5-10-1996

Contention-free Scheduling of Communication Induced by Array Operations on 2D Meshes

Andreas Bernhard Georg Eberhart
Portland State University

THESIS APPROVAL

The abstract and thesis of Andreas Bernhard Georg Eberhart for the Master of Science in Computer Science were presented May 10, 1996, and accepted by the thesis committee and the department.

COMMITTEE APPROVALS:

[redacted]
Jingke Li, Chair

[redacted]
Andrew P. Tolmach

[redacted]
Michael A. Driscoll

[redacted]
Bradford R. Crain
Representative of the Office of Graduate Studies

DEPARTMENT APPROVAL:

[redacted]
John McHugh, Chair
Department of Computer Science

****

ACCEPTED FOR PORTLAND STATE UNIVERSITY BY THE LIBRARY

by [redacted] on 13 June 1996

ABSTRACT

An abstract of the thesis of Andreas Bernhard Georg Eberhart for the Master of Science in Computer Science presented May 10, 1996.

Title: Contention-Free Scheduling of Communication Induced by Array Operations on 2D Meshes

Whole array operations and array section operations are important features of many data-parallel languages. Efficient implementation of these operations on distributed-memory multicomputers is critical to the scalability and high-performance of data-parallel programs. This thesis presents an approach for analyzing communication patterns induced by array operations and for using run-time information to schedule the message flow. The distributed, dynamic scheduling algorithms guarantee link-contention-free data transfer and utilize network resources almost optimally. They incur little overhead, which is important in order not to reduce the speedup gained by the parallel execution. The algorithms can be used by compilers for the generation of efficient code for array operations. Implemented in a runtime library, they can derive a schedule depending on parameters passed by the parallel application. Simulation results demonstrate the algorithms' superiority to the asynchronous transfer mode that is commonly used for this type of communication.

CONTENTION-FREE SCHEDULING OF COMMUNICATION INDUCED BY ARRAY OPERATIONS ON 2D MESHES

by

ANDREAS BERNHARD GEORG EBERHART

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
COMPUTER SCIENCE

Portland State University
1996

Acknowledgements

There are many which need to be acknowledged with regard to the preparation of this thesis. First and foremost, I thank my advisor, Prof. Jingke Li, for many long hours he spent discussing issues of this work with me. Without his careful guidance and insights this thesis would not have been possible. I also thank the members of my thesis committee, Prof. Andrew Tolmach, Prof. Michael Driscoll, and Prof. Bradford Crain, for their very helpful comments and careful reading. Finally, I would like to thank the tutors of the computer science department for their help on LATEX and system-related questions.

This work was supported in part by a National Science Foundation grant ASC-9123141.

Contents

List of Figures

Chapter 1

Introduction

Massively Parallel Processing (MPP) techniques have advanced rapidly in recent years, as witnessed by the emergence of new systems such as the TMC CM5, the Intel Paragon, the Cray T3D, the IBM SP-2, and the SGI Power Challenge. These systems all have powerful processors, large storage space, and fast communication hardware, providing the required high performance for computationally intensive scientific and engineering applications. MPP systems promise to achieve computational power that exceeds the performance of traditional vector-supercomputers by several orders of magnitude. However, there are some obstacles for applying MPP systems: programming these machines is extremely difficult and there is a lack of software tools that simplify the programming while using the immense raw computational power efficiently.

1.1 Distributed-Memory Multicomputers

Figure 1.1 shows an example of a 2D mesh parallel architecture, which is the focus of this thesis. The circles represent the processors with their local memory. These units are called *nodes*. Each of these nodes executes a program which is stored locally. The programs can be identical on each node, but, the data processed is different. This category of multicomputers is also called MIMD (Multiple Instruction Multiple Data).

Results of local computations can be exchanged via the interconnection network using the *message-passing* approach. A node can send data along with the destination node's location to an attached communication unit, called router. A message path

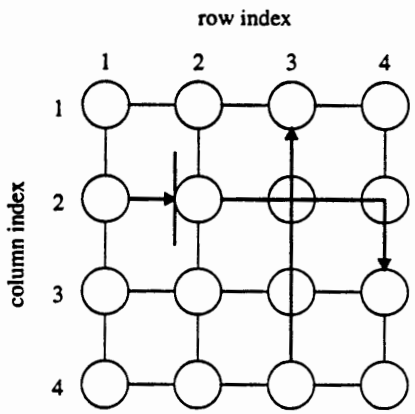


Figure 1.1: 4×4 mesh network.

is established and the data is sent to the destination node's router from where it is forwarded to the local memory.

1.2 Data-Parallel Languages

Data-parallel languages such as Fortran 90 and High Performance Fortran (HPF)[4, 15, 29] greatly simplify programming on MPP systems and are widely seen as an effective means for developing portable application programs. A major feature of these languages is the *array operation*, in which an element-wise operation $\circ$ over whole arrays or array sections is specified simply as $A \circ B$ or $A(l{:}h{:}s) \circ B(l'{:}h'{:}s')$ if the array section notation (see Chapter 2) is used. With these expressions, array-based parallel algorithms can be expressed clearly and concisely. The following Fortran 90 example demonstrates the use of array sections:

```
program matrix
integer, dimension(1000,1000) :: a, b
integer, dimension(1000) :: c
a = (a + b) * 2
c = a(2,:)
stop
end program
```

The simple statement $a = (a + b) * 2$ adds two matrices and multiplies the resulting matrix by two. The second statement copies the second column[1] of matrix a to the vector c. Due to the array operations, the programming style is closer to the code for single processor systems. Programmers do not have to worry about where the arrays are located and how they are distributed across the processors. No explicit send and receive statements are necessary, decoupling the program from the architecture. Compilers can generate different code with different message-passing steps for specific topologies. The lack of portability has been a great problem for many parallel applications. Array operations, make programs easier to read, and more importantly, easier to compile for parallel execution.

1.3 Translation of Array Statements

When a data-parallel program is compiled to an MPP system, the data arrays in the program are often decomposed and distributed over the system's distributed memory modules. Several languages provide the programmer with directives for specifying the array alignment and distribution; others rely on the compiler to do the job. Due to the data distribution, array elements involved in an array operation may be scattered over different locations. To perform the operation, corresponding elements must be brought together to the same processor, resulting in data movement across the interconnection network. An analysis shows that even with very regular

[1]Fortran 90 represents matrices in column major order. In this thesis, row indices precede column indices.

data alignment and distribution, a simple array operation can induce very complex communication patterns.

Assume the array A in the example above is distributed evenly over 10×10 processors, each holding a submatrix of size 100×100. When the target program is generated the compiler generates code on node (r, c) for the statement $a = (a + b) * 2$, which corresponds to:

for $i = 0$ to 99
  for $j = 0$ to 99
    $a(r \cdot 100+i, c \cdot 100+j) = (a(r \cdot 100+i, c \cdot 100+j)+b(r \cdot 100+i, c \cdot 100+j)) \cdot 2$
  end for
end for

The compiler must make sure that before this code is executed on node (r, c), the corresponding data of array b (i.e. all elements with indices from $100 \cdot r$ to $100 \cdot r + 99$ and $100 \cdot c$ to $100 \cdot c + 99$) is available on the node. If this is not the case, the compiler must (1) identify the array elements involved in the operation, (2) determine the required communication, and (3) generate the appropriate code that will send the messages when the program is executed on a parallel machine. This is the place where the communication algorithms presented in this thesis are applicable.

Some communication can be avoided if programmers use a clever data distribution. However, similar to programming parallel computers, this process is very difficult. Since the main purpose of data-parallel languages is to provide simple and user-friendly environment for programming parallel computers, the scheduling algorithms should be prepared to transfer data between arrays with arbitrary data distribution parameters.

1.4 Handling the Communication

A simple approach for handling communication induced by an array operation (or by any statement in a program) is to generate messages for bringing the data from the sources to their destinations directly, without evaluating the communication

pattern. This method is called asynchronous or unscheduled transfer. As observed by many users of MPP systems, a large number of unorchestrated concurrent messages can cause high levels of resource contention, message blockage, buffer overflow, and even deadlock [14]. To rectify this, many data-parallel language compilers perform communication optimizations, including the identification of special collective communication patterns such as broadcast, multicast, and reduction[19, 25, 27], which are either one-to-many or many-to-one patterns. The analysis of the communication induced by array operations in Chapters 3 and 4 will show that this type of communication does not fall into these categories.

For arbitrary and totally unstructured communication patterns, an alternative approach is to analyze and schedule the message flow. The goal is to optimize the usage of the limited network resources. A typical scheduling algorithm decomposes a complex pattern into simpler patterns and carries them out in separate steps. Efficient implementation of collective communication is an example of this approach[14, 20, 31].

In this thesis, the problem of optimizing communication induced by array operations is studied and distributed, dynamic scheduling algorithms for these communication patterns are proposed. The algorithms have several distinct features: (1) they produce message-passing steps which are link contention-free, and in many cases, the schedules can be proven to be optimal with respect to the number of message-transmission steps; (2) no global information exchange is required at runtime, thus they incur little overhead; (3) they derive a message-passing schedule from array distribution and array operation statements, hence they are suitable to be used by compilers and in libraries. If all parameters are constants, then the algorithms are executed at compile-time. The generated schedule is then hard-wired in the program code with statements like:

```
send message 3 in step 5
```

If some parameters, for example the array section stride, is variable and only known at run-time, then the compiler places a call to the scheduling algorithm in the code. The algorithm then determines the schedule at run-time with low software overhead. (4) The algorithms have a modular structure, allowing convenient fine-tuning for specific interconnection networks. A simulation study for a wormhole-routed network

was conducted, showing that there is a significant performance improvement by using scheduled messages.

1.5 Outline of the Thesis

In Chapter 2, a description of the supported hardware platform is given. Furthermore, array operations and possible data alignments on processor networks are explained in detail.

With this information, the communication induced by array operations can be examined. This is done in Chapter 3. At first, strong restrictions are imposed in order to obtain easy scheduling solutions. Then step by step, more general cases are solved by extending the algorithms for the restricted cases. For each phase, simulation results are presented that evaluate both the scheduling solutions and the unscheduled transfer, demonstrating the effects of link-contention.

Chapter 4 briefly describes how the results from Chapter 3 are used to extend the scheduling algorithms to transfers between arrays with transposed alignments.

Several simulations results for the different cases and comparisons to the unscheduled transfer are presented in Chapter 5. This also includes a detailed analysis of some worst case examples that occur during an unscheduled transfer.

A summary of the results and possible future research is given in Chapter 6.

Chapter 2

Problem Formulation

This chapter defines the hardware platform and the data distribution of data-parallel languages across the processors.

2.1 Platform

This study considers mesh networks, where the nodes are arranged in a matrix. Each interior node is connected to four neighbors; the border nodes are connected to two or three neighbors (Figure 1.1). Each connection between neighboring nodes consists of two channels that can be used simultaneously: one transports messages from left to right, the other one from right to left. The analogous statement can be made for the vertical connections.

The network is assumed to be a multiport architecture, which means that nodes can send and receive messages at the same time. For our algorithms, however, it is only necessary that a node can send and receive one pair of messages concurrently. If this is not the case, the algorithms are still applicable but will not be as effective (see Section 2.1.4).

2.1.1 Network Routing

The routing algorithm defines how messages are propagated through the interconnection network from the sending node to the destination node. It is assumed that the underlying network uses *dimension-ordered routing*. With dimension-ordered routing, a message travels continuously in one dimension of the network, then switches to an-

other dimension. For 2D mesh networks, dimension-ordered routing means either *X-Y routing* (traveling in the X (horizontal) dimension first, then in the Y (vertical) dimension) or *Y-X routing* (the opposite). As indicated in Figure 1.1, the algorithms will be presented with respect to X-Y routing, but they can be easily converted for Y-X routing networks by exchanging the row and column indices in the scheduling sequence.

2.1.2 Link Contention and Network Switching

This section explains how the message path is established. Since not every pair of nodes has a unique connection, channels are used by several different nodes. This can cause conflicts if two messages request a channel at the same time. The way this *link-contention* is resolved depends on the switching strategy. Virtual cut-through, circuit switching, and wormhole routing[24, 32] are considered.

Circuit switching tries to establish the complete path to the destination before sending the message. If several nodes try to occupy the same channel, only one succeeds. The other nodes have to retreat and retry. Heavy contention in the network causes nodes to retreat several times before they are able to send the message, causing significant overhead.

Virtual cut-through works similarly to circuit switching. However, if a required channel is occupied, the message gets buffered at the node adjacent to the busy channel until it is released. Depending on the size of the message and the start-up latency, frequent buffering causes large overheads.

Wormhole routing partitions a message into fixed size packets. Those packets are then sent sequentially along the path to the destination node. When a message worm requests a channel that is already in use, the worm blocks. It remains in the network, holding all the channels that it has acquired. These channels then remain blocked for other worms without actually being used for message transmission. In Figure 1.1 the message from node $(2,1)$ is blocked because it requests the channel from node $(2,2)$

to node (2,3). Note that the other two messages do not conflict even though their paths cross each other.

All of these switching strategies in some way exploit the benefits of pipelining, which makes them superior to the store-and-forward strategy. The difference lies in the conflict resolution. Wormhole routing has become the most popular among this class of strategies since it allows useful extensions such as virtual channels[10]. Most of the recent MPP systems use the wormhole routing strategy and therefore, this thesis reports simulations on networks with this strategy to evaluate the scheduling algorithms.

2.1.3 Communication Model

A simple model is used, which assumes the time to send a message of L units to be: $\alpha + L\beta$. Wormhole-routing, virtual cut-through, and circuit switching transfer the data in a pipelined fashion through the connection network. The elapsed time from the source node's request to send a message to the arrival of the first data elements at the destination node is described as the *start-up* cost (latency) α. Due to the pipelined transfer, the distance between the nodes has no major effect on the total transfer time unless the message is very small. In this context, β is defined as the time to send one data unit across the network. This is the *reciprocal bandwidth.* Thus, β depends on both the channel bandwidth and the definition of the data unit.

2.1.4 Synchronization

Some kind of step synchronization must be available in the network, so that all processors can execute statements of the form

if $step = i$ then ...

guaranteeing that messages transmitted at time $i - 1$ are no longer in the network, unless they were blocked.

It is important to note that any kind of scheduling scheme needs the notion of steps; otherwise there is only the possibility of sending unsynchronized messages with

the chance of network contention. Some MPP systems support efficient synchronization, which enables them to obtain the full advantage of communication scheduling. For example, both the Cray T3D and the TMC CM-5 have specialized hardware for supporting synchronization[7, 26]. Recently, Hall[16, 17, 18] has shown the design and implementation of a simple secondary coordination processor system which can be attached to an MPP system to speed up synchronization and other global operations.

For systems that rely on regular message-passing for synchronization, however, the benefit of communication scheduling will be reduced by the cost of synchronization. The duration of one synchronized step depends on the amount of data transferred. When large messages are synchronized, then the relative cost of synchronization is comparably small, even if message-passing is used.

Besides the implementational cost of synchronisation, there is also an overhead involved if the messages sent during one step have different transmission times. This can be caused by non-uniform message sizes, different start-up latencies due to varying path lengths, or node contention. Messages from the next step always have to wait until all messages have left the network. Therefore, if the maximum transmission time is much longer than the average time, sending synchronized messages is less effective compared to the asynchronous case.

This thesis shows, that in our setting, almost all messages are of the same size. Furthermore, for data-parallel applications, it is likely that large arrays are distributed over the processors, causing large messages to be sent. Thus, varying start-up latencies do not have a significant impact on the transmission times. Finally, node contention can have an impact on single-port architectures. In Section 3.1.4, some ideas to handle the problem with this architecture are presented.

2.2 Language Parameters

2.2.1 Array Operations

The triplet notation $A(l{:}h{:}s)$ allows convenient denotation for a subset of A, called *array section*:

$$A(l:h:s) = \{A(l+is) : 0 \leq i \leq (h-l)/s, s > 0\}$$

The triplet consists of the lower-bound, the upper-bound, and the stride of the array section (the stride can be omitted if it is 1). The upper-bound h is normalized so that $h = l + is$ holds for some integer i.

Consider array operations of the following forms:

$$A(l{:}h{:}s) \quad \circ \quad B(l'{:}h'{:}s') \tag{2.1}$$

$$A(l_r{:}h_r{:}s_r, l_c{:}h_c{:}s_c) \quad \circ \quad B(l'_r{:}h'_r{:}s'_r, l'_c{:}h'_c{:}s'_c) \tag{2.2}$$

$A(l{:}h{:}s)$ and $B(l'{:}h'{:}s')$ are two conforming array sections (one-dimensional), and so are $A(l_r{:}h_r{:}s_r, l_c{:}h_c{:}s_c)$ and $B(l'_r{:}h'_r{:}s'_r, l'_c{:}h'_c{:}s'_c)$ (two-dimensional). $\circ$ represents an element-wise operation.

The two expressions (2.1) and (2.2) specify element-wise operations on the corresponding (at the same relative position in the sections) elements of two array sections. Without loss of generality, it is assumed that the computation takes place at the location of the second operand, i.e. on the processors where B's section resides.[1] Therefore, A's section must be *transferred* to the locations of B's section (if it is not already there).[2]

[1] Computation location for an array operation can be optimized using approaches such as described in [6]. Once the location is determined, the scheduling algorithms are applicable.

[2] Solutions for the transfer of an array section can also be applied to implement the *redistribute* command, which changes the distribution of an array.

	$P_{-,0}$			$P_{-,1}$			$P_{-,2}$		
$P_{0,-}$	0,0	...	0,9	0,10	...	0,19	0,20	...	0,29
	...	...	...	...	...	...	...	...	...
	3,0	...	3,9	3,10	...	3,19	3,20	...	3,29
$P_{1,-}$	4,0	...	4,9	4,10	...	4,19	4,20	...	4,29
	...	...	...	...	...	...	...	...	...
	7,0	...	7,9	7,10	...	7,19	7,20	...	7,29
$P_{2,-}$	8,0	...	8,9	8,10	...	8,19	8,20	...	8,29
	...	...	...	...	...	...	...	...	...
	11,0	...	11,9	11,10	...	11,19	11,20	...	11,29
$P_{3,-}$	12,0	...	12,9	12,10	...	12,19	12,20	...	12,29
	...	...	...	...	...	...	...	...	...
	15,0	...	15,9	15,10	...	15,19	15,20	...	15,29

Figure 2.1: Two-dimensional block-distribution. Each node of the 4×3 grid holds a 4×10 subarray.

2.2.2 Data Distribution

Block-Distributions

In a block-distribution, an array is distributed over p processors each storing one block of k consecutive data elements. k is the blocksize of the distribution. If the array has n elements, then the following condition holds: $n = pk$. The function $\mathcal{P}(i)$ describes the location of the *processor* holding $A(i)$. It is defined as:

$$\mathcal{P}(i) = i \text{ div } k$$

If two-dimensional arrays are distributed across a processor mesh, then the distribution must be specified for each dimension. The parameters for the row and column distribution are (p_r, k_r) and (p_c, k_c). Figure 2.1 shows the layout of the array across the processors. The functions $\mathcal{P}_r(i)$ and $\mathcal{P}_c(i)$ are defined as $\mathcal{P}(i)|_{k=k_r}$ and $\mathcal{P}(i)|_{k=k_c}$. All parameters or functions referring to the *destination* array section are marked with prime. For example $\mathcal{P}'_r(i)$ is defined as $\mathcal{P}(i)|_{k=k'_r}$ and returns the row-index of the processor holding $B(i)$.

The only assumption on Equations 2.1 and 2.2 is that the stride cannot be larger

than the blocksize. This guarantees that at least one section element is located on each node.

Two Level Mappings

In a two-level mapping, an auxiliary cartesian grid called *template* is used. Arrays are aligned to templates, and templates are distributed across the processors. This allows one to define an additional offset a and stride b for the mapping of the templates to the processor. If A is aligned with a two-level mapping, then $A(l{:}h{:}s)$ generates the same distribution as $A(a+lb : a+hb : sb)$ with A mapped directly to the processors. Due to this observation, the algorithms can be applied to two-level mappings as well.

Dimension Alignment

Two-dimensional arrays can be aligned to the processor grid in two ways: array rows to processor rows or array rows to processor columns. Chapter 3 deals with the transfer between arrays that have the same dimension alignment and Chapter 4 presents solutions for the other case.

Chapter 3

Scheduling Solutions for Identical Alignments

The following three sections cover transfers between block-distributed array sections with identical dimension alignment. The problem is split into three difficulty levels. The transfer subclass of shifts is analyzed in Section 3.1. In a shift operation, each source node sends out a single message to its destination. This approach is generalized to regular transfers in Section 3.2. Rather than single source destination pairs, groups of nodes perform an all-to-all communication during a regular transfer. Finally all restrictions are dropped in Section 3.3, which deals with transfers between arbitrary block-distributions. Note that shifts are a subset of regular transfers, which in them are a subset of transfers between arbitrarily block-distributed array sections. Thus, the more general scheduling algorithms will still work for the special cases.

3.1 Identical Block-Distributions

First, we define regular array sections. They have the property that all nodes, including those holding the first and last section element, store the same number of data elements.

Definition With respect to a (p, k) block-distribution, an array section $A(l : h : s)$ is *regular* if $(l \bmod k = (h + s) \bmod k < s)$ and $(s|k)$.

If an array A is distributed with four elements per node ($k = 4$), then $A(1 : 11 : 2)$

processor 0				processor 1				processor 2			
0	**1**	2	**3**	4	**5**	6	**7**	8	**9**	10	**11**

is a regular section. Two divides four and therefore, $k/s = 4/2 = 2$ elements are located on each node. The conditions $1 \bmod 4 = 1 < 2$ and $(11 + 2) \bmod 4 = 1 < 2$ make sure that this statement is true for the first and last processors, too. The array section $A(3 : 11 : 2)$

processor 0				processor 1				processor 2			
0	1	2	**3**	4	**5**	6	**7**	8	**9**	10	**11**

is not regular, because $3 \bmod 4 = 3 \not< 2$. This indicates that the block of the first processor contains only one element. Array section $A(2 : 11 : 3)$

processor 0				processor 1				processor 2			
0	1	**2**	3	4	**5**	6	7	**8**	9	10	**11**

is not regular either, because three does not divide four, so that some processors hold $\lfloor k/s \rfloor = 1$ element whereas others store $\lceil k/s \rceil = 2$.

In this section the easiest form of array section transfers, the shift, is presented:

Definition A transfer of a 1-dimensional regular array section $A(l:h:s)$ to another regular array section $B(l':h':s')$ is called a *shift* if $k/s = k'/s'$, which means that all nodes holding part of either array section store the same number of data elements. For 2-dimensional array sections, a transfer is called shift if the transfers in both dimensions are shifts and the arrays are aligned in the same way.

3.1.1 Communication Pattern

Given the locations of the nodes holding the upper left ($A(l_r, l_c)$) and lower right ($A(h_r, h_c)$) array elements, it can be deduced that $\mathcal{N}_r \times \mathcal{N}_c$ source nodes perform the shift, where:

$$\mathcal{N}_r = \mathcal{P}_r(h_r) - \mathcal{P}_r(l_r) + 1, \qquad \mathcal{N}_c = \mathcal{P}_c(h_c) - \mathcal{P}_c(l_c) + 1 \qquad (3.1)$$

The destination processor grid is of the same size since the same number of data elements is stored on both the source and the destination nodes: $\mathcal{N}'_r \times \mathcal{N}'_c = \mathcal{N}_r \times \mathcal{N}_c$. As illustrated in Figure 3.1, each node sends out one message to one destination at the same relative position in the destination grid. The shifting offset is:

$$\mathcal{P}'_r(l'_r) - \mathcal{P}_r(l_r) \text{ vertically and } \mathcal{P}'_c(l'_c) - \mathcal{P}_c(l_c) \text{ horizontally.}$$

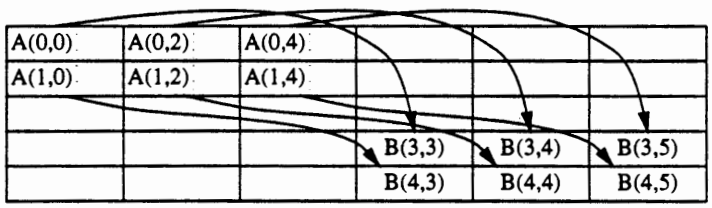


Figure 3.1: Shift of the array section $A(0:1, 0:4:2)$ to $B(3:4, 3:5)$. All blocksizes are one except for $k_c = 2$. The shifting offset is $(3, 3)$.

The following method is used to develop an optimal algorithm: first, the communication *bottlenecks* are identified. Then *lower bounds* for the number of communication steps needed are established by counting the messages that have to (sequentially) pass the bottlenecks. The source nodes of messages that get routed through the same bottleneck are combined into *conflicting sets*. Finally, a scheduling algorithm is developed, which assures that nodes out of the same conflicting set never send their messages in the same step. This algorithm guarantees *contention-free* communication in the network.

3.1.2 Establishing a Lower Bound

At first, the case in which the source area and destination area do not overlap is considered. This can be characterized by the conditions

$$|\mathcal{P}'_r(l'_r) - \mathcal{P}_r(l_r)| \geq \mathcal{N}_r \text{ and } |\mathcal{P}'_c(l'_c) - \mathcal{P}_c(l_c)| \geq \mathcal{N}_c$$

indicating that the shifting offset in each dimension is larger than the number of source nodes.

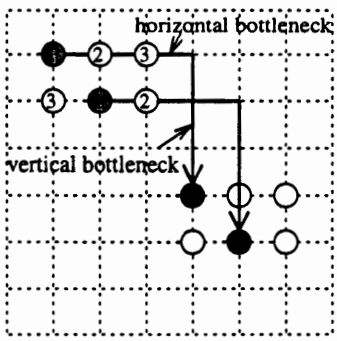


Figure 3.2: Step 1 of 3 of the collision free shift from Figure 3.14. Sending the diagonals concurrently avoids contention in the network while allowing maximum parallelism for the transmission.

In the example shown in Figure 3.2, the data located at the six nodes in the top left corner are shifted. Every message travels 3 columns and 3 rows. All the messages sent from nodes in the first row use the channel marked "horizontal bottleneck," according to the X-Y routing algorithm. Since each channel can only allow one message to pass in each step, the three messages from the first row must go through the horizontal bottleneck sequentially. In the same way, the messages from the nodes in the first column get routed through the channel marked "vertical bottleneck." Similar bottlenecks exist for the other rows and columns.

For each bottleneck channel, a *conflicting set* is defined, which consists of nodes whose messages must go through the channel. The cardinality of the largest conflicting set hence is a lower bound for the number of transmission steps.

As illustrated in Figure 3.2, the source nodes belonging to the same row form a conflicting set; so do the nodes belonging to the same column. Therefore, a lower bound for the number of steps required for the transmission is $\max(\mathcal{N}_r, \mathcal{N}_c)$.

3.1.3 Diagonal Scheduling Scheme

Any communication schedule that matches a lower bound is *optimal.* One such schedule is proposed here. Consider the nodes on a diagonal line. They each belong

to a different row, hence their messages do not share any horizontal channels. In addition they each belong to a different column, which implies that the destinations also belong to a different column. Therefore, the messages from a diagonal line do not share any column channels either.

The following scheduling algorithm decomposes the source nodes into $\max(\mathcal{N}_r, \mathcal{N}_c)$ diagonal sets, and transfers the data in an optimal number of steps (*row* and *col* are my row and column IDs):

```
forall (source nodes) in parallel do
    if col − row + 1 ≤ 0
        send my message to destination in step col − row + 1 + max(N_r, N_c)
    else
        send my message to destination in step col − row + 1
    end if
end forall
```

This strategy invokes the sending order shown in Figure 3.2, with a total of three steps.

In general, the source area and the destination area may overlap. Figure 3.3 shows the shifting of 3×5 source nodes by an offset of 2 rows and 3 columns. The lower-right part of the source area overlaps with the upper-left part of the destination. Message traffic increases in the overlapped area because the nodes there have double identities: they are both senders and receivers.

However, contrary to the intuition that increased message traffic would increase communication delay, the cost of shifting can actually be reduced if there is overlap. As illustrated in Figure 3.3a, messages from sources that are two rows or three columns apart do not conflict.

With this observation and the diagonal scheduling approach, the following strategy to handle the general shifts is derived: divide the source nodes into *sections* of size $|\mathcal{P}'_r(l'_r) - \mathcal{P}_r(l_r)| \times |\mathcal{P}'_c(l'_c) - \mathcal{P}_c(l_c)|$ (assuming $|\mathcal{P}'_r(l'_r) - \mathcal{P}_r(l_r)| \leq \mathcal{N}_r$ and $|\mathcal{P}'_c(l'_c) - \mathcal{P}_c(l_c)| \leq \mathcal{N}_c$) and then use the diagonal scheduling approach developed for the simple case on the sections (Figure 3.3b). The sections of size 2×3 are visu-

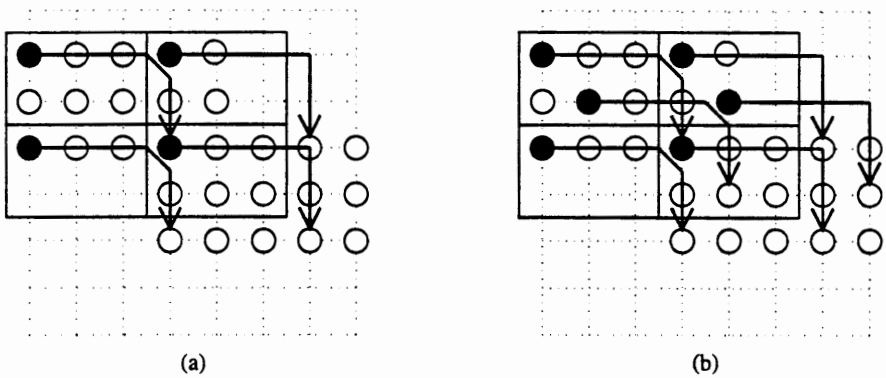


Figure 3.3: Shift with a smaller offset. The scheme from Figure 3.2 is applied on the grid of 2×3 submatrices in parallel. Step 1 out of 3.

alized by the solid lines. In general, the size of each section is $\max(1, \min(|\mathcal{P}'_r(l'_r) - \mathcal{P}_r(l_r)|, \mathcal{N}_r) \times \max(1, \min(|\mathcal{P}'_c(l'_c) - \mathcal{P}_c(l_c)|, \mathcal{N}_c)$ since it is bounded by the minimal size one and the total number of source nodes in the row or column. If the source and the destination area do not overlap, then only one section covers all source nodes yielding the scheduling order described in the beginning of this section. For the convenience of presentation, the notation $[x]_b^a$ is introduced:

$$[x]_b^a = \begin{cases} a & \text{if } x < a \\ x & \text{if } a \leq x \leq b \\ b & \text{if } b < x \end{cases} \tag{3.2}$$

and define the section sizes

$$sec\_ver = [|\mathcal{P}'_r(l'_r) - \mathcal{P}_r(l_r)|]^1_{\mathcal{N}_r}, \quad sec\_hor = [|\mathcal{P}'_c(l'_c) - \mathcal{P}_c(l_c)|]^1_{\mathcal{N}_c} \tag{3.3}$$

Algorithm 1 implements the strategy described above. It is to be run by every processor in a distributed manner. The for loop goes through all the communication steps with the *sending condition* triggering the node's message transfer.

Theorem 1 *Algorithm 1 shifts the data of $\mathcal{N}_r \times \mathcal{N}_c$ source nodes in* $\max(sec\_ver, sec\_hor)$ *link-contention-free steps, which is optimal.*

Algorithm 1 (Diagonal Scheduling Scheme on Sections)
Scheduling algorithm for shifts. The two parameters row and col denote the node's location in the network

```
Main Program
    if this_node is a source node
        determine sec_ver and sec_hor                    /* use Equation 3.3 */
        section_row = (row − P_r(l_r)) mod sec_ver        /* determine */
        section_col = (col − P_c(l_c)) mod sec_hor        /* relative position */
        diagonal_num = section_row − section_col + 1
        if diagonal_num ≤ 0
            diagonal_num = diagonal_num + max(sec_ver, sec_hor)
        end if
        for step = 1 to max(sec_ver, sec_hor)             /* steps needed */
            if step = diagonal_num                        /* send condition */
                send_message_to(row + P'_r(l'_r) − P_r(l_r), col + P'_c(l'_c) − P_c(l_c))
            end if
        end for
    end if
end Main
```

Proof: See appendix.

3.1.4 Avoiding Node Contention

For Algorithm 1 it was assumed, that concurrent sending and receiving at a node does not cause any delay. On some architectures however, the layout of the router does not support this feature. In those cases node contention might cause some overhead [42].

We present an idea on how to modify Algorithm 1 so that it avoids both link and node contention without any additional overhead. Algorithm 1 divides the source nodes into smaller sections of size $sec\_ver \times sec\_hor$ and transmits all messages in $\max(sec\_ver, sec\_hor)$ steps. In case $sec\_ver \neq sec\_hor$, the matrix can be divided into square sections of size $\max(sec\_ver, sec\_hor) \times \max(sec\_ver, sec\_hor)$ without requiring additional steps. Using this different section avoids node contention, because inside a section the diagonal sending is no longer the diagonal receiving. Figure 3.4 shows the

example from Figure 3.3 with a section size of 3×3 rather than 2×3. The node in the top left corner of the destination area does not send a message with the adjusted section size. In both examples, three steps are needed to complete the shift.

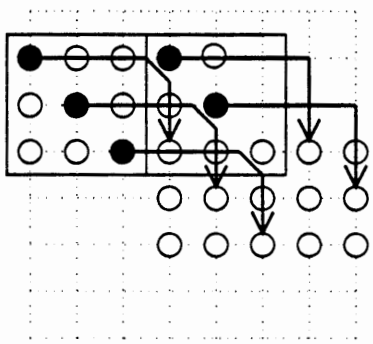


Figure 3.4: Avoiding both link and node contention, by adjusting the section size. The example from Figure 3.3 has a modified section size of 3×3. Step 1 out of 3.

If ($sec\_ver = sec\_hor$), then the section must be changed to ($sec\_ver + 1 \times sec\_ver$). This requires an extra step as well as the case, where $sec\_ver = 0$ or $sec\_hor = 0$. In those cases the section sizes are set to ($1 \times sec\_hor + 1$) and ($sec\_ver + 1 \times 1$) respectively.

3.2 Regular Block-Distributions

This section extends the results from the previous section to regular transfers, where nodes can have more than one source or destination.

Definition A transfer of a regular array section $A(l:h:s)$ to another regular array section $B(l':h':s')$ is called a ***regular transfer***, if one section's data elements per node is a multiple of the other's ($(k/s)|(k'/s')$ or $(k'/s')|(k/s)$).

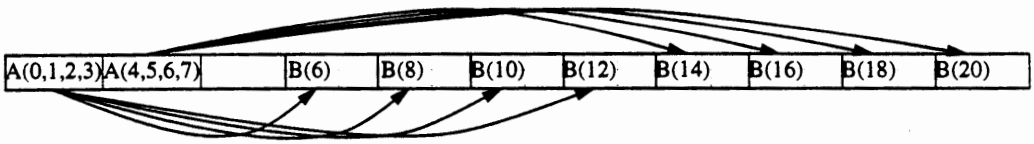


Figure 3.5: Transfer of the array section $A(0{:}7)$ to $B(6{:}20{:}2)$. A and B are distributed with block-sizes 4 and 2. Each source node must send messages to a cluster of $C' = 4$ destination nodes.

3.2.1 Communication Pattern

For regular transfers, each source node holds k/s data elements and each destination node holds k'/s' elements. The induced communication falls into two cases: (1) if $k/s \geq k'/s'$, then each source node *scatters* its data to $\frac{k/s}{k'/s'}$ adjacent destination nodes (Figure 3.5); (2) if $k/s < k'/s'$, then the data from $\frac{k'/s'}{k/s}$ source nodes is *gathered* to a single destination node. In other words, a 1D regular array operation induces either a collection of $1 \rightarrow \frac{k/s}{k'/s'}$ communication patterns or a collection of $\frac{k'/s'}{k/s} \rightarrow 1$ communication patterns. These patterns are called *base communication patterns* for the array operation. With

$$C = \lceil \frac{k'/s'}{k/s} \rceil \quad \text{and} \quad C' = \lceil \frac{k/s}{k'/s'} \rceil \tag{3.4}$$

the two base communication patterns for the 1D array operation can both be represented as $C \rightarrow C'$. In Figure 3.5, for example, the pattern is $\lceil \frac{2/2}{4/1} \rceil \rightarrow \lceil \frac{4/1}{2/2} \rceil = 1 \rightarrow 4$.

The communication pattern induced by a 2D array operation is basically a composition of two 1D communication patterns, one for each dimension. Consequently, there are four basic communication patterns:

- *scatter/scatter* $(1 \times 1 \rightarrow \frac{k_r/s_r}{k'_r/s'_r} \times \frac{k_c/s_c}{k'_c/s'_c})$
- *scatter/gather* $(1 \times \frac{k'_c/s'_c}{k_c/s_c} \rightarrow \frac{k_r/s_r}{k'_r/s'_r} \times 1)$
- *gather/scatter* $(\frac{k'_r/s'_r}{k_r/s_r} \times 1 \rightarrow 1 \times \frac{k_c/s_c}{k'_c/s'_c})$
- *gather/gather* $(\frac{k'_r/s'_r}{k_r/s_r} \times \frac{k'_c/s'_c}{k_c/s_c} \rightarrow 1 \times 1)$

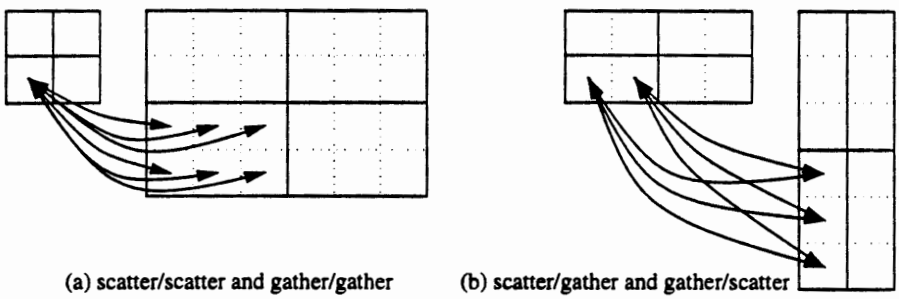


Figure 3.6: Communication pattern of a regular transfer for each of the four cases. The squares represent nodes and solid lines mark source and destination nodes of a base pattern that communicate *exclusively* with each other. The processors on the left in Figure (a) hold six times as much data of the array section as the processors on the right, resulting in a $1 \times 1 \leftrightarrow 2 \times 3$ pattern. In Figure (b) the left nodes have three times the elements vertically and half of the elements horizontally compared to the right nodes. The later distributions yield a $1 \times 2 \leftrightarrow 3 \times 1$ pattern.

Figure 3.6 illustrates these cases. Extending the notation in (3.4) to row and column parameters, it is possible to represent all four basic patterns with one formula: $\mathcal{R} \times \mathcal{C} \rightarrow \mathcal{R}' \times \mathcal{C}'$, where:

$$\mathcal{R} = \lceil \frac{k_r'/s_r'}{k_r/s_r} \rceil, \quad \mathcal{R}' = \lceil \frac{k_r/s_r}{k_r'/s_r'} \rceil, \quad \mathcal{C} = \lceil \frac{k_c'/s_c'}{k_c/s_c} \rceil, \quad \mathcal{C}' = \lceil \frac{k_c/s_c}{k_c'/s_c'} \rceil \tag{3.5}$$

3.2.2 Extended Diagonal Scheduling Scheme

With the information about the communication pattern derived in the previous section, the message traffic on a mesh network can be analyzed and scheduled by simply extending the concept presented in Section 3.1. Figure 3.7 shows how to generalize the diagonal scheme from single nodes to *regions* of nodes for handling array section operations. The size of the source and destination regions is set to $\mathcal{R} \cdot \mathcal{C}' \times \mathcal{C} \cdot \mathcal{R}$ and $\mathcal{R}' \cdot \mathcal{C}' \times \mathcal{C}' \cdot \mathcal{R}$ (explained below). The following text shows that, similarly to the single-element case, regions located on a "diagonal line" can send out messages concurrently, which are guaranteed to be collision-free since the regions consist of base patterns with disjoint destinations.

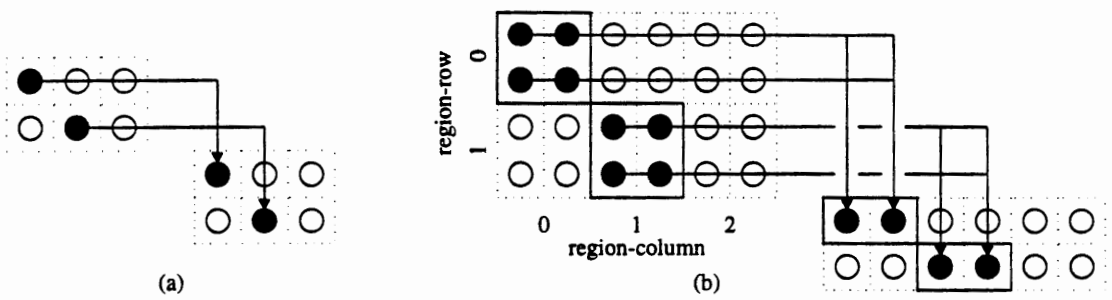


Figure 3.7: Figure (a) shows the first of three steps of the single element case. Messages of nodes from different rows or columns do not conflict. In Figure (b) this diagonal scheme is generalized to the case where the source nodes become regions of four nodes (solid boxes) performing two 1×2 base patterns (dotted lines). Two base patterns are grouped into regions in order to have two links in each dimension. Again, messages sent from regions (solid boxes) from different rows and columns do not conflict. Thus, the diagonal scheme is applied to regions rather than single nodes.

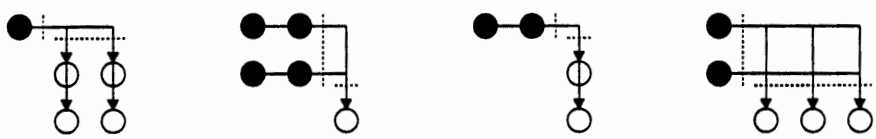


Figure 3.8: The dotted lines show the links available for scatter/scatter, gather/gather, scatter/gather, and gather/scatter (left to right).

Regions In Section 3.2.1, the four base communication patterns for a 2D array operation were represented by the single formula $\mathcal{R} \times \mathcal{C} \rightarrow \mathcal{R}' \times \mathcal{C}'$. Since X-Y routing is used, for a basic communication pattern, there are $\mathcal{R}$ horizontal links and $\mathcal{C}'$ vertical links available for its data transfer (Figure 3.8). Potentially, $\min(\mathcal{R}, \mathcal{C}')$ messages can be transferred concurrently without collision. However, if $\mathcal{R} \neq \mathcal{C}'$, then some links will be wasted. Figure 3.7b shows the induced communication pattern from a 2D array operation, where the base communication pattern is $2 \times 1 \rightarrow 1 \times 1$. Since $\mathcal{C}' = 1$, one base pattern by itself would not allow any concurrent message transfer and would always leave one of the two horizontal links unused.

In order to achieve optimal link utilization, the same number of links should be

available in both horizontal and vertical dimensions for a concurrent communication step. This can be accomplished by grouping $\mathcal{C}' \times \mathcal{R}$ adjacent base patterns into a *region*.[1]

The extended communication pattern becomes $\mathcal{R} \cdot \mathcal{C}' \times \mathcal{C} \cdot \mathcal{R} \rightarrow \mathcal{R}' \cdot \mathcal{C}' \times \mathcal{C}' \cdot \mathcal{R}$ and it has $\mathcal{C}'\mathcal{R}$ links in both dimensions. Section 3.2.4 shows how an optimal number of $\mathcal{C}'\mathcal{R}$ non-conflicting messages can be sent in parallel using *all* available links.

Sections As pointed out earlier, in the single element case, nodes can be grouped into sections to allow more messages to be transferred concurrently (Figure 3.3). In applying this scheme to the array case, where many-to-many rather than one-to-one communication is used, it is not sufficient to consider just the shift offset. In Figure 3.9a there is no additional horizontal offset involved. Sections are established by determining the nodes requiring a common link. The two right-most source nodes need the link to their right. The second source node is not included since it does not require that link for data transfer. This is continued until all nodes are classified.

In Figure 3.9b the situation is different since the source of the base pattern is larger than 1×1. The five right-most nodes in each source row must utilize the link to the left of them. Those nodes belong to two source regions. Since regions always send messages during a step, the six right-most nodes or the two right-most regions are grouped into one section, which is also the largest in the source area. No collision would occur because regions sending messages are in different columns and at least two regions (six nodes) apart.[2] In both examples, the largest section is the bottleneck for the transfer. Thus, partitioning the grid into sections that have the size of the largest section avoids conflicts, and the transfer is as fast as for any other partitioning.

Horizontally, the largest section is always located at the left or the right end of the arrays. Thus, its size can be computed by the offset of the left-most (right-most) nodes of the source and destination areas (those are the nodes holding $A(l_c)$ and $A(l'_c)$)

[1] In order to keep the regions smaller, only $\mathcal{C}'/\gcd(\mathcal{R},\mathcal{C}') \times \mathcal{R}/\gcd(\mathcal{R},\mathcal{C}')$ can be grouped. The number of bottlenecks in each dimension is still the same.

[2] The closest nodes are only four columns apart, but the region-communication implementation makes sure that only nodes at the same relative position in the regions send at the same time (e.g. the first nodes of each region).

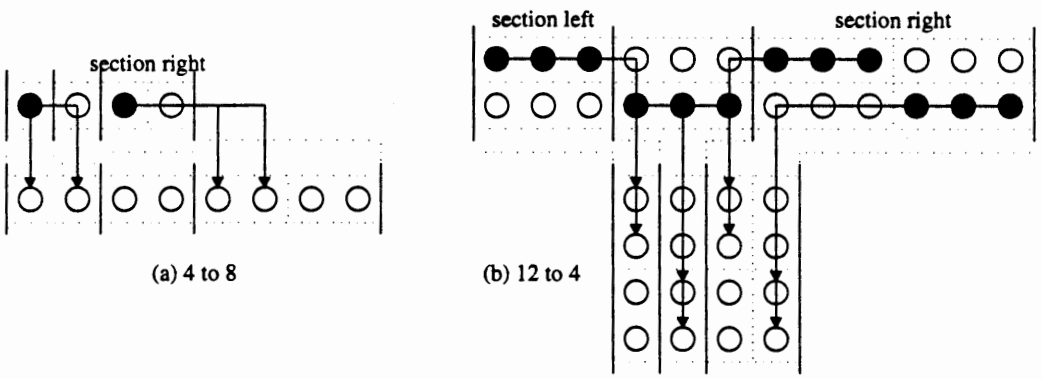


Figure 3.9: Partitioning the arrays into sections of nodes that require a common link. The size of the largest section determines the degree of parallelism and is located either on the left or the right side of the arrays. Messages sent from regions of different sections do not conflict. Note that the scatter/gather base patterns in Figures (a) and (b) both have one link in each dimension ($\mathcal{RC}' = 1$; no grouping of base patterns is necessary).

divided by the horizontal region-size of the destination or the source, depending on whether the outmost nodes belong to the source or the destination (if and otherwise cases). The min operators reflect that the number of nodes cannot exceed the total number of nodes holding the source ($\mathcal{N}_c$) and the destination array ($\mathcal{N}'_c$) in this dimension.

The larger number of regions determines the maximal section size:

$$\begin{aligned} max\_section\_size &= \mathsf{max}(sec\_ver, sec\_hor) \\ sec\_hor &= \mathsf{max}(sec\_left, sec\_right) \\ sec\_ver &= \mathsf{max}(sec\_up, sec\_down) \end{aligned} \tag{3.6}$$

where

$$sec\_left = \begin{cases} \lceil \frac{\min(\mathcal{P}_c(l_c)-\mathcal{P}'_c(l'_c),\mathcal{N}'_c)}{\mathcal{C}'\mathcal{R}} \rceil & \text{if } \mathcal{P}_c(l_c) - \mathcal{P}'_c(l'_c) > 0 \\ \lceil \frac{\min(\mathcal{P}'_c(l'_c)-\mathcal{P}_c(l_c),\mathcal{N}_c)}{\mathcal{C}\mathcal{R}} \rceil & \text{otherwise} \end{cases}$$

$$sec\_right = \begin{cases} \lceil \frac{\min(\mathcal{P}'_c(h'_c)-\mathcal{P}_c(h_c),\mathcal{N}'_c)}{\mathcal{C}'\mathcal{R}} \rceil & \text{if } \mathcal{P}'_c(h'_c)-\mathcal{P}_c(h_c)>0 \\ \lceil \frac{\min(\mathcal{P}_c(h_c)-\mathcal{P}'_c(h'_c),\mathcal{N}_c)}{\mathcal{C}\mathcal{R}} \rceil & \text{otherwise} \end{cases} \quad (3.7)$$

and *sec_up* and *sec_down* can be defined accordingly by changing subscripts from column to row, $\mathcal{C}'\mathcal{R}$ to $\mathcal{R}'\mathcal{C}'$ and $\mathcal{C}\mathcal{R}$ to $\mathcal{R}\mathcal{C}'$. It is possible to split up the whole source into sections consisting of *sec_ver* × *sec_hor* regions and apply the diagonal scheme to each section in parallel. Thus, max(*sec_ver*, *sec_hor*) region transfers are required. In Figure 3.9c this was already done: there are two sections of 2×2 regions, and regions on all first diagonals are sending their messages. If the vertical offset would be zero rather than 3 rows, regions of the first row could send independently from nodes of the second row.

3.2.3 Scheduling Algorithm

Algorithm 2 consists of a main program and a region-communication subroutine. The algorithm is to be run in a distributed manner by every processor.

The program identifies sections and arranges messages from diagonal regions in each section to be sent in parallel. The main program calls a subroutine to carry out region-to-region data transfers. The actual implementation of the subroutine does not affect the overall scheduling approach, as long as it assures that the low level data transfer between the regions is contention-free and done with optimal link usage. The algorithm requires *reg* · max(*sec_ver*, *sec_hor*) steps, where *reg* is the time it takes the subroutine to transfer a region.

Theorem 2 *If the largest quotient (one of sec_left, sec_right, sec_up, or sec_down) determining max_section_size in Equation (3.7) has no remainder, then the schedule*

Algorithm 2 (Extended Diagonal Scheduling Scheme)
Scheduling algorithm for regular transfers. The two parameters row and col denote the node's location in the network

```
Main Program
    if this_node is a source node
        region_row = (row − P_r(l_r)) div (RC')          /* determine my */
        region_col = (col − P_c(l_c)) div (CR)           /* region number */
        determine sec_ver and sec_hor                    /* use Equation 3.6 */
        section_row = region_row mod sec_ver             /* region's position */
        section_col = region_col mod sec_hor             /* inside section */
        diagonal_num = section_row − section_col + 1
        if diagonal_num ≤ 0
            diagonal_num = diagonal_num + max(sec_ver, sec_hor)
        end if

        for step = 1 to max(sec_ver, sec_hor)            /* region-steps needed */
            if step = diagonal_num
                region_communication
            end if
        end for

    end if
end Main
```

generated by the algorithm is optimal with respect to the number of data transfer steps. If the quotient has a remainder, then the algorithm wastes fewer than reg steps.

Proof: See appendix.

3.2.4 Region Communication Subroutine

Depending on the message size and architectural parameters such as start-up latency and channel bandwidth, there are several different region-communication implementations that are most suitable for certain cases. The procedural layout allows us to use different solutions interchangeably. In the following sections, a transfer using local scatter and gather operations, the direct one-to-one transfer, and a hybrid

solution are presented.

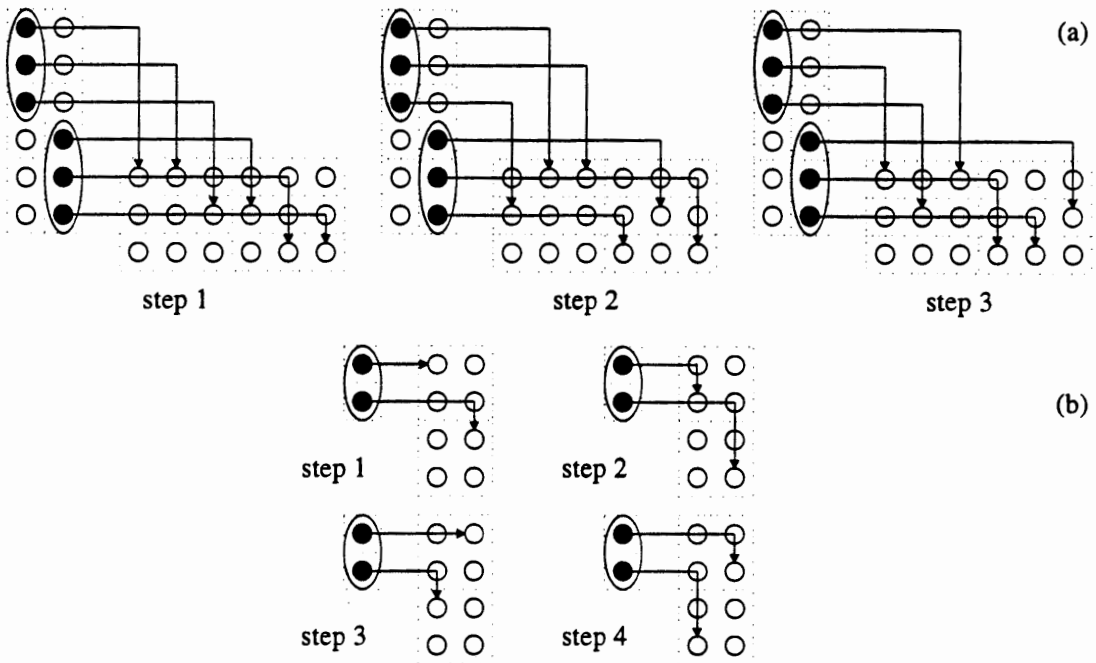


Figure 3.10: One-to-one transfer of a region. The ovals represent the groups of vertically aligned nodes sending in parallel. Even though the sizes of the regular sections (2 and 3) do not correspond in Figure (a), the scheduling algorithm still makes optimal use of the six vertical and horizontal links. In Figure (b) the pattern is similar, but each source node has two destinations in the same column.

One-To-One

During a shift operation, a region consists of $\mathcal{C}' \times \mathcal{R}$ base patterns; each has $\mathcal{R} \times \mathcal{C}$ source nodes. In the beginning of the main program, each node determines the region it is in by dividing its relative position by the region-size. Then, in the beginning of the subroutine, the node's index inside the region is determined. Using this index, $\mathcal{C}'$ vertically aligned nodes are grouped together. Even though those nodes are located in the same column, they can still send messages concurrently since each base pattern has $\mathcal{C}'$ vertical links. However, it must be ensured that the nodes' destinations are in

disjoint columns. This is done by the following method: $\mathcal{C}'$ steps are required since each node has $\mathcal{C}'$ destinations. During step j of the transfer, node i of the group (counting from zero) sends its message to the node in the $((i+j) \bmod \mathcal{C}')$th column of its base pattern destination. The groups themselves are sent using the diagonal scheme. Figure 3.10a shows an example.

Figure 3.10b shows a scatter/scatter case where each node has destinations in two rows. Each "step" then consists of $\mathcal{R}'$ message transfers. In Figure 3.9b the scatter/gather requires six messages per step. The general number for all cases is $\mathcal{R}'\mathcal{C}$. The overall number of steps for the transfer of an $\mathcal{R} \cdot \mathcal{C}' \times \mathcal{C} \cdot \mathcal{R}$ region to its $\mathcal{R}' \cdot \mathcal{C}' \times \mathcal{C}' \cdot \mathcal{R}$ destination is

$$reg_{1\to 1} = \mathcal{R} \cdot \mathcal{C} \cdot \mathcal{R}' \cdot \mathcal{C}' \cdot (\alpha + \beta), \tag{3.8}$$

with $\alpha + \beta$ being the time to transmit one message.

The direct approach has the advantage of the lowest software overhead possible since no intermediate nodes are used. Furthermore, no initial gathering of data is necessary. However, several message startups are required.

Local Gather and Scatter

Rather than sending each message separately, this subroutine *gathers* all the messages to a single node in the base pattern source, sends a single message to the base pattern destination, and *scatters* the data from there.

Figure 3.11a shows this process for the example from the previous section. The local gather and scatter nodes are marked grey. Base patterns at position $(i, j) = (\mathit{local\_row} \text{ div } \mathcal{R}, \mathit{local\_col} \text{ div } \mathcal{C})$ within the region gather the data on the node with index $(j, 0) = (\mathit{local\_row} \bmod \mathcal{R}, \mathit{local\_col} \bmod \mathcal{C})$ within the base pattern and scatter the data from the node $(0, i)$. Figure 3.11b shows a scatter/scatter operation where the message is sent to a node in the top row by default. Note that the gather and scatter operations can be performed in parallel for all regions. Calls to the subroutines local_gather and local_scatter must be added into the blank lines before and after the sf for loop in Algorithm 2. Since gather and scatter are standard routines of the

Algorithm 3 (One-To-One Base Pattern Subroutine)
Subroutine for regions consisting of $\mathcal{R} \times \mathcal{C} \rightarrow \mathcal{R}' \times \mathcal{C}'$ base patterns. The two parameters row and col denote the node's location in the network

```
Procedure region_communication
    local_row = (row − P_r(l_r)) mod (RC')                    /* determine my */
    local_col = (col − P_c(l_c)) mod (CR)                     /* offset in region */
    diagonal_num = local_row div C' − local_col div C + 1
    if diagonal_num ≤ 0                                       /* diagonal groups */
        diagonal_num = diagonal_num + R
    end if
    for step = 1 to R                                         /* R groups of C'×C */
        if step = diagonal_num                                /* perform group */
            for j = 1 to C'                                   /* C' dest. columns */
                forall rows: send R'C messages from C node(s) to all destinations
                in the ((local_row mod C' + j) mod C')th column sequentially
            end for
        end if
    end for
end Procedure
```

Message Passing Interface (MPI)[13], no explicit listing is presented here. Efficient implementations can be found in [1, 2, 3].

This method has the lowest transfer-time possible since all data passes the bottleneck channels with a single message, saving several message start-ups:

$$reg_{gather/scatter} = \alpha + \mathcal{R} \cdot \mathcal{C} \cdot \mathcal{R}' \cdot \mathcal{C}' \cdot \beta. \tag{3.9}$$

The drawback is the overhead caused by the initial gather and the scatter operations at the end. Implemented with recursive halving and doubling [36] and assuming power of two number of nodes these operations require:

$$overhead_{gather/scatter} = \alpha \cdot (\log_2(\mathcal{R} \cdot \mathcal{C}) + \log_2(\mathcal{R}' \cdot \mathcal{C}')) + \beta \cdot (2 \cdot \mathcal{R} \cdot \mathcal{C} \cdot \mathcal{R}' \cdot \mathcal{C}' - \mathcal{R} \cdot \mathcal{C} - \mathcal{R}' \cdot \mathcal{C}') \tag{3.10}$$

Which subroutine runs faster depends on the ratio of the startup time α and the transfer time for a unit message β. Furthermore the number of sequential region transmissions is important. With growing number of regions, the constant overhead of the local gather/scatter scheme plays a lesser role in the overall time.

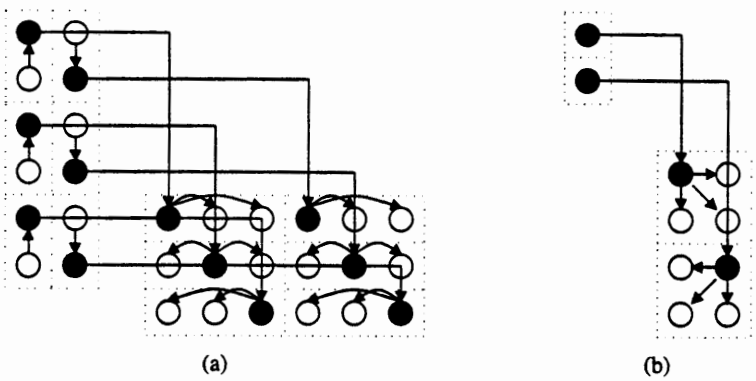


Figure 3.11: Local gather/scatter transfer of a region.

So far, only the two most extreme cases for the implementation of building blocks were presented. Sometimes it may not be clear which solution is more appropriate and therefore, it makes sense to introduce a method that combines part of the properties of both extremes.

Hybrid

The example in Figure 3.13 shows different degrees of message combination for the transfer from 4 to 2 nodes. Figures (a) and (f) represent the one-to-one case (no message combination) and the local gather/scatter case (complete message combination). There is only one bottleneck in each dimension connecting the two processor arrays.

		a	b	c	d	e	f
gather/scatter overhead	α	0	1	1	2	2	3
	β	0	2	4	6	6	10
	time($\alpha = \beta$)	0	3	5	8	8	13
transmission through bottleneck	α	8	4	4	2	2	1
	β	8	8	8	8	8	8
	time($\alpha = \beta$)	16	12	12	10	10	9
total for 1 region		16	**15**	17	18	18	22
total for 4 regions		64	51	53	**48**	**48**	49
total for 10 regions		160	123	125	108	108	**103**

Algorithm 4 (Local Gather and Scatter Base Pattern Subroutine)
Subroutine for regions consisting of $\mathcal{R} \times \mathcal{C} \rightarrow \mathcal{R}' \times \mathcal{C}'$ base patterns. The two parameters row and col denote the node's location in the network

```
Procedure region_communication
    local_row = (row − P_r(l_r)) mod (RC')          /* determine my */
    local_col = (col − P_c(l_c)) mod (CR)           /* offset in region */
    if (local_row mod R) = (local_col div C)
      and (local_col mod C) = 0
        send to base pattern destination (0, local_row div R)
    end if
end Procedure

Procedure local_gather                              /* called before the loop */
    forall base pattern sources
        gather data at node (local_col div C, 0)
    end forall
end Procedure

Procedure local_scatter                             /* called after the loop */
    forall base pattern destinations
        scatter data from node (0, local_row div R')/* use R' for dest. node */
    end forall
end Procedure
```

This table shows the performance of the different transfers split up into the communication overhead caused by the gather and scatter operation and the data-transfer across the bottleneck. α and β are defined as in Section 2.1.3. Each source node holds two data packages that take 2β pure transmission time and each destination receives four packages. The times are computed with $\alpha = \beta$. Note that the local operations require some additional processing time on the nodes. This is omitted in the table. From left to right, the transmission time decreases but the overhead time increases. The pattern from Figure (c) is always worse than its counterpart in Figure (b) because a gather operation on the larger array is more efficient than a scatter on the smaller destination array since more channels are in use and thus, the message sizes are smaller.

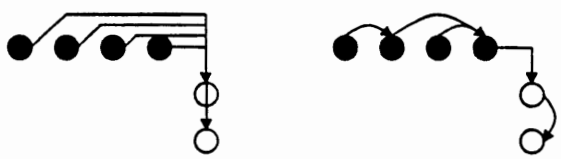


Figure 3.12: The transfer patterns presented so far are applied on a $4 \rightarrow 2$ base pattern with a single bottleneck. The figures correspond to Figures 3.13a and 3.13f showing the one-to-one and the local gather/scatter transfers.

The bottom lines show the total time required for the transfer of 1, 4, and 10 regions with $\alpha = \beta$. For a small number of regions, patterns with a lesser degree of combination do better (b and d/e) but for 10 regions, pattern (f) is the most efficient.

The idea is to determine the optimal degree of message gathering for specific values of α, β, and the number of sources and destinations. Algorithm 5 starts out with the situation from Figure (a). It compares the cost of an immediate one-to-one transfer ($\mathit{regions} \cdot \mathit{sources} \cdot \mathit{dests} \cdot (\alpha + \beta)$) with the cost of a gather in the source region followed by a one-to-one transfer ($(\alpha + \beta_s) + \mathit{regions} \cdot \mathit{sources}/2 \cdot \mathit{dests} \cdot (\alpha + 2\beta)$) or a one-to-one transfer followed by a scatter in the destination region ($\mathit{regions} \cdot \mathit{sources} \cdot \mathit{dests}/2 \cdot (\alpha + 2\beta) + (\alpha + \beta_d)$), depending on which region is larger. β_s and β_d are the times to transfer the data located on the source and destination nodes. If the immediate one-to-one transfer has a higher cost, then the parameters are adjusted with respect to the scatter or gather operation. This cycle is repeated until the one-to-one pattern is the fastest.

Note that Algorithm 5 only works for region-sizes that are a power of two. The actual communication algorithm can be derived from Algorithms 3 and 4 but is omitted here.

3.3 Arbitrary Block-Distributions

In this section all restrictions from Section 3.2 are dropped and a solution for *arbitrary* block-distributions is presented.

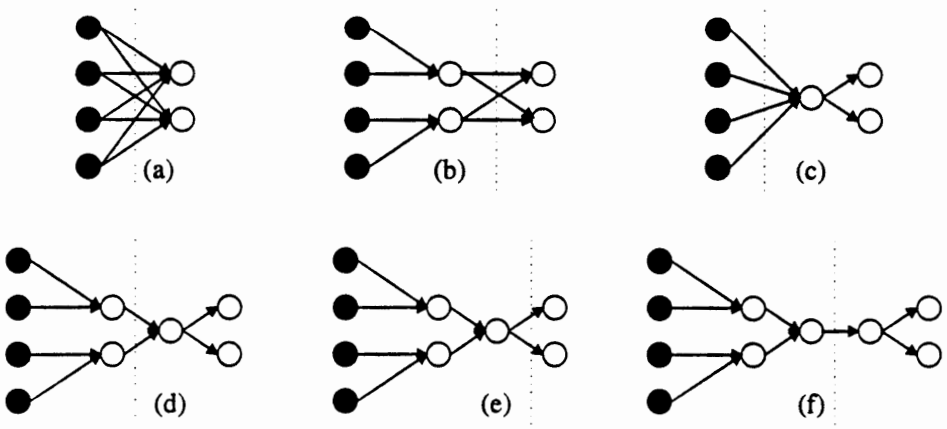


Figure 3.13: Hybrid solutions for a $4 \rightarrow 2$ transfer. The dotted line represents the bottleneck between the source (on the left side) and the destination (on the right side). The transfers left of the dotted line represent local gather operations and the transfers right of the dotted line represent local scatter operations. Messages through the bottleneck must be sent sequentially, the other transfers can be performed in parallel.

The functions $\mathcal{F}(n)$ and $\mathcal{L}(n)$ describe the array section indices of the first and the last elements of $A(l:h:s)$ that are located on node n (these are elements $A(l+s\cdot\mathcal{F}(n))$ and $A(l+s\cdot\mathcal{L}(n))$).

$$\mathcal{F}(n) = \lceil (nk-l)/s \rceil, \qquad \mathcal{L}(n) = \mathcal{F}(n+1) - 1$$

Functions $\mathcal{F}'(n)$, and $\mathcal{L}'(n)$ are defined similarly for the destination array section B.

3.3.1 Communication Pattern

From the topology and the hardware routing algorithm, the communication pattern for the transfer of an array section can be derived. Figure 3.14 shows an example. Since s does not necessarily divide k, the source nodes hold either $\lfloor k/s \rfloor$ or $\lceil k/s \rceil$ data elements. An exception are nodes at the beginning or the end of the array section. Those nodes might store less data (e.g. source 3). Analogously, up to $\lceil k'/s' \rceil$ elements are located on the destination nodes. In the example, the source nodes hold more data and therefore, each source node *scatters* its data to a cluster of destination nodes.

Algorithm 5 (Hybrid Base Pattern Subroutine)
Subroutine to determine the optimal degree of message combination for the transfer of (regions) sequential transfers of $\mathcal{R} \times C \rightarrow \mathcal{R}' \times C'$ regions using a single bottleneck channel.

```
Procedure region_communication
    sources = RC,  dests = R'C'
    β_s = β · dests,  β_d = β · sources                    /* initial situation is */
    gather_steps = 0,  scatter_steps = 0                   /* one to one transfer */
    loop
        save_on_one_to_one = (regions · sources · dests · α)/2
        cost_combine = α+min(β_s, β_d)
        exit loop if (cost_combine > save_on_one_to_one) or (sources = dests = 1)
        if sources > dests                                 /* perform gather on source */
            sources = sources/2,  β_s = 2β_s               /* twice the data on half nodes */
            gather_steps = gather_steps + 1
        else                                               /* perform scatter on dest. */
            dests = dests/2,  β_d = 2β_d
            scatter_steps = scatter_steps + 1
        end if
    end loop
end Procedure
```

Whereas the middle nodes of a cluster get only one message (destination 2), the two boundary nodes can each get an additional message from a different source node (the cluster 1-2-3 gets messages from source 1, the middle node 2 gets one message, while 1 and 3 each get two messages). If the source nodes hold less data, then the pattern is reversed and a cluster of source nodes *gathers* its data onto one destination node. Just as in the regular case, these scatter and gather operations are called *base communication patterns*. Figure 3.15 illustrates the four 2D patterns.

The figures show that the transfer of array sections yields quite complicated message patterns where neither the number of destinations for a source node nor the size of the messages that have to be sent is constant. Two-dimensional transfers are even more complicated since they involve 1D patterns in both dimensions.

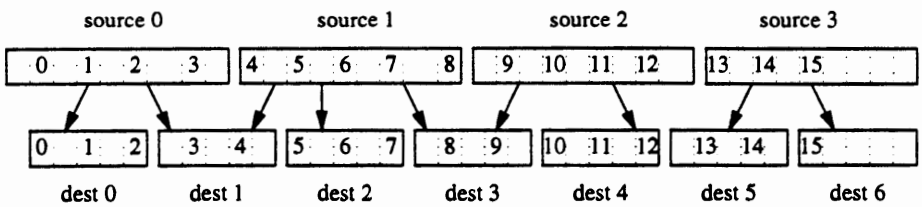


Figure 3.14: Communication pattern for the transfer of a 1D array section with blocksizes $k = 9, k' = 5$ and section-strides $s = s' = 2$. The nodes are symbolized by the solid boxes. A number i indicates the ith array section element. The source nodes hold either 4 or 5 array section elements and the destination nodes hold either 2 or 3. The arrows show the resulting communication pattern.

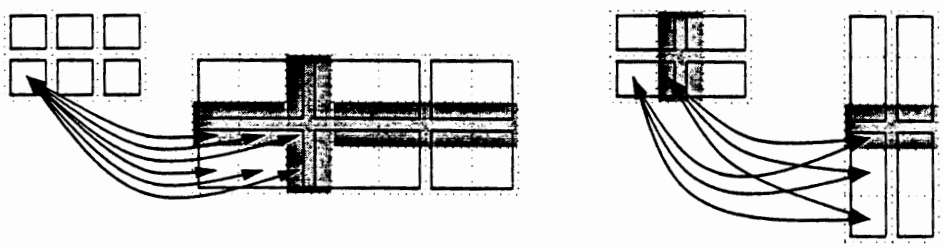
3.3.2 Initial and Final Shifts

In order to derive a scheduling solution for the communication induced by array operations, it is necessary to simplify complex communication patterns. The idea is to avoid message path overlaps among concurrent gather and scatter operations, and to obtain regionally independent operations that can be integrated in an overall scheduling approach.

Scatter In a scatter situation (e.g. Figure 3.14), this goal can be achieved by combining array segments that must be sent to a common destination from two adjacent source nodes. By convention, the two array segments are combined to the left node. Figure 3.16 shows the new communication pattern for the example in Figure 3.14, after the array segments are combined. This *initial shift* involves only neighbor-communication; therefore, the messages can all be sent in parallel inducing only a constant overhead of less than $\alpha+\beta$. Each source node n can detect if it has to shift data by checking whether the destinations of its first and the preceding node's last section elements are the same. The first source node does not shift its data.

$$\mathcal{P}'(l' + s' \cdot \mathcal{F}(n)) = \mathcal{P}'(l' + s' \cdot \mathcal{L}(n-1)) \tag{3.11}$$

In Figure 3.16, for example, source one's first array section element (4) has the same destination (dest 1) as source zero's last array section element (3). Thus, source



(a) scatter/scatter and gather/gather

(b) scatter/gather and gather/scatter

Figure 3.15: Communication pattern for the transfer of a 2D array section. The dotted squares represent the nodes and the solid lines mark the sources and destinations of the base communication patterns. Processors that receive two scatter-messages or send two gather-messages are colored grey. In Figure (a), each source scatters data to a processor grid of size 2×3 or 2×2. Pairs of nodes communicate with a 3×1 processor grid in Figure (b).

one must shift all data that must be transferred to destination one.

Gather If the source nodes hold less data than the destination nodes,[3] then clusters of source nodes gather data. Sources that store data for two destinations send everything to the left destination. After all gathers are completed, the data is shifted right to the correct location (*final shift*). Each destination node n can detect if it will receive data that must be forwarded to the next node. This is the case if the sources of its last and the next node's first section elements are the same:

$$\mathcal{P}(l + s \cdot \mathcal{L}'(n)) = \mathcal{P}(l + s \cdot \mathcal{F}'(n+1)) \tag{3.12}$$

Two-Dimensional Array Sections For scatter/scatter or gather/gather operations (Figure 3.15a), two shifts (one horizontally and one vertically) must be performed sequentially in any order before (scatter/scatter) or after (gather/gather) the main data transfer. Scatter/gather and gather/scatter operations (Figure 3.15b) require one shift before and one after the main data transfer.

[3]For this comparison the real values of k/s and k'/s' must be used.

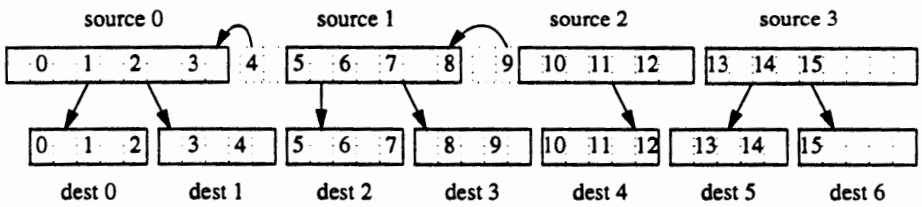


Figure 3.16: Example from Figure 3.14 after nodes that provide only a partial data set for their leftmost destination shifted the data to their left neighbor. Non-overlapping scatter operations are obtained.

Determining the Base Pattern Index In order to schedule the individual base communication patterns, each source node must determine the base pattern it belongs to. The base patterns are identified by an index. If a scatter operation is performed by node n, then the base pattern index is equal to n's location relative to the first node in the cluster. In Figure 3.16, for example, the enumeration of the source nodes reflects their pattern indices.

If node n performs a gather operation, then n's base pattern index is determined via its destination node (source and destination have the same pattern index). The destination node is found with the array section index of n's first data element which is transferred to $B(l'+s'\cdot\mathcal{F}(n))$. Thus, the destination node is located at $\mathcal{P}'(l'+s'\cdot\mathcal{F}(n))$. Its pattern index $\mathcal{I}'$ can be determined with the method described in the previous paragraph.

$$\mathcal{I}(n) = \begin{cases} n - \mathcal{P}(l) & \text{if scatter pattern} \\ \mathcal{I}'(\mathcal{P}'(l' + s' \cdot \mathcal{F}(n))) & \text{if gather pattern} \end{cases}$$

$$\mathcal{I}'(n) = \begin{cases} n - \mathcal{P}'(l') & \text{if gather pattern} \\ \mathcal{I}(\mathcal{P}(l + s \cdot \mathcal{F}'(n))) & \text{if scatter pattern} \end{cases} \tag{3.13}$$

To determine the pattern index of destination 3, for example, its first array section element is used (i.e. 8). The corresponding array element is located on source 1. Thus, source 1 and destination 3 have the same pattern index 1.

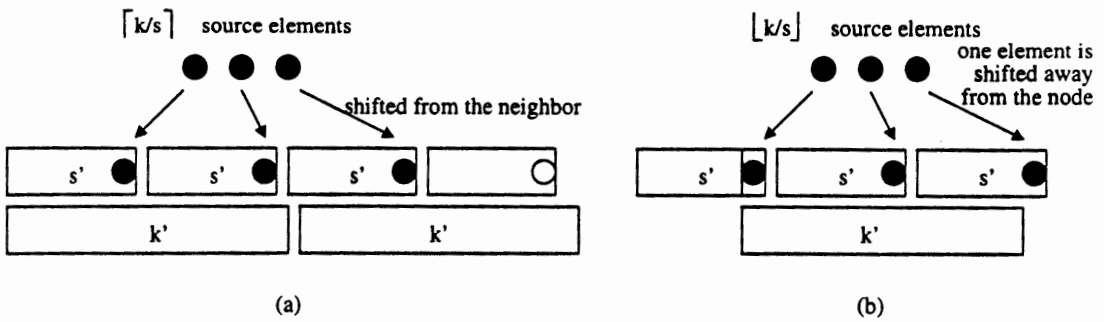


Figure 3.17: Smallest and largest base communication patterns possible.

Determining the Base Pattern Size The size of the base patterns is no longer fixed. In a scatter operation, the source can have up to $\lceil k/s \rceil$ data elements. The destinations of these elements are located on a subarray of the destination array with size $\lceil k/s \rceil \cdot s'$. The ceiling of this size divided by the blocksize k' yields the maximum number of destination nodes (Figure 3.17a):

$$C = 1, \quad C' = \lceil \frac{\lceil k/s \rceil \cdot s'}{k'} \rceil \tag{3.14}$$

The base pattern size for the gather case can be derived by exchanging s and s', as well as k and k'.

The minimum number of destination nodes (Figure 3.17b) is determined as follows: at least $\lfloor k/s \rfloor$ data elements are located on the source. The destination elements are located on a $(\lfloor k/s \rfloor - s') \cdot s' + 1$ submatrix. Again, this size is divided by k' to obtain the number of destinations. The floor operator is used here, since partial data for a node is shifted away:

$$C = 1, \quad C' = \lfloor \frac{(\lfloor k/s \rfloor - s') \cdot s' + 1}{k'} \rfloor \tag{3.15}$$

3.3.3 Applying the Extended Diagonal Scheduling Scheme

After the initial and final shifts are applied, the array section case has similar properties compared to the regular case described in Section 3.2. The only difference

is that some base patterns might have fewer participating nodes (Figure 3.18a). The following text shows how to apply the regular case to this problem.

Equation 3.16 determines the offset for one dimension. In this context, the offset represents the maximum number of base patterns a message traverses on its way. It is obtained via the base pattern index of the destination node next to the first and last source node. The offset is the maximum of the offsets on both ends. It cannot exceed the total number of base patterns, which is $\mathcal{I}(\mathcal{P}(h)) - \mathcal{I}(\mathcal{P}(l)) + 1$. In Figure 3.18a, for example, the first destination node is located in the fifth column. Since the left neighbors belong to the horizontal base pattern one, the offset is two (four nodes belonging to two base patterns).

$$first = \begin{cases} \mathcal{I}'(\mathcal{P}(l) - 1) + 1 & \text{if } \mathcal{P}(l) > \mathcal{P}'(l') \\ 0 & \text{if } \mathcal{P}(l) = \mathcal{P}'(l') \\ \mathcal{I}(\mathcal{P}'(l') - 1) + 1 & \text{if } \mathcal{P}(l) < \mathcal{P}'(l') \end{cases}$$

$$last = \begin{cases} \mathcal{I}'(\mathcal{P}'(h')) - \mathcal{I}'(\mathcal{P}(h) + 1) + 1 & \text{if } \mathcal{P}(h) < \mathcal{P}'(h') \\ 0 & \text{if } \mathcal{P}(h) = \mathcal{P}'(h') \\ \mathcal{I}(\mathcal{P}(h)) - \mathcal{I}(\mathcal{P}'(h') + 1) + 1 & \text{if } \mathcal{P}(h) > \mathcal{P}'(h') \end{cases} \tag{3.16}$$

$$offset = \min(\max(first, last), number\_of\_patterns)$$

Equation 3.17 divides the base pattern offset by the number of base patterns that are grouped into a region. In the regular case, $C' \times \mathcal{R}$ base patterns are combined. For general transfers this number can vary slightly (see Section 3.3.4).

$$sec = \lceil \frac{offset}{patterns\_per\_region} \rceil \tag{3.17}$$

The following terminology is introduced: ***single nodes*** are sources of scatter operations or destinations of gather operations. A *cluster* is the group of nodes receiving scatter messages or sending gather messages.

Theorem 3 *If all clusters consist of at least one node, then base patterns that are (1) diagonally aligned or (2) located at the same relative position within adjacent sections of* ($sec\_ver \times sec\_hor$) *base patterns do not collide.*

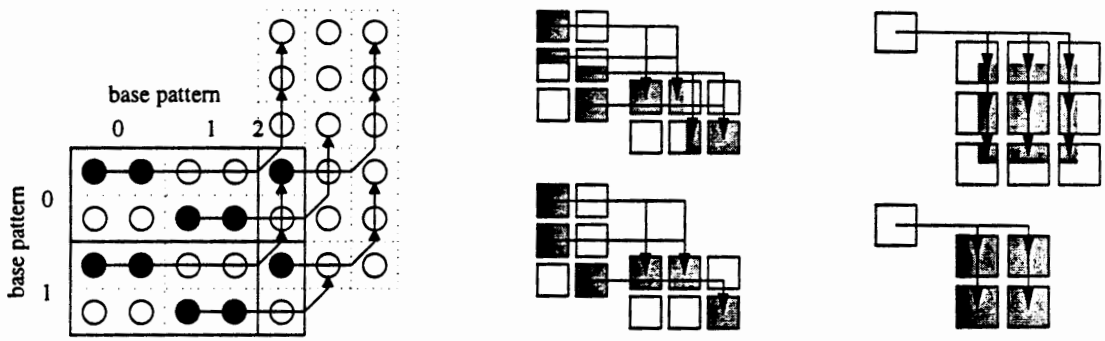


Figure (a) Figure (b) Figure (c)

Figure 3.18: Figure (a) presents an example of a scheduled scatter/gather transfer. The dotted lines mark the base communication patterns; the solid lines partition the source grid into four sections performing the diagonal scheme in parallel. The offset is two in both dimensions. Examples (b) and (c) show communication patterns before (top) and after (bottom) the shifts. The squares represent the nodes and the grey areas mark the transferred data. In (b) four gather/scatter base patterns with four messages of different sizes are transformed to four patterns with 1, 2, 2, and 4 fixed-size messages. Figure (c) shows a scatter/scatter base pattern with 9 messages that is transformed into a regular $1 \rightarrow 2 \times 2$ communication.

Proof: See appendix.

For the special case of a scatter operation, where both $k/s \geq k'/s'$ and $\lfloor k/s \rfloor < \lceil k'/s' \rceil$ are fulfilled, some cluster might contain no nodes. In this case, an offset of $q+1$ must be used. Due to the very rare occurrence of this case, this detail is omitted in the algorithm.

With Equation 3.16 and the theorem, the transfer can be partitioned into independent sections which can be handled concurrently with the diagonal scheduling scheme. Figure 3.18a shows an example of the transfer. Similarly to Section 3.2, base patterns can be grouped into regions in order to optimize the channel usage. Slight adjustments are necessary in the region communication subroutines and the definition of the base pattern sizes $\mathcal{R}$, $\mathcal{R}'$, $\mathcal{C}$, and $\mathcal{C}'$ (see next section).

3.3.4 General Scheduling Algorithm

Algorithm 6 runs in a distributed manner on every processor. It consists of two parts: the main program and the region communication subroutine.

Main Program In the middle part, the diagonal scheduling scheme is applied to each section. Before and after that, the node participates in the initial and final shifts for each dimension if necessary. The main program calls a subroutine to carry out the region communication. The actual implementation of the subroutine does not affect the overall scheduling approach as long as it assures that the low level data transfer is contention-free and done with high link usage.

Base Pattern In the regular case, the base pattern size is always fixed but for general block distributions, it can vary by one in each dimension. The adjustments that have to be made in the region communication subroutine are presented for the one-to-one and the local gather/scatter approach. Figure 3.19 repeats the one-to-one example from Figure 3.10. The maximum pattern sizes are the same but some patterns are smaller.

Figure 3.19 repeats the local gather/scatter example from Figure 3.11. Since the actual transfer through the bottleneck is done in one step, the region can only have one column of base patterns. Otherwise, the messages of neighboring base patterns with only one node would conflict (Figure a). Therefore, $\mathcal{R}$ and $\mathcal{C}'$ are set to the smaller sizes. $\mathcal{R}'$ and $\mathcal{C}$ do not affect the number of channels between the regions and can be defined as in the previous case (see Figure b).

The only problem remaining is the enumeration of the nodes. For example, the nodes in the fourth row of the source region in Figure 3.19a have index 3, but for the region communication they must have index 4. Thus, the computation of *local_row* and *local_col* must be changed to:

$$local\_row = base\_pattern\_row \bmod \mathcal{C}' + row\_offset$$
$$local\_col = base\_pattern\_col \bmod \mathcal{R} + col\_offset$$

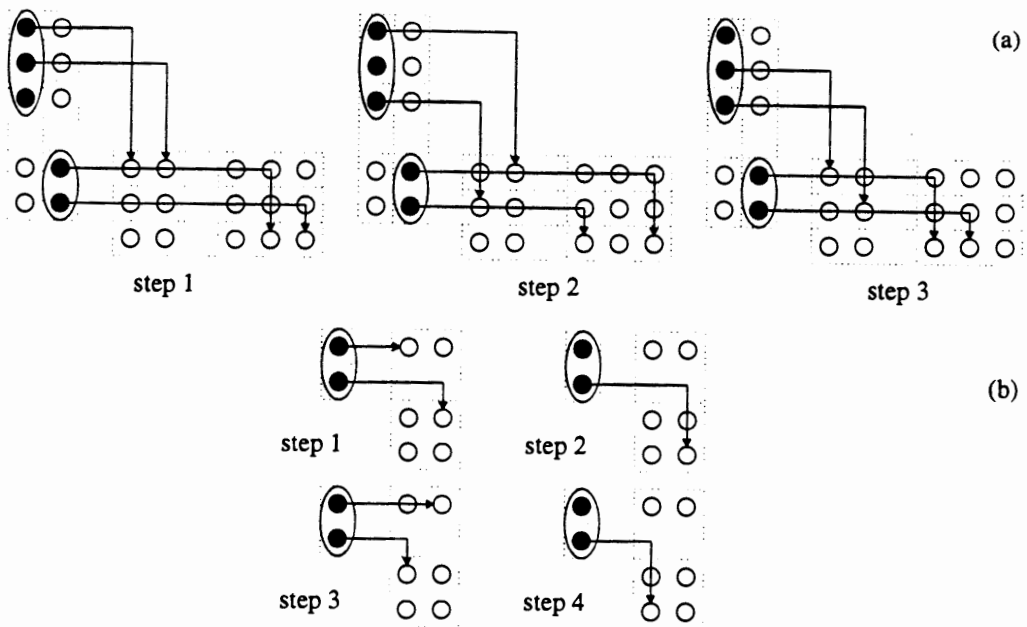


Figure 3.19: One-to-one transfer of a region with varying base pattern sizes. Figure (a) contains patterns with sizes $(1/2) \times 1 \rightarrow 1 \times (2/3)$; Figure (b) shows sizes $1 \times 1 \rightarrow (1/2) \times 2$. The regions are composed of 3×2 and 1×2 patterns according to the larger sizes of $\mathcal{R}$ and $\mathcal{C}'$.

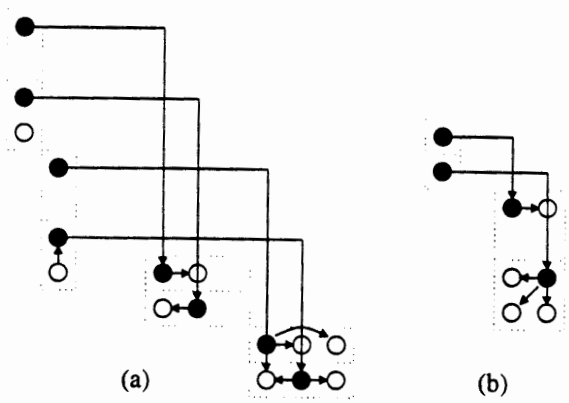


Figure 3.20: Local gather/scatter transfer of a region with varying base pattern sizes. Figure (a) contains patterns with sizes $(1/2) \times 1 \rightarrow 1 \times (2/3)$; Figure (b) shows sizes $1 \times 1 \rightarrow (1/2) \times 2$. The regions are composed of 2×1 and 1×2 patterns according to the smaller sizes of $\mathcal{R}$ and $\mathcal{C}'$.

Algorithm 6 (Initial and Final Shifts with Diagonal Scheduling)
Scheduling algorithm for a transfer of an array section. The two parameters row and col denote the node's location in the network. Source and dest are flags indicating whether the node is a source or a destination node. scatter_ver, gather_ver, scatter_hor, and gather_hor characterize the base pattern for each dimension.

```
Main Program
    if (source)                                              /* initial shift */
        if (scatter_ver) and (not top node) and (common dest with row − 1)
            initial_shift_up                                 /* use Equation 3.11 */
        end if
        if (scatter_hor) and (not leftmost node) and (common dest with col − 1)
            initial_shift_left                               /* use Equation 3.11 */
        end if                                               /* diagonal scheme */
        determine base_pattern_row and base_pattern_col      /* use Equation 3.13 */
        region_row = base_pattern_row div C'                 /* determine my */
        region_col = base_pattern_row div R                  /* region number */
        determine offset sec_ver and sec_hor                 /* use Equation 3.16 */
        section_row = region_row mod sec_ver                 /* pattern's position*/
        section_col = region_col mod sec_hor                 /* inside section */
        diagonal_num = section_row − section_col + 1
        if diagonal_num ≤ 0
            diagonal_num = diagonal_num + max(sec_ver, sec_hor)
        end if
        for step = 1 to max(sec_ver, sec_hor) /* sequential transfer needed */
            if step = diagonal_num
                region_communication
            end if
        end for
    end if                                                   /* final shift */
    if (gather_ver) and (dest) and (common source with row + 1)
        final_shift_down                                     /* use Equation 3.12 */
    end if
    if (gather_hor) and (dest) and (common source with col + 1)
        final_shift_right                                    /* use Equation 3.12 */
    end if
end Main
```

Chapter 4

Scheduling Solutions for Transposed Alignments

This chapter deals with transfers between array sections that have a transposed alignment. Without loss of generality we assume that the first array has its rows aligned with the rows of the processor grid whereas the rows of the second array are distributed over a column of processors.[1]

Similarly to Chapter 3 the communication pattern can be partitioned into three categories. The following two sections describe the changes with respect to the identical alignment case.

4.1 Identical Distributions

Definition For 2-dimensional array sections, a transfer is called a *transposition* if the transfers in both dimensions are shifts and the arrays have transposed alignments.

The offset between the nodes holding the top left element of the source and the destination array section is defined similarly to Equation 3.3. Since the alignment of the destination array is transposed with respect to the processor grid, $\mathcal{P}'_r(l'_r)$ and $\mathcal{P}'_c(l'_c)$ are exchanged. The vertical and horizontal offsets are denoted by r and c:

$$r = \mathcal{P}'_c(l'_c) - \mathcal{P}_r(l_r), \qquad c = \mathcal{P}'_r(l'_r) - \mathcal{P}_c(l_c)$$

[1]The algorithms can handle the other setting (array with row to processor column alignment transferred to array with row to processor row alignment) by swapping the row and column indices for all array sections.

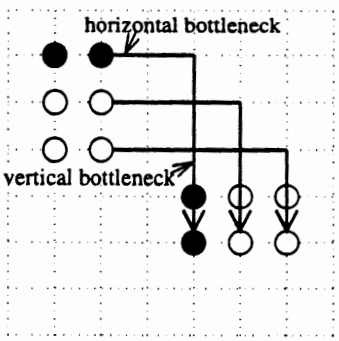


Figure 4.1: Transposition of 3 × 2 nodes with offset (3,3). Contention in both horizontal and vertical channels might result from messages sent by the nodes of the first row. Messages from the same column do not conflict.

Again we consider the non-overlapping case first where the offset is larger than the number of nodes holding the array section in both dimensions. The argument concerning the bottlenecks for shifts in Section 3.1 can be applied here as well, but there is a difference. The nodes occupying the bottleneck channels are always located in the same row. Due to the X-Y routing, the messages from the same column of nodes do not conflict with each other since they use disjoint channels (Figure 4.1). The lower bound on the number of steps hence is $\mathcal{N}_c$, the number of columns.

An algorithm reaching this lower bound can be developed easily: simply schedule the messages according to their senders' column positions.

The general case for transposition is more complex: we have to look at each row separately. Figure 4.2 shows the transposition of 2 × 3 nodes with an offset of (0,1). Using the simple case algorithm (i.e. one step per column), we would need 3 steps. Figures (a) and (b) show a better schedule with only two steps.

4.1.1 Finding a Tight Lower Bound

A tight lower bound equals the size of the largest conflicting set because messages from all nodes of this set have to be transmitted sequentially. First, we find all the conflicting sets of nodes for a single row of the source nodes. We make the following

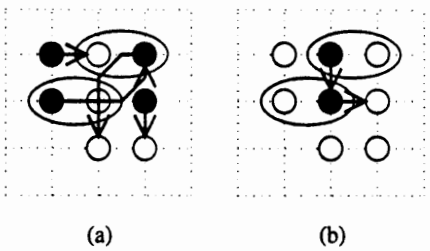


Figure 4.2: Conflicting sets of nodes for transposition of 2×3 nodes with an offset of (0,1). The ovals mark nodes whose messages require the same channel. In the first row, the two nodes on the right side both use the channel downward from the middle node. In the second row, the two nodes on the left side both require the channel to the right of the middle node. Figures (a) and (b) show both steps of an optimal contention-free message transmission.

observation: the most important node in a row is the one where the destination column intersects the source row (if the row and the column do not overlap, extend them so they do intersect). All messages of that row get routed through this node. Looking at Figure 4.3, all nodes to the left of the intersection send their message through channel 1 and all nodes to the right transmit through channel 2. The nodes above and below the intersection get their message through channel 3 and 4. Thus, for this particular row, we have four sets of nodes that might conflict. The sizes of those sets is equal to the number of nodes on each side of the intersection node.

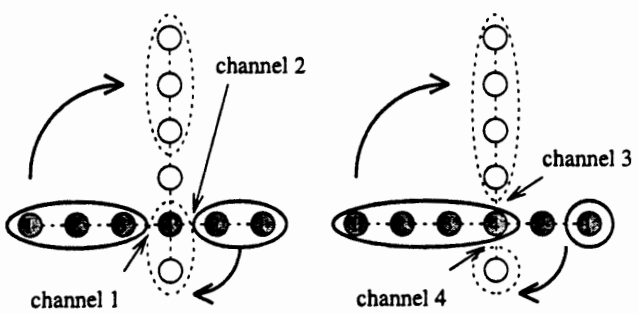


Figure 4.3: Four critical channels with the corresponding sets of nodes. The pictures highlight the nodes transmitting through the same horizontal (left picture) and vertical (right picture) channel. The numbers indicate the sending order.

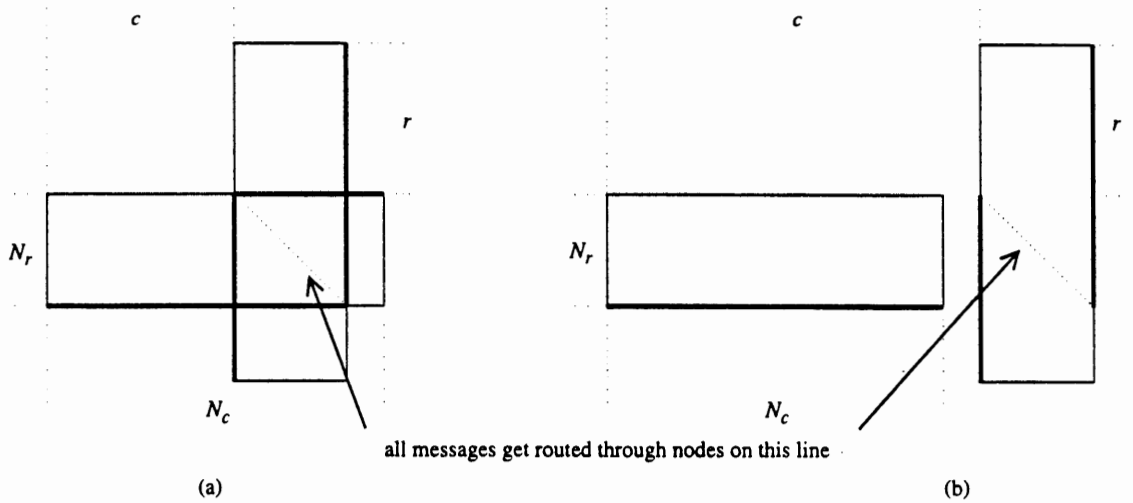


Figure 4.4: Obtaining $S_{\mathbf{cmax}}$ from the parameters $\mathcal{N}_r$, $\mathcal{N}_c$, r and c. It equals the largest number of nodes to either side of one of the intersection nodes, visualized by the four thick lines. Figure (a) shows the case where the matrices overlap and (b) shows an example with non-overlapping matrices.

We also observe that the sets of nodes sending their messages through channels 1 and 2 are aligned to the left and right border of the row. This is also the case for the nodes sending messages through channels 3 or 4, which are marked by the ovals.

Figure 4.4 shows how the *size of the largest conflicting set*, $S_{\mathbf{cmax}}$, can be obtained from parameters $\mathcal{N}_r$, $\mathcal{N}_c$, r and c. All intersection nodes are located on a diagonal line. We have:

$$S_{\mathbf{cmax}} = \mathbf{max}(\ [\mathcal{N}_r - r - 1]^0_{\mathcal{N}_c},\ [\mathcal{N}_r + c - 1]^0_{\mathcal{N}_c},\ [\mathcal{N}_c + r - 1]^0_{\mathcal{N}_c},\ [\mathcal{N}_c - c - 1]^0_{\mathcal{N}_c}\ ) \quad (4.1)$$

where the four terms in the max function represent the maximum number of nodes above, to the left, below, or to the right of an intersection node.

The above equation applies to all cases, whether the source and destination overlap or not since the operator $[\ ]^0_{\mathcal{N}_c}$ keeps the formula for the cardinality of the sets in the range between 0 and $\mathcal{N}_c$. This is visualized by the thick line at the bottom of the source area in Figure 4.4b which represents the largest conflicting set of nodes left of the intersections (marked by the diagonal dotted line). In this case all columns have

to be sent sequentially.

4.1.2 Deriving an Optimal Scheduling Algorithm

To avoid contention, a scheduling algorithm must make sure that nodes from the same conflicting set never send messages at the same step.

Consider the example in Figure 4.3: the three leftmost nodes form a conflicting set (with respect to channel 1) and the four leftmost nodes form another (with respect to channel 3). We have to consider the larger set, i.e. sending messages from four nodes sequentially. The same observation can be made on the right end of the row. We can derive the following schedule: start at both ends of the row and send the messages concurrently. Step two repeats this for the inward neighboring nodes. In step three we are not allowed to send the middle two nodes' messages in parallel because they are in the conflicting set of nodes using channel 3. This yields a scheduling sequence which is represented by the numbers in Figure 4.3. It does not matter whether the largest conflicting set is aligned to the left or to the right. The scheduling scheme is contention-free for both cases. This is important for the application of the scheme to other operations, such as counter clockwise rotation and transposition.

Since we know that messages sent from nodes in different rows do not conflict, we can apply this scheme to all the rows in parallel. The important difference between Algorithm 7 and the simple version of sending the columns sequentially is that two nodes of one row send at the same time, yielding a speedup of up to 100% for a maximal conflict set size of $\mathcal{N}_c/2$. For the previously discussed case where $\min(|r|, |c|) \geq \max(\mathcal{N}_r, \mathcal{N}_c)$, the largest conflicting set has a size of $\mathcal{N}_c$. This means that the condition *right_sender* $> S_{\mathrm{cmax}}$ is never satisfied. Thus, the algorithms behave in the exact same manner. The statement "I have to send a message" checks whether the node's destination is different from the node itself.

Theorem 4 *Algorithm 7 transposes $\mathcal{N}_r \times \mathcal{N}_c$ nodes with an offset (r, c) in S_{cmax} link-contention-free steps, which is optimal.*

Proof: See appendix.

Algorithm 7 (Transpose)
Scheduling algorithm for a transposition. The two parameters row and col denote the node's location in the network

```
Main Program
    if this_node is a source node
        determine S_cmax                                   /* use Equation 3.3 */
        for step = 1 to S_cmax                             /* steps needed */
            left_sender = step
            right_sender = N_c + 1 − step
            if (col − P_c(l_c) = left_sender)              /* sending condition 1 */
              and (I have to send a message)
                send_message_to(row + P'_c(l'_c) − P_r(l_r), col + P'_r(l'_r) − P_c(l_c))
            end if
            if (col − P_c(l_c) = right_sender)             /* sending condition 2 */
              and (right_sender > S_cmax)                  /* not in same conf. set
                                                              as left_sender */
              and (I have to send a message)
                send_message_to(row + P'_c(l'_c) − P_r(l_r), col + P'_r(l'_r) − P_c(l_c))
            end if
        end for
    end if
end Main
```

4.2 Arbitrary Block-Distributions

The regular and the general transfer cases can be described with base patterns again. The sizes of the destination area are exchanged and the patterns can be represented by $\mathcal{R} \times \mathcal{C} \rightarrow \mathcal{C}' \times \mathcal{R}'$. Figure 4.5 shows base patterns for the four different cases. Similarly to the case with identical alignments, Algorithm 7 can be generalized from single nodes to base patterns. The initial and final shifts are applicable for general transfers as well.

Due to the comparably small impact of scheduling on transfers between transposed arrays (see Section 5.3) and the fact that all enumeration methods necessary for the extension have been presented in Chapter 3 already, this chapter focuses only on the main differences.

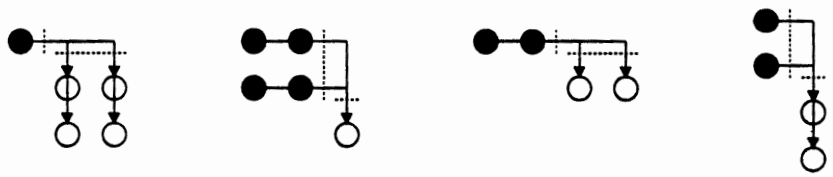


Figure 4.5: The dotted lines show the links available for scatter/scatter, gather/gather, scatter/gather, and gather/scatter (left to right).

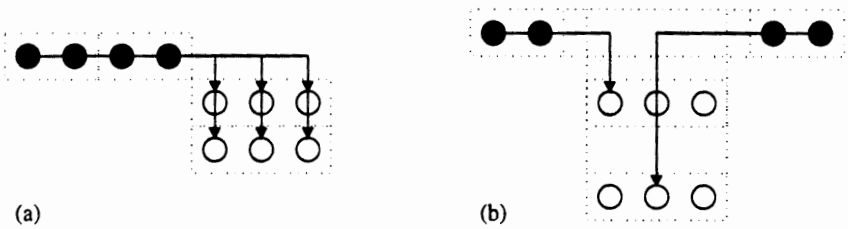


Figure 4.6: Two $1 \times 2 \rightarrow 1 \times 3$ base patterns combined still have one horizontal and three vertical bottlenecks (Figure a). Figure (b) shows how two bottlenecks can be used.

4.2.1 Grouping Base Patterns into Regions

Since base patterns of different rows are independent of one another it only makes sense to group patterns of the same row. However, this does not change the number of bottleneck channels, as illustrated in Figure 4.6a. If the base patterns are located on different sides of the intersection, then they can be sent in parallel even with both destinations being below the intersection (Figure 4.6b).

Figure 4.7 shows how the number of base patterns on each side of the intersection is determined. This number is divided by two if there is more than one channel

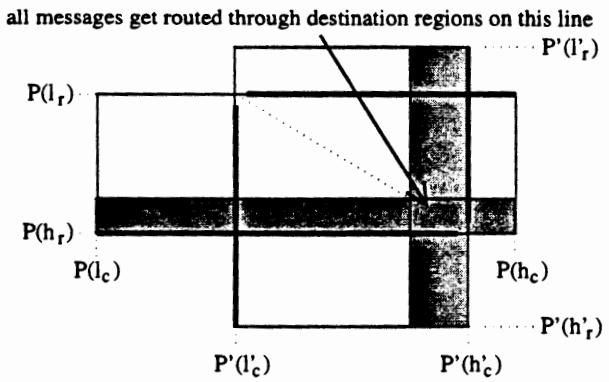


Figure 4.7: Determining the size of the largest section in each direction. All messages from a region have to pass the intersection of the $\mathcal{R}$ rows where the sources located on, and the $\mathcal{R}'$ columns where the destinations are located on (grey areas). The largest section can be determined similarly to the example in Figure 4.4.

available to send or receive messages. The size of the largest section is:

$$sec = \max(sec_{left}, sec_{right}, sec_{up}, sec_{down})$$

$$sec_{left} = \left\lceil \lceil \frac{[\mathcal{P}'(h'_c) - \mathcal{P}(l_c)]^0_{\mathcal{N}_c}}{C} \rceil / \min(\mathcal{R}, 2) \right\rceil \quad sec_{right} = \left\lceil \lceil \frac{[\mathcal{P}(h_c) - \mathcal{P}'(l'_c)]^0_{\mathcal{N}_c}}{C} \rceil / \min(\mathcal{R}, 2) \right\rceil$$

$$sec_{up} = \left\lceil \lceil \frac{[\mathcal{P}(h_r) - \mathcal{P}'(l'_r)]^0_{\mathcal{N}_r}}{C'} \rceil / \min(\mathcal{R}', 2) \right\rceil \quad sec_{down} = \left\lceil \lceil \frac{[\mathcal{P}'(h'_r) - \mathcal{P}(l_r)]^0_{\mathcal{N}_r}}{C'} \rceil / \min(\mathcal{R}', 2) \right\rceil \tag{4.2}$$

4.2.2 Communication Subroutine

Regardless whether the one-to-one, the local gather/scatter, or the hybrid communication subroutine is considered, the implementation must make sure that the extra communication channel is used properly. The following convention assures this: Since at most two base patterns of a row can send concurrently the *right_sender* always uses the additional channel. If this channel is not available, then the *right_sender* must be on the other side of the intersection (the size of the conflicting set was not divided by two).

Chapter 5

Simulation Results for Wormhole Routed Networks

In order to evaluate the performance of Algorithm 1, a network simulator was used. Both the algorithm and unscheduled case were tested.

5.1 The Simulator

A simulator was developed [12] that models wormhole-routed mesh networks and focuses on channel contention. During a shift, all messages are of unit size. Therefore, the time for transmitting one unit message between any pair of nodes without contention is one step in the simulations. All participating nodes in the simulator work in synchronous steps. The receiving of a message is handled implicitly, i.e. no explicit receive step is needed. For simulating a scheduling algorithm, the simulator emulates the communication steps produced by the algorithm. In the ith step, the nodes that have messages scheduled for the step take action; while the other nodes wait. Since the communication schedule is link-contention-free, the number of simulation steps is equal to that of communication steps. Due to the optimality of the algorithm, both the lower bound and the transmission steps are represented by the dotted line in Figure 5.1.

In the unscheduled case, every node performs independently. It is assumed that any node that has a message to send tries to send it as soon as possible and if there are multiple messages competing for the same channel during any simulation step,

one of them is picked randomly to succeed; the other messages get blocked and have to wait for the next step. Since a wormhole-routed network is simulated, a message being blocked at a channel occupies all the channels along the path from the blocking point back to its source node, until it succeeds in reaching its destination.

The random conflict-resolution approach described above causes variations in the number of steps required for a given communication pattern. By chance, the result might be optimal (one that equals the result of an optimal scheduling algorithm), but in many cases, the result can be far from optimal. Taking this property into account, 1000 experiments for each communication pattern were performed and the min, max, and average numbers of steps were recorded. In the figures, they are represented by the vertical bars. The standard deviation for the rightmost bar in all Figures is between two and four. The only exception is Figure 5.2 where the standard deviation is only 0.82 due to the small average number of steps.

5.2 Identical Alignment

First the results for transfers with identically aligned arrays are presented. The following sections cover the three levels of complexity.

5.2.1 Results for Identical Block-Distributions

Figures 5.1a-d and 5.2 show the results for shifting data stored on a square $n \times n$ processor grid with different offsets. For shifting offsets that are larger than the number of source nodes in each dimension (Figure 5.1a), the algorithm needs n steps to transmit the data. Without scheduling, larger array section cause more network traffic (n^2 messages) through relatively fewer bottleneck channels (n). The likelihood that arbitrary sending orders yield a good resource utilization decreases with an increasing number of source nodes. This explains the growing discrepancy between the linear behavior of the scheduled case compared to the non-linear behavior of the unscheduled case which shows a worsening trend with for an increasing number of source nodes.

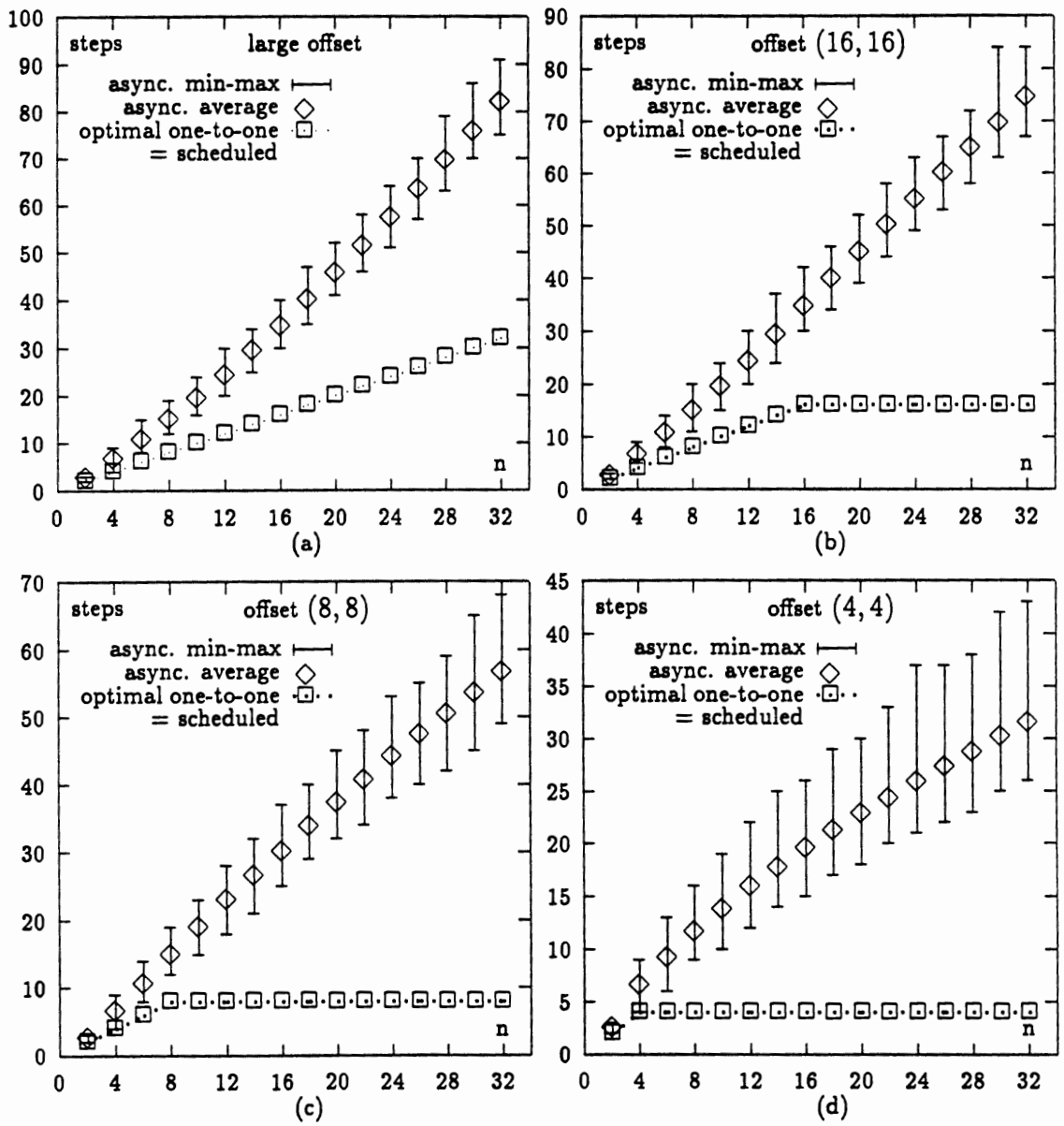


Figure 5.1: Simulation results for a $n \times n \rightarrow n \times n$ shift with different offsets.

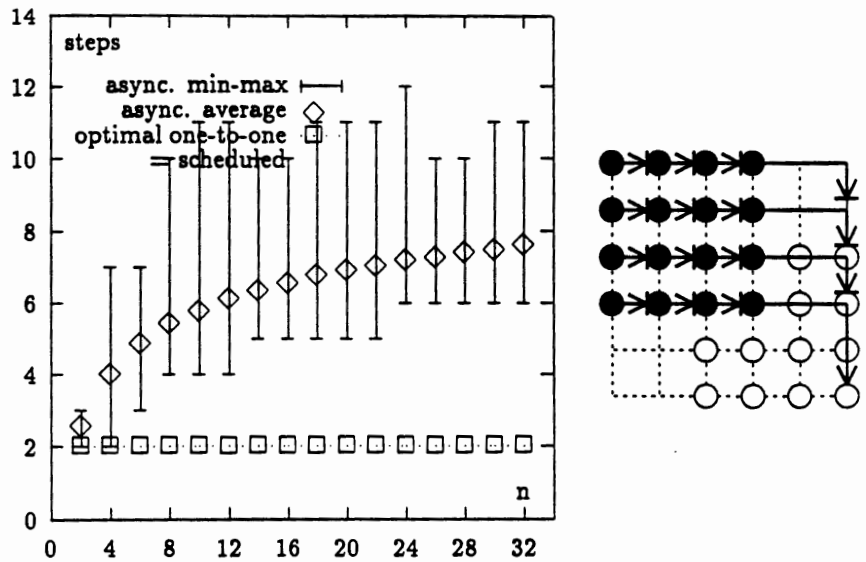


Figure 5.2: Simulation results for a $n \times n \rightarrow n \times n$ shift with an offset of (2,2). The right figure shows the shifting of 4×4 nodes by an offset of (2,2). Each node sends its message at the same time, resulting in heavy contention. Only one message reaches its destination.

Figure 5.1b shows the behavior for an offset of (16,16). For processor configurations up to (16,16) the results are similar to the case with large offset, because the source and the destination matrices do not overlap. From then on, the results without scheduling are only slightly better compared to Figure 5.1a, whereas the algorithm exploits the overlap and requires only 16 steps. Similar observations can be made for Figures (c) and (d) as well.

Figure 5.2 (left figure) shows the results for a similar experiment. The shifting offset was reduced to (2,2). The scheduling algorithm allows all messages to reach their destination after two steps. Thus, the optimal number of steps is not affected by increasing n. The results without scheduling, however, yield dramatically worse results. For example, a configuration of 196 processors has a worst case transmission time that is five times higher. The average number of steps increases only slightly for configurations with more than 196 processors, but at this point it is already more than three times the optimal number of two steps.

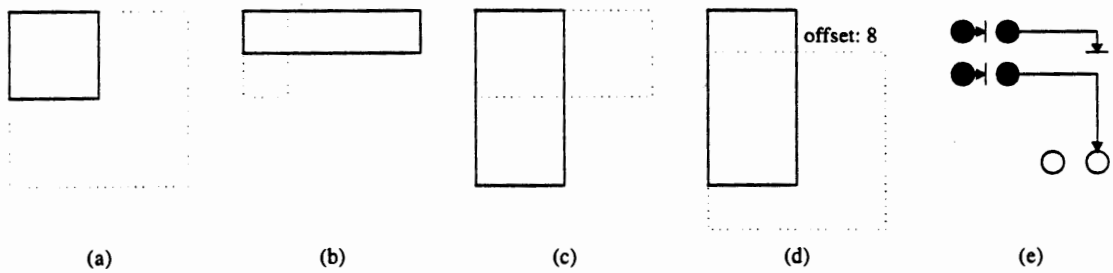


Figure 5.3: Figures (a) through (d) show the positioning of the source arrays (solid lines) and the destination arrays (dotted lines) during the experiments. The experiment in Figure (d) has a constant vertical shift offset. Figure (e) shows how contention causes one link to be unused.

Figure 5.2 (right figure) shows the worst case for the first transmission step. A 4×4 processor grid shifts messages with an offset of (2,2). All messages except for one are blocked while eight can be transmitted concurrently. This explains the very poor behavior of the unscheduled case. Since all paths are short, conflicts only have an effect on a smaller region rather than the whole row or column as it is the case for large offsets. Therefore, the increase of the average number of steps for large matrices has a different character in Figures 5.1 and 5.2.

5.2.2 Results for Regular Block-Distributions

In order to examine only the benefit of avoiding contention in the network, the one-to-one region-communication subroutine is used since it has the same message pattern as the asynchronous transfer. As indicated by the theorem, the algorithm's performance can be slightly worse than the optimal times. Therefore, different lines visualize the times for the scheduled (solid line) and the optimal transfer (dotted line). The minimum, the average, and the maximum times of 1000 test runs of the asynchronous case are represented by the vertical bars.

Figure 5.4a shows the simulation results for the regular transfer from an $n \times n$ to an $2n \times 2n$ processor grid (scatter/scatter) shown in Figure 5.3.[1] The asynchronous

[1]Due to the similar link usage, the gather/gather transfer yields only slightly different results.

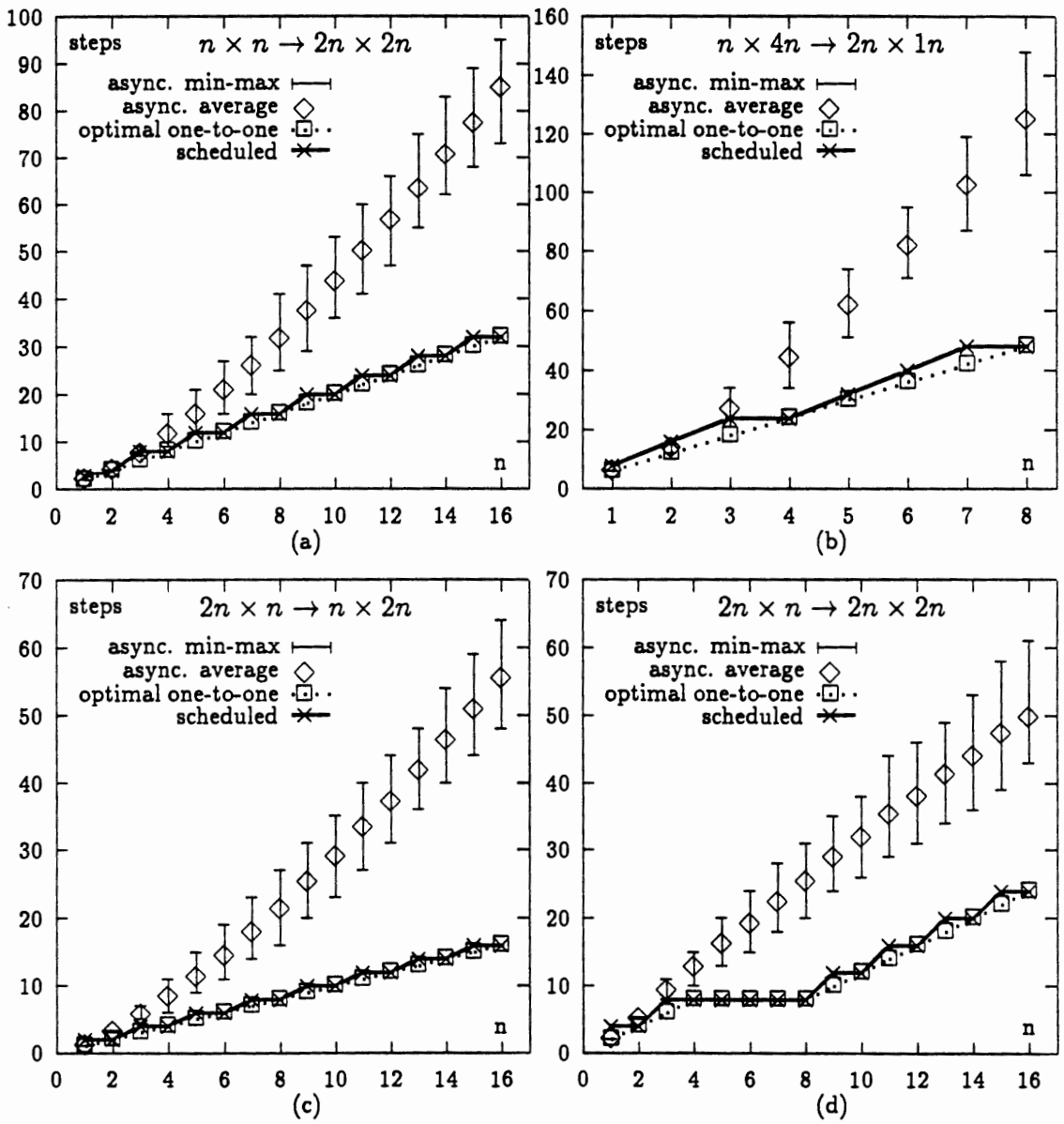


Figure 5.4: Simulation results for regular transfers.

transfer matches the linearly-growing lower bound for small processor configurations, but departs when n gets larger. This indicates that with the growing number of messages sent, it becomes more and more unlikely for the arbitrary conflict resolution to pick the right message to proceed every time. Consequently, link contention occurs and communication bandwidth gets wasted, as shown in Figure 5.3e. The performance of the one-to-one scheduling implementation, matches the lower bound curve quite well, regardless of the value of n. For even n's, the matches are in fact exact. The figure clearly indicates, even without taking full advantage of the multiport architecture, the scheduling algorithm improves the performance substantially.

Figure 5.4b presents results for a scatter/gather operation. Due to the fact that only a single link is available for the scatter/gather pattern, all eight messages of the base pattern must be sent sequentially. Conflicts among those messages do not negatively affect the performance; a feature that brings the curve of the asynchronous transfer mode slightly closer to the lower bound curve. However, the performance improvements produced by the scheduling algorithm is still substantial.

For the gather/scatter operation from Figure 5.4c, two links are available in each direction for the base pattern. Furthermore, the $2n \times n$ source area can be partitioned into a square grid of regions. Those two features allow maximum parallelism for the transfer. Therefore, almost every collision causes worse link-usage. This explains the higher speedup of 3.46 for the largest grid in Figure 5.4c versus the speedup of 2.60 in Figure 5.4b.

The horizontal part of the lower bound curve in Figure 5.4d is caused by the constant vertical offset of eight. Starting from source configuration 8×4, the source and the destination overlap. Due to this overlap, the number of messages that must cross the vertical links, which are the bottleneck for the operation, is constant from this point on. Starting from source configuration 16×8, the horizontal links become the bottleneck and the lower bound rises linearly again. The asynchronous case does not reflect the step-character of the lower bound.

In general it can be concluded that the performance of the scheduling algorithm relative to the asynchronous transfer mode is best when the communication pattern bears a high degree of parallelism. In those cases spontaneously sent messages are

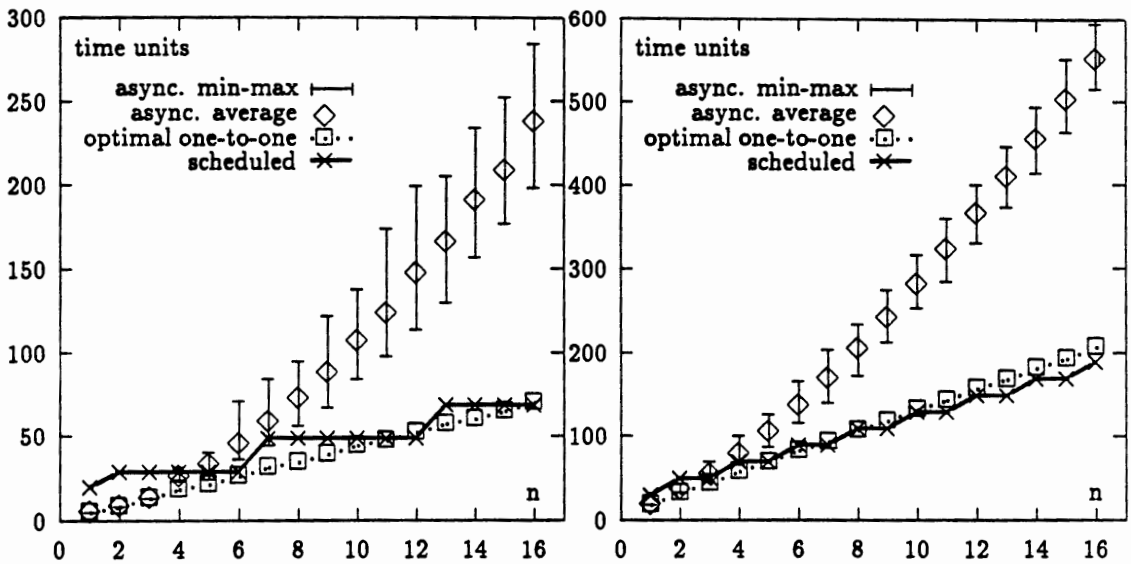


Figure 5.5: Simulation results for a gather/scatter transfer from a $1.5n \times n$ to a $n \times 1.5n$ processor grid (left figure) and for a scatter/scatter transfer from a $n \times n$ to a $2n+1 \times 2n+1$ processor grid (right figure).

unlikely to take full advantage of all available links. The performance is worst if the pattern allows no parallelism and all messages must be sent sequentially anyway.

5.2.3 Results for General Block-Distributions

In both graphs in Figure 5.5, the dotted line represents the lower bound of any non-combining one-to-one communication pattern, which is the maximum number of messages crossing through a single bottleneck channel (determined by counting). Since the main interest is the effect of scheduling, the one-to-one region-communication subroutine is used. Thus, the communication pattern is only modified by the initial and final shifts.

The results in Figure 5.5 show that the scheduling solution increases the performance significantly. Several other simulations showed similar trends. With a growing number of processors, the chances increase that a message is blocked in the network several times. Therefore, the performance of the asynchronous case shows both

a steeper increase and a worsening trend. The transmission time of the scheduled solution increases linearly.

In the simulation, the transmission time β for the smallest message unit is equal to the start-up time α. The source nodes in the first and second simulation hold 9 and 16 units. With these values, the initial and final shifts cause a constant overhead of $(\alpha + 3\beta) + (\alpha + 4\beta) = 9\alpha$ and $2(\alpha + 4\alpha) = 10\alpha$ in the simulations. Figures 3.18a and 3.18b show the base patterns of the simulated transfers. With the initial and final shifts, the average number of messages sent per node is decreased from 4 to 2.25 in Figure 3.18b and from 9 to 4 in Figure 3.18c, saving several message start-ups. In the examples, the start-up time is small compared to the transmission time (1:9 and 1:16). Still the scheduling algorithm even performs better than the lower bound for some configurations. This indicates both that the schedule's resource usage is close to optimal and that the figures reflect mostly the benefit from avoiding link-contention. The performance of the algorithm compared to the asynchronous case increases with growing start-up latency and smaller message sizes. Base pattern implementations other than the one-to-one transfer (e.g. MPI gather/scatter routines) can further optimize the scheduling results.

5.3 Transposed Alignment

This section covers simulations for transpositions. Figure 5.6 shows results for transposition of a single row $(1, \mathcal{N}_c)$ of processors with an offset of $(-\lfloor \mathcal{N}_c/3 \rfloor, 2\lfloor \mathcal{N}_c/3 \rfloor)$. The average performance of the unscheduled case is slightly worse than the optimal number of steps, and the maximum number of steps needed is about 50% higher. Since several different sending orders yield optimal transmission times, during 1000 tries, the optimal number of steps was matched in the unscheduled case for all processor grid sizes. The figure on the right explains how contention slows the transmission down.

The most common case, however, the transposition of an $\mathcal{N}_c \times \mathcal{N}_c$ configuration with no offset, always yields the optimal number of steps without scheduling. The reason for this is that messages that get stuck only block the way for messages of

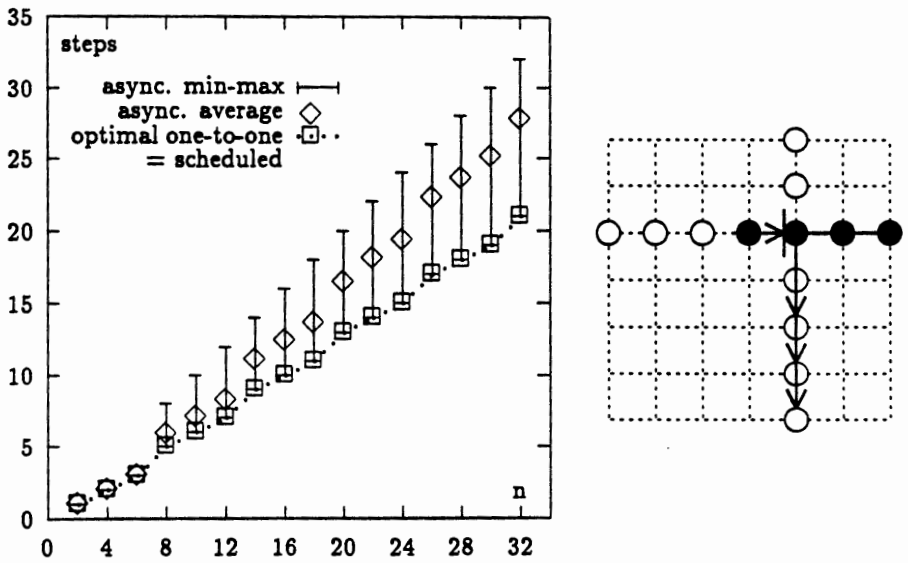


Figure 5.6: The figure on the left shows simulation results for transposition of a single row with offset. The figure on the right shows the worst case for the transmission: the fourth node of the row has to wait until the three right nodes complete their transmission, blocking the way for the three left nodes, which could send concurrently. Seven rather than four steps are needed.

the same conflicting set, which have to go sequentially anyway. Therefore, the actual transmission dynamics are only a permutation of the sending order imposed by the scheduling algorithm. On networks with switching strategies other than wormhole-routing, buffering and message retreat caused by collisions incurs a large overhead. Thus, the scheduling still pays off.

Chapter 6

Conclusion

6.1 Summary

This thesis analyzed communication patterns that are generated by array operations in data-parallel languages. Depending on the array section and distribution parameters, the induced communication ranges from simple shifts up to several overlapping many-to-many patterns with different sizes for the source and destination areas. This analysis is helpful for the development of scheduling solutions for other platforms. Furthermore, it provides guidelines on how to distribute data in order to minimize the communication overhead.

The diagonal scheduling scheme is a powerful tool to schedule this type of communication. For the more complicated patterns, this approach was augmented by with region communication subroutines. This modularity allows convenient fine tuning to optimize the performance for different hardware parameters, message sizes, and transfer patterns. The hybrid subroutine allows doing this even at run-time. The algorithms support the generation of efficient code for data-parallel languages which is a major problem in the field of parallel processing. The general concepts such as the diagonal scheduling scheme, bottleneck channels, and conflicting sets can be used to analyze and schedule other patterns.

The generated communication schedule was proven to be either optimal or very close to optimal with respect to the number of steps. The for loops in the algorithm perform one iteration per sending step along with some initial arithmetic operations. Since more complicated operations are necessary anyway in order to determine the

local address sequence and the set of destinations, the algorithm's runtime overhead does not play a significant role in the overall process.

The algorithms were compared to the asynchronous transfer case, meeting the evaluation requirements suggested by Panda[33]. Several test cases showed that the scheduled transfer increases the performance significantly, especially for large processor configurations. Since the one-to-one region communication subroutine was used in the simulations, the pure benefit of avoiding link-contention was demonstrated. Further improvements are possible with the hybrid algorithm.

6.2 Related Work

This section gives a brief overview of the field of communication algorithms and points out the differences as well as useful concepts and ideas that were used in this work.

Ranka and Wang[34, 35, 42, 43] proposed communication scheduling algorithms which produce node-contention-free schedules. Due to the assumption that no knowledge about the structure of the communication pattern and the underlying network is known, each node has to collect and evaluate the entire global communication pattern before starting the data transfer. This approach would also be considered dynamic, however, it is no longer distributed since information local to each node needs to be exchanged in the scheduling process. In the setting of this thesis, knowledge about the operation to be performed and the network architecture is encoded in the algorithm a priori. Giving up some flexibility makes it possible to speed up array operations by selecting communication schedules with low overhead and high resource utilization.

Chatterjee et al. and Kennedy et al. provide complementary work for this thesis by addressing the problem of determining source/destination pairs and generating communication sets for arbitrary array section parameters and alignments. Furthermore, they provide solutions to the problem of whether to perform a binary array operation at the location of the first or the second operand [5, 6, 23]. In this thesis, those issues were omitted, but for an actual implementation those algorithms are inevitable.

Barnett et al. examine the performance of collective communication algorithms[1, 2, 3] for varying parameters such as message size, channel bandwidth, and startup latency. The objective is to determine which algorithm should be applied for a specific environment. Some of their methods were used to determine the optimal degree of message combination for the hybrid region-communication subroutine implementation.

One-to-many communication patterns on meshes have been thoroughly studied by several research groups, and different algorithms that implement contention-free schedules for those patterns have been suggested [28, 11, 41]. Concepts and solutions from those papers[1, 31] can be used to for the implementation of the local scatter and gather operations. McKinley et al. as well as Ho and Kao show how architectural features allow speedups for those operations[21, 36, 40]. Those papers show interesting solutions besides recursive halving and doubling that allow further optimizations. Publications by Scott [38] and Sundar et al.[39] cover all to all communication on meshes which is particularly important once scheduling algorithms are developed for cyclic and multidimensional distributions.

6.3 Future Work

The work presented is applicable to the problem of optimizing the communication for array operations on a specific hardware platform. In this section, some ideas on how the concepts and basic ideas should be extended.

Array section operations are commonly used in data parallel programs. However, the algorithms could be extended to schedule some similar patterns with a high message density, such as those occurring during the simulation of an artificial neural network on a MPP system.

An important goal is to extend the algorithms to different topologies such as hypercubes, fat trees, or even clusters of workstations with an ATM network[22]. Routing strategies other than X-Y could be considered as well. The approach to avoid node contention from Section 3.1.4 needs to be extended to the general transfers.

One solution to the problem of a worm blocking a whole channel is the concept

of virtual channels [8, 9, 10]. This allows several messages to share a single link. Virtual channels are used by several previously suggested algorithms[32, 41]. Different implementations of virtual channels have been suggested. Two or three virtual channels might be used to avoid deadlock in mesh networks with wraparound channels (torus)[37]. In [30] virtual channels allow several messages to share links. This concept would prevent some messages from getting stuck in the network. But this concept is also not without drawbacks: if two paths share only one physical link, the bandwidth of both paths is only half of the original bandwidth, even though all other channels are not shared. The bottleneck of one shared channel slows the whole transmission down, wasting resources. Furthermore, the number of virtual channels is limited, so there still is the possibility of contention. The exact effects of different implementations of virtual channels have to be analyzed carefully, but again scheduling will pay off in most cases.

The effects of link contention in networks with virtual cut through and circuit switching have to be examined, in order to predict the benefit of the scheduling. The conflict resolution of these switching strategies (buffering and retreat-retry) has a much higher latency compared to wormhole routing. This indicates that the unscheduled transfer will take much longer for these strategies compared to the unscheduled transfer with wormhole routing.

Appendix A

Proofs of the Theorems

A.1 Proof of Theorem 1

Theorem 1 *Algorithm 1 shifts the data of $\mathcal{N}_r \times \mathcal{N}_c$ source nodes in* max(*sec_ver*, *sec_hor*) *link-contention-free steps, which is optimal.*

Proof: Define the row and column index of nodes relative to the node holding the upper left array section element:

$$arr\_row = row - \mathcal{P}_r(l_r)$$
$$arr\_col = col - \mathcal{P}_c(l_c)$$

First it is shown that the communication steps are contention-free. Define: $dia = section\_row - section\_col + 1$. The set of sender nodes for the ith step can be described as:

$$S_i = \begin{cases} \{(arr\_row, arr\_col) : i = dia\} & \text{if } dia > 0 \\ \{(arr\_row, arr\_col) : i = dia + \max(sec\_ver, sec\_hor)\} & \text{if } dia \leq 0 \end{cases}$$

Since conflicts can only occur among nodes of the same row or column, two cases are considered. In Case 1, the row is fixed, i.e. the nodes in S_i which have the same *arr_row* are examined. Given this condition, it can be inferred that for these nodes, the *section_row* is also fixed. Furthermore, according to the definition of S_i, *section_col* must also be a constant. Since

$$section\_col = arr\_col \bmod sec\_hor,$$

arr_col must have values in the form of $x \cdot sec\_hor + section\_col$. In other words, the senders from the same row must appear in columns that are *sec_hor* steps apart. Therefore, collisions in horizontal channels are impossible. Analogous argument can be applied to Case 2, where the column is fixed, resulting in a conclusion that collisions in vertical channels are not possible, either.

Each node's *diagonal_num* is between 1 and $\max(sec\_ver, sec\_hor)$ and therefore, the sending condition is fulfilled for each node exactly once duing the algorithm execution. Since no contention occurs, all messages reach their destination without delay. This proofs the first part of the theorem.

Now only the optimality of $\max(sec\_ver, sec\_hor)$ must be proven. Consider all the possible values for the offset $off := \mathcal{P}'_c(l'_c) - \mathcal{P}_c(l_c)$ in Equation 3.3, (1) if $0 < off < \mathcal{N}_c$, then the rightmost *off* nodes of each row have to send through the bottleneck channel to the right of the rightmost node; (2) if $off \geq \mathcal{N}_c$, all $\mathcal{N}_c$ nodes use one channel; (3) if $off < 0$, the situation is symmetrical; and (4) if $off = 0$, no conflicts occur. Thus, for each of these cases, *sec_hor* steps are necessary. Analogous analysis shows that *sec_ver* steps are necessary as well. The larger of these two values hence is a lower bound for the number of steps. The algorithm is optimal since it matches this lower bound. □

A.2 Proof of Theorem 2

Theorem 2 *If the largest quotient (one of sec_left, sec_right, sec_up, or sec_down) determining max_section_size in Equation (3.7) has no remainder, then the schedule generated by the algorithm is optimal with respect to the number of data transfer steps. If the quotient has a remainder, then the algorithm wastes fewer than reg steps.*

Proof: The quotient n/r with n nodes and region size r determines *max_section_size*. Case 1: Due to the definition of sections and the condition $r|n$, all n nodes require a common bottleneck channel c to send or receive their data. The n nodes are partitioned into n/r regions, one of them always performing a one-to-one region-communication subroutine that must use c. Thus, the schedule is optimal since

the bottleneck is permanently used for data transfer. Case 2: In the worst case, $n \bmod r = 1$ and the algorithm sends or receives the data from $r - 1$ nodes sequentially that could have been handled concurrently with other nodes (in Figure 3.9b, for example, section right is one node larger than it has to be). The loss is smaller than *reg*, the time to send one region (r nodes). □

A.3 Proof of Theorem 3

Theorem 3 *If all clusters consist of at least one node, then base patterns that are (1) diagonally aligned or (2) located at the same relative position within adjacent sections of* (*sec_ver* × *sec_hor*) *nodes do not collide.*

Proof: (1) Since base patterns do not overlap, all sending nodes must be in disjunct rows and receiving nodes must be in disjunct columns. Thus, with X-Y routing no conflicts can occur. (2) Single nodes that send or receive concurrently are *offset* columns apart (they belong to some base patterns i and $i + offset$ and are located in columns $c + i$ and $c + i + offset$). Since all clusters consist of at least one node, the clusters belonging to these two base patterns are at least *offset* columns apart. Therefore, collisions can only occur if a message from the left base pattern in column $c + i$ travels further than column $c + i + offset$ or if a message from the right base pattern travels further than column $c + i$. Assume the second option takes place (Figure A.1). Then a message of base pattern *offset* must travel further than column c, the location of the leftmost base pattern's single node. This is a contradiction to the definition of *offset* which is the maximum of *first* and *last*, the pattern offsets on each side. Analogously, conflicts for the second option can be excluded. The proof is completed by repeating the arguments for the vertical dimension. □

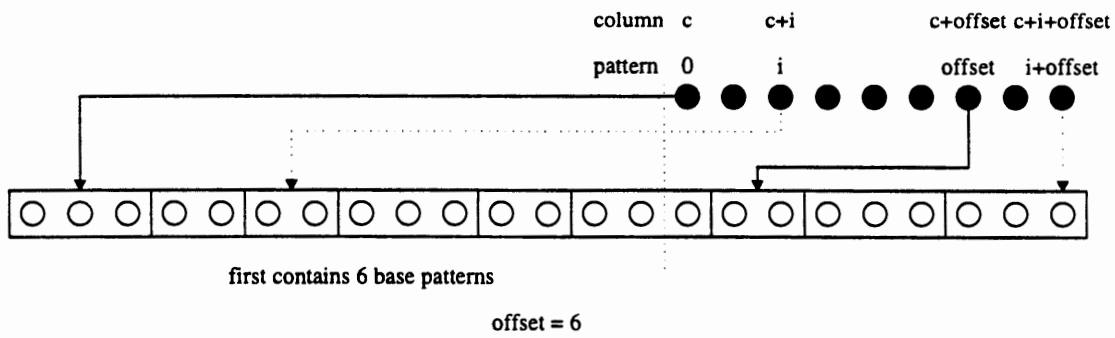


Figure A.1: The definition of *offset* guarantees that no message crosses more than 6 base patterns.

A.4 Proof of Theorem 4

Theorem 4 *Algorithm 7 transposes $\mathcal{N}_r \times \mathcal{N}_c$ nodes with an offset (r, c) in S_{cmax} link-contention-free steps, which is optimal.*

Proof: First it is shown that the communication steps are contention-free. Assume the contrary, that two messages collide in horizontal channels. Then the two *sending conditions* in Algorithm 7 must be satisfied for two nodes of the same row. These two nodes must also be located on the same side of the intersection of that row with its destination column. Since S_{cmax} is the size of the largest conflicting set, it must be greater than number of nodes to the left of the intersection. With the *sending condition right_sender* $> S_{\text{cmax}}$, it can be concluded that *right_sender* cannot be on the left side. Furthermore, with

$$\textit{left\_sender} = \textit{step} = \mathcal{N}_c + 1 - \textit{right\_sender} \leq \mathcal{N}_c - S_{\text{cmax}}$$

it can be seen that *left_sender* cannot be on the right side of the intersection. Analogous arguments exclude contention in the vertical channels. Thus, the algorithm is link-contention-free.

If $S_{\text{cmax}} \geq \frac{\mathcal{N}_c}{2}$ then

$$\{1, 2, ..., S_{\text{cmax}}\} = \{\text{values of } \textit{left\_sender}\}$$

$$\{S_{\mathsf{cmax}}+1, S_{\mathsf{cmax}}+2, ..., \mathcal{N}_c\} = \{\text{values of } \textit{right\_sender}\}$$

and all columns get to send to their destination. Only one constellation remains, in which S_{cmax} can be smaller. Equation 4.1 shows that $\mathcal{N}_r$ must be as small as possible ($\mathcal{N}_r = 1$) to obtain a minimal S_{cmax}. Thus, a single row gets transposed. If r and c are not between 0 and $\mathcal{N}_c - 1$, then the largest conflicting set automatically has size $\mathcal{N}_c$. Thus, the offset must be between those bounds in order to get a minimal S_{cmax}. Now Equation 4.1 can be rewritten for this case:

$$\begin{aligned} S_{\mathsf{cmax}} &= \mathsf{max}(-r, c, \mathcal{N}_c + r - 1, \mathcal{N}_c - c - 1) \geq \\ &\geq \tfrac{-r+c+\mathcal{N}_c+r-1+\mathcal{N}_c-c-1}{4} = \tfrac{\mathcal{N}_c-1}{2} \end{aligned} \tag{A.1}$$

Equality can only be achieved if all four arguments of the max operator are equal. This yields $-r = c = \frac{\mathcal{N}_c-1}{2}$. Since the offset must be an integer, $\mathcal{N}_c$ also has to be odd. In this situation the sending conditions for node $(1, \frac{\mathcal{N}_c+1}{2})$ are not fulfilled. However, it does not need to send a message because its data must remain in the same location anyway: $(\frac{\mathcal{N}_c+1}{2} - \frac{\mathcal{N}_c-1}{2}, 1 + \frac{\mathcal{N}_c-1}{2}) = (1, \frac{\mathcal{N}_c+1}{2})$. Since no contention occurs all messages reach their destination without delay. This proofs the first part of the theorem.

Now only the optimality of $\mathsf{max}(\ [\mathcal{N}_r - r - 1]^0_{\mathcal{N}_c},\ [\mathcal{N}_r + c - 1]^0_{\mathcal{N}_c},\ [\mathcal{N}_c + r - 1]^0_{\mathcal{N}_c},\ [\mathcal{N}_c - c - 1]^0_{\mathcal{N}_c}\ )$ remains to be proven. The four terms reflect numbers of nodes on each side of the intersection of some source rows and their destination columns. Since messages to or from those nodes must be sent sequentially each term is a lower bound. The algorithm matches the tightest lower bound and therefore, it must be optimal. □

Appendix B

The WARP Simulator

Wormhole-routed mesh ARchitecture simulation Program (WARP) is a tool to model the message flow on a message-passing multicomputer [12]. Each virtual node runs its copy of the node program that generates messages (Multiple Instruction Multiple Data - MIMD).

WARP models bidirectional 2D mesh networks with physical bidirectional interconnections. Messages crossing the same link in opposite directions do not collide or share bandwidth. The network is a multiport architecture, which means that nodes can receive and send messages at the same time.

Time Model WARP uses a discrete time model to simulate the message flow in the network. The transmission process is partitioned into equal time-intervals of length δ. All participating nodes in the simulator work in synchronous steps. A message can only be sent at some time $n \cdot \delta$ where $n \in N$. The message remains in the network for some $\triangle t \cdot \delta$, with $\triangle t \in N$.

Communication Model Messages are sent with start-up cost (latency) and bandwidth. WARP uses a simple model that assumes the time to send a message of L units to be: $\alpha + L\beta$. Links are occupied during the whole transmission time, including the latency.

Latency-Transmission Time Ratio Both the latency α and the pure transfer $L\beta$ must be incorporated in the total time specified for a message. If all messages are of

unit size, then the time can always be one. If two message types are used, one twice as large as the other and $\alpha = \beta$, then the times should be $\alpha + \beta = 2$ and $\alpha + 2\beta = 3$.

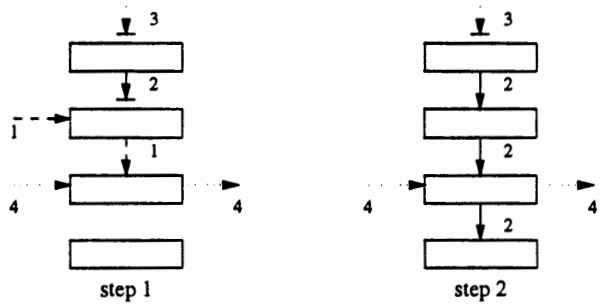


Figure B.1: Transmission dynamics of four messages in the network. Message 1 (dashed line) occupies the channel from the second to the third node blocking message 2 (solid line). Message 2 blocks message 3 (dotted line). Message 4 does not interfere with other messages because it crosses the third node horizontally. In the second step message 1 left the network, message 2 proceeds in the network while message 3 is still blocked.

Conflict Resolution If there are multiple messages competing for the same channel during any simulation step, one of them is picked randomly to succeed; the other messages get blocked and have to wait for the next step. Since a wormhole-routed network is simulated, a message being blocked at a channel occupies all the channels along the path from the blocking point back to its source node, until it succeeds in reaching its destination (Figure B.1).

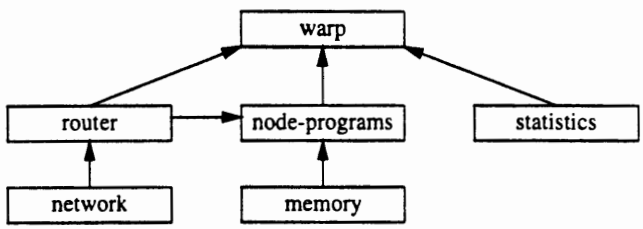


Figure B.2: WARP's module tree.

Implementation WARP is written in C. The simulations were performed on the Computer Science Department's Workstations. Figure B.2 shows the module tree of the program. The individual modules are:

- The *Network Module* is responsible for the representation of the network with its channels. It provides functions to acquire and release channels.
- The *Router Module* handles the router of each node where messages are buffered until they enter the network.
- The *Statistics Module* contains all the functions and variables to evaluate the performance of the simulation process.
- The *Node Program Module* contains the node program which is a C-function. This function is called for every source node in every simulation step. The index of the node and the number of the current step are passed as parameters. With this information, the body of the node program can generate messages.
- The *Memory Module* contains functions supporting additional local memory for each node to support the node programs and simulate real data transfer.
- The *Main Module* holds the simulation control.

Performance The time to simulate a complete message transfer depends on the number of source nodes, the number of messages sent by each node, the length of the message paths, and the number of time steps it takes to complete the transfer:

$$\textit{run-time} = \mathcal{O}(\textit{sources} \cdot \textit{messages} \cdot \textit{pathlength} \cdot \textit{steps})$$

The number of steps obviously must be somewhere between one (all messages are sent in parallel) and $\textit{sources} \cdot \textit{messages}$ (all messages are sent sequentially). On a Sun Sparc 2 workstation, simulating 1000 unscheduled shifts of 32×32 source nodes by offset $(32, 32)$ on a 64×64 mesh takes about 1 hour 28 minutes. The same simulation with 16×16 source nodes and offset $(16, 16)$ only takes 10 minutes.

Bibliography

[1] M. Barnett, S. Gupta, D. G. Payne, L. Shuler, R. A. van de Geijn, and J. Watts. Building a high-performance collective communication library. In *Proceedings of Supercomputing '94*, pages 107–116, 1994.

[2] M. Barnett, R. Littlefield, D. G. Payne, and R. A. van de Geijn. Global combine algorithms on mesh architectures with wormhole routing. In *7th International Parallel Processing Symposium, Newport Beach CA*, pages 156–162, April 1993.

[3] M. Barnett, D. G. Payne, R. A. van de Geijn, and J. Watts. Broadcasting on meshes with worm-hole routing. Technical report, University of Texas, Department of Computer Sciences, 1993. submitted to Journal of Parallel and Distributed Computing.

[4] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, Fall 1992.

[5] S. Chatterjee, J. R. Gilbert, F. J. E. Long, R. Schreiber, and S. Teng. Generating local addresses and communication sets for data-parallel programs. In *Proceedings of the Forth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 149–158, 1993.

[6] S. Chatterjee, J. R. Gilbert, R. Schreiber, and S. Teng. Optimal evaluation of array expressions on masssively parallel machines. In *Proceedings of the Second Workshop on Languages, Compilers, and Runtime Environments for Distributed Memory Multiprocessors*, 1992.

[7] Thinking Machines Corporation. CM-5: the connection machine CM-5 technical summary. Technical report, Thinking Machines Corporation, Cambridge, Massachusetts, November 1992.

[8] W. J. Dally. Virtual-channel flow control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194–205, March 1992.

[9] W. J. Dally and C. L. Seitz. The torus routing chip. *Distributed Computing*, 1(3):187–196, 1986.

[10] W. J. Dally and C. L. Seitz. Deadlock free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, 36(5):547–553, May 1987.

[11] D.F.Robinson, P. K. McKinley, and B.H.C.Cheng. Optimal multicast communication in wormhole-routed torus networks. *IEEE Transactions on parallel and distributed systems*, 6(10):1029–1042, October 1995.

[12] A. Eberhart. The WARP simulator for wormhole-routed 2D mesh networks. Technical report, Portland State University, Department of Computer Science, 1995.

[13] Message Passing Interface Forum. Document for standard message-passing interface. Technical Report CS-93-214, University of Tennessee, November 1993.

[14] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors.* Prentice Hall, 1988.

[15] G. C. Fox, S. Hiranandani, K. Kennedy, C. Koebel, U. Kremer, C.-W.Tseng, and M.-Y. Wu. Fortran D language specification. Technical report, Department of Computer Science, Rice University, Houston, TX, 1990.

[16] D. V. Hall. *Hardware for fast global operations on distributed memory multicomputers and multiprocessors.* PhD thesis, Portland State University, 1994.

[17] D. V. Hall and M. A. Driscoll. Hardware for fast coordination of distributed memory multicomputers. In *Proc. of the 9th Int. Parallel Processing Symp.*, pages 673–679, April 1995.

[18] D. V. Hall and M. A. Driscoll. Hardware for fast global operations on workstation cluster multicomputers. In *Proc. of the 15th Int. Conf. on Distributed Computing Systems*, pages 475–482, May 1995.

[19] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, 1992.

[20] C.-T. Ho and S. L. Johnsson. Distributed routing algorithms for broadcasting and personalized communication in hypercube. In *Proceedings of the 1986 International Conference on Parallel Processing*, 1986.

[21] C. T. Ho and M. Y. Kao. Optimal broadcast in all-port wormhole-routed hypercubes. In *IEEE Transactions on Parallel and Distributed Systems*, pages 200–204, 1995.

[22] C. C. Huang and P. K. McKinley. Communication issues in parallel computing across ATM networks. Technical Report MSU-CPS-94-34, Department of Computer Science, Michigan State University, East Lansing, Michigan, 1994.

[23] K. Kennedy, N. Nedeljkovic, and A. Sethi. A linear-time algorithm for computing the memory access sequence in data-parallel programs. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 102–111, 1995.

[24] P. Kermani and L. Kleinrock. Virtual cut-through: A new computer communication switching technique. *Computer Networks*, 3:267–286, 1979.

[25] C. Koelbel. Compile-time generation of regular communications patterns. In *Proceedings of Supercomputing '91*, Minneapolis, MN, November 1991.

[26] R. K. Koeninger, M. Furtney, and M. Walker. A shared memory MPP from Cray Research. *Digital Technical Journal*, 6(2), Spring 1994.

[27] J. Li and M. Chen. Compiling communication-efficient programs for massively-parallel machines. *IEEE Trans. on Parallel and Distributed Systems*, 2(3), July 1991.

[28] X. Lin and L. Ni. Multicast communication in multicomputer networks. Technical Report MSU-CPS-ACS-19, Department of Computer Science, Michigan State University, East Lansing, Michigan, August 1990.

[29] D. B. Loveman. High Performance Fortran. In *IEEE Parallel and Distributed Technology, vol. 1*, pages 25–42, December 1993.

[30] P. K. McKinley and Christian Trefftz. MultiSim: A simulation tool for the study of large-scale multiprocessors. In *Proceedings of the 1993 International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Networks (MASCOTS)*, pages 57–62, San Diego, California, January 1993.

[31] P. K. McKinley, Y. Tsai, and David F. Robinson. A survey of collective communication in wormhole-routed massively parallel computers. Technical Report MSU-CPS-94-35, Department of Computer Science, Michigan State University, East Lansing, Michigan, June 1994.

[32] L. M. Ni and P. K. McKinley. A survey of wormhole routing techniques in direct networks. *IEEE Computer*, 26(2):62–76, February 1993.

[33] D. K. Panda. Issues in designing efficient and practical algorithms for collective communication on wormhole-routed systems. In *Proceedings of the 1995 ICPP Workshop on Challenges for Parallel Processing*, August 1995.

[34] S. Ranka and J. Wang. Static and runtime scheduling of unstructured communication. Technical report, University of Texas, Department of Computer Sciences, July 1993.

[35] S. Ranka, J. C. Wang, and Manoj Kumar. Static and Runtime Algorithms for All-to-Many Personalized Communications on Permutation Networks. In *Proceedings of the 1992 International Conference on Parallel and Distributed Systems, HsinChu, Taiwan*, pages 211–218, December 1992.

[36] D. F. Robinson, D. Judd, P. K. McKinley, and B. H. C. Cheng. Efficient collective data distribution in all-port wormhole-routed hypercubes. In *Proceedings of Supercomputing'93*, pages 792–801, Portland, Oregon, November 1993.

[37] D. F. Robinson, P. K. McKinley, and B. H. C. Cheng. Optimal multicast communication in torus networks. In *Proceedings of the 1994 International Conference on Parallel Processing*, St. Charles, Illinois, August 1994.

[38] D. Scott. Efficient all-to-all communication patterns in hypercube and mesh topologies. In *Proceedings of the Symposium on Parallel and Distributed Processing*, pages 398–403, 1991.

[39] N. Sundar, D. N. Jayasimha, D. K. Panda, and P. Sadayappan. Complete exchange in 2D meshes. In *Scalable High Performance Computing Conference*, pages 406–413, 1994.

[40] Y. Tsai and P. K. McKinley. A broadcast algorithm for all-port wormhole routed torus networks. In *Proceedings of Frontiers'95: Symposium on the Frontiers of Massively Parallel Computation*, McLean, Virginia, February 1995.

[41] Y.-C. Tseng, D. K. Panda, and T.-H. Lai. A trip-based multicasting model in wormhole-routed networks with virtual channels. In *IEEE Transactions on Parallel and Distributed Systems*, 1995.

[42] J. C. Wang. *Load Balancing and Communication Support for Irregular Problems.* PhD thesis, Syracuse University, December 1993.

[43] J. C. Wang, T. H. Lin, and S. Ranka. Distributed Scheduling of Unstructured Collective Communication on the CM-5. In *Proceedings of the 27th Hawaii International Conference on System Sciences, Maui, Hawaii, Volume II*, pages 483–492, January 1994.