') to cyclic(y') where $x' = gcd(x', y') \times x$ and $y' = gcd(x', y') \times y$ is as simple as increasing all message lengths by a factor of $gcd(x', y')$. The scheduling does not change. Here, $0 \leq k < g$, meaning there are g total classes, with $g = gcd(P_N \times x,\ Q_N \times y)$ and u, v such that $x \times u - y \times v = 1$.

$$\begin{pmatrix} p_i \\ q_j \end{pmatrix} \text{ with } \begin{pmatrix} i \\ j \end{pmatrix} = \lambda \times \begin{pmatrix} y \\ x \end{pmatrix} + k \times \begin{pmatrix} u \\ v \end{pmatrix} \bmod \begin{pmatrix} P_N \\ Q_N \end{pmatrix}; \quad 0 \leq \lambda < \frac{P_N \times Q_N}{g} \tag{4.20}$$

These classes have multiple interesting characteristics that will not be proven here, but can be found in the paper, in section 4.2. If $k \in [1 - x, y - 1] (\bmod g)$, that means all pairs in the class communicate. If not, none of them communicate. Furthermore, all processor pairs in the same class exchange messages of the same length. Classes are also disjoint from each other, that is to say, each possible pair (p_i, q_j) for all possible values of i and j is in only one class. And finally, all classes are of the same cardinality, that is $\frac{P_N \times Q_N}{g}$. This concept of classes is derived from the general redistribution block equation, the general case version of the equation that was used to derive an optimal schedule in the paper [1], which we covered in subsection 4.2.3. The exact process by which the paper arrives at the class definition can be found in sections 4.1 and 4.2 of the paper itself.

Knowing this about classes, we can begin to derive the optimal schedules. This is simplest in the case where both $gcd(x, Q_N) = 1$ and $gcd(y, P_N) = 1$. This is because these conditions guarantee that the pairs belonging to each class are evenly distributed across P and Q, which follows from proposition 3 and the proof provided for it in section 4.3.2 of the paper. For a total of $\frac{P_N \times Q_N}{g}$ elements, each value i of p_i is represented $Q_N \div g$ times, meaning, a given processor p_i sends to $\frac{Q_N}{g}$ destination processors in a given class. In turn, each value of j for q_j is represented $\frac{P_N}{g}$ times, which means a given processor q_j receives messages from $\frac{P_N}{g}$ different source processors in a given class. This allows for scheduling communications on a class by class basis. For a contention free communication, we know no processor can either send or receive more than one message in a given communication step. This means that, to schedule a single class, we need $\frac{max(P_N,Q_N)}{g}$ steps. Each step, then, naturally, must consist of $min(P_N, Q_N)$ messages. In the steps themselves, blocks of size g for both source and destination processors are considered. For each $p_{i'}$ within such a block of source processors, a message must be sent to each member of the blocks of destination processors $q_{j'}$ that have index $j' = y^{-1} \times (p_{i'} \times x - k)$, where y^{-1} is the inverse of y modulo g, which we know is defined because it is assumed $gcd(x, Q_N) = gcd(y, P_N) = 1$. This formula also follows from proposition 3 in section 4.3.2 of the paper. Knowing this, the only thing left is to arrange a suitable permutation of the blocks pairs such that no single block appears twice in a given step. How this is accomplished does not matter as the order of communication is irrelevant. This scheduling method guarantees that each communication step includes only messages of the same size, as each step only consists of pairs of the same class. This means the cost of each step is minimized and therefore, the total cost, because the minimum number of steps is already guaranteed.

Now, let us consider this algorithm as it pertains to three sub-cases. The first is when $g > (x + y - 1)$. This means that there are classes of pairs that do not communicate, according to the characteristics defined earlier. Therefore, the communication is not all-to-all. This does not matter for this algorithm, however, as only communicating classes are considered. If $g = x + y - 1$, we have all-to-all communication and can apply the algorithm to all classes. Finally, if $g < x + y - 1$, the classes k that have the same value modulo g are summarized into one and the algorithm is applied. The algorithm works easily for any relation of g to x and y assuming the basic assumptions hold. Perhaps noteworthy is also that this algorithm allows for the same results as the technique Walker and Otto presented in this papers predecessor [1] (subsection 4.2.3), for a redistribution from cyclic(x) to cyclic(kx) if $gcd(P_N, k) = 1$. By reducing the case of said redistribution to a redistribution from cyclic(1) to cyclic(k), as we know is possible by simply scaling down message lengths accordingly, this algorithms first condition of $gcd(x, Q_N) = 1$ is already fulfilled. Then merely the second, $gcd(y, P_N) = 1$ needs to hold, which is equivalent to $gcd(P_N, k) = 1$ in this case. Applying the algorithm then leads to the same results, showing that this paper can replicate these original findings.

Now, let us consider the general case, where no assumptions are held regarding $gcd(x, Q_N)$ and $gcd(y, P_N)$, thus making the algorithm generally applicable to all one-dimensional redistributions. As the conditions from earlier are no longer maintained, it is now not guaranteed that each classes processor pairs are evenly distributed for each possible p_i and q_j. However, we can determine in what manner they are distributed. Given $gcd(x, Q_N) = x'$ and $gcd(y, P_N) = y'$, and correspondingly $P_N = P'_N \times y'$, $Q_N = Q'_N \times x'$ as well as $g_0 = gcd(P'_N, Q'_N)$, Lemma 3 in section 4.4.1 of the paper states the following: the processor pairs of all classes are distributed as follows.

- $\frac{P'_N}{g_0}$ entries for Q'_N destination processors, none for the remaining $Q_N - Q'_N = (x' - 1) \times Q'_N$ processors
- $\frac{Q'_N}{g_0}$ entries for P_N destination processors, none for the remaining $P_N - P'_N = (y' - 1) \times P_N$ processors

This means that a method akin to the previous algorithm wherein we schedule communication class by class is no longer feasible. This would lead to a schedule in which, in every communication step, wthere is no longer a permutation of either every sending processor or every receiving processor (whichever set is smaller). This is obviously inefficient, as it means we are missing out on possible contention-free communications in every step: we would need more communication steps to compensate, which increases the total cost of the schedule. What we must do is determine the number of communication steps we can feasibly achieve: as each processor can only send or receive one message per step, the minimum number of communication steps is necessarily limited by the number of messages the source processor that communicates with the most destination processor sends, m_R, or the number of messages the destination processor that communicates with the most source processors receives, m_C. Proposition 5 in section 4.4.2 of the paper states these values are defined as shown in Equation 4.21, assuming communication is not all-to-all, so $g > (x + y - 1)$,. This is assumed from here on out.

$$\begin{aligned} m_R &= \frac{Q'_N}{g_0} \times \left\lceil \frac{x+y-1}{y'} \right\rceil \\ m_C &= \frac{P'_N}{g_0} \times \left\lceil \frac{x+y-1}{x'} \right\rceil \end{aligned} \tag{4.21}$$

Proof of this proposition can be found in the same section. Now we know the number of steps H is at minimum $max(m_R, m_C)$. Knowing this, there are now have two ways forward in defining the algorithm: either achieving exactly that minimum step number, or minimizing the total cost of the schedule according to the earlier definition, which could lead to a greater amount of steps. With either method, the paper makes use of a bipartite matching. The communication taking place can be represented as a weighted graph $G = (V, E)$, vertices and edges respectively. The vertices then consist of $P \cup Q$, the sending and receiving processors, which are always considered separate sets here even if they might not be disjoint in reality. The edges then consist of pairs (p_i, q_j) that communicate so all pairs in classes k where $k \in [1 - x, y - 1] (\bmod g)$. Their weight is equivalent to the length of the message sent. G is a bipartite graph: in other words, its vertices can be divided into two disjoint sets (in this case, P and Q) and for any edge (p_i, q_j), p_i is member of one set (in this case P) and q_j of the other (in this case Q).

The degree of G is the maximum degree of its vertices d_G, that is to say, the largest amount of edges connected to any one vertice. As the vertices consist of the source and destination processors and the edges are the messages sent, this is naturally equal to what was determined earlier, $max(m_R, m_C)$. This degree is in turn equal to the edge coloring number of the graph, as it is bipartite, as dictated by the König's edge coloring theorem. By definition, edge coloring means dividing edges into different groups such that no two edges in the same group are incident - they do not share a common vertice. These groups are called maximum matchings. The edge coloring number, then, is the minimum amount of colors (matchings) needed to make this possible. Translating this to our case, this means the amount of communication steps needed to divide up the communicating pairs such that no two pairs in a step share the same source or destination processor.

Knowing this number is $d_G = max(m_R, m_C)$, it is easy now, to optimize for the first criteria, so, creating a schedule with the minimum number of steps. We must iterate d_G times and, in each step, extract from E a maximum matching that saturates all maximum degree vertices. This means that all of the vertices with the most amount of edges left over are included in the matching of a given step - i.e. the ones that send or receive in every step. In choosing these matchings, it is logical to pick them such that in every step, it is a matching with the maximum weight. Weight being defined as the sum of the weight of all edges in the matching. The schedule then consists of d_G steps, each one performing all the communications between the processor pairs in a given matching.

On the other hand, if we want to optimize for total cost of all steps, the paper suggests making use of a greedy algorithm. Here we must simply select maximum weighted matchings iteration after iteration until the whole graph has been consumed.

For both algorithms, the following model is used. First, A is defined, the $|V| \times |E|$ sized incidence matrix of G, which means that $a_{ij} = 1$ if edge j is incident to vertex j, 0 otherwise. Then, a set of vectors $s \in \mathbb{Q}^{|E|}$ is considered for the matching. These are defined by the following two criteria :

- $s(e) \geq 0 \quad \forall e \in E$
- $\sum_{v \in e} s(e) \leq 1 \quad \forall v \in V$

In short, $s(e) = 1$ if and only if edge e is selected in the matching. For the sake of this matching algorithm, this set of vectors can be written as shown in Equation 4.22.

$$s \geq 0, A \times s \leq b, \text{where b} = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} \in \mathbb{Q}^{|V|} \tag{4.22}$$

These restrictions on s are very important: naturally, it makes no sense for the vector defining the processor pairs involved in a communication step to have entries > 0. In addition, $A \times s$ being upwards bounded by a vector of 1's is necessary, because an entry > 1 at any point in the vector would imply two edges in the matching containing the same vertice. In other words, contention, which we want to avoid.

Now, in order to find the maximum weighted matching needed for the steps of both algorithms (one to minimize costs, one to minimize number of steps), we must determine s such that it maximizes $c^t \times s$ with $c \in \mathbb{N}^{|E|}$ being the weight vector of the edges. All conditions for s described above still apply. This makes sense intuitively: $s(e)$ is 1 exactly when the edge is included in the matching, so $c^t \times s$ thus sums the weights for all edges included in the matching, which is what we aim to maximize. This is all that is necessary in order to minimize the total cost using the greedy algorithm: simply determining this maximum matching iteratively until all steps are concluded. In order to minimize the number of communication steps, one condition for the value of s is adjusted: for each vertex v at position i of the vector b, the constraint regarding $(A \times s)_i \leq b_i$ becomes $(A \times s)_i = b_i$. This is because in order to minimize the number of steps, all vertices of maximum degree, i.e. processors that send/receive the most messages, must send/receive in every step. This is what bounds the number of steps to $d_G = max(m_R, m_C)$, as was discussed earlier.

Let us illustrate one step of the algorithm for an appropriate case. Consider a redistribution from cyclic(2) on 5 processors to cyclic(5) on 6 processors for an array of size 30. In this case, $gcd(x, Q_N) = gcd(2, 6) = 2 \neq 1$ and $gcd(y, P_N) = gcd(5, 5) = 5 \neq 1$ - clearly, we do not have an even distribution across classes. Furthermore, $gcd(x, y) = gcd(2, 5) = 1$ and $x + y - 1 \leq gcd(P_N \times x, Q_N \times y) = 2 + 5 - 1 \leq gcd(5 \times 2, 6 \times 5) \equiv 6 < 10$. In other words, we do not have all-to-all communication. Furthermore, this is a general case redistribution where $y \neq kx$ and we have different processor sets. This makes it a case that would be impossible to handle using previous algorithms. In this paper we simply construct the respective communication graph and weight table after index computation, which can be seen in Figure 4.2 and Table 4.12 respectively. For this example, extracting maximum matchings is relatively trivial - in the first step, there are clearly multiple options for such a matching. One such example would be: (p_0, q_0), (p_1, q_2), (p_2, p_4), (p_3, q_1), (p_4, q_3). Continuing this way, we might arrive at the schedule shown in Table 4.13, with both the greedy or the stepwise algorithm. Interesting to note here is that there are alternative schedules that are just as optimal depending on the first matching.

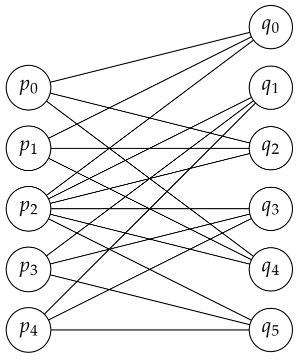

Figure 4.2: Weighted, bipartite graph of the communication between P and Q

	p_0	p_1	p_2	p_3	p_4
q_0	2	2	1	–	–
q_1	–	–	1	2	2
q_2	2	2	1	–	–
q_3	–	–	1	2	2
q_4	2	2	1	–	–
q_5	–	–	1	2	2

Table 4.12: Weights of edges in graph

k / p	$\mathbf{p_0}$	$\mathbf{p_1}$	$\mathbf{p_2}$	$\mathbf{p_3}$	$\mathbf{p_4}$
$\mathbf{k_0}$	0	2	4	1	3
$\mathbf{k_1}$	2	4	1	3	5
$\mathbf{k_2}$	4	0	2	5	1
$\mathbf{k_3}$	–	–	3	–	–
$\mathbf{k_4}$	–	–	3	–	–

Table 4.13: A possible communication matrix for the problem described above, using the bipartite matching scheme

Overall, it can be said that this paper is somewhat of a milestone: it expands the algorithm for acquiring an optimal schedule to a much more general format than the previous works, which essentially required use of the multi-phase method for a redistribution that was not in the format of the one-divides-the-other case. Furthermore, the schedule is derives is optimal. The only problem is that the algorithm itself has a very high complexity. This is because the best algorithm for weighted, bipartite matching at the time, the Hungarian method has a cost of $O(|V|^3)$. Then, since up to $max(P_N, Q_N)$ iterations of this algorithm are required to produce the schedule and $|V| = P_N + Q_N$, we end up with an algorithm with a runtime complexity in $O((P_N + Q_N)^4)$, which is obvious problematic for larger processor sets. This means that while it may optimize communication, the algorithm itself is not necessarily an optimal way to arrive at that schedule. It is worth noting, however, that there is a chance this runtime could be improved upon using newer, potentially faster solutions to the problem of weighted, bipartite maximum matching (the assignment problem).

4.2.6 1999: Efficient Algorithms for Block-Cyclic Array Redistribution Between Processor Sets

This paper [2] deals with the problem of communication scheduling regarding redistribution of the one-divides-the-other case, more specifically, cyclic(x) to cyclic(kx) from P_N to Q_N processors. The paper does not specifically delve into multidimensional redistribution and focuses on the one-dimensional case, developing a new algorithm making use of generalized circulant matrix characteristics to determine an optimal schedule. The goal is to create a schedule that uses the minimum number of communication steps and fully utilizes network bandwidth. Primarily, the purpose of this algorithm is to compute the optimal schedule quickly, much unlike the paper we just covered. Compared to previous papers that covered the one-divides-the-other case, it stands out by covering the case where the source and destination processor sets are different.

Much like the papers [1] (subsection 4.2.3) and [18] (subsection 4.2.5), this paper restricts itself strictly to observing the first superblock of elements in the array that is to be redistributed. Since this paper deals with the one-divides-the-other case, this is once again defined as $L = lcm(P_N, k \times Q_N)$. As such, in order to represent both the initial and the final distribution, only the first superblock needs to be considered. Then, the source distribution can be represented as a two-dimensional table D_{init} with the dimensions $(L \div P_N) \times P_N$. Each column stands for a source processor p_i and each row is a specific local block index lb_i. The entries of the table, then, are the indexes of the global block stored at the respective local block index on the processor. D_{final} is the corresponding table for the destination distribution, with the dimensions $(L \div Q_N) \times Q_N$. Table 4.14 shows an example of D_{init} for a redistribution from cyclic(1) on 4 processors to cyclic(2) on 6 processors.

index / p	**p0**	**p1**	**p2**	**p3**
lb0	0	1	2	3
lb1	4	5	6	7
lb2	8	9	10	11

Table 4.14: D_{init} for the problem described above

Additionally, the destination source processor table T is defined. It is the same as D_{init} except instead of storing the global array index as its entries, it stores the destination processor the corresponding local block on the respective processor is sent to. The example D_{init} we constructed earlier would have the corresponding T shown in Table 4.15.

index / p	**p0**	**p1**	**p2**	**p3**
lb0	0	0	1	1
lb1	2	2	3	3
lb2	4	4	5	5

Table 4.15: T corresponding to the earlier D_{init}

T can now be viewed as a potential schedule for converting from D_{init} to D_{final}, where each row i represents a communication step k_i. This is identical to the concept of a communication matrix first described in this chapters introduction and then used later in [1] (subsection 4.2.3. Naturally, using T as the schedule is an inefficient scheme, as there is contention in several steps, since more than one source processor sends to the same destination processor. Therefore, the goal is now to reorganize the elements of T into C_{send}, such that C_{send} achieves a contention-free communication if communication is scheduled according to rows. Of course, in this conversion from T to C_{send}, elements can only be changed row wise: since each column represents a processor, any changes between columns represents inter-process communication, which only occurs using the schedule, not while constructing it. What this in-column reorganization is meant to achieve is transforming the distribution table T into a generalized circulant matrix, where each row consists of a permutation of some P_N sized subset of $0, \ldots, Q_N - 1$. Here it is assumed $Q_N \geq P_N$.

A generalized circulant matrix is defined as an $M \times N$ matrix, such that the matrix can be divided to submatrices of size $m \times n$, where $M \bmod m = 0$ and $N \bmod n = 0$, so, M is a multiple of m and N and N a multiple of n. Then, the block matrix consisting of the submatrices is a circulant matrix, as are the individual submatrices. A circulant matrix of size $m \times n$ is defined by the following two criteria:

1. If $m \leq n$, row j = row 0 circularly right shifted j times, $0 \leq j < m$
2. If $m > n$, column l = column 0 circularly down shifted l times $0 \leq l < n$

Using these definitions of a circulant matrix and a generalized circulant matrix, we can demonstrate examples for both the case where $m > n$ and the one where $m < n$ in Figure 4.3. Furthermore, we devise a generalized circulant matrix where $N = M = 8$ and $n = m = 4$, as shown in Figure 4.4. The respective equivalent submatrices are colored in white and grey respectively.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 1 & 2 & 3 \\ 3 & 4 & 1 & 2 \end{bmatrix} \qquad \begin{bmatrix} 1 & 4 & 3 \\ 2 & 1 & 4 \\ 3 & 2 & 1 \\ 4 & 3 & 2 \end{bmatrix}$$

Figure 4.3: Two circulant matrices - on the left: m = 3, n = 4, on the right: m = 4, n = 3

It is very clear why working with (generalized) circulant matrices here is sensible: their definition already guarantees that the first criteria for a process send schedule/communication matrix are met, assuming the very first row is a permutation of a subset of Q. As a reminder, those two criteria, adjusted slightly to consider that Q can be different from P in the problem this paper is solving for:

1. The rows of the process send schedule are permutations of the process ranks, so, permutations of a P_N sized subset of $0, 1, \ldots, Q_N - 1$
2. The ith column of the send schedule is a permutation of the processes to which process p_i must send data

1	2	3	4	5	6	7	8
4	1	2	3	8	5	6	7
3	4	1	2	7	8	5	6
2	3	4	1	6	7	8	5
5	6	7	8	1	2	3	4
8	5	6	7	4	1	2	3
7	8	5	6	3	4	1	2
6	7	8	5	2	3	4	1

Figure 4.4: A generalized circulant matrix

In a circulant matrix, if the first row meets the first criteria, so do all other rows. This is also true for the generalized variant. Assuming for now that this is the case, we know criterion 2 is also met: because the generalized circulant C_{send} is constructed from T only by moving around elements in a given column (e.g. belonging to the same processor), we know it must be given, because it is already given by T, per definition, as T represents all communication within a superblock, which is representative of all communication of the whole redistribution. Now, all that is left is, firstly, the way to efficiently compute the entries of C_{send}. Secondly, demonstrating that C_{send} is indeed simply a reorganized version of T. Then, finally, demonstrating that the first row of said matrix does indeed consist of P_N distinct elements.

The way to determine the send schedule differs somewhat depending on what type of communication there is, specifically, whether it is all-to-all or not. The criteria used to determine whether or not this is the case is more or less the same one from [18] (subsection 4.2.5) and also follows from the basic redistribution equation: if $gcd(P_N, k \times Q_N) > k$, we do not have all-to-all communication, otherwise we do. It slightly differs from the criteria in the aforementioned paper only because here it is not assumed that $gcd(x, kx) = 1$ (and that message lengths are scaled accordingly if this is not the case). It is important to note, when discussing the all-to-all case, this paper assumes that not all processors send messages of the same length, because if they do, the result is simply a trivial round robin schedule, which need not be discussed further.

In practice, determining the entries of C_{send} is actually very simple in all cases. They are simply computed according to Equation 4.23 with variables as defined by Equation 4.24 if the communication is not all-to-all.

$$C_{send}(i,j) = ((n \times (j_1 - i_1)) \bmod P_1 + ((i_2 - j_2) \bmod Q_1) \times P_1) \bmod Q_N \tag{4.23}$$

$$\begin{aligned}
&P_1 = \frac{P_N}{G_1} \quad G_1 = gcd(P_N, k) \quad Q_1 = \frac{Q_N}{G_2} \quad G_2 = gcd(P_1, Q_N) \quad k_1 = \frac{k}{G_1} \\
&i_1 = \frac{i}{Q_1} \quad j_1 = \frac{j}{G_1} \quad i_2 = i \bmod Q_1 \quad j_2 = j \bmod G_1 \\
&m, n \text{ s.t. } n \times k_1 - m \times P_1 = 1
\end{aligned} \tag{4.24}$$

Consider a redistribution from cylic(2) on 4 processors to cyclic(4) on 6 processors. In this case, $k = 2$ and $gcd(P_N, k \times Q_N) = gcd(4, 12) = 4 > 2$, meaning we do not have all-to-all communication. The size of the communication table is given by $\frac{L}{P_N} \times P_N = \frac{12}{4} \times 4 = 3 \times 4$. Then, applying the above formulas to each entry (i, j), indexed from 0, we arrive at the results shown in Table 4.16. The schedule is contention-free and minimal.

We can also see that the resulting communication matrix for the schedule is indeed a generalized circulant matrix: it can be viewed as a 1×2 block matrix, which is necessarily circulant. The two blocks, each of size 3×2, are also circulant matrices, since we can observe that for both of them, the second column is merely the first column circularly down shifted by one. This means that the conditions for a generalized circulant matrix are fulfilled: the communication matrix has been split into blocks such that the block matrix and the individual blocks are all circulant matrices.

k / p	**p0**	**p1**	**p2**	**p3**
$\mathbf{k_0}$	0	2	1	3
$\mathbf{k_1}$	4	0	5	1
$\mathbf{k_2}$	2	4	3	5

Table 4.16: A communication matrix for the problem described above, derived using a generalized circulant matrix algorithm

Moving on to the case in which we do have all-to-all communication, the approach must be changed slightly, because some processors now potentially send more than one local block to the same destination processor. $gcd(P_N, k \times Q) \leq k$, the condition for all-to-all communication, implies that the column size $lcm(P_N, k \times Q_N) \div P_N \geq Q_N$. This is because of the mathematics demonstrated in Equation 4.25.

$$\begin{aligned} & lcm(P_N, k \times Q_N) \div P_N \geq Q_N \\ \equiv & lcm(P_N, k \times Q_N) \div P_N \times gcd(P_N, k \times Q_N) \geq Q_N \times gcd(P_N, k \times Q) \\ \equiv & P_N \times k \times Q_N \div P_N \geq Q_N \times gcd(P_N, k \times Q_N) \\ \equiv & k \times Q_N \geq gcd(P_N, k \times Q_N) \times Q_N \geq k \times Q_N \end{aligned} \tag{4.25}$$

As such, we can construct a send schedule of size $Q_N \times P_N$. Q_N can also be written as $Q_1 \times G_2$, with Q_1 being the column size of each submatrix making up the total generalized circulant matrix that is the send schedule according to the proof of Theorem 1. In turn, the block matrix made up of these submatrices has size $K_1 \times P_1$ according to the same theorem, which also means $\frac{L}{P_N}$ can be written as $K_1 \times Q_1$. This means that, essentially, every row of submatrices with index $i \geq G_2$ can be folded into the row with index i mod G_2. Because $\frac{L}{P_N}$ mod Q_N is not necessarily 0, this means some processors must send longer messages than others. Precisely, the first $(K_1 \bmod G_2) \times Q_1$ communication steps now involve sending messages of (relative) size $\lceil \frac{K_1}{G_2} \rceil$ and the others ones of size $\lfloor \frac{K_1}{G_2} \rfloor$. In terms of computing the schedule, all it means is that we now need to compute Q_N rows of P_N entries using the algorithm, as opposed to the previous $lcm(P_N, k \times Q_N) \div P_N$ rows.

For example, given a redistribution from cyclic(1) on 4 processors to cyclic(3) on 6 processors, we have $gcd(4, 3 \times 6) = 2 < 3 = k$, hence, all-to-all communication. Before folding over the rows, we would have a column size of $lcm(4, 3 \times 6) \div 4 = 9 > 6 = Q_N$. Hence, we would fold over the last 3 rows into the first 3: the new schedule now has the dimensions 6×4 and in the first 3 communication steps each processor sends two blocks, in the final 3 just one block.

The process of arriving at the earlier formula is an extensive proof that demonstrated in the proof for Theorem 2 in section 4.2 of the paper. This proof in turn requires parts of the proof of Theorem 1, also in section 4.2 of the paper, to function. What is important is that this formula is the result of Theorem 1. Theorem 1 states that C_{send} can be derived from T by column-wise rearrangement, and that the first row C_{send}, and thereby every other row, contains P_N distinct elements of Q. The proof for this theorem that shows these characteristics is very extensive, too much to reasonably include it in this summary.

As a whole, this paper represents a marked improvement over [18] (subsection 4.2.5) in terms of its performance - since every processor only needs to apply the formula to the parts of the communication table relevant to itself, that is to say a single column for each source processor. Then taking into account that destination processors must also use a similar formula to compute the receive schedule, the runtime complexity is in $O(max(P_N, Q_N)$ because a source processor must at most compute an entry for every destination processor in the case of all-to-all communication, and vice versa. Furthermore, in the time it takes to compute the schedule we can also compute the index computation using a table called D_{send} with the same dimensions as C_{send} which computes each local block number corresponding to the destination processor entry in C_{send}. This means that it is possible to perform both schedule computation and index computation in $O(max(P_N, Q_N)$.

4.3 Performance Comparison

When comparing the performance of scheduling, it is sensible to divide it into three parts: first, the comparison in performance between an unscheduled communication and a scheduled communication. This allows us to gauge the performance benefits of communication scheduling in general, because all of the algorithms we discussed compute optimal schedules. Then, secondly, the performance of the algorithms we discussed in the previous section - how do they compare in terms of the time it takes to actually compute the optimal schedule, and how do the computed schedules differ. Here we have to be careful and take into consideration that many of these algorithms only consider the case of one-divides-the-other redistribution. Finally, we want to evaluate how a single-phase redistribution compares to multi-phase for several cases.

4.3.1 Scheduled vs non-scheduled

The 2001 paper [8] provides such a comparison, using its own indexing algorithm, which is similar to the PITFALLS method. It also uses its own scheduling algorithm, similar in nature to that of [17] (subsection 4.2.2), which provides a contention-free schedule in which only the processors communicate that must necessarily exchange messages - in short, the data transfer time should be similar to the results of the generalized matrix based algorithm from [2] (subsection 4.2.6), notwithstanding inefficiencies due to not taking into account message sizes.

It compares a case using a one-dimensional array of size $N = 120.000$, varying the number of processors P_N from 4 to 256 and using 3 communication steps, presumably the optimal number. The specific distribution parameters of the source and destination distribution are not stated, though we know that it must be a cyclic(x) to cyclic(kx) one-divides-the-other case, as that it all the this papers algorithm deals with. What we observe is that the scheduled communication outpaces the unscheduled one in every case, with the speed-up increasing significantly as the number of processors does. At the minimum amount of 4 processors, the speed-up appears to be roughly 1.15x, and at the maximum of 256 processors, more than 4x. This appears to strongly support the conclusion that scheduling is worthwhile. This would also be backed up by the general consensus that contention slows down communication - synchronous algorithms with contention in each step are slower than ones without contention. It is nevertheless important to note that a brute force schedule that is asynchronous in nature, such as the one presented in [1] (subsection 4.2.3) can nonetheless be faster, as is shown both in that paper and in the 2006 paper [19]. However, this carries with it the disadvantage of much greater memory requirements.

The performance analysis done in [1] (subsection 4.2.3) for cyclic(x) to cyclic(kx) redistribution, gives us a similar insight. Performance tests are done for multiple algorithms: the initial versions of the synchronized and unsynchronized algorithm presented in paper, as well as the synchronous algorithm using a random schedule and the synchronous algorithm with the optimal schedule computed by the scheduling algorithm. They were tested for several different values of P_N (source and destination set always equal): 3, 10, 16, 32 and 64. Furthermore, k was varied from 2 to 22. The results were as follows: the synchronized communication algorithm had significantly worse performance than the unsynchronized version, taking over 3 times the amount of time if $k = 22$ and $P_N = 64$. Generally, the higher k and the greater P_N, the worse the performance of the synchronized communication algorithm, both absolutely and comparatively. This can be attributed to the fact that more processors open the door to a greater amount of possible contention and hotspots. Furthermore, the schedule provided by the basic version of a synchronized algorithm is especially contention heavy with high values of k, because communication steps will have up to $max(P_N, k)$ processors sending to the same processor.

When the schedule is randomized, as predicted, the performance improves significantly and the impact of higher k and P_N declines - the random schedule is only roughly 30 percent slower than the unsynchronized version. The synchronized algorithm with the optimal schedule does even better - it is only marginally worse than the unsynchronized version, to the point it is almost negligible. In short, the conclusion we draw is the following: the difference in performance between an unsynchronized communication algorithm and a synchronization algorithm using an optimal schedule is only slight - however, the large benefits with regards to memory usage means scheduling is usually still worthwhile.

4.3.2 Schedule time comparison

The paper [2] (subsection 4.2.6) provides a very interesting comparison between the generalized circulant matrix based algorithm it develops, as well as the caterpillar algorithm developed in [13] (subsection 4.2.4) and the bipartite matching scheme described in [18] (subsection 4.2.5). These algorithms are especially of interest to us because they all allow for redistribution from one processor set to another and are among the most modern of the ones covered in this thesis. Furthermore, for the subsets of the problems these papers address that are covered by earlier papers such as [1] (subsection 4.2.3) and [17] (subsection 4.2.2), the schedules produced are just as optimal.

In this paper both complexity analysis for the scheduling algorithm as well as the actual communication time resulting from the schedule are provided. In doing so, the paper considers two cases: one with non all-to-all communication and another with all-to-all communication such that not all message sizes are equal. Furthermore, several example redistributions are used to show how the algorithms compare in practice, all of which consider only the cyclic(x) to cyclic(kx) case. The third case with all-to-all communication and equal message sizes is not only trivial, but leads to the same result for all three algorithms, so it is not considered.

The complexities of the scheduling algorithms themselves were explained in the earlier sections pertaining to the papers themselves. The caterpillar algorithm as well as the generalized circulant matrix algorithm are in $O(max(P_N, Q_N))$ for cases of both all-to-all and non all-to-all communication while the bipartite matching scheme is in $O((P_N + Q_N)^4)$ for non all-to-all communication, but does not explicitly consider the second case. This means that in terms of the time it takes to compute the schedule, the caterpillar algorithm and the generalized circulant matrix algorithm are roughly evenly matched, both far ahead of the bipartite matching scheme.

The actual data transfer time is modelled in terms of two quantities: t_s and t_e, the start-up time and the unit transmission time respectively, which we are acquainted with from the papers [15] (subsection 4.2.1) and [17] (subsection 4.2.2). Much like it was explained in these papers, one unit of the start-up time cost is invoked for each communication and one unit of the transmission time cost for each unit of message size. So a message of size m being sent from one processor to another has a total cost of $t_s + m \times t_e$. As we explained before, the cost of a communication step is thus dominated by the largest message sent as everything is sent in parallel. Therefore, to determine the data transfer cost of the three algorithms, one must know the number of communication steps and the size of the messages being sent.

The number of communication steps is given by $L_s = lcm(P_N, k \times Q_N) \div P_N$ in the case of the generalized circulant matrix algorithm, which is also the minimum. In the caterpillar Algorithm, this amount is $max(P_N, Q_N)$ because it does not check explicitly if any given pair of processors communicate, but rather iterates once through all of them. The bipartite matching scheme can also guarantee the minimum number of steps, so, the same as the matrix algorithm. Furthermore, as all messages sent are of the same size $\frac{M}{L_s}$ with $M = \frac{N}{P_N}$ assuming, as this analysis does, that we are dealing with cyclic(x) to cyclic(kx) redistribution and that we are in the case of non all-to-all communication. In case we have all-to-all communication with different message sizes, things change: the communication step amount of the matrix based algorithm is now also $max(P_N, Q_N)$ and the message size in the first blocks is increased, as we saw when analyzing the paper. Precisely speaking, the average message costs changes to $M \div max(P_N, Q_N)$. For the caterpillar algorithm, message sizes are now hard to predict: the maximum size of a message in each step is considered individually, which is described as m_i for step i. Then, assuming $Q_N \geq P_N$, as is the case in all of this papers tests, we have the following communication costs, shown in Table 4.17 and Table 4.18.

	Data transfer cost	Schedule comp. cost
Caterpillar algorithm	$Q_N \times (t_s + \frac{M}{L_s} \times t_e)$	$O(Q_N)$
Bipartite scheme	$L_s \times (t_s + \frac{M}{L_s} \times t_e)$	$O((P_N + Q_N)^4)$
Matrix algorithm	$L_s \times (t_s + \frac{M}{L_s} \times t_e)$	$O(Q_N)$

Table 4.17: Data transfer cost and schedule computation cost for non all-to-all communication

	Data transfer cost	Schedule comp. cost
Caterpillar algorithm	$Q_N \times t_s + t_e \times \sum_{i=0}^{Q_N-1} m_i$	$O(Q_N)$
Bipartite scheme	–	–
Matrix algorithm	$Q_N \times t_s + M \times t_e$	$O(Q_N)$

Table 4.18: Data transfer cost and schedule computation cost for all-to-all communication

Overall, when it comes to non all-to-all communication the caterpillar algorithm and the matrix algorithm take a similar amount of time to construct the schedules, but the schedule produced by the matrix algorithm is superior: because $L_s < Q_N$ in the case of all-to-all communication, the data transfer cost when using the matrix algorithm is less. The bipartite matching scheme produces the same schedule as the matrix algorithm, but is far inferior to both others ín terms of the schedule construction time. Switching the focus to all-to-all communication, we observe the change in the data transfer cost: while it is not obvious at first glance, the matrix algorithm comes out ahead again compared to the caterpillar algorithm. Since it guarantees that all messages in one communication step are of the same size, this means we minimize the impact of the different message sizes. The caterpillar algorithm gives no such guarantees, hence, in theory, the performance of the matrix algorithms schedule will always be greater or equal.

In practice, the results were tested for three examples, all of them cyclic(x) to cyclic(kx) redistributions. The parameters of the examples can be seen in Table 4.19.

The first tests were for non all-to-all communication, and what was measured was total redistribution time - that is to say, schedule computation time, index computation time, packing/unpacking time and data transfer time. For all three examples, the size of the array

	P_N	Q_N	k
Example 1	18	76	8
Example 2	30	66	15
Example 3	46	50	25

Table 4.19: Parameters for the first set of examples that are tested

to be redistributed varied from 808.704 to 14.174.980. The results were similar for all three examples: the caterpillar algorithm had by far the worst performance, whilst the matrix algorithm and bipartite matching scheme were closer, with the greater scheduling time of the bipartite matching scheme making it inferior in all cases. The absolute gap between the caterpillar algorithm and the other two grows with growing array size, but proportionately it stays roughly the same. Going from example 1 to example 3, as the ratio of Q_N to P_N grows smaller and k grows larger, the gap between the caterpillar algorithm and the other two decreases. In example 1, the speed-up of the matrix algorithm over the caterpillar algorithm is between 1.76 and 2.06x. In example 2, it is also roughly 2x and in example 3, it is reduced to about 1.6x. The gap in performance is the result of the gap in the number of communication steps: for example 1, the caterpillar algorithm requires around 78 steps, where as the bipartite matching scheme and the matrix algorithm only need 39. Hence, roughly a 2x speed-up. The same holds for the other examples.

Meanwhile, the gap between the bipartite matching scheme and the matrix algorithm comes down largely to the size of the array. For a small array, in the examples, the bipartite matching schemes scheduling time can make up to 17 percent of the total redistribution time. Even more severely, using a redistribution from 40 to 56 processors with $k = 25$ and an array size of roughly 900.000, it takes up more than 50 percent of the time of data transfer itself, which is highly significant.

The second set of tests covering all-to-all communication uses examples with different values of k, shown in Table 4.20.

	P_N	Q_N	k
Example 1	18	76	8
Example 2	30	66	8
Example 3	46	50	18

Table 4.20: Parameters for the second set of examples that are tested

Here, the redistributions time are significantly closer together for all examples and the matrix algorithm only has a speed-up of up to 1.25x over the caterpillar algorithm at high array sizes. At smaller array sizes the start-up costs and other aspects of redistribution time dominate so that the speed-up becomes almost negligible.

To summarize, the preference for using the three algorithms in cases of cyclic(x) to cyclic(kx) redistribution would be as follows. In the case of a redistribution involving non all-to-all communication, the clear favorite is the generalized circulant matrix based algorithm, as it produces an optimal schedule with less costs than the bipartite matching scheme and also deals with the index computation simultaneously. However, for very large array sizes, the two are close to interchangeable, as the schedule computation costs become almost negligible as long as the processor sets are not excessively large. The caterpillar algorithm, by comparison, creates a very sub-optimal schedule, making it the worst choice by a clear margin. If we are dealing with an all-to-all communication involving different message sizes, and the choice is only between the caterpillar algorithm and the generalized circulant matrix based algorithm, the choice is still clearly the latter due to the more optimal schedule. At small array sizes, however, this gap can also be considered negligible due to the dominance of other factors, such as start-up costs.

On the other hand, if we are to consider general case redistributions, where one parameter is not a multiple of the other, the results becomes a lot less clear. Both the caterpillar algorithm and the bipartite matching scheme are capable of doing redistributions of this kind in a single step - between them, it seems likely based on the results we discussed, that the bipartite matching scheme would have the edge. However, if we take into account the generalized circulant matrix based algorithm, we have to consider how multi-phase redistributions stack up to single-phase redistributions and in which case one may be preferable to the other. The theoretical aspect of this is already covered in [15] (subsection 4.2.1) and [17] (subsection 4.2.2).

4.3.3 Single-phase vs multi-phase

This subsection will serve to illustrate some practical examples of how single phase and multi-phase redistribution could compare based on the algorithms and the tests done in [17] (subsection 4.2.2). Here it is important to note that these tests only consider the algorithms of said paper, which are largely inferior to ones presented in later papers such as the ones discussed in the earlier section. Furthermore, they test how the two strategies compare pertaining to the case of cyclic(x) to cyclic(kx) redistribution, when one of the appeals of a multi-phase strategy is their application of algorithms that only work for the one-divides-the-other case, to the general case.

The first test compares the two strategies for a redistribution of an array from cyclic(240) to cyclic(8) on 32 processors. The array to be redistributed consists of elements of size 8 bytes. The size of the array is varied from below 10.000 up to more than 120.000, which is markedly smaller than a lot of the test cases used in other papers. The second test does the same, except the redistribution is cyclic(192) to cyclic(8), in other words, a smaller k is used. The multi-phase redistribution is, in both cases, a two-step redistribution. The intermediate distribution is chosen as cyclic(40) for the first case and cyclic(48) for the second. In all cases, the tests measure total redistribution time: much like in the previous section, that means indexing time, packing/unpacking time and communication time.

The test results for the first case show that the multiphase redistribution outperforms the single-phase up to an array size of roughly 100.000, at which point the single-phase strategy does better. In the second test case, we see similar results: the single-phase strategy starts outperforming the multi-phase strategy at an array size of around 90.000. This can be attributed to the increase in message size in each communication step resulting from the increased array size: the success of the multi-phase method depends on the significance of the total start-up time costs compared to the total data transfer costs. Rising array size, all else equal, means the single-phase strategies relative times will keep improving. The lower array size threshold needed for the single-phase strategy to start outperforming the multi-phase strategy in the second test case compared to the third is due to the smaller k. For a redistribution from cyclic(240) to cyclic(8), $k = 30$, where for cyclic(192) to cyclic(8), $k = 26$. A smaller k means a greater ratio of data transfer to communication start-up time, because the total amount of communication steps across all redistributions decreases, which means less start-ups.

The conclusion we draw is that a multi-phase approach can clearly be preferable to a single-phase approach even if the algorithms being used for each distribution are evenly matched. The optimal conditions for this are a relatively small array size and, in the case of a cyclic(x) to cyclic(kx) redistribution, large k: in other words, a significant gap in the block sizes of the source and target distributions. The exact numbers in which one may be preferable to the other depend on the ratio of t_s to t_e, the message start-up and data transfer times.

5 Discussion

5.1 Conclusion

Redistributing between block-cyclic arrays is of great importance for improving performance of certain algorithms, such as for example the Alternating-direction Implicit method or Fast-Fourier Transform. Such redistribution is generally split into two phases. First is index computation, where the data each source processor must send and what data each destination processor must receive is calculated. We found that the basic-cycle calculation method from [3] and later [14] is generally the most efficient when it comes to pure index computation, as its runtime is independent of factors such as the size of the array being redistributed or the amount of processors involved in the redistribution, which are usually scaling factors that can greatly hinder the performance for very large cases. [14] and [5] also extend this method to be able to handle arbitrary processor sets as well as multi-dimensional arrays.

The second part of array redistribution is communication scheduling: in order to minimize both memory requirements and redistribution time, it is sensible to use a synchronous communication algorithm with scheduling that minimizes node contention. In order to do this, an efficient communication schedule that deals away with node contention must be computed. We found that the best way to do this is in the case of a redistribution involving block sizes that are multiples of one another (e.g. cyclic(x) to cyclic(kx)) is to use the generalized circulant matrix formulism devised by [2]. This formulism can also be used for redistributions with no limits on the nature of the source and distribution factors. This is done by using a multi-phase redistribution strategy in which a series of redistributions are performed. This trades off a decrease in the amount of communication start-ups for increased total data transfer time, and is thus worthwhile especially for smaller array sizes. On the other hand, the bipartite matching scheme presented in [18] can perform all redistributions of one-dimensional arrays directly, but the schedule computation time is comparatively much higher. Both of these methods can handle arbitrary source and target processor sets, but do not explicitly provide an extension to the multidimensional case. Much like in the one-dimensional case, however, a dimension-by-dimension redistribution involving a series of of one-dimensional redistributions is feasible.

5.2 Future work

The topic of array redistribution between processor sets is very wide and this thesis focuses on only a small subsection of it. An interesting extension to this paper would be to focus on redistribution as it pertains to irregular distributions. The paper "Efficient Data Redistribution Algorithms From Irregular to Block Cyclic Data Distribution" [20] represents an interesting starting point. It focuses on one and two-dimensional irregular to block-cyclic redistribution on a fixed processor set. Specifically, it presents algorithms covering the following cases.

- one-dimensional, irregular to two-dimensional, block-cyclic
- one-dimensional, irregular to one-dimensional, regular
- two-dimensional regular to two-dimensional, regular

Furthermore, there are some more specific forms of redistribution that may also be interesting to examine. Take for example the redistribution of sparse matrices, which are often stored using compressed representations and may need to be treated differently as a result. The paper "Sparse Matrix Block-Cyclic Redistribution" [21] covers this topic with regards to block-cyclic redistributions on a fixed processor set. Also, shape changing redistributions that involve collapsed dimensions could also be studied further, since they are relevant to algorithms like ADI and FFT [7].

In order to properly apply the work this thesis does to modern systems, it could also be interesting to look into redistribution within task-based runtime systems specifically designed to be highly parallel. The papers "Flexible Data Redistribution in a Task-Based Runtime System" [22] and "Evaluating Data Redistribution in PaRSEC" [23] deal with this topic as it pertains to both regular and irregular distributions. The focus is on the task-based runtime system PaRSEC, developed as part of the exascale computing project [24].

Finally, a practical version of this thesis may be relevant as well. While this thesis largely covers the theoretical basis of redistribution, it is important to note that a lot of the tests performed are outdated performance wise. As such, looking into how the algorithms presented in this paper compare to each other on modern architectures using appropriate test cases seems sensible.

List of Figures

2.1 An array of size 18 in a cyclic, block and block-cyclic(3) distribution 4
2.2 An array of size 8×8 distributed in block-cyclic(2, 2) 5
2.3 A 2×2 grid of processors 5
2.4 A 1×4 grid of processors 6

3.1 An array of size 16 distributed in cyclic(1), then cyclic(4) 9
3.2 An array of size 20 distributed in cyclic(4), then cyclic(2) 11
3.3 A visual representation of a FALLS (2, 3, 6, 3), each line representing a block . 13
3.4 A visual representation of a FALLS (3, 5, 6, 2), followed by another FALLS (15, 16, 0, 1) each line representing a block 13
3.5 A visual representation of two FALLS and their intersection 14
3.6 A visual representation of the translation into a local address 16
3.7 An array of size 4×4 colored to depict $SLA_{i,0}$ and $SLA_{i,1}$ 20

4.1 An array of size 20 distributed across 5 processors, first in cyclic(4), then in cyclic(2) 34
4.2 Weighted, bipartite graph of the communication between P and Q 50
4.3 Two circulant matrices - on the left: m = 3, n = 4, on the right: m = 4, n = 3 . . 53
4.4 A generalized circulant matrix 53

List of Tables

3.1 Table that describes SLA's, DLA's, SDPP and DDPP for a given problem 19

4.1 Example of a communication matrix 29
4.2 The trade-offs between single-phase and multi-phase redistribution 32
4.3 Communication matrix for the problem described above 35
4.4 Communication matrix for the problem described above using the algorithm from the multi-phase method 36
4.5 Communication matrix for the problem described above using the synchronized algorithm without scheduling 38
4.6 Communication matrix for the problem described above using the synchronized algorithm with random sending order 39
4.7 Block send schedule for the problem described above, using the synchronized algorithm with scheduling in the reduced system 42
4.8 Process send schedule for the problem described above, using the synchronized algorithm with scheduling in the reduced system 42
4.9 Block send schedule for the problem described above, using the synchronized algorithm with scheduling in the full system 43
4.10 Process send schedule for the problem described above, using the synchronized algorithm with scheduling in the full system 43
4.11 Communication matrix for the problem described above using the caterpillar algorithm 44
4.12 Weights of edges in graph 50
4.13 A possible communication matrix for the problem described above, using the bipartite matching scheme 51
4.14 D_{init} for the problem described above 52
4.15 T corresponding to the earlier D_{init} 52
4.16 A communication matrix for the problem described above, derived using a generalized circulant matrix algorithm 55
4.17 Data transfer cost and schedule computation cost for non all-to-all communication 59
4.18 Data transfer cost and schedule computation cost for all-to-all communication 59
4.19 Parameters for the first set of examples that are tested 60
4.20 Parameters for the second set of examples that are tested 60

Acronyms

ADI	Alternating-direction Implicit. 1, 63
BBC	Basic-Block Calculation. 20, 22, 26
BCC	Basic-Cycle Calculation. 17, 18, 19, 20, 22, 23, 24, 25, 26
CDC	Complete-Dimension Calculation. 20, 22
DDPP	Destination Distribution Pattern Position. 18, 19
DLA	Destination Local Array. 3, 20
FALLS	Family of Line Segments. 12, 13, 14, 15, 17
FFT	Fast-Fourier Transform. 1, 63
GCD	Greatest Common Divisor. 16, 18, 25, 26
HPF	High-Performance Fortran. 17, 45
LCM	Least Common Multiple. 14, 16, 25, 26
LS	Line Segments. 12, 14
PITFALLS	Processor Index Tagged Family of Line Segments. 12, 14, 15, 17, 23, 24, 25, 26
SDPP	Source Distribution Pattern Position. 18, 19
SLA	Source Local Array. 3, 18, 20

Bibliography

[1] D. Walker and S. Otto. “Redistribution of block-cyclic data distributions using MPI”. In: *Concurrency: Practice and Experience* 8.9 (1996), pp. 707–728. DOI: https://doi.org/10.1002/(SICI)1096-9128(199611)8:9<707::AID-CPE269>3.0.CO;2-V. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/%28SICI%291096-9128%28199611%298%3A9%3C707%3A%3AAID-CPE269%3E3.0.CO%3B2-V. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291096-9128%28199611%298%3A9%3C707%3A%3AAID-CPE269%3E3.0.CO%3B2-V.

[2] N. Park, V. Prasanna, and C. Raghavendra. “Efficient algorithms for block-cyclic array redistribution between processor sets”. In: *IEEE Transactions on Parallel and Distributed Systems* 10.12 (1999), pp. 1217–1240. DOI: 10.1109/71.819945.

[3] Y.-C. Chung, C.-H. Hsu, and S.-W. Bai. “A basic-cycle calculation technique for efficient dynamic data redistribution”. In: *IEEE Transactions on Parallel and Distributed Systems* 9.4 (1998), pp. 359–377. DOI: 10.1109/71.667897.

[4] IBM. *Distribution Techniques*. 2021. URL: https://www.ibm.com/docs/en/pessl/5.3.0?topic=distributions-distribution-techniques (visited on 12/13/2024).

[5] Y.-C. Chung and C.-H. Hsu. “Efficient methods for multi-dimensional array redistribution”. In: *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.98EX192)*. 1998, pp. 410–417. DOI: 10.1109/PACT.1998.727299.

[6] S. Blackford. *The Two-dimensional Block-Cyclic Distribution*. 1997. URL: https://netlib.org/scalapack/slug/node75.html (visited on 02/06/2025).

[7] R. Thakur, A. Choudhary, and G. Fox. “Runtime array redistribution in HPF programs”. In: *Proceedings of IEEE Scalable High Performance Computing Conference*. 1994, pp. 309–316. DOI: 10.1109/SHPCC.1994.296659.

[8] G. M. and N. I. “A Framework for Efficient Data Redistribution on Distributed Memory Multicomputers”. In: *The Journal of Supercomputing 20* (2001), pp. 243–246.

[9] S. Ramasulamy and P. Banerjee. “Automatic generation of efficient array redistribution routines for distributed memory multicomputers”. In: *Proceedings Frontiers '95. The Fifth Symposium on the Frontiers of Massively Parallel Computation*. 1995, pp. 342–349. DOI: 10.1109/FMPC.1995.380436.

[10] S. Ramasulamy and P. Banerjee. *Automatic generation of efficient array redistribution routines for distributed memory multicomputers*. 1994. URL: https://ntrs.nasa.gov/citations/19940030437 (visited on 02/06/2025).

[11] S. Ramaswamy, B. Simons, and P. Banerjee. “Optimizations for Efficient Array Redistribution on Distributed Memory Multicomputers”. In: *Journal of Parallel and Distributed Computing* 38.2 (1996), pp. 217–228. ISSN: 0743-7315. DOI: https://doi.org/10.1006/

jpdc.1996.0142. URL: https://www.sciencedirect.com/science/article/pii/S0743731596901422.

[12] R. Thakur, A. Choudhary, and J. Ramanujam. "Efficient algorithms for array redistribution". In: *IEEE Transactions on Parallel and Distributed Systems* 7.6 (1996), pp. 587–594. DOI: 10.1109/71.506697.

[13] L. Prylli and B. Tourancheau. "Fast Runtime Block Cyclic Data Redistribution on Multiprocessors". In: *Journal of Parallel and Distributed Computing* 45.1 (1997), pp. 63–72. ISSN: 0743-7315. DOI: https://doi.org/10.1006/jpdc.1997.1351. URL: https://www.sciencedirect.com/science/article/pii/S0743731597913514.

[14] C.-H. Hsu, S.-W. Bai, Y.-C. Chung, and C.-S. Yang. "A generalized basic-cycle calculation method for efficient array redistribution". In: *Parallel and Distributed Systems, IEEE Transactions on* 11 (Jan. 2001), pp. 1201–1216. DOI: 10.1109/71.895789.

[15] S. D. Kaushik, C.-H. Huang, R. W. Johnson, and P. Sadayappan. "An approach to communication-efficient data redistribution". In: *Proceedings of the 8th International Conference on Supercomputing*. ICS '94. Manchester, England: Association for Computing Machinery, 1994, pp. 364–373. ISBN: 0897916654. DOI: 10.1145/181181.181563. URL: https://doi.org/10.1145/181181.181563.

[16] D. K. Panda. "Optimal Phase Barrier Synchronization in K-ary N-cube Wormhole-routed Systems Using Multirendezvous Primitives". In: 1993. URL: https://api.semanticscholar.org/CorpusID:14256750.

[17] S. Kaushik, C.-H. Huang, J. Ramanujam, and P. Sadayappan. "Multi-phase array redistribution: modeling and evaluation". In: *Proceedings of 9th International Parallel Processing Symposium*. 1995, pp. 441–445. DOI: 10.1109/IPPS.1995.395968.

[18] F. Desprez, J. Dongarra, A. Petitet, C. Randriamaro, and Y. Robert. "Scheduling block-cyclic array redistribution". In: *IEEE Transactions on Parallel and Distributed Systems* 9.2 (1998), pp. 192–205. DOI: 10.1109/71.663945.

[19] E. Jeannot and F. Wagner. "Scheduling Messages For Data Redistribution: An Experimental Study". In: *The International Journal of High Performance Computing Applications* 20 (2006), pp. 443–454. URL: https://api.semanticscholar.org/CorpusID:6046514.

[20] S. Li, H. Jiang, D. Dong, C. Huang, J. Liu, X. Liao, and X. Chen. "Efficient Data Redistribution Algorithms From Irregular to Block Cyclic Data Distribution". In: *IEEE Transactions on Parallel and Distributed Systems* 33.12 (2022), pp. 3667–3677. DOI: 10.1109/TPDS.2022.3166484.

[21] G. Bandera and E. Zapata. "Sparse matrix block-cyclic redistribution". In: *Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing. IPPS/SPDP 1999*. 1999, pp. 355–359. DOI: 10.1109/IPPS.1999.760500.

[22] Q. Cao, G. Bosilca, W. Wu, D. Zhong, A. Bouteiller, and J. Dongarra. "Flexible Data Redistribution in a Task-Based Runtime System". In: *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. 2020, pp. 221–225. DOI: 10.1109/CLUSTER49012.2020.00032.

[23] Q. Cao, G. Bosilca, N. Losada, W. Wu, D. Zhong, and J. Dongarra. “Evaluating Data Redistribution in PaRSEC”. In: *IEEE Transactions on Parallel and Distributed Systems* 33.8 (2022), pp. 1856–1872. DOI: `10.1109/TPDS.2021.3131657`.

[24] E. C. Project. *PARSEC*. 2024. URL: `https://www.exascaleproject.org/research-project/parsec/` (visited on 01/29/2025).