TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

Dissertation

SPIRAL/DMP:

A Generator for Optimized Parallel Signal Transforms

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Doktors der technischen Wissenschaften
unter der Leitung von

Ao. Univ.-Prof. Dipl.-Ing. Dr. techn. Christoph W. Überhuber
E101 – Institut für Analysis und Scientific Computing

eingereicht an der Technischen Universität Wien
Fakultät für Informatik

von

Dipl.-Ing. Andreas Bonelli

Matrikelnummer 9926348
Margaretenstrasse 5/27
A-1040 Wien

Wien, am 5. September 2006

Kurzfassung

Die *diskrete Fourier-Transformation* (DFT) stellt eines der bedeutendsten mathematischen Werkzeuge in den Naturwissenschaften und der Technik dar. Sie ist aus einem überaus breiten Spektrum von Anwendungen, wie z.B. der Lösung partieller Differentialgleichungen der Physik, nicht wegzudenken. Andere Anwendungsgebiete findet man z.B. in der Geophysik, in den verschiedensten Formen der digitalen Signalverarbeitung, wie etwa der digitalen Bildverarbeitung, oder nicht zuletzt bei der Proteinfaltung in der Molekularbiologie.

Lange wurde vermutet, dass die Komplexität der DFT bei $O(N^2)$ läge. 1965 veröffentlichten Cooley und Tukey [11] einen Algorithmus, der als die *schnelle Fourier-Transformation* (*fast Fourier transform*, FFT) bekannt wurde. Dieser Algorithmus vermindert die Komplexität der DFT von $O(N^2)$ auf $O(N \log N)$.

Die Leistung von Algorithmen für einen Prozessor wird im Allgemeinen an der Anzahl der vom Prozessor ausgeführten Arbeitsschritte pro Zeiteinheit gemessen. Bei Programmen für parallele Rechnerarchitekturen geht auch die Zeit, die für den Datenaustausch zwischen den Prozessoren benötigt wird, maßgeblich in die Leistungsbewertung ein. Dabei hängt das Verhältnis zwischen der Rechen- und der Kommunikations-Leistung sehr stark vom verwendeten Computersystem ab.

Die vorliegende Arbeit stellt Spiral/dmp vor, ein Software-System zum automatischen Generieren und Optimieren von Algorithmen und Codes für die effiziente Ausführung der DFT auf Parallelrechnern. Spiral/dmp wurde als Erweiterung des Programm-Generierungs- und Optimierungs-Systems Spiral implementiert. Basis dieses Systems ist die Darstellung von DFT-Algorithmen in der problemspezifischen Sprache Spl. Spiral wurde mit Hilfe eines Markierungs- und Formel-Manipulations-Systems erweitert, um parallele Algorithmen darstellen, herleiten und optimieren zu können.

Spiral/dmp hat die besondere Eigenschaft, skalierende DFT-Algorithmen generieren zu können, die in der Lage sind, eine automatische Datenumverteilung auf die optimale Anzahl an Prozessoren durchzuführen, um den Kommunikationsaufwand so weit wie möglich zu senken. Eine Besonderheit des neu entwickelten Verfahrens liegt darin, dass die Umverteilung der Daten während der, für die Berechnung der DFT unvermeidlichen Kommunikation durchgeführt wird, und dadurch keine *zusätzlichen* Kommunikationsschritte notwendig sind. Spirals Suchmechanismus findet heraus, welche der möglichen Algorithmen und deren Implementierungen auf einer gegebenen Plattform die besten Leistungswerte erzielen. Auf diese Weise wird plattformoptimierter DFT-Code automatisch erzeugt. Experimente mit den von Spiral/dmp generierten Mpi-Programmen ergaben mit den skalierenden DFT-Algorithmen eine Leistungssteigerung von bis zu 30%.

Abstract

The *discrete Fourier transform* (DFT) is one of the principal algorithmic tools in the field of scientific computing. DFT methods can be used to solve various problems in science and engineering. For example, the DFT is an essential tool in digital signal processing. Moreover, DFT methods are heavily used for the numerical solution of partial differential in mathematical physics, like the ones arising in computational fluid dynamics. Other applications of DFT methods occur in geophysical research, vibration analysis, speech recognition and synthesis, protein folding etc.

The order of complexity of the DFT was long thought to be $O(N^2)$, as the DFT requires the evaluation of a special matrix-vector product. In 1965 Cooley and Tukey [11] published an algorithm, the *fast Fourier transform* (FFT), which reduced the DFT's computational complexity to $O(N \log N)$. Since then numerous studies have been published on how to implement the FFT on advanced computer systems efficiently.

The performance of numerical algorithms for single processors is usually characterized by the number of processor cycles per time unit, measured during its execution. The performance of parallel programs is influenced by an additional factor, i.e., the time needed for data communication. The major difficulty in developing efficient code for parallel systems is the machine dependent ratio of processor and network performance.

This work introduces Spiral/dmp, a formal framework for automatically generating performance optimized implementations of the DFT for distributed memory computers. Spiral/dmp is an extension of the program generation and optimization system Spiral. DFT algorithms are represented as mathematical formulas in Spiral's internal language Spl. Using a tagging mechanism and formula rewriting, Spiral has been extended to automatically generate parallelized formulas.

Spiral/dmp has been used to generate rescaling DFT algorithms, which redistribute the data in intermediate steps to the optimal number of processors to reduce communication overhead. It is a novel feature of the methods implemented in Spiral/dmp that redistribution steps are merged with communication steps to avoid additional communication overhead. Among the multitude of possible algorithm variants, Spiral's search mechanism determines the fastest one for a given platform, hence effectively generates hardware adapted code without human intervention.

Experiments with DFT Mpi programs generated by Spiral/dmp demonstrate that the new methods enable performance gains of up to 30%.

Acknowledgments

First of all, I would like to thank my advisor Christoph Ueberhuber for his guidance, his comments on drafts of this work, and for giving me the chance to gain experience with high-performance computing equipment that is usually not accessible to students.

Many thanks also go to Juergen Lorenz, Stefan Kral, and Franz Franchetti. Most of my knowledge about supercomputing probably origins in conversiations we had at the Institute for Analysis and Scientific Computing.

I am most grateful to my friend Melanie Schuster for her encouragement in bad times, for sharing the good times with me and for her love throughout the years.

The greatest thanks go to my parents. Their encouragement, advice, and support throughout the years have made it possible for me to get this far.

Andreas Bonelli

Contents

Chapter 1

Introduction

Throughout history merchants, engineers, and scientists have been unsatisfied with the possibilities, performance, and error rate of mental arithmetic. Therefore, devices to support calculations have been developed since about 5,000 years.

The first known arithmetic computation machine was the Chinese *suanpan*. It consisted of a flat stone covered with sand. Lines were drawn into the sand and pebbles used as representation of decimal numbers in a combination of base-2 and base-5 system. Later versions use beads on wires, attached to a wooden frame. It was adapted and improved by other cultures including the Babylonians, the Japanese, the Roman, and the Russians. Today, Japanese pupils still learn to use the *soroban* in elementary school.

Since the fourteenth century, calculation machines of this kind are referred to as *abacus*[1]. These devices allow addition, subtraction, multiplication, division, square root, and cube root operations. Today, abaci are still used, especially as calculation aid for Asian shopkeepers and visually impaired.

Asian shopkeepers still use the abacus today. It is also used to teach mathematics to blind children in situations where a sighted person would use pencil and paper.

In 1623, Wilhelm Schickard developed the first mechanical calculation device, called the *calculating clock*. It was used, among others, by Johannes Kepler. Machines by Blaise Pascal and Gottfried Wilhelm von Leibniz followed. Based on Leibniz' work, Charles Xavier Thomas created the first mass-produced mechanical calculator in 1820. The *Thomas Arithmometer* could add, subtract, multiply, and divide. Mechanical calculators, like the *base-ten addiator*, the *comptometer*, the *Monroe*, the *Curta* and the *Addo-X* were used until the 1970s. Leibniz also introduced the binary numeral system as it is still used today.

In 1801, Joseph-Marie Jacquard developed a loom controlled by *punched cards*. The pattern woven by the loom could be changed by changing the cards. This was the first appearance of a programmable machine. In 1833, Charles Babbage created the *analytical engine* which was the first mechanical multi-purpose calculation device. It utilized a steam engine as power supply.

The U. S. Census Bureau made first use of punched cards and a sorting machine, designed by Herman Hollerith, for the census in 1890. Hollerith's company eventually became the core of IBM. Until the 1950s, IBM continued developing punched

[1] Latin, originating from *abakos*, the Greek genitive form of *abax* ("*calculating-table*").

card machines to a powerful tool in business data processing. Until the 1970s, computers have used punched cards as input for their calculations.

Beginning in the 1900s, the mechanical calculation devices were redesigned to use electric motors rather than steam or human power.

Before World War II, analog computers were the state-of-the-art. Among many experiments made at that time, the most notable one is probably the *water integrator* built in the Soviet Union in 1936. It consisted of a room full with pumps and interconnected pipes. The water level in the pipes represented the stored numbers. The early digital computers were flexible, but could not solve complex problems. Analog computers, on the other hand, were specialized to solve one complex problem, trading off flexibility.

With the development of electric circuits, relays, and vacuum tubes, the era of modern digital computing started in the 1940s. Famous prototypes of digital computers are Konrad Zuse's *Z3*, Thomas Harold Flowers' *Colossus*, Howard Aiken's *Harvard Mark I* built in cooperation with IBM, George Stibitz' *Model K* at Bell Labs, and *ENIAC* (Electronic Numerical Integrator and Computer) is the last famous representative of machines attempting to use the decimal system for calculations developed by the U.S. Army.

During World War II, a lot of German codes, among others the infamous *Enigma* system, were successfully broken with the aid of *Colossus* at the British intelligence center in Bletchley Park. The most famous mathematician working at this institution was Alan Mathison Turing, whose work was groundbreaking for modern algorithm and computing science. He devised the *Turing Machine*, a theoretical device to formalize the notion of algorithm execution.

After recognizing the limitations of *ENIAC*, John von Neumann wrote a widely-circulated report describing the *EDVAC* (Electronic Discrete Variable Automatic Computer) design, in which the programs and working data are both stored in a single, unified store. This idea, which became known as the "von Neumann architecture", serves as the basis for the development of the first really flexible general-purpose digital computers and is still used today.

The early von Neumann machines *Manchester Mark I*, developed at the University of Manchester 1948, *EDSAC* (Electronic Delay Storage Automatic Calculator), developed at the University of Cambridge, and *EDVAC* are the direct predecessors of modern computers.

The invention of the transistor in 1947 resulted in a decrease in size and power consumption compared to the vacuum tube era and allowed the mass production of computers. Printed circuits, and especially integrated circuits, allowed further miniaturization and cheaper production. Thus, computers became a bulk product in the 1970s and 1980s. The first computer, dedicated to personal use, which was a commercial success was MITS' *Altair 8800* in 1974. Alarmed by the success of home computers, the market leader in mainframe computers and electric

typewriters, IBM, introduced the *IBM 5150 Personal Computer*, which was the progenitor for IBM-PC compatible hardware platform and today's *PC*s, in 1981.

Since the positioning of computers as a mass product, the ever increasing sales allow the continuous development of new, faster computers at an amazing pace.

In 1965, Gordon Moore, co-founder of Intel, made a prophetic statement [42]:

> *The complexity for minimum component costs has increased at a rate of roughly a factor of two per year ... Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000. I believe that such a large circuit can be built on a single wafer.*

In 1975, Moore adjusted his statement and projected a doubling only every two years. Assuming that complexity is reflected by the number of transistors, this statement still holds true today. Figure 1.1 shows the complexity of Intel processors released since the 1970s and Moore's law.

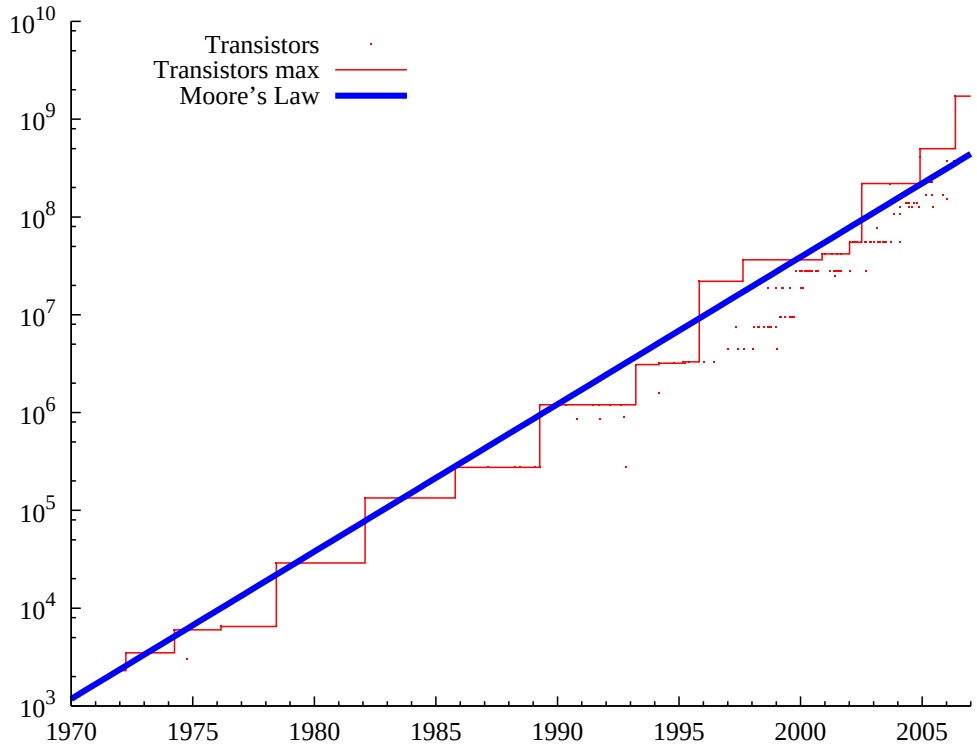


Figure 1.1: Complexity of general-purpose Intel processors. Intel's complexity data can be found in Table 1.1.

Date	Processor	Transistors
1971/11/15	4004	2,300
1972/04/01	8008	3,500
1974/04/01	8080	6,000
1976/03/01	8085	6,500
1978/06/08	8086	29,000
1982/02/01	80286	134,000
1985/10/17	80386DX	275,000
1989/04/10	80486DX	1,200,000
1993/03/22	Pentium/1	3,100,000
1994/03/07	Pentium/2	3,200,000
1995/03/01	Pentium/3	3,300,000
1995/11/01	Pentium Pro/1	22,000,000
1997/08/18	Pentium Pro/3	36,500,000
2000/11/20	Pentium 4 “Willamette”	42,000,000
2002/01/07	Pentium 4 “Northwood”	55,000,000
2002/07/08	Itanium 2 “Madison”	220,000,000
2004/12/01	Itanium 2 “Madison” 9M	500,000,000
2006/05/15	*Itanium 2 “Montecito”*	*1,720,000,000*

Table 1.1: Intel’s complexity *records* displayed in Figure 1.1.

Schmidhuber’s law [54] states that the time between radical breakthroughs in computer science decreases exponentially. He states that every new one will come about twice as fast as the last one. His timeline up to today is:

1623 Wilhelm Schickard’s first mechanical calculator.

$\sim$ 200 years later

1833 Charles Babbage’s first programmable calculation device.

$\sim$ 100 years later

1930s – 1940s Kurt Gödel’s and Alan Turing’s work on algorithms, and Konrad Zuse’s first working programmable computer.

$\sim$ 50 years later

1990 World-Wide Web (WWW) created by Tim Berners-Lee at CERN.

Following this rule, the next breakthrough is due in 2015. This is the expected date, when the fastest computers will match human brains in terms of raw computing power according to Moore’s law. Schmidhuber further speculates that, by this time, universal learning algorithms and optimal, incremental search in algorithm space will be realized. Schmidhuber’s law would culminate around 2040

in an *Omega point*, a term introduced by Pierre Teilhard de Chardin [12, 58]. This *Omega point* can also be associated with John von Neumann's *technological singularity*, a term in future studies, which represents a point past which models of the future cease to give reliable answers. This is possibly the case after the eventual development of *strong artificial intelligence* (strong AI), i. e., a form of AI which can truly reason and solve problems.

However, all these prediction models may be futile, if the current technology progress is, in fact, the early part of a logistic growth rather than an exponential one. The future will tell.

1.1 High Performance Computing

Since the first programmable computers it is an issue to optimize the efficiency of programs to make best use of the hardware provided. As this section will point out, this applies today more than ever before.

The performance of computing hardware is exponentially increasing for decades. The measure for a computer's performance in scientific computing are floating-point operations per second (flop/s). Intel's processors are capable of performing floating-point operations since the 80486DX in 1989. Since the 8086 in the late 1970s Intel was offering *coprocessors* which provided floating-point functionality. If the hardware did not support floating-point arithmetic at all, the software had had to implement these operations as several fix-point operations, which had a dramatic impact on performance.

Up to the 1990s speed-ups were mainly achieved by raitaskssing the processors' clock speeds. If a processor's clock speed is increased by a certain factor, every program will automatically speed up by the same factor *for free* because the issued instructions remain the same, but simply the execution speed is risen. However, by this time the processor developers started to address other performance limiting issues by implementing special features.

Memory speed does not increase at the same pace as processor speed, thus, in the late 1980s, the gap between memory and processor speed started to emerge. It became increasingly harder to provide the processing unit with enough data to keep it busy. This was the reason why *caches* were introduced in the late 1980s. The processor does not access data from main memory directly, but from cache. Before the processor can perform an operation, the data elements, along with the operation itself, have to be loaded into the cache. This is transparent to the program, however the order of the statements in the program have a great effect on the efficiency of the caching mechanism. The first caches were separate chips on the mainboard. Later, they were integrated onto the processor core to allow faster access. Nowadays, there are up to three levels of cache totaling up to 40MB capacity.

The 80486DX was only able to execute one floating-point operation every eighth cycle. The reason of this is that one floating-point operation consists of multiple steps requiring one cycle each. In 1993, the Pentium processor introduced *instruction pipelining*. This feature made it possible to issue an operation every cycle by overlapping the different steps of adjacent operations. However, some requirements had to be met to achieve this speedup. Foremost, consecutive operations may not operate on the same data [37]. The Pentium II introduced *out of order execution*, which allowed the processor to automatically reorder a sequence of statements to meet this goal.

The Pentium III "Katmai" introduced Single Instruction Multiple Data (SIMD) operations called Streaming SIMD Extensions (SSE). SSE allowed the execution of four floating-point operations per cycle, two additions and two multiplications. However, in practice it is almost impossible to achieve this performance as it would require the program to (*i*) contain exactly the same amount of additions and multiplications and (*ii*) issue vector additions and multiplications alternatingly to the processor. Thus, two programs which implement the same algorithm, and therefore, execute the same number of floating-point additions and multiplications, can differ in runtime by a remarkable factor, which depends only on the execution order of the statements.

By the year 2000, the curve of clock speeds in Figure 1.2 started flattening. The reason for this is that the manufacturing technology hits physical obstacles. Despite the constant miniaturization process and decreasing processor core voltages, the processors require a lot of energy. This energy is reemitted as heat. The Pentium 4 "Prescott" processors generate more than 100 Watt waste heat on a surface of a size of a thumbnail. This is comparable to, or above, the output of a heating coil.

As the customers' demand for faster processors cannot be satisfied by raising the clock frequency anymore, other measures have to be taken to continue the increase in computing power. One of the remedies was the development of multicore processors. Intel's first mainstream dualcore processor was the Pentium 4D "Smithfield" which debuted in May 2005.

A dualcore processor represents two physically separated processor cores on one chip. Even though its clock rate does not match high-end single-core processors, its performance is superior to them, because it is capable of executing the double amount of operations per cycle. Other examples for multicore processors are AMD's *Athlon 64 X2* dualcore series and IBM's *Cell* processor, which utilizes even *nine* processor cores on one chip. It is forseeable that this trend will continue in the near future and the number of cores on a chip will be steadily increased.

As with every performance increasing measure, except increasing the clock rate, the additional performance does not come *for free*. Usually there are data dependencies between the operations of one program, so the operating system cannot

simply *spread* them over two processor cores. A program has to be separated to two parts, or *threads*, to make use of the two cores. Furthermore, the Pentium 4D, has two separate caches for the two cores, which has the effect that, if one processor changes data, the other processor does not take immediate notice of that. The data has to be rewritten to main memory and fetched by the other processor, to avoid inconsistencies.

Figure 1.2 shows the progression of the peak performance and clock rates of Intel's mainstream processors since 1970. Up to 2003, the clock rate, which increases at exponential speed, is the main factor of performance gains. In fact many sources applied Moore's law to the clock-speed increase in the 1990s. However, this progress stopped in early 2003. Extrapolating Moore's law beyond this point would imply that there should be 10GHz processors available today, which is, by far, out of reach. Instead, the latest increases in performance have been achieved by introducing multicore processors in 2005.

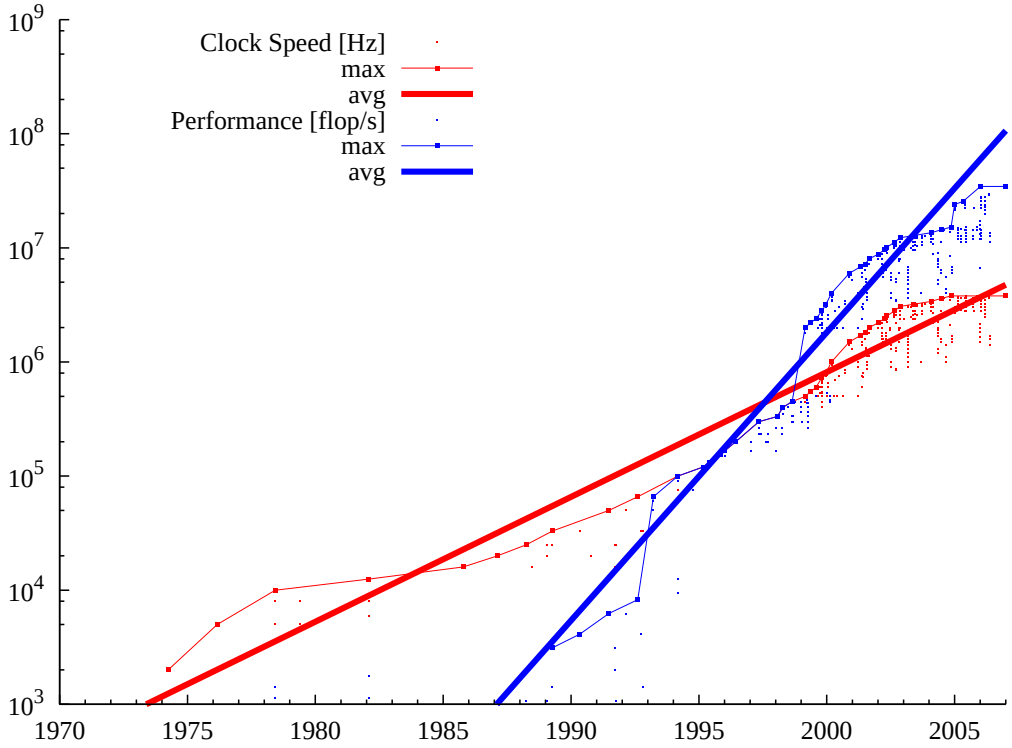


Figure 1.2: Clock speed and floating-point performance of Intel processors. The data underlying this diagram can be found in Table 1.2.

The annoying effect of this evolution is that the factor, which provided applications with *free* performance increases throughout the last decades, has disappeared, as Herb Sutter pointed out [56].

Multicore processors represent one more step in the ladder of specific performance

Date	**Processor**	**Integration [nm]**	**Clock [MHz]**	**flops/cycle**
1989/04/10	80486DX/25	1000	25	1/8
1990/05/07	80486DX	1000	33	1/8
1991/06/24	80486DX	800	50	1/8
1992/08/10	80486DX2	800	66	1/8
1993/03/22	Pentium	800	66	1
1994/03/07	Pentium Rev. *I*	600	100	1
1995/03/01	Pentium Rev. *II*	350	120	1
1995/06/01	Pentium Rev. *II*	350	133	1
1995/11/01	Pentium Pro	600	155	1
1996/01/04	Pentium Rev. *II*	350	166	1
1996/06/10	Pentium Rev. *II*	350	200	1
1997/05/07	Pentium II "Klamath"	350	300	1
1998/01/26	Pentium II "Deschutes"	250	333	1
1998/04/15	Pentium II "Deschutes"	250	400	1
1998/08/24	Pentium II "Deschutes"	250	450	1
1999/02/26	Pentium III "Katmai"	250	500	4
1999/05/16	Pentium III "Katmai"	250	550	4
1999/08/02	Pentium III "Katmai"	250	600	4
1999/10/25	Pentium III "Coppermine"	180	700	4
1999/12/20	Pentium III "Coppermine"	180	800	4
2000/03/08	Pentium III "Coppermine" Rev. *I*	180	1000	4
2000/11/20	Pentium 4 "Willamette"	180	1500	4
2001/04/23	Pentium 4 "Willamette"	180	1700	4
2001/07/02	Pentium 4 "Willamette"	180	1800	4
2001/08/26	Pentium 4 "Willamette"	180	2000	4
2002/01/07	Pentium 4 "Northwood"	130	2200	4
2002/04/02	Pentium 4 "Northwood"	130	2400	4
2002/05/06	Pentium 4 "Northwood" B	130	2530	4
2002/08/25	Pentium 4 "Northwood" B	130	2800	4
2002/11/14	Pentium 4 "Northwood" B	130	3066	4
2003/05/21	Pentium 4 "Northwood" C	130	3200	4
2004/02/01	Pentium 4 "Northwood" C	130	3400	4
2004/06/21	Pentium 4 "Prescott" 560	90	3600	4
2004/11/12	Pentium 4 "Prescott" 570J	90	3800	4
2005/01/01	Pentium 4 Xeon MP 7040 "Paxville"	90	3000	8
2005/05/01	Pentium 4 EE "Smithfield"	90	3200	8
2006/01/05	*Core Duo Yonah T2600*	*65*	*2160*	*16*

Table 1.2: Intel's performance *records* (see Figure 1.2).

improvement features, which have been introduced since the 1990s; one more factor, which has to be taken into account by the application developer. For instance, if a conventional scalar program is run on a dualcore processor, it will reach only 50 % of the processor's peak performance in the optimal case, because it cannot make use of the second core.

In today's multitasking operating systems, this does not seem to be a serious obstacle, as in usual desktop operation multiple applications are running at the same time and can, thus, be worked on in parallel. However, with an increasing number of cores being part of one processor that situation may change in the future. Especially in the case of scientific computing already two cores are a serious issue, as a single numerical program usually utilizes the processor to an extent of nearly 100 % of CPU time.

1.2 Parallel Computing

Despite the steady increase in computing power, there are always scientific problems which exceed the storage space or performance one computer system can provide. To overcome this limitation, multiple computers are connected with some kind of network to work on one large problem *parallelly*.

Parallel computing is the execution of the same, or different, tasks on multiple devices in parallel for the solution of one superior problem. Depending on the type of parallelization the problem needs to be divided into more or less autonomous subproblems to perform parallelization effectively. Parallelism in modern computing appears in various flavors:

Vector Instructions

Multicore Processors

Multiprocessor Boards (Shared Memory Parallelism)

Distributed Memory Parallel Computing

Grid Computing

This list ranges from tightly coupled SIMD instructions that perform two or four operations simultaneously to loosely connected grids consisting of separate computers interconnected by some kind of network. These types of parallelization are not exclusive. One program may even utilize all of those, which are present on a given computer system.

The first three instances of parallel computing have already been covered in the last section. These are on-chip or on-board features to improve the performance of one computer without raising its clock rate.

Usually parallelism comes with various kinds of requirements. Vectorization requires the operands to be aligned and subsequently ordered in memory. A program running on multicore processors has to address the problem of incoherent caches, i. e., if two processors load the same data element and one processor changes it, the other processor is not given notice of that change. In case of multiprocessor boards every processor is assigned a certain part of the whole system's memory, thus, memory coherence is an issue. Distributed computer systems and computational grids consist of multiple computers interconnected by a network. Synchronizing data over a network is very costly, so it is an issue for the performance of the overall program to minimize the time needed for communication.

All forms of parallelism inevitably generate some kind of overhead compared to sequential calculation, because there is a need to synchronize the devices in some way. Therefore, doubling the number of parallelized devices hardly ever results in a doubling of the performance. In rare cases experiments can show even super linear speedup. This phenomenon is usually due to cache-effects—the whole problem did not fit into cache, but the reduced one does. However, such cases are bound to problem sizes of a certain, usually very small, range, and can be disregarded in general.

This section deals with issues arising in distributed memory parallelization. Grid computing is similar to DMP computing, but DMP computing requires homogeneous nodes, whereas grids, in general, consist of heterogeneous sets of computers.

If a certain problem cannot be solved on a single computer, there might be two reasons: (*i*) The problem does not fit into the computer's memory or (*ii*) solving the problem takes too long.

The fastest supercomputers manufactured by well known players in industry make use of state-of-the-art computing hardware and use high performance interconnection networks specially developed for parallel computers. As communication is often a bottleneck in parallel computing, a considerable part of the total cost of such a computer system is caused by an expensive network infrastructure.

The best parallel computer systems have a flagship function for the companies which develop them. The publicity and image gain of having "the fastest computer on the world" is invaluable for companies like IBM, Cray, or NEC. Twice a year a list of the 500 best performing computer systems[2] worldwide is published by Jack Dongarra et al. This list orders the computer systems by their performance for the Linpack benchmark [13].

On the other side there are *Computer clusters* which represent a relatively easy and cheap method to obtain more computing power. Such a cluster is assembled by off-the-shelf desktop computers. Usually the nodes are interconnected with cheap 10/100/1000 GBit Ethernet network adapters. The downside of the

[2] `http://www.top500.org/`

cheap infrastructure is the limited performance. Table 1.3 shows the performance parameters of Ethernet network adapters compared to specialized high performance networks. Nevertheless, the cost-benefit ratio, and the large community, and thus independence from a certain hardware manufacturer or vendor, makes them a popular alternative to more expensive solutions. The current top 500 list[3] contains 364 high-performance clusters (73 %), with the best one ranked 5th. Computer clusters have been initially developed by Donald Becker and Thomas Sterling at NASA in 1994.

Using commodity processors in computer clusters does not necessarily have a negative impact on performance. Nowadays server processors are very similar to desktop processors. They mainly differ in cache size and power consumption. However, the various types of network interconnection hardware differ significantly in both latency and bandwith (Table 1.3). The impact of these numbers on the overall performance of a scientific program highly depends on the design and the structure of the program.

Network	**Latency** $[\mathbf{10^{-6}s}]$	**Bandwidth** $[\mathbf{10^{9}bit/s}]$
Ethernet	175	0.01
Fast Ethernet	175	0.1
Gigabit Ethernet	175	1
Myrinet	6	3.9
SCI (Scali)	5 – 6	5.3
IBM Colony SP Switch	2	2.4
Quadrics	2	7.2
Infiniband	2	10 – 30

Table 1.3: Communication Network Parameters.

There are applications, which are well suited for parallel execution, and thus, are relatively insensitive with respect to network performance. For instance, parameter studies, where all processes operate on the same data set, the *master process* tells the *children* which parameters to test, and receives a short answer after the completion of the task, are perfectly suited for parallel execution. The network performance will not have a huge impact on such programs, as long as they are well designed and implemented.

A contrary example is the fast Fourier transform (FFT), which this work is dealing with. As each output data point is dependent on every input data point, the computation of the FFT requires a reasonable amount of network communication, and thus heavily depends on the network's performance. Experiments

[3] 27th top 500 list, June 2006

have shown that an FFT can spend up to, and above, 70 % of its execution time communicating data if it is run on a slow network [1].

As mentioned before, computer clusters and supercomputers also differ in energy consumption. Current supercomputers often use processors which rely on embedded, low voltage, computing. The best example are IBM's *Blue Gene* systems. BlueGene/L is the number one of the current top 500 list[4]. It consists of 131,072 PowerPC 440 embedded processors running at 700 MHz. Compared with current desktop computers this is a rather low clock rate, but the high degree of parallelization and the specialized architecture result in an unmatched performance in the Linpack benchmark. Such a degree of parallelization would not be possible with standard desktop processors. Running the same number of desktop processors would result in an energy consumption and heat output of 10 Megawatt.

Regardless of the type of processors used, such a degree of parallelization raises another problem. Unlike processors in a desktop computer, which are powered approximately 2 to 10 hours a day, the processors in a supercomputer are in constant use. This decreases the average life span of a processor integrated into a server. Assuming an expected life span of about three years, on the long term 120 processors would break every day or every twelve minutes one of the processors breaks. This only regards processors but, of course, other pieces of hardware can break too. The conclusion is that most probably *something* is broken at any given time. Furthermore it has to be expected that if a user runs a large, long running job, it is probable that some hardware utilized by this job will break *during* the execution time of this job. Practical observations with IBM's BlueGene/L system show that the mean time between failure is much better than this pessimistic scenario. Still, fault-tolerance in parallel computation has become an important issue recently [17].

As noted above there are various providers for interconnection networks. Each company develops its own libraries to access the network adapter's functionality. Using such *vendor libraries* for communication results in extreme incompatibility. Therefore, the Message Passing Interface (MPI) communication protocol has been created in 1994. Today MPI is the de-facto standard for network communication among the processes of parallel programs. MPI provides functions on different abstraction layers, reaching from explicit send-receive statements between two processes to high-level *collective* communication calls, e. g., representing matrix transposition. Vendors usually provide their own MPI libraries, specially adapted to their hardware, which implement the standardized interface. A program using MPI can be adapted to a certain computer system by simply linking it with the MPI implementation of the network infrastructure vendor. Moreover, there are open source MPI libraries, which are compatible to a wide range of network types

[4] 27th top 500 list, June 2006

like MPICH[5] or LAM-MPI[6].

Parallel computer systems require software that organizes job execution and hardware allocation, i. e., a *batch system*. When somebody wants to use the parallel computer he has to create a *job*. This job's definition has to include information like the number of processors and/or nodes required, the maximum execution time, the location and parameters of the executable, as well as the location of the output files. If there are not enough free resources to meet the job's requirements the job is *queued*. As soon as it is possible to run the job the batch system assigns certain nodes and starts the job. If the maximum execution time is exceeded, the job is deleted. Some batch systems also allow the user to work in *interactive parallel shells*. When an interactive parallel shell is created and the requested resources are available, the user can *manually* start jobs on the assigned nodes from the command line.

Resources are exclusively assigned to one job. Thus, a job can be sure that there is no other process matching for its CPU time. However, depending on the network topology, there may still be bottlenecks in the communication network deteriorating the job's performance.

1.2.1 Network Topologies

There are basically two categories of network topologies in high-performance computing: (*i*) static direct interconnection networks and (*ii*) dynamically switched networks. In direct interconnection networks one connection always connects two computers pairwise. Illustrations of the various network types are presented in Figure 1.3 for direct interconnection networks. Graph theoretical properties of these topologies are listed in Table 1.4.

	Nodes	**Connections**	**Max. Route**	**Avg. Route**
Ring	p	p	$p-1$	$p/2$
Star	p	$p-1$	2	$2(p-1)/p$
nD Mesh	$p=k^n$	$nk(k^{n-1}-1)$	$n(k-1)$	$O(n)$
nD Torus	$p=k^n$	nk^n	$nk/2$	$nk/4$
nD Hypercube	$p=2^n$	$n2^{n-1}$	n	$n/2$
Fully Connected	p	$p(p-1)/2$	1	1

Table 1.4: Graph theoretical characteristics of direct interconnection networks.

Ring. In a ring based network all computers are connected in a circle, thus, every node is connected to two other nodes. Usually data packets can only

[5] `http://www-unix.mcs.anl.gov/mpi/mpich/`
[6] `http://www.lam-mpi.org/`

be sent into one direction, however, it is possible to enhance this topology to a *double ring* which allows sending into both directions. This network type is very error prone because one broken node stops any communication through it.

Star. In a star topology a central node is connected to all other nodes. The maximum route length is very short, however, the whole network's bandwith is limited to the throughput capacity of the central node. A broken central nodes renders the whole network inoperable.

Mesh. A Mesh network requires more connections than the above mentioned. Every connection is bidirectional. Thus, this network topology is less error prone than the above mentioned. Multiple pairwise communication steps can be performed simultaneously without interfering each other.

Torus. The torus topology represents a multidimensional ring topology. It is very popular in high performance computing. With slightly more connections than a mesh it reduces the maximum and average route lengths by a factor of two, and allows even more simultaneous communication steps.

Hypercube. Especially in higher dimensions, a hypercube topology requires substantial amounts of network hardware. The number of nodes of an nD hypercube is always 2^n. In practice only computers with a small number of nodes, or small portions of larger machines, are connected as a hypercube.

Fully Connected. Fully connecting a parallel computer means that each node is connected with every other node. This topology is rarely found in practice.

Contrary to direct interconnection networks, in switch based networks, any compute node is connected to a *switch*. Such switches have multiple ports and when a package arrives, the switch parses it for its destination and sends it to the correct output port, which can be connected to another switch or the target computer. Figure 1.4 shows two representations of switch based networks.

Tree. Classic switch based networks are arranged according to a *simple tree*. A certain number of compute nodes are connected to one switch. This group is called *frame*. Multiple frame's switches are connected to each other by a *master switch*. Typically frames and master switches are multiply connected to provide a higher bandwith. Parameters like the height of the tree, the number of layers, and frame sizes, can be adjusted to the desired performance.

Fat Tree. Unlike in a simple tree, in this case every frame switch has an equal number of uplink and downlink ports. Thus, fat trees require more master switches than simple trees. As the bandwith limiting factor is the line's

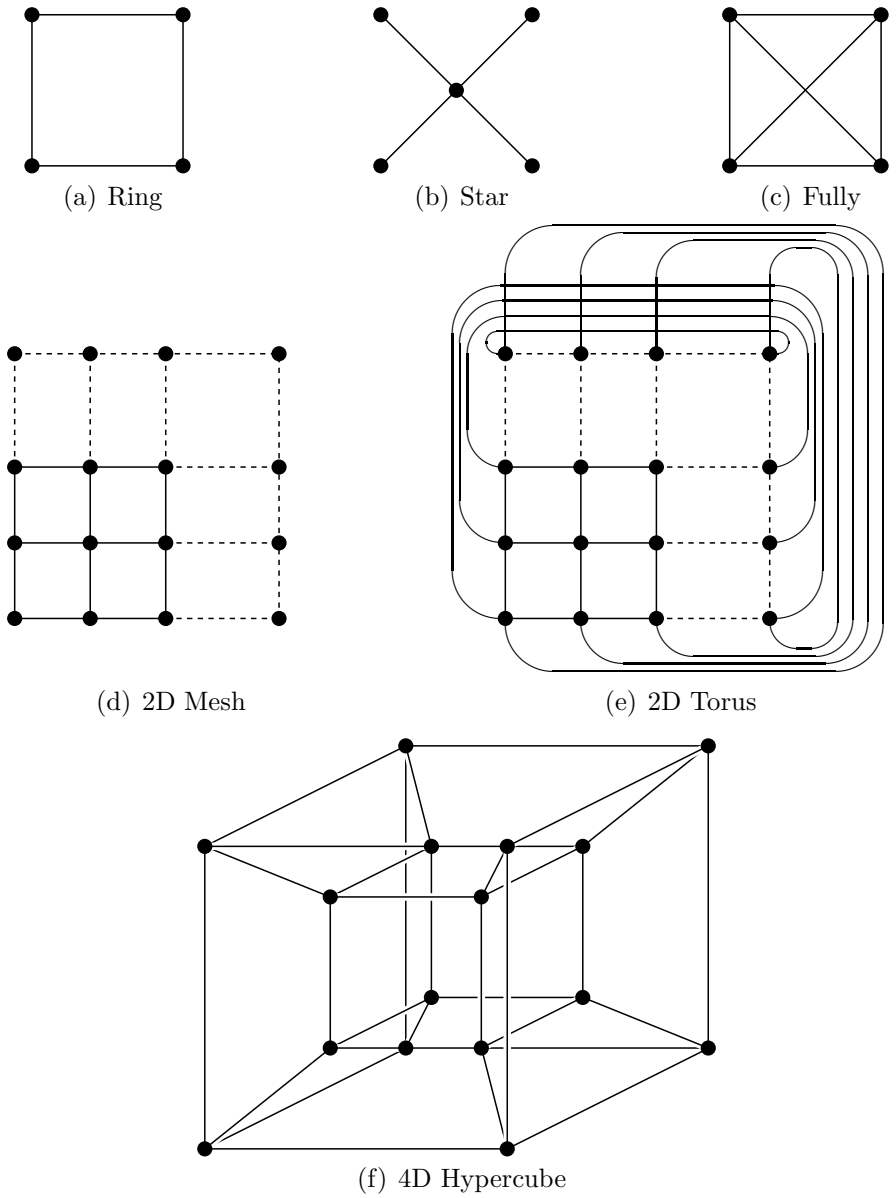


Figure 1.3: Direct Interconnection Network Topologies.

bandwith and not the switches' throughput, congestion is theoretically not possible in a fat tree. This kind of network can maintain service at limited bandwith even if all but one master switches are offline.

Supercomputers often use more than one network. Often a maintenance network is installed in addition to the high-performance network. Therefore, administrative tasks do not interfer with the parallel jobs. Furthermore, it is possible to install multiple high-performance networks. The reason for this is, that point-to-point and collective operations prefer different topologies.

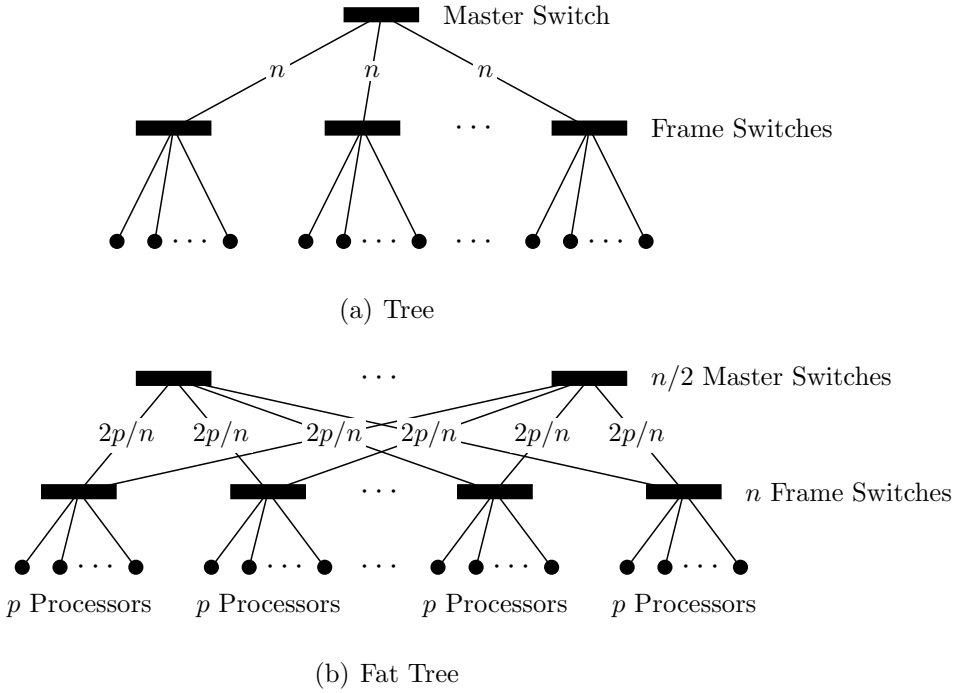


Figure 1.4: Switch Based Network Topologies.

For instance, IBM's Blue Gene/L supercomputer, which consists of 131,072 processors on 65,536 nodes, has five different networks. The main network is a $64 \times 32 \times 32$ 3D Torus which ensures fast point-to-point message passing between the compute nodes. Besides that, an adaptive tree network is installed for certain collective operations. Three more auxiliary networks are dedicated to I/O and maintenance operations.

Supercomputers, in general, do not utilize wireless interconnects, as their bandwidth and latency properties are not suitable for high-performance computing.

1.3 Problems in High Performance Computing

The utilized algorithm is not the only performance impacting factor nowadays. The way it is implemented, and optimized to the certain processor architecture's characteristics, is very important too. It is even possible that algorithms, which are mathematically suboptimal, yield the best performance because e. g., the number of additions and multiplications is better balanced for utilization of SSE extensions. Furthermore, due to the memory-processor bottleneck, the order of the statements has a reasonable effect to the performance too.

Figure 1.5 shows a histogram of all 15,778 algorithms Spiral can generate for a

$\mathrm{DCT}_{16}^{(\mathrm{IV})}$. All of these algorithms are fast algorithms and have approximately the same number of arithmetic operations. The number of additions varies by 8 %, the number of multiplications by 16 %. However, the runtime differs by a factor of two because compilers can handle certain sequences of operations better than others.

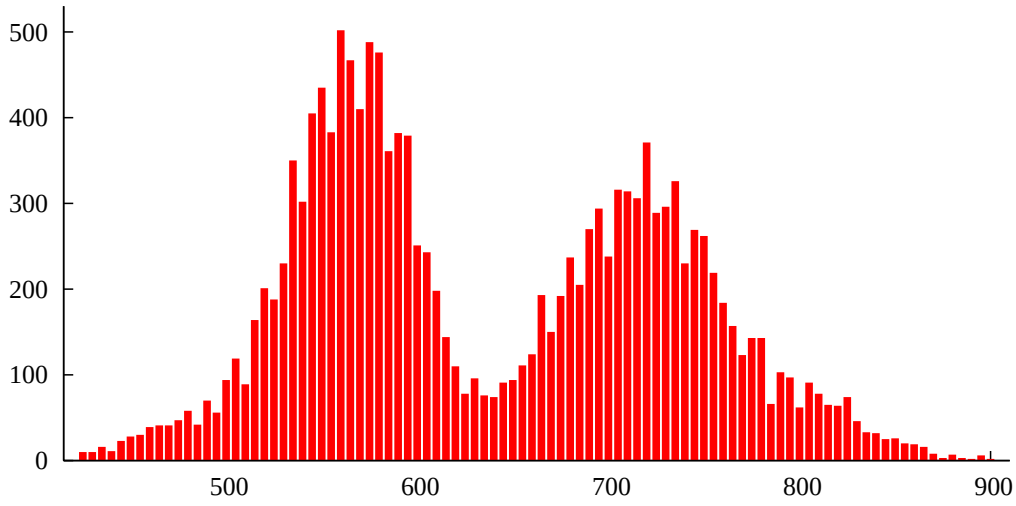


Figure 1.5: Histogram of the runtimes (in nanoseconds) of all 15,778 algorithms for a $\mathrm{DCT}_{16}^{(\mathrm{IV})}$, implemented in straight-line code on a Pentium 4, 1.8 GHz, running Linux. [52]

Performance increasing features built into processors nowadays, raise the computer's theoretical peak performance, but not necessarily the observed (measured) performance of a certain program. Without adaptations to the source code a general purpose compiler can hardly make use of them. The reason for this phenomenon is that most tuning potential is not obvious, but based on domain specific characteristics of the implemented algorithm. Therefore, creating software which utilizes the provided architecture's features and achieves a satisfactory performance, requires highly skilled developers. Thus, implementing high performance numerical software does not only require good knowledge about programming and the domain's algorithms. It also requires the developer to know about the features of the target architecture and how to use them. Even with this knowledge, optimizing a programm is still a tough task and involves a lot of trial and error.

Code running very fast on one particular processor may perform sub-optimal on another one and vice versa. Different vendor's processors may even require the use of different instruction sets. If a developer has created a program that performs well on one particular processor, he has achieved that for this processor architecture only. If the program is to be ported to another architecture the whole creation process has to be redone, or at least some retuning effort is needed.

Table 1.2 shows that the exponentially increasing performance in the last decades had the side effect that the processor architectures' life cycles constantly became shorter.

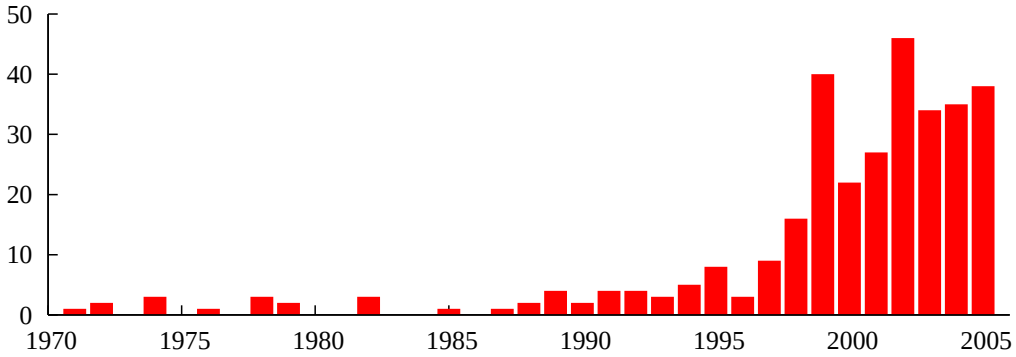


Figure 1.6: Approximate number of different Intel processors released per year.

Taking into account all the facts mentioned above, the economic efficiency of manually tuning programs to achieve high performance is questionable, at least. Due to the rapid development in processor technology, the vendor companies tend to provide high performance libraries (like Intel's MKL) for certain problems, which promise a promotional effect, while ignoring less important ones.

1.4 Code Generation for Numerical Software

Current general purpose compilers are not able to generate code competitive with hand-tuned code in efficiency. To overcome this shortcoming, automatic program generation, optimization, and platform adaptation has been introduced [44]. Important examples include FFTW [28, 29], ATLAS [65], and SPIRAL [51].

The effort to implement fast programs is steadily increasing together with the introduction of processor features, which have to be utilized to achieve maximum performance. At the same time processor architectures' life-cycles are becoming shorter. This leads to an escalating requirement of human resources to produce platform optimized software in time. Thus, on the long term, manual implementation and tuning of high performance software is economically unbearable. The only way to counter this development is to generate these programs with reusable code generators, which can be continually adapted to cover programming techniques for new platforms.

In general, a code generator shall produce code which yields optimal performance for a certain problem in it's domain. Often code generators use divide and conquer dynamic programming mechanisms. The original problem is broken down to smaller ones, which are easier to optimize. The generator knows several alternative

methods to implement these building blocks. Because of today's computer architectures' complexity, the runtime of a source code segment on a certain computer is not exactly predictable. Thus, generators usually rely on runtime measurements to find out which alternative performs best. Up to a certain extent, model based optimization can support this process, but measuring runtimes is unavoidable to produce reliable results. Once, the best code for the certain problem's building blocks is found, the generator works back up, utilizing the solutions for the subproblems found before, and again, searching over alternative ways to implement their combination.

To be able to generate fast code, the code generator has to dispose of code pieces which actually are performant on the target architecture. For this purpose, ATLAS uses manually implemented and tuned code segments. FFTW also includes hard-coded *codelets*, but they have been automatically generated with `genFFT` [26] before. SPIRAL dynamically generates the building blocks itself upon the generation of the whole problem. All of these software products produce highly satisfactory results and are competitive with vendor-supplied highly platform optimized libraries.

Even though automatic generation would also allow more flexibility, current code generators have as restrictive interfaces to the surrounding application as static libraries. The application has to meet certain requirements to allow a straightforward utilization of the code generator. As described later, this is one of the issues adressed by the work presented in this paper.

1.4.1 Code Generation for Parallel Systems

While on single-processor machines highly satisfactory efficiency is achieved [20], the current situation on parallel computers—in particular on distributed memory machines—is far from being optimal.

The addition of network communication to the issues of scalar performance optimization adds an additional layer to the, already complex, issues of optimization. Futhermore, communication performance is very hard to predict and can fluctuate. As the measured program will not be the only one running on the parallel system, the current network traffic of other applications can result in a drastic decrease of the communication performance. Especially tree networks (see Fig. 1.4a) are very vulnerable to such congestions, as many nodes share the frame's uplink connections. The network traffic of other applications is not predictable and can change randomly. Therefore, network congestion is usually accepted as a non influenceable fact, which can cause reasonable fluctuations of the application's performance.

When a user launches a parallel program, he will usually not utilize the whole parallel computer but rather a portion of it. These nodes, assigned by the batch

system, form a *sub-topology* of the whole system's one. Furthermore, on computers with multiple processors per node, the user often cannot steer how many processors should be used per node. These issues can also cause runtime oscillations, but in theory they are known at the beginning of code execution. In practice, however, there is no standardized way to gather this information across platforms.

Due to the low demand, high performance network interconnection hardware is very expensive, so computer clusters often show the tendency to provide high computation performance while containing relatively poor networks. Some algorithms can be adapted to this fact by trading off communication and computation effort. It is even possible to decrease the *granularity* of the computation, i. e., to reduce the number of processors involved in computation, which results in a decrease of the communication volume, while the number of operations per processor is increased.

If an algorithm does not allow an implementation which guarantees the same workload to be carried out on every processor the balancing of the workload between the processes is critical to make maximum profit of the parallel system. If the workload even depends on the input data, e. g., iterative algorithms which are executed until the result meets a certain precision, dynamic load balancing may be of use. However, it must be kept in mind that the communication caused by the load balancing system itself may cause a significant communication overhead too.

State-of-the-Art in Code Generation for Parallel Systems.

The following projects have in common that they deal with generating optimized message passing parallel code for given target systems. The respective domains range from general problems in linear algebra and signal transformation to more specific computations, for example, in quantum chemistry. The parallelization methods dealt with vary from classic loop transformation to formula manipulation on a mathematical level.

A compiler framework generating MPI code for arbitrarily tiled for-loop nests by performing various loop transformations to gain *inherent* coarse-grained parallelism is presented in [32].

The publications [50, 61, 62] describe the generation of collective communication MPI code by automatically searching for the best algorithm on a given system. Another approach for empirically generating efficient all-to-all communication routines for Ethernet switched clusters is introduced in [18, 19].

ScaLapack [4] is a portable library of high performance linear algebra routines for distributed memory systems following the message passing model. Built upon Lapack, it is highly scalable on parallel architectures using arbitrary processor numbers. ScaLapack requires the user to define the processor configuration and to distribute the matrix data himself.

[2] presents a parallel code generator for a class of computational problems in quantum chemistry. The input described by tensor contractions is manipulated using algebraic transformations reducing the operation count. Data partitioning and memory usage optimization is performed for a specified number of processors on a given target system by using dynamic programming search.

[40] describes the extension of a sequential self-adapting package for the Walsh-Hadamard transform (WHT) to generate MPI code. Different WHT matrix factorizations provided in Kronecker notation exhibit different data distributions and communication patterns. Searching the space of WHT formulas leads to the best performing factorization on a given platform.

FFTW [28] is a self-adapting FFT library for one or higher dimensional real and complex data of arbitrary input size. Typically, FFTW produces code that runs faster than other publicly available FFT codes and compares well to vendor libraries. MPI support, i. e., MPI-FFTW, is available in FFTW 2.1.5 only but not in the more recent version 3.1 [29]. A comprehensive instruction to FFTW is given in Section 4.10.

1.5 SPIRAL/DMP and its Novelties

Linear digital signal processing (DSP) transforms such as the ubiquitous fast Fourier transform (FFT) are very important tools in computational science and engineering. Typically, FFTs are used as subroutines in compute intensive applications.

Requirements. Application programmers would require black-box FFT routines that (*i*) can easily be plugged into existing applications with as little additional hand coding as possible and (*ii*) return their computational results as fast as possible. Unfortunately, today's high performance FFT libraries, such as the state of the art library FFTW, are not able to meet these user requests due to performance reasons [15, 35], which will be explained in the following.

Distributed memory parallel FFTs require remote data access operations, imposing costly network communication between successive computational stages. Practical experiments [1] show that substantial portions of a parallel FFT's runtime are spent on network communication.

Performance Aspects. To achieve a satisfactory performance, FFT libraries like FFTW normally prescribe specific input and output data distributions (usually slab decompositions) while application programmers often prefer other types of data layout. The resulting necessity of transforming data having different distributions back and forth leaves the user with a demanding task. Even worse, the additional runtime needed for user-implemented data redistributions reduces the

overall performance and may outweigh the benefit of a high performance FFT library in many cases.

Additionally, FFTW tries to optimize the number of processors really used for a scaled FFT computation, thus reacting on the tradeoff between communication and computation. However, the number of processors used for a certain transform is not static, but can vary every time the plan is recreated. Thus, programs to be run on a large variety of machines require an adaptive redistribution routine, which has to be hand coded and called prior and posterior to calling the FFT routine as it is not possible to know in advance how many processors FFTW will require to calculate on.

The data reorganization steps prior and posterior to any parallel FFTW call constitute two additional communication steps, which cannot be taken into account in FFTW's runtime optimization. If these steps are not implemented optimally a significant deterioration may occur. Thus, even if the FFT's runtime is optimized, the overall runtime on a parallel system may be far from being optimal.

These issues are not discussed and evaluated in the benchmarks of any FFT routine, despite the fact that they may have a huge impact on the application's overall performance.

As compatibility and performance seem to be competing issues, application developers often shy from using high performance FFT libraries. They rather decide to create their own tailor-made FFT routines to exactly fit the application's general framework [16].

SPIRAL. The signal processing library SPIRAL [51], follows a comprehensive approach to code generation. Contrary to other software systems aiming at that purpose, SPIRAL includes multiple layers of rewriting and optimization ranging from formula rewriting down to loop unrolling and optimization of integer expressions used as array indices.

SPIRAL's rewriting system represents a powerful instrument for formula manipulation on a mathematical level. By utilizing this tool it is possible to expose parallelism in signal transform formulas to prepare them for parallelized implementation and execution. Furthermore, SPIRAL contains various types of optimization, like vectorization [21] or fused multiply-add utilization [64], which are helpful in the automatic generation of efficient parallel code. An overview of SPIRAL's architecture and features will be given in Section 5.1.

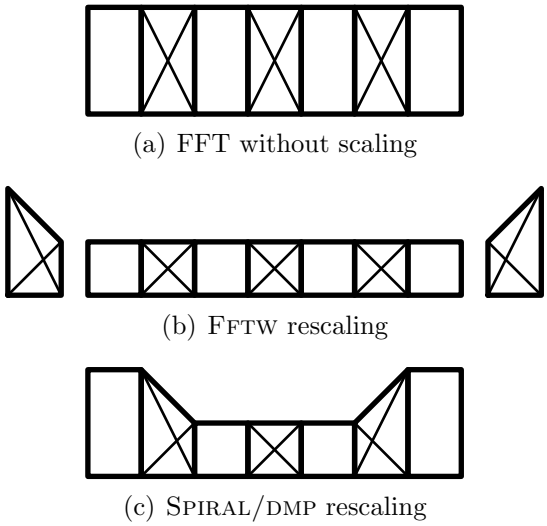
SPIRAL/DMP. The newly developed SPIRAL/DMP, an exension to SPIRAL, provides a framework for automatically generating distributed memory parallel code that calculates single and multi-dimensional signal transforms.

The development goal of SPIRAL/DMP was to provide an automatic generator for *high-performance* parallel code, while maintaining *flexibility* and *usability* on both the generation level and the application level. Using SPIRAL/DMP in the context

of larger applications does not require any additional hand coding to be done by the application developer.

Input to SPIRAL/DMP is a descriptive definition of the desired signal transform. In case the application, in which the transform shall be embedded, uses a data layout other than slab distribution, the user may pass a mathematical definition of this individual distribution to SPIRAL/DMP, where a transform customized to this particular application is generated.

Like FFTW, SPIRAL/DMP automatically optimizes the number of processors actually used for the calculation in relation to communication speed, but any *downscaling* to be done in this context is totally transparent to the surrounding application. SPIRAL/DMP does not require the application developer to manually implement code that carries out any kind of data redistribution. Moreover, SPIRAL/DMP itself does not carry out any redistribution *outside* the scaled FFT, as illustrated in Fig. 1.7. Crossed blocks represent communication steps, uncrossed ones computation. Ascending and descending trapezoidal blocks symbolize the rescaling steps. The small blocks are rescaled blocks. The execution order is from left to right.



(a) FFT without scaling

(b) FFTW rescaling

(c) SPIRAL/DMP rescaling

Figure 1.7: Plain and rescaled execution of a transform which requires three communication steps, e. g., one-dimensional FFT's. Crossed blocks represent communication steps, uncrossed ones computation. Ascending and descending trapezoidal blocks symbolize the rescaling steps. The small blocks are rescaled blocks. The execution order is from left to right.

Data rescaling is done by SPIRAL/DMP in the first and the last communication step, needed for the transform's implementation anyway. Compared to FFTW's downscaling approach this kind of rescaling saves two communication steps. Moreover, the computational parts outside the outermost communication steps profit from the increased granularity. Larger, composite, signal transforms with multiple

communication steps, like FFT based convolutions, make particular profit from this kind of optimization.

The output of SPIRAL/DMP's generation process is *one* function that calculates the requested signal transform. This function is automatically customized to the user's data layout and optimized to the given hardware and its communication features. Just calling this function is all an application developer has to do for integrating this transform into his application.

As all the code needed to carry out any specific transform is generated by SPIRAL/DMP, it is possible to measure and optimize the runtime of parallel FFTs as a whole. Thus, any optimization is only applied, if it really yields profit for the surrounding application.

1.6 Synopsis

Chapter 2 summarizes the mathematical framework required to express the new results presented in this thesis. The main focus is on expressing numerical algorithms by way of matrix factorizations. Furthermore, specific properties of matrix operations, needed in automatic parallelization, are introduced.

Chapter 3 describes properties of certain classes of permutations, which are important for the representation and optimization of network communication. Especially stride permutations are important in this context.

Chapter 4 introduces the fast Fourier transform (FFT) algorithms as products of sparse matrices and derives possibilities to compute them in parallel systems.

Chapter 5 gives and overview over SPIRAL's architecture and features and introduces the newly developed SPIRAL/DMP. A comprehensive insight is provided into the objects and rules necessary to break down a DSP transform to parallelizable factors and to generate optimized parallel code within SPIRAL/DMP.

Chapter 6 shows the results of performance measurements of SPIRAL/DMP.

Chapter 7 outlines ongoing work and potential for future improvements as well as enhancements of SPIRAL/DMP.

Chapter 2

The Kronecker Product

In this chapter, Kronecker products and their algebraic properties are introduced from a point of view well suited to algorithmic and programming needs. It will be shown that mathematical formulas, involving Kronecker product operations, are easily translated into various programming constructs and how they can be implemented on vector and parallel machines.

The Kronecker product formalism has a long and well established history in mathematics and physics but until recently, it has gone virtually unnoticed by computer scientists.

This is changing because of the strong connection between certain Kronecker product constructs and advanced computer architectures. Properties of Kronecker algebra provide powerful mechanisms to express (signal processing) algorithms as compositions of factors representing equivalent stages of computation in a hardware transformation environment representing hardware functional primitives (Johnson et al. [39]). By this identification, Kronecker products have emerged as a powerful tool for designing algorithms for parallel computer systems.

By algebraically manipulating Kronecker product formulas, different programs that achieve the same computation, but have different data flow and performance characteristics, can be obtained.

Algorithms may be expressed as distinct organizations of computational kernels, which therefore stand for suitable, high-level implementations of various hardware structures. On the basis of this high-level approach, it yields a simplification in processing to select and reorganize this basic building blocks for changing hardware architectural demands. Different algorithms correspond to different sparse matrix factorizations.

In this work the Kronecker product formalism, also known as *direct* or *tensor product*, offers a unifying basis for the description of FFT algorithms. The mathematical description of parallelism and data distribution makes it possible to conceptualize parallel programs, manipulate them using linear algebra identities and thus better map them onto our target parallel architectures.

Van Loan [63] uses this technique for a state of the art presentation of FFT algorithms in his remarkable book "Computational Frameworks for the Fast Fourier Transform". In the twenty-five years between the publications of Pease [48] and Van Loan [63], only a few authors used this powerful technique: Temperton [57]

and Johnson et al. [39] for FFT implementations on classic vector computers and Norton and Silberger [45] on parallel computers with MIMD architecture. Recently, Gupta [34], Pitsianis [49], and Püschel et al. [53] used the Kronecker product formalism to synthesize FFT programs.

As a consequence, the Kronecker product approach to FFT algorithm design antiquates more conventional techniques like signal flow graphs, where no well defined methodology for modifying FFT algorithms is available. They rely on the spatial symmetry of a graph representation of FFT algorithms, whereas the Kronecker product exploits matrix algebra.

2.1 Notation

The notational conventions introduced in this section are used throughout the following text.

2.1.1 Vector and Matrix Notation

In this text, vectors appear as lowercase letters $x, y, z, \ldots$ while matrices appear as capital letters $A, B, C, \ldots$. For the purpose of a unified notation with the signal processing literature, row and column indices of vectors and matrices start from *zero* unless otherwise stated. The vector space of complex n-vectors is denoted by $\mathbb{C}^n$.

Example 2.1 (Vector Notation) A 2-dimensional complex vector $x \in \mathbb{C}^2$ is expressed as

$$x = \begin{pmatrix} x_0 \\ x_1 \end{pmatrix}, \quad x_0, x_1 \in \mathbb{C}.$$

Complex m-by-n matrices are denoted by $\mathbb{C}^{m \times n}$.

Example 2.2 (Matrix Notation) A 2-by-3 complex matrix $A \in \mathbb{C}^{2 \times 3}$ is expressed as

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{pmatrix}, \quad a_{00}, \ldots, a_{12} \in \mathbb{C}.$$

Note that rows and columns are indexed from zero.

2.1.2 Submatrix Specification

Submatrices of $A \in \mathbb{C}^{m \times n}$ are denoted by $A(u, v)$, where u and v are called *index vectors* with the purpose to specify the rows and columns of A used to construct the respective submatrix.

Index vectors are specified using the *colon notation*:

$$u = k : j \quad \leftrightarrow \quad u = (k, k+1, \ldots, j), \quad k \leq j.$$

Example 2.3 (Submatrix Notation) $A(2:4, 3:7) \in \mathbb{C}^{3\times 5}$ is a 3-by-5 submatrix of $A \in \mathbb{C}^{m\times n}$ $(m \geq 4, n \geq 7)$ defined by the rows $2, 3$ and 4 and the columns $3, 4, \ldots, 7$ of A.

There are special notational conventions when all rows or columns are extracted from their parent matrix. In particular, if $A \in \mathbb{C}^{m\times n}$, then

$$\begin{array}{ll} A(u,:) & \Leftrightarrow \; A(u, 0:n-1) \\ A(:,v) & \Leftrightarrow \; A(0:m-1, v). \end{array}$$

Vectors with non-unit increment are specified by the notation

$$u = k : j : i \quad \Leftrightarrow \quad u = (k, k+i, \ldots, j),$$

where $i \in \mathbb{Z}\backslash\{0\}$ denotes the increment.

2.1.3 Column and Row Partitioning

Let $A \in \mathbb{C}^{m\times n}$ and $a_{:,j} \in \mathbb{C}^m$ designate the jth column and $a_{k,:} \in \mathbb{C}^n$ the kth row of A, then

$$A = (a_{:,0} \mid a_{:,0} \mid \cdots \mid a_{:,n-1})$$

is a *column partitioning*, and

$$A = \begin{pmatrix} a_{0,:} \\ a_{1,:} \\ \vdots \\ a_{m-1,:} \end{pmatrix}$$

is a *row partitioning*.

2.1.4 Direct Vector Sum

Definition 2.1 (Direct Vector Sum) *Let $y_N = (x_1, \ldots, x_{n-1})$ be a vector of length $N = nm$ and*

$$x_0 = (u_0, \ldots, u_{m-1}), \ldots, x_i = (u_{mi}, \ldots, u_{m(i+1)-1}), \ldots, x_{n-1} = (u_{mn}, \ldots, u_{(mn)-1})$$

are subvectors partitioned of y_n, then the direct sum of vectors is defined by

$$y_N = \bigoplus_{i=0}^{n-1} x_i = x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1} = (u_0, \ldots, u_{(nm)-1})^\top \in \mathbb{C}^N.$$

2.1.5 Direct Matrix Sum

Definition 2.2 (Direct Matrix Sum) *For matrices of any dimension, e. g., A_0, A_2, ..., A_3, the direct sum is defined as the block diagonal matrix*

$$\sum_{i=1}^{n-1} A_i = A_0 \oplus A_1 \oplus \ldots \oplus A_{n-1} = \begin{pmatrix} A_0 & 0 & \ldots & 0 \\ 0 & A_1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & A_{n-1} \end{pmatrix}.$$

2.1.6 Elementwise Multiplication

If $x, y \in \mathbb{C}^n$, then $x * y \in \mathbb{C}^n$ is defined elementwise by

$$(x * y)_i := x_i y_i, \quad i = 0 : n-1.$$

More generally, if $A, B \in \mathbb{C}^{m\times n}$, then the product $C = A * B \in \mathbb{C}^{m\times n}$ is defined element wise by

$$c_{kj} := a_{kj} b_{kj}, \quad k = 0 : m-1, \quad j = 0 : n-1.$$

The result of the elementwise multiplication is also known as *Schur or Hadamard product*.

2.1.7 Storage Conventions in Fortran and C

There are two different approaches to arrange a data vector into a matrix.

Let $N = n_1 n_2$

1. The data vector $x \in \mathbb{C}^N$ is arranged into a matrix $x_{n_1\times n_2} \in \mathbb{C}^{n_1\times n_2}$ in *column major order* (Fortran storage convention), i. e.,

$$[x_{n_1\times n_2}]_{k,j} := x_{k+jn_1} \quad \text{with} \quad k = 0 : n_1 - 1, \quad j = 0 : n_2 - 1 \tag{2.1}$$

2. The data vector $x \in \mathbb{C}^N$ is arranged into a matrix $x_{n_2\times n_1} \in \mathbb{C}^{n_2\times n_1}$ in *row major order* (C storage convention), i. e.,

$$[x_{n_2\times n_1}]_{k,j} := x_{j+kn_1} \quad \text{with} \quad j = 0 : n_1 - 1, \quad k = 0 : n_2 - 1 \tag{2.2}$$

(2.2) corresponds to the transposed (2.1) storage convention, i. e.,

$$x_{n_2\times n_1} := x_{n_1\times n_2}^{\top}.$$

2.2 Kronecker Products

The block structures arising in factorizations of the DFT matrix are highly regular and to describe and manipulate them, a special notation is used.

Definition 2.3 (Kronecker Product) *The Kronecker product (direct product or tensor product) of the matrices $A \in \mathbb{C}^{m_1 \times n_1}$ and $B \in \mathbb{C}^{m_2 \times n_2}$ is the block structured matrix*

$$A \otimes B := \begin{pmatrix} a_{0,0}B & \dots & a_{0,n_1-1}B \\ \vdots & \ddots & \vdots \\ a_{m_1-1,0}B & \dots & a_{m_1-1,n_1-1}B \end{pmatrix} \in \mathbb{C}^{m_1 m_2 \times n_1 n_2}.$$

2.3 Algebraic Properties of Kronecker Products

Kronecker products have the following algebraic properties (Horn, Johnson [38]).

Property 2.1 (Associativity) *If A, B, C are arbitrary matrices, then*

$$(A \otimes B) \otimes C = A \otimes (B \otimes C).$$

Thus, the expression $A \otimes B \otimes C$ is unambiguous.

Property 2.2 (Transposition) *If A, B are arbitrary matrices, then*

$$(A \otimes B)^\top = A^\top \otimes B^\top.$$

Property 2.3 (Inversion) *If A, B are arbitrary matrices, then*

$$(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}.$$

Property 2.4 (Mixed-Product Property) *If A, B, C, D are arbitrary matrices for which the products AC and BD are defined, then*

$$(A \otimes B)\,(C \otimes D) = A\,C \otimes B\,D.$$

Special cases of this property occur when $A = C = I$ or $B = D = I$.

The mixed-product property can be generalized in two different ways (for matrices of appropriate size):

$$(A_1 \otimes A_2 \otimes \cdots \otimes A_k)(B_1 \otimes B_2 \otimes \cdots \otimes B_k) = A_1B_1 \otimes A_2B_2 \cdots \otimes A_kB_k,$$

and

$$(A_1 \otimes B_1)(A_2 \otimes B_2) \cdots (A_k \otimes B_k) = (A_1A_2 \cdots A_k) \otimes (B_1B_2 \cdots B_k).$$

The Kronecker product can be distributed with respect to multiplication with an identity matrix.

Property 2.5 (Distributivity) *If A is an arbitrary matrix, then*

$$I_p \otimes (I_q \otimes A) = I_{pq} \otimes A.$$

A consequence of this property is the following decomposition.

2.3.1 The Kronecker Product Decomposition

Corollary 2.1 (Decomposition) *If $A \in \mathbb{C}^{m_1 \times n_1}$ and $B \in \mathbb{C}^{m_2 \times n_2}$, then*

$$A \otimes B = AI_{n_1} \otimes I_{m_2}B = (A \otimes I_{m_2})(I_{n_1} \otimes B),$$
$$A \otimes B = I_{m_1}A \otimes BI_{n_2} = (I_{m_1} \otimes B)(A \otimes I_{n_2}).$$

According to this, a Kronecker product $A \otimes B$ can be multiplied with an input vector x by executing in two stages. One performing a *vector* and another performing a *parallel* operation. As will be seen shortly, each of these factors establishes equivalence between Kronecker product compositions and computational structures.

Stride permutations, which are very common in use with Kronecker products, can transform a parallel into a vector factor and vice versa when they are used to regroup data of the multiplied input operand.

2.3.2 The Kronecker Vector-Matrix Product

The multiplication of a vector by a matrix consiting of the Kronecker product of two submatrices, one of them being an identity, can alternatively be executed by a matrix-product of two dense matrices, describe in

Property 2.6 (Column Multiply) *If $A \in \mathbb{C}^{r \times r}$ and $x \in \mathbb{C}^n$ with $n = rc$, then*

$$y = (I_c \otimes A)x \quad \Leftrightarrow \quad y_{r \times c} = Ax_{r \times c}.$$

and

Property 2.7 (Row Multiply) *If $A \in \mathbb{C}^{c \times c}$ and $x \in \mathbb{C}^n$ with $n = rc$, then*

$$y = (A \otimes I_r)x \quad \Leftrightarrow \quad y_{r \times c} = x_{r \times c}A^\top.$$

2.4 Kronecker Products and Parallel Programming

A connection between Kronecker products and computer architecture can be established by associating special types of Kronecker products with particular types of processor and / or memory organization.

2.4.1 Kronecker Parallel Factors

$I_n \otimes B$ is the *direct sum* (Definition 2.2) of n block diagonal copies of B and therefore called *Kronecker parallel factor*. If the vector x is distributed block-wise to n processors, a matrix-vector multiplication can be done in parallel without any inter-processor communication, which yields optimal results on parallel architectures.

Let $B_k \in \mathbb{C}^{k\times k}$ and $I_l \in \mathbb{C}^{l\times l}$ be the identity matrix. Then

$$(I_l \otimes B_k)x = \begin{pmatrix} B_k & & & \\ & B_k & & \\ & & \ddots & \\ & & & B_k \end{pmatrix} \begin{pmatrix} x(0:k-1) \\ x(k:2k-1) \\ \vdots \\ x((l-1)k:kl-1) \end{pmatrix}.$$

Expressions of the form $I_l \otimes B_k$ are called *parallel stages* because they can be implemented efficiently as distributed, independent workloads on parallel architectures.

Example 2.4 (Parallel Stages) Let $B_2 \in \mathbb{C}^{2\times 2}$ and let $I_3 \in \mathbb{C}^{3\times 3}$ be the identity matrix. Then

$$y := (I_3 \otimes B_2)x$$

is given by

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{pmatrix} := \begin{pmatrix} b_{0,0} & b_{0,1} & & & & \\ b_{1,0} & b_{1,1} & & & & \\ & & b_{0,0} & b_{0,1} & & \\ & & b_{1,0} & b_{1,1} & & \\ & & & & b_{0,0} & b_{0,1} \\ & & & & b_{1,0} & b_{1,1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix}.$$

This matrix-vector product can be realized by splitting the input vector $x \in \mathbb{C}^6$ into 3 subvectors of length 2 and performing the respective matrix-vector products

$$B_2 x(2j : 2(j+1)-1), \quad j = 0, 1, 2$$

in parallel on 3 processors P_j. Each processor P_j has to compute:

$$P_0: \quad \begin{pmatrix} y_0 \\ y_1 \end{pmatrix} \begin{pmatrix} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix},$$

$$P_1: \quad \begin{pmatrix} y_0 \\ y_1 \end{pmatrix} \begin{pmatrix} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix},$$

$$P_2: \quad \begin{pmatrix} y_0 \\ y_1 \end{pmatrix} \begin{pmatrix} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix}.$$

The operation $y := (I_l \otimes B_k)x$ can be implemented either sequentially (as a loop) or as a parallel operation (Johnson et al. [39]).

Algorithm 2.1 ($y := (I_l \otimes B_k)x$)
do $i = 0 : l-1$
$y(ik : (i+1)k-1) := B_k x(ik : (i+1)k-1)$
end do

2.4.2 Kronecker Vector Factors

$A \otimes I_n$ is called a *Kronecker vector factor* because it can be used to represent a vector operation on vectors of length n. Vector x is distributed on n processors for vector computation, and therefore a cyclic distribution of x is required without requiring either replication or communication of data.

Let $A_k \in \mathbb{C}^{k\times k}$ and let $I_l \in \mathbb{C}^{l\times l}$ be the identity matrix. Then

$$\begin{aligned}(A_k \otimes I_l)x &= \begin{pmatrix} a_{0,0}I_l & \dots & a_{0,k-1}I_l \\ \vdots & \ddots & \vdots \\ a_{k-1,0}I_l & \dots & a_{k-1,k-1}I_l \end{pmatrix} \begin{pmatrix} x(0:l-1) \\ x(l:2l-1) \\ \vdots \\ x((k-1)l : kl-1) \end{pmatrix} \\ &= \begin{pmatrix} a_{0,0}(0:l-1) + \ldots + a_{0,k-1}x((k-1)l : kl-1) \\ \vdots \\ a_{k-1,0}(0:l-1) + \ldots + a_{k-1,k-1}x((k-1)l : kl-1) \end{pmatrix}.\end{aligned}$$

Since these operations may be performed by a vector processor, expressions of the form $A_k \otimes I_l$ are called *vector stages*.

Example 2.5 (Vector Stage) Let $A_2 \in \mathbb{C}^{2\times 2}$ and let $I_3 \in \mathbb{C}^{3\times 3}$ be the identity matrix. Then

$$y := (A_2 \otimes I_3)x$$

is given by

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{pmatrix} := \begin{pmatrix} a_{0,0} & & & a_{0,1} & & \\ & a_{0,0} & & & a_{0,1} & \\ & & a_{0,0} & & & a_{0,1} \\ a_{1,0} & & & a_{1,1} & & \\ & a_{1,0} & & & a_{1,1} & \\ & & a_{1,0} & & & a_{1,1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix}.$$

This matrix-vector product can be computed by splitting the input vector $x \in \mathbb{C}^6$ into 2 subvectors of length 3 and performing single scalar multiplications with these subvectors:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} := a_{0,0} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} + a_{0,1} \begin{pmatrix} x_3 \\ x_4 \\ x_5 \end{pmatrix},$$

$$\begin{pmatrix} y_3 \\ y_4 \\ y_5 \end{pmatrix} := a_{1,0} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} + a_{1,1} \begin{pmatrix} x_3 \\ x_4 \\ x_5 \end{pmatrix}.$$

Chapter 3

Permutations

3.1 Stride Permutations

Stride permutations are frequently used tools in Kronecker product representations of FFT algorithms because of their ability to commute[1] a Kronecker product factor. A stride permutation L_n^{mn} is an $mn \times nm$ permutation matrix.

Definition 3.1 (Stride Permutation) *For a vector $x \in \mathbb{C}^{mn}$ the stride permutation L_n^{mn} is defined by*

$$\mathrm{L}_n^{mn}\, x := \begin{pmatrix} x(0 : (m-1)n : n) \\ x(1 : (m-1)n + 1 : n) \\ \vdots \\ x(n-1 : mn-1 : n) \end{pmatrix},$$

$$\mathrm{L}_n^{mn}\, x = j \to x(j \cdot n \ mod \ mn - 1), \quad for \ j = 0, \ldots, mn-2; \quad mn-1 \to mn-1.$$

The permutation operator L_n^{mn} sorts the components of x according to their index modulo n. Thus, starting from the first element, elements with indices equal to $0 \bmod n$ come first and then starting from the second element, elements of x with indices equal to $1 \bmod n$ and so on.

The notation L_n^{mn} indicates that the elements of a vector of length mn are loaded into m segments each at stride n. This operation is also called an *n-way perfect shuffle permutation*.

3.1.1 Even-Odd Sort Permutations

The permutation $y := \mathrm{L}_2^n\, x$ (n even) is called an *even-odd sort permutation*, because it groups the even-indexed and odd-indexed components together.

[1] Convert a parallel factor into a vector factor and vice versa.

Example: For $x \in \mathbb{C}^8$, $y := \mathrm{L}_2^8\, x$ is given by

$$\mathrm{L}_2^8\, x = \begin{pmatrix} 1 & . & . & . & . & . & . & . \\ . & . & 1 & . & . & . & . & . \\ . & . & . & . & 1 & . & . & . \\ . & . & . & . & . & . & 1 & . \\ . & 1 & . & . & . & . & . & . \\ . & . & . & 1 & . & . & . & . \\ . & . & . & . & . & 1 & . & . \\ . & . & . & . & . & . & . & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} = \begin{pmatrix} x_0 \\ x_2 \\ x_4 \\ x_6 \\ x_1 \\ x_3 \\ x_5 \\ x_7 \end{pmatrix}$$

with the zeroes represented as dots.

If $x \in \mathbb{C}^n$, then the even-odd sort permutation $y := \mathrm{L}_2^n\, x$ ($y \in \mathbb{C}^n$) can be implemented as

Algorithm 3.1 ($y := \mathrm{L}_2^n\, x$)

$n_* := n/2$
do $i = 0 : n_* - 1$
  $y(i) := x(2i)$
  $y(i + n_*) := x(2i + 1)$
end do

3.1.2 Perfect Shuffle Permutations

The permutation $y := \mathrm{L}_{n/2}^n\, x$ (n even) is called a *perfect shuffle permutation*, since its action on a deck of cards would be the shuffling of two equal piles of cards so that the cards are interleaved one from each pile. Because of its importance, the perfect shuffle permutation $\mathrm{L}_{n/2}^n$ is denoted in short by Π_n.

Example: For $x \in \mathbb{C}^8$, $y := \mathrm{L}_4^8\, x = \Pi_8 x$ is given by

$$\Pi_8 x = \begin{pmatrix} 1 & . & . & . & . & . & . & . \\ . & . & . & . & 1 & . & . & . \\ . & 1 & . & . & . & . & . & . \\ . & . & . & . & . & 1 & . & . \\ . & . & 1 & . & . & . & . & . \\ . & . & . & . & . & . & 1 & . \\ . & . & . & 1 & . & . & . & . \\ . & . & . & . & . & . & . & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} = \begin{pmatrix} x_0 \\ x_4 \\ x_1 \\ x_5 \\ x_2 \\ x_6 \\ x_3 \\ x_7 \end{pmatrix}$$

with the zeroes represented as dots.

For $x \in \mathbb{C}^n$ the perfect shuffle permutation $y := \Pi_n x$ ($y \in \mathbb{C}^n$) can be implemented as

Algorithm 3.2 ($y := \Pi_n x$)

$n_* := n/2$

do $i = 0 : n_* - 1$
  $y(2i) := x(i)$
  $y(2i+1) := x(n_* + i)$
end do

3.1.3 Algebraic Properties of Stride Permutations

Stride permutations have the following algebraic properties for arbitrary positive integers.

Property 3.1 (Identity)

$$\mathrm{L}_1^m = \mathrm{L}_m^m = \mathrm{I}_m$$

Property 3.2 (Inversion/Transposition)

$$(\mathrm{L}_n^{mn})^{-1} = (\mathrm{L}_n^{mn})^\top = \mathrm{L}_m^{nm}$$

Example:

$$(\mathrm{L}_2^{2^i})^{-1} = \mathrm{L}_{2^{i-1}}^{2^i} = \Pi_{2^i}$$

Property 3.3 (Multiplication)

$$\mathrm{L}_m^{pmn}\,\mathrm{L}_n^{pmn} = \mathrm{L}_n^{pmn}\,\mathrm{L}_m^{pmn} = \mathrm{L}_{mn}^{pmn}$$

One central reason why it is practical to use stride permutations in combination with Kronecker products is that Kronecker products are not commutative. However, the following holds.

Property 3.4 (Commutation) *If $A \in \mathbb{C}^{m\times m}$, and $B \in \mathbb{C}^{n\times n}$, then*

$$\mathrm{L}_n^{mn}(A \otimes B) = (B \otimes A)\,\mathrm{L}_n^{mn}\,.$$

Using property (3.2) leads to

$$A \otimes B = \mathrm{L}_m^{nm}(B \otimes A)\,\mathrm{L}_n^{mn}\,.$$

A stride permutation matrix partitions the computations and communications by changing the data flows and can therefore be used to define the required data distribution for a parallel operation, or the communication pattern needed to transform parallel block structure into cyclic vector structure and vice versa. The inverse permutation matrix finally reorders data to initial distribution.

Frequently used properties which can be traced back to those stated before are the following.

Property 3.5 *If $A \in \mathbb{C}^{m\times m}$, and $B \in \mathbb{C}^{n\times n}$, then*

$$A \otimes B = \mathrm{L}^{nm}_m(\mathrm{I}_n \otimes A)\,\mathrm{L}^{mn}_n(\mathrm{I}_m \otimes B).$$

Property 3.6 *If $N = rst$, then*

$$\mathrm{L}^N_{st} = \mathrm{L}^{rst}_s\,\mathrm{L}^{rst}_t = \mathrm{L}^{rst}_t\,\mathrm{L}^{rst}_s\,.$$

Property 3.7 *If $N = rst$, then*

$$\mathrm{L}^{rst}_t = (\mathrm{L}^{rt}_t \otimes \mathrm{I}_s)(\mathrm{I}_r \otimes \mathrm{L}^{st}_t).$$

Lemma 3.1 *If $N = rs^2t$, then*

$$\mathrm{L}^{rs^2t}_{rs} = (\mathrm{L}^{rst}_{rs} \otimes \mathrm{I}_s)(\mathrm{I}_{rt} \otimes \mathrm{L}^{s^2}_s)(\mathrm{I}_t \otimes \mathrm{L}^{rs}_r \otimes \mathrm{I}_s).$$

3.2 Stride Permutations and Parallelism

In parallel programming stride permutations do not only represent local permutations, but also global communication among the involved processors. As global communication requires other kinds of implementation than local permutations do, it is necessary to separate these two parts.

Definition 3.2 (Communication Factor) *A communication factor for communicating elements of a vector of length n on p processors has the form*

$$C \otimes \mathrm{I}_b \tag{3.1}$$

where $pmb = n$ and the communication pattern C satisfies

$$C = \left[c_{i,j}\right]_{0\le i,j<pm}, \quad c_{i,j} \in \{0,\,1\}\,,$$

$$\forall i : \exists | j : c_{ij} = 1, \quad \forall j : \exists | i : c_{ij} = 1,$$
$$\forall j \in \left\{\lfloor \tfrac{i}{m} \rfloor m, \ldots, (\lfloor \tfrac{i}{m} \rfloor + 1)m - 1\right\} \setminus \{i\} : c_{ij} = 0.$$

The product (3.1) represents the communication of m blocks of size b per processor with communication pattern C.

The definition of a communication factor requires each of the $p \times m$ data blocks to be either communicated over the network or to stay in its original memory location. This excludes local permutations which are handled in Kronecker parallel stages as introduced in Section 2.4.1.

Example 3.1 (Communication Pattern Restrictions) The following matrix illustrates the restrictions of a communication pattern with $p = m = 4$ and $b = 1$.

$$\left[\begin{array}{cccc|cccc|cccc|cccc}
\cdot & 0 & 0 & 0 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
0 & \cdot & 0 & 0 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
0 & 0 & \cdot & 0 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
0 & 0 & 0 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\hline
\cdot & \cdot & \cdot & \cdot & \cdot & 0 & 0 & 0 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & 0 & \cdot & 0 & 0 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & 0 & 0 & \cdot & 0 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & 0 & 0 & 0 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\hline
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 0 & 0 & 0 & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 0 & \cdot & 0 & 0 & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 0 & 0 & \cdot & 0 & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 0 & 0 & 0 & \cdot & \cdot & \cdot & \cdot & \cdot \\
\hline
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 0 & 0 & 0 \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 0 & \cdot & 0 & 0 \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 0 & 0 & \cdot & 0 \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 0 & 0 & 0 & \cdot
\end{array}\right]$$

Each row and each column may only contain one element 1 to maintain the permutation. The zeros in the diagonal blocks would are mandatory. An element 1 on one of these positions would represent a local permutation which is prohibited.

3.2.1 All-to-All Communication

The stride permutation $\mathrm{L}^{p^2}_p$ is a special pattern for communication among p processors.

A communication factor of the form $\mathrm{L}^{p^2}_p \otimes \mathrm{I}_b$ communicates one data block of size b from each processor to every other processor. One such data block also remains locally on each processor. These non-communicated data blocks do not only remain on the same processors, but also in the same memory locations, as the corresponding matrix entries are on the main diagonal. Therefore $\mathrm{L}^{p^2}_p$ satisfies the restrictions of a communication pattern according to Definition 3.2.

Because

$$\mathrm{L}^{p^2}_p \, \mathrm{L}^{p^2}_p = \mathrm{L}^{p^2}_{p^2} = \mathrm{I}_{p^2}$$

it follows that $\mathrm{L}^{p^2}_p$ is a self-inverse permutation and has a maximum cycle length of two. This means that it only consists of pairwise exchanges of data blocks and fixpoints. These properties make the stride permutation $\mathrm{L}^{p^2}_p$ a very attractive communication pattern as it can be split into pairwise communication steps for implementation.

Example 3.2 (All-to-All Communication Matrices) $\mathrm{L}_p^{p^2}$ all-to-all communication matrices among 2, 3, and 4 processors:

$$\mathrm{L}_2^4 = \left[\begin{array}{cc|cc} \mathbf{1} & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot \\ \hline \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \mathbf{1} \end{array}\right] \qquad \mathrm{L}_3^9 = \left[\begin{array}{ccc|ccc|ccc} \mathbf{1} & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \hline \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \mathbf{1} & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \hline \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \mathbf{1} \end{array}\right]$$

$$\mathrm{L}_4^{16} = \left[\begin{array}{cccc|cccc|cccc|cccc} \mathbf{1} & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \hline \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \mathbf{1} & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \hline \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \mathbf{1} & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \hline \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \mathbf{1} \end{array}\right]$$

The following example shows that $\mathrm{L}_p^{p^2}$ effectively *transposes* a distributed matrix stored as an array. The bold ones correspond to the data blocks which remain locally on the same processor in the same memory location.

Example 3.3 ($\mathbf{L_3^9}$ Applied to a Data Vector) L_3^9 applied to a vector A, which represents a 3×3 matrix.

$$A = \left(\begin{array}{c} a_1 \\ a_2 \\ a_3 \\ \hline a_4 \\ a_5 \\ a_6 \\ \hline a_7 \\ a_8 \\ a_9 \end{array}\right) \widehat{=} \left[\begin{array}{ccc} a_1 & a_2 & a_3 \\ \hline a_4 & a_5 & a_6 \\ \hline a_7 & a_8 & a_9 \end{array}\right] \qquad \Longrightarrow \qquad \mathrm{L}_3^9 \cdot A = \left(\begin{array}{c} a_1 \\ a_4 \\ a_7 \\ \hline a_2 \\ a_5 \\ a_8 \\ \hline a_3 \\ a_6 \\ a_9 \end{array}\right) \widehat{=} \left[\begin{array}{ccc} a_1 & a_4 & a_7 \\ \hline a_2 & a_5 & a_8 \\ \hline a_3 & a_6 & a_9 \end{array}\right]$$

Parallel matrix transposition [8] is often required in parallel algorithms. Thus, the message passing library MPI [47] provides the function `MPI_Alltoall` which performs a parallel matrix transposition and saves the application programmer

to care about details of the implementation. However, the pairwise direct total exchange algorithm, introduced in the following section, shows a relatively easy way of implementing the parallel matrix transposition split into steps of pairwise communication.

The following lemma provides formulas to split a general stride permutation L_n^{mn} into explicit matrix transpositions and local permutation steps.

Lemma 3.2 *If $p|m$ and $p|n$, then*

$$\begin{aligned} & \mathrm{L}_m^{mn} &=& \left(\mathrm{I}_p \otimes \mathrm{L}_{m/p}^{mn/p}\right)\left(\mathrm{L}_p^{p^2} \otimes \mathrm{I}_{mn/p^2}\right)\left(\mathrm{I}_p \otimes \mathrm{L}_p^n \otimes \mathrm{I}_{m/p}\right) \\ m = p \Longrightarrow \quad & \mathrm{L}_p^{np} &=& \left(\mathrm{L}_p^{p^2} \otimes \mathrm{I}_{n/p}\right)\left(\mathrm{I}_p \otimes \mathrm{L}_p^n\right) \\ n = p \Longrightarrow \quad & \mathrm{L}_m^{mp} &=& \left(\mathrm{I}_p \otimes \mathrm{L}_{m/p}^m\right)\left(\mathrm{L}_p^{p^2} \otimes \mathrm{I}_{m/p}\right). \end{aligned}$$

3.2.2 Pairwise Direct Total Exchange

The all-to-all personalized communication or simply direct total exchange algorithm allows to perform a parallel matrix transposition with optimal usage of the communication channels, disregarding network congestion issues. It is a direct algorithm, meaning each data packet is sent directly from source to destination without intermediate buffering. Requiring $p-1$ steps, where p is the number of processors, in step $i = 1, 2, \ldots, p-1$, each node exchanges data with the node determined by taking the bitwise exclusive-or of its number and i. Therefore this algorithm has the property that the entire communication pattern is decomposed into a sequence of pairwise exchanges.

Algorithm 3.3 (Direct Total Exchange Algorithm) At step i the processor $j-1$ exchanges data with the processor number $\mathrm{XOR}(j-1, step)$.

do $i_{step} = 1 : p-1$
  do $j_{proc} = 0 : p$
    $i_{dest} = \mathrm{XOR}(j_{proc}, i_{step})$
  end do
end do

XOR denotes the logical exclusive OR operator applied to the binary representation of integer values.

The diagonal blocks in Step 0 do not need to be globally transposed because each block still holds its prior position after transposition.

	Proc 0	Proc 1	Proc 2	Proc 3	Proc 4	Proc 5	Proc 6	Proc 7
Step 1	1	0	3	2	5	4	7	6
Step 2	2	3	0	1	6	7	4	5
Step 3	3	2	1	0	7	6	5	4
Step 4	4	5	6	7	0	1	2	3
Step 5	5	4	7	6	1	0	3	2
Step 6	6	7	4	5	2	3	0	1
Step 7	0	6	5	4	3	2	1	0

Table 3.1: Send table of a direct exchange matrix transposition of size 8×8.

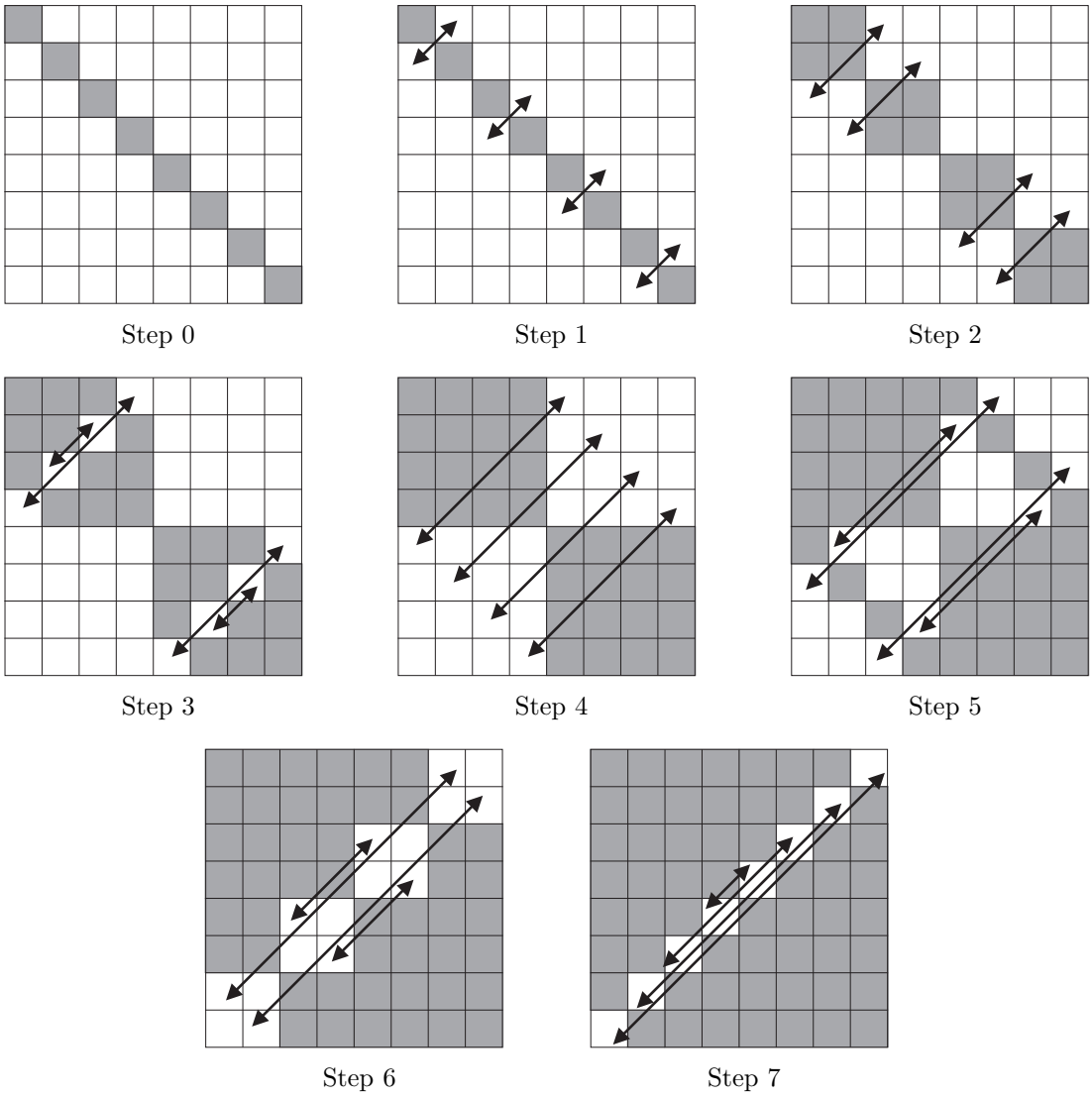


Figure 3.1: Direct exchange matrix transposition of size 8×8.

3.3 Extended Stride Permutations

The original definition of stride permutations as given in Definition 3.1 is somewhat limiting. Especially the set of stride permutations is not closed with respect to multiplication operations. Therefore the definition of stride permutations will be extended as follows.

Definition 3.3 (Stride Permutation) *For $b \in Unit(\mathbb{N}_{a-1})$ The stride permutation's generating function $\ell^a_b(j)$ and the stride permutation matrix L^a_b are defined by*

$$\begin{aligned} \ell^a_b(j) &:= \begin{cases} jb \mod a-1, & \text{for } j = 0,\ldots,a-2 \\ a-1 & \text{for } j = a-1 \end{cases} \\ \mathrm{L}^a_b &:= \left[e_a^{\ell^a_b(0)}, e_a^{\ell^a_b(1)}, \ldots, e_a^{\ell^a_b(a-1)}\right]^T \end{aligned}$$

with e^i_j being the i-th unitvector of length j.

Because of

$$\mathrm{L}^{mn}_n = \mathrm{L}^a_b \quad \Rightarrow \quad b|a \quad \Rightarrow \quad b \nmid a-1 \quad \Rightarrow \quad b \in Unit(\mathbb{N}_{a-1}).$$

this is an extension of Definition 3.1.

Property 3.8 (Cosets of Stride Permutations)

$$\mathrm{L}^a_b = \mathrm{L}^a_c \qquad \Leftrightarrow \qquad b \equiv c \mod a-1$$

The former multiplication property (Property 3.3) can now be easily redefined to the following more elegant property.

Property 3.9 (Multiplication of Extended Stride Permutations)

$$\mathrm{L}^a_b\,\mathrm{L}^a_c = \mathrm{L}^a_{bc} = \mathrm{L}^a_{bc \mod a-1}.$$

With this redefinition, e. g., both the products $\mathrm{L}^6_2\,\mathrm{L}^6_2$ and $\mathrm{L}^6_3\,\mathrm{L}^6_3$ can be described as the stride permutation L^6_4, which was not possible using Definition 3.1.

$$\begin{array}{ll} \mathrm{L}^6_2\,\mathrm{L}^6_2 & = \mathrm{L}^6_4 \\ \mathrm{L}^6_3\,\mathrm{L}^6_3 = \mathrm{L}^6_9 & = \mathrm{L}^6_4 \end{array} \qquad \mathrm{L}^6_4 = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 \end{bmatrix}$$

The set of extended stride permutations of a certain size $\{\mathrm{L}^a_b\,|b \in Unit(a-1)\}$ is isomorphic to the unit-group of $\mathbb{N}_{a-1}$ and thus $(\mathrm{L}^a_\cdot, \cdot)$ is closed and a group.

3.4 Digit Permutations

Stride permutations only cover a very limited class of permutations. Products of tensor products of stride permutations and unit matrices (e. g., $(\mathrm{I}_k \otimes \mathrm{L}^l_m)\,\mathrm{L}^{kl}_n$) are no stride permutations any more. As such products arise during the decomposition of signal transform algorithms, it is necessary to find a way to handle these constructs in a satisfying way.

Stride permutations of size 2^n can be interpreted as permutations of the n digits of the binary representation of the index. For instance, L^8_2 corresponds to the permutation (1 2 3) applied on the index' three digits.

$$\mathrm{L}^8_2\,[0,1,2,3,4,5,6,7]^T = [0,2,4,6,1,3,5,7]^T$$

	digits			permuted			
index	x_1	x_2	x_3	x_2	x_3	x_1	result
0	0	0	0	0	0	0	0
1	0	0	1	0	1	0	2
2	0	1	0	1	0	0	4
3	0	1	1	1	1	0	6
4	1	0	0	0	0	1	1
5	1	0	1	0	1	1	3
6	1	1	0	1	0	1	5
7	1	1	1	1	1	1	7

Digit permutations expand the domain of stride permutations by replacing the binary representation with some arbitrary—not necessarily prime—factorization of the index domain. If n is the length of the vector to be permuted, a representation $n_1, n_2, \ldots, n_k$ of n is chosen. n_1 is the most significant digit, n_k the least significant one. This allows to uniquely represent every number in the index domain $i \in \{0, 1, \ldots, n-1\}$ by digits $(i_1^{n_1}, i_2^{n_2}, \ldots, i_k^{n_k})$ where the superscript numbers indicate the domain of the digits.

This convention allows to handle a significantly larger class of permutations than by using simple stride permutations.

Definition 3.4 (Digit Permutation) *With n, $n_1, \ldots, n_k$, i, and $i_1^{n_1}, \ldots, i_k^{n_k}$ as defined above, and p being a permutation on $\{0, 1, \ldots, k-1\}$, the digit permutation's generating function $d_p^{(n_1,n_2,\ldots,n_k)}(i)$ and matrix $\mathrm{D}_p^{(n_1,n_2,\ldots,n_k)}$ can be defined by*

$$\begin{aligned} d_p^{(n_1,n_2,\ldots,n_k)}(i^n) &= d_p^{(n_1,n_2,\ldots,n_k)}((i_1^{n_1}, i_2^{n_2}, \ldots, i_k^{n_k})) \\ &:= (i_{p(1)}^{n_{p(1)}}, i_{p(2)}^{n_{p(2)}}, \ldots, i_{p(k)}^{n_{p(k)}}) \\ \mathrm{D}_p^{(n_1,n_2,\ldots,n_k)}\, i^n &:= \left[e_n^{d_p^{(n_1,n_2,\ldots,n_k)}(0)}, e_n^{d_p^{(n_1,n_2,\ldots,n_k)}(1)}, \ldots, e_n^{d_p^{(n_1,n_2,\ldots,n_k)}(n-1)}\right]^T. \end{aligned}$$

Example 3.4 (Digit Permutation) The digit permutation $\mathrm{D}^{(2,3,2)}_{(2\ 3)}$ operates on a vector of length 12. The indices $i \in \{0, 1, \ldots 11\}$ are converted to the representation (i_1^2, i_2^3, i_3^2). The permutation $p = (2\ 3)$ is applied to this representation's digits and finally the indices are transformed back to the scalar representation.

$$\mathrm{D}^{(2,3,2)}_{(2\ 3)}\, x = \mathrm{D}^{(2,3,2)}_{(2\ 3)}((x_1^2, x_2^3, x_3^2) = (x_1^2, x_3^2, x_2^3)$$

	digits			permuted			
index	i_1^2	i_2^3	i_3^2	i_1^2	i_3^2	i_2^3	result
0	0	0	0	0	0	0	0
1	0	0	1	0	1	0	3
2	0	1	0	0	0	1	1
3	0	1	1	0	1	1	4
4	0	2	0	0	0	2	2
5	0	2	1	0	1	2	5
6	1	0	0	1	0	0	6
7	1	0	1	1	1	0	9
8	1	1	0	1	0	1	7
9	1	1	1	1	1	1	10
10	1	2	0	1	0	2	8
11	1	2	1	1	1	2	11

Thus the permutation matrix of the digit permutation $\mathrm{D}^{(2,3,2)}_{(2\ 3)}$ is given by

$$\underbrace{\begin{bmatrix}
1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1
\end{bmatrix}}_{\mathrm{D}^{(2,3,2)}_{(2\ 3)}}
\begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \end{bmatrix}
=
\begin{bmatrix} 0 \\ 3 \\ 1 \\ 4 \\ 2 \\ 5 \\ 6 \\ 9 \\ 7 \\ 10 \\ 8 \\ 11 \end{bmatrix}.$$

Stride permutations are a specific subset of digit permutations, namely those, whose permutation is a transposition of adjacent digits.

Property 3.10

$$\begin{aligned}
\mathrm{L}^{mn}_{m} &= \mathrm{D}^{(m,n)}_{(1\ 2)} \\
\mathrm{I}_a \otimes \mathrm{L}^{mn}_{m} &= \mathrm{D}^{(a,m,n)}_{(2\ 3)} \\
\mathrm{L}^{mn}_{m} \otimes \mathrm{I}_b &= \mathrm{D}^{(m,n,b)}_{(1\ 2)} \\
\mathrm{I}_a \otimes \mathrm{L}^{mn}_{m} \otimes \mathrm{I}_b &= \mathrm{D}^{(a,m,n,b)}_{(2\ 3)}
\end{aligned}$$

This property allows to identify $\mathrm{D}^{(2,3,2)}_{(2\ 3)}$ in Example 3.4 as $\mathrm{I}_2 \otimes \mathrm{L}^6_3$ which becomes apparent observing the permutation matrix:

$$\mathrm{D}^{(2,3,2)}_{(2\ 3)} = \left[\begin{array}{cccccc|cccccc} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \hline \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 \end{array}\right] = \mathrm{I}_2 \otimes \mathrm{L}^6_3 \,.$$

In general, digit permutations cannot be obtained by simply composing permutations. This is only possible if the resulting factorization of the first permutation is equivalent to the initial factorization of the second one.

Property 3.11 (Composition of Digit Permutations)

$$\mathrm{D}^{P(n_1,n_2,\ldots,n_k)}_{Q}\,\mathrm{D}^{(n_1,n_2,\ldots,n_k)}_{P} \;=\; \mathrm{D}^{(n_1,n_2,\ldots,n_k)}_{Q\circ P}$$

The inverse of a digit permutation is the digit permutation with the inverse permutation applied to the permuted factorization.

Property 3.12 (Inverse Digit Permutation)

$$\left(\mathrm{D}^{(n_1,n_2,\ldots,n_k)}_{P}\right)^{-1} = \mathrm{D}^{P(n_1,n_2,\ldots,n_k)}_{P^{-1}} = \left(\mathrm{D}^{(n_1,n_2,\ldots,n_k)}_{P}\right)^{T}$$

Chapter 4

The Fast Fourier Transform (FFT)

4.1 The Fourier Transform

The Fourier transform (FT) essentially decomposes a waveform, signal or function into sinusoids of different frequency whose sum reproduces the original function.

The *Fourier transform* F of a function f is defined as

$$F(\omega) := \int_{-\infty}^{\infty} f(t) e^{2\pi i \omega t} dt, \tag{4.1}$$

provided the integral exists as a Cauchy principal value[1] for any $\omega \in \mathbb{R}$. This transform maps the function $f(t)$, a signal taken as a function of time, into a complex valued function $F(\omega)$ which represents the signal as a function of frequency.

The inverse operation to (4.1) is

$$f(t) = \int_{-\infty}^{\infty} F(\omega) e^{-2\pi i \omega t} \, d\omega, \tag{4.2}$$

i. e., the *inverse Fourier transform*. Using (4.2) signals can be (re-)transformed from the frequency domain into the time domain.

The representation of a signal as a function of time is also said to be a representation in the *time domain*; the representation of a signal as a function of frequency is said to be a representation in the *frequency domain*. Both representations portray the same signal. Often the representation in the time domain appears "more natural" (in particular when the signal is taken or measured in this form), whereas the representation in the frequency domain is more suitable for filtering purposes and for any kind of manipulation of the spectrum.

Continuous functions f do not appear often in technical applications, instead individual observations (*samples*) of such functions at certain time points occur. In the following it is assumed that the sampling of f is carried out at equidistant time points whose distance is the sampling interval Δ. The quantity $1/\Delta$ is the *sampling rate*; it gives the number of discrete values of f per time unit.

[1] The Cauchy principal value of $\int_{-\infty}^{\infty} x(t)\,dt$ is defined as $\lim_{A\to\infty} \int_{-A}^{A} x(t)\,dt$.

original function	Fourier transform
real and even	real and even
real and odd	imaginary and odd
imaginary and even	imaginary and even
complex and even	complex and even
complex and odd	complex and odd
real and asymmetric	complex and asymmetric
imaginary and asymmetric	complex and asymmetric
real even plus imaginary odd	real
real odd plus imaginary even	imaginary
even	even
odd	odd

Table 4.1: Symmetry properties of the Fourier transform.

Definition 4.1 (Nyquist Frequency) *For any sampling interval* Δ*,*

$$\omega_c := \frac{1}{2\Delta}$$

denotes the Nyquist frequency.

The importance of this quantity is made clear in the following section.

4.2 The Discrete Fourier Transform

The following considerations deal with the Fourier transform of discrete (sampled) data sequences. Assuming that there are N data points, where, for sake of simplicity, it is presupposed that N is even (although all considerations remain valid when N is odd):

$$f_k := f(t_k), \qquad t_k := (k_0 N + k)\Delta, \qquad k = 0, 1, \ldots, N-1\,.$$

The form $k_0 N$ has been chosen because of notational simplifications.

For a piecewise continuous functions f with

$$\int_{-\infty}^{\infty} |f(t)|\, dt \;<\; \infty,$$

the Fourier transform F always exists. In this case the infinite integral (4.1) can be approximated by a finite sum (for a suitable selection of N, Δ and k_0) using numerical integration:

$$F(f) = \int_{-\infty}^{\infty} f(t) e^{2\pi i \omega t} dt \;\approx\; \sum_{k=0}^{N-1} f_k e^{2\pi i \omega t_k} \Delta.$$

Moreover, f can be interpolated, according to Theorem 4.1, using a continuous function g with

$$\lim_{t\to\pm\infty} g(t) = 0$$

and

$$G(\omega) = 0 \qquad \text{for} \quad \omega \notin (-\omega_c, \omega_c)$$

at the points t_k, i. e.

$$f(t_k) = g(t_k), \qquad k = 0, 1, \ldots, N-1.$$

The Fourier transform F of f, however, is now represented approximately by estimates of its function values at the points

$$\omega_n := \frac{n}{N\Delta}, \qquad n = -\frac{N}{2}, -\frac{N}{2}+1, \ldots, \frac{N}{2}-1 :$$

$$F(\omega_n) \approx \Delta \sum_{k=0}^{N-1} f_k e^{2\pi i \frac{n}{N\Delta}(k_0 N+k)\Delta} = \Delta \sum_{k=0}^{N-1} f_k e^{2\pi i k n/N}. \tag{4.3}$$

Using this formula, N discrete function values f_k lead to N discrete frequencies ω_n. Thus, instead of determining the Fourier transform $F(\omega)$ in the range $[-\omega_c, \omega_c]$, $F(\omega)$ is only determined for the discrete frequencies

$$\omega_n := \frac{n}{N\Delta}, \qquad n = -\frac{N}{2}, -\frac{N}{2}+1, \ldots, \frac{N}{2}-1.$$

The formula (4.3) is the *discrete Fourier transform* (DFT), and in this work will always be denoted F_n (which leaves out the factor Δ):

$$F_n := \sum_{k=0}^{N-1} f_k e^{2\pi i k n/N}\,. \tag{4.4}$$

This transform is periodic in n with the period N, i. e., $F_{N+n} = F_n$ is valid for any $n \in \mathbb{Z}$.

The connection between the continuous Fourier transform F of the function f and the discrete Fourier transform F_n of the data f_k, obtained by sampling f with a sampling interval Δ is represented as follows:

$$F(\omega_n) \approx \Delta F_n. \tag{4.5}$$

In (4.5) $n = 0$ corresponds to the frequency $\omega = 0$; positive frequencies $0 < \omega < \omega_c$ correspond to $1 \le n \le N/2-1$; and negative frequencies $-\omega_c < \omega < 0$ correspond to $N/2+1 \le n \le N-1$. For $n = N/2$, n corresponds both to the frequencies ω_c and $-\omega_c$.

The *inverse discrete Fourier transform* (inverse DFT), which can be derived from (4.2) analogously to (4.3), is given by:

$$f_k = \frac{1}{N} \sum_{n=0}^{N-1} F_n e^{-2\pi i k n/N}. \tag{4.6}$$

The only differences between (4.6) and (4.4) are the negative sign of the exponent and the division by N. Therefore, for the calculation of the discrete Fourier transform and its inverse, the same routines can be used to a large extent.

For a sinusoidal wave the Nyquist frequency ω_c is the highest frequency that is still reconstructable at a fixed length of the sampling interval Δ because, on the one hand, the argument of the expression of the form $\sin(2\pi\omega_0 t)$ with a frequency $\omega_0 \leq 1/(2\Delta)$ can move 2 quadrants, at the most, from one sampling to the next on the unit circle; and if, on the other hand, for the given sampling points, there exists a sinusoidal wave with a frequency larger than $1/(2\Delta)$, then there must always be one sinusoidal wave with a frequency smaller than $1/(2\Delta)$, since

$$\begin{aligned} &\sin\left[2\pi\left(\frac{1}{2\Delta}+\varepsilon\right)(t_0+k\Delta)\right] \\ =& -\sin\left[2\pi\left(\frac{1}{2\Delta}-\varepsilon\right)\left(\frac{t_0}{\varepsilon\Delta-\frac{1}{2}}+(t_0+k\Delta)\right)\right] \end{aligned}$$

for all $t_0, \epsilon \in \mathbb{R}$ and $k \in \mathbb{Z}$. Thus, at least two sample points per period are required in order to reconstruct a sine wave correctly.

Another restriction on the sampling of a function with the frequency ω appears if $N\omega/2\omega_c$ is not an integer. Then, in the discrete spectrum none of the discrete frequencies $\omega_n = 2\omega_c n/N$ appears. The frequency in the discrete spectrum which is next to the original frequency ω is clearly the largest; however, all other frequencies also appear more or less strongly in the spectrum. This phenomenon is called *leakage*.

Theorem 4.1 (Shannon's Sampling Theorem) *Let f denote a function with $\int_{-\infty}^{\infty} |f(t)|^2\, dt < \infty$, i. e., a signal with finite energy which is sampled at the rate $1/\Delta$. If f is band limited in its continuous frequency spectrum by the Nyquist frequency $\omega_c = 1/(2\Delta)$, i. e., if the Fourier transform F satisfies $F(\omega) = 0$ for any ω with $|\omega| \geq \omega_c$, then the function f can be recovered exactly from its sample values using the interpolation function*

$$g(t) = \frac{\sin(2\pi\omega_c t)}{2\pi\omega_c t}. \tag{4.7}$$

Thus, f may be expressed as

$$f(t) = \sum_{n=-\infty}^{\infty} f_n g(t - n\Delta), \tag{4.8}$$

where $f_n := f(n\Delta)$ are the samples of f.

If the function f is not band-limited, then in the Fourier transform of f all components of the frequency spectrum that lie outside of the range $[-\omega_c, \omega_c]$ are moved into this frequency range. This spectral overlap is called *aliasing*. In this case the function cannot be completely reconstructed using its sample values.

Thus, the length Δ of the sampling interval has to be selected so that the critical frequency ω_c is higher than all frequencies appearing in the spectrum of the data. Whether this condition applies to certain data can be seen from the frequency spectrum $F(\omega)$ of the data decreasing to zero, when the frequency ω tends to ω_c. If this is not the case, then a remedial precaution can be taken by shortening the sampling interval or by restricting the signal in its frequency spectrum (for instance, by using low-pass filtering) before sampling.

Another approach to understanding the DFT is to consider the transform as a simple change of coordinates. The column vectors of the transformation matrix are orthogonal and represent different frequency components. When multiplying the system matrix with the inverse matrix, the components of the solution vector indicate, how these column vectors compose the input.

When analyzing real-world data in form of a signal, this signal is represented (exactly or approximately) by a finite sequence of numbers. Computers are then used to calculate transformations of those signals. This is done by applying the *discrete Fourier transform (DFT).*

The following matrix notation is used to increase readability and to understand the FFT idea better. It will lead finally to the powerful Kronecker notation. Furthermore, the variables t and ω are replaced by indexing the vectors (arrays) x and y.

The DFT vector

$$y = (y_0, \ldots, y_{N-1})^\top \in \mathbb{C}^N$$

of the data vector

$$x = (x_0, \ldots, x_{N-1})^\top \in \mathbb{C}^N$$

is defined by

$$y_k := \sum_{j=0}^{N-1} \omega_N^{kj} x_j, \quad k = 0 : N-1, \tag{4.9}$$

where

$$\omega_N := \cos(2\pi/N) - i\,\sin(2\pi/N) = e^{-2\pi i/N}, \quad i = \sqrt{-1}.$$

The powers of ω_N are called *twiddle-factors*.

4.2.1 The Twiddle Factors

In matrix-vector terms, the DFT can be written as the matrix-vector product

$$y = F_N x.$$

The evaluation of a matrix-vector product needs $O(n^2)$ floating-point operations, resulting in quadratically increasing calculation times.

The elements of the matrix $F_N \in \mathbb{C}^{N\times N}$ are given by

$$[F_N]_{k,j} := \omega_N^{kj} = e^{-2\pi ikj/N}, \quad k, j = 0 : N-1,$$

where N is the number of samples.

The powers of ω_N are called *twiddle-factors* (Gentleman and Sande [31]).

Example 4.1 (DFT matrices) The DFT matrices F_1, F_2, F_3, F_4 and F_5 are given by

$$F_1 = (1), \quad F_2 = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \quad F_3 = \begin{pmatrix} 1 & 1 & 1 \\ 1 & \omega_3 & \omega_3^2 \\ 1 & \omega_3^2 & \omega_3 \end{pmatrix},$$

$$F_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix}, \quad F_5 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & \omega_5 & \omega_5^2 & \omega_5^3 & \omega_5^4 \\ 1 & \omega_5^2 & \omega_5^4 & \omega_5^5 & \omega_5^3 \\ 1 & \omega_5^3 & \omega_5^5 & \omega_5^4 & \omega_5^2 \\ 1 & \omega_5^4 & \omega_5^3 & \omega_5^2 & \omega_5 \end{pmatrix}.$$

Example 4.2 (DFT Matrix in ω Notation) The F_8 DFT matrix in ω notation

$$F_8 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \\ 1 & \omega^2 & \omega^4 & \omega^6 & 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega & \omega^4 & \omega^7 & \omega^2 & \omega^5 \\ 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^5 & \omega^2 & \omega^7 & \omega^4 & \omega & \omega^6 & \omega^3 \\ 1 & \omega^6 & \omega^4 & \omega^2 & 1 & \omega^6 & \omega^4 & \omega^2 \\ 1 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega \end{pmatrix},$$

where $\omega := \omega_8 = e^{-2\pi i/8}$.

For a graphic representation, the twiddle factors are mapped onto the unit circle. By Fig. 4.1 it can be seen that:

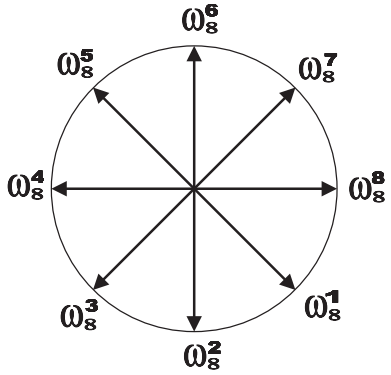


Figure 4.1: Twiddle factors of 8 bit sample sequence represented as vectors in the unit circle.

1. The twiddle factors are periodic round the unit circle.
2. The vectors are symmetric.
3. The vectors are equally spaced around the circle with spacing $\Delta\omega$.

Property 4.1 (Periodicity) *The twiddle factors map onto the unit circle and are periodic.*

$$\omega_k = k\frac{\Omega}{N}; \quad k = 0, 1, \ldots, N-1. \tag{4.10}$$

In equation (4.10) Ω is the sampling frequency 2π and N is the number of samples. The frequencies have been normalized over the unit circle, i. e., the sampling frequency is assumed to be 2π and it takes 2π radians to go round the unit circle: (i) *Once round* reaches the sampling frequency, (ii) *twice round* reproduces the results from first time round (in range 0 to 2π).

Property 4.2 (Spacing) *The vectors are equally spaced around the unit circle with spacing*

$$\Delta\omega = \frac{2\pi}{N} = \frac{\Omega}{N}.$$

$\Delta\omega$ is also called the frequency resolution of the DFT outputs, the spacing of each sample in the frequency domain.

Property 4.3 (Symmetry) *The twiddle factors are inversely symmetric about the unit circle origin.*

Example 4.3 Taken from Fig. 4.1:

$$\omega_8^1 = -\omega_8^5, \ \omega_8^2 = -\omega_8^6, \ \ldots$$

According to this observation, the first half, 0 to π, of the twiddle factors contains all the necessary information as the second half is the inverse of the first half.[2] Therefore only $N/2$ twiddle factors need to be computed.

[2]Up to half of the sampling frequency, or equivalently, up to half of the Nyquist frequency.

4.3 The Fast Fourier Transform

The DFT is one of the most important tools in modern engineering. Therefore the algorithm was continuously optimized over the years. The key improvement, which reduced computation time and costs dramatically, was to exploit the intrinsic symmetry of the DFT matrix. The breakthrough algorithm of Cooley-Tukey enabled a reduction of the number of operations to be carried out to $\text{const} \times N \log N$—the constant varying between 3 and 5 depending on the specific variant of the algorithm used.

N	N^2	$N \log_2 N$	DFT (sec)	FFT (sec)	Speed-up
4	1.60×10^{1}	8.00×10^{0}	1.60×10^{-8}	8.00×10^{-9}	2
16	2.56×10^{2}	6.40×10^{1}	2.56×10^{-7}	6.40×10^{-8}	4
64	4.10×10^{3}	3.84×10^{2}	4.07×10^{-6}	3.84×10^{-7}	11
256	6.55×10^{4}	2.05×10^{3}	6.55×10^{-5}	2.05×10^{-6}	32
1,024	1.05×10^{6}	1.02×10^{4}	1.05×10^{-3}	1.02×10^{-5}	102
4,096	1.68×10^{7}	4.92×10^{4}	1.68×10^{-2}	4.92×10^{-5}	341
16,384	2.68×10^{8}	2.29×10^{5}	2.68×10^{-1}	2.29×10^{-4}	1,170
65,536	4.29×10^{9}	1.05×10^{6}	4.30×10^{0}	1.05×10^{-3}	4,096
262,144	6.87×10^{10}	4.72×10^{6}	6.87×10^{1}	4.72×10^{-3}	14,564
1,048,576	1.10×10^{12}	2.10×10^{7}	1.10×10^{3}	2.10×10^{-2}	52,429
4,194,304	1.76×10^{13}	9.23×10^{7}	1.76×10^{4}	9.23×10^{-2}	190,650

Table 4.2: Theoretical execution times of the "classical" DFT and the FFT on a 1 GHz processor running at peak performance.

In applications where the DFT has to be performed many times (for example, in meteorology) and in applications where large transformation lengths are needed (especially in seismic applications) the speed-up achieved by using an FFT algorithm is invaluable.[3]

The key idea behind the Cooley-Tukey algorithm is to use the divide and conquer paradigm. This idea can be explained by means of the 8×8 DFT matrix

[3]For the vector length $N = 2^{22} = 4,194,304$ the execution time can be reduced from 5 hours to 100 milliseconds! Note, however, that there are factors which influence ideal execution times. These factors will be subject of a more detailed discussion in later chapters of this work. For instance, the operations of an FFT algorithm are *complex* multiplications and additions, requiring a larger number of real floating-point instructions. Then, there are dependencies between the instructions, which may inhibit the completion of one operation at every cycle. Finally, there is overhead caused by the function calls and there are memory latencies due to cache misses, which may increase ideal execution times substantially.

$$F_8 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \\ 1 & \omega^2 & \omega^4 & \omega^6 & 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega & \omega^4 & \omega^7 & \omega^2 & \omega^5 \\ 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^5 & \omega^2 & \omega^7 & \omega^4 & \omega & \omega^6 & \omega^3 \\ 1 & \omega^6 & \omega^4 & \omega^2 & 1 & \omega^6 & \omega^4 & \omega^2 \\ 1 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega \end{pmatrix},$$

where $\omega := \omega_8 = e^{-2\pi i/8}$.

Any matrix F_N, N even, can be rearranged by the perfect shuffle permutation Π_N which groups the even-indexed columns first and then the odd-indexed columns second:

$$F_N\Pi_N = (F_N(:, 0 : N-2 : 2)|F_N(:, 1 : N-1 : 2)).$$

Rearranging the columns of F_8 in this manner

$$F_8\Pi_8 = \left(\begin{array}{cccc|cccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega^2 & \omega^4 & \omega^6 & \omega & \omega^3 & \omega^5 & \omega^7 \\ 1 & \omega^4 & 1 & \omega^4 & \omega^2 & \omega^6 & \omega^2 & \omega^6 \\ 1 & \omega^6 & \omega^4 & \omega^2 & \omega^3 & \omega & \omega^7 & \omega^5 \\ \hline 1 & 1 & 1 & 1 & \omega^4 & \omega^4 & \omega^4 & \omega^4 \\ 1 & \omega^2 & \omega^4 & \omega^6 & \omega^5 & \omega^7 & \omega & \omega^3 \\ 1 & \omega^4 & 1 & \omega^4 & \omega^6 & \omega^2 & \omega^6 & \omega^2 \\ 1 & \omega^6 & \omega^4 & \omega^2 & \omega^7 & \omega^5 & \omega^3 & \omega \end{array}\right)$$

establishes a connection between F_8 and F_4. Using the fact that $\omega_8^2 = \omega_4$ is a 4th root of unity, and therefore

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega_8^2 & \omega_8^4 & \omega_8^6 \\ 1 & \omega_8^4 & 1 & \omega_8^4 \\ 1 & \omega_8^6 & \omega_8^4 & \omega_8^2 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega_4 & \omega_4^2 & \omega_4^3 \\ 1 & \omega_4^2 & 1 & \omega_4^2 \\ 1 & \omega_4^3 & \omega_4^2 & \omega_4 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix} = F_4.$$

This leads to the partitioning

$$F_8\Pi_8 = \begin{pmatrix} F_4 & \Omega_4 F_4 \\ F_4 & \omega_8^4 \Omega_4 F_4 \end{pmatrix},$$

where $\Omega_4 = \mathrm{diag}(1, \omega_8, \omega_8^2, \omega_8^3)$. Given the fact that $\omega_8^4 = -1$, the factorization

$$F_8\Pi_8 = \begin{pmatrix} \mathrm{I}_4 & \mathrm{I}_4 \\ \mathrm{I}_4 & -\mathrm{I}_4 \end{pmatrix} \begin{pmatrix} \mathrm{I}_4 & \\ & \Omega_4 \end{pmatrix} \begin{pmatrix} F_4 & \\ & F_4 \end{pmatrix}$$

is obtained. Thus,

$$F_8\Pi_8(\mathrm{I}_2 \otimes \Pi_4) = \begin{pmatrix} \mathrm{I}_4 & \mathrm{I}_4 \\ \mathrm{I}_4 & -\mathrm{I}_4 \end{pmatrix} \begin{pmatrix} \mathrm{I}_4 & \\ & \Omega_4 \end{pmatrix} \begin{pmatrix} F_4\Pi_4 & \\ & F_4\Pi_4 \end{pmatrix}.$$

Now, both of the smaller transforms can be split up again into two transforms of half the original size, namely 4. This kind of recursive splitting can be performed $\log_2 N$ times. Each of these steps involves the whole input vector, on which $O(N)$ operations are performed. Therefore the total arithmetic complexity is $O(N \log_2 N)$.

$$\begin{pmatrix} F_4\Pi_4 & \\ & F_4\Pi_4 \end{pmatrix} =$$

$$= \begin{pmatrix} \mathrm{I}_2 & \mathrm{I}_2 & & \\ \mathrm{I}_2 & -\mathrm{I}_2 & & \\ & & \mathrm{I}_2 & \mathrm{I}_2 \\ & & \mathrm{I}_2 & -\mathrm{I}_2 \end{pmatrix} \begin{pmatrix} \mathrm{I}_2 & & & \\ & \Omega_2 & & \\ & & \mathrm{I}_2 & \\ & & & \Omega_2 \end{pmatrix} \begin{pmatrix} F_2 & & & \\ & F_2 & & \\ & & F_2 & \\ & & & F_2 \end{pmatrix},$$

where $\Omega_2 = \mathrm{diag}(1, \omega_4)$. Thus, a complete factorization of F_8 is obtained.

This matrix representation points out, that the Cooley-Tukey idea can be programmed easily using a recursive approach, which in pseudo-code, would have the following form (if the permutation of the input vector is already done):

Algorithm 4.1 (Recursive FFT)

if *length of input vector equals* 1 **then return**
$u :=$ **fft** *of upper part of input*
$l :=$ **fft** *of lower part of input*
$l := l \times$weight_vector
overwrite upper part of output with $u + l$
overwrite lower part of output with $u - l$
return output

Unfortunately, such recursive algorithms have some unfavorable characteristics. For example, each recursive call needs additional memory. Therefore recursive FFT algorithms have to be implemented carefully for real applications. Anyhow, they are important tools to understand the Cooley-Tukey concept.

4.4 Cooley-Tukey Radix-2 Factorization

Using

$$\Omega_{N/2} := \mathrm{diag}(1, \omega_N, \ldots, \omega_N^{N/2-1}), \qquad N \text{ even},$$

the four symmetry conditions $(k, j = 0 : N/2)$

$$\begin{array}{llll}
[F_N \Pi_N]_{k,j} & = \omega_N^{k(2j)} & = \omega_{N/2}^{kj} & = [F_{N/2}]_{k,j} \\
[F_N \Pi_N]_{k+N/2,j} & = \omega_N^{(k+N/2)(2j)} & = \omega_{N/2}^{(k+N/2)j} & = [F_{N/2}]_{k,j} \\
[F_N \Pi_N]_{k,j+N/2} & = \omega_N^{k(2j+1)} & = \omega_N^k \omega_{N/2}^{kj} & = [\Omega_{N/2} F_{N/2}]_{k,j} \\
[F_N \Pi_N]_{k+N/2,j+N/2} & = \omega_N^{(k+N/2)(2j+1)} & = -\omega_N^{k(2j+1)} & = [-\Omega_{N/2} F_{N/2}]_{k,j}
\end{array}$$

imply the *radix-2 splitting*

$$F_N \Pi_N = \begin{pmatrix} F_{N/2} & \Omega_{N/2} F_{N/2} \\ F_{N/2} & -\Omega_{N/2} F_{N/2} \end{pmatrix}. \tag{4.11}$$

These relations can be easily established with simple computations due to the fact that $\omega_N^2 = \omega_{N/2}$ and $\omega_N^{N/2} = -1$.

The term *radix-2 splitting* indicates that the relation (4.11) establishes a connection between the full-sized DFT matrix F_N and the half-sized DFT matrix $F_{N/2}$. Recursive application of this splitting process is the heart of all radix-2 FFT algorithms. More generally, if p divides N, it is possible to relate F_N to $F_{N/p}$.

Thus, F_N can be factorized as

$$F_N = \begin{pmatrix} \mathrm{I}_{N/2} & \mathrm{I}_{N/2} \\ \mathrm{I}_{N/2} & -\mathrm{I}_{N/2} \end{pmatrix} \begin{pmatrix} \mathrm{I}_{N/2} & \\ & \Omega_{N/2} \end{pmatrix} \begin{pmatrix} F_{N/2} & \\ & F_{N/2} \end{pmatrix} \mathrm{L}_2^N .$$

This factorization can be expressed in terms of Kronecker products.

Theorem 4.2 (Cooley-Tukey Radix-2 Splitting) *For $N \geq 2$, N even*

$$F_N = (F_2 \otimes \mathrm{I}_{N/2}) T_N (\mathrm{I}_2 \otimes F_{N/2})\, \mathrm{L}_2^N,$$

$$T_N = (\mathrm{I}_{N/2} \oplus \Omega_{N/2}),$$

where $\oplus$ denotes the direct matrix sum operator.

Example 4.4 (FFT Factorization) The 4-point FFT F_4 is factorized into its product of 4 sparse matrices:

$$\begin{aligned}
F_4 &= (F_2 \otimes \mathrm{I}_2) T_2^4 (\mathrm{I}_2 \otimes F_2)\, \mathrm{L}_2^4 \\
&= \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & i \end{pmatrix} \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.
\end{aligned}$$

Using this formulation, the Cooley-Tukey theorem can be interpreted as a *rewrite rule*. It says that the Fourier transform matrix can be replaced by the product of four matrices. The FFT is derived by recursively applying this rewrite rule.

Example 4.5 (Applying the Cooley-Tukey Theorem) The 8-point FFT F_8 is obtained by applying the Cooley-Tukey theorem to F_8 and then applying the Cooley-Tukey theorem to F_4:

$$\begin{aligned} F_8 &= (F_2 \otimes \mathrm{I}_4) T_4^8 (\mathrm{I}_2 \otimes F_4)\, \mathrm{L}_2^8 \\ &= (F_2 \otimes \mathrm{I}_4) T_4^8 (\mathrm{I}_2 \otimes ((F_2 \otimes \mathrm{I}_2) T_2^4 (\mathrm{I}_2 \otimes F_2)\, \mathrm{L}_2^4))\, \mathrm{L}_2^8 . \end{aligned}$$

4.5 General CT Factorizations – Radix-p Kernels

The splitting idea of Theorem 4.2 is not restricted to dividing $N = 2^2$ and $N = 2^4$, it can be used for all $N = N_1 \times N_2$.

Theorem 4.3 (Fundamental Radix-p Factorization) (Johnson et al. [39]) *For* $N = pk \geq 2$

$$F_N = (F_p \otimes \mathrm{I}_k) T_k^{pk} (\mathrm{I}_p \otimes F_k)\, \mathrm{L}_p^{pk},$$

with

$$T_k^{pk} = \mathrm{diag}(\mathrm{I}_k, \Omega_{p,k}, \ldots, \Omega_{p,k}^{p-1}),$$

where

$$\Omega_{p,k} := \mathrm{diag}(1, \omega_N, \ldots, \omega_N^{k-1}).$$

The mathematical opportunity to split the DFT matrix into four factor matrices illustrates that also the DFT's execution can be divided into four subproblems:

1. a stride permutation,
2. p DFTs of length q,
3. the multiplication by twiddle-factors, and finally
4. q DFTs of length p.

Hence this factorization can be performed by applying two sets of smaller DFTs and two further operations, a stride permutation and a scaling by twiddle-factors, that are computationally less expensive. It is a remarkable intrinsic phenomenon that the DFT can essentially be reduced to a set of identical, just smaller problems.

To decompose the DFT matrices in the split representation, repeated application of the splitting theorem may be employed, obtaining a further factorization of

the DFT matrix and, hence, superseding the execution of a transform to even smaller DFTs, as will be presented now.

For $N = p^n$, repeated application of Theorem 4.3 leads to the following factorization of the DFT matrix F_{p^n}.

Theorem 4.4 (Fundamental Single-Radix Factorization)

$$F_{p^n} = \Big[\prod_{i=1}^{n}(\mathrm{I}_{p^{i-1}} \otimes F_p \otimes \mathrm{I}_{p^{n-i}})(\mathrm{I}_{p^{i-1}} \otimes T^{p^{n-i+1}}_{p^{n-i}})\Big] R_{p^n} \tag{4.12}$$

This factorization makes it possible to split a p^n-point DFT into n DFTs of size p. The number p is called the *radix* of the FFT algorithm. The permutation matrix R_{p^n} is the index reversal matrix responsible for the permutation of the input data sequence (see Van Loan [63]).

By using (2.4), i. e., the mixed-product property of the Kronecker product, the expression

$$(\mathrm{I}_{p^{i-1}} \otimes F_p \otimes \mathrm{I}_{p^{n-i}})(\mathrm{I}_{p^{i-1}} \otimes T^{p^{n-i+1}}_{p^{n-i}})$$

can be written as

$$\mathrm{I}_{p^{i-1}} \otimes ((F_p \otimes \mathrm{I}_{p^{n-i}})\mathrm{diag}(\mathrm{I}_{p^{n-i}}, \Omega_{p,p^{n-i}}, \ldots, \Omega^{p-1}_{p,p^{n-i}})).$$

The matrix

$$B_{p,p^{n-i+1}} := (F_p \otimes \mathrm{I}_{p^{n-i}})\mathrm{diag}(\mathrm{I}_{p^{n-i}}, \Omega_{p,p^{n-i}}, \ldots, \Omega^{p-1}_{p,p^{n-i}})$$

is said to be a *radix-p butterfly matrix*.

Using this matrix, (4.12) becomes

$$F_{p^n} = \Big[\prod_{i=1}^{n}(\mathrm{I}_{p^{i-1}} \otimes B_{p,p^{n-i+1}})\Big] R_{p^n}. \tag{4.13}$$

If $x \in \mathbb{C}^N$ and $N = p^n$, the FFT computation $x := F_{p^n}x$ can be implemented as

Algorithm 4.2 (Radix-p FFT)

$x := R_{p^n}x$
do $i = 1 : n$
  $L := p^i$
  $r := N/L$
  do $k = 0 : r - 1$
    $x(kL : (k+1)L - 1) := B_{p,L}x(kL : (k+1)L - 1)$
  end do
end do

The computationally most intensive part of any radix-p FFT algorithm is

$$x := B_{p,L}x,$$

i. e., the butterfly update which occurs in the innermost loop. Thus, the arithmetic complexity of any radix-p FFT algorithm depends primarily on the design and implementation of the butterfly kernel.

4.6 DIT and the DIF Decomposition

The abbreviations DIT and DIF stand for *decimation-in-time* and *decimation-in-frequency*. They denote two distinct ways of splitting DFT matrices. Thus, different factorizations are obtained leading to different classes of algorithms.

A signal can be viewed from two different standpoints: (i) The frequency domain, and (ii) the time domain.

Decimation is the process of breaking down something into its constituent parts.

4.6.1 Decimation-In-Time (DIT) Splitting

The DIT splitting is essentially the fundamental splitting of Theorem 4.3, the one derived by Cooley and Tukey. The fact that a time sampled data vector is first divided into parts to which, secondly, FFTs of appropriate lengths are applied gave rise to this method's name.

4.6.2 Decimation-In-Frequency (DIF) Splitting

The DIF splitting was described independently by Gentleman and Sande [31] and by Cooley and Stockham [9]. As long as FFTs were written in sum notation, it was only possible to state that DIT and DIF splitting involves similar operations in a different order. In Kronecker product notation, however, their relation is obvious: The DIF splitting corresponds to the transposed DIT splitting. Since the DFT matrix is symmetric, transposition leaves the result unchanged whereas its factorization is altered in its sequence of appliance.

According to Properties 3.3 (multiplication) and 3.4 (commutation), transposition of Theorem 4.3 results in

Theorem 4.5 (Fundamental DIF Radix-p Splitting) *For $N = pk \geq 2$*

$$F_N = \mathrm{L}^{pk}_k(\mathrm{I}_p \otimes F_k)T^{pk}_k(F_p \otimes \mathrm{I}_k),$$

with T_k^{pk} defined as in Theorem 4.3.

In this case, first the DFT of length k, the multiplication by the twiddle-factors and then the DFT of length p are employed and finally the provisional result is stride permuted and stored to the output vector. The output vector's index, sometimes called the "frequency" which may be involved decisively in the last stride permutation and save operation, coined the name for this decomposition methodology.

In the same way as with the DIT factorization, Theorem 4.5 can be applied recursively to obtain the following factorization of the DFT matrix F_{p^n}.

Theorem 4.6 (Single-Radix DIF Factorization)

$$F_{p^n} = R_{p^n}\Big[\prod_{j=1}^{n}(\mathrm{I}_{p^{n-j}} \otimes T_{p^{j-1}}^{p^j})(\mathrm{I}_{p^{n-j}} \otimes F_p \otimes \mathrm{I}_{p^{j-1}})\Big] \tag{4.14}$$

Finally, another link between the DIT and the DIF splitting can be established. The radix-p DIT splitting of length $N = pk$ is assumed

$$F_N = (F_p \otimes \mathrm{I}_k)T_k^N(\mathrm{I}_p \otimes F_k)\,\mathrm{L}_p^N\,.$$

Applying Property 3.4 (commutation) to both of the terms with Kronecker products leads to

$$F_N = \mathrm{L}_p^N(\mathrm{I}_k \otimes F_p)\,\mathrm{L}_k^N\,T_k^N\,\mathrm{L}_p^N(F_k \otimes \mathrm{I}_p)\,\mathrm{L}_k^N\,\mathrm{L}_p^N\,.$$

Using Properties 3.4 and 3.3 this can be written as

$$F_N = \mathrm{L}_p^N(\mathrm{I}_k \otimes F_p)T_p^N(F_k \otimes \mathrm{I}_p),$$

which is equal to the radix-k DIF splitting. Hence, by this relation, a general structural resemblance between the radix-r DIT splitting and the radix-N/r DIF splitting is indicated.

This directly leads to the next considerations, because it is also possible to apply Property 3.4 (commutation) to only one of the terms with Kronecker products, which leads to completely new types of recursive algorithmic steps, namely 4- and 6-step decomposition (Franchetti, Lorenz, Ueberhuber [24]).

4.7 Multidimensional Fast Fourier Transforms

One-dimensional DFTs operate on one-dimensional datasets. Multidimensional DFTs extend this domain to multidimensional arrays. Such arrays will be denoted

$$f_{k_1,\ldots,k_d}, \quad i = 1,\ldots,d : k_i = 0,\ldots,N_i - 1.$$

Throughout this section d denotes the number of dimensions of the data-array to be transformed and $N = N_1 N_2 \cdots N_d$ denotes the total number of data points. The multidimensional DFT of such an array is defined by

$$\begin{aligned} F_{n_1,\ldots,n_d} : &= \sum_{i_1=0}^{N_1-1} \sum_{i_2=0}^{N_2-1} \cdots \sum_{i_d=0}^{N_d-1} \omega_{N_1 N_2 \cdots N_d}^{n_1 i_1 + n_2 i_2 + \cdots + n_d i_d} f_{i_1,i_2,\ldots,i_d} \\ &= \sum_{i_1=0}^{N_1-1} \sum_{i_2=0}^{N_2-1} \cdots \sum_{i_d=0}^{N_d-1} \omega_{N_1}^{n_1 i_1} \omega_{N_2}^{n_2 i_2} \cdots \omega_{N_d}^{n_d i_d} f_{i_1,i_2,\ldots,i_d} \\ &= \sum_{i_1=0}^{N_1-1} \omega_{N_1}^{n_1 i_1} \sum_{i_2=0}^{N_2-1} \omega_{N_2}^{n_2 i_2} \cdots \sum_{i_d=0}^{N_d-1} \omega_{N_d}^{n_d i_d} f_{i_1,i_2,\ldots,i_d}. \end{aligned} \tag{4.15}$$

(4.15) shows that the multidimensional DFT consists of scalar DFTs calculated along each of the d dimensions of the data array. This representation can be split into iterative steps calculating each component's one-dimensional DFTs.

$$F_{n_1,\ldots,n_d}^{[1]} = \sum_{k=0}^{N_1-1} \omega_{N_1}^{n_1 i_1} f_{k,i_2,\ldots,i_d}, \quad i = 2\ldots d : n_i = 0,\ldots,N_i - 1 \tag{4.16}$$

$$F_{n_1,\ldots,n_d}^{[2]} = \sum_{k=0}^{N_2-1} \omega_{N_2}^{n_2 i_2} F_{i_1,k,\ldots,i_d}^{[1]}, \quad i = 1,3\ldots d : n_i = 0,\ldots,N_i - 1 \tag{4.17}$$

$$\vdots$$

$$F_{n_1,\ldots,n_d}^{[d]} = \sum_{k=0}^{N_d-1} \omega_{N_d}^{n_d i_d} F_{i_1,i_2,\ldots,k}^{[d-1]}, \quad i = 1\ldots d-1 : n_i = 0,\ldots,N_i - 1 \tag{4.18}$$

$$F_{n_1,\ldots,n_d} = F_{n_1,\ldots,n_d}^{[d]}.$$

The two common ways for a computer program to store multidimensional arrays in its memory are (i) row-major order (used in C and most other programming languages), and (ii) column-major order (used in Fortran). This work will focus on the row-major order storage convention. The index-transformation rules between a multidimensional array f and the matching one-dimensional array f' are

$$\begin{aligned} f_{i_1,i_2,\ldots,i_d} &= f'_{i_1 N_2 \cdots N_d + i_2 N_3 \cdots N_d + \cdots + i_{d-1} N_d + i_d} \\ \Rightarrow f'_i &= f_{\lfloor i/(N_2 N_3 \cdots N_d) \rfloor \bmod(N_1), \lfloor i/(N_3 N_4 \cdots N_d) \rfloor \bmod(N_2), \ldots, \lfloor i/N_d \rfloor \bmod(N_{d-1}), i \bmod(N_d)}. \end{aligned}$$

According to these rules the index-transformation function s is defined as follows

$$s(i_1, i_2, \ldots, i_d) := i_1 N_2 \cdots N_d + i_2 N_3 \cdots N_d + \cdots + i_{d-1} N_d + i_d. \quad (4.19)$$

This implies the following distances for two adjacent elements

$$s(i_1, \ldots, i_{m-1}, k+1, i_{m+1}, \ldots, i_d) - s(i_1, \ldots, i_{m-1}, k, i_{m+1}, \ldots, i_d) = \\ = N_{m+1} N_{m+2} \cdots N_d$$

and especially

$$s(k+1, i_2, \ldots i_d) - s(k, i_2, \ldots i_d) = N_2 N_3 \cdots N_d, \qquad k = 0 \ldots N_1 - 2$$
$$s(i_1, \ldots i_{d-1}, k+1) - s(i_1, \ldots i_{d-1}, k) = 1, \qquad k = 0 \ldots N_d - 2.$$

Examining formulas (4.16) to (4.18) under the aspect that a scalar array is to be transformed shows that effectively (4.16) are N/N_1 scalar DFTs of length N_1 with stride N/N_1, (4.17) are N_1 blocks of N/N_2 scalar DFTs of length N_2 with stride $N/(N_1 \cdot N_2)$, and (4.18) are N/N_d blocks of N/N_d scalar DFTs of length N_d with stride 1.

Such formulas can be easier dealt with by using the Kronecker product notation introduced in Chapter 2. Accordingly, (4.16) to (4.18) translate to

$$(F_{n_1} \otimes \mathrm{I}_{n_2} \otimes \cdots \otimes \mathrm{I}_{n_d})(\mathrm{I}_{n_1} \otimes F_{n_2} \otimes \cdots \otimes \mathrm{I}_{n_d}) \cdots (\mathrm{I}_{n_1} \otimes \mathrm{I}_{n_2} \otimes \cdots \otimes F_{n_d}).$$

Applying Property 2.4, i. e., the mixed product property, finally yields the following compact representation of multidimensional DFTs.

Property 4.4 (Multidimensional Fast Fourier Transform)

$$F_{n_1, \ldots, n_d} = F_{n_1} \otimes F_{n_2} \otimes \cdots \otimes F_{n_d} \quad (4.20)$$

4.8 Parallel Fast Fourier Transforms

When FFTs are to be computed on huge datasets it might turn out that one single processor is not able to do this calculation fast enough, or the available memory is not capable of storing the whole dataset. In such cases the computation has to be *parallelized*, i. e., multiple processors are sharing the work. The difficulty arising in parallel computation is that each processor is only able to access its own memory. If data from other processors' storage is required, it has to be sent over some network. Depending on the network's latency and bandwith this causes a delay during which no computation is possible.

In this section a very simple data distribution over the processors will be assumed. The most intuitive way to distribute a dataset over p processors is the *slab decomposition*.

Definition 4.2 (Slab Decomposition) *The data vector* $D = [d_0, \ldots, d_{n-1}]$ *of length* n *is distributed over* p *processors* $P_0, \ldots, P_{p-1}$ *such that processor* i *holds the data*

$$P_i \ni [d_{i(n/p)}, \ldots, d_{(i+1)(n/p)-1}].$$

This means that the data vector D is uniformly divided in p parts and processor P_i holds the i-th of these parts. Therefore it is required that p divides n.

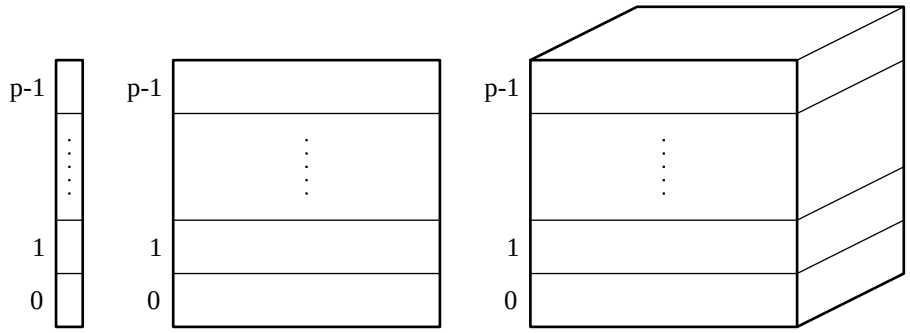


Figure 4.2: Slab decomposed one-, two-, and three-dimensional data arrays on p processors.

The challenge when dealing with parallel FFTs is to rewrite the formulas so that they are compatible with parallel computation. Therefore the FFT has to be split into Kronecker parallel computation stages (Section 2.4.1)

$$\mathrm{I}_p \otimes A,$$

and communication stages (Section 3.2)

$$C \otimes \mathrm{I}_b\,.$$

When parallelizing one-dimensional FFTs, parallel twiddle-factor matrices arise. It is neither possible nor required to express them as parallel computation stages. As the implementation of such an element-wise multiplication is trivial it suffices at this point to define a symbol for the parallel twiddle-factor matrix as follows.

Definition 4.3 (Parallel Twiddle-Factor Matrix) *A parallel twiddle-factor matrix* $T^{mn}_{m,p}$ *represents the serial twiddle-factor matrix* T^{mn}_{m} *spread over* p *processors.*

4.8.1 Breaking Down Parallel FFTs

This section illustrates how an FFT can be broken down into parallel computation parts and communication parts. This process can be carried out automatically by *teaching* a formula manipulation program the required rules.

Example 4.6 (Parallel 3D FFT Breakdown) A 3D FFT F_{n_1,n_2,n_3} transformation starts with the parallel FFT formula introduced in Property 4.4. The FFT is to be parallelized on p processors. It is required that $p|n_1$ and $p|n_2n_3$.

$$\begin{aligned} F_{n_1,n_2,n_3} &= (F_{n_1} \otimes F_{n_2} \otimes F_{n_3}) = \\ &= (F_{n_1} \otimes \mathrm{I}_{n_2} \otimes \mathrm{I}_{n_3})(\mathrm{I}_{n_1} \otimes F_{n_2} \otimes \mathrm{I}_{n_3})(\mathrm{I}_{n_1} \otimes \mathrm{I}_{n_2} \otimes F_{n_3}) = \\ &= (F_{n_1} \otimes \mathrm{I}_{n_2n_3}) \\ &\quad (\mathrm{I}_p \otimes \mathrm{I}_{n_1/p} \otimes F_{n_2} \otimes \mathrm{I}_{n_3}) \\ &\quad (\mathrm{I}_p \otimes \mathrm{I}_{n_1n_2/p} \otimes F_{n_3}). \end{aligned} \tag{4.21}$$

The last expression shows that the last two factors are already parallel factors while the first factor $(F_{n_1} \otimes \mathrm{I}_{n_2n_3})$ has yet to be transformed. Applying Property 3.4 to this first term yields

$$\begin{aligned} (F_{n_1} \otimes \mathrm{I}_{n_2} \otimes \mathrm{I}_{n_3}) &= F_{n_1} \otimes \mathrm{I}_p \otimes \mathrm{I}_{n_2n_3/p} = \\ &= \left(\mathrm{L}^{pn_1}_{n_1}(\mathrm{I}_p \otimes F_{n_1})\,\mathrm{L}^{pn_1}_{p}\right) \otimes \mathrm{I}_{n_2n_3/p} = \\ &= (\mathrm{L}^{pn_1}_{n_1} \otimes \mathrm{I}_{n_2n_3/p})(\mathrm{I}_p \otimes F_{n_1} \otimes \mathrm{I}_{n_2n_3/p})(\mathrm{L}^{pn_1}_{p} \otimes \mathrm{I}_{n_2n_3/p}). \end{aligned}$$

Thus (4.21) can be further expanded to

$$\begin{aligned} (4.21) &= (\mathrm{L}^{pn_1}_{n_1} \otimes \mathrm{I}_{n_2n_3/p}) \\ &\quad (\mathrm{I}_p \otimes F_{n_1} \otimes \mathrm{I}_{n_2n_3/p}) \\ &\quad (\mathrm{L}^{pn_1}_{p} \otimes \mathrm{I}_{n_2n_3/p}) \\ &\quad (\mathrm{I}_p \otimes \mathrm{I}_{n_1/p} \otimes F_{n_2} \otimes \mathrm{I}_{n_3}) \\ &\quad (\mathrm{I}_p \otimes \mathrm{I}_{n_1n_2/p} \otimes F_{n_3}). \end{aligned} \tag{4.22}$$

At this stage all computation factors are parallelized. The only work left is splitting the stride permutations $\mathrm{L}^{pn_1}_{n_1}$ and $\mathrm{L}^{pn_1}_{p}$ to local permutation and global communication steps. The two special cases of Lemma 3.2 result in

$$\begin{aligned} (\mathrm{L}^{pn_1}_{n_1} \otimes \mathrm{I}_{n_2n_3/p}) &= \left((\mathrm{I}_p \otimes \mathrm{L}^{n_1}_{n_1/p})(\mathrm{L}^{p^2}_{p} \otimes \mathrm{I}_{n_1/p})\right) \otimes (\mathrm{I}_{n_2n_3/p}) = \\ &= (\mathrm{I}_p \otimes \mathrm{L}^{n_1}_{n_1/p} \otimes \mathrm{I}_{n_2n_3/p})(\mathrm{L}^{p^2}_{p} \otimes \mathrm{I}_{n_1n_2n_3/p^2}) \end{aligned}$$

and

$$\begin{aligned} (\mathrm{L}^{pn_1}_{p} \otimes \mathrm{I}_{n_2n_3/p}) &= \left((\mathrm{L}^{p^2}_{p} \otimes \mathrm{I}_{n_1/p})(\mathrm{I}_p \otimes \mathrm{L}^{n_1}_{p})\right) \otimes (\mathrm{I}_{n_2n_3/p}) = \\ &= (\mathrm{L}^{p^2}_{p} \otimes \mathrm{I}_{n_1n_2n_3/p^2})(\mathrm{I}_p \otimes \mathrm{L}^{n_1}_{p} \otimes \mathrm{I}_{n_2n_3/p}). \end{aligned}$$

Thus (4.22) be rewritten to the fully parallelized formula

$$\begin{aligned} (4.22) &= (\mathrm{I}_p \otimes \mathrm{L}^{n_1}_{n_1/p} \otimes \mathrm{I}_{n_2n_3/p})(\mathrm{L}^{\mathbf{p^2}}_{\mathbf{p}} \otimes \mathrm{I}_{\mathbf{n_1n_2n_3/p^2}})(\mathrm{I}_p \otimes F_{n_1} \otimes \mathrm{I}_{n_2n_3/p})(\mathrm{L}^{\mathbf{p^2}}_{\mathbf{p}} \otimes \mathrm{I}_{\mathbf{n_1n_2n_3/p^2}}) \\ &\quad (\mathrm{I}_p \otimes \mathrm{L}^{n_1}_{p} \otimes \mathrm{I}_{n_2n_3/p})(\mathrm{I}_p \otimes \mathrm{I}_{n_1/p} \otimes F_{n_2} \otimes \mathrm{I}_{n_3})(\mathrm{I}_p \otimes \mathrm{I}_{n_1n_2/p} \otimes F_{n_3}). \end{aligned}$$

This formula includes parallel computation blocks and the two emphasized parallel matrix transpositions. When read in the order of execution—bottom to top—the first step is to compute the scalar FFTs along the third and second component, then the matrix is transposed to compute FFTs along the third, beforehand distributed, dimension. Finally, the matrix is to be transposed back to re-obtain the original data layout.

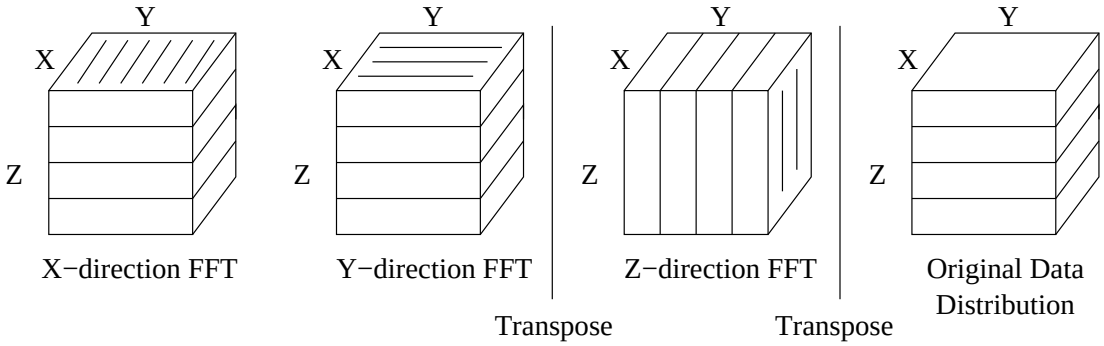


Figure 4.3: Data distribution during the three steps.

As illustrated in Fig. 4.3 this procedure exactly represents the row-column algorithm for three-dimensional FFTs [14]. The requirements $p|n_1$ and $p|n_2n_3$ are necessary to *generate* this algorithm because it must be possible to divide the data array into slabs along at least two different dimensions—once for the original data layout and once during the transformed computation stage.

Example 4.7 (Parallel 1D FFT Breakdown) The 1D FFT $F_{n_1n_2}$ is to be parallelized using the Cooley-Tukey radix-p factorization introduced in Theorem 4.3. Therefore it is required that $p|n_1$ and $p|n_2$.

$$
\begin{aligned}
F_{n_1n_2} &= (F_{n_1} \otimes \mathrm{I}_{n_2}) T^{n_1n_2}_{n_2,p} (\mathrm{I}_{n_1} \otimes F_{n_2}) \, \mathrm{L}^{n_1n_2}_{n_1} = \\
&= (\mathrm{L}^{pn_1}_{n_1} \otimes \mathrm{I}_{n_2/p})(\mathrm{I}_p \otimes F_{n_1} \otimes \mathrm{I}_{n_2/p})(\mathrm{L}^{pn_1}_{p} \otimes \mathrm{I}_{n_2/p}) \\
&\quad T^{n_1n_2}_{n_2,p} (\mathrm{I}_p \otimes \mathrm{I}_{n_1/p} \otimes F_{n_2}) \, \mathrm{L}^{n_1n_2}_{n_1} = \\
&= (\mathrm{I}_p \otimes \mathrm{L}^{n_1}_{n_1/p} \otimes \mathrm{I}_{n_2/p})(\mathrm{L}^{\mathbf{p^2}}_{\mathbf{p}} \otimes \mathrm{I}_{\mathbf{n_1n_2/p^2}})(\mathrm{I}_p \otimes F_{n_1} \otimes \mathrm{I}_{n_2/p}) \\
&\quad (\mathrm{L}^{\mathbf{p^2}}_{\mathbf{p}} \otimes \mathrm{I}_{\mathbf{n_1n_2/p^2}})(\mathrm{I}_p \otimes \mathrm{L}^{n_1}_{p} \otimes \mathrm{I}_{n_2/p}) T^{n_1n_2}_{n_2,p} (\mathrm{I}_p \otimes \mathrm{I}_{n_1/p} \otimes F_{n_2}) \cdot \\
&\quad (\mathrm{I}_p \otimes \mathrm{L}^{n_1n_2/p}_{n_1/p})(\mathrm{L}^{\mathbf{p^2}}_{\mathbf{p}} \otimes \mathrm{I}_{\mathbf{n_1n_2/p^2}})(\mathrm{I}_p \otimes \mathrm{L}^{n_2}_{p} \otimes \mathrm{I}_{n_1/p})
\end{aligned}
$$

This algorithm requires three parallel matrix transpositions. It is not possible to perform a single distributed one-dimensional FFT with one radix-p step with less than three global transposes.

4.9 Non-FFT Signal Transforms

Even though the FFT is the most important linear signal transform there is are many other similar algorithms of great practical importance.

These algorithms can partially be handled with generic rules, like the ones shown above, but require additional rules to *describe* the transforms in Kronecker product notation. Some signal transforms, which will be dealt with in the scope of this work, will be introduced in the following sections.

4.9.1 Arbitrary Data Layouts – Conjugated FFTs

Parallel FFT algorithms require a relatively large amount of communication compared to the amount of computation. To calculate a parallel multidimensional FFT effectively, it is mandatory that all data points of—at least—one dimension are resident on the memory of one processor.

However, most FFTs are just computational stages used in large applications. These applications have different demands on the data layout. For instance, the fast multipole method (FMM) prefers a *volumetric decomposition* because of its demands to locality [33]. The developer of a general FFT routine cannot know which data distribution is suitable for the remaining parts of some application.

Most of today's FFT libraries force the data to be distributed in slabs, as this is beneficial for the FFT computation. An exception is the volumetric FFT [16], which requires a volumetric distribution, as it is adapted to certain applications that provide the data in this layout. For an application developer, who wants to include an FFT into his program, it would be optimal if the FFT would adapt to the application's data layout *and* yield optimal performance.

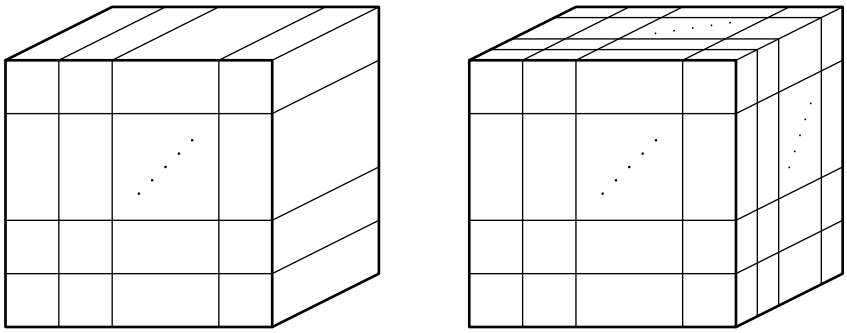


Figure 4.4: Three-dimensional data array in rod- and volumetric decomposition.

In case the user's application stores the data in a layout that is not compatible to the FFT library of his choice, the application developer has to manually redistribute the data to a distribution supported by the given FFT routine. Coding such a redistribution manually is not trivial and it is even less trivial to implement it effectively.

Due to the conflicting issues of compatibility and performance, application developers often shy from using high performance FFT libraries. They rather decide to create their own tailor-made FFT routines to exactly fit the application's general framework.

This custom entails another problem—namely the FFT routine might be well optimized, but the manual data reordering steps may have a negative impact on

the overall application's run time. This overhead is indirectly caused by the FFT but does not show up in any of the FFT library benchmarks. Furthermore, as the reordering code is not part of the FFT library, its run time cannot be included in the FFT optimization process.

The first step towards a solution of this problem is to formally define arbitrary data distributions.

Definition 4.4 (Data Distribution) *The data vector $D = [d_0, \ldots, d_{n-1}]$ of length n is distributed according to decomposition A if the permutation A satisfies*

$$D = AD_S$$

where D_S is a data vector of length n distributed in slabs (Definition 4.2).

Example 4.8 (Arbitrary Data Distribution) The data vector D represents a 4×4 matrix M distributed to 4 processors along every dimension.

$$D = \left(\begin{array}{c} a_0 \\ a_1 \\ a_4 \\ a_5 \\ \hline a_2 \\ a_3 \\ a_6 \\ a_7 \\ \hline a_8 \\ a_9 \\ a_{12} \\ a_{13} \\ \hline a_{10} \\ a_{11} \\ a_{14} \\ a_{15} \end{array}\right) \widehat{=} \left[\begin{array}{cc|cc} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \\ \hline a_8 & a_9 & a_{10} & a_{11} \\ a_{12} & a_{13} & a_{14} & a_{15} \end{array}\right] \qquad D_S = \left(\begin{array}{c} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \hline a_4 \\ a_5 \\ a_6 \\ a_7 \\ \hline a_8 \\ a_9 \\ a_{10} \\ a_{11} \\ \hline a_{12} \\ a_{13} \\ a_{14} \\ a_{15} \end{array}\right) \widehat{=} \left[\begin{array}{cccc} a_0 & a_1 & a_2 & a_3 \\ \hline a_4 & a_5 & a_6 & a_7 \\ \hline a_8 & a_9 & a_{10} & a_{11} \\ \hline a_{12} & a_{13} & a_{14} & a_{15} \end{array}\right]$$

$$\Longrightarrow D = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 \end{bmatrix} \cdot D_S = (\mathrm{I}_2 \otimes \mathrm{L}_2^4 \otimes \mathrm{I}_2) D_S$$

To provide *black-box FFT* routines for arbitrary data distributions it is necessary to inform the FFT generator about the application's data layout that defines the data distribution before and after the FFT computation. These data distributions have no restrictive influence on the methods that are used to calculate the FFT.

The following definitions show how a conjugated FFT can be interpreted as an FFT of a vector having arbitrary data distribution A.

Definition 4.5 (Conjugated FFT) *The $FFT_{n_1,\ldots,n_d}$ conjugated with the permutation $P \in \{0,1\}^{(n_1\cdots n_d)\times(n_1\cdots n_d)}$ is defined by*

$$FFT^{P}_{n_1,\ldots,n_d} := P \cdot FFT_{n_1,\ldots,n_d} \cdot P^{-1}$$

Definition 4.6 (FFT on Arbitrarilly Distributed Data Arrays) *A $FFT_{n_1,\ldots,n_d}$ to the data-array D, which is stored according to distribution A, can be calculated by using the conjugated FFT*

$$FFT^{A}_{n_1,\ldots,n_d}.$$

These definitions show that formally the data array is redistributed to slab-decomposition, then the FFT is computed, and finally the data array is transformed back to the original data layout. This exactly represents the steps which would be executed if an application developer would implement the data reordering steps, prior and posterior to the FFT.

Even without any further optimization this method has two advantages over the manual redistribution of the data-array:

- As the code for the data reordering steps is generated together with the FFT code, its run time can be included in performance measurements and thus can be included in performance optimization.
- No additional hand-coding is required for using the FFT routine.

This method provides the opportunity to optimize both the FFT and the data redistribution *together*. This often results in a significant performance increase as some of the data redistribution steps may turn out to be redundant.

Example 4.9 (Conjugated FFT) This example deals with an $FFT_{4,4,4}$ to be applied to a data vector which is distributed over 4 processors with distribution L_4^{16}. This represents a slab decomposition along the second dimension.

$$\begin{aligned}
FFT^{\mathrm{L}_4^{16}\otimes \mathrm{I}_4}_{4,4,4} &= (\mathrm{L}_4^{16}\otimes \mathrm{I}_4) FFT_{4,4,4} (\mathrm{L}_4^{16}\otimes \mathrm{I}_4) \\
&= (\mathrm{L}_{\mathbf{4}}^{\mathbf{16}}\otimes \mathrm{I}_{\mathbf{4}})(\mathrm{L}_{\mathbf{4}}^{\mathbf{16}}\otimes \mathrm{I}_{\mathbf{4}})(\mathrm{I}_4\otimes F_4\otimes \mathrm{I}_4)(\mathrm{L}_{\mathbf{4}}^{\mathbf{16}}\otimes \mathrm{I}_{\mathbf{4}}) \\
&\quad (\mathrm{I}_4\otimes F_{n_2}\otimes \mathrm{I}_4)(\mathrm{I}_4\otimes \mathrm{I}_4\otimes F_{n_3})(\mathrm{L}_{\mathbf{4}}^{\mathbf{16}}\otimes \mathrm{I}_{\mathbf{4}}) = \qquad (4.23) \\
&= (\mathrm{I}_4\otimes F_4\otimes \mathrm{I}_4)(\mathrm{L}_{\mathbf{4}}^{\mathbf{16}}\otimes \mathrm{I}_{\mathbf{4}})(\mathrm{I}_4\otimes F_{n_2}\otimes \mathrm{I}_4)(\mathrm{I}_4\otimes \mathrm{I}_4\otimes F_{n_3})(\mathrm{L}_{\mathbf{4}}^{\mathbf{16}}\otimes \mathrm{I}_{\mathbf{4}}). \qquad (4.24)
\end{aligned}$$

Formula (4.23) is a correct, yet unoptimized, algorithm. It is comparable with calculations that would be carried out if the application developer reorganized the data to slab distribution manually before and after calling an FFT routine. There are two back-to-back parallel matrix transpositions executed after the last communication step. One of them would be implemented by the application developer, one by the FFT generator. As these are self-inverse they are redundant and can both be removed, resulting in Formula (4.24). This very simple optimization saves two parallel matrix transpositions i. e., 50 % of the communication effort, but it can only be applied if the FFT generator *knows* about the data redistribution and is enabled to handle it.

4.9.2 FFT Based Convolution

Let g and h be functions of time whose Fourier transforms are G and H:

$$\begin{aligned} g(t) &= F_\nu^{-1}\left[G(\nu)\right](t) = \int_{-\infty}^{\infty} G(\nu)e^{2\pi i\nu t}d\nu, \\ h(t) &= F_\nu^{-1}\left[H(\nu)\right](t) = \int_{-\infty}^{\infty} H(\nu)e^{2\pi i\nu t}d\nu. \end{aligned}$$

The convolution $g * h$ of g and h is defined by

$$g * h := \int_{-\infty}^{\infty} h(t')g(t-t')dt'.$$

By using the functions' Fourier transforms this expression can be transformed as follows:

$$\begin{aligned} g * h &= \int_{-\infty}^{\infty} h(t')\left[\int_{-\infty}^{\infty} G(\nu)e^{2\pi i\nu(t-t')}\right]dt' \\ &= \int_{-\infty}^{\infty} G(\nu)\left[\int_{-\infty}^{\infty} h(t')e^{-2\pi i\nu t'}dt'\right]e^{2\pi i\nu t}d\nu \\ &= \int_{-\infty}^{\infty} G(\nu)H(\nu)e^{2\pi i\nu t}d\nu \\ &= F_\nu^{-1}\left[G(\nu)H(\nu)\right](t). \end{aligned}$$

Applying the Fourier transform to each side leads to the convolution theorem.

Definition 4.7 (Convolution Theorem) *With F denoting the Fourier transform the convolution of two functions g and h can be expressed as*

$$F[g * h] = F[g]F[h].$$

Alternative forms are

$$\begin{aligned} F[gh] &= F[g] * F[h] \\ F^{-1}\left(F[g]F[h]\right) &= g * h \\ F^{-1}\left(F[g] * F[h]\right) &= gh. \end{aligned}$$

Definition 4.8 (Discrete Cyclic Convolution) *Consider two vectors g and h of size n. The discrete cyclic convolution $g * h$ is the following vector c of size n:*

$$c_k := \sum_{i=0}^{n-1} g_{k-i \bmod(n)} h_{i \bmod(n)}, \quad 0 \leq k < n.$$

A naive algorithm for the cyclic convolution requires $O(n^2)$ operations. By applying a discrete variant of the convolution theorem the complexity can be lowered to $O(n \log n)$.

Property 4.5 (Discrete Convolution Theorem) *Let g and h be vectors of size n, $g * h$ the cyclic convolution of g and h, and F_n the n-point DFT matrix then*

$$\begin{aligned} g * h &= F_n^{-1} \operatorname{diag}\left(F_n g\right) F_n h \\ \textit{or} \qquad & F_n^{-1}(F_n g \diamond F_n h) \end{aligned}$$

with the pointwise vector multiplication

$$\begin{aligned} \cdot \diamond \cdot : \quad & K^n \times K^n \longrightarrow K^n \\ & a \diamond b \mapsto [a_0 b_0, a_1 b_1, \ldots, a_{n-1} b_{n-1}]. \end{aligned}$$

Proof: Tolimimieri, An, Lu [59].

The distributed computation of FFT based convolutions requires a significant amount of communication. As a one-dimensional FFT requires three communication steps, a straightforward implementation includes nine all-to-all communication steps.

Note, the vector does not have to be stored in its correct order to allow the pointwise multiplication. Thus, the last transposition of the forward FFTs and the first transposition of the reverse FFT are not needed, which reduces the overall communication effort by 1/3 (compare Example 4.7). This—very simple—optimization is not possible if the convolution is coded manually and uses an FFT library that does not allow to skip initial and/or final transposition steps.

The FFT based convolution shows that it is often not feasible to assemble an algorithm from—even well optimized—solutions for subproblems, because some optimization potential may exceed their scope. A more reasonable approach is to attempt to generate, and optimize, computational problems as a whole.

4.10 Parallel FFT Software

As the fast Fourier transform is one of the most important algorithms in science and engineering, there are numerous software libraries providing implementations of this algorithm. The best known open source library for distributed FFTs is FFTW which reliably provides cross platform high-performance code.

4.10.1 FFTW

FFTW[4] is the acronym for "*Fastest Fourier Transform in the West*". It has been developed by Matteo Frigo and Steven Johnson [28, 29] at the Massachusetts Institute of Technology (MIT).

FFTW 2.1.5 is the latest version supporting MPI based parallel transforms while FFTW 3.1 is the most recent version implementing scalar transforms.

FFTW is an open source C subroutine library, thus portable to practically any platform. FFTW computes the discrete Fourier transform (DFT) in one or several dimensions, with both real and complex data of arbitrary size on an arbitrary number of processors.

Even though FFTW is an open source software it outperforms most other FFT programs. On some machines it performs even better than vendor optimized FFT programs as serial benchmarks on the FFTW homepage[5] show.

FFTW takes advantage of *run time compilation*, a special technique of compiling parts or whole programs to native code before executing. This new paradigm in software development tries to gain performance optimization while maintaining portable code. FFTW is able to automatically adapt its computations to a specific hardware by applying its special kind of self-configuring style. The inner computation loop of FFTW, which accounts for about 95 % of the sequential code, is generated automatically by a special-purpose compiler that takes the underlying computer architecture into account and thereby increases execution performance.

To do so, FFTW carries out a sequence of execution time measurements with its executable components, called *codelets*, on the target system. The best assembled configuration of such codelets is used on the specific hardware environment. This code generation process ensures that FFTW performs well on every machine without modification.

Thus before an FFT can be calculated using FFTW a *planner function* must be executed. This function decides which codelets to use in later calls of a certain problem size on the current system. This information ist stored in a *plan*, i. e.,

[4] `http://www.fftw.org/`
[5] `http://www.fftw.org/speed/`

a special data structure. Plans can be stored on disk for reuse in future computations. Such a (disk) file is called *wisdom*. It also contains information about the smaller problem sizes FFTW had to work on to generate the computations of a larger FFT. Thus, for instance, if a plan for a scalar one-dimensional FFT of size 8 has been created the generated *wisdom*-file also contains the best plans for FFTs of lengths 2 and 4.

FFTW is more adaptable than most other FFT libraries. For instance, it can be forced to skip initial and final permutations, which may be necessary to compute an FFT, and thus, leaving the vector in a scrambled order. This is useful for optimizing the communication effort for distributed composite transforms, e. g., the FFT based convolution (Section 4.9.2).

Chapter 5

SPIRAL/DMP

SPIRAL/DMP is an extension to SPIRAL which generates optimized code for distributed memory parallel signal transforms. This chapter provides an in-depth documentation to its features. The source codes belonging to the objects and rules introduced in this chapter are included in Appendix A. The first section gives a basic overview of SPIRAL's architecture.

5.1 Introducing SPIRAL

SPIRAL [53, 43] is a program generator for optimized linear signal transforms such as the DFT and many others. It uses a formal framework to efficiently generate alternative algorithms for a given transform and to translate them into code. Then, SPIRAL uses search and learning techniques to find the best tuned implementation for a certain platform, among this set of alternatives. SPIRAL's internal structure is shown in Figure 5.1. As input, the user provides a descriptive definition of the desired signal transform, e. g., "DFT_{256}", the output is platform optimized source code.

Formula Generation. Figure 5.1 shows that SPIRAL relies on a feedback loop to find the algorithm and implementation which suites a given environment best. Therefore, it requires a repository of *breakdown rules* to break high-level descriptions of algorithms down to formulas, which roughly represent a certain implementation. This representation of signal transform algorithms is called Signal Processing Language (SPL).

$$\mathrm{DFT}_{km} \quad \rightarrow \quad (\mathrm{DFT}_k \otimes \mathrm{I}_m)\, \mathrm{T}^{km}_m (\mathrm{I}_k \otimes \mathrm{DFT}_m)\, \mathrm{L}^{km}_k \tag{5.1}$$

Breakdown rules are derived from mathematical formulas. For instance the Cooley-Tukey FFT formula, introduced in Theorem 4.3, leads to Rule 5.1. SPIRAL's formula generation engine recursively applies breakdown rules to generate one out of many possible SPL formulas.

If the resulting formula shall meet certain requirements, this information has to be provided to the SPL compiler by *tagging* the expression. For instance, a DFT_{256}

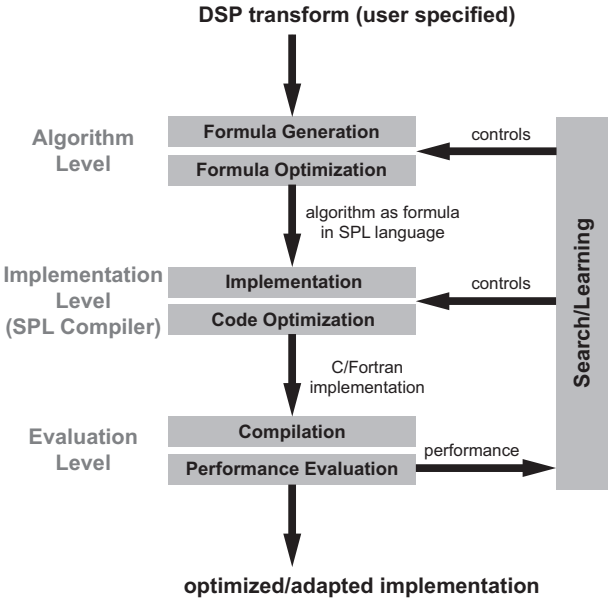


Figure 5.1: SPIRAL's architecture.

tagged for 4-way vectorization is denoted as

$$\underbrace{\mathrm{DFT}_{256}}_{\mathrm{vec}(4,*)}.$$

This tagging system steers the application of rules during code generation process and ensures that the output formulas meet the user's requirements.

Loop Optimization. SPL formulas contain computational blocks which are to be implemented as loops. Loop implementation contains a lot of tuning potential to improve data locality, but SPL does not provide a formalism to steer such optimizations. Therefore, SPL formulas are translated to Σ-SPL.

$$(\mathrm{I}_k \otimes \mathrm{DFT}_m)\,\mathrm{L}^{km}_k \;\rightarrow\; \left(\sum_{j=0}^{k-1} \mathrm{S}_{(j)_m \otimes \imath_m}\,\mathrm{DFT}_k\,\mathrm{G}_{(j)_m \otimes \imath_m}\right) \mathrm{perm}\left(\ell^{km}_k\right) \tag{5.2}$$

$$\rightarrow\; \sum_{j=0}^{k-1} \left(\mathrm{S}_{(j)_m \otimes \imath_m}\,\mathrm{DFT}_k\,\mathrm{G}_{\ell^{km}_k \circ \left((j)_m \otimes \imath_m\right)}\right) \tag{5.3}$$

Rule 5.2 shows how the second factor of Rule 5.1 would be translated to Σ-SPL. In this representation loops resulting from tensor products are displayed as sums. $\mathrm{perm}\left(\ell^{km}_k\right)$ symbolizes the loop performing the stride permutation ℓ^{km}_k on the data vector.

This formalism allows to express and apply loop optimizations, especially loop merging, on a rule based, symbolic, level. Rule 5.3, for instance, merges the two

loops by fusing the permutation ℓ_k^{km} into the addressing of the DFT calculation. Except loop merging, Σ-SPL also allows index optimizations to simplify the array indexing expressions. A detailed introduction to Σ-SPL is given in [22].

Code Generation. The Σ-SPL compiler translates Σ-SPL to programming language independent intermediate-code (iCode). A parameter passed to the compiler controls the unrolling depth. Further optimizations on code level include array scalarization, constant folding, and copy propagation. Further details about SPIRAL's code generation are given in [66] and [51].

Formula (5.3) denotes transform in complex arithmetic. As the output shall be real code, the expression has to be mapped to real arithmetic. This mapping is denoted by the the application of the bar operator $\overline{(\cdot)}$ and doubles the size of the transform.

$$\begin{aligned}
& \overline{\sum_{j=0}^{k-1} \left(\mathrm{S}_{(j)_m \otimes \imath_m} \, \mathrm{DFT}_k \, \mathrm{G}_{\ell_k^{km} \circ \left((j)_m \otimes \imath_m \right)} \right)} \\
= & \sum_{j=0}^{k-1} \left(\overline{\mathrm{S}_{(j)_m \otimes \imath_m}} \; \overline{\mathrm{DFT}_k} \; \overline{\mathrm{G}_{\ell_k^{km} \circ \left((j)_m \otimes \imath_m \right)}} \right) \\
= & \sum_{j=0}^{k-1} \left(\mathrm{S}_{\left((j)_m \otimes \imath_m \right) \otimes \imath_2} \, \overline{\mathrm{DFT}_k} \, \mathrm{G}_{\left(\ell_k^{km} \circ \left((j)_m \otimes \imath_m \right) \right) \otimes \imath_2} \right) \qquad (5.4)
\end{aligned}$$

Compiling (5.4) with parameters $k = m = 4$ and an unrolling blocksize of 8 yields the iCode in Table 5.1.

Finally the iCode is unparsed to platform independent C source code. Note that simple optimizations like precomputing repeatedly used expressions are left to the C compiler.

Optimization Loop. Because of the reasons mentioned in Chapter 1, strictly deterministic optimization is not possible on today's computing hardware. Thus, SPIRAL performs a heuristic search over the SPL formula space, to optimize to a given target architecture [55]. Therefore run times of code parts are measured and fed into a feedback loop. This feedback loop controls the breakdown rule application in the formula generation process. The feedback loop's search strategies include dynamic programming and evolutionary search.

```
loop(i43, [ 0 .. 3 ],
   decl([ s108, s107, s105, s109, s110, s111, s112, s106 ],
      chain(
         assign(s105, nth(X, mul(2, i43))),
         assign(s106, nth(X, add(8, mul(2, i43)))),
         assign(s107, nth(X, add(9, mul(2, i43)))),
         assign(s108, nth(X, add(mul(2, i43), 1))),
         assign(s109, nth(X, add(16, mul(2, i43)))),
         assign(s110, nth(X, add(17, mul(2, i43)))),
         assign(s111, nth(X, add(24, mul(2, i43)))),
         assign(s112, nth(X, add(25, mul(2, i43)))),
         assign(nth(Y, mul(8, i43)), add(s105, s106, s109, s111)),
         assign(nth(Y, add(mul(8, i43), 1)), add(s108, s107, s110, s112)),
         assign(nth(Y, add(mul(8, i43), 2)), add(sub(sub(s105, s107), s109), s112)),
         assign(nth(Y, add(mul(8, i43), 3)), sub(sub(add(s108, s106), s110), s111)),
         assign(nth(Y, add(mul(8, i43), 4)), sub(add(sub(s105, s106), s109), s111)),
         assign(nth(Y, add(mul(8, i43), 5)), sub(add(sub(s108, s107), s110), s112)),
         assign(nth(Y, add(mul(8, i43), 6)), sub(sub(add(s105, s107), s109), s112)),
         assign(nth(Y, add(mul(8, i43), 7)), add(sub(sub(s108, s106), s110), s111))
      )
   )
)
```

Table 5.1: iCode for $\overline{(\mathrm{I}_4 \otimes \mathrm{DFT}_4)\,\mathrm{L}_4^{16}}$.

```
extern void sub (double *, double *);
void sub(double *Y, double *X) {
    {int i43; for(i43 = 0; i43 <= 3; i43++)
      { double s108, s107, s105, s109, s110, s111, s112, s106;
        {
          s105 = X[2*i43];
          s106 = X[(8 + 2*i43)];
          s107 = X[(9 + 2*i43)];
          s108 = X[(2*i43 + 1)];
          s109 = X[(16 + 2*i43)];
          s110 = X[(17 + 2*i43)];
          s111 = X[(24 + 2*i43)];
          s112 = X[(25 + 2*i43)];
          Y[8*i43] = (s105 + s106 + s109 + s111);
          Y[(8*i43 + 1)] = (s108 + s107 + s110 + s112);
          Y[(8*i43 + 2)] = (((s105 - s107) - s109) + s112);
          Y[(8*i43 + 3)] = (((s108 + s106) - s110) - s111);
          Y[(8*i43 + 4)] = (((s105 - s106) + s109) - s111);
          Y[(8*i43 + 5)] = (((s108 - s107) + s110) - s112);
          Y[(8*i43 + 6)] = (((s105 + s107) - s109) - s112);
          Y[(8*i43 + 7)] = (((s108 - s106) - s110) + s111);
        }
      }
}}
```

Table 5.2: C-code corresponding to the iCode in Table 5.1.

5.2 Parallel SPL

The process of generating a parallel SPL formula for a certain transform is similar to generating a scalar one with the restrictions introduced in Section 4.8, but it is required to steer the application of the rewrite rules such that the output only consists of Kronecker parallel factors and communication steps. Therefore, a *tag* is defined which contains the information that the tagged formula shall be parallelized and how many processors are involved in the distributed computation.

Definition 5.1 (Parallel Tags)

Parallelization to p processors, regardless of the type of parallelization, e. g., shared memory parallelism (SMP) or distributed memory parallelism (DMP), is denoted as

$$\texttt{APar(p)} := par(p, *).$$

SPIRAL/DMP *Code A.1*

Distributed memory parallelization to p processors is denoted as

$$\texttt{AParDistr(p)} := par(p, dmp).$$

SPIRAL/DMP *Code A.2*

Definition 5.2 (Application of Parallel Tags) *A matrix A tagged for distributed memory parallelization to p processors is denoted as*

$$\texttt{TPar(A,AParDistr(p))} := \underbrace{A}_{par(p,\, dmp)}$$

SPIRAL/DMP *Code A.4*

Utilizing this tag a multidimensional distributed memory parallel DFT can be written as

$$\underbrace{\mathrm{MDDFT}(n_1, \ldots, n_d)}_{\mathrm{par}(p,\, \mathrm{dmp})}.$$

which corresponds to the following SPIRAL object

```
TPar(MDDFT([n1,...,nd]),AParDistr(p)).
```

To shorten formulas and rules in the remainder of this chapter, DFT matrices will be denoted as F_n, and multidimensional DFT matrices as $F_{n_1,\ldots,n_d}$.

The rules introduced in this section allow the breakdown of parallel SPL transforms (non-terminals) into SPL terminals that represent the output of this stage.

5.2.1 Parallel Computation Blocks

The first step to parallelization is to split the multidimensional DFT matrix into a product of matrices according to Property 4.4. This creates the tensor products in the formula, which are required to rewrite it to parallel factors in a later stage.

Rule 5.1 (MDDFT Split) $k = 1, \ldots, d-1$:

$$\underbrace{F_{n_1,\ldots,n_d}}_{par(*,*)} \longrightarrow \underbrace{(F_{n_1,\ldots,n_k} \otimes \mathrm{I}_{n_{k+1}\cdots n_d})}_{par(*,*)} \underbrace{(\mathrm{I}_{n_1\cdots n_k} \otimes F_{n_{k+1},\ldots,n_d})}_{par(*,*)}$$

SPIRAL/DMP *Code A.15 (`MDDFT_tSPL_RowCol`)*

Rule 5.1 splits $F_{n_1,\ldots,n_d}$ into two factors of tensor products of DFT matrices and identity matrices. A one-dimensional DFT is rewritten in a similar way utilizing the Cooley-Tukey radix-p factorization introduced in Theorem 4.3, but in this case the procedure is slightly more complex due to the occurence of a parallel twiddle-factor matrix.

Rule 5.2 (DFT Split)

$$\underbrace{F_n}_{par(*,*)} = \underbrace{F_{n_1 n_2}}_{par(*,*)} \longrightarrow \underbrace{(F_{n_1} \otimes \mathrm{I}_{n_2})}_{par(*,*)} \underbrace{T^n_{n_2}}_{par(*,*)} \underbrace{(\mathrm{I}_{n_1} \otimes F_{n_2})}_{par(*,*)} \underbrace{L^{n_1 n_2}_{n_1}}_{par(*,*)}$$

SPIRAL/DMP *Code A.16 (`DFT_tSPL_CT`)*

Rules 5.1 and 5.2 are rules which define the SPL representation of a certain transform. These rules are not specifically implemented for distributed memory parallel transforms. SPIRAL executes the same rules for shared memory parallel, vectorized, or straightforward scalar transforms. The additional information is passed on to the children in the tags. Such rules are called tSPL rules.

The tSPL rules split the initial transform into tensor products of matrices. In general the tensor products arising in these formulas have to be further broken down by using the non-terminal objects `TTensor` and `TTensorI`.

Definition 5.3 (Tensor)

$$\texttt{Tensor(A,B,[pv])} := \underbrace{A \otimes B}_{pv}$$

SPIRAL/DMP *Code A.7 (`TTensor`)*

Tensor products with identity matrices recieve a special treatment, as the final representation of all computational factors has to meet the pattern $I_p \otimes A$.

Definition 5.4 (Tensor I) $A \in \mathbb{C}^{n_1 \times n_2}$
The third and fourth parameters to `TTensorI` *can both be either* `APar` *or* `AVec`*. Thus, there are four possible definitions for this object having two interpretations, depending on whether the inner tensor product is interpreted as* $I \otimes A$ *or* $A \otimes I$*.*

$$\begin{aligned}
\texttt{TTensorI(A,n,APar,APar,pv)} &:= \underbrace{I_n \otimes A}_{pv} = \underbrace{\mathrm{L}_n^{nn_1}(A \otimes I_n)\,\mathrm{L}_{n_2}^{nn_2}}_{pv} \\
\texttt{TTensorI(A,n,AVec,AVec,pv)} &:= \underbrace{\mathrm{L}_n^{nn_1}(I_n \otimes A)\,\mathrm{L}_{n_2}^{nn_2}}_{pv} = \underbrace{A \otimes I_n}_{pv} \\
\texttt{TTensorI(A,n,APar,AVec,pv)} &:= \underbrace{(I_n \otimes A)\,\mathrm{L}_n^{nn_1}}_{pv} = \underbrace{\mathrm{L}_n^{nn_1}(A \otimes I_n)}_{pv} \\
\texttt{TTensorI(A,n,AVec,APar,pv)} &:= \underbrace{\mathrm{L}_n^{nn_1}(I_n \otimes A)}_{pv} = \underbrace{(A \otimes I_n)\,\mathrm{L}_{n_2}^{nn_2}}_{pv}
\end{aligned}$$

SPIRAL/DMP *Code A.8 (*`TTensorI`*)*

The parameters `APar` and `AVec` of `TTensorI` specify whether the rows or the columns are to be shuffled by L_n^* or not. For the further breakdown-process it is advantageous not to explicitely create these permutations, but to store them in form of tags.

The following rule splits one general tensor product into two tensor products with identity matrices according to Corollary 2.1.

Rule 5.3 (Tensor Split)

$$\underbrace{A \otimes B}_{par(*,*)} \longrightarrow \underbrace{A \otimes I}_{par(*,*)}\,\underbrace{I \otimes B}_{par(*,*)}$$

SPIRAL/DMP *Code A.19 (*`AxI_IxB`*)*

or

$$\underbrace{A \otimes B}_{par(*,*)} \longrightarrow \underbrace{I \otimes B}_{par(*,*)}\,\underbrace{A \otimes I}_{par(*,*)}$$

SPIRAL/DMP *Code A.20 (*`IxB_AxI`*)*

The application of this rule leads to a state where all computational factors are encapsulated in `TTensorI` objects. However, not all of them are of the desired pattern $I_p \otimes A$. Thus, the following rewrite rule is required to transform all computational factors to parallelizable computation stages.

Rule 5.4 (Tensor Rewrite) $p|n, A \in \mathbb{C}^{n_1 \times n_2}$

1. `TTensorI(A,n,APar,APar,AParDistr(p))`

$$\underbrace{I_n \otimes A}_{par(p,\ dmp)} \longrightarrow \underbrace{I_p \otimes (I_{n/p} \otimes A)}_{par(p,\ dmp)}$$

SPIRAL/DMP *Code A.21 (`IxA_parDMP`)*

2. `TTensorI(A,n,AVec,AVec,AParDistr(p))`

$$\underbrace{A \otimes I_n}_{par(p,\ dmp)} \longrightarrow \underbrace{(\mathrm{L}^{n_1 p}_{n_1} \otimes I_{n/p})}_{par(p,\ dmp)} \underbrace{\big(I_p \otimes (A \otimes I_{n/p})\big)}_{par(p,\ dmp)} \underbrace{(\mathrm{L}^{n_2 p}_{p} \otimes I_{n/p})}_{par(p,\ dmp)}$$

SPIRAL/DMP *Code A.22 (`AxI_parDMP`)*

3. `TTensorI(A,n,APar,AVec,AParDistr(p))`

$$\underbrace{(I_n \otimes A)\,\mathrm{L}^{nn_2}_{n}}_{par(p,\ dmp)} \longrightarrow \underbrace{\big(I_p \otimes \big(I_{n/p} \otimes A\big)\big)}_{par(p,\ dmp)} \underbrace{\mathrm{L}^{nn_2}_{n}}_{par(p,\ dmp)}$$

SPIRAL/DMP *Code A.23 (`AxIpv_parDMP`)*

4. `TTensorI(A,n,AVec,APar,AParDistr(p))`

$$\underbrace{(A \otimes I_n)\,\mathrm{L}^{nn_2}_{n_2}}_{par(p,\ dmp)} \longrightarrow \underbrace{\big(\mathrm{L}^{n_1 p}_{n_1} \otimes I_{n/p}\big)}_{par(p,\ dmp)} \underbrace{\big(I_p \otimes \big((A \otimes I_{n/p}) L^{nn_2/p}_{n_2}\big)\big)}_{par(p,\ dmp)}$$

SPIRAL/DMP *Code A.24 (`AxIvp_parDMP`)*

When a tensor product has reached the state of a parallel computation block it is transformed into a `DMPTensor` object. This is an SPL terminal and, thus, is of no interest for the SPL rewriting system any more.

Definition 5.5 (DMP Tensor)

$$\texttt{DMPTensor(p, A, AParDistr)} := \underbrace{\mathrm{I}_p \otimes A}_{par(p,\ dmp)} \tag{5.5}$$

SPIRAL/DMP *Code A.51 (`DMPTensor`)*

Distributed twiddle-factor matrices, as introduced in Definition 4.3, cannot be avoided for parallel one-dimensional FFTs. It is not possible to express such matrices as Kronecker parallel factors because they have different values on every processor. At the SPL level it is sufficient to define a transformation object for distributed diagonal matcies.

Definition 5.6 (Distributed Diagonal Matrix) *If D is a diagonal matrix then the distributed diagonal matrix representing D is defined by*

$$\texttt{TDiag(D, AParDistr (p))} := \underbrace{D}_{par(p,\, dmp)}$$

SPIRAL/DMP *Code A.12 (`TDiag`)*

These rules enable SPIRAL to express all computation blocks as Kronecker parallel factors. When a factor has been transformed to $I_p \otimes A$ the par-tag is dropped, i. e., it is *not* passed on to the embedded matrix A. So A is not tagged as parallel anymore and SPIRAL can tap its full potential of scalar code generation and optimization. These scalar code parts are executed on all processors involved in the parallel computation.

The multiplication of factors in SPL correlates to a composition of the functions represented by the factors. Parallel compositions require a different handling than scalar ones in the Σ-SPL optimization and code generation steps. Therefore, the terminal object `DMPCompose` is required.

Definition 5.7 (DMP Compose)

$$\texttt{DMPCompose([p, A1, ...,An])} := \underbrace{A_1 \cdots A_n}_{par(p,\, dmp)}$$

SPIRAL/DMP *Code A.44 (`DMPCompose`)*

The object's initialization function has been tweaked such that it accepts all kind of lists and nested lists, which contain SPL objects and integer numbers. After the list is flattened the first integer number arising in the flattened list is interpreted as p, and all other integers are dropped. The remaining SPL objects are composed. This makes the coding of some Σ-SPL rules easier.

5.2.2 Stride Permutations

During the process of forcing computation blocks to a pattern compatible with distributed memory parallel processing additional stride permutation matrices are produced (Rules 5.1, 5.2, 5.4).

Often, when stride permutations arise, they appear in factors like $L^{mn}_m \otimes I_r$, $I_l \otimes L^{mn}_m$, or even $I_l \otimes L^{mn}_m \otimes I_r$. It is feasible to treat such blocks of stride permutations and identity matrices as a whole. These blocks represent a class of permutations that extend the definition of stride permutation.

Definition 5.8 (Stride Permutation Block) *n|m*

$$\texttt{TL(m, n, l, r, pv)} := \underbrace{\mathrm{I}_l \otimes \mathrm{L}^m_n \otimes \mathrm{I}_r}_{pv}$$

This includes the special cases

$$\begin{aligned}
\texttt{TL(m, n, 1, r, pv)} &:= \underbrace{\mathrm{L}^m_n \otimes \mathrm{I}_r}_{pv} \\
\texttt{TL(m, n, l, 1, pv)} &:= \underbrace{\mathrm{I}_l \otimes \mathrm{L}^m_n}_{pv} \\
\texttt{TL(m, n, 1, 1, pv)} &:= \underbrace{\mathrm{L}^m_n}_{pv}
\end{aligned}$$

SPIRAL/DMP *Code A.9 (TL)*

As derived in Section 3.2 stride permutations cannot be implemented in a straightforward manner. They rather have to be split to global communication and local permutation parts. The desirable patterns $\mathrm{I}_p \otimes P$, and $\mathrm{L}^{p^2}_p \otimes \mathrm{I}_b$ are both instances of the class of permutations introduced in Definition 5.8. As P itself can be a tensor product of stride permutations and identity matrices too, suitable targets of the rewrite process are

$$\begin{aligned}
\underbrace{\mathrm{I}_p \otimes (\mathrm{I}_l \otimes L^m_n \otimes \mathrm{I}_r)}_{\mathrm{par}(p,\mathrm{dmp})} &= \texttt{TL(m,n,p*l,r,AParDistr(p))} \\
\underbrace{\mathrm{L}^{p^2}_p \otimes \mathrm{I}_b}_{\mathrm{par}(p,\mathrm{dmp})} &= \texttt{TL(p\^{}2,p,1,b,AParDistr(p))}.
\end{aligned}$$

To obtain separations yielding these results Lemma 3.2 can be used. The rewrite rules resulting from these properties are defined in the following.

Rule 5.5 (Stride Split) *This rewrite rule handles expressions of the kind* $\underbrace{L^m_n \otimes \mathrm{I}_r}_{par(p,\ dmp)}$ *with* $p|n$, *and* $p|(m/n)$.

1. *General rule for splitting a tensor product to parallelizable factors.*

$$\underbrace{L^m_n \otimes \mathrm{I}_r}_{par(p,\ dmp)} \longrightarrow \underbrace{(I_p \otimes L^{m/p}_{n/p} \otimes \mathrm{I}_r)}_{par(p,\ dmp)} \underbrace{(L^{p^2}_p \otimes I_{mr/p^2})}_{par(p,\ dmp)} \underbrace{(I_p \otimes L^{m/n}_p \otimes I_{nr/p})}_{par(p,\ dmp)} \tag{5.6}$$

SPIRAL/DMP *Code A.34 (`IxLxI_DMP_LCL`)*

2. *If* $n = p$ *the right term of the result would degenerate to* I_{lmr} *so (5.6) can be simplified to*

$$\underbrace{L^m_p \otimes \mathrm{I}_r}_{par(p,\ dmp)} \longrightarrow \underbrace{(L^{p^2}_p \otimes I_{mr/p^2})}_{par(p,\ dmp)} \underbrace{(I_p \otimes L^{m/p}_p \otimes I_r)}_{par(p,\ dmp)} \tag{5.7}$$

SPIRAL/DMP *Code A.35 (`IxLxI_DMP_CL`)*

3. *If* $m/n = p$ *the left term of the result would degenerate to* I_{lmr} *so (5.6) can be simplified to*

$$\underbrace{L^{np}_n \otimes \mathrm{I}_r}_{par(p,\ dmp)} \longrightarrow \underbrace{(I_p \otimes L^n_{n/p} \otimes \mathrm{I}_r)}_{par(p,\ dmp)} \underbrace{(L^{p^2}_p \otimes I_{nr/p})}_{par(p,\ dmp)} \tag{5.8}$$

SPIRAL/DMP *Code A.36 (`IxLxI_DMP_LC`)*

Theoretically the special cases of Rule 5.5 would not be necessary. Anyway, without implementing the special rules (5.7) and (5.8) additional identity matrices, which result in no arithmetical operation, would be generated as computation factors. This is not feasible and would impact the performance of the code generation process.

Rule 5.6 (Special Stride Rules) *These rewrite rules are required to maintain consistency and allow the further breakdown of scalar formulas.*

1. *If* $p = 1$ *the expression represents a local permutation and the par-tag is dropped.*

$$\underbrace{\mathrm{I}_l \otimes L^m_n \otimes \mathrm{I}_r}_{par(1,\ dmp)} \longrightarrow \mathrm{I}_l \otimes L^m_n \otimes \mathrm{I}_r$$

SPIRAL/DMP *Code A.32 (`IxLxI_nopar`)*

2. *If $n = 1$ or $n = m$ the central stride permutation L^m_n degenerates to I_m.*

$$\underbrace{\mathrm{I}_l \otimes L^m_1 \otimes \mathrm{I}_r}_{par(p,\, dmp)} \longrightarrow \underbrace{\mathrm{I}_{lmr}}_{par(p,\, dmp)}$$
$$\underbrace{\mathrm{I}_l \otimes L^m_m \otimes \mathrm{I}_r}_{par(p,\, dmp)} \longrightarrow \underbrace{\mathrm{I}_{lmr}}_{par(p,\, dmp)}$$

SPIRAL/DMP *Code A.33 (`IxLxI_trivial`)*

3. *If $p|l$ the whole expression represents a tensor product of local permutations and becomes a computational block.*

$$\underbrace{I_l \otimes L^m_n \otimes \mathrm{I}_r}_{par(p,\, dmp)} \longrightarrow \underbrace{I_p \otimes (I_{l/p} \otimes L^m_n \otimes \mathrm{I}_r)}_{par(p,\, dmp)}$$

SPIRAL/DMP *Code A.37 (`IxLxI_DMP_L`)*

5.2.3 Communication Patterns

Rules 5.5 and 5.6 are sufficient to split all tensor products, which arise during the break-down process of parallel one- and multidimensional FFTs, to global communication and local permutation steps. All matrices, which represent inter processor communication, are exactly of the form $L^{p^2}_p \otimes I_b$. The advantages of such a pattern have been discussed in Section 3.2.1.

Definition 5.9 (Global Matrix Transposition) *`TDMPGlobalTranspose`, an* SPL *non-terminal represents a communication stage representing a global matrix transposition as introduced in Section 3.2.1.*

$$\texttt{TDMPGlobalTranspose(n, AParDistr (p))} := \underbrace{\mathrm{L}^{p^2}_p \otimes \mathrm{I}_n}_{par(p,\, dmp)}$$

SPIRAL/DMP *Code A.11 (`TDMPGloablTranspose`)*

In fact all global transposes generated in Rule 5.5 do not result in stride permutations (`TL`) but are generated as `TDMPGlobalTranspose` objects right away. Global transposes have advantages but are very restrictive too.

Definition 5.10 (Arbitrary Communication Stage) *The* `TComm` SPL *nonterminal represents a communication stage with an arbitrary communication pattern according to Definition 3.2.*

```
TComm(p, n, w_jk, pi, piinv, r_jk, j, k, AParDistr (p))
```

$$:= \underbrace{\mathrm{comm}_n^{p\circlearrowleft}\left({w_{jk}}^{1\to N},\ \pi^{mp\circlearrowleft},\ {r_{jk}}^{1\to N}\right)}_{par(p,\ dmp)}$$

SPIRAL/DMP *Code A.10 (*`TComm`*)*

The exact semantics of the parameters to $\mathrm{comm}_n^{p\circlearrowleft}$ and `TComm` will be dealt with later in Definition 5.15 because they represent Σ-SPL functions. The basic idea is to split the permutation matrix P into a matrix C' without fixpoints and a matrix containing only the fixpoints F. After this splitting the empty rows and columns of C' are collapsed by multiplication with non-square matrices W and $R = W^T$:

$$P = F + C' = F + WCW^T = F + WCR.$$

The functions ${w_{jk}}^{1\to N}$ and ${r_{jk}}^{1\to N}$ generate the matrices W and R. The permutation $\pi^{mp\circlearrowleft}$ represents the communication matrix C without fixpoints. In case the communication shall be performed in place, i. e., the input and output arrays are the same, the copy-operations implied by the fixpoint matrix F can be skipped.

This approach simplifies the code generation, as each line of the permutation matrix $\pi^{mp\circlearrowleft}$ represents the sending of a data packet of size n from one processor to another and no further checks for fixpoints or local permutations are required at later stages of the code generation process.

Example 5.1 (TComm) This example illustrates the decomposition of the communication step $\underbrace{\mathrm{L}^4_2 \otimes \mathrm{I}_2}_{\mathrm{par}(2,\mathrm{dmp})}$.

$$\underbrace{\underbrace{\begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 \end{bmatrix}}_{\mathrm{L}^4_2 \otimes \mathrm{I}_2} = \underbrace{\begin{bmatrix} 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 \end{bmatrix}}_{P=\mathrm{L}^4_2} \otimes \underbrace{\begin{bmatrix} 1 & \cdot \\ \cdot & 1 \end{bmatrix}}_{\mathrm{I}_2} =}_{\mathrm{par}(2,\mathrm{dmp})}$$

$$= \underbrace{\left(\underbrace{\begin{bmatrix} 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 \end{bmatrix}}_{F} + \underbrace{\begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}}_{C'} \right) \otimes \underbrace{\begin{bmatrix} 1 & \cdot \\ \cdot & 1 \end{bmatrix}}_{\mathrm{I}_2} =}_{\mathrm{par}(2,\mathrm{dmp})}$$

$$= \underbrace{\left(\underbrace{\begin{bmatrix} 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 \end{bmatrix}}_{F} + \underbrace{\begin{bmatrix} \cdot & \cdot \\ 1 & \cdot \\ \cdot & 1 \\ \cdot & \cdot \end{bmatrix}}_{W} \cdot \underbrace{\begin{bmatrix} \cdot & 1 \\ 1 & \cdot \end{bmatrix}}_{C} \cdot \underbrace{\begin{bmatrix} \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot \end{bmatrix}}_{R=W^T} \right) \otimes \underbrace{\begin{bmatrix} 1 & \cdot \\ \cdot & 1 \end{bmatrix}}_{\mathrm{I}_2}}_{\mathrm{par}(2,\mathrm{dmp})}$$

This factorization leads to the following communication pattern among two processors

$$\underbrace{C \otimes \mathrm{I}_2}_{\mathrm{par}(2,\mathrm{dmp})} = \left[\begin{array}{c|c} \cdot & 1 \\ \hline 1 & \cdot \end{array}\right] \otimes \begin{bmatrix} 1 & \cdot \\ \cdot & 1 \end{bmatrix} = \left[\begin{array}{cc|cc} \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & 1 \\ \hline 1 & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot \end{array}\right].$$

Thus, one data packet of size 2 is sent from processor 0 to processor 1 and vice versa. The information about exact locations of the sent and received packages is contained in the matrices R and W.

The terminals `DMPGlobalTranspose` (Code A.46)and `DMPComm` (Code A.45) are the terminals for communication kernels. The rules `TComm_Base` (Code A.42) and `TDMPGlobalTranspose_Base` (Code A.30) create the terminal from the respective transforms.

The reason for going this way, rather than immediately creating the terminal, is that there is the special rule `TDMPGlobalTranspose2TComm` (Code A.31) which creates a `TComm` object from `TDMPGlobalTranspose`.

Rule 5.7 (Global Transpose to Comm)

$$\underbrace{\mathrm{L}^{p^2}_{p} \otimes \mathrm{I}_n}_{par(p,\,mpi)} \quad \longrightarrow \quad \mathrm{comm}^{p\circlearrowleft}_{n}\left(\zeta^{p}_{jk} \otimes \imath_n,\ \gamma_p \circ \ell^{p^2}_{p} \circ \gamma'_p,\ \zeta^{p}_{jk} \otimes \imath_n\right)$$

with
$$\gamma'_p : \mathbb{I}_{p(p-1)} \to \mathbb{I}_{p^2};\ i \mapsto i + 1 + \lfloor \tfrac{i}{p} \rfloor$$
$$\gamma_p : \mathbb{I}_{p^2} \to \mathbb{I}_{p(p-1)};\ i \mapsto i - 1 - \lfloor \tfrac{i}{p+1} \rfloor$$
$$\zeta^{N}_{jk} : \mathbb{I}_1 \to \mathbb{I}_N;\ i \mapsto \begin{cases} k & \text{if } k < j \\ k+1 & \text{else} \end{cases}$$

SPIRAL/DMP *Code A.31 (`TDMPGlobalTranspose2TComm`)*

$\ell^{p^2}_{p}$ *is the generating function of* $\mathrm{L}^{p^2}_{p}$.

The semantics of Rule 5.6 is to split the communication pattern as demonstrated in Example 5.1. γ'_p and γ_p drop the fixpoints out of the communication matrix. The p fixpoints of $\mathrm{L}^{p^2}_{p}$ are located in every $p+1$-th row. ζ^{N}_{jk} is required to address the correct vector elements after the permutation matrix has been reduced.

The communication implementation can be selected by globally switching between `TDMPGlobalTranspose_Base` and `TDMPGlobalTranspose2TComm`. If both rules are left switched on the optimizing system will test both possibilities and chose the faster one.

5.2.4 Summary

The rules introduced in this chapter allow the automatic breakdown of transforms from a descriptive definition to a fast parallel algorithm in Kronecker product notation.

The application of the parallel rules implemented in SPIRAL/DMP together with SPIRAL's own rules for scalar code leads to a *ruletree*. The following example shows a ruletree for $\underbrace{\mathrm{MDDFT}_{8,4}}_{\mathrm{par}(2,\,\mathrm{dmp})}$.

Example 5.2 (Ruletree) Random SPIRAL ruletree for `TPar(DFT([8,4]),AParDistr(2))`.

```
TPar(MDDFT([ 8, 4 ], 1, [  ], false), AParDistr(2))     {TPar_setpv}
 `--MDDFT([ 8, 4 ], 1, [ AParDistr(2) ], false)     {MDDFT_tSPL_RowCol}
    `--TTensor(MDDFT([ 8 ], 1, [  ], false), MDDFT([ 4 ], 1, [  ], false), [ AParDistr(2) ])     {IxB_AxI}
       `--TCompose([ TTensorI(MDDFT([ 4 ], 1, [  ], false), 8, APar, APar, [  ]), TTensorI(MDDFT([ 8 ], 1, [  ], false), 4, AVec, AVec, [  ]) ], [ AParDistr(2) ])     {TCompose_DMP_Base}
          |--TTensorI(MDDFT([ 4 ], 1, [  ], false), 8, APar, APar, [ AParDistr(2) ])     {IxA_parDMP}
          |   `--TTensorI(MDDFT([ 4 ], 1, [  ], false), 4, APar, APar, [  ])     {IxA_base}
          |       `--MDDFT([ 4 ], 1, [  ], false)     {MDDFT_Base}
          |           `--DFT(4, 1)     {DFT_tSPL_CT}
          |               `--TCompose([ TTensorI(DFT(2, 1), 2, AVec, AVec, [  ]), TDiag(T(4, 2, 1), [  ]), TTensorI(DFT(2, 1), 2, APar, AVec, [  ]) ], [  ])     {TCompose_base}
          |                   |--TTensorI(DFT(2, 1), 2, AVec, AVec, [  ])     {AxI_base}
          |                   |   `--DFT(2, 1)     {DFT_Base}
          |                   |--TDiag(T(4, 2, 1), [  ])     {TDiag_base}
          |                   `--TTensorI(DFT(2, 1), 2, APar, AVec, [  ])     {IxA_L_base}
          |                       `--DFT(2, 1)     {DFT_Base}
          `--TTensorI(MDDFT([ 8 ], 1, [  ], false), 4, AVec, AVec, [ AParDistr(2) ])     {AxI_parDMP}
              |--TL(16, 8, 1, 2, [ AParDistr(2) ])     {IxLxI_DMP_LC}
              |   |--TL(8, 4, 2, 2, [ AParDistr(2) ])     {IxLxI_DMP_L}
              |   |   `--TTensorI(TL(8, 4, 1, 2, [  ]), 2, APar, APar, [ AParDistr(2) ])     {IxA_parDMP}
              |   |       `--TL(8, 4, 1, 2, [  ])     {L_base}
              |   `--TDMPGlobalTranspose(8, AParDistr(2))     {TDMPGlobalTranspose_Base}
              |--TTensorI(TTensorI(MDDFT([ 8 ], 1, [  ], false), 2, AVec, AVec, [  ]), 2, APar, APar, [ AParDistr(2) ])     {IxA_parDMP}
              |   `--TTensorI(MDDFT([ 8 ], 1, [  ], false), 2, AVec, AVec, [  ])     {AxI_base}
              |       `--MDDFT([ 8 ], 1, [  ], false)     {MDDFT_Base}
              |           `--DFT(8, 1)     {DFT_tSPL_CT}
              |               `--TCompose([ TTensorI(DFT(4, 1), 2, AVec, AVec, [  ]), TDiag(T(8, 2, 1), [  ]), TTensorI(DFT(2, 1), 4, APar, AVec,
   [  ]) ], [  ])     {TCompose_base}
              |                   |--TTensorI(DFT(4, 1), 2, AVec, AVec, [  ])     {AxI_base}
              |                   |   `--DFT(4, 1)     {DFT_tSPL_CT}
              |                   |       `--TCompose([ TTensorI(DFT(2, 1), 2, AVec, AVec, [  ]), TDiag(T(4, 2, 1), [  ]),
 TTensorI(DFT(2, 1), 2, APar, AVec, [  ]) ], [  ])     {TCompose_base}
              |                   |           |--TTensorI(DFT(2, 1), 2, AVec, AVec, [  ])     {AxI_base}
              |                   |           |   `--DFT(2, 1)     {DFT_Base}
              |                   |           |--TDiag(T(4, 2, 1), [  ])     {TDiag_base}
              |                   |           `--TTensorI(DFT(2, 1), 2, APar, AVec, [  ])     {IxA_L_base}
              |                   |               `--DFT(2, 1)     {DFT_Base}
              |                   |--TDiag(T(8, 2, 1), [  ])     {TDiag_base}
              |                   `--TTensorI(DFT(2, 1), 4, APar, AVec, [  ])     {IxA_L_base}
              |                       `--DFT(2, 1)     {DFT_Base}
              `--TL(16, 2, 1, 2, [ AParDistr(2) ])     {IxLxI_DMP_CL}
                  |--TDMPGlobalTranspose(8, AParDistr(2))     {TDMPGlobalTranspose_Base}
                  `--TL(8, 2, 2, 2, [ AParDistr(2) ])     {IxLxI_DMP_L}
                      `--TTensorI(TL(8, 2, 1, 2, [  ]), 2, APar, APar, [ AParDistr(2) ])     {IxA_parDMP}
                          `--TL(8, 2, 1, 2, [  ])     {L_base}
```

Such a ruletree exactly displays the sources and targets of each step of the breakdown process. Example 5.3 demonstrates the application of the command `SPLRuleTree` which translates the ruletree to easier readable SPL objects.

Example 5.3 (SPL Ruletree) SPL formula of the ruletree created in Example 5.2.

```
spiral> SPLRuleTree(r);
DMPTensor(Tensor(
  I(4),
  Tensor(F(2), I(2)) *
  T(4, 2, 1) *
  Tensor(I(2), F(2)) *
  L(4, 2)
), 2, AParDistr(2)) *DMP*
DMPTensor(Tensor(L(8, 4), I(2)), 2, AParDistr(2)) *DMP*
DMPGlobalTranspose(AParDistr(2), 8) *DMP*
DMPTensor(Tensor(
  Tensor(
    Tensor(F(2), I(2)) *
    T(4, 2, 1) *
    Tensor(I(2), F(2)) *
    L(4, 2),
    I(2)
  ) *
  T(8, 2, 1) *
  Tensor(I(4), F(2)) *
  L(8, 4),
  I(2)
), 2, AParDistr(2)) *DMP*
DMPGlobalTranspose(AParDistr(2), 8) *DMP*
DMPTensor(Tensor(L(8, 2), I(2)), 2, AParDistr(2))
```

This algorithm is equivalent to the following expression in Kronecker product notation

$$\underbrace{\left(\mathrm{I}_2 \otimes \left(\mathrm{I}_4 \otimes \left((F_2 \otimes \mathrm{I}_2)\,\mathrm{T}^4_2(\mathrm{I}_2 \otimes F_2)\,\mathrm{L}^4_2\right)\right)\right)}_{\mathrm{par}(2,\mathrm{dmp})} \underbrace{\left(\mathrm{I}_2 \otimes \left(\mathrm{L}^8_4 \otimes \mathrm{I}_2\right)\right)}_{\mathrm{par}(2,\mathrm{dmp})} \underbrace{\left(\mathrm{L}^4_2 \otimes \mathrm{I}_8\right)}_{\mathrm{par}(2,\mathrm{dmp})}$$
$$\underbrace{\left(\mathrm{I}_2 \otimes \left(\left(\left(\left(\left(F_2 \otimes \mathrm{I}_2\right)\mathrm{T}^4_2(\mathrm{I}_2 \otimes F_2)\,\mathrm{L}^4_2\right) \otimes \mathrm{I}_2\right)\mathrm{T}^8_2(\mathrm{I}_4 \otimes F_2)\,\mathrm{L}^8_4\right) \otimes \mathrm{I}_2\right)\right)}_{\mathrm{par}(2,\mathrm{dmp})}$$
$$\underbrace{\left(\mathrm{L}^4_2 \otimes \mathrm{I}_8\right)}_{\mathrm{par}(2,\mathrm{dmp})} \underbrace{\left(\mathrm{I}_2 \otimes \left(\mathrm{L}^8_2 \otimes \mathrm{I}_2\right)\right)}_{\mathrm{par}(2,\mathrm{dmp})}.$$

5.3 Σ-SPL

After a transform has been broken down to an algorithm in SPL notation it has to be converted to Σ-SPL to optimize its loop structure. A parallel computation part has similar semantics as a scalar loop. The only differences are that (*i*) the iterations are not executed sequentially but parallelly and (*ii*) each iteration can only access certain parts of the data, i. e., the data which resides in the memory of the processor represented by that loop iteration. The transformation semantics from SPL to Σ-SPL are coded in the SPL terminals' `.sums()` functions.

Σ-SPL (Franchetti et al. [22]) is briefly introduced in Section 5.1. The rules defined in the following will be displayed in both pseudo-code, which simplifies the reading of the rules in Appendix A, and Σ-SPL notation.

5.3.1 Conversion of SPL to Σ-SPL

The key-object for parallel computation blocks in SPIRAL/DMP is the parallel iterative sum. All parallel computation blocks are represented as iterative sums in Σ-SPL.

Definition 5.11 (DMP Iterative Sum)

$$\texttt{DMPISum(var, domain, expr, AParDistr(p))} := \underbrace{\sum_{var=0}^{domain-1} expr}_{par(domain,\, dmp)} \quad (5.9)$$

SPIRAL/DMP *Code A.49 (`DMPISum`)*

Passing the par tag to `DMPISum` seems redundant as the sum's domain has to be equal to the number of processors. However, certain optimizations may require more complex tags. The optimizations introduced in Section 5.6 are an example where the par tag contains more information than just the number of processes.

`DMPTensor.sums()` converts the SPL tensor product to an iterative sum in Σ-SPL, a SPL formulas usually contain tensor products rather than direct sums.

Rule 5.8 (Tensor to Iterative Sum) $A \in \mathbb{C}^{m\times m}$:

```
DMPTensor(A,n,pv)  ⟶  DMPISum(i,n,
                        DMPScat(i,p,fId(Rows(A))) *
                        A.sums() *
                        DMPGath(i,p,fId(Rows(A))),pv)
```

$$\underbrace{\mathrm{I}_p \otimes A}_{par(p,\, dmp)} = \underbrace{\bigoplus_{i=0}^{p-1} A}_{par(p,\, dmp)} \longrightarrow \underbrace{\sum_{i=0}^{p-1} \mathrm{S}_{(i)_p \otimes \imath_m} A\, \mathrm{G}_{(i)_p \otimes \imath_m}}_{par(p,\, dmp)}$$

SPIRAL/DMP *Code A.55 (`DMPTensor.sums()`)*

The parallel gather and scatter routines that occur in `DMPTensor.sums()` are adapted versions of the scalar `Gath` and `Scat` objects. Parallel gather and scatter functions must match the pattern $(i)_p \otimes f^{n\to n}$. So they have been defined such that they do not accept the whole gather or scatter function as parameter, but only p, i, and $f^{n\to n}$.

Definition 5.12 (Parallel Scatter) *The parallel scatter function* `DMPScat` *symbolically writes a processor's part of the data array into the global data vector.*

$$\begin{aligned} \texttt{DMPScat(i, p, f)} &:= \underbrace{\mathrm{S}_{(i)_p \otimes f^{n\to n}}}_{par(p,\,dmp)} \\ &= \left(\left[\bigm|_{i=0}^{p-1} \right] \left(\left[\bigm|_{j=0}^{n} \right] e^N_{((i)_p \otimes f^{n\to n}(j))} \right) \right)^T \\ &= \left[\overline{}_{i=0}^{p-1} \right] \left(\left[\overline{}_{j=0}^{n} \right] \left(e^N_{((i)_p \otimes f^{n\to n}(j))} \right)^T \right) \end{aligned}$$

SPIRAL/DMP *Code A.48 (*`DMPScat`*)*

Definition 5.13 (Parallel Gather) *The parallel gather function* `DMPGath` *selects a certain processor's part of the global data vector.*

$$\begin{aligned} \texttt{DMPGath(i, p, f)} &:= \underbrace{\mathrm{G}_{(i)_p \otimes f^{n\to n}}}_{par(p,\,dmp)} = \underbrace{\mathrm{S}^T_{(i)_p \otimes f^{n\to n}}}_{par(p,\,dmp)} \\ &= \left[\bigm|_{i=0}^{p-1} \right] \left(\left[\bigm|_{j=0}^{n} \right] e^N_{((i)_p \otimes f^{n\to n}(j))} \right) \end{aligned}$$

SPIRAL/DMP *Code A.47 (*`DMPGath`*)*

One parallel iterative sum's iteration represents one processor's workload. The parallel gather and scatter functions are required to maintain the mathematical correctness of the expression represented by the iterative sum. In practice they do not require any code to be generated as long as $f^{n\to n} = fIdn$, which is always true for the expressions treated in this chapter. So it is sufficient to generate code for the iterative sum's computational kernel A as the following example shows.

Example 5.4 (Iterative Sum) This example will analyze $\underbrace{\mathrm{I}_4 \otimes F_2}_{\mathrm{par}(4,\,\mathrm{dmp})}$:

$$\begin{aligned} \underbrace{\mathrm{I}_4 \otimes F_2}_{\mathrm{par}(4,\,\mathrm{dmp})} &\longrightarrow \underbrace{\bigoplus_{i=0}^{3} F_2}_{\mathrm{par}(4,\,\mathrm{dmp})} \\ &\longrightarrow \underbrace{\mathrm{S}_{(i)_4 \otimes \imath_2}\, F_2\, \mathrm{G}_{(i)_4 \otimes \imath_2}}_{\mathrm{par}(4,\,\mathrm{dmp})} \end{aligned}$$

The original expression is tagged for parallelization to four processors with two data points resident on every processor. The resulting Σ-SPL formula symbolizes gathering each processor's local data points, executing F_2 on them, and storing them back into the global data array.

$$i=0: \quad \begin{bmatrix} 1 & \cdot \\ \cdot & 1 \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix} = \begin{bmatrix} 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ -1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix} = A_0$$

$$\vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots$$

$$i=3: \quad \begin{bmatrix} \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ 1 & \cdot \\ \cdot & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 \end{bmatrix} = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 1 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & -1 & 1 \end{bmatrix} = A_3$$

This illustrates that, omitting the gather and scatter steps, each iteration's kernel represents one processor's job, namely computing F_2. Anyways, including these steps has the effect that the iterative sum as a whole represents the operation on the global data vector.

$$\Longrightarrow \quad \sum_{i=0}^{3} A_i = \left[\begin{array}{cc|cc|cc|cc} 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ -1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \hline \cdot & \cdot & 1 & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & -1 & 1 & \cdot & \cdot & \cdot & \cdot \\ \hline \cdot & \cdot & \cdot & \cdot & 1 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & -1 & 1 & \cdot & \cdot \\ \hline \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 1 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & -1 & 1 \end{array}\right] = \mathrm{I}_4 \otimes F_2$$

The direct sum's requirement that, on a certain position, only one matrix may contain a non-zero element naturally fits to parallel programming, as every data point is only stored on one processor. It is not possible that two processor's kernels operate on one element at the same time.

The SPL terminals for diagonal matrices and communication steps (`Diag`, `DMPComm`, and `DMPGlobalTranspose` serve as Σ-SPL objects right away. Therefore, the `.sums()` routines of these objects (Code A.53 and A.54) just return the objects themselves. Communication and parallel computation blocks are composed by `DMPCompose`. `DMPCompose` is both an SPL terminal and a Σ-SPL object too, but upon the calling of `DMPCompose.sums()` (Code A.52) its chidren are recursively converted to Σ-SPL too.

This enables SPIRAL/DMP to convert any given FFT's SPL formula to Σ-SPL. Example 5.5 shows the Σ-SPL representation of the ruletree from Example 5.2.

Example 5.5 (Σ-SPL) Σ-SPL representation of the ruletree created in Example 5.2.

```
DMPISum(i12, 2,
  DMPScat(([i12]_2 X I16)) *
  ISum(i13, 4,
    Scat(([i13]_4 X I4)) *
    ISum(i16, 2,
      Scat((I2 X [i16]_2)) *
      Blk([ [ 1, 1 ], [ 1, -1 ] ]) *
      Gath((I2 X [i16]_2))
    ) *
    Diag(T(4, 2, 1)) *
    ISum(i17, 2,
      Scat(([i17]_2 X I2)) *
      Blk([ [ 1, 1 ], [ 1, -1 ] ]) *
      Gath(([i17]_2 X I2))
    ) *
    Prm(L(4, 2)) *
    Gath(([i13]_4 X I4))
  ) *
  DMPGath(([i12]_2 X I16))
) *DMP*
DMPISum(i19, 2,
  DMPScat(([i19]_2 X I16)) *
  Prm((L(8, 4) X I2)) *
  DMPGath(([i19]_2 X I16))
) *DMP*
DMPGlobalTranspose(8, AParDistr(2)) *DMP*
DMPISum(i20, 2,
  DMPScat(([i20]_2 X I16)) *
  ISum(i22, 2,
    Scat((I8 X [i22]_2)) *
    ISum(i24, 2,
      Scat((I4 X [i24]_2)) *
      ISum(i26, 2,
        Scat((I2 X [i26]_2)) *
        Blk([ [ 1, 1 ], [ 1, -1 ] ]) *
        Gath((I2 X [i26]_2))
      ) *
      Diag(T(4, 2, 1)) *
      ISum(i27, 2,
        Scat(([i27]_2 X I2)) *
        Blk([ [ 1, 1 ], [ 1, -1 ] ]) *
        Gath(([i27]_2 X I2))
      ) *
      Prm(L(4, 2)) *
      Gath((I4 X [i24]_2))
    ) *
    Diag(T(8, 2, 1)) *
    ISum(i29, 4,
      Scat(([i29]_4 X I2)) *
      Blk([ [ 1, 1 ], [ 1, -1 ] ]) *
      Gath(([i29]_4 X I2))
    ) *
    Prm(L(8, 4)) *
    Gath((I8 X [i22]_2))
  ) *
  DMPGath(([i20]_2 X I16))
) *DMP*
DMPGlobalTranspose(8, AParDistr(2)) *DMP*
DMPISum(i31, 2,
  DMPScat(([i31]_2 X I16)) *
  Prm((L(8, 2) X I2)) *
  DMPGath(([i31]_2 X I16))
)
```

5.3.2 Communication in Σ-SPL

The parameters to the SPL comm structure (Definition 5.10) are rather complex because they contain a lot of information. This information was not immediately necessary at the SPL rewriting level but is advantageous in Σ-SPL and, especially, at the code generation level. In fact the parameters represent Σ-SPL functions which will be discussed in this section.

First of all the new operator inplace, which is required to handle communication in Σ-SPL, is defined.

Definition 5.14 (Inplace Operator) $A^{m\times n} = \left[a_{i,j}\right]_{0\le i<m, 0\le j<n}$

$$\underbrace{A^{m\times n}}_{inplace} := \left[a'_{i,j}\right]_{0\le i<m, 0\le j<n}$$

$$with \quad a'_{i,j} = \begin{cases} 1 & if\ i = j\ \ and\ a_{i,j} = 0\ \forall\ 0 \le j < n \\ 0 & if\ i \neq j\ \ and\ a_{i,j} = 0\ \forall\ 0 \le j < n \\ a_{i,j} & else \end{cases}$$

The inplace operator sets empty rows' main diagonal elements to 1. This formally represents an elementwise matrix addition $A + B$ with B containing the added elements. This is necessary, because communication patterns shall only contain communicated elements, but no fixpoints.

Example 5.6 (Inplace)

$$\underbrace{\begin{bmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix}}_{\text{inplace}} = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 \end{bmatrix} =$$

$$= \begin{bmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix} + \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 \end{bmatrix}$$

In case that communication is inplace, i. e., the source array and the target array are the same, the communication step does not have to take care about these

elements. If the communication is out-of-place, the communiction's fixpoints have to be locally copied from the source to the destination.

This allows to define the Σ-SPL representation of comm as follows.

Definition 5.15 (Blockwise Communication)

$$\begin{aligned}&\mathrm{comm}_n^{p\circlearrowleft}\left({w_{jk}}^{1\to N},\ \pi^{mp\circlearrowleft},\ {r_{jk}}^{1\to N}\right)\\ :=&\underbrace{\bigoplus_{j=0}^{p-1}\mathrm{S}_{[\_\_]_{k=0}^{m-1}\left({w_{jk}}^{1\to N}\otimes\imath_n\right)}\ \mathrm{perm}\left(\pi^{mp\circlearrowleft}\otimes\imath_n\right)\bigoplus_{j=0}^{p-1}\mathrm{G}_{[\_\_]_{k=0}^{m-1}\left({r_{jk}}^{1\to N}\otimes\imath_n\right)}}_{inplace,\ par(p,\,mpi)}\end{aligned}$$

The central permutation represents the communication pattern. The gather and scatter matrices select which elements should be communicated and where they should be placed at their destination. They also filter out fixpoints. Formally the iterative sum's matrix representation would contain zeros in the rows and columns which contain the fixpoints. These are re-added by the inplace operator.

The comm object representing a global matrix transposition in SPL has been introduced in Rule 5.7. Definition 5.15 leads to the following Σ-SPL representation for this special communication pattern.

Rule 5.9 (Global Matrix Transposition)

$$\begin{aligned}&\mathrm{comm}_n^{p\circlearrowleft}\left(\zeta_{jk}^p,\ \gamma_p\circ\ell_p^{p^2}\circ\gamma'_p,\ \zeta_{jk}^p\right)\\ \longrightarrow&\underbrace{\bigoplus_{j=0}^{p-1}\mathrm{S}_{[\_\_]_{k=0}^{m-1}\left(\zeta_{jk}^p\otimes\imath_n\right)}\ \mathrm{perm}\left(\left(\gamma_p\circ\ell_p^{p^2}\circ\gamma'_p\right)\otimes\imath_n\right)\bigoplus_{j=0}^{p-1}\mathrm{G}_{[\_\_]_{k=0}^{m-1}\left(\zeta_{jk}^p\otimes\imath_n\right)}}_{inplace,\ par(p,\,mpi)}\end{aligned}$$

with

$$\begin{aligned}&\gamma'_p:\mathbb{I}_{p(p-1)}\to\mathbb{I}_{p^2};\ i\mapsto i+1+\left\lfloor\tfrac{i}{p}\right\rfloor\\ &\gamma_p:\mathbb{I}_{p^2}\to\mathbb{I}_{p(p-1)};\ i\mapsto i-1-\left\lfloor\tfrac{i}{p+1}\right\rfloor\\ &\zeta_{jk}^N:\mathbb{I}_1\to\mathbb{I}_N;\ i\mapsto\begin{cases}k & \text{if } k<j\\ k+1 & \text{else}\end{cases}\end{aligned}$$

5.3.3 Σ-SPL Optimization

The root object in Example 5.5 is `DMPCompose`. `DMPCompose` is not displayed as a function, but as the operator `*DMP*` between its components to make it distinguishable from the scalar composition `*`. What is not apparent in this example is that there is not one parallel composition containing all communication and computation objects, but multiple levels of nested compositions. This is a result of converting the structure of the ruletree. Before any meaningful optimization can be applied the Σ-SPL structure has to be flattened by repeated application of the following rule.

Rule 5.10 (Flatten DMP Compose) *Repeated application of this rule flattens nested parallel compositions.*

```
DMPCompose([p,A,...,B,DMPCompose([p,X,...Y]),C,...,D)
  ⟶ DMPCompose([p,A,...,B,X,...Y,C,...,D])
```

$$\underbrace{\left(A \cdots B \underbrace{(X \cdots Y)}_{par(p,\,dmp)} C \cdots D\right)}_{par(p,\,dmp)} \longrightarrow \underbrace{(A \cdots BX \cdots YC \cdots D)}_{par(p,\,dmp)}$$

SPIRAL/DMP *Code A.57 (`DMPComposeAssoc`)*

After application of this rule the Σ-SPL formula only contains one parallel composition which contains iterative sums, communication steps, and multiplications with diagonal matrices.

Example 5.7 (Diagonal Matrices Outside Iterative Sums)

```
DMPISum(i12, 2,
  DMPScat(([i12]_2 X I16)) *
   ...
  DMPGath(([i12]_2 X I16))
) *DMP*
Diag(T(32,4,1)) *DMP*
DMPISum(i19, 2,
  DMPScat(([i19]_2 X I16)) *
   ...
  DMPGath(([i19]_2 X I16))
) ...
```

In its final state, the Σ-SPL formula may only contain communication parts and iterative sums, so Rule 5.11 pulls diagonal matrices into the iterative sums.

Rule 5.11 (Pull Diagonal Matrices into Iterative Sums) *This rule pulls diagonal matrices which are direct children of a parallel composition into adjacent parallel iterative sums.*

```
DMPCompose([p,..,DMPISum(var,domain,expr1),Diag(expr2)..)
  ⟶ DMPCompose([p,..,DMPISum(var,domain,expr1 * Diag(expr2)),..)
```

$$\underbrace{\left(\sum_{i=0}^{p-1} \mathrm{S}_{(i)_p \otimes \imath_n} \cdots \mathrm{G}_{(i)_p \otimes \imath_n}\right)}_{par(p,\, dmp)} \underbrace{\mathrm{diag}\left(f^{n \to \mathbb{C}}\right)}_{par(p,\, dmp)}$$
$$\longrightarrow \underbrace{\left(\sum_{i=0}^{p-1} \mathrm{S}_{(i)_p \otimes \imath_n} \cdots \mathrm{G}_{(i)_p \otimes \imath_n} \mathrm{diag}\left(f^{n \to \mathbb{C}}\right)\right)}_{par(p,\, dmp)}$$

SPIRAL/DMP *Code A.58 (`DiagDMPISumLeft`)*

```
DMPCompose([p,..,Diag(expr1),DMPISum(var,domain,expr2),..)
⟶ DMPCompose([p,..,DMPISum(var,domain,Diag(expr1) * expr2),..)
```

$$\underbrace{\mathrm{diag}\left(f^{n \to \mathbb{C}}\right)}_{par(p,\, dmp)} \underbrace{\left(\sum_{i=0}^{p-1} \mathrm{S}_{(i)_p \otimes \imath_n} \cdots \mathrm{G}_{(i)_p \otimes \imath_n}\right)}_{par(p,\, dmp)}$$
$$\longrightarrow \underbrace{\left(\sum_{i=0}^{p-1} \mathrm{diag}\left(f^{n \to \mathbb{C}}\right) \mathrm{S}_{(i)_p \otimes \imath_n} \cdots \mathrm{G}_{(i)_p \otimes \imath_n}\right)}_{par(p,\, dmp)}$$

SPIRAL/DMP *Code A.59 (`DiagDMPISumRight`)*

Example 5.8 (Diagonal Matrices Outside Iterative Sums) Example 5.7 after application of Rule 5.11.

```
DMPISum(i12, 2,
  DMPScat(([i12]_2 X I16)) *
   ...
  DMPGath(([i12]_2 X I16))
  Diag(T(32,4,1))
) *DMP*
DMPISum(i19, 2,
  DMPScat(([i19]_2 X I16)) *
   ...
  DMPGath(([i19]_2 X I16))
) ...
```

or

```
DMPISum(i12, 2,
  DMPScat(([i12]_2 X I16))
   ...
  DMPGath(([i12]_2 X I16))
) *DMP*
DMPISum(i19, 2,
  Diag(T(32,4,1))
  DMPScat(([i19]_2 X I16)) *
   ...
  DMPGath(([i19]_2 X I16))
) ...
```

After a diagonal matrix has been pulled into the iterative sum the sum's body does not show the correct structure. The frst and last matrices inside the sum have to be gather and scatter matrices which address the correct part of the data

vector for a certain iteraion. Therefore, two rules are required which pull diagonal matrices into the gather-scatter block.

Commuting a diagonal matrix with a scatter, or gather, matrix essentially splits it into p smaller parts. This requires an extension of the diagonal matrix' generating function. It has to be tweaked such that it selects the correct values for each iteration.

Rule 5.12 (Commute Diagonal Matrices with Scatter/Gather) *These rules reestablish gather and scatter matrices as the first and last factors of parallel iterative sums.*

```
Compose(..,DMPGath(var,domain,expr1),Diag(expr2))
⟶ Compose(..,Diag(expr2*expr1),DMPGath(var,domain,expr1))
```

$$\sum_{i=0}^{p-1} \underbrace{\left(\cdots \mathrm{G}_{(i)_p \otimes \imath_n} \operatorname{diag}\left(f^{n\to\mathbb{C}}\right)\right)}_{par(p,\,dmp)} \longrightarrow \sum_{i=0}^{p-1} \underbrace{\left(\cdots \operatorname{diag}\left(f^{n\to\mathbb{C}} \circ \left((i)_p \otimes \imath_{n/p}\right)\right) \mathrm{G}_{(i)_p \otimes \imath_n}\right)}_{par(p,\,dmp)}$$

SPIRAL/DMP *Code A.60 (`CommuteDMPGathDiag`)*

```
Compose(Diag(expr1),DMPGath(var,domain,expr2),..)
⟶ DMPGath(var,domain,expr2),Compose(Diag(expr1*expr2),..)
```

$$\sum_{i=0}^{p-1} \underbrace{\left(\operatorname{diag}\left(f^{n\to\mathbb{C}}\right) \mathrm{S}_{(i)_p \otimes \imath_n} \cdots\right)}_{par(p,\,dmp)} \longrightarrow \sum_{i=0}^{p-1} \underbrace{\left(\mathrm{S}_{(i)_p \otimes \imath_n} \operatorname{diag}\left(f^{n\to\mathbb{C}} \circ \left((i)_p \otimes \imath_{n/p}\right)\right) \cdots\right)}_{par(p,\,dmp)}$$

SPIRAL/DMP *Code A.61 (`CommuteDiagDMPScat`)*

At this point all diagonal matrices have been pulled into parallel computation steps and the outmost `DMPCompose` only contains communication steps and iterative sums.

Example 5.9 (Adjacent Iterative Sums)

```
DMPISum(i12, 2,
  DMPScat(([i12]_2 X I16)) *
   ...
  DMPGath(([i12]_2 X I16))
) *DMP*
DMPISum(i19, 2,
  DMPScat(([i19]_2 X I16)) *
   ...
  DMPGath(([i19]_2 X I16))
) ...
```

Two adjacent parallel kronecker products in the SPL formula result in such back-to-back iterative sums in Σ-SPL. All scalar code is encapsuled in such iterative sums. As scalar code optimization cannot optimize across multiple `DMPISum` objects it is preferrable to work towards a structure without multiple adjacent iterative sums. Iterative sums should only by surrounded by communication parts but not by other iterative sums.

Rule 5.13 (Merge Iterative Sums) *This rule merges two adjacent parallel iterative sums.*

DMPCompose([p,...,DMPISum(v1,d1,expr1),DMPISum(v2, d2, expr2),...)
*⟶ DMPCompose([p,...,DMPISum(var, domain, expr1 * expr2),...)*

$$\underbrace{\left(\sum_{i=0}^{p-1} \mathrm{S}_{(i)_p \otimes \imath_n}\, A\, \mathrm{G}_{(i)_p \otimes \imath_n}\right)}_{par(p,\, dmp)} \underbrace{\left(\sum_{i=0}^{p-1} \mathrm{S}_{(i)_p \otimes \imath_n}\, B\, \mathrm{G}_{(i)_p \otimes \imath_n}\right)}_{par(p,\, dmp)}$$

$$\longrightarrow \underbrace{\sum_{i=0}^{p-1} \mathrm{S}_{(i)_p \otimes \imath_n}\, A\, \mathrm{G}_{(i)_p \otimes \imath_n}\, \mathrm{S}_{(i)_p \otimes \imath_n}\, B\, \mathrm{G}_{(i)_p \otimes \imath_n}}_{par(p,\, dmp)}$$

SPIRAL/DMP *Code A.62 (`MergeDMPISums`)*

In fact this rule is slightly more complex than shown here. If sum_1 and sum_2, with the index variables v_1 and v_2, are merged, sum_2 is dropped and its children are appended to sum_1. However, as sum_2 is dropped, v_2 is not initialized anymore. So the former children of sum_2 need all occuring v_2 variables replaced by v_1 recursively.

Example 5.10 demonstrates the effect of this rule on the code-piece in Example 5.9. Repeated application of this rule leads a Σ-SPL formula with alternating iterative sums and communication parts.

Example 5.10 (Adjacent Gather and Scatter Functions) Example 5.9 after application of Rule 5.13.

```
DMPISum(i51, 2,
  DMPScat(([i51]_2 X I16)) *
   ...
  DMPGath(([i51]_2 X I16)) *
  DMPScat(([i51]_2 X I16)) *
   ...
  DMPGath(([i51]_2 X I16))
) ...
```

Rule 5.13 resulted in such DMPGath*DMPScat constructs. Gather and scatter matrices with the same generating function are transposed to each other. Their structure further implies that they are pseudo-inverse.

$$\begin{aligned} & \mathrm{G}_{f^{m\to n}} = (\mathrm{S}_{f^{m\to n}})^T \\ \Longrightarrow \quad & \mathrm{G}_{f^{m\to n}}\, \mathrm{S}_{f^{m\to n}} = \mathrm{I}_m \end{aligned}$$

Thus, the adjacent DMPGath and DMPScat inside the composition in DMPISum have no effect and can be cancelled.

Rule 5.14 (Clear Composed Gather Scatter)

Compose([..,A,DMPGath(v,dom,expr),DMPScat(var,dom,expr),B,..)
$$\longrightarrow \textit{Compose([..,A,B,..)}$$

$$\underbrace{\left(\sum_{i=0}^{p-1} \cdots A\, \mathrm{G}_{(i)_p \otimes \imath_n}\, \mathrm{S}_{(i)_p \otimes \imath_n}\, B \cdots\right)}_{par(p,\, dmp)} \longrightarrow \underbrace{\left(\sum_{i=0}^{p-1} \cdots AB \cdots\right)}_{par(p,\, dmp)}$$

SPIRAL/DMP *Code A.63 (ComposeDMPGathDMPScat)*

Σ-SPL optimization for the parts of the formula representing parallel parts of the algorithm goes together with scalar optimization. All rules are applied until no rule can be applied any more. Example 5.11 shows the Σ-SPL formula from Example 5.5 after the optimization process.

Example 5.11 (Σ-SPL) The formula from Example 5.5 after the application of Σ-SPL optimization rules.

```
DMPISum(i12, 2,
  DMPScat(([i12]_2 X I16)) *
  ISum(i13, 4,
    ISum(i16, 2,
      Scat(([i13]_4 X I2 X [i16]_2)) *
      Blk([ [ 1, 1 ], [ 1, -1 ] ]) *
      Gath((I2 X [i16]_2))
    ) *
    ISum(i17, 2,
      Scat(([i17]_2 X I2)) *
      Diag((FData(D4) o ([i17]_2 X I2))) *
      Blk([ [ 1, 1 ], [ 1, -1 ] ]) *
      Gath((I2 X [i13]_4 X [i17]_2))
    )
  ) *
  DMPGath(([i19]_2 X I16))
) *DMP*
DMPGlobalTranspose(8, AParDistr(2)) *DMP*
DMPISum(i20, 2,
  DMPScat(([i20]_2 X I16)) *
  ISum(i22, 2,
```

```
    ISum(i24, 2,
      ISum(i26, 2,
        Scat((I2 X [i26]_2 X [i24]_2 X [i22]_2)) *
        Blk([ [ 1, 1 ], [ 1, -1 ] ]) *
        Gath((I2 X [i26]_2))
      ) *
      ISum(i27, 2,
        Scat(([i27]_2 X I2)) *
        Diag((FData(D5) o ([i27]_2 X I2))) *
        Blk([ [ 1, 1 ], [ 1, -1 ] ]) *
        Gath((I2 X [i27]_2 X [i24]_2))
      )
    ) *
    ISum(i29, 4,
      Scat(([i29]_4 X I2)) *
      Diag((FData(D6) o ([i29]_4 X I2))) *
      Blk([ [ 1, 1 ], [ 1, -1 ] ]) *
      Gath((I2 X [i29]_4 X [i22]_2))
    )
  ) *
  DMPGath(([i20]_2 X I16))
) *DMP*
DMPGlobalTranspose(8, AParDistr(2)) *DMP*
DMPISum(i31, 2,
  DMPScat(([i31]_2 X I16)) *
  Prm((L(8, 2) X I2)) *
  DMPGath(([i31]_2 X I16))
)
```

5.3.4 Complex-to-Real Transformation

Example 5.11 still represents complex operations. As the output of the code generation process is C-code, which does not provide complex operations, the formula has to be transformed to operate on a vector of the double length containing the real, and imaginary, parts of the original vector as real numbers.

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{pmatrix} \longrightarrow \begin{pmatrix} x_{0,\mathrm{re}} \\ x_{0,\mathrm{im}} \\ x_{1,\mathrm{re}} \\ x_{1,\mathrm{im}} \\ x_{2,\mathrm{re}} \\ x_{2,\mathrm{im}} \\ \vdots \\ x_{n-1,\mathrm{re}} \\ x_{n-1,\mathrm{im}} \end{pmatrix}$$

This transformation requires an adaption of the formula representing the algorithm. The most significant changes have to be applied at the scalar code parts, where the actual computation takes place. SPIRAL already provides this functionality, as this is necessary for scalar code generation as well.

To perform the complex-to-real transformation the whole formula is embedded into an `RC` function and then the ruleset, which passes this function down through the formula, is called. Again, all rules are applied until no one can be applied any more. During this process the RC tag percolates through the whole formula-tree and initiates changes where required or is passed on to the objects' children.

SPIRAL/DMP's task in complex-to-real transformation mainly narrows down to inflating communication, permutation, and iterative sums by a factor of two and to pass the RC-call down to the scalar code parts. Thus, the transformation rules are relatively simple.

As mentioned before, the outmost object of a parallel Σ-SPL formula is the parallel composition. As `DMPCompose` does not store the size of its child matrices it is sufficient to pass the RC tag down.

Rule 5.15 (Parallel Compositions C→R)

```
RC(DMPCompose([p,A1,...,Ak]))
⟶ DMPCompose([p,RC(A1),...,RC(Ak)])
```

$$\underbrace{\mathrm{RC}(A_1 A_2 \cdots A_k)}_{par(p,\, dmp)} \longrightarrow \underbrace{\mathrm{RC}(A_1)\,\mathrm{RC}(A_2) \cdots \mathrm{RC}(A_k))}_{par(p,\, dmp)}$$

SPIRAL/DMP *Code A.64 (`RCDMPCompose`)*

At this stage the parallel composition can only contain iterative sums and communication parts. Also the iterative sum does nothing but passing the tag on to its child which is a scalar composition.

Rule 5.16 (Parallel Iterative Sums C→R)

```
RC(DMPISum(var,domain,expr)) ⟶ DMPISum(var,domain,RC(expr)))
```

$$\underbrace{\mathrm{RC}\left(\sum_{i=0}^{p-1} \mathrm{S}_{(i)_p \otimes \imath_n}\, A\, \mathrm{G}_{(i)_p \otimes \imath_n}\right)}_{par(p,\, dmp)}$$
$$\longrightarrow \underbrace{\sum_{i=0}^{p-1} \mathrm{RC}\left(\mathrm{S}_{(i)_p \otimes \imath_n}\, A\, \mathrm{G}_{(i)_p \otimes \imath_n}\right)}_{par(p,\, dmp)}$$

SPIRAL/DMP *Code A.65 (`RCDMPISum`)*

The complex-to-real transformation of a scalar composition is handled by SPIRAL itself, but it, again, does not do anything but passing the tag down to its children.

Thus, the RC-tag reaches the parallel gather and scatter functions `DMPGath` and `DMPScat`. This is the first spot where a small adaption is required. As the real data vector has the double length of the original one, but is still spread over the same number of processors, the gather and scatter matrices have to be infalted by a factor of two. This is achieved by appending $\otimes \imath_2$ to the matrices' generating functions.

Rule 5.17 (Parallel Gather and Scatter Matrices C→R)

1. *DMPGath*

$$\begin{aligned} &\texttt{RC(DMPGath(var,domain,expr))} \\ \longrightarrow\ &\texttt{DMPGath(var,domain,fTensor(expr, fId(2)))} \end{aligned}$$

$$\underbrace{\mathrm{RC}(\mathrm{G}_{(i)_p \otimes f^{n \to N}})}_{par(p,\, dmp)} \longrightarrow \underbrace{\mathrm{G}_{(i)_p \otimes f^{n \to N} \otimes \imath_2}}_{par(p,\, dmp)}$$

SPIRAL/DMP *Code A.66 (`RCDMPGath`)*

2. *DMPScat*

$$\begin{aligned} &\texttt{RC(DMPScat(var,domain,expr1))} \\ \longrightarrow\ &\texttt{DMPScat(var,domain,fTensor(expr, fId(2)))} \end{aligned}$$

$$\underbrace{\mathrm{RC}(\mathrm{S}_{(i)_p \otimes f^{n \to N}})}_{par(p,\, dmp)} \longrightarrow \underbrace{\mathrm{S}_{(i)_p \otimes f^{n \to N} \otimes \imath_2}}_{par(p,\, dmp)}$$

SPIRAL/DMP *Code A.67 (`RCDMPScat`)*

The other factors inside the iterative sum are scalar code parts which can now handle the transformation themselves.

The parallel composition also passed the RC-tag to the communication objects. As communication represents (non-local) permutations it is sufficient to double its size too. As mentioned in Section 5.3.2 the central permutation matrix representing the communication pattern should remain as small as possible to keep the number of communication steps low.

Thus, the inflation of the whole matrix represented by the comm object is achieved by doubling the size of the communicated blocks.

As shown in Definition 5.15 the gather and scatter matrices ${r_{jk}}^{1\to N}$ and ${w_{jk}}^{1\to N}$ only represent the selection of the block to communicate. The size of the block itself is indicated by the $\otimes \imath_n$ attached to the gather and scatter functions. So it sufficient to set the block size to $2n$ and leave the gather and scatter functions untouched.

Rule 5.18 (General Communication Steps C→R)

$$\texttt{RC(DMPComm(p,n,w\_jk,pi,piinv,r\_jk))} \longrightarrow \texttt{RC(DMPComm(p,2*n,w\_jk,pi,piinv,r\_jk))}$$

$$\underbrace{\mathrm{RC}\left(\mathrm{comm}_n^{p\circlearrowleft}\left({w_{jk}}^{1\to N},\ \pi^{mp\circlearrowleft},\ {r_{jk}}^{1\to N}\right)\right)}_{par(p,\,dmp)} \longrightarrow \underbrace{\mathrm{comm}_{2n}^{p\circlearrowleft}\left({w_{jk}}^{1\to N},\ \pi^{mp\circlearrowleft},\ {r_{jk}}^{1\to N}\right)}_{par(p,\,dmp)}$$

$$= \underbrace{\bigoplus_{j=0}^{p-1} \mathrm{S}_{[\_\_]_{k=0}^{m-1}\left({w_{jk}}^{1\to N}\otimes \imath_{2n}\right)}\ \mathrm{perm}\left(\pi^{mp\circlearrowleft}\otimes \imath_{2n}\right) \bigoplus_{j=0}^{p-1} \mathrm{G}_{[\_\_]_{k=0}^{m-1}\left({r_{jk}}^{1\to N}\otimes \imath_{2n}\right)}}_{inplace,\ par(p,\,mpi)}$$

SPIRAL/DMP *Code A.68 (RCDMPComm)*

The complex-to-real rule for `DMPGlobalTranspose` is very similar to `DMPComm`, with the difference that most of the parameters drop out because the permutation is constant.

Rule 5.19 (Parallel Matrix Transpositions C→R)

$$\texttt{RC(DMPGlobalTranspose(n,pv))} \longrightarrow \texttt{RC(DMPGlobalTranspose(2*n,pv))}$$

$$\mathrm{comm}_n^{p\circlearrowleft}\left(\zeta_{jk}^p,\ \gamma_p\circ \ell_p^{p^2}\circ \gamma_p',\ \zeta_{jk}^p\right) \longrightarrow \mathrm{comm}_{2n}^{p\circlearrowleft}\left(\zeta_{jk}^p,\ \gamma_p\circ \ell_p^{p^2}\circ \gamma_p',\ \zeta_{jk}^p\right)$$

$$= \underbrace{\bigoplus_{j=0}^{p-1} \mathrm{S}_{[\_\_]_{k=0}^{m-1}\left(\zeta_{jk}^p\otimes \imath_{2n}\right)}\ \mathrm{perm}\left(\left(\gamma_p\circ \ell_p^{p^2}\circ \gamma_p'\right)\otimes \imath_{2n}\right) \bigoplus_{j=0}^{p-1} \mathrm{G}_{[\_\_]_{k=0}^{m-1}\left(\zeta_{jk}^p\otimes \imath_{2n}\right)}}_{inplace,\ par(p,\,mpi)}$$

with

$$\gamma_p' : \mathbb{I}_{p(p-1)} \to \mathbb{I}_{p^2};\ i \mapsto i+1+\left\lfloor \tfrac{i}{p} \right\rfloor$$
$$\gamma_p : \mathbb{I}_{p^2} \to \mathbb{I}_{p(p-1)};\ i \mapsto i-1-\left\lfloor \tfrac{i}{p+1} \right\rfloor$$
$$\zeta_{jk}^N : \mathbb{I}_1 \to \mathbb{I}_N;\ i \mapsto \begin{cases} k & \text{if } k<j \\ k+1 & \text{else} \end{cases}$$

SPIRAL/DMP *Code A.69 (RCDMPGlobalTranspose)*

5.3.5 Summary

After the complex-to-real transformation is finished, and thus, the Σ-SPL formula does not contain RC-tags anymore, the optimization rules from Section 5.3.3 are run again. This is done because it is possible that new optimization potential has arisen during the transformation.

As an example the final, optimized, complex-to-real transformed Σ-SPL formula of Example 5.11 is displayed in Example 5.12

Example 5.12 (Σ-SPL) The formula from Example 5.11 after complex-to-real transformation and Σ-SPL optimization.

```
DMPISum(i12, 2,
  DMPScat(([i12]_2 X I32)) *
  ISum(i13, 4,
    ISum(i16, 2,
      Scat(([i13]_4 X I2 X [i16]_2 X I2)) *
      Blk([ [ 1, 0, 1, 0 ], [ 0, 1, 0, 1 ], [ 1, 0, -1, 0 ], [ 0, 1, 0, -1 ] ]) *
      Gath((I2 X [i16]_2 X I2))
    ) *
    ISum(i17, 2,
      Scat(([i17]_2 X I4)) *
      RCDiag((FData(D4) o ([i17]_2 X I2))) *
      Blk([ [ 1, 0, 1, 0 ], [ 0, 1, 0, 1 ], [ 1, 0, -1, 0 ], [ 0, 1, 0, -1 ] ]) *
      Gath((I2 X [i13]_4 X [i17]_2 X I2))
    )
  ) *
  DMPGath(([i12]_2 X I32))
) *DMP*
DMPGlobalTranspose(16, AParDistr(2)) *DMP*
DMPISum(i20, 2,
  DMPScat(([i20]_2 X I32)) *
  ISum(i22, 2,
    ISum(i24, 2,
      ISum(i26, 2,
        Scat((I2 X [i26]_2 X [i24]_2 X [i22]_2 X I2)) *
        Blk([ [ 1, 0, 1, 0 ], [ 0, 1, 0, 1 ], [ 1, 0, -1, 0 ], [ 0, 1, 0, -1 ] ]) *
        Gath((I2 X [i26]_2 X I2))
      ) *
      ISum(i27, 2,
        Scat(([i27]_2 X I4)) *
        RCDiag((FData(D5) o ([i27]_2 X I2))) *
        Blk([ [ 1, 0, 1, 0 ], [ 0, 1, 0, 1 ], [ 1, 0, -1, 0 ], [ 0, 1, 0, -1 ] ]) *
        Gath((I2 X [i27]_2 X [i24]_2 X I2))
      )
    ) *
    ISum(i29, 4,
      Scat(([i29]_4 X I4)) *
      RCDiag((FData(D6) o ([i29]_4 X I2))) *
      Blk([ [ 1, 0, 1, 0 ], [ 0, 1, 0, 1 ], [ 1, 0, -1, 0 ], [ 0, 1, 0, -1 ] ]) *
      Gath((I2 X [i29]_4 X [i22]_2 X I2))
    )
  ) *
  DMPGath(([i20]_2 X I32))
) *DMP*
DMPGlobalTranspose(16, AParDistr(2)) *DMP*
DMPISum(i31, 2,
  DMPScat(([i31]_2 X I32)) *
  Prm(((L(8, 2) X I2) X I2)) *
  DMPGath(([i31]_2 X I32))
)
```

5.4 Code Generation

The Σ-SPL formula in Example 5.12 displays a structure similar to a computer program's source code. The scalar iterative sums represent loops. The gather and scatter matrices describe which elements of the data array are addressed in a certain iteration. The multiplication with a diagonal matrix is implemented by an elementwise multiplication with an array containing the matrix' values. The functionality of converting a scalar Σ-SPL formula to iCode is provided by SPIRAL through the Function

```
CodeSums(bksize, sums).
```

Similar to the conversion from SPL terminals to Σ-SPL this function calls the `.code()` function of the Σ-SPL formula's root object. Each object is responsible to trigger this conversion for its child objects. The parameter `bksize` determines the level of scalar loop unrolling.

A general introduction to iCode in SPIRAL is given in [51]. The following section will introduce SPIRAL/DMP's extensions to code generation for computation parts. Section 5.4.2 will cover code generation of communication steps.

5.4.1 Parallel Computation

ICode covers the operations, and methods, available in a structured programming language, but displays them in a function-like way. Therefore, it is programming language independent, but can easilly be mapped to, e. g., Fortran or C source code. Furthermore, it is easier to formulate and apply optimization rules on iCode than on C-code. Example 5.13 shows a line of C-code with its representation in iCode.

Example 5.13 (Scalar iCode) The iCode object

```
assign(nth(T,i),add(nth(T,sub(i,1)),nth(T,sub(i,2))))
```

would be unparsed to the following line in C-code.

```
T[i] = T[i-1] + T[i-2];
```

The most important iCode objects available in scalar SPIRAL are listed in Table 5.4.1, together with its translations to C-code.

The top-level Σ-SPL object is the parallel composition. `DMPCompose` translates to a `chain` of its children's codes. As a the mathematical formula symbolizes a right to left application of matrices, but the code is generally executed left to right, the composed elements have to be reversed.

Expressions

iCode	C-code
`nth(x,i)`	`x[i]`
`add(x,y)`	`x + y`
`sub(x,y)`	`x - y`
`mul(x,y)`	`x * y`
`div(x,y)`	`x / y`
`imod(x,y)`	`x % y`

Commands

iCode	C-code
`assign(x,y)`	`x = y;`

`loop(i, [ j .. k ], expr)`	`for(i=j;i<=k;i++) {`
	`  expr`
	`}`

`chain(expr1,expr2,...)`	`expr1 expr2 ...`

Declarations

iCode	C-code
`decl([var1,var2,...],expr)`	declare `var1`, `var2`, ...; `expr`

Table 5.3: SPIRAL iCode objects and their translation into C-code.

Rule 5.20 (Code Generation for Parallel Compositions)

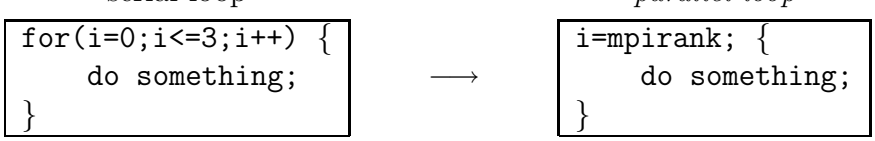
```
DMPCompose([p,A1,A2,...,Ak]).code()

⟶  chain (
        Ak.code(),
        ...
        A2.code(),
        A1.code()
    )
```

SPIRAL/DMP *Code A.75 (`DMPCompose.code()`)*

As explained in Section 5.3 the Σ-SPL rules lead to a state where parallel compositions only contain communication steps and parallel iterative sums `DMPISum`.

Thus, the only child objects representing parallel computation are of the type `DMPISum`.

To generate code for a parallel computation part the distributed iterative sum, together with its gather and scatter matrices must handle the code generation of its body such that the code addresses the correct parts of the data vector. As discussed in Section 5.3 a parallel loop is very similar to a scalar loop, with the difference that each processor executes one iteration of the loop.



Spiral/dmp defines the iCode-object representing a parallel loop as follows.

Definition 5.16 (Parallel Loop) *`mpirank` is a variable containing the current processor's rank, set upon the initialization phase of the program.*

iCode		***C-code***
dmploop(loopvar, range, cmd, pv)	:=	{ *loopvar = mpirank;* { *cmd* } }

Spiral/dmp *Code A.70 (`dmploop`)*

`dmploop` exactly represents the semantics of the `DMPISum` Σ-Spl object. So the rule for generating a parallel code segment in iCode is very simple.

Rule 5.21 (Code Generation for Iterative Sums)

DMPISum(var,domain,expr).code(y,x)
→ *dmploop(var,domain,expr.code(y,x))*

Spiral/dmp *Code A.74 (`DMPISum.code()`)*

the `x` and `y` parameters to the `.code()` functions define the input, and output, variables for the operation. If they are both the same *inplace* code is generated, and therefore, the function `.ipcode()` is called.

Only a scalar composition `Compose` can be the child of a parallel iterative sum. The composition contains the matrices representing the operations to be executed. The rules in Section 5.3.3 assured that the first and last of the composed elements inside a parallel iterative sum are always `DMPGath` and `DMPScat`. As explained before, they do not require any code to be generated, so the `Compose.code()` function was altered to ignore `DMPGath` and `DMPScat` objects and only generate code for the remaining objects.

Rule 5.22 (Code Generation for Scalar Compositions)

```
Compose([DMPScat(...),A1,A2,...,Ak,DMPGath(...)]).code()

    ⟶  chain (
            Ak.code(),
            ...
            A2.code(),
            A1.code()
        )
```

SPIRAL/DMP *Code A.75 (`DMPCompose.code()`)*

These rules, together with SPIRAL's code generation functionality, are sufficient to generate mpi parallel code for transforms which do not require communication.

Example 5.14 (Code Generation for Parallel Computation) This example shows iCode generated for the transform $\underbrace{\mathrm{I}_4 \otimes F(4)}_{\mathrm{par}(4,\,\mathrm{dmp})}$.

```
dmploop(i10, [ 0 .. 3 ],
   decl([ t30, t29, t31, t28, t32, t27, t26, t25 ],
      chain(
         assign(t25, add(nth(X, 0), nth(X, 4))),
         assign(t26, add(nth(X, 2), nth(X, 6))),
         assign(t27, add(nth(X, 1), nth(X, 5))),
         assign(t28, add(nth(X, 3), nth(X, 7))),
         assign(t29, sub(nth(X, 0), nth(X, 4))),
         assign(t30, sub(nth(X, 3), nth(X, 7))),
         assign(t31, sub(nth(X, 1), nth(X, 5))),
         assign(t32, sub(nth(X, 2), nth(X, 6))),
         assign(nth(Y, 0), add(t25, t26)),
         assign(nth(Y, 1), add(t27, t28)),
         assign(nth(Y, 4), sub(t25, t26)),
         assign(nth(Y, 5), sub(t27, t28)),
         assign(nth(Y, 2), sub(t29, t30)),
         assign(nth(Y, 3), add(t31, t32)),
         assign(nth(Y, 6), add(t29, t30)),
         assign(nth(Y, 7), sub(t31, t32))
      )
   )
)
```

The corresponding C-program computes this transform with the input array `x` and stores the output to `y`. The source code includes the function `init_sub` which, in this case, only sets the `mpirank` and `mpisize` variables for later use in the computation. The initialization function has to be called prior to the actual calculation.

```
/* [ 32, 32 ] */
#include <mpi.h>

int mpirank, mpisize;

void init_sub ();
void sub(double *Y, double *X) {
    { /* dmploop */
      int i10 = mpirank;
      double t30, t29, t31, t28, t32, t27, t26, t25;
      t25 = (X[0] + X[4]);
      t26 = (X[2] + X[6]);
      t27 = (X[1] + X[5]);
      t28 = (X[3] + X[7]);
      t29 = (X[0] - X[4]);
      t30 = (X[3] - X[7]);
      t31 = (X[1] - X[5]);
      t32 = (X[2] - X[6]);
      Y[0] = (t25 + t26);
      Y[1] = (t27 + t28);
      Y[4] = (t25 - t26);
      Y[5] = (t27 - t28);
      Y[2] = (t29 - t30);
      Y[3] = (t31 + t32);
      Y[6] = (t29 + t30);
      Y[7] = (t31 - t32);
    }
}
void init_sub() {
    MPI_Comm_size(MPI_COMM_WORLD, &mpisize);
    MPI_Comm_rank(MPI_COMM_WORLD, &mpirank);
}
```

5.4.2 Communication

The code generation for communication parts is distinct from generating parallel computation parts. For the latter one SPIRAL's own code generation was utilized, but steered to address the correct parts of the data array. As SPIRAL offered no support for MPI communication yet, the communication steps have to implemented from scratch by SPIRAL/DMP.

Code generation for the general communication object `DMPComm` will be introduced first. According to Definition 5.15 the Σ-SPL representation of a general communication kernel

```
DMPComm(p, n, w_jk, pi, piinv, r_jk, j, k, AParDistr(p))
```

is

$$\mathrm{comm}_n^{p\circlearrowleft}\left({w_{jk}}^{1\to N},\ \pi^{mp\circlearrowleft},\ {r_{jk}}^{1\to N}\right) = \underbrace{= \bigoplus_{j=0}^{p-1} \mathrm{S}_{[\_\_]_{k=0}^{m-1}\left({w_{jk}}^{1\to N}\otimes \imath_n\right)}\ \mathrm{perm}\left(\pi^{mp\circlearrowleft}\otimes \imath_n\right) \bigoplus_{j=0}^{p-1} \mathrm{G}_{[\_\_]_{k=0}^{m-1}\left({r_{jk}}^{1\to N}\otimes \imath_n\right)}}_{\text{inplace, par}(p,\text{mpi})}.$$

The implementation of the `DMPComm` object can be outlined as follows

1. **Gather** communicated elements from `x` to `t1` according to ${r_{jk}}^{1\to N}$.
2. **Communicate** elements from `t1` to `t2` according to $\pi^{mp\circlearrowleft}$.
3. **Scatter** communicated elements from `t2` to `x` according to ${w_{jk}}^{1\to N}$.
4. **Copy** all elements from `x` to `y`.

This procedure reflects a communication approach as illustrated in Example 5.1. The gather-step copies the data elements which have to be communicated to a temporary array. The reduced data vector `t1` allows the direct application of the communication pattern $\pi^{mp\circlearrowleft}$. The code for this communication is generated by `dmpcomm`. The resulting data is stored to `t2`. After the communication the scatter-step restores the communicated data from `t2` to `x`. Finally the whole array `x` is copied to the output vector `y`.

After the communication the fixpoints are still resident in `x`. This is the reason why, after the communication, the data is copied to `x` before writing to the output vector `y`. If the data would immediately be copied to the output vector, the fixpoints would be missing. Inplace communication can be generated by skipping the final copy-step. As there are no restrictions for the communication pattern this approach to code generation for communication is very versatile.

Rule 5.23 (Code Generation for General Communication)

DMPComm(p,n,w_jk,pi,piinv,r_jk, j, k, AParDistr (p)) $\longrightarrow$

```
chain(
  decl([t1,t2],
    DMPIterDirectSum(self.j,self.j.range,
      IterVStack(self.k,self.k.range,Gath(self.r_jk))).sums().code(t1,x)
    dmpcomm(t2, t1, p, n, pi, piinv)
    DMPIterDirectSum(self.j,self.j.range,
      IterHStack(self.k,self.k.range,Scat(self.w_jk))).sums().code(x,t2) ),
  dmploop(dmploopix, p, loop(i,Rows(self)/p, assign(nth(y,i),nth(x,i) ) ) ) )
```

SPIRAL/DMP *Code A.77 (DMPComm.code())*

The core of the `DMPComm.code` function is the `dmpcomm` code object. This object, and it's C-unparser, are presented in Code A.71. The parameters to `dmpcomm` are the output, and input, arrays `loc1` and `loc2`, the number of processes `p`, the blocksize `n`, the permutation pattern `pi`, and the inverse communication pattern `piinv`.

The code generated by `dmpcomm` creates non-blocking send and recieve statements[1] which exactly represent the reduced communication matrix $\pi^{mp\circlearrowleft} \otimes \imath_n$. Non-blocking communication has been chosen for the general communication implementation because this way the generator does not have to care about the order, in which the send and recieve statements are issued.

Upon code generation the communication pattern is evaluated to the lists `pi` and `piinv`. At runtime these lists are parsed to perform the communication. This way no complicated permutation evaluations are required in the final program.

Example 5.15 (dmpcomm) This example displays the code generated for $\mathrm{L}_4^{16} \otimes \mathrm{I}_{32}$ on 4 processors.

`dmpcomm(Y, X, 4, 32, [L(16,4)], [L(16,4)])` $\longrightarrow$

```
{
  MPI_Request reqs[2][3];
  int mpii;
  static commpat    pi[4][3] = {{{1, 0}, {2, 0}, {3, 0}}, {{0, 0}, {2, 1}, {3, 1}},
                                {{0, 1}, {1, 1}, {3, 2}}, {{0, 2}, {1, 2}, {2, 2}}};
  static commpat piinv[4][3] = {{{1, 0}, {2, 0}, {3, 0}}, {{0, 0}, {2, 1}, {3, 1}},
                                {{0, 1}, {1, 1}, {3, 2}}, {{0, 2}, {1, 2}, {2, 2}}};
  for(mpii=0;mpii<3;mpii++){
    MPI_Isend(X+32*mpii,
              32,
              MPI_DOUBLE,
              piinv[mpirank][mpii].proc,
              piinv[mpirank][mpii].offset,
              MPI_COMM_WORLD,
              reqs[0]+mpii);
    MPI_Irecv(Y+32*mpii,
              32,
              MPI_DOUBLE,
              pi[mpirank][mpii].proc,
              mpii,
              MPI_COMM_WORLD,
              reqs[1]+mpii);
  }
  MPI_Waitall(3, reqs[0], MPI_STATUSES_IGNORE);
  MPI_Waitall(3, reqs[1], MPI_STATUSES_IGNORE);
}
```

Because of its flexibility this implementation lacks performance in some ways. Especially the fact that two temporary arrays are required and that data elements are locally copied up to three times make this implementation sub-optimal.

As introduced in Section 3.2.1 there are better communication methods for certain cases, especially the global matrix transposition. In Section 5.2.3 the DFT's

[1] `MPI_Isend` and `MPI_Irecv`.

decompostion has been steered to generate communication steps which actually represent matrix transpositions. Thus, much more streamlined code can be generated for `DMPGlobalTranspose`.

Rule 5.24 (Code Generation for the Global Matrix Transposition)

1. *`DMPGlobalTranspose(n,pv).ipcode(x)` (inplace code generation)* $\longrightarrow$

```
dmpglobaltranspose(x,x,pv,n)
```

2. *`DMPGlobalTranspose(n,pv).code(y,x)`* $\longrightarrow$

```
chain(
  dmpglobaltranspose(x,x,pv,n),
  loop(i,Rows(self)/p,assign(nth(y,i),nth(x,i)))
)
```

SPIRAL/DMP *Code A.78 (`DMPGlobalTranspose.code()`)*

`DMPGlobalTranspose.ipcode()` simply generates one iCode object `dmpglobaltranspose`. In the case of out-of-place code generation the resulting array is copied to the output array after the transposition.

`dmpglobaltranspose` implements the pairwise direct total exchange algorithm as introduced in Section 3.2.2. The declaration of `dmpglobaltranspose`, together with its C-unparser, are listed in Code A.72.

Example 5.16 (dmpglobaltranspose) This example displays the code generated for $L_4^{16} \otimes I_{64}$ on 4 processors.

```
dmpglobaltranspose(X, X, AParDistr(4), 32)  ⟶
```

```
{
  int mpii;
  MPI_Status stat;
  for(mpii=1;mpii<4;mpii++){
    MPI_Sendrecv_replace(X+64*(mpii^mpirank),
                         64,
                         MPI_DOUBLE,
                         mpii^mpirank,
                         0,
                         mpii^mpirank,
                         0,
                         MPI_COMM_WORLD,
                         &stat);
  }
}
```

These rules, finally, allow the generation of parallel C/MPI code for parallel DFTs. Code B.1 shows the iCode for the Σ-SPL formula in Example 5.12, Code B.2 the unparsed C-code.

5.5 Runtime Environment

Spiral relies on runtime measurements to figure out which algorithms and which of their implementations perform well. However, launching a parallel program is a very sophisticated task. A *job* has to be submitted to the batch system. When the requested resources are free the batch system launches the job on some nodes. The job can be started immediately, a few minutes or hours later, or even a few days or weeks later. Depending on the batch system the user has more or less control over the node assignment, i. e., which processors are chosen on which nodes. These issues would make the process of measuring parallel runtime lengthy and unreliable.

A workaround for these problems is using *interactive jobs*. Upon the start of an interactive job, the batch system figures out whether the requested resources are available or not. If they are not available the program immediately terminates with a corresponding error message. If the resources are available a new shell is opened on one of the nodes assigned by the batch system. At this point, the nodes assigned to the user are exclusively reserved and parallel programs can be started on these nodes manually.

One advantage of this method is that the node configuration can be inspected, and the user can decide whether it fits his needs or not. Furthermore it is assured that jobs will run immediately, because the resources are already reserved.

Spiral/dmp has to be started within the framework of such an interactive job. The environment variable `$JOBID` is expected to hold the job id of the current shell's Mpi job. The job ids are required to identify multiple instances of Spiral/dmp running simultaneously. Each instance creates the directory `$HOME/tmp-spiral/$JOBID` to store its working data. This is done because some intermediate data, e. g., the executables for the runtime measurement process, have to be accessible on all nodes as not the whole file system tree but only specific directories are shared among the nodes. Usually `/home` is one of them. The temporary directory `/tmp`, which is used by Spiral to store intermediate files, is never shared between the nodes to limit network traffic. Furthermore, as two instance of Spiral/dmp must have different job ids, their temporary directories are seperated, so it is impossible that they overwrite each other's data.

Spiral/dmp can be launched at any node of the parallel computer. If Spiral/dmp has to measure a certain piece of code, the respective compilation takes place on the node the main program is running on. As the binary is stored in a location that is shared among all network nodes, it is possible to issue the measurements with a simple command like

```
mpirun -np [numofprocs] -hostfile [hostfile] [binary] [binaryopts] > [outfile].
```

The hostfile is a text file containing one hostname per line. The number of hosts has to match or exceed the MPI jobsize. Usually batch systems create such a file upon the initialization of the interactive shell to provide the user with information about which nodes have been reserved for him. Before launching SPIRAL/DMP this file has to be copied to `$HOME/$JOBID/machines`.

The file `spiral/spiral.conf` contains the definition of variables containing the information about the exact statements necessary to launch jobs on the given system. An example of the SPIRAL/DMP section of this file is given in Table 5.4.

```
...
[GROUP c.mpi]
  matrix_lib = $spiral_dir/timer/src/mpi_compute_matrix.o
  compiler = mpicc
  linker = mpicc
  compiler_invocation = %cmd  -I$spiral_dir %flags -c %target_src -o %obj

  [PROFILE c.mpi.mpich]
    compiler_flags = -O3 -fomit-frame-pointer -malign-double -fstrict-aliasing
    test_invocation = /usr/local/ibgd/mpi/osu/gcc/mvapich-0.9.5/bin/mpirun_rsh \
        -rsh -np %numofprocs_mpi -hostfile $tmp_dir/machines %exe %flags
...
```

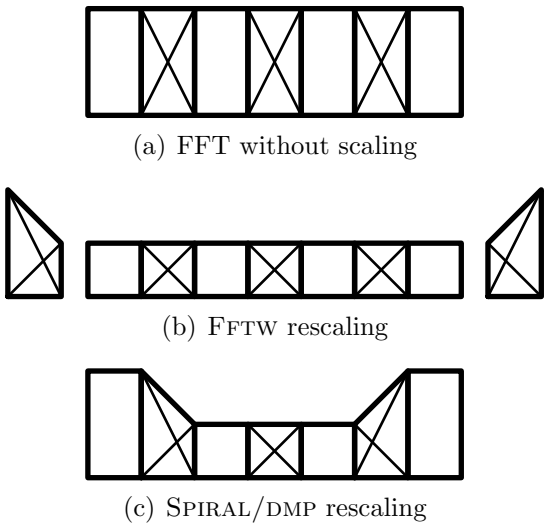
Table 5.4: `spiral/spiral.conf` example

A transform's generated source code only contains the subroutine representing the transform, and its initialization function. This file has to be linked with a pre-generated *stub library* that provides generally used functions like timing and storage allocations.

The normal, scalar, stub library had to be altered to meet the requirements of MPI parallel computation. Explaining these changes in detail would lead too far, but especially `compute_matrix.c` and `time.c` had to be modified to `mpi_compute_matrix.c` and `mpi_time.c` (Code B.3 and B.4). Thus, SPIRAL/DMP is able to create a stub library for parallel execution, which supports distributed matrices and provides reliable parallel timing routines.

5.6 Rescaling

Performance optimization of FFTs often requires down-scaling, i. e., computations are to be carried out on a smaller number of processors than the user provided. Normally, this requires two additional data redistribution steps, one prior and one posterior of the FFT calculation. In the following, a method to generate FFT code which implements re-scaling without the requirement of additional communication is presented.



(a) FFT without scaling

(b) FFTW rescaling

(c) SPIRAL/DMP rescaling

Figure 5.2: Plain and rescaled execution of a transform which requires three communication steps, e. g., a one-dimensional FFT. Crossed blocks represent communication steps, uncrossed ones computation. Ascending and descending trapezoidal blocks symbolize the rescaling steps. The small blocks are rescaled blocks. The execution order is from left to right.

Advantages of Self-scaling. The data is not redistributed explicitly before and after the FFT calculation but intertwined with the FFT's communication steps, i. e., additional communication overhead is avoided.

Down-scaling is performed within the first occurring communication step, while up-scaling is performed within the last communication step. Hence, all encapsulated communication steps profit from the reduced communication effort as illustrated in Figure 5.2.

Computational parts of the FFT to be carried out prior and/or posterior of the first and last communication step are performed on the maximum number of available processors and thus benefit from the larger granularity of the respective computational blocks.

Formula Manipulation Rules for Re-scaling. First of all a formal indicator is needed, which tags a formula or subexpression to be a re-scaling operation.

For this purpose the $par(*,*)$ tag from Definition 5.1 is extended. Down-scaling from p to q parallel instances is denoted by $par(q \swarrow p, *)$ and the corresponding up-scaling by $par(p \nwarrow q, *)$ whereby $q < p$. $par(p \nwarrow q \swarrow p, *)$ indicates that both down- and up-scaling still have to be applied to the tagged expression.

The rewriting rules in Table 5.5 are used to automatically derive a self re-scaling algorithm. Table 5.6 illustrates how these rules may be used to yield a self scaling one-dimensional parallel FFT formula. The numbers indicate which manipulation rules have been applied to the formula.

$$\underbrace{A}_{\mathrm{par}(p,*)}\ \underbrace{B}_{\mathrm{par}(p,*)} \rightarrow \underbrace{AB}_{\mathrm{par}(p,*)} \tag{5.10}$$

$$\underbrace{A}_{\mathrm{par}(p,*)} \rightarrow \underbrace{A}_{\mathrm{par}(p^{\nwarrow}q_{\swarrow}p,*)} \quad \forall q : q|p \tag{5.11}$$

$$\underbrace{\mathrm{DFT}_{mn}}_{\mathrm{par}(p^{\nwarrow}q_{\swarrow}p,*)} \rightarrow \underbrace{(\mathrm{DFT}_m \otimes \mathrm{I}_n)}_{\mathrm{par}(p^{\nwarrow}q,*)} \underbrace{\mathrm{T}^{mn}_n(\mathrm{I}_m \otimes \mathrm{DFT}_n)}_{\mathrm{par}(q,*)} \underbrace{\mathrm{L}^{mn}_m}_{\mathrm{par}(q_{\swarrow}p,*)} \tag{5.12}$$

$$\underbrace{(A_m \otimes \mathrm{I}_n)}_{\mathrm{par}(p^{\nwarrow}q,*)} \rightarrow \underbrace{L^{mn}_m}_{\mathrm{par}(p^{\nwarrow}q,*)} \underbrace{(\mathrm{I}_n \otimes A_m)L^{mn}_n}_{\mathrm{par}(q,*)} \tag{5.13}$$

$$\underbrace{(A_m \otimes \mathrm{I}_n)}_{\mathrm{par}(q_{\swarrow}p,*)} \rightarrow \underbrace{L^{mn}_m(\mathrm{I}_n \otimes A_m)}_{\mathrm{par}(q,*)} \underbrace{L^{mn}_n}_{\mathrm{par}(q_{\swarrow}p,*)} \tag{5.14}$$

$$\underbrace{\mathrm{L}^{mn}_m}_{\mathrm{par}(p,*)} \rightarrow \underbrace{\left(\mathrm{I}_p \otimes \mathrm{L}^{mn/p}_{m/p}\right)\left(\mathrm{L}^{p^2}_p \otimes \mathrm{I}_{mn/p^2}\right)\left(\mathrm{I}_p \otimes \mathrm{L}^n_p \otimes \mathrm{I}_{m/p}\right)}_{\mathrm{par}(p,*)} \tag{5.15}$$

$$\underbrace{\mathrm{L}^{mn}_m}_{\mathrm{par}(q_{\swarrow}p,*)} \rightarrow \underbrace{\left(\mathrm{I}_q \otimes \mathrm{I}_{p/q} \otimes \mathrm{L}^{mn/p}_{m/p}\right)}_{\mathrm{par}(q,*)} \underbrace{\left(\mathrm{L}^{p^2}_p \otimes \mathrm{I}_{mn/p^2}\right)}_{\mathrm{par}(q_{\swarrow}p,*)} \underbrace{\left(\mathrm{I}_p \otimes \mathrm{L}^n_p \otimes \mathrm{I}_{m/p}\right)}_{\mathrm{par}(p,*)} \tag{5.16}$$

$$\underbrace{\mathrm{L}^{mn}_m}_{\mathrm{par}(p^{\nwarrow}q,*)} \rightarrow \underbrace{\left(\mathrm{I}_p \otimes \mathrm{L}^{mn/p}_{m/p}\right)}_{\mathrm{par}(p,*)} \underbrace{\left(\mathrm{L}^{p^2}_p \otimes \mathrm{I}_{mn/p^2}\right)}_{\mathrm{par}(p^{\nwarrow}q,*)} \underbrace{\left(\mathrm{I}_q \otimes \mathrm{I}_{p/q} \otimes \mathrm{L}^n_p \otimes \mathrm{I}_{m/p}\right)}_{\mathrm{par}(q,*)} \tag{5.17}$$

Table 5.5: Parallel Rescaling Rules.

Estimation of the Communication Effort. The communication parts of the formula in Table 5.6 are

$$\underbrace{\left(\mathrm{L}^{p^2}_p \otimes \mathrm{I}_{mn/p^2}\right)}_{\mathrm{par}(p^{\nwarrow}q,*)} \underbrace{\left(\mathrm{L}^{q^2}_q \otimes \mathrm{I}_{mn/q^2}\right)}_{\mathrm{par}(q,*)} \underbrace{\left(\mathrm{L}^{p^2}_p \otimes \mathrm{I}_{mn/p^2}\right)}_{\mathrm{par}(q_{\swarrow}p,*)}$$

opposed to the usual implementation

$$\underbrace{\left(\mathrm{L}^{p^2}_p \otimes \mathrm{I}_{mn/p^2}\right)}_{\mathrm{par}(p,*)} \underbrace{\left(\mathrm{L}^{p^2}_p \otimes \mathrm{I}_{mn/p^2}\right)}_{\mathrm{par}(p,*)} \underbrace{\left(\mathrm{L}^{p^2}_p \otimes \mathrm{I}_{mn/p^2}\right)}_{\mathrm{par}(p,*)}.$$

A straightforward implementation of $\mathrm{L}^{p^2}_p \otimes \mathrm{I}_n$ requires the communication of $np(p-1)$ data elements. With the same communication effort, a rescaling redistribution can be performed, which reduces the cost of additional all-to-all communication parts. These require only $np(q-1)$ data elements to be communicated, i. e., a reduction by $(q-1)/(p-1)$ is achieved.

Even, down-scaling to just one processor is possible, if enough main memory is available to store the whole global data vector D locally. In this case, all communication parts except the first and the last ones are eliminated.

$$
\begin{array}{ll}
\underbrace{\mathrm{DFT}_{mn}}_{\mathrm{par}(p,*)} & \overset{(5.11)}{\longrightarrow} \quad \underbrace{\mathrm{DFT}_{mn}}_{\mathrm{par}(p\nwarrow q\swarrow p,*)} \\
& \overset{(5.12)}{\longrightarrow} \quad \underbrace{(\mathrm{DFT}_m \otimes \mathrm{I}_n)}_{\mathrm{par}(p\nwarrow q,*)} \underbrace{\mathrm{T}^{mn}_n (\mathrm{I}_m \otimes \mathrm{DFT}_n)}_{\mathrm{par}(q,*)} \underbrace{\mathrm{L}^{mn}_m}_{\mathrm{par}(q\swarrow p,*)} \\
& \overset{\substack{(5.10)\\(5.13)}}{\longrightarrow} \quad \underbrace{L^{mn}_m}_{\mathrm{par}(p\nwarrow q,*)} \underbrace{(\mathrm{I}_n \otimes \mathrm{DFT}_m) L^{mn}_n \mathrm{T}^{mn}_n (\mathrm{I}_m \otimes \mathrm{DFT}_n)}_{\mathrm{par}(q,*)} \underbrace{\mathrm{L}^{mn}_m}_{\mathrm{par}(q\swarrow p,*)} \\
& \overset{\substack{(5.10)\\(5.15)\\(5.16)\\(5.17)}}{\longrightarrow} \quad \underbrace{\left(\mathrm{I}_p \otimes \mathrm{L}^{mn/p}_{m/p}\right)}_{\mathrm{par}(p,*)} \underbrace{\left(\mathrm{L}^{p^2}_p \otimes \mathrm{I}_{mn/p^2}\right)}_{\mathrm{par}(p\nwarrow q,*)} \underbrace{\left(\mathrm{I}_q \otimes \mathrm{I}_{p/q} \otimes \mathrm{L}^n_p \otimes \mathrm{I}_{m/p}\right)(\mathrm{I}_n \otimes \mathrm{DFT}_m)}_{\mathrm{par}(q,*)} \\
& \qquad \underbrace{\left(\mathrm{I}_q \otimes \mathrm{L}^{mn/q}_{m/q}\right)\left(\mathrm{L}^{q^2}_q \otimes \mathrm{I}_{mn/q^2}\right)\left(\mathrm{I}_q \otimes \mathrm{L}^n_q \otimes \mathrm{I}_{m/q}\right)\mathrm{T}^{mn}_n (\mathrm{I}_m \otimes \mathrm{DFT}_n)}_{\mathrm{par}(q,*)} \\
& \qquad \underbrace{\left(\mathrm{I}_q \otimes \mathrm{I}_{p/q} \otimes \mathrm{L}^{mn/p}_{m/p}\right)}_{\mathrm{par}(q,*)} \underbrace{\left(\mathrm{L}^{p^2}_p \otimes \mathrm{I}_{mn/p^2}\right)}_{\mathrm{par}(q\swarrow p,*)} \underbrace{\left(\mathrm{I}_p \otimes \mathrm{L}^n_p \otimes \mathrm{I}_{m/p}\right)}_{\mathrm{par}(p,*)}
\end{array}
$$

Table 5.6: Self-scaling One-dimensional Parallel FFT Formula.

This down-scaling approach means a tradeoff between communication and computation effort. Whether it makes sense to apply this optimization depends on the transform which is to be calculated and the ratio of scalar and network performance. Note that down-scaling is only applied if it yields a speedup for the whole FFT application.

Redistribution Data Layout. After choosing a number of processes to rescale to, there is still to decide which q of the p processors to use for calculation.

For example, assume a distributed memory parallel system with 2 processors per node. A given program is initially started on four processors p_0, p_1, p_2, p_3 where p_0 and p_1 are located on one node, p_2 and p_3 on another one. The data block d_i locally resides in processor p_i's memory (i. e., $p_i[d_i] \quad \forall i = 0 \ldots 3$). $[d_0, d_1, d_2, d_3]$ represents the global data vector D.

For this particular setting, two obvious possibilities for rescaling would be either (i) $(p_0[d_0, d_1], p_2[d_2, d_3])$ or (ii) $(p_0[d_0, d_1], p_1[d_2, d_3])$. One advantage of redistribution layout (i) is that only shared memory communication is required for the rescaling process. Thus, any communication taking place in-between the two rescaling steps consists of pure inter-node communication. Additionally, there is one processor free per node which usually increases scalar performance.

By contrast, in case (ii) both processes are merged to one node. This requires

more inter-node communication during the redistribution, with the benefit of any further communication being intra-node. Option (*ii*) also requires more data to be transferred during redistribution because only one data block d_0 remains on it's original process. In case (*i*) d_0 and d_2 remain local.

The optimum choice of the redistribution data layout mainly depends on the relation between network, and shared memory performance.

Chapter 6

Numerical Experiments

Numerical experiments were carried out to demonstrate the applicability and the performance benefits of the newly developed parallelization and down-scaling methods.

Section 6.1 introduces the *Phoenix cluster*, which served as testbed for the benchmarks presented in this chapter. Section 6.2 first presents a run time analysis of SPIRAL/DMP's plain non-rescaled code, and then illustrates the effect of the down-scaling methods.

6.1 Benchmarking Environment

Experiments were carried out on the *Phoenix* Cluster[1], located at the Vienna University of Technology. It consists of 65 Sun V20z compute nodes. Each node is equipped with two AMD Opteron 250 processors running at 2.4 GHz and 4 GByte memory. The high speed cluster interconnect is a 10 Gb/s InfiniBand.

The nodes are interconnected with nine switches. Each of these switches has 24 ports. There are three *master-switches* and six *frame-switches*. 12 nodes form a frame and are connected to one frame-switch with one line each. The remaining 12 lines of one frame-switch are used as uplink to the master-switches. One frame-switch is connected to each of the three master-switches with four lines. Thus, the interconnect topology is a *fat tree*, as introduced in Section 1.2.1. Unlike a *simple tree*, every frame-switch has an equal number of uplink and downlink ports. As the bandwidth's limiting factor is the line's bandwidth and not the switches' throughput, congestion is theoretically not possible in this network topology.

The compilation and parallel runtime environment on this computer are the GNU C compiler 3.4.4, the mvapich 0.9.5 MPI library and the Sun N1 Grid Engine (S1GE) 6.0u4. In general all codes have been compiled with the `-O3` option.

Performance data are given in pseudo-Mflop/s, i. e., $5N \log N/T$, or pseudo-Gflop/s, for the investigated complex-to-complex FFTs. This unit of measurement is a scaled inverse of the run time T (in μs) and thus preserves run time relations and gives a realistic indication of the absolute floating-point performance [28].

[1] `http://www.zserv.tuwien.ac.at/phoenix/`

6.2 Experimental Results of SPIRAL/DMP

This section shows the results of run time experiments of one-dimensional double precision FFT codes generated by SPIRAL/DMP and FFTW-2.1.5. In general, three communication steps are necessary to compute a parallel one-dimensional FFT, while multi-dimensional ones only require two. Therefore, multi-dimensional FFTs are no suitable target for the rescaling optimizations introduced in this work, because there is no embedded communication step which would benefit from the optimization.

The results presented in this chapter show runtimes of the given FFT programs including code parts needed to use the FFT subroutines in a specific application. As FFTW supplies a routine that tells the user at runtime, which data distribution it expects, additional code is required. This additional routine is to provide to each process the size of the local data vector and the specific part of the global data vector it represents. Data has to be copied from the application's workspace to FFTW's one. If FFTW decides to calculate on fewer processors than the user provided, the necessary data redistribution is performed during this step. After the FFT calculation, the data is restored into the application's data array.

This is the approach suggested in the FFTW 2.1.5 manual [27]. Accordingly the necessary redistributions are included in FFTW's runtime diagrams.

SPIRAL/DMP's Performance without Rescaling. Fig. 6.2 shows the speed-up of SPIRAL/DMP, without any rescaling optimizations, compared to FFTW. Plots showing the performance behavior in Gflop/s are enclosed in Fig. 6.3.

In general, it is not efficient to calculate smaller problemsizes in parallel because most of the time is lost in communication latencies. Thus, both program's FFTs require a reasonable problemsize to become efficient (compare Fig. 6.3).

SPIRAL/DMP generates one specific routine for every vector length. In contrast, FFTW's executor causes a significant overhead especially for small problem sizes, i. e., short runtimes. This is the reason why SPIRAL/DMP is significantly faster than FFTW for smaller problem sizes. As soon as the execution times are larger and thus, FFTW's overhead becomes less significant, but SPIRAL/DMP still performs bettern than FFTW in every test case.

Down-Scaling Effect. Fig. 6.1 compares the floating point performance of SPIRAL/DMP and FFTW for one-dimensional FFTs of three specific test-cases. The diagrams (*i*) to (*iii*), each for one single FFT size, are comparing the FFT's performance of SPIRAL/DMP's calculation on 16 processors with all different scaling possibilities down to 1. FFTW is using 16 processors in any case.

Diagrams (*i*) and (*ii*) show that SPIRAL/DMP's internal down-scaling from 16 to 8 CPUs increases the overall performance of the FFT. Both plots show a

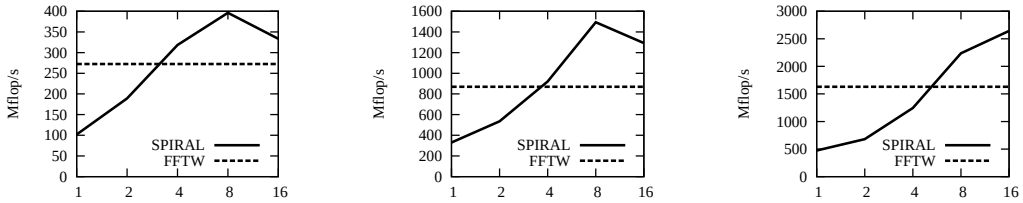


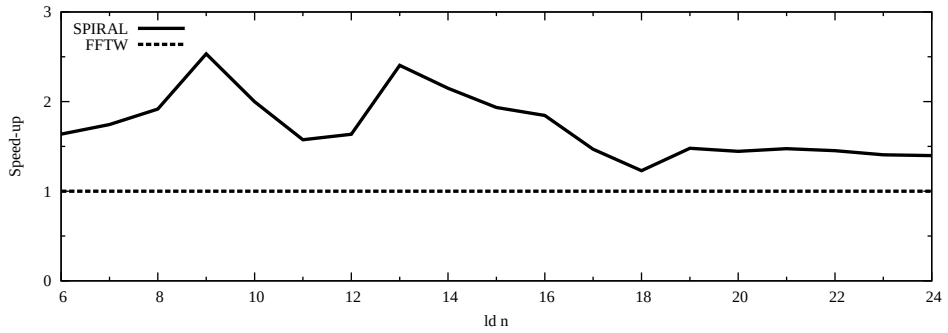
Figure 6.1: Speed-up of Rescaling. Floating-point performance (Gflop/s) of SPIRAL/DMP compared to FFTW 2.1.5 carrying out one-dimensional FFTs. The three diagrams, each for one particular FFT size, illustrate the down-scaling effect starting with 16 processors down to 1 processor.

significant speed-up when down-scaling to 8 processors. In contrast, plot (*iii*) illustrates that down-scaling is not advantageous for larger problem sizes as the performance maximum is achieved using 16 CPUs and decreases for less.

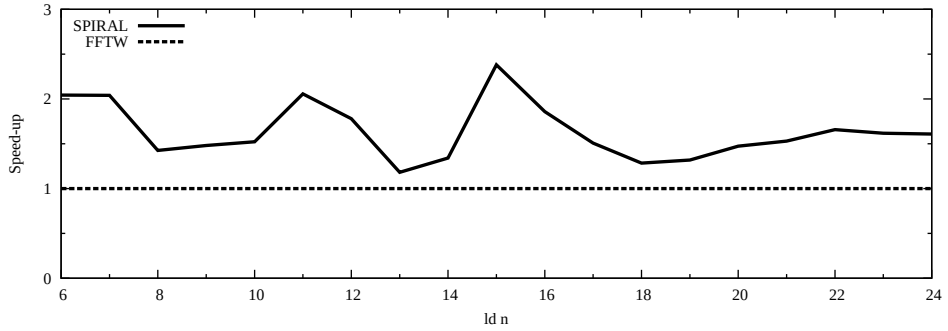
Fig. 6.4 shows the performance gain of SPIRAL/DMP's down-scaling FFT routines compared to normal FFTs without any rescaling, as shown in Fig. 6.2. Again, results are shown relative to FFTW's performance. Diagrams showing the floating point performance of these test-cases are enclosed in Fig. 6.5.

As already seen in Fig. 6.1, especially smaller problem sizes profit from SPIRAL/DMP's down-scaling. Using a slower interconnection network, e. g., Gigabit Ethernet, than the state of the art network used in these experiments would bring the down-scaling benefits to larger problem sizes. Composite signal transforms like the FFT based convolution, introduced in Section 4.9.2, with a more sophisticated data flow and a larger number of communication steps allows SPIRAL/DMP to exploit more tuning potential than naive implementations.

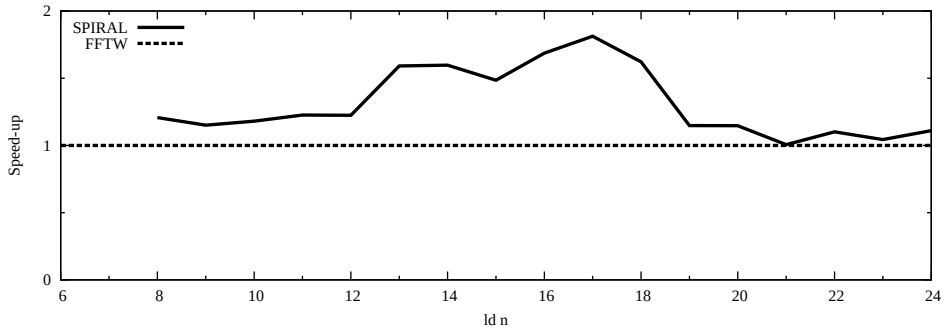
The performance improvement due to down-scaling for certain problem sizes depends on the relation of network and calculation performance and the communication volume necessary for the computation. Note that there are also other benefits of down-scaling such as reduced energy consumption and economic efficiency. Besides performance gain there might also be economic reasons to use down-scaling.



(*i*) 4 processors

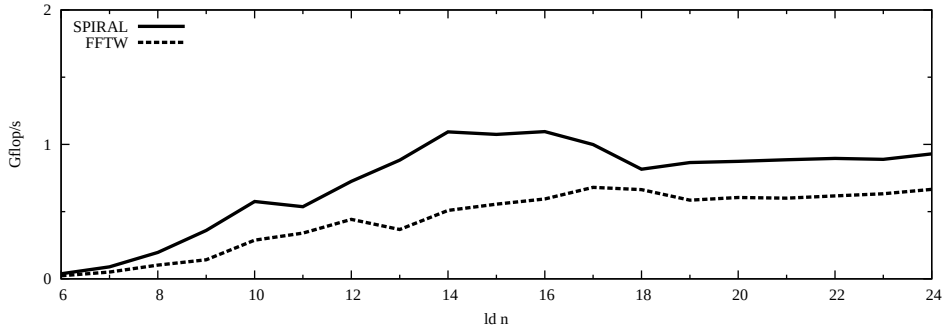


(*ii*) 8 processors

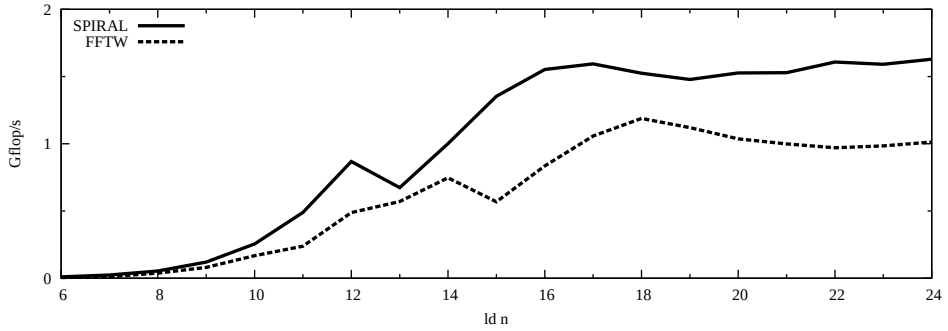


(*iii*) 16 processors

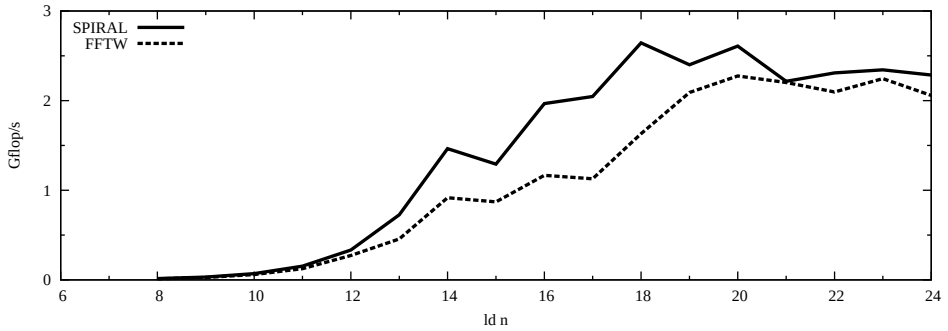
Figure 6.2: Speed-up of SPIRAL/DMP. Performance of SPIRAL/DMP computing 1D FFTs, with downscaling disabled, compared to FFTW on 4, 8, and 16 processors.



(*i*) 4 processors

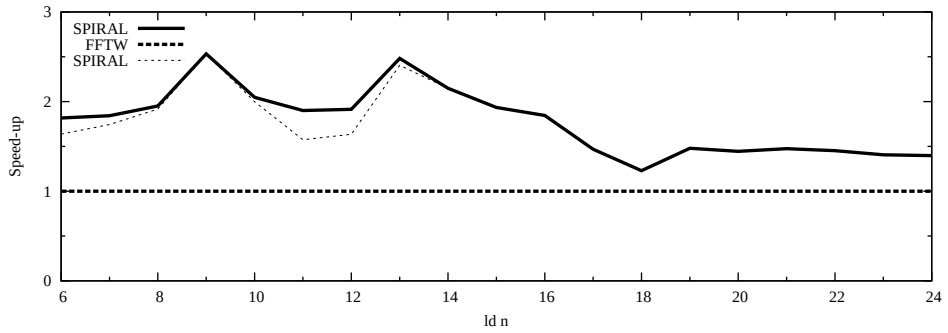


(*ii*) 8 processors

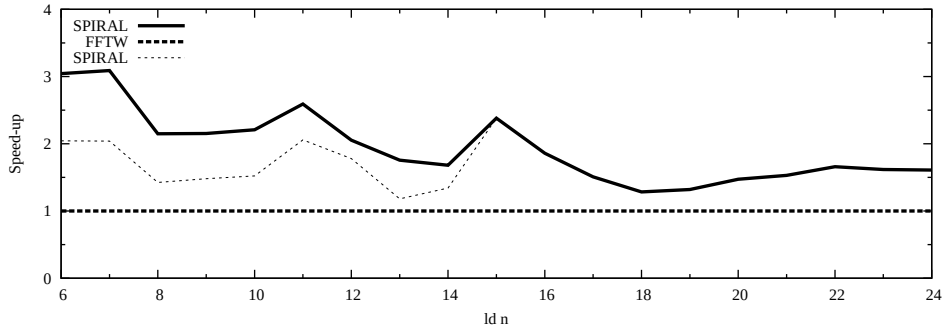


(*iii*) 16 processors

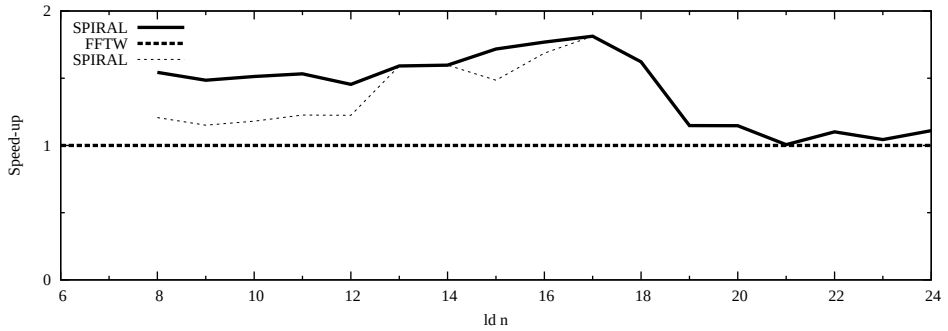
Figure 6.3: Performance of Non-rescaled SPIRAL/DMP. Floating-point performance (Gflop/s) of SPIRAL/DMP computing 1D FFTs, with downscaling disabled, compared to FFTW on 4, 8, and 16 processors.



(*i*) 4 processors

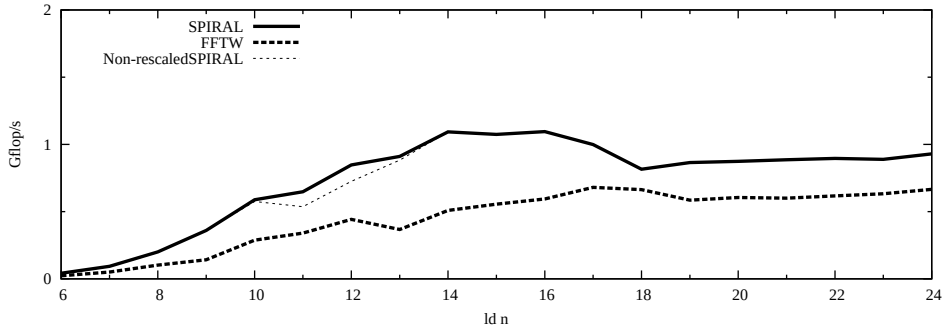


(*ii*) 8 processors

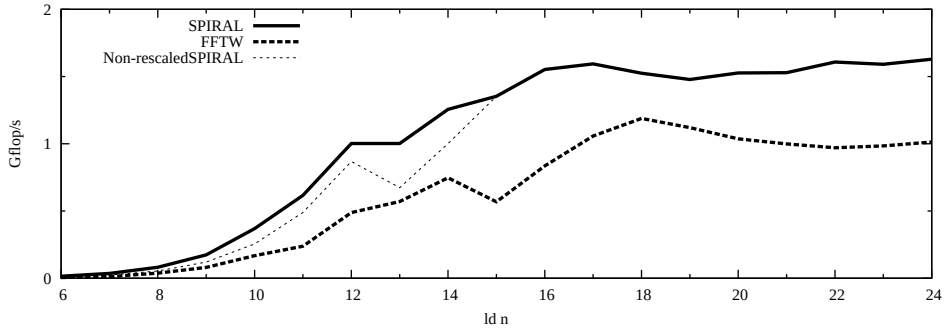


(*iii*) 16 processors

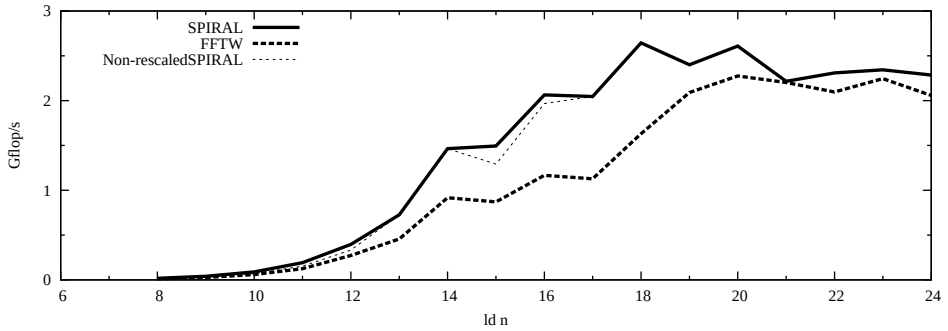
Figure 6.4: Speed-up of Down-scaling. (*i*) SPIRAL/DMP calculating on 4, 8, and 16 CPUs without down-scaling and (*ii*) rescaled SPIRAL/DMP calculating on the half number of CPUs (for certain FFT sizes) compared to FFTW.



(*i*) 4 processors



(*ii*) 8 processors



(*iii*) 16 processors

Figure 6.5: Performance of Down-scaling. Floating-point performance (Gflop/s) of (*i*) SPIRAL/DMP calculating on 4, 8, and 16 CPUs without down-scaling and (*ii*) rescaled SPIRAL/DMP calculating on the half number of CPUs (for certain FFT sizes) compared to FFTW.

Chapter 7

Outlook

This section outlines ongoing work and potential for future improvements as well as enhancements of Spiral/dmp.

As described in Chapter 1, optimizing a scalar program's performance on a certain platform is a very complex task. Distributed memory parallelism adds one more level of complexity. So, even though signal transforms in general, and the FFT in particular, are well researched topics in modern science, there is still a lot of optimization potential.

Spiral provides a toolbox which allows a faster, more structured, implementation of certain optimizations, and offers the possibility to combine them with already existing ones. Especially the optimization of parallel code benefits a lot of Spiral's approach, because there is a well defined interface between the parallel communication parts and scalar code. This allows the developer to focus on communication issues while using the existing scalar code optimization features nearly without any additional effort.

In addition to its core functionality, i. e., generating basic distributed memory parallel code, Spiral/dmp constitutes an expandable framework and provides a base where further optimizations can be built upon.

Future enhancements to Spiral/dmp could be added with respect to (*i*) communication implementation, (*ii*) communication structure, and (*iii*) usability.

7.1 Communication Implementation Progress

The communication performance of a given program does not only depend on the scheduling of communication steps but also on the communication library it is using. Distributed memory parallel programs mostly use Mpi, the current de-facto standard, for implementing communication.

Testing Different MPI Communication Routines. The Mpi standard defines multiple interfaces to implement both point-to-point and collective communication. It is common knowledge that collective communication should be superior to analogous point-to-point implementations where applicable. However, experiments [1, 6] have shown that not even this standard paradigm can be assumed to be valid without verification. Without the evaluation of different im-

plementations' performance the validity of this assumption is not easy to predict on different parallel computers.

In a real application's complex environment the MPI standard's various communication subroutines may perform totally different than in synthetic benchmarks. For instance, some implementation's `MPI_Alltoall` may be severely blocked if one of the processes lags behind, while non-blocking functions like the `MPI_Isend` `MPI_Irecv` pair automatically lead to a better balanced communication. Due to this observation it is not clear a priori which function should be used for implementing the communication of a certain program on a given platform.

Currently the communication parts in SPIRAL/DMP use the non-blocking point-to-point communication pair

`MPI_Isend MPI_Irecv`

and the blocking point-to-point exchange

`MPI_Sendrecv_replace`.

It would be possible not only to choose between those two different types of communication, but to test a larger number of different MPI routines and to choose among them. For instance, using `MPI_Alltoall` might increase the communication performance on some platforms.

Vendor Libraries. Distributed memory parallel programs do not necessarily have to use MPI. Every vendor of network interconnection hardware provides its own proprietary interface to control communication operations. As those libraries are not standardized at all, and often not even well documented, they are not easy to use.

However, if the MPI installation on a given computer system does not match the expected performance it is feasible to skip the MPI layer and to use the vendor's communication library instead. This may yield a substantial communication speedup, but also requires a substantial amount of work to be done according to the reasons mentioned above.

One-Sided Communication. Common MPI implementations are normally not capable of maintaining a reasonable scalar performance if data is communicated over the network concurrently. The reason for this phenomenon is that the network communication program has to determine every received data packet's target address. This usually requires supplementary computations on the CPU and interrupts the current calculation. Accordingly, parallel algorithms are usually separated into dedicated computation and communication phases.

Thus, even if peak network performance is reached during the communication phase, the network is left idle during the computation phase. One way to maintain scalar computations during communication is to use one-sided communication.

The difference between the usual point-to-point communication and one-sided communication is, that a one-sided communication library adds information about the target memory address to the package data. Thus, the receiving node's network controller is able to store the packet's data directly into the memory without requiring the CPU, and thus without interrupting the current computation.

One-sided communication is implemented in GASNet [5], a language-independent, low-level networking layer that provides network-independent, high-performance communication primitives. Currently GASNet is used by the Partitioned Global Address Space (PGAS) languages Unified Parallel C (UPC) [10], Titanium [36], and Co-Array Fortran [46].

Utilizing one-sided communication in SPIRAL/DMP would require a new rewriting ruleset. Currently SPIRAL/DMP is separating the transform into communication and computation steps. The order in which each computation step's results are calculated doesn't matter, as they will not be touched until the next phase. One-sided communication removes the necessity of explicit communication steps, but now the order of calculation does matter. It is critical to finish consecutive blocks of output data as soon as possible to be able to send them over the network and make best use of overlapping computation and communication [3].

7.2 Communication Structure Advancements

Formulas containing complex permutations are an obstacle for SPIRAL/DMP at the moment. The rules introduced in Section 3.1 and Section 3.2 are appropriate devices for breaking down permutations to one communication step embedded between two local permutations, provided an FFT is to be calculated on a data array that is distributed in slabs or rods.

However, these rules may not yield satisfactory results (*i*) for permutations arising in other signal transforms than FFTs, (*ii*) if the data array's initial or final distribution is more complex (e. g., volumetric), or (*iii*) other data distributions than the trivial ones mentioned before are better suited for the embedded computation parts. The introduction of extended stride permutations (Section 3.3), and especially the utilization of digit permutations (Section 3.4) are first steps towards a more comprehensive permutation treatment.

7.3 Usability Improvements

Spiral/dmp currently measures run times of code-parts to create a foundation for deciding which algorithm and which of its implementations leads to the best performance. This is the reason for the complex and uncomfortable task to set up Spiral/dmp on a given parallel computer. There are three possibilities to remove this difficulty and thus to increase the benefit of using Spiral/dmp.

Remove Runtime Measurements for Communication Parts. The code generation engine itself does not require any adaptation to the parallel computer, the code should be optimized for, when generating Mpi code. Such an adaptation is only needed to actually run parallel code during the optimization process. If the runtime measurements of parallel parts would be skipped and replaced by an *educated guess*, each scalar Spiral installation which includes the Spiral/dmp package would be able to generate parallel code without any additional setup. This could, at least, be a fallback alternative whenever Spiral/dmp finds out that it is not properly set up for parallel runtime measurement.

Model-Based Optimization. This alternative improves the first one by an estimation which algorithm and implementation will perform best on the designated target system. Therefore the user would have to provide empirically obtained data about the target machine's communication performance (e. g., latency, bandwith, and communication-computation performance ratio).

The best way to obtain these data would be to provide a small Mpi test program, which the user has to compile and run on the target machine. This program could do all necessary measurements and store the gathered data in a file, which has to be provided to Spiral/dmp in some way.

Web-Interface. The Spiral team currently aims to provide Spiral's functionality to the user community via web interfaces (e. g., the Spiral DFT IP generator[1]). Combined with a data gathering tool as explained in the last paragraph, such a web interface would provide a perfect environment to provide users with a very comfortable way to access the services of Spiral/dmp.

Of course the last alternative would be the most professional and effective one. Usually parallel libraries are not trivial to set up and are quite error-prone

[1] `http://spiral.net/hardware/dftgen.html`

when it comes to non-standard distributed memory environments. Providing SPIRAL/DMP's services through a web interface would provide high-performance C-MPI-code, which can be easily applied in parallel applications, requiring only an insignificant manual effort.

Appendix A

SPIRAL/DMP Source Codes

SPL Terminals/Σ-SPL Objects

SPL Terminal to Σ-SPL Transformation Rules

Σ-SPL Optimization Rules

Σ-SPL Complex to Real Transformation Rules

iCode Objects and C Unparser

A.1 SPL Tags

Code A.1 (APar) *Corresponding to Definition 5.1.*

```
Class(ABase, rec(
    isPV:=false,
    isVec := false,
    isPar := false,
    isReg := false,
    isMem := false,
    isMemL := false,
    isMemR := false,
    isSMP := false,
    isDMP := false,
    isFreq := false,
    isStream := false,
    isDMPRescale := false,
    DMPAllowRescale := false,
    p_all := 0,
    p:=0,
    v:=0,
    bs:=0
));
Class(A_PV, ABase, rec(isPV:=true));
Class(APar, A_PV, rec(isPar:=true));
```

Code A.2 (AParDistr) *Corresponding to Definition 5.1.*

```
Class(AParDistr, APar, rec(
    __call__ := (self, p) >> WithBases(self, rec(p:=p)),
    print := (self) >> Print(self.name, "(", self.p, ")"),
    isDMP := true,
    DMPAllowRescale := true,
    operations := Inherit(PrintOps, rec(\= := (self, other) >>
        ObjId(other) = ObjId(self) and
        self.p = other.p))
));
```

Code A.3 (`AParDMPRescale`)

```
Class(AParDMPRescale, APar, rec(
    __call__ := (self, p, q, method) >> WithBases(self,
        rec(p:=p, p_high:=p, p_low:=q, method:=method)),
    print := (self) >> Cond(self.isDMPRescaleUp or self.isDMPRescaleDown,
        Print(self.name, "(", self.p_low,
            Cond(self.isDMPRescaleDown,"<",""),"-(", self.method,")-",
            Cond(self.isDMPRescaleUp,">",""),self.p_high,")"),
        Print(self.name, "(", self.p, " (", self.method,"))")),
    procs_in  := (self) >> Cond(self.isDMPRescaleUp,   self.p_low, self.p),
    procs_out := (self) >> Cond(self.isDMPRescaleDown, self.p_low, self.p),
    implements_rescale := (self) >> self.isDMPRescaleUp or self.isDMPRescaleDown,
    method:=0,
    p_high:=0,
    p_low:=0,
    isDMPRescale := true,
    isDMP := true,
    isDMPRescaleUp := true,
    isDMPRescaleDown := true,
    operations := Inherit(PrintOps,
        rec(\= := (self, other) >>
            ObjId(other) = ObjId(self) and
            self.p = other.p and
            self.p_high = other.p_high and
            self.p_low = other.p_low and
            self.method = other.method))
))
```

A.2 SPL Non-Terminals

Code A.4 (`TPar`) *Corresponding to Definition 5.2.*

```
Class(TPar, NonTerminal, rec(
    abbrevs :=  [ (nt,pv) -> Checked(IsNonTerminal(nt) and pv.isPV, [nt, pv]) ],
    dims := self >> self.params[1].dims(),
    terminate := self >> self.params[1].terminate(),
    transpose := self >> TPar(self.params[1].transpose(), self.params[2]),
    isReal := self >> self.params[1].isReal(),
    isDMP := self >> Length(self.params[2]) > 0 and
        IsRec(self.params[2]) and
        IsBound(self.params[2].isDMP) and
        self.params[2].isDMP,
    doNotMeasure := true
))
```

Code A.5 (`MDDFT`)

```
Class(MDDFT, NonTerminal, rec(
    abbrevs := [
    P     -> Checked(IsList(P), ForAll(P,IsPosInt), Product(P) > 1,
             [ RemoveOnes(P), 1, [], false ]),
    (P,k) -> Checked(IsList(P), ForAll(P,IsPosInt), IsInt(k), Product(P) > 1,
                     Gcd(Product(P), k)=1,
             [ RemoveOnes(P), k mod Product(P), [], false ]),
    (P,k,pv) -> Checked(IsList(P), ForAll(P,IsPosInt), IsInt(k), Product(P) > 1,
                     Gcd(Product(P), k)=1, IsList(pv),
             [ RemoveOnes(P), k mod Product(P), pv, false ]),
    (P,k,pv,rc) -> Checked(IsList(P), ForAll(P,IsPosInt), IsInt(k), Product(P) > 1,
                     Gcd(Product(P), k)=1, IsList(pv),
             [ RemoveOnes(P), k mod Product(P), pv, rc ])
    ],
    dims := self >> let(n := Product(self.params[1]), When(self.isReal(), 2*[n,n], [n, n])),

    terminate := self >> let(t:=Tensor(List(self.params[1],
                                i -> DFT(i, self.params[2]).terminate())),
                            When(self.isReal(), MatAMat(RC(t).toAMat()), t)
                        ),

    transpose := self >> Copy(self),

    isReal := self >> self.params[4],

    setAB := meth(self, ab)
       self.a := ab[1];
       self.b := ab[2];
       return self;
    end,

    setpv := meth(self, pv)
        local s;
        s:= Copy(self);
        s.params[3] := pv;
        return s;
    end
));
```

Code A.6 (DFT)

```
Class(DFT, NonTerminal, rec(
    abbrevs := [ (n)   -> Checked(IsInt(n), n > 0,
                          [n, 1, [], false]),
             (n,k) -> Checked(IsInt(n), n > 0, IsInt(k), Gcd(n,k) = 1,
                          [n, k mod n, [], false]),
             (n,k,pv) -> Checked(IsInt(n), n > 0, IsInt(k), Gcd(n,k) = 1, IsList(pv),
                          [n, k mod n, pv, false]),
             (n,k,pv, rc) -> Checked(IsInt(n), n > 0, IsInt(k), Gcd(n,k) = 1, IsList(pv),
                          [n, k mod n, pv, rc])
                           ],
    dims := self >> When(self.isReal(),
        2* [ self.params[1], self.params[1] ],
        [ self.params[1], self.params[1] ]),
    terminate := self >> let(N := self.params[1], K := self.params[2],
        t := List([0..N-1], r -> List([0..N-1], c -> E(4*N)^(K*self.omega4pow(r,c)))),
            When(self.isReal(), MatAMat(RC(Mat(t)).toAMat()), Mat(t))),
    isReal := self >> self.params[4],
    SmallRandom := () -> Random([2..16]),
    LargeRandom := () -> 2 ^ Random([6..15]),
    setpv := meth(self, pv)
        local s;
        s:= Copy(self);
        s.params[3] := pv;
```

```
        return s;
    end,
    print := meth(self, indent, indentStep)
        local lparams, mparams;
        if not IsBound(self.params) then Print(self.name); return; fi;
        Print(self.name, "(");
        if IsList(self.params) then
            lparams := Filtered(self.params, i->not (IsList(i) and i=[]));
            mparams := Filtered(lparams, i->not (IsBool(i) and not i));
            DoForAllButLast(mparams, x -> Print(x, ", "));
            Print(Last(mparams));
        else
            Print(self.params);
        fi;
    Print(")", When(self.transposed, ".transpose()", ""));
    end,
))
```

Code A.7 (TTensor) *Corresponding to Definition 5.3.*

```
Class(TTensor, NonTerminal, rec(
    abbrevs :=  [ (A, B) -> Checked(IsNonTerminal(A) and IsNonTerminal(B), [A,B,[]]),
                  (A,B,pv) -> Checked(IsNonTerminal(A) and IsNonTerminal(B), [A,B,pv]) ],
    dims := self >> let(a:=self.params[1].dims(), b:=self.params[2].dims(),[a[1]*b[1],a[2]*b[2]]),
    terminate := self >> Tensor(self.params[1], self.params[2]),
    transpose := self >>
        TTensor(self.params[1].transpose(), self.params[2].transpose(), self.params[3]),
    isReal := self >> self.params[1].isReal() and self.params[2].isReal(),
    setpv := (self, pv) >> TTensor(self.params[1], self.params[2], pv)
))
```

Code A.8 (TTensorI) *Corresponding to Definition 5.4.*

```
Class(TTensorI, NonTerminal, rec(
    abbrevs :=  [ (nt, s, l, r) -> Checked(
        IsNonTerminal(nt)and IsPosInt(s) and l.isPV and r.isPV, [nt, s, l, r, []]),
                  (nt, s, l, r, pv) -> Checked(
        IsNonTerminal(nt)and IsPosInt(s) and l.isPV and r.isPV, [nt, s, l, r, pv])],
    dims := self >> self.params[1].dims()*self.params[2],
    terminate := self >>
        let(A:= self.params[1], n:= self.params[2], l:=self.params[3],  r:=self.params[4],
            Cond(l.isPar and r.isPar, Tensor(I(n), A.terminate()),
                 l.isVec and r.isVec, Tensor(A.terminate(), I(n)),
                 l.isPar and r.isVec, Tensor(I(n), A.terminate()) * L(A.dims()[2]*n, n),
                 l.isVec and r.isPar, let(m:=A.dims()[2], Tensor(A.terminate(), I(n)) * L(m*n, m))
        )),
    transpose := self >>
        TTensorI(self.params[1].transpose(), self.params[2],
            self.params[4], self.params[3], self.params[5]),
    isReal := self >> self.params[1].isReal(),
    setpv := (self, pv) >>
        TTensorI(self.params[1], self.params[2], self.params[3],
            self.params[4], Flat(pv)),
    doNotMeasure := true
))
```

Code A.9 (TL)

```
Class(TL, NonTerminal, rec(
    abbrevs :=  [ (size, stride) -> Checked(ForAll([size, stride], IsPosInt),
                      [size, stride, 1, 1, [] ]),
                  (size, stride, left, right) -> Checked(ForAll([size, stride, left, right], IsPosInt),
                      [size, stride, left, right, [] ]),
                  (size, stride, left, right, pv) -> Checked(ForAll([size, stride, left, right], IsPosInt),
                      [size, stride, left, right, pv ]) ],
    dims := self >> Replicate(2, self.params[1]*self.params[3]*self.params[4]),
    terminate := self >> Tensor(I(self.params[3]), L(self.params[1], self.params[2]), I(self.params[4])),
    transpose := self >> TL(self.params[1], self.params[1]/self.params[2],
        self.params[3], self.params[4], self.params[5]),
    isReal := self >> true,
    setpv := (self, pv) >> TL(self.params[1], self.params[2], self.params[3], self.params[4], Flat(pv))
));
```

Code A.10 (`TComm`) *Corresponding to Definition 5.10.*

```
Class(TComm, NonTerminal, rec(
    abbrevs := [
        (p, n, w_jk, pi, piinv, r_jk, j, k, pv) -> CheckedD(IsInt(p), IsInt(n), p > 1, n > 0,
            [p, n, w_jk, pi, piinv, r_jk, j, k, pv])
        ],
    dims := self >> [
        self.params[3].N * self.params[1],
        self.params[6].N * self.params[1] ],
    terminate := self >> Comm(self.params[1],self.params[2],self.params[3],
        self.params[4],self.params[5],self.params[6], self.params[7],self.params[8],self.params[9]).toAMat(),
    isReal := False,
    SmallRandom := () -> Let(m=>Random([2..8]), n=>Random([2..8]), [m*n,m]),
    LargeRandom := () -> Let(m=>2^Random([2..8]), n=>2^Random([2..7]), [m*n,m]),
    setpv := (self, pv) >> TComm(
        self.params[1],
        self.params[2],
        self.params[3],
        self.params[4],
        self.params[5],
        self.params[6],
        self.params[7],
        self.params[8],
        pv)
))
```

Code A.11 (`TDMPGlobalTranspose`) *Corresponding to Definition 5.9.*

```
Class(TDMPGlobalTranspose, NonTerminal, rec(
    abbrevs := [
        ( n , pv ) -> CheckedD(IsInt(n) , pv[1].isDMP, [ n , pv[1] ])],
    dims := self >> [ self.params[2].p^2 * self.params[1] ,
                      self.params[2].p^2 * self.params[1] ],
    terminate := self >> Tensor( L(self.params[2].p^2,self.params[2].p) , I(self.params[1]) ),
    transpose := self >> Copy(self),
    isReal := False,
    SmallRandom := () -> Let(m=>Random([2..8]), n=>Random([2..8]), [m*n,m]),
    LargeRandom := () -> Let(m=>2^Random([2..8]), n=>2^Random([2..7]), [m*n,m]),
    doNotMeasure := true
))
```

Code A.12 (`TDiag`) *Corresponding to Definition 5.6.*

```
Class(TDiag, NonTerminal, rec(
    abbrevs :=  [ (D) -> [D,[]],
                  (D,pv) -> [D,pv] ],
    dims := self >> self.params[1].dims(),
    terminate := self >> self.params[1].terminate(),
    transpose := self >> TDiag(self.params[1].transpose(), self.params[2]),
    isReal := self >> self.params[1].isReal(),
    setpv := (self, pv) >> TDiag(self.params[1], pv),
    doNotMeasure := true
));
```

A.3 SPL Rewrite Rules

A.3.1 SPL Rewrite Rules for TPar

Code A.13 (TPar_setpv)

```
TPar_setpv := rec(
    switch := true,
    info := "TPar(nt, pv) -> nt.setpv(pv)",
    forTransposition := false,
    isApplicable := P -> Length(P[2]) > 0 and P[2].isDMP,
    allChildren := P -> [[P[1].setpv([P[2]])]],
    rule := (P, C) -> C[1]
)
```

Code A.14 (TPar_genRescale)

```
TPar_genRescale := rec(
    switch := false,
    info := "TPar(nt, pv) -> nt.setpv(pv)",
    forTransposition := false,
    isApplicable := P -> Length(P[2]) > 0 and P[2].isDMP and not P[2].isDMPRescale,
    allChildren := P ->
        List(Flat(List([1],method->List(DropLast(DivisorsInt(P[2].p),1),p_low->
            [P[1].setpv([APar DMPRescale(P[2].p,p_low,method)])]))),ch->[ch]),
    rule := (P, C) -> C[1]
)
```

A.3.2 SPL Rewrite Rules for MDDFT

Code A.15 (MDDFT_tSPL_RowCol) *Corresponding to Rule 5.1.*

```
MDDFT_tSPL_RowCol := rec(
    info := "tSPL MDDFT_n -> MDDFT_n/d, MDDFT_d",
    isApplicable := P -> Length(P[1]) > 1,
    allChildren := P -> let(
        dims := P[1],
        len := Length(dims),
        List([1..len-1],
            i -> [ TTensor(MDDFT(dims{[1..i]}, P[2]), MDDFT(dims{[i+1..len]}, P[2]), P[3]) ])),
    rule := (P,C) -> C[1],
    switch := false
)
```

A.3.3 SPL Rewrite Rules for DFT

Code A.16 (DFT_tSPL_CT) *Corresponding to Rule 5.2.*

```
DFT_tSPL_CT := rec(
    info          := "tSPL DFT(mn,k) -> DFT(m, k%m), DFT(n, k%n)",
    maxSize       := false,
    isApplicable  := (self,P) >> P[1] > 2 and
        (self.maxSize=false or P[1] <= self.maxSize) and not IsPrime(P[1]) and
        not (
            Length(P[3])>0 and P[3][1].isDMP and P[3][1].isDMPRescale and
            (P[3][1].isDMPRescaleDown or P[3][1].isDMPRescaleUp))),
    allChildren   := P -> Map2(DivisorPairs(P[1]),
        (m,n) -> [ TCompose([TTensorI(DFT(m, P[2] mod m), n, AVec, AVec),
                    TDiag(T(m*n, n, P[2])),
                    TTensorI(DFT(n, P[2] mod n), m, APar, AVec)], P[3]) ]),
    rule := (P,C,nt) -> C[1],
    switch := false
)
```

Code A.17 (DFT_tSPL_CT_DMPRescale)

```
DFT_tSPL_CT_DMPRescale := rec(
    info          := "tSPL DFT(mn,k) -> DFT(m, k%m), DFT(n, k%n)",
    maxSize       := false,
    forTransposition := false,
    isApplicable := (self,P) >> (
        P[1] > 2 and
        (self.maxSize=false or P[1] <= self.maxSize) and
        not IsPrime(P[1]) and
        Length(P[3])>0 and
        P[3][1].isDMP and
        P[3][1].isDMPRescale and
        (P[3][1].isDMPRescaleUp or P[3][1].isDMPRescaleDown)) and
        P[3][1].p_low >1,
    allChildren  := P -> Map2(DivisorPairs(P[1]),
        (m,n) -> [
            TTensorI(DFT(m, P[2] mod m), n, AVec, AVec).setpv(
                [WithBases(P[3][1],rec(isDMPRescaleDown := false))]),
            TDiag(T(m*n, n, P[2])).setpv(
                [WithBases(P[3][1],rec(p:=P[3][1].p_low, isDMPRescaleDown:=false, isDMPRescaleUp:=false))]),
            TTensorI(DFT(n, P[2] mod n), m, APar, AVec).setpv(
                [WithBases(P[3][1],rec(isDMPRescaleUp := false))])
        ]),
    rule := (P,C,nt) -> DMPCompose(P[3][1].p,C),
    switch := true
)
```

Code A.18 (DFT_tSPL_CT_DMPRescale_One)

```
DFT_tSPL_CT_DMPRescale_One := rec(
    info          := "tSPL DFT(mn,k) -> DFT(m, k%m), DFT(n, k%n)",
    maxSize       := false,
    forTransposition := false,
    isApplicable := (self,P) >> (
        P[1] > 2 and
        (self.maxSize=false or P[1] <= self.maxSize) and
        not IsPrime(P[1]) and
        Length(P[3])>0 and
        P[3][1].isDMP and
```

```
        P[3][1].isDMPRescale and
        (P[3][1].isDMPRescaleUp or P[3][1].isDMPRescaleDown)) and
        P[3][1].p_low = 1,
    allChildren  := P -> Map2(DivisorPairs(P[1]),
        (m,n) -> [
            TTensorI(DFT(m, P[2] mod m), n, AVec, AVec).setpv(
                [WithBases(P[3][1],rec(isDMPRescaleDown := false))]),
            TDiag(T(m*n, n, P[2])).setpv(
                [WithBases(P[3][1],rec(p:=P[3][1].p_low, isDMPRescaleDown:=false, isDMPRescaleUp:=false))]),
            TTensorI(DFT(n, P[2] mod n), m, APar, AVec).setpv(
                [WithBases(P[3][1],rec(isDMPRescaleUp := false))])
        ]),
    rule := (P,C,nt) -> DMPCompose(P[3][1].p,C),
    switch := true
)
```

A.3.4 SPL Rewrite Rules for TTensor

Code A.19 (AxI_IxB) *Corresponding to Rule 5.3.*

```
AxI_IxB := rec(
    info := "(A x B) -> (A x I)(I x B)",
    forTransposition := false,
    isApplicable := P -> true,
    allChildren := P -> [[TCompose([
        TTensorI(P[1], P[2].dims()[1], AVec, AVec),
        TTensorI(P[2], P[1].dims()[2], APar, APar)], P[3])]],
    rule := (P, C) -> C[1]
)
```

Code A.20 (IxB_AxI) *Corresponding to Rule 5.3.*

```
IxB_AxI := rec(
    info := "(A x B) -> (I x B)(A x I)",
    forTransposition := false,
    isApplicable := P -> true,
    allChildren := P -> [[TCompose([
        TTensorI(P[2], P[1].dims()[1], APar, APar),
        TTensorI(P[1], P[2].dims()[2], AVec, AVec)], P[3])]],
    rule := (P, C) -> C[1]
)
```

A.3.5 SPL Rewrite Rules for TTensorI

Code A.21 (IxA_parDMP) *Corresponding to Rule 5.4.*

```
IxA_parDMP := rec(
    info := "I_n x A -> I_p x (I_n/p x A)",
    forTransposition := false,
    isApplicable := P ->
        Length(P[5]) > 0 and
        ForAll([P[3], P[4], P[5][1]], i-> i.isPar) and
        IsInt(P[2]/P[5][1].p) and P[5][1].isDMP and
        not (P[5][1].isDMPRescale and
            (P[5][1].isDMPRescaleUp or P[5][1].isDMPRescaleDown)),
```

```
    allChildren := P -> let(r:=P[2] / P[5][1].p, [[
        When(r=1,
            P[1],
            TTensorI(P[1], r, APar, APar)) ]]),
    rule := (P, C) -> DMPTensor(C[1] , P[5][1].p , P[5][1] )
)
```

Code A.22 (AxI_parDMP) *Corresponding to Rule 5.4.*

```
AxI_parDMP := rec(
    info := "A_rxs x I_n -> L^rn_rn/p (I_p x (I_n/p x (A_rxs x I_n/p))) L^sn_p",
    forTransposition := false,
    isApplicable := P ->
        P[3].isVec and P[4].isVec and Length(P[5]) > 0 and P[5][1].isPar and
        IsInt(P[2]/P[5][1].p) and P[5][1].isDMP and
        not (
            P[5][1].isDMPRescale and
            (P[5][1].isDMPRescaleUp or P[5][1].isDMPRescaleDown)),
    allChildren := P -> let(n  := P[2], pv := P[5], p := P[5][1].p,
        A  := P[1],a  := P[1].dims(),l  := P[2]/P[5][1].p, [[
        TL(n*a[1]/l,a[2],1,l).setpv(pv),
        TTensorI(TTensorI(A, l, AVec, AVec),p,APar,APar).setpv(pv),
        TL(n*a[2]/l,p,1,l).setpv(pv)]]),
    rule := (P, C) -> DMPCompose(P[5][1].p, C )
)
```

Code A.23 (AxIpv_parDMP) *Corresponding to Rule 5.4.*

```
AxIpv_parDMP := rec(
    forTransposition := false,
    isApplicable := P ->
        P[3].isPar and
        P[4].isVec and
        Length(P[5]) > 0 and
        P[5][1].isPar and
        IsInt(P[2]/P[5][1].p) and
        P[5][1].isDMP and
        not (P[5][1].isDMPRescale and (P[5][1].isDMPRescaleUp or P[5][1].isDMPRescaleDown)),
    allChildren := P -> let(
        n  := P[2],
        pv := P[5],
        p  := P[5][1].p,
        A  := P[1],
        a  := P[1].dims(),
        l  := P[2]/P[5][1].p,
        [[
            TTensorI(TTensorI(A, l, APar, AVec),p,APar,APar).setpv(pv),
            TL(p*a[2],p,1,l).setpv(pv)]]),
    rule := (P, C) -> DMPCompose( P[5][1].p, C )
)
```

Code A.24 (AxIvp_parDMP) *Corresponding to Rule 5.4.*

```
AxIvp_parDMP := rec(
    forTransposition := false,
    isApplicable := P ->
        P[3].isVec and
        P[4].isPar and
        Length(P[5]) > 0 and
```

```
        P[5][1].isPar and
        IsInt(P[2]/P[5][1].p) and
        P[5][1].isDMP and
        not (P[5][1].isDMPRescale and (P[5][1].isDMPRescaleUp or P[5][1].isDMPRescaleDown)),
    allChildren := P -> let(
        n  := P[2],
        pv := P[5],
        p  := P[5][1].p,
        A  := P[1],
        a  := P[1].dims(),
        l  := P[2]/P[5][1].p, [[
            TL(a[1]*p,a[1],1,l).setpv(pv),
            TTensorI(TTensorI(A, l, AVec, APar),p,APar,APar).setpv(pv)]]),
    rule := (P, C) -> DMPCompose( P[5][1].p, C )
)
```

Code A.25 (AxIvp_parDMP_rescale)

```
AxIvp_parDMP_rescale := rec(
    forTransposition := false,
    isApplicable := P ->
        P[3].isVec and
        P[4].isPar and
        Length(P[5]) > 0 and
        P[5][1].isPar and
        IsInt(P[2]/P[5][1].p) and
        P[5][1].isDMP and
        P[5][1].isDMPRescale and
        (P[5][1].isDMPRescaleUp or P[5][1].isDMPRescaleDown) and
        (not(P[5][1].isDMPRescaleUp and P[5][1].isDMPRescaleDown)),
    allChildren := P -> let(
        n  := P[2],
        pv := P[5][1],
        p  := P[5][1].p,
        A  := P[1],
        a  := P[1].dims(),
        l  := P[2]/P[5][1].p,
        [[
            TL(a[1]*p,a[1],1,l).setpv(                              [ Copy(pv) ]),
            TTensorI(TTensorI(A, l, AVec, APar),p,APar,APar).setpv( [
                Cond(pv.isDMPRescaleDown ,
                    WithBases(pv,rec(isDMPRescaleDown:=false)),
                    WithBases(pv,rec(isDMPRescaleUp  :=false, p:=pv.p_low)))])
        ]]),
    rule := (P, C) -> DMPCompose(P[5][1].p,C)
)
```

Code A.26 (AxIpv_parDMP_rescale)

```
AxIpv_parDMP_rescale := rec(
    forTransposition := false,
    isApplicable := P ->
        P[3].isPar and
        P[4].isVec and
        Length(P[5]) > 0 and
        P[5][1].isPar and
        IsInt(P[2]/P[5][1].p) and
        P[5][1].isDMP and
        P[5][1].isDMPRescale and
        (P[5][1].isDMPRescaleUp or P[5][1].isDMPRescaleDown) and
        (not(P[5][1].isDMPRescaleUp and P[5][1].isDMPRescaleDown)),
    allChildren := P -> let(
```

```
        n  := P[2],
        pv := P[5][1],
        p  := P[5][1].p,
        A  := P[1],
        a  := P[1].dims(),
        l  := P[2]/P[5][1].p,
            [[
                TTensorI(TTensorI(A, l, APar, AVec),p,APar,APar).setpv( [
                    Cond(pv.isDMPRescaleDown,
                        WithBases(pv, rec(isDMPRescaleDown := false, p:=pv.p_low)),
                        WithBases(pv, rec(isDMPRescaleUp   := false)))
                ] ),
                TL(p*a[2],p,1,l).setpv(                                  [ Copy(pv) ] )
            ]]),
    rule := (P, C) -> DMPCompose(P[5][1].p,C)
)
```

Code A.27 (AxI_parDMP_rescaleBoth)

```
AxI_parDMP_rescaleBoth := rec(
    info := "A_rxs x I_n -> L^rn_rn/p (I_p x (I_n/p x (A_rxs x I_n/p))) L^sn_p",
    forTransposition := false,
    isApplicable := P ->
        P[3].isVec and
        P[4].isVec and
        Length(P[5]) > 0 and
        P[5][1].isPar and
        IsInt(P[2]/P[5][1].p) and
        P[5][1].isDMP and
        P[5][1].isDMPRescale and
        (P[5][1].isDMPRescaleUp and P[5][1].isDMPRescaleDown),
    allChildren := P -> let(
        n  := P[2],
        pv := P[5][1],
        p  := P[5][1].p,
        A  := P[1],
        a  := P[1].dims(),
        l  := P[2]/P[5][1].p,
        [[
            TL(n*a[1]/l,a[2],1,l).setpv(
                [ WithBases(pv, rec(isDMPRescaleDown:=false)) ] ),
            TTensorI(TTensorI(A, l, AVec, AVec),p,APar,APar).setpv(
                [ WithBases(pv, rec(isDMPRescaleDown:=false,
                                isDMPRescaleUp:=false, p:=pv.p_low ) ) ] ),
            TL(n*a[2]/l,p,1,l).setpv(
                [ WithBases(pv, rec(isDMPRescaleUp:=false)) ])
        ]]),
    rule := (P, C) -> DMPCompose(P[5][1].p,C)
)
```

Code A.28 (AxI_parDMP_rescaleDown_unpulled)

```
AxI_parDMP_rescaleDown_unpulled := rec(
    info := "A_rxs x I_n -> L^rn_rn/p (I_p x (I_n/p x (A_rxs x I_n/p))) L^sn_p",
    forTransposition := false,
    isApplicable := P ->
        P[3].isVec and
        P[4].isVec and
        Length(P[5]) > 0 and
        P[5][1].isPar and
        IsInt(P[2]/P[5][1].p) and
        P[5][1].isDMP and
```

```
        P[5][1].isDMPRescale and
        P[5][1].isDMPRescaleUp = false and
        P[5][1].isDMPRescaleDown,
    allChildren := P -> let(
        n  := P[2],
        pv := P[5][1],
        p  := P[5][1].p,
        A  := P[1],
        a  := P[1].dims(),
        l  := P[2]/P[5][1].p,
        [[
            TL(n*a[1]/l,a[2],1,l).setpv(
                [ WithBases(pv, rec(isDMPRescaleDown:=false, p:=pv.p_low )) ] ),
            TTensorI(TTensorI(A, l, AVec, AVec),p,APar,APar).setpv(
                [ WithBases(pv, rec(isDMPRescaleDown:=false, p:=pv.p_low )) ] ),
            TL(n*a[2]/l,p,1,l).setpv(
                [ Copy(pv) ])
        ]]),
    rule := (P, C) -> DMPCompose(P[5][1].p,C)
)
```

Code A.29 (AxI_parDMP_rescaleUp)

```
AxI_parDMP_rescaleUp := rec(
    info := "A_rxs x I_n -> L^rn_rn/p (I_p x (I_n/p x (A_rxs x I_n/p))) L^sn_p",
    forTransposition := false,
    isApplicable := P ->
        P[3].isVec and
        P[4].isVec and
        Length(P[5]) > 0 and
        P[5][1].isPar and
        IsInt(P[2]/P[5][1].p) and
        P[5][1].isDMP and
        P[5][1].isDMPRescale and
        P[5][1].isDMPRescaleUp and
        P[5][1].isDMPRescaleDown = false,
    allChildren := P -> let(
        n  := P[2],
        pv := P[5][1],
        p  := P[5][1].p,
        A  := P[1],
        a  := P[1].dims(),
        l  := P[2]/P[5][1].p, [[
            TL(n*a[1]/l,a[2],1,l).setpv(
                [ Copy(pv) ] ),
            TTensorI(TTensorI(A, l, AVec, AVec),p,APar,APar).setpv(
                [ WithBases(pv, rec(isDMPRescaleUp:=false, p:=pv.p_low )) ] ),
            TL(n*a[2]/l,p,1,l).setpv(
                [ WithBases(pv, rec(isDMPRescaleUp:=false, p:=pv.p_low )) ])
        ]]),
    rule := (P, C) -> DMPCompose(P[5][1].p,C)
)
```

A.3.6 SPL Rewrite Rules for TDMPGlobalTranspose

Code A.30 (TDMPGlobalTranspose_Base)

```
TDMPGlobalTranspose_Base := rec(
    switch           := true,
    info             := "TDMPGlobalTranspose(n,pv) -> DMPGlobalTranspose(n,pv) )",
```

```
    forTransposition := false,
    isApplicable     := P -> true,
    allChildren := P -> [[ ]],
    rule := (P, C) -> DMPGlobalTranspose( P[1] , P[2] )
)
```

Code A.31 (TDMPGlobalTranspose2TComm) *Corresponding to Code A.31.*

```
TDMPGlobalTranspose2TComm := rec(
    switch           := false,
    info             := "TDMPGlobalTranspose(n,pv) -> TComm(...) )",
    forTransposition := false,
    isApplicable     := P -> Length(P[2])>0,
    allChildren := P ->
        Let(p => P[2].p, n => P[1], pv => P[2], j => Ind( P[2].p ), k => Ind( P[2].p - 1 ),
            [[TComm(p,n,fTensor(fZeta(p,j,k),fId(n)),
                fCompose(fGamma(p),L(p^2,p),fGammaPrime(p)),
                fCompose(fGamma(p),L(p^2,p),fGammaPrime(p)),
                fTensor(fZeta(p,j,k),fId(n)),j,k).setpv(pv)]]),
    rule := (P, C) -> C[1]
)
```

A.3.7 SPL Rewrite Rules for TL

Code A.32 (IxLxI_nopar) *Corresponding to Rule 5.6.*

```
IxLxI_nopar := rec (
    switch := true,
    info             := "L^{mn}_1, L^{mn}_{mn} -> I_mn",
    forTransposition := false,
    isApplicable     := P -> Length(P[5]) > 0 and P[5][1].isDMP and P[5][1].p = 1,
    allChildren := P -> [[TL(P[1],P[2],P[3],P[4])]],
    rule := (P, C) -> C[1]
)
```

Code A.33 (IxLxI_trivial) *Corresponding to Rule 5.6.*

```
IxLxI_trivial := rec (
    switch := true,
    info             := "L^{mn}_1, L^{mn}_{mn} -> I_mn",
    forTransposition := false,
    isApplicable     := P -> P[2]=1 or P[2]=P[1],
    allChildren := P -> [[]],
    rule := (P, C) -> I(P[1]*P[3]*P[4])
)
```

Code A.34 (IxLxI_DMP_LCL) *Corresponding to Rule 5.5.*

```
IxLxI_DMP_LCL := rec (
    switch := true,
    forTransposition := false,
    isApplicable     := P ->
        true
        and Length(P[5]) > 0
        and P[5][1].isDMP
```

```
        and P[3] = 1
        and (P[5][1].p > 1)
        and (not P[1]/P[2] = P[5][1].p)
        and (not P[2] = P[5][1].p)
        and IsInt(P[1]/P[2])
        and IsInt(P[2]/P[5][1].p)
        and IsInt((P[1]/P[2])/P[5][1].p)
        and not (P[5][1].isDMPRescale and (P[5][1].isDMPRescaleUp or P[5][1].isDMPRescaleDown))
    ,
    allChildren := P ->
        Let(mn => P[1], m => P[2], r => P[4],  n=> P[1]/P[2], pv => P[5], p => P[5][1].p ,
        [[
            TL(mn/p,m/p,p,r).setpv(pv),
            TDMPGlobalTranspose(r*mn/p^2, pv),
            TL(n,p,p,r*m/p).setpv(pv)
        ]]),
    rule := (P, C) -> DMPCompose( P[5][1].p, C )
)
```

Code A.35 (IxLxI_DMP_CL) *Corresponding to Rule 5.5.*

```
IxLxI_DMP_CL   := rec(
    switch           := true,
    forTransposition := false,
    isApplicable     := P ->
        true
        and Length(P[5]) > 0
        and P[5][1].isDMP
        and P[3] = 1
        and (P[5][1].p > 1)
        and IsInt(P[1]/P[2])
        and IsInt((P[1]/P[2])/P[5][1].p)
        and P[2] = P[5][1].p
        and not (P[5][1].isDMPRescale and (P[5][1].isDMPRescaleUp or P[5][1].isDMPRescaleDown))
        ,
    allChildren      := P -> Let(p => P[2], pv=>P[5], r => P[4], n => P[1]/P[2],
        [[
            TDMPGlobalTranspose( r*n/p, pv),
            TL(n,p,p,r).setpv(pv)
        ]]),
    rule := (P, C) -> DMPCompose( P[5][1].p, C )
)
```

Code A.36 (IxLxI_DMP_LC) *Corresponding to Rule 5.5.*

```
IxLxI_DMP_LC := rec(
    switch := true,
    forTransposition := false,
    isApplicable     := P ->
        true
        and Length(P[5]) > 0
        and P[5][1].isDMP
        and P[3] = 1
        and (P[5][1].p > 1)
        and P[1]/P[2] = P[5][1].p
        and IsInt(P[2]/P[5][1].p)
        and not (P[5][1].isDMPRescale and (P[5][1].isDMPRescaleUp or P[5][1].isDMPRescaleDown))
        ,
    allChildren      := P -> Let(p => P[5][1].p, r => P[4], m => P[2], pv=>P[5],
        [[
            TL(m,m/p,P[5][1].p,r).setpv(Copy(pv)),
```

```
            TDMPGlobalTranspose(r*m/p, Copy(pv))
        ]]),
    rule := (P, C) -> DMPCompose( P[5][1].p, C )
)
```

Code A.37 (IxLxI_DMP_L) *Corresponding to Rule 5.6.*

```
IxLxI_DMP_L    := rec(
    forTransposition := false,
    isApplicable     := P ->
        true
        and Length(P[5]) > 0
        and P[5][1].isDMP
        and P[5][1].p > 1
        and P[3] > 1
        and IsInt(P[3]/P[5][1].p)
        and not (P[5][1].isDMPRescale and (P[5][1].isDMPRescaleUp or P[5][1].isDMPRescaleDown))
        ,
    allChildren      := P -> Let(p => P[5][1].p,
        m => P[2], pv => P[5], j => Ind(P[5][1].p), k => Ind(P[5][1].p-1),
        [[
            TTensorI(
            TL(P[1],P[2],P[3]/P[5][1].p,P[4]), P[5][1].p,APar,APar).setpv(Copy(pv))
        ]]),
    rule := (P, C) -> C[1],
    switch := true
)
```

Code A.38 (IxLxI_DMP_LCL_rescale)

```
IxLxI_DMP_LCL_rescale := rec (
    switch := true,
    forTransposition := false,
    isApplicable     := P ->
        true
        and Length(P[5]) > 0
        and P[5][1].isDMP
        and P[3] = 1
        and (P[5][1].p > 1)
        and (not P[1]/P[2] = P[5][1].p)
        and (not P[2] = P[5][1].p)
        and IsInt(P[1]/P[2])
        and IsInt(P[2]/P[5][1].p)
        and IsInt((P[1]/P[2])/P[5][1].p)
        and (
            P[5][1].isDMPRescale
            and (P[5][1].isDMPRescaleUp or P[5][1].isDMPRescaleDown)
            and not (P[5][1].isDMPRescaleUp and P[5][1].isDMPRescaleDown)),
    allChildren := P ->
        Let(mn => P[1], m => P[2], r => P[4],  n=> P[1]/P[2], pv => P[5][1], p => P[5][1].p ,
            [[
                TL(mn/p,m/p,p,r).setpv(
                    [ Cond(pv.isDMPRescaleDown,
                        WithBases(pv, rec(isDMPRescaleDown:=false, p:=pv.p_low)),
                        WithBases(pv, rec(isDMPRescaleUp:=false)))] ),
                TDMPGlobalTranspose(r*mn/p^2, [ pv ] ),
                TL(n,p,p,r*m/p).setpv(
                    [ Cond(pv.isDMPRescaleDown,
                        WithBases(pv, rec(isDMPRescaleDown:=false)),
                        WithBases(pv, rec(isDMPRescaleUp:=false, p:=pv.p_low))) ] )
            ]]),
    rule := (P, C) -> DMPCompose( P[5][1].p, C )
)
```

Code A.39 (IxLxI_DMP_CL_rescale)

```
IxLxI_DMP_CL_rescale   := rec(
    switch           := true,
    forTransposition := false,
    isApplicable     := P ->
        true
        and Length(P[5]) > 0
        and P[5][1].isDMP
        and P[3] = 1
        and (P[5][1].p > 1)
        and IsInt(P[1]/P[2])
        and IsInt((P[1]/P[2])/P[5][1].p)
        and P[2] = P[5][1].p
        and (
            P[5][1].isDMPRescale
            and (P[5][1].isDMPRescaleUp or P[5][1].isDMPRescaleDown)
            and not (P[5][1].isDMPRescaleUp and P[5][1].isDMPRescaleDown)),
    allChildren := P -> Let(p => P[2], pv=>P[5][1], r => P[4], n => P[1]/P[2],
        [[
            TDMPGlobalTranspose( r*n/p, [ Copy(pv) ] ),
            TL(n,p,p,r).setpv                    (
                [ Cond(pv.isDMPRescaleDown,
                    WithBases(pv,rec(isDMPRescaleDown:=false)),
                    WithBases(pv,rec(isDMPRescaleUp:=false, p:=pv.p_low))) ] )
        ]]),
    rule := (P, C) -> DMPCompose( P[5][1].p , C )
)
```

Code A.40 (IxLxI_DMP_LC_rescale)

```
IxLxI_DMP_LC_rescale := rec(
    switch := true,
    forTransposition := false,
    isApplicable     := P ->
        true
        and Length(P[5]) > 0
        and P[5][1].isDMP
        and P[3] = 1
        and (P[5][1].p > 1)
        and P[1]/P[2] = P[5][1].p
        and IsInt(P[2]/P[5][1].p)
        and (P[5][1].isDMPRescale and
            (P[5][1].isDMPRescaleUp or P[5][1].isDMPRescaleDown) and
            not (P[5][1].isDMPRescaleUp and P[5][1].isDMPRescaleDown)),
    allChildren := P -> Let(p => P[5][1].p, r => P[4], m => P[2], pv=>P[5][1],
        [[
            TL(m,m/p,P[5][1].p,r).setpv(
                [ Cond(pv.isDMPRescaleDown,
                    WithBases(pv,rec(isDMPRescaleDown:=false, p:=pv.p_low)),
                    WithBases(pv,rec(isDMPRescaleUp:=false))) ]),
            TDMPGlobalTranspose( r*m/p, [ Copy(pv) ] )
        ]]),
    rule := (P, C) -> DMPCompose( P[5][1].p, C )
)
```

A.3.8 SPL Rewrite Rules for TCompose

Code A.41 (TCompose_DMP_Base)

```
TCompose_DMP_Base := rec(
    info := "TCompose DMP",
    forTransposition := false,
    isApplicable := P -> Length(P[2]) > 0 and P[2][1].isDMP and
        not (P[2][1].isDMPRescale and (P[2][1].implements_rescale())),
    allChildren := P -> [List(P[1], i->i.setpv(P[2]))],
    rule := (P, C) -> DMPCompose(P[2][1].p,C)
)
```

A.3.9 SPL Rewrite Rules for TComm

Code A.42 (TComm_Base)

```
TComm_Base     := rec(
    switch           := true,
    info             := "",
    forTransposition := false,
    isApplicable     := P -> true,
    allChildren      := P -> [[]],
    rule             := (P,C) -> DMPComm(P[1],P[2],P[3],P[4],P[5],P[6],P[7],P[8])
)
```

A.3.10 SPL Rewrite Rules for TDiag

Code A.43 (TDiag_DMP)

```
TDiag_DMP := rec(
    info := "TDiag DMP",
    forTransposition := false,
    isApplicable := P -> Length(P[2]) > 0 and P[2][1].isDMP,
    rule := (P, C) -> P[1]
)
```

A.4 SPL Terminals/Σ-SPL Objects

Code A.44 (DMPCompose)

```
Declare(DMPCompose);
Class(DMPCompose, BaseOperation, rec(
    #-------------------------------------------------------------------------
    _spl_name := "DMPCompose",
    #-------------------------------------------------------------------------
    abbrevs :=  [arg -> [arg]],
    #-------------------------------------------------------------------------
    isDMP := self >> true,
    #-------------------------------------------------------------------------
    checkDims := children -> DoForAll([1..Length(children)-1], i ->
        When(not Cols(children[i])=Rows(children[i+1]),
        Error("Dimensions of children do not match (i=",i,",",i+1,")"), 0)),
    #-------------------------------------------------------------------------
    new := meth(self, L)
        local dim,factors,procs,pars;
        Constraint(IsList(L) and L<>[]);
```

```
        pars := Flat(L);
        factors := Filtered(pars,x -> IsSPL(x));
        procs := Flat(Filtered(pars,x -> IsInt(x)))[1];
        dim := Rows(factors[1]);
        self.checkDims(factors);
        factors := Filtered(factors, x -> not IsIdentitySPL(x));
        if Length(factors) = 1 then return factors[1]; fi;
        if factors = [] then   # all factors are identities
            return formgen.I(dim);
        else
            return SPL(WithBases( self,
                rec( _children := factors,
                    p := procs,
                    dimensions := [ factors[1].dimensions[1],
                    factors[Length(factors)].dimensions[2] ] )));
        fi;
    end,
    #---------------------------------------------------------------------
    dims := self >> Let(c => self._children, [Rows(c[1]), Cols(Last(c))]),
    #---------------------------------------------------------------------
    isPermutation := self >> ForAll(self._children, IsPermutationSPL),
    #---------------------------------------------------------------------
    toAMat := meth(self)
        return Product(List(self._children, AMatSPL));
    end,
    #---------------------------------------------------------------------
    transpose := self >>  # we use inherit to copy all fields of self
        Inherit(self, rec(
            _children := Reversed(List(self._children, TransposedSPL)),
            dimensions := Reversed(self.dimensions))),
    #---------------------------------------------------------------------
    print := meth(self, indent, indentStep)
        local s, newline;
        s := self.children();
        if Length(s) = 2 and ((IsBound(s[1]._sym) and IsBound(s[2]._mat))
                          or (IsBound(s[2]._sym) and IsBound(s[1]._mat)))
           or ForAll(s, x->IsBound(x._sym))
           then newline := Ignore;
        else newline := self._newline; fi;
        DoForAllButLast(s, c->Chain(SPLOps.Print(c, indent, indentStep),
            Print(" *DMP* "), newline(indent)));
        Last(s).print(indent, indentStep);
    end,
    #---------------------------------------------------------------------
    arithmeticCost := (self, costMul, costAddMul) >>
        Sum(List(self.children(), x -> x.arithmeticCost(costMul, costAddMul)))
))
```

Code A.45 (DMPComm)

```
DMPComm := WithBases(BaseContainer, rec(
    name := "DMPComm",
    #---------------------------------------------------------------------
    isDMP := self >> true,
    #---------------------------------------------------------------------
    new := meth(self, p, n, w_jk, pi, piinv, r_jk, j, k)
    return SPL(WithBases( self,
        rec( _children := [],
            p := p,         # processors
            n := n,         # packet size
            w_jk := w_jk,   #
            pi := pi,       # communication perm
            piinv := piinv, # communication perm
            r_jk := r_jk,   #
```

```
            j := j,
            k := k,
            dimensions := [ w_jk.range() * p , r_jk.range() * p ]
        )));
    end,
    #---------------------------------------------------------------------
    dims := self >> self.dimensions,
    #---------------------------------------------------------------------
    area := self >> self.dimensions[1] * self.dimensions[2],
    #---------------------------------------------------------------------
    isPermutation := False,
    #---------------------------------------------------------------------
    toAMat := meth(self)
        local i, exp, m, p, sf, gf, s, g, perm;
        p := self.p;
        g:= IterDirectSum(self.j,self.j.range,IterVStack(self.k,self.k.range,Gath(self.r_jk)));
        s:= IterDirectSum(self.j,self.j.range,IterHStack(self.k,self.k.range,Scat(self.w_jk)));
        perm := Prm(fTensor(self.pi,fId(self.n)));
        exp := MatSPL(s*perm*g);
        for i in [1..Length(exp)] do
            if ForAll(exp[i], k->k=0) then
                exp[i][i]:=1;
            fi;
        od;
        return AMatMat(exp);
    end,
    #---------------------------------------------------------------------
    equals := meth(self, other)
        return
            IsSPL(other)
            and ObjId(other) = ObjId(self)
            and other.scalar = self.scalar
            and IsIdenticalSPL(other._children[1], self._children[1]);
    end,
    #---------------------------------------------------------------------
    print := meth(self, indent, indentStep)
        self._print(indent, indentStep, self.name,
            [ self.p,", ",self.n,", ",self.w_jk,", ",self.pi,,", ",self.r_jk ],"");
    end,
    #---------------------------------------------------------------------
    export := meth(self, indent, indentStep)
        Print("unsupported ", self._spl_name);
    end,
    #---------------------------------------------------------------------
    arithmeticCost := (a,b,c) >> 0,
    #---------------------------------------------------------------------
    isReal := meth(self) return true; end
))
```

Code A.46 (DMPGlobalTranspose)

```
DMPGlobalTranspose := WithBases(BaseContainer, rec(
    name := "DMPGlobalTranspose",
    #---------------------------------------------------------------------
    isDMP := self >> true,
    #---------------------------------------------------------------------
    new := (self, n, pv) >> SPL(WithBases(self,rec(
         _children:=[],
         pv:=pv,
         p:=pv.p,
         n:=n,
         dimensions:=[ n*(pv.p)^2 , n*(pv.p)^2 ]))),
    #---------------------------------------------------------------------
    dims := self >> self.dimensions,
```

```
    #-------------------------------------------------------------------------
    area := self >> self.dimensions[1] * self.dimensions[2],
    #-------------------------------------------------------------------------
    isPermutation := False,
    #-------------------------------------------------------------------------
    toAMat := self >> Gath(fTensor(L(self.p^2,self.p),fId(self.n))).toAMat(),
    #-------------------------------------------------------------------------
    equals := meth(self, other)
        return
            IsSPL(other)
            and ObjId(other) = ObjId(self)
            and other.scalar = self.scalar
            and IsIdenticalSPL(other._children[1], self._children[1]);
    end,
    #-------------------------------------------------------------------------
    print := meth(self, indent, indentStep)
        self._print(indent, indentStep, self.name,
            [ self.pv ,", ",self.n],"");
    end,
    #-------------------------------------------------------------------------
    export := meth(self, indent, indentStep)
        Print("unsupported ", self._spl_name);
    end,
    #-------------------------------------------------------------------------
    arithmeticCost := (a,b,c) >> 0,
    #-------------------------------------------------------------------------
    isReal := meth(self) return true; end
))
```

Code A.47 (DMPGath)

```
Declare(DMPScat);
Class(DMPGath, BaseMat, SumsBase, rec(
    isDMP := self >> true,
    #-------------------------------------------------------------------------
    rChildren := self >> [self.func],
    #-------------------------------------------------------------------------
    rSetChild := rSetChildFields("func"),
    #-------------------------------------------------------------------------
    new := meth(self, pvar,pdomain,pexpr)
        local pfunc;
        pfunc := fTensor(fBase(pdomain,pvar),pexpr);
        return SPL(WithBases(self,rec(
            dimensions := [pfunc.domain(), pfunc.range()],
            var := pvar,
            expr := pexpr,
            domain := pdomain,
            func := FF(pfunc))));
    end,
    #-------------------------------------------------------------------------
    sums := self >> self,
    #-------------------------------------------------------------------------
    area := self >> self.func.domain(),
    #-------------------------------------------------------------------------
    isReal := self >> true,
    #-------------------------------------------------------------------------
    transpose := self >> DMPScat(self.var,self.domain.self.expr),
    #-------------------------------------------------------------------------
    print := (self,i,is) >> Print(self.name, "(" , self.func, ")"),
    #-------------------------------------------------------------------------
    toAMat := meth(self)
        local n,N,lfunc;
        n := self.dimensions[1];
        N := self.dimensions[2];
```

```
        lfunc := self.func.lambda();
        return AMatMat(List([0..n-1], row -> BasisVec(N, lfunc.at(row).ev())));
        end
))
```

Code A.48 (DMPScat)

```
Class(DMPScat, BaseMat, SumsBase, rec(
    isDMP := self >> true,
    #---------------------------------------------------------------------
    rChildren := self >> [self.func],
    #---------------------------------------------------------------------
    rSetChild := rSetChildFields("func"),
    #---------------------------------------------------------------------
    new := meth(self, pvar,pdomain,pexpr)
        local pfunc;
        pfunc := fTensor(fBase(pdomain,pvar),pexpr);
        return SPL(WithBases(self,rec(
            dimensions := [pfunc.range(), pfunc.domain()],
            var := pvar,
            expr := pexpr,
            domain := pdomain,
            func := FF(pfunc))));
    end,
    #---------------------------------------------------------------------
    func := self >> FF(fTensor(fBase(self.domain,self.var),self.expr)),
    #---------------------------------------------------------------------
    sums := self >> self,
    #---------------------------------------------------------------------
    area := self >> self.func.domain(),
    #---------------------------------------------------------------------
    isReal := self >> true,
    #---------------------------------------------------------------------
    transpose := self >> DMPGath(self.var,self.domain,self.expr),
    #---------------------------------------------------------------------
    print := (self,i,is) >> Print(self.name, "(", self.func, ")"),
    #---------------------------------------------------------------------
    toAMat := meth(self)
        local n,N,lfunc;
        n := self.dimensions[1];
        N := self.dimensions[2];
        lfunc := self.func.lambda();
        return TransposedAMat(AMatMat(List([0..N-1], row -> BasisVec(n, lfunc.at(row).ev()))));
    end
))
```

Code A.49 (DMPISum) *Corresponding to Definition 5.11.*

```
Class(DMPISum, ISum, rec(
    new := meth(self, var, domain, expr, pv)
        Constraint(IsSPL(expr));
        Constraint(not IsList(domain));
        return SPL(WithBases( self,
            rec(expr := expr,
                var := var,
                domain := domain,
                p := domain,
                pv := pv,
                dimensions := Dimensions(expr) )));
    end,
    #---------------------------------------------------------------------
    isDMP := self >> true,
```

```
    #-------------------------------------------------------------------------
    setVar := meth( self, newvar )
        local oldvar, replace;
        oldvar := self.var;
        replace := function ( c )
            if IsBound( c.var ) and c.var.id = oldvar.id then
                c.var := newvar;
            fi;
            return Copy(c).mutateChildren( replace );
        end;
        return Copy(self).mutateChildren( replace );
    end,
    #-------------------------------------------------------------------------
    unroll := meth(self)
        return SUM( List([0..self.domain-1], index_value ->
            SubstBottomUp(
                Copy(self.expr),
                @(1).target(var).cond(e->e.id=self.var.id),
                e -> V(index_value))));
    end
))
```

Code A.50 (DMPIterDirectSum)

```
Class(DMPIterDirectSum, IterDirectSum, rec(
    isDMP:=self>>true
))
```

Code A.51 (DMPTensor)

```
Class(DMPTensor, Tensor, rec(
    new := (self, L) >> SPL(WithBases(self, rec(type := "tensor",
        _children := [
            I(L[2]),
            L[1]],
        dimensions := L[1].dimensions * L[2],
        p := L[2],
        ptype := L[3],
        pv:=L[3]))),
    #-------------------------------------------------------------------------
    print := (self,i,is) >> Print(self.name, "(", self.child(2), ", ", self.p, ", ", self.ptype, ")"),
    #-------------------------------------------------------------------------
    isDMP := self >> true,
    #-------------------------------------------------------------------------
    isPermutation := False
))
```

A.5 SPL Terminal to Σ-SPL Transformation Rules

Code A.52 (DMPCompose.sums)

```
DMPCompose.sums := self >> Cond(
    self.isPermutation(),
    Prm(Rows(self),
        ApplyFunc(fCompose, List(Reversed(self.children()), (c) -> SumsSPL(c).direct)),
        ApplyFunc(fCompose, List(self.children(), (c) -> SumsSPL(c).inverse))),
    DMPCompose(self.p,Map(self.children(), (c) -> SumsSPL(c))
    )
);
```

Code A.53 (DMPComm.sums)

```
DMPComm.sums := self >> self;
```

Code A.54 (DMPGlobalTranspose.sums)

```
DMPGlobalTranspose.sums := self >> self;
```

Code A.55 (DMPTensor.sums) *Corresponding to Rule 5.8.*

```
DMPTensor.sums :=self >> Let(
    A => self.child(2),
    p => self.p,
    i => Ind(self.p),
    DMPISum(i, i.range,
        DMPScat(i,p,fId(Rows(A))) *
        SumsSPL(A) *
        DMPGath(i,p,fId(Rows(A)))
    ,self.pv)
);
```

Code A.56 (DMPIterDirectSum.sums)

```
DMPIterDirectSum.sums := self >> let(
    bkcols := Cols(self.child(1)),
    bkrows := Rows(self.child(1)),
    nblocks := self.domain,
    cols := Cols(self),
    rows := Rows(self),
    DMPISum(self.var, self.domain,
        Compose(
            DMPScat(self.var,nblocks,fId(bkrows)),
            SumsSPL(self.expr),
            DMPGath(self.var,nblocks,fId(bkcols)))
    ,self.pv)
);
```

A.6 Σ-SPL Optimization Rules

Code A.57 (DMPComposeAssoc) *Corresponding to Rule 5.10.*

```
DMPComposeAssoc := ARule( DMPCompose, [ @(1,DMPCompose) ],
    e -> [@(1).val.p, @(1).val.children()])
```

Code A.58 (DiagDMPISumLeft) *Corresponding to Rule 5.11.*

```
DiagDMPISumLeft := ARule( DMPCompose,
    [ @(1, DMPISum, canReorder) , @(2, Diag) ],
        e -> [ e.p, DMPISum(@1.val.var,
                            @1.val.domain,
                            @1.val.expr * @2.val,
                            @1.val.pv).attrs(@(1).val) ])
```

Code A.59 (DiagDMPISumRight) *Corresponding to Rule 5.11.*

```
DiagDMPISumRight := ARule( DMPCompose,
    [ @(1, Diag), @(2,[DMPISum] , canReorder) ],
        e -> [ e.p, DMPISum(@2.val.var,
                            @2.val.domain,
                            @1.val * @2.val.expr,
                            @2.val.pv).attrs(@(2).val) ])
```

Code A.60 (CommuteDMPGathDiag) *Corresponding to Rule 5.12.*

```
CommuteDMPGathDiag := ARule( Compose,
    [ @(1, DMPGath), @(2, Diag) ],
        e -> [ Diag(fCompose(@2.val.element, @1.val.func)).attrs(@(2).val), @1.val ])
```

Code A.61 (CommuteDiagDMPScat) *Corresponding to Rule 5.12.*

```
CommuteDiagDMPScat := ARule( Compose,
    [ @(1, Diag), @(2, DMPScat) ], # <-1 <-2 o
        e -> [ @2.val, Diag(fCompose(@1.val.element, @2.val.func)).attrs(@(1).val) ])
```

Code A.62 (MergeDMPISums) *Corresponding to Rule 5.13.*

```
MergeDMPISums := ARule(DMPCompose, [ @(1, [DMPISum]), @(2, [DMPISum]) ],
    e -> [ e.p, DMPISum( @(1).val.var , @(1).val.domain ,
        Compose(
            @(1).val.children() ,
            @(2).val.setVar(@(1).val.var).children()), @(1).val.pv)])
```

Code A.63 (ComposeDMPGathDMPScat) *Corresponding to Rule 5.14.*

```
ComposeDMPGathDMPScat := ARule(Compose,
    [ @(1, DMPGath), @(2, DMPScat).cond(
        x -> IsEqualObj( @(1).val.expr , x.expr) and @(1).val.var = x.var) ],
            e -> [ ]
    )
```

A.7 Σ-SPL Complex to Real Transformation Rules

Code A.64 (RCDMPCompose) *Corresponding to Rule 5.15.*

```
RCDMPCompose := Rule([RC, @(1, DMPCompose)],
     e -> DMPCompose(@(1).val.p, List(@(1).val.children(), RC)))
```

Code A.65 (RCDMPISum) *Corresponding to Rule 5.16.*

```
RCDMPISum := Rule([RC, @(1, DMPISum)],
     e -> DMPISum(@(1).val.var, @(1).val.domain, RC(@(1).val.child(1)), @(1).val.pv ))
```

Code A.66 (RCDMPGath) *Corresponding to Rule 5.17.*

```
RCDMPGath := Rule([RC, @(1, DMPGath)],
     e -> DMPGath(@(1).val.var,@(1).val.domain, fTensor(@(1).val.expr, fId(2))))
```

Code A.67 (RCDMPScat) *Corresponding to Rule 5.17.*

```
RCDMPScat := Rule([RC, @(1, DMPScat)],
     e -> DMPScat(@(1).val.var,@(1).val.domain,fTensor(@(1).val.expr, fId(2))))
```

Code A.68 (RCDMPComm) *Corresponding to Rule 5.18.*

```
RCDMPComm := Rule([RC, @(1, DMPComm)],
     e -> DMPComm(@(1).val.p,
         @(1).val.n*2,
         fTensor( @(1).val.w_jk, fId(2) ),
         @(1).val.pi,
         @(1).val.piinv,
         fTensor( @(1).val.r_jk, fId(2) ),
         @(1).val.j,
         @(1).val.k))
```

Code A.69 (RCDMPGlobalTranspose) *Corresponding to Rule 5.19.*

```
RCDMPGlobalTranspose := Rule([RC, @(1, DMPGlobalTranspose)],
     e -> DMPGlobalTranspose( @(1).val.n*2 , @(1).val.pv ))
```

A.8 iCode Objects and C Unparser

For the benefit of a better overview the unparser functions are appended directly to each object's definition. Acutally, they are rather member methods of the DMPUnparser class. The functions for printing the header and footer of a C-program are listed in Appendix A.10. Apart from that the class is inherits everything from CUnparser.

Code A.70 (dmploop) *Corresponding to Definition 5.16.*

```
Class(dmploop, Command, rec(
    __call__ := meth(self, loopvar, range, cmd, pv)
        local result;
        Constraint(IsVar(loopvar));
        Constraint(IsCommand(cmd));
        range := toRange(range);
        if range = 0 then
            return skip();
        else
```

```
            loopvar.setRange(range);
            range := listRange(range);
            result := WithBases(self,
               rec(operations := CmdOps, cmd := cmd,
                   var := loopvar, range := range, pv:=pv));
            loopvar.isLoopIndex := true;
            return result;
        fi;
    end,

    rChildren := decl.rChildren,
    rSetChild := decl.rSetChild,

    print := (self, i, is) >> Chain(
        Print(self.name, "(", self.var, ", ", self.range, ",\n", Blanks(i+is)),
        self.cmd.print(i+is, is),
        Print("\n", Blanks(i), ")")),

    free := meth(self)
        local c;
        c := self.cmd.free();
        SubtractSet(c, Set([self.var]));
        return c;
    end,

    isDMP := self >> true,
    doUnroll := false,
    unroll := self >> Copy(self)
))
```

Unparser

```
dmploop := meth(self, o, i, is)
    local v, lo, hi;
    Constraint(IsRange(o.range));
    v := o.var; lo := o.range[1]; hi := Last(o.range);
    Print(Blanks(i),"{ /* dmploop */\n");
    Print(Blanks(i+is),"int ", v, " = mpirank;\n");

    if o.pv.isDMPRescale and o.pv.p <> o.pv.p_high then
        Print(Blanks(i+is),"if (!(",v,"%",o.pv.p_high/o.pv.p_low,")) {\n");
        self(o.cmd, i+2*is, is);
        Print(Blanks(i+is),"}\n");
    else
        self(o.cmd, i+is, is);
    fi;
    Print(Blanks(i), "}\n");
end,
```

Code A.71 (dmpcomm)

```
Class(dmpcomm, Command, rec(
    __call__ := (self, loc1, loc2, p, n, pi, piinv) >> WithBases(self,
        rec(operations := CmdOps,
            loc1 := toAssignTarget(loc1),
            loc2 := toAssignTarget(loc2),
            p := p,
            n := n,
            pi := pi,
            piinv := piinv)),
    rChildren := self >> [], #[self.loc1, self.loc2],
    free := self >> [],
```

```
    print := (self,i,is) >> Print(self.name,
        "(", self.loc1,", ",self.loc2,", ",self.p,", ",self.n,", ",self.pi,")"),
    doUnroll := false,
    isDMP := self >> true
))
```

Unparser

```
dmpcomm := meth(self, o, i, is)
    local pilist, piinvlist, m, alpha, mypi, mypiinv, listunparser;

    m       :=        o.pi.N / o.p;
    mypi    := (j) -> fCompose(o.pi,fTensor(fBase(o.p,j),fId(m)));
    mypiinv := (j) -> fCompose(o.piinv,fTensor(fBase(o.p,j),fId(m)));
    alpha   := (i) -> [Int(i/m), Mod(i,m)];

    pilist := List([0..o.p-1],
        i-> List([0..m-1],
            block-> alpha(mypi(i).lambda().at(block).ev())
        )
    );
    piinvlist := List([0..o.p-1],
        i-> List([0..m-1],
            block-> alpha(mypiinv(i).lambda().at(block).ev())
        )
    );

    listunparser := function ( lst )
        local  i;
        if not IsList(lst) then Print(lst); return; fi;
        Print("{");
        listunparser(lst[1]);
        for i  in [ 2 .. Length(lst) ]  do Print(", ", listunparser(lst[i])); od;
        Print("}");
    end;

    Print("\n/*============ GLOBAL COMM below ============*/\n");
    Print(Blanks(i), "{\n", Blanks(is-1));
    Print(Blanks(i+is),
        "MPI_Request reqs[2][", m ,"];\n");
    Print(Blanks(i+is), "int mpii;\n");
    Print(Blanks(i+is),
        "static commpat    pi[",o.p,"][",m,"] = ",listunparser(pilist),";\n");
    Print(Blanks(i+is),
        "static commpat piinv[",o.p,"][",m,"] = ",listunparser(piinvlist),";\n");
    Print(Blanks(i+is),"for(mpii=0;mpii<",m,";mpii++){\n");
    Print(Blanks(i+2*is),
        "MPI_Isend(",o.loc2.cprint(),"+",o.n,"*mpii,",o.n,
        ",MPI_DOUBLE,piinv[mpirank][mpii].proc,piinv[mpirank]",
        "[mpii].offset,MPI_COMM_WORLD,reqs[0]+mpii);\n");
    Print(Blanks(i+2*is),"MPI_Irecv(",o.loc1.cprint(),"+",o.n,
        "*mpii,",o.n,",MPI_DOUBLE,pi[mpirank][mpii].proc,mpii,",
        "MPI_COMM_WORLD,reqs[1]+mpii);\n");
    Print(Blanks(i+is),"}\n");
    Print(Blanks(i+is),"MPI_Waitall(",m,",reqs[0],MPI_STATUSES_IGNORE);\n");
    Print(Blanks(i+is),"MPI_Waitall(",m,",reqs[1],MPI_STATUSES_IGNORE);\n");
    Print(Blanks(i), "}\n");
    Print("\n/*============ GLOBAL COMM above ============*/\n");
end,
```

Code A.72 (dmpglobaltranspose)

```
Class(dmpglobaltranspose, Command, rec(
```

```
    __call__ := ( self, loc1 , loc2 , pv , n ) >> WithBases(self,
        rec(operations := CmdOps,
            loc1 := toAssignTarget(loc1),
            loc2 := toAssignTarget(loc2),
            pv:= pv,
            p := pv.p,
            n := n)),
    rChildren := self >> [self.loc1, self.loc2],
    rSetChild := rSetChildFields("loc1", "loc2"),
    print := (self,i,is) >> Print(self.name,
        "(",self.loc1,", ",self.loc2,", ",self.pv,", ",self.n,")"),
    isDMP := self >> true
))
```

Unparser

```
dmpglobaltranspose := meth(self,o,i,is)
    Print("\n/*============ GLOBAL COMM below ============*/\n");

    if o.pv.isDMPRescale then
        Print(Blanks(i),"if (!(mpirank%",o.pv.p_high/o.pv.p_low,")) {\n");
        Print(Blanks(i+is), "int scalefac = ",o.pv.p_high/o.pv.p_low,";\n");
        Print(Blanks(i+is), "int myrank = mpirank/",o.pv.p_high/o.pv.p_low,";\n");
        Print(Blanks(i+is), "int mpii;\n");
        Print(Blanks(i+is), "MPI_Status stat;\n");
        Print(Blanks(i+is),"for(mpii=1;mpii<",o.p,";mpii++){\n");
        Print(Blanks(i+2*is),"MPI_Sendrecv_replace(",o.loc1.cprint(),"+",o.n,
            "*(mpii^myrank),",o.n,",MPI_DOUBLE,(mpii^myrank)*scalefac",
            ",0 /*sendtag*/,(mpii^myrank)*scalefac,0 /*recvtag*/,",
            "MPI_COMM_WORLD,&stat);\n");
        Print(Blanks(i+is),"}\n");
        Print(Blanks(i), "}", Blanks(is-1));
    else
        Print(Blanks(i), "{\n");
        Print(Blanks(i+is), "int mpii;\n");
        Print(Blanks(i+is), "MPI_Status stat;\n");
        Print(Blanks(i+is),"for(mpii=1;mpii<",o.p,";mpii++){\n");
        Print(Blanks(i+2*is),"MPI_Sendrecv_replace(",o.loc1.cprint(),"+",o.n,
            "*(mpii^mpirank),",o.n,",MPI_DOUBLE,mpii^mpirank,0 ",
            "/*sendtag*/,mpii^mpirank,0 /*recvtag*/,MPI_COMM_WORLD,&stat);\n");
        Print(Blanks(i+is),"}\n");
        Print(Blanks(i), "}", Blanks(is-1));
    fi;
    Print("\n/*============ GLOBAL COMM above ============*/\n");
end,
```

Code A.73 (dmpglobaltranspose_rescale)

```
Class(dmpglobaltranspose_rescale, Command, rec(
    __call__ := ( self, loc1 , loc2 , pv , n ) >> WithBases(self,
        rec(operations := CmdOps,
            loc1 := toAssignTarget(loc1),
                loc2 := toAssignTarget(loc2),
                pv := pv,
                n := n)),
    rChildren := self >> [self.loc1, self.loc2],
    rSetChild := rSetChildFields("loc1", "loc2"),
    print := (self,i,is) >> Print(self.name,
        "(",self.loc1,", ",self.loc2,", ",self.pv,", ",self.n,")"),
    isDMP := self >> true
))
```

Unparser

```
dmpglobaltranspose_rescale := meth(self,o,i,is)
    local listunparser, scalefac,p,q, sendblks,
        recvblks, bpp_in, bpp_out,adr_in,adr_out,blockdest,list;

    p:=o.pv.p_high;
    q:=o.pv.p_low;
    scalefac:= p/q;

    bpp_in    := Cond(o.pv.isDMPRescaleDown,p,(p^2)/q);
    bpp_out   := Cond(o.pv.isDMPRescaleDown,(p^2)/q,p);

    adr_in  := (i) -> [Int(i/bpp_in)  *
        Cond(o.pv.isDMPRescaleUp,scalefac,1)   , Mod(i,bpp_in)];
    adr_out := (i) -> [Int(i/bpp_out) *
        Cond(o.pv.isDMPRescaleDown,scalefac,1) , Mod(i,bpp_out)];

    blockdest := (i) -> p*Mod(i,p) + Int(i/p);

    list:=List([0..(p^2)-1],i-> [adr_in(i),adr_out(blockdest(i))]);

    listunparser := function ( lst )
        local  i;
        if not IsList(lst) then Print(lst); return; fi;
        Print("{");
        listunparser(lst[1]);
        for i  in [ 2 .. Length(lst) ]  do Print(", ", listunparser(lst[i])); od;
        Print("}");
    end;

    if o.pv.isDMPRescaleDown then
        Print("\n/************* GLOBAL COMM below --- DOWNSCALE ",
            o.loc2.cprint(), "-->", o.loc1.cprint() ," ************/\n");
        Print(Blanks(i), "{\n");
        Print(Blanks(i+is),  "MPI_Request req_send[",p,"];\n");
        Print(Blanks(i+is),  "MPI_Request req_recv[",scalefac*(p-1),"];\n");
    else
        Print("\n/************* GLOBAL COMM below ---   UPSCALE ",
            o.loc2.cprint(), "-->", o.loc1.cprint() ," ************/\n");
        Print(Blanks(i), "{\n");
        Print(Blanks(i+is),  "MPI_Request req_send[",scalefac*(p-1),"];\n");
        Print(Blanks(i+is),  "MPI_Request req_recv[",p,"];\n");
    fi;

    Print(Blanks(i+is),  "int sendctr=0, recvctr=0, i, j;\n");
    Print(Blanks(i+is),  "int pi[",p^2,"][2][2] = ",listunparser(list),";\n");
    Print(Blanks(i+is),  "/* issue mpi comm */\n");
    Print(Blanks(i+is),  "for(i=0;i<",p^2,";i++) {\n");
    Print(Blanks(i+is+is),  "if(pi[i][0][0]==mpirank && ",
        "pi[i][0][0]!=pi[i][1][0]) {\n");
    Print(Blanks(i+is+is+is), "MPI_Isend(",o.loc2.cprint(),"+",o.n,
        "*pi[i][0][1],",o.n,
        ",MPI_DOUBLE,pi[i][1][0],pi[i][0][0]*mpisize+pi[i][0][1],",
        "MPI_COMM_WORLD,&(req_send[sendctr++]));\n");
    Print(Blanks(i+is+is),  "}\n");
    Print(Blanks(i+is+is),  "if(pi[i][1][0]==mpirank && ",
        "pi[i][0][0]!=pi[i][1][0]) {\n");
    Print(Blanks(i+is+is+is),  "MPI_Irecv(",o.loc1.cprint(),"+",o.n,
        "*pi[i][1][1],",o.n,
        ",MPI_DOUBLE,pi[i][0][0],pi[i][0][0]*mpisize+pi[i][0][1],",
        "MPI_COMM_WORLD,&(req_recv[recvctr++]));\n");
    Print(Blanks(i+is+is),  "}\n");
    Print(Blanks(i+is),  "}\n");

    Print(Blanks(i+is),  "/* issue local perms */\n");
    Print(Blanks(i+is),  "for(i=0;(!(mpirank%",scalefac,")) && i<",
```

```
        p^2,";i++) \n");
    Print(Blanks(i+is+is),  "if(pi[i][0][0]==mpirank && ",
        "pi[i][0][0]==pi[i][1][0]) \n");
    Print(Blanks(i+is+is+is),  "for(j=0;j<",o.n,";j++)\n");
    Print(Blanks(i+is+is+is+is),  o.loc1.cprint(),"[j+",o.n,"*pi[i][1][1]]=",
        o.loc2.cprint(),"[j+",o.n,"*pi[i][0][1]];\n");
    Print(Blanks(i+is),  "/* wait for mpi comm */\n");
    Print(Blanks(i+is),  "if(recvctr>0) {\n");
    Print(Blanks(i+is+is),  "MPI_Waitall(recvctr, req_recv,
        MPI_STATUSES_IGNORE);\n");
    Print(Blanks(i+is),  "}\n");
    Print(Blanks(i), "}\n");
    if o.pv.isDMPRescaleDown then
        Print("/*============ GLOBAL COMM above --- DOWNSCALE ",
            o.loc2.cprint(), "-->", o.loc1.cprint() ," ============*/\n");
    else
        Print("/*============ GLOBAL COMM above ---   UPSCALE ",
            o.loc2.cprint(), "-->", o.loc1.cprint() ," ============*/\n");
    fi;
end,
```

A.9 Σ-SPL to iCode Transformation Rules

Code A.74 (`DMPISum.code`) *Corresponding to Rule 5.21.*

```
DMPISum.code := meth(self, y, x)
    local ret;
    ret := dmploop(self.var, self.domain, _CodeSums(self.child(1), y, x), self.pv);
    return ret;
end;

DMPISum.ipcode := meth(self, x)
    local ret;
    ret := self.code(x, x);
    return ret;
end;
```

Code A.75 (`DMPCompose.code`) *Corresponding to Rule 5.20.*

```
DMPCompose.code := meth(self,y,x)
    local ch, numch, vecs, allow, i, ret, ret2;
    ch := Filtered(self.children(), i-> not i.name in ["DMPGath","DMPScat"]);
    numch := Length(ch);
    vecs := [y];
    allow := (x<>y);

    for i in [1..numch-1] do
        if ObjId(ch[i])=DMPGlobalTranspose and
        (
            (not ch[i].pv.isDMPRescale) or
            (not ch[i].pv.implements_rescale())
        ) then
            # global transpose without rescaling -> inplace
            vecs[i+1] := vecs[i];
        else
            if ObjId(ch[i])=DMPGlobalTranspose and ch[i].pv.implements_rescale() then
                # global transpose with rescaleing
                if ch[i].pv.isDMPRescaleUp then
```

```
                    vecs[i+1] := TempVec(TArray(TempArrayType(y, x), Cols(ch[i])/ch[i].pv.p_low));
                else
                    vecs[i+1] := TempVec(TArray(TempArrayType(y, x), Cols(ch[i])/ch[i].p));
                fi;
            else
                # standard parallel block
                vecs[i+1] := TempVec(TArray(TempArrayType(y, x), Cols(ch[i])/ch[i].p));
            fi;
        fi;
    od;

    vecs[numch+1] := x;
    for i in Reversed([1..numch]) do
        if allow and ObjId(ch[i])=Inplace then
            vecs[i] := vecs[i+1];
        fi;
    od;

    if vecs[1] = vecs[numch+1] then # everything was inplace
        vecs[1] := y;
    fi;

    if vecs[1] = vecs[numch+1] then # everything was inplace
        vecs[numch+1] := x;
    fi;

    vecs := Reversed(vecs);
    ch := Reversed(ch);
    ret := chain(
        List([1..numch],
            i -> When(vecs[i+1]=vecs[i],
                compiler._IPCodeSums(ch[i], vecs[i]),
                _CodeSums(ch[i], vecs[i+1], vecs[i])
        )));
    ret.isDMP := self >> true;
    ret.noopt := true;
    return decl(Difference(vecs{[2..Length(vecs)-1]}, [x,y]),ret);
end;
```

Code A.76 (`Compose.code`) *Corresponding to Rule 5.22. This function is part of original* SPIRAL. *It has been modified for* SPIRAL/DMP *to ignore* ***`DMPGath`*** *and* ***`DMPScat`*** *child objects.*

```
Compose.code := meth(self,y,x)
    local ch, numch, vecs, allow, i;
#VIENNA filtering DMPGath, DMPScat out here because they do not generate code
#    ch := self.children();
    ch := Filtered(self.children(), i-> not i.name in ["DMPGath","DMPScat"]);
    numch := Length(ch);
    vecs := [y];
    allow := (x<>y);
    for i in [1..numch-1] do
        if allow and ObjId(ch[i])=Inplace then vecs[i+1] := vecs[i];
        else vecs[i+1] := TempVec(TArray(TempArrayType(y, x), Cols(ch[i])));
        fi;
    od;
    vecs[numch+1] := x;
    for i in Reversed([1..numch]) do
        if allow and ObjId(ch[i])=Inplace then vecs[i] := vecs[i+1];
        fi;
    od;
```

Code A.77 (`DMPComm.code`) *Corresponding to Rule 5.23.*

```
DMPComm.code:= meth(self, y, x)
    local m, p, n, pi, piinv, scat, gath, prm, t1, t2, c, i, dmploopix, ret;
    p := self.p;
    m := self.pi.N / p;
    n := self.n;
    pi := self.pi;
    piinv := self.piinv;
    gath:= FTDA(DMPIterDirectSum(self.j,self.j.range,IterVStack(self.k,self.k.range,Gath(self.r_jk))).sums());
    scat:= FTDA(DMPIterDirectSum(self.j,self.j.range,IterHStack(self.k,self.k.range,Scat(self.w_jk))).sums());
    t1 := TempVec(TArray(Global.GetArrayType(y, x), m * n ));
    t2 := TempVec(TArray(Global.GetArrayType(y, x), m * n ));
    t1.t.mp := rec(procs := p, size := m * n * p);
    t2.t.mp := rec(procs := p, size := m * n * p);

    c := chain(
        gath.code(t1,x),
        dmpcomm(t2, t1, p, n, pi, piinv) ,
        scat.code(x, t2));
    c.isDMP := self >> true;

    c := decl([t1,t2],c);

    i := Ind();
    dmploopix := Ind();
    ret := chain(c,
        dmploop(dmploopix, p,
            loop(i,Rows(self)/p,
                assign(nth(y,i),nth(x,i))
            )
        )
    );
    ret.isDMP := self >> true;
    ret.noopt := true;
    return ret;
end;
```

Code A.78 (`DMPGlobalTranspose.code`)

```
DMPGlobalTranspose.code:= meth(self, y, x)
    local p,pv, n, i, ret;
    pv := self.pv;
    p  := pv.p;
    n  := self.n;
    i  := Ind();

    if not (pv.isDMPRescale and pv.implements_rescale()) then
        # standard global transpose code
        if y=x then
            ret := dmpglobaltranspose(x,x,pv,n);
        else
            ret := chain(
                dmpglobaltranspose(x,x,pv,n),
                loop(i,Rows(self)/p,assign(nth(y,i),nth(x,i))));
        fi;
    else
        # Note: CUnparser for dmpglobaltranspose_rescale should to everything
        #       includig local copying
        ret := dmpglobaltranspose_rescale(y, x, pv, n);
    fi;
    ret.isDMP := self >> true;
```

```
    return ret;
end;

DMPGlobalTranspose.ipcode := (self, x) >> self.code(x, x);
```

A.10 iCode Unparser to C-Code

Code A.79 (DMPUnparser.header)

```
header := meth(self, subname, o)
    local loopvars, precomputed_data;
    Print(self.copyright);
    DoForAll(self.includes, inc -> Print("#include ", inc, "\n"));

    if IsBound(o.dimensions) then Print("/* ", o.dimensions, " */\n"); fi;
    Print("#include <mpi.h>\n");

    if IsBound(o.runtime_data) then
        Print(self.omega_decl);
        DoForAll(o.runtime_data, x->
            Print(self.arrayDataModifier, " ", self.declare(x.t, x, 0, 4), ";\n"));
        Print("\n");
    fi;

    Print("#ifndef COMPLEX_T\n");
    Print("  typedef struct { double r,i; } complex_t;\n");
    Print("  #define COMPLEX_T\n");
    Print("#endif\n");
    Print("\n");
    Print("typedef struct {int proc, offset;} commpat;\n");
    Print("int mpirank, mpisize;\n");
    Print("\n");

    precomputed_data := List(Collect(o, data), x->[x.var, x.value]);
    DoForAll(precomputed_data, d -> self.genData(d[1], d[2]));

    Print("void init_",subname," ();\n");
    Print("void ", subname, "(", TDouble.ctype, " *Y, ", TDouble.ctype, " *X) {\n");

    loopvars := List(Collect(o, @(1).cond(IsLoop)), x->x.var);
    loopvars := Set(loopvars);
    if loopvars <> [] then Print(Blanks(4), "int ", PrintCS(loopvars), ";\n"); fi;
end,
```

Code A.80 (DMPUnparser.footer)

```
footer := meth(self, subname, o)
    local init,loopvars;
    Print("}\n");
    Print("\n");
    Print("void init_", subname, "() {\n");
    Print("    MPI_Comm_size(MPI_COMM_WORLD, &mpisize);\n");
    Print("    MPI_Comm_rank(MPI_COMM_WORLD, &mpirank);\n");
    if IsBound(o.runtime_init) then # unparse initialization code
    loopvars := Union(List(o.runtime_init, cc -> List(Collect(cc, @(1).cond(IsLoop)), x->x.var)));
    if loopvars <> [ ]  then
        Print(Blanks(4), "int ", PrintCS(loopvars), ";\n"); fi;
        for init in o.runtime_init do
            self(SReduce(init), 4, 4);
```

```
        od;
    fi;
    Print("}\n");
end
```

Appendix B

Miscellaneous Source Codes

Code B.1 (iCode) The Σ-SPL formula from Example 5.12 converted to iCode.

```
decl([ T1, T2 ],
   chain(
      dmploop(i62, [ 0 .. 1 ],
         chain(
            assign(nth(T2, 0), nth(X, 0)),
            assign(nth(T2, 1), nth(X, 1)),
            assign(nth(T2, 2), nth(X, 2)),
            assign(nth(T2, 3), nth(X, 3)),
            assign(nth(T2, 4), nth(X, 8)),
            assign(nth(T2, 5), nth(X, 9)),
            assign(nth(T2, 6), nth(X, 10)),
            assign(nth(T2, 7), nth(X, 11)),
            assign(nth(T2, 8), nth(X, 16)),
            assign(nth(T2, 9), nth(X, 17)),
            assign(nth(T2, 10), nth(X, 18)),
            assign(nth(T2, 11), nth(X, 19)),
            assign(nth(T2, 12), nth(X, 24)),
            assign(nth(T2, 13), nth(X, 25)),
            assign(nth(T2, 14), nth(X, 26)),
            assign(nth(T2, 15), nth(X, 27)),
            assign(nth(T2, 16), nth(X, 4)),
            assign(nth(T2, 17), nth(X, 5)),
            assign(nth(T2, 18), nth(X, 6)),
            assign(nth(T2, 19), nth(X, 7)),
            assign(nth(T2, 20), nth(X, 12)),
            assign(nth(T2, 21), nth(X, 13)),
            assign(nth(T2, 22), nth(X, 14)),
            assign(nth(T2, 23), nth(X, 15)),
            assign(nth(T2, 24), nth(X, 20)),
            assign(nth(T2, 25), nth(X, 21)),
            assign(nth(T2, 26), nth(X, 22)),
            assign(nth(T2, 27), nth(X, 23)),
            assign(nth(T2, 28), nth(X, 28)),
            assign(nth(T2, 29), nth(X, 29)),
            assign(nth(T2, 30), nth(X, 30)),
            assign(nth(T2, 31), nth(X, 31))
         )
      ),
      dmpglobaltranspose(T2, T2, AParDistr(2), 16),
      dmploop(i51, [ 0 .. 1 ],
         chain(
            decl([ t140, t158, t136, t135, t134, t133, t139, t129, t130, t131, t132, t137, t138,
                   a19, t156, s66, a18, a17, s69, s68, t142, t141, s67, a20, s73, t157, a21,
                   t117, s72, t118, t119, t120, t121, t122, t123, t124, t125, t126, t127, t128,
                   t162, t161, a24, s71, t160, t169, t159, t170, t155, t154, t153, t152, t151,
                   t150, t149, t148, t147, t146, t145, t171, t172, t144, t143, t164, t163, t165,
                   a23, a22, t168, s70, t167, t166 ],
               chain(
                  assign(t117, add(nth(T2, 0), nth(T2, 16))),
                  assign(t118, add(nth(T2, 8), nth(T2, 24))),
                  assign(t119, add(t117, t118)),
```

```
assign(t120, sub(t117, t118)),
assign(t121, add(nth(T2, 1), nth(T2, 17))),
assign(t122, add(nth(T2, 9), nth(T2, 25))),
assign(t123, sub(t121, t122)),
assign(t124, add(t121, t122)),
assign(t125, sub(nth(T2, 0), nth(T2, 16))),
assign(t126, sub(nth(T2, 9), nth(T2, 25))),
assign(t127, sub(t125, t126)),
assign(t128, add(t125, t126)),
assign(t129, sub(nth(T2, 1), nth(T2, 17))),
assign(t130, sub(nth(T2, 8), nth(T2, 24))),
assign(t131, add(t129, t130)),
assign(t132, sub(t129, t130)),
assign(t133, add(nth(T2, 4), nth(T2, 20))),
assign(t134, add(nth(T2, 12), nth(T2, 28))),
assign(t135, add(t133, t134)),
assign(t136, sub(t133, t134)),
assign(t137, add(nth(T2, 5), nth(T2, 21))),
assign(t138, add(nth(T2, 13), nth(T2, 29))),
assign(t139, sub(t137, t138)),
assign(t140, add(t137, t138)),
assign(a17, mul(0.70710678118654746, sub(nth(T2, 4), nth(T2, 20)))),
assign(a18, mul(0.70710678118654746, sub(nth(T2, 5), nth(T2, 21)))),
assign(s66, sub(a17, a18)),
assign(a19, mul(0.70710678118654746, sub(nth(T2, 13), nth(T2, 29)))),
assign(a20, mul(0.70710678118654746, sub(nth(T2, 12), nth(T2, 28)))),
assign(s67, add(a20, a19)),
assign(t141, add(s66, s67)),
assign(t142, sub(s66, s67)),
assign(s68, add(a17, a18)),
assign(s69, sub(a20, a19)),
assign(nth(T1, 0), add(t119, t135)),
assign(nth(T1, 1), add(t124, t140)),
assign(nth(T1, 16), sub(t119, t135)),
assign(nth(T1, 17), sub(t124, t140)),
assign(nth(T1, 8), sub(t120, t139)),
assign(nth(T1, 9), add(t123, t136)),
assign(nth(T1, 24), add(t120, t139)),
assign(nth(T1, 25), sub(t123, t136)),
assign(t143, add(s68, s69)),
assign(t144, sub(s68, s69)),
assign(nth(T1, 4), add(t127, t142)),
assign(nth(T1, 20), sub(t127, t142)),
assign(nth(T1, 5), add(t131, t143)),
assign(nth(T1, 21), sub(t131, t143)),
assign(nth(T1, 13), add(t132, t141)),
assign(nth(T1, 29), sub(t132, t141)),
assign(nth(T1, 12), sub(t128, t144)),
assign(nth(T1, 28), add(t128, t144)),
assign(t145, add(nth(T2, 2), nth(T2, 18))),
assign(t146, add(nth(T2, 10), nth(T2, 26))),
assign(t147, add(t145, t146)),
assign(t148, sub(t145, t146)),
assign(t149, add(nth(T2, 3), nth(T2, 19))),
assign(t150, add(nth(T2, 11), nth(T2, 27))),
assign(t151, add(t149, t150)),
assign(t152, sub(t149, t150)),
assign(t153, sub(nth(T2, 2), nth(T2, 18))),
assign(t154, sub(nth(T2, 11), nth(T2, 27))),
assign(t155, add(t153, t154)),
assign(t156, sub(t153, t154)),
assign(t157, sub(nth(T2, 3), nth(T2, 19))),
assign(t158, sub(nth(T2, 10), nth(T2, 26))),
assign(t159, sub(t157, t158)),
assign(t160, add(t157, t158)),
```

```
            assign(t161, add(nth(T2, 6), nth(T2, 22))),
            assign(t162, add(nth(T2, 14), nth(T2, 30))),
            assign(t163, add(t161, t162)),
            assign(t164, sub(t161, t162)),
            assign(t165, add(nth(T2, 7), nth(T2, 23))),
            assign(t166, add(nth(T2, 15), nth(T2, 31))),
            assign(t167, sub(t165, t166)),
            assign(t168, add(t165, t166)),
            assign(a21, mul(0.70710678118654746, sub(nth(T2, 6), nth(T2, 22)))),
            assign(a22, mul(0.70710678118654746, sub(nth(T2, 7), nth(T2, 23)))),
            assign(s70, sub(a21, a22)),
            assign(a23, mul(0.70710678118654746, sub(nth(T2, 15), nth(T2, 31)))),
            assign(a24, mul(0.70710678118654746, sub(nth(T2, 14), nth(T2, 30)))),
            assign(s71, add(a24, a23)),
            assign(t169, sub(s70, s71)),
            assign(t170, add(s70, s71)),
            assign(s72, add(a21, a22)),
            assign(s73, sub(a24, a23)),
            assign(nth(T1, 2), add(t147, t163)),
            assign(nth(T1, 3), add(t151, t168)),
            assign(nth(T1, 18), sub(t147, t163)),
            assign(nth(T1, 19), sub(t151, t168)),
            assign(nth(T1, 10), sub(t148, t167)),
            assign(nth(T1, 11), add(t152, t164)),
            assign(nth(T1, 26), add(t148, t167)),
            assign(nth(T1, 27), sub(t152, t164)),
            assign(t171, add(s72, s73)),
            assign(t172, sub(s72, s73)),
            assign(nth(T1, 6), add(t156, t169)),
            assign(nth(T1, 22), sub(t156, t169)),
            assign(nth(T1, 7), add(t160, t171)),
            assign(nth(T1, 23), sub(t160, t171)),
            assign(nth(T1, 15), add(t159, t170)),
            assign(nth(T1, 31), sub(t159, t170)),
            assign(nth(T1, 14), sub(t155, t172)),
            assign(nth(T1, 30), add(t155, t172))
         )
      )
   )
),
dmpglobaltranspose(T1, T1, AParDistr(2), 16),
dmploop(i43, [ 0 .. 1 ],
   chain(
      decl([ t266, t267, t268, t252, t251, t254, t241, t243, t242, t253, t249, t265,
             t246, t240, t255, t239, t256, t238, t257, t237, t258, t259, t260, t261,
             t262, t263, t264, t245, t250, t244, t247, t248 ],
         chain(
            assign(t237, add(nth(T1, 0), nth(T1, 16))),
            assign(t238, add(nth(T1, 2), nth(T1, 18))),
            assign(t239, add(nth(T1, 1), nth(T1, 17))),
            assign(t240, add(nth(T1, 3), nth(T1, 19))),
            assign(t241, sub(nth(T1, 0), nth(T1, 16))),
            assign(t242, sub(nth(T1, 3), nth(T1, 19))),
            assign(t243, sub(nth(T1, 1), nth(T1, 17))),
            assign(t244, sub(nth(T1, 2), nth(T1, 18))),
            assign(nth(Y, 0), add(t237, t238)),
            assign(nth(Y, 1), add(t239, t240)),
            assign(nth(Y, 4), sub(t237, t238)),
            assign(nth(Y, 5), sub(t239, t240)),
            assign(nth(Y, 2), sub(t241, t242)),
            assign(nth(Y, 3), add(t243, t244)),
            assign(nth(Y, 6), add(t241, t242)),
            assign(nth(Y, 7), sub(t243, t244)),
            assign(t245, add(nth(T1, 4), nth(T1, 20))),
            assign(t246, add(nth(T1, 6), nth(T1, 22))),
```

```
                assign(t247, add(nth(T1, 5), nth(T1, 21))),
                assign(t248, add(nth(T1, 7), nth(T1, 23))),
                assign(t249, sub(nth(T1, 4), nth(T1, 20))),
                assign(t250, sub(nth(T1, 7), nth(T1, 23))),
                assign(t251, sub(nth(T1, 5), nth(T1, 21))),
                assign(t252, sub(nth(T1, 6), nth(T1, 22))),
                assign(nth(Y, 8), add(t245, t246)),
                assign(nth(Y, 9), add(t247, t248)),
                assign(nth(Y, 12), sub(t245, t246)),
                assign(nth(Y, 13), sub(t247, t248)),
                assign(nth(Y, 10), sub(t249, t250)),
                assign(nth(Y, 11), add(t251, t252)),
                assign(nth(Y, 14), add(t249, t250)),
                assign(nth(Y, 15), sub(t251, t252)),
                assign(t253, add(nth(T1, 8), nth(T1, 24))),
                assign(t254, add(nth(T1, 10), nth(T1, 26))),
                assign(t255, add(nth(T1, 9), nth(T1, 25))),
                assign(t256, add(nth(T1, 11), nth(T1, 27))),
                assign(t257, sub(nth(T1, 8), nth(T1, 24))),
                assign(t258, sub(nth(T1, 11), nth(T1, 27))),
                assign(t259, sub(nth(T1, 9), nth(T1, 25))),
                assign(t260, sub(nth(T1, 10), nth(T1, 26))),
                assign(nth(Y, 16), add(t253, t254)),
                assign(nth(Y, 17), add(t255, t256)),
                assign(nth(Y, 20), sub(t253, t254)),
                assign(nth(Y, 21), sub(t255, t256)),
                assign(nth(Y, 18), sub(t257, t258)),
                assign(nth(Y, 19), add(t259, t260)),
                assign(nth(Y, 22), add(t257, t258)),
                assign(nth(Y, 23), sub(t259, t260)),
                assign(t261, add(nth(T1, 12), nth(T1, 28))),
                assign(t262, add(nth(T1, 14), nth(T1, 30))),
                assign(t263, add(nth(T1, 13), nth(T1, 29))),
                assign(t264, add(nth(T1, 15), nth(T1, 31))),
                assign(t265, sub(nth(T1, 12), nth(T1, 28))),
                assign(t266, sub(nth(T1, 15), nth(T1, 31))),
                assign(t267, sub(nth(T1, 13), nth(T1, 29))),
                assign(t268, sub(nth(T1, 14), nth(T1, 30))),
                assign(nth(Y, 24), add(t261, t262)),
                assign(nth(Y, 25), add(t263, t264)),
                assign(nth(Y, 28), sub(t261, t262)),
                assign(nth(Y, 29), sub(t263, t264)),
                assign(nth(Y, 26), sub(t265, t266)),
                assign(nth(Y, 27), add(t267, t268)),
                assign(nth(Y, 30), add(t265, t266)),
                assign(nth(Y, 31), sub(t267, t268))
            )
        )
    )
  )
)
```

Code B.2 (C-code) The iCode from Code B.1 unparsed to C-code.

```
/* [ 64, 64 ] */
#include <mpi.h>

typedef struct {int proc, offset;} commpat;
int mpirank, mpisize;

void init_sub ();
void sub(double *Y, double *X) {
```

```
    static double T1[32];
    static double T2[32];
    { /* dmploop */
      int i62 = mpirank;
      T2[0] = X[0];
      T2[1] = X[1];
      T2[2] = X[2];
      T2[3] = X[3];
      T2[4] = X[8];
      T2[5] = X[9];
      T2[6] = X[10];
      T2[7] = X[11];
      T2[8] = X[16];
      T2[9] = X[17];
      T2[10] = X[18];
      T2[11] = X[19];
      T2[12] = X[24];
      T2[13] = X[25];
      T2[14] = X[26];
      T2[15] = X[27];
      T2[16] = X[4];
      T2[17] = X[5];
      T2[18] = X[6];
      T2[19] = X[7];
      T2[20] = X[12];
      T2[21] = X[13];
      T2[22] = X[14];
      T2[23] = X[15];
      T2[24] = X[20];
      T2[25] = X[21];
      T2[26] = X[22];
      T2[27] = X[23];
      T2[28] = X[28];
      T2[29] = X[29];
      T2[30] = X[30];
      T2[31] = X[31];
    }

/*============ GLOBAL COMM below ============*/
    {
      int mpii;
      MPI_Status stat;
      for(mpii=1;mpii<2;mpii++){
        MPI_Sendrecv_replace(T2+16*(mpii^mpirank),16,MPI_DOUBLE,mpii^mpirank,0 /*sendtag*/,
                             mpii^mpirank,0 /*recvtag*/,MPI_COMM_WORLD,&stat);
      }
    }
/*============ GLOBAL COMM above ============*/
    { /* dmploop */
      int i51 = mpirank;
      double t140, t158, t136, t135, t134, t133, t139, t129, t130, t131, t132, t137, t138, a19,
             t156, s66, a18, a17, s69, s68, t142, t141, s67, a20, s73, t157, a21, t117, s72,
             t118, t119, t120, t121, t122, t123, t124, t125, t126, t127, t128, t162, t161,
             a24, s71, t160, t169, t159, t170, t155, t154, t153, t152, t151, t150, t149, t148,
             t147, t146, t145, t171, t172, t144, t143, t164, t163, t165, a23, a22, t168, s70,
             t167, t166;
      t117 = (T2[0] + T2[16]);
      t118 = (T2[8] + T2[24]);
      t119 = (t117 + t118);
      t120 = (t117 - t118);
      t121 = (T2[1] + T2[17]);
      t122 = (T2[9] + T2[25]);
      t123 = (t121 - t122);
      t124 = (t121 + t122);
      t125 = (T2[0] - T2[16]);
```

```
t126 = (T2[9] - T2[25]);
t127 = (t125 - t126);
t128 = (t125 + t126);
t129 = (T2[1] - T2[17]);
t130 = (T2[8] - T2[24]);
t131 = (t129 + t130);
t132 = (t129 - t130);
t133 = (T2[4] + T2[20]);
t134 = (T2[12] + T2[28]);
t135 = (t133 + t134);
t136 = (t133 - t134);
t137 = (T2[5] + T2[21]);
t138 = (T2[13] + T2[29]);
t139 = (t137 - t138);
t140 = (t137 + t138);
a17 = 0.70710678118654746*(T2[4] - T2[20]);
a18 = 0.70710678118654746*(T2[5] - T2[21]);
s66 = (a17 - a18);
a19 = 0.70710678118654746*(T2[13] - T2[29]);
a20 = 0.70710678118654746*(T2[12] - T2[28]);
s67 = (a20 + a19);
t141 = (s66 + s67);
t142 = (s66 - s67);
s68 = (a17 + a18);
s69 = (a20 - a19);
T1[0] = (t119 + t135);
T1[1] = (t124 + t140);
T1[16] = (t119 - t135);
T1[17] = (t124 - t140);
T1[8] = (t120 - t139);
T1[9] = (t123 + t136);
T1[24] = (t120 + t139);
T1[25] = (t123 - t136);
t143 = (s68 + s69);
t144 = (s68 - s69);
T1[4] = (t127 + t142);
T1[20] = (t127 - t142);
T1[5] = (t131 + t143);
T1[21] = (t131 - t143);
T1[13] = (t132 + t141);
T1[29] = (t132 - t141);
T1[12] = (t128 - t144);
T1[28] = (t128 + t144);
t145 = (T2[2] + T2[18]);
t146 = (T2[10] + T2[26]);
t147 = (t145 + t146);
t148 = (t145 - t146);
t149 = (T2[3] + T2[19]);
t150 = (T2[11] + T2[27]);
t151 = (t149 + t150);
t152 = (t149 - t150);
t153 = (T2[2] - T2[18]);
t154 = (T2[11] - T2[27]);
t155 = (t153 + t154);
t156 = (t153 - t154);
t157 = (T2[3] - T2[19]);
t158 = (T2[10] - T2[26]);
t159 = (t157 - t158);
t160 = (t157 + t158);
t161 = (T2[6] + T2[22]);
t162 = (T2[14] + T2[30]);
t163 = (t161 + t162);
t164 = (t161 - t162);
t165 = (T2[7] + T2[23]);
t166 = (T2[15] + T2[31]);
```

```
      t167 = (t165 - t166);
      t168 = (t165 + t166);
      a21 = 0.70710678118654746*(T2[6] - T2[22]);
      a22 = 0.70710678118654746*(T2[7] - T2[23]);
      s70 = (a21 - a22);
      a23 = 0.70710678118654746*(T2[15] - T2[31]);
      a24 = 0.70710678118654746*(T2[14] - T2[30]);
      s71 = (a24 + a23);
      t169 = (s70 - s71);
      t170 = (s70 + s71);
      s72 = (a21 + a22);
      s73 = (a24 - a23);
      T1[2] = (t147 + t163);
      T1[3] = (t151 + t168);
      T1[18] = (t147 - t163);
      T1[19] = (t151 - t168);
      T1[10] = (t148 - t167);
      T1[11] = (t152 + t164);
      T1[26] = (t148 + t167);
      T1[27] = (t152 - t164);
      t171 = (s72 + s73);
      t172 = (s72 - s73);
      T1[6] = (t156 + t169);
      T1[22] = (t156 - t169);
      T1[7] = (t160 + t171);
      T1[23] = (t160 - t171);
      T1[15] = (t159 + t170);
      T1[31] = (t159 - t170);
      T1[14] = (t155 - t172);
      T1[30] = (t155 + t172);
    }

/*============ GLOBAL COMM below ============*/
    {
      int mpii;
      MPI_Status stat;
      for(mpii=1;mpii<2;mpii++){
        MPI_Sendrecv_replace(T1+16*(mpii^mpirank),16,MPI_DOUBLE,mpii^mpirank,0 /*sendtag*/,
                             mpii^mpirank,0 /*recvtag*/,MPI_COMM_WORLD,&stat);
      }
    }
/*============ GLOBAL COMM above ============*/
    { /* dmploop */
      int i43 = mpirank;
      double t266, t267, t268, t252, t251, t254, t241, t243, t242, t253, t249, t265, t246,
             t240, t255, t239, t256, t238, t257, t237, t258, t259, t260, t261, t262, t263,
             t264, t245, t250, t244, t247, t248;
      t237 = (T1[0] + T1[16]);
      t238 = (T1[2] + T1[18]);
      t239 = (T1[1] + T1[17]);
      t240 = (T1[3] + T1[19]);
      t241 = (T1[0] - T1[16]);
      t242 = (T1[3] - T1[19]);
      t243 = (T1[1] - T1[17]);
      t244 = (T1[2] - T1[18]);
      Y[0] = (t237 + t238);
      Y[1] = (t239 + t240);
      Y[4] = (t237 - t238);
      Y[5] = (t239 - t240);
      Y[2] = (t241 - t242);
      Y[3] = (t243 + t244);
      Y[6] = (t241 + t242);
      Y[7] = (t243 - t244);
      t245 = (T1[4] + T1[20]);
      t246 = (T1[6] + T1[22]);
```

```
      t247 = (T1[5] + T1[21]);
      t248 = (T1[7] + T1[23]);
      t249 = (T1[4] - T1[20]);
      t250 = (T1[7] - T1[23]);
      t251 = (T1[5] - T1[21]);
      t252 = (T1[6] - T1[22]);
      Y[8] = (t245 + t246);
      Y[9] = (t247 + t248);
      Y[12] = (t245 - t246);
      Y[13] = (t247 - t248);
      Y[10] = (t249 - t250);
      Y[11] = (t251 + t252);
      Y[14] = (t249 + t250);
      Y[15] = (t251 - t252);
      t253 = (T1[8] + T1[24]);
      t254 = (T1[10] + T1[26]);
      t255 = (T1[9] + T1[25]);
      t256 = (T1[11] + T1[27]);
      t257 = (T1[8] - T1[24]);
      t258 = (T1[11] - T1[27]);
      t259 = (T1[9] - T1[25]);
      t260 = (T1[10] - T1[26]);
      Y[16] = (t253 + t254);
      Y[17] = (t255 + t256);
      Y[20] = (t253 - t254);
      Y[21] = (t255 - t256);
      Y[18] = (t257 - t258);
      Y[19] = (t259 + t260);
      Y[22] = (t257 + t258);
      Y[23] = (t259 - t260);
      t261 = (T1[12] + T1[28]);
      t262 = (T1[14] + T1[30]);
      t263 = (T1[13] + T1[29]);
      t264 = (T1[15] + T1[31]);
      t265 = (T1[12] - T1[28]);
      t266 = (T1[15] - T1[31]);
      t267 = (T1[13] - T1[29]);
      t268 = (T1[14] - T1[30]);
      Y[24] = (t261 + t262);
      Y[25] = (t263 + t264);
      Y[28] = (t261 - t262);
      Y[29] = (t263 - t264);
      Y[26] = (t265 - t266);
      Y[27] = (t267 + t268);
      Y[30] = (t265 + t266);
      Y[31] = (t267 - t268);
    }
}

void init_sub() {
    MPI_Comm_size(MPI_COMM_WORLD, &mpisize);
    MPI_Comm_rank(MPI_COMM_WORLD, &mpirank);
}
```

Code B.3 (`mpi_compute_matrix.c`) SPIRAL's `compute_matrix.c` can determine the matrix a generated program corresponds to. Therefore it executes the respective function on the unit vectors of appropriate length. This allows fully automatic verification of generated code. However, some modifications were necessary to make this feature compliant with distributed data vectors.

```
/*****************************************************************************
 * SPL Matrix                                                                *
```

```
 *                                                                          *
 * Computes matrix that corresponds to SPL generated routine                *
 ****************************************************************************/

#include <limits.h>
#include <time.h>
#include <stdio.h>
#include <assert.h>
#include <sys_conf/sys_conf.h>
#include <sys_conf/systemops.h>
#include <sys_conf/conf.h>
#include <sys_conf/vector.h>
#include <sys_conf/opt_macros.h>
#include <sys_conf/xmalloc.h>
#include <sys_conf/io.h> /* sys_fatal() */
#include <sys_conf/spl_prog.h> /* spl_stub_t */
#include <sys_conf/vector_def.h> /* data_type */
#include <mpi.h>

int cm_mpirank;
int cm_mpisize;
int cm_mpimaster;

extern spl_stub_t * get_stub0();
void (*Analyzed_function) (void *y, void *x);

vector_t * Input;
vector_t ** Outputs;

struct compute_matrix_config_t {
    int gap_output;
    int transpose;
    int inplace;
    config_profile_t * profile;
} MatrixConfig;

void init_options() {
    MatrixConfig.gap_output = 0;
    MatrixConfig.transpose = 0;
    MatrixConfig.inplace = 0;
}

void parse_options(int argc, char **argv) {
    init_options();
    SHIFT();
    while( argc > 0 && HAS_MINUS(argv[0]) ) {
        if      (OPT("-v"))  sys_set_verbose(1);
        else if (OPT("-g"))  MatrixConfig.gap_output = 1;
        else if (OPT("-t"))  MatrixConfig.transpose = 1;
        else if (OPT("-h") || OPT("--help")) sys_fatal(EXIT_CMDLINE, usage());
        else if (OPT("-inplace")) MatrixConfig.inplace = 1;
        else if (OPT("-fp")) {
            char *fp;
            GET_ARG(fp);
            sys_setenv("spiral_fp", fp, 1);
        }
        else if (OPT("-zb")) {
            char *zb;
            GET_ARG(zb);
            sys_setenv("spiral_zb", zb, 1);
        }
        else sys_fatal(EXIT_CMDLINE, "illegal option '%s'\n", argv[0]);
        SHIFT();
    }
    if(argc > 0) /* garbage at the end remains */
```

```
        sys_fatal(EXIT_CMDLINE, usage());

    if (get_stub0()->dim_cols == DIM_UNKNOWN || get_stub0()->dim_rows == DIM_UNKNOWN)
        sys_fatal(EXIT_CMDLINE, "compute_matrix: dimension is not fully specified\n");
}

void initialize(int argc, char **argv) {
    sys_set_progname(argv[0]);
    config_init(argv[0], "SPIRAL_", DEFAULT_CONFIG_FILE);
    srand(time(0));
    Analyzed_function = get_stub0()->func;
    parse_options(argc,argv);
    MatrixConfig.profile = config_find_profile(get_stub0()->profile_name);
    assert(MatrixConfig.profile != 0);
    config_set_default_profile(MatrixConfig.profile);

    if(MatrixConfig.transpose) {
        Outputs = xmalloc(sizeof(vector_t*) * 1);
        Outputs[0] = vector_create_zero(get_stub0()->data_type, get_stub0()->dim_rows);
    }
    else {
        int i;
        Outputs = xmalloc(sizeof(vector_t*) * get_stub0()->dim_cols);
        for(i=0; i < get_stub0()->dim_cols; i++)
            Outputs[i] = vector_create_zero(get_stub0()->data_type, get_stub0()->dim_rows);
    }
    Input = vector_create_zero(get_stub0()->data_type, get_stub0()->dim_cols);
    get_stub0()->init_func();
}

void finalize() {
    if(MatrixConfig.transpose)
        vector_destroy(Outputs[0]);
    else {
        int i;
        for(i=0; i < get_stub0()->dim_cols; i++)
            vector_destroy(Outputs[i]);
    }
    vector_destroy(Input);
    xfree(Outputs);
}

void compute_matrix() {
    int i, j;
    int x,y;
    MPI_Status stat;

        if(MatrixConfig.transpose) {
            if(MatrixConfig.gap_output) printf("[ ");
            for(x = 0; x < get_stub0()->dim_cols; x++) {
                vector_basis(Input, x);
                Analyzed_function (Outputs[x]->data, Input->data);
                vector_copy(Outputs[x], Input);

                if(!cm_mpirank) {
                    if(MatrixConfig.gap_output) {
                        if(x!=0) printf(",\n  [ ");
                        else printf("[ ");
                    }
                    else
                        printf("col=%i | ", x);


                    for(y = 0; y < get_stub0()->dim_rows; y++) {
                        if(MatrixConfig.gap_output) {
```

```
                        if(y!=0) printf(", ");
                        get_stub0()->data_type->fprint_gap(stdout, NTH(Outputs[x], y));
                    }
                    else {
                        get_stub0()->data_type->fprint(stdout, NTH(Outputs[x], y));
                        printf("\t");
                    }
                }

                if(MatrixConfig.gap_output) printf(" ]");
                else printf("\n");
            }
        }
        if(MatrixConfig.gap_output) if(!cm_mpirank) printf("\n];\n");
    }
    else {
        for(x = 0; x < get_stub0()->dim_cols; x++) {
            vector_basis(Input, x);
            Analyzed_function (Outputs[x]->data,
               &(((double*)Input->data)[cm_mpirank * (get_stub0()->dim_cols / cm_mpisize)]));

            MPI_Gather(Outputs[x]->data,
                       (get_stub0()->dim_cols)/cm_mpisize,
                       MPI_DOUBLE,
                       Outputs[x]->data,
                       (get_stub0()->dim_cols)/cm_mpisize,
                       MPI_DOUBLE,
                       0,
                       MPI_COMM_WORLD);

//              if (MatrixConfig.inplace)
//                  vector_copy(Outputs[x], Input);
            }
            if(!cm_mpirank) {

                if(MatrixConfig.gap_output) printf("[ ");
                for(y = 0; y < get_stub0()->dim_rows; y++) {
                    if(MatrixConfig.gap_output) {
                        if(y!=0) printf(",\n  [ ");
                        else printf("[ ");
                    }
                    else
                        printf("row=%i | ", y);

                    for(x = 0; x < get_stub0()->dim_cols; x++) {

                        if(MatrixConfig.gap_output) {
                            if(x!=0) printf(", ");
                            get_stub0()->data_type->fprint_gap(stdout, NTH(Outputs[x], y));
                        }
                        else {
                            get_stub0()->data_type->fprint(stdout, NTH(Outputs[x], y));
                            printf("\t");
                        }

                    }
                    if(MatrixConfig.gap_output) printf(" ]");
                    else printf("\n");
                }
                if(MatrixConfig.gap_output) printf("\n];\n");
            }
        }

}
```

```
int main(int argc, char** argv) {
    MPI_Init( &argc , &argv );
    MPI_Comm_size( MPI_COMM_WORLD , &cm_mpisize );
    MPI_Comm_rank( MPI_COMM_WORLD , &cm_mpirank );
    initialize(argc,argv);
    compute_matrix();
    finalize();
    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

Code B.4 (`mpi_time.c`) SPIRAL's `time.c` measures run times of generated functions. `mpi_time.c` is the tweaked, parallel, version of this file to provide reliable run time measurement for MPI parallel code.

```
/*************************************************************************
 * SPL Timer                                                             *
 * time.c                                                                *
 *                                                                       *
 * This is a standard timing driver, taken from SPL compiler             *
 *************************************************************************/

#include <limits.h>
#include <time.h>
#include <stdio.h>
#include <assert.h>
#include <sys_conf/sys_conf.h>
#include <sys_conf/systemops.h>
#include <sys_conf/vector.h>
#include <sys_conf/conf.h>
#include <sys_conf/opt_macros.h>
#include <sys_conf/io.h> /* sys_fatal() */
#include <sys_conf/spl_prog.h> /* spl_stub_t */
#include <sys_conf/vector_def.h> /* data_type */

#ifdef WIN32
#include <xmmintrin.h>
#endif

#include <mpi.h>

extern spl_stub_t * get_stub0();
void (*Timed_function) (void *y, void *x);

vector_t * Input;
vector_t * Output;

struct timer_config_t {
    int gap_output;
    config_profile_t * profile;
} TimerConfig;

char* usage() {
    return "Usage: test [-v] [-g]\n";
}

void init_options() {
    TimerConfig.gap_output = 0;
}

void parse_options(int argc, char **argv) {
    init_options();
```

```
    SHIFT();
    while( argc > 0 && HAS_MINUS(argv[0]) ) {
        /* common for all timers */
        if      (OPT("-v"))  sys_set_verbose(1);
        else if (OPT("-g"))  TimerConfig.gap_output = 1;
        else if (OPT("-h") || OPT("--help")) sys_fatal(EXIT_CMDLINE, usage());
        else if (OPT("-fp")) {
            char *fp;
            GET_ARG(fp);
            sys_setenv("spiral_fp", fp, 1);
        }
        else if (OPT("-zb")) {
            char *zb;
            GET_ARG(zb);
            sys_setenv("spiral_zb", zb, 1);
        }
        else sys_fatal(EXIT_CMDLINE, "illegal option '%s'\n", argv[0]);
        SHIFT();
    }

    if(argc > 0) /* garbage at the end remains */
        sys_fatal(EXIT_CMDLINE, usage());

    if (get_stub0()->dim_cols==DIM_UNKNOWN || get_stub0()->dim_rows==DIM_UNKNOWN)
        sys_fatal(EXIT_CMDLINE, "timer: dimension is not fully specified\n");
}

void initialize(int argc, char **argv) {

    int mpisize;
    MPI_Comm_size(MPI_COMM_WORLD, &mpisize);
    sys_set_progname(argv[0]);
    config_init(argv[0], "SPIRAL_", DEFAULT_CONFIG_FILE);
    srand(time(0));
    TimerConfig.profile = config_find_profile(get_stub0()->profile_name);
    assert(TimerConfig.profile != 0);
    config_set_default_profile(TimerConfig.profile);
    Timed_function = get_stub0()->func;
    parse_options(argc,argv);
    Output = vector_create_random(get_stub0()->data_type, (get_stub0()->dim_rows)/mpisize);
    Input = vector_create_random(get_stub0()->data_type, (get_stub0()->dim_cols)/mpisize);
    get_stub0()->init_func();
}

void finalize() {
    vector_destroy(Output);
    vector_destroy(Input);
}

double dtime();

void perform_timing() {
        int i, nloop;
        double t;
        int mpirank;
        MPI_Comm_rank(MPI_COMM_WORLD, &mpirank);

        Timed_function(Output->data, Input->data);
        nloop=1;
        t=0.0;
        MPI_Barrier(MPI_COMM_WORLD);
        while (t < 1.0) {
            nloop = nloop*2;
            dtime();
            for (i=0; i<nloop; i++)
```

```
                Timed_function (Output->data, Input->data);
            MPI_Barrier(MPI_COMM_WORLD);
            t = dtime();
        }

        if (!mpirank)
          if (TimerConfig.gap_output == 0)
            printf("%10.4e\n", t/nloop);
          else {
            double result = t/nloop;
            double_record->fprint_gap(stdout, &result);
            printf(";\n");
          }
}

#if HAVE_GETTIMEOFDAY

#include <sys/time.h>
double dtime() {
     static int first=1;
     static struct timeval t0, t1;
     struct timeval diff;
     if (first) {
          gettimeofday(&t0, 0);
          first = 0;
          return 0;
     }
     gettimeofday(&t1, 0);
     diff.tv_sec = t1.tv_sec - t0.tv_sec;
     diff.tv_usec = t1.tv_usec - t0.tv_usec;
     while (diff.tv_usec < 0) {
          diff.tv_usec += 1000000L;
          diff.tv_sec -= 1;
     }
     t0=t1;
     return (double)diff.tv_sec+(double)diff.tv_usec*1e-6;
}
#else

#include <time.h>
double dtime() {
     static int first=1;
     static clock_t t0, t1;
     clock_t diff;
     if (first) {
          t0 = clock();
          first = 0;
          return 0;
     }
     t1 = clock();
     diff = t1-t0;
     t0 = t1;
     return (double)diff/(double)CLOCKS_PER_SEC;
}

#endif

int main(int argc, char** argv) {
  MPI_Init(&argc,&argv);
  initialize(argc,argv);
  perform_timing();
  finalize();
  MPI_Finalize();
  return EXIT_SUCCESS;
}
```

References

[1] A. Adelmann, A. Bonelli, W. P. Petersen, and C. W. Ueberhuber, *Communication Efficiency of Parallel 3D FFTs*, Proceedings of 6th International Conference on High Performance Computing in Computational Sciences VecPar 2004, vol. III, 2004, pp. 901–907.

[2] Gerald Baumgartner et al., *Synthesis of High-Performance Parallel Programs for a Class of Ab Initio Quantum Chemistry Models*, in [44] (2005), 276–292.

[3] C. Bell, D. Bonachea, R. Nishtala, and K.Yelick, *Optimizing Bandwidth Limited Problems Using One-Sided Communication and Overlap*, Parallel and Distributed Processing Symposium, 2006 (IPDPS'06), April 2006, pp. 10–19.

[4] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, ScaLapack *Users' Guide*, SIAM, Philadelphia, PA, 1997.

[5] Dan Bonachea, *GASNet Specification, v1.1*, Tech. Report UCB/CSD-02-1207, U. C. Berkeley, October 2002.

[6] Andreas Bonelli, *Communication Efficiency of Parallel 3D FFTs*, Master's thesis, Institute of Analysis and Scientific Computing, Vienna University of Technology, November 2004.

[7] Andreas Bonelli, Franz Franchetti, Juergen Lorenz, Markus Püschel, and Christoph W. Ueberhuber, *Automatic Performance Optimization of DSP Transforms on Distributed Memory Computers*, The 2006 International Symposium on Parallel and Distributed Processing and Applications (ISPA'06), 2006.

[8] Jaeyoung Choi, Jack Dongarra, and David W. Walker, *Parallel Matrix Transpose Algorithms on Distributed Memory Concurrent Computers*, Parallel Computing **21** (1995), no. 9, 1387–1405.

[9] W. T. Cochran, J. W. Cooley, D. L. Favin, H. D. Helms, R. A. Kaenel, W. W. Lang, G.-C. Maling, D. E. Nelson, C. E. Rader, and P. D. Welch, *What is the fast fourier transform?*, IEEE Trans. Audio Electroacoustics (1967).

[10] UPC Consortium, *UPC Language Specifications, v1.2*, Tech. Report LBNL-59208, Lawrence Berkeley National Lab, 2005, `http://upc.lbl.gov/`.

[11] J. W. Cooley and J. W. Tukey, *An algorithm for the machine calculation of complex fourier series*, Math. Comp. **19** (1965), 297–301.

[12] Pierre Teilhard de Chardin, *The future of man*, 1950.

[13] Jack J. Dongarra, P. Luszczek, and A. Petitet, *The LINPACK benchmark: Past, present, and future*, Concurrency and Computation: Practice and Experience **15** (2003), 803–820.

[14] Alan Edelman, *Optimal matrix transposition of bit reversal on hypercubes: All-to-all personalized communication*, J. Parallel Distrib. Comput. **11** (1991), no. 4, 328–331.

[15] Maria Eleftheriou, Blake Fitch, Aleksandr Rayshubskiy, T.J. Christopher Ward, and Robert Germain, *Performance Measurements of the 3D FFT on the BlueGene/L Supercomputer*, Proceedings of the International Euro-Par Conference (2005), 795–803.

[16] ———, *Scalable Framework for 3D FFTs on the Blue Gene/L Supercomputer: Implementation and Early Performance Measurements*, IBM Journal of Research and Development **49** (2005), no. 2/3, 457–464.

[17] Graham E. Fagg, Thara Angskun, George Bosilca, Jelena Pjesivac-Grbovic, and Jack J. Dongarra, *Scalable fault tolerant MPI: Extending the recovery algorithm*, Recent Advances in Parallel Virtual Machine and Messaging Passing Interface Users' Group Meeting Euro PVMMPI 2005, Lecture Notes in Computer Science, vol. 3666, Springer Heidelberg, 2005, pp. 67–75.

[18] Ahmad Faraj and Xin Yuan, *An Empirical Approach for Efficient All-to-All Personalized Communication on Ethernet Switched Clusters*, Proceedings of the International Conference on Parallel Processing (ICPP), 2005, pp. 321–328.

[19] ———, *Automatic Generation and Tuning of MPI Collective Communication Routines*, Proceedings of the 19th Annual International Conference on Supercomputing (ICS), 2005, pp. 393–402.

[20] F. Franchetti, S. Kral, J. Lorenz, and C.W. Ueberhuber, *Efficient Utilization of SIMD Extensions*, in [44] (2005), 409–425.

[21] F. Franchetti and M. Püschel, *Short vector code generation for the discrete fourier transform*, Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS'03) (Los Alamitos, USA), Comp. Society Press, 2003.

[22] F. Franchetti, Y. Voronenko, and M. Püschel, *Formal Loop Merging for Signal Transforms*, Proc. Programming Language Design and Implementation (PLDI), 2005, pp. 315–326.

[23] Franz Franchetti, Andreas Bonelli, Ekapol Chuangsuwanich, Yu-Chiang J. Lee, Jürgen Lorenz, Thomas Peter, Hau Shen, Marek Telegarsky, Yevgen Voronenko, Markus Püschel, J. M. F. Moura, and Christoph W. Ueberuber, *Parallelism in Spiral*, Workshop on Programming Models for Ubiquitous Parallelism, Seattle, WA, USA, 2006.

[24] Franz Franchetti, Juergen Lorenz, and Christoph W. Ueberhuber, *Latency hiding parallel FFTs*, Tech. report, Institute for Applied and Numerical Analysis, Vienna University of Technology, 2002.

[25] Donald Fraser, *Array permutation by index-digit permutation*, J. ACM **23** (1976), no. 2, 298–309.

[26] M. Frigo, *A fast fourier transform compiler*, Proceedings of the PLDI 1999, vol. 3, 1999, p. 1381.

[27] Matteo Frigo, FFTW*, Release 2.1.5—Documentation and User Manual*, 2003.

[28] Matteo Frigo and Steven G. Johnson, FFTW*: An Adaptive Software Architecture for the FFT*, Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing, vol. 3, IEEE, 1998, pp. 1381–1384.

[29] ———, *The Design and Implementation of* FFTW*3*, in [44] (2005), 216–231.

[30] Aca Gačić, *Automatic Implementation and Platform Adaptation of Discrete Filtering and Wavelet Algorithms*, Ph.D. thesis, Carnegie Mellon University, December 2004.

[31] W. M. Gentleman and G. Sande, *Fast fourier transforms – for fun and profit*, Proceedings of the Fall Joint Computer Conference (Reston, VA.), AFIPS, 1966, pp. 563–578.

[32] G. Goumas, N. Drosinos, M. Athanasaki, and N. Koziris, *Automatic Parallel Code Generation for Tiled Nested Loops*, Proceedings of the ACM Symposium on Applied Computing (SAC), ACM Press, 2004, pp. 1412–1419.

[33] L. Greengard and V. Rokhlin, *A fast algorithm for particle simulations*, Journal of Computational Physics **73** (1987), 325–348.

[34] S. K. S. Gupta, Z. Li, and J. H. Reif, *Synthesizing efficient out-of-core programs for block recursive algorithms using block-cyclic data distributions*, Technical Report TR-96-04, Dept. of Computer Science, Duke University, Durham, USA, 1996.

[35] F. Gygi, E. Draeger, B. R. de Supinski, R. K. Yates, F. Franchetti, S. Kral, J. Lorenz, C. W. Ueberhuber, J. Gunnels, and J. Sexton, *Large-Scale First-Principles Molecular Dynamics Simulations on the BlueGene/L Platform using the Qbox Code*, Proceedings of Supercomputing 2005. Gordon Bell Prize finalist, 2005.

[36] P. Hilfinger, D. Bonachea, K. Datta, D. Gay, S. Graham, B. Liblit, G. Pike, J. Su, and K. Yelick, *Titanium Language Reference Manual*, Tech. Report UCB/EECS-2005-15, U.C. Berkeley, 2005, `http://titanium.cs.berkeley.edu/`.

[37] Helmut Hlavacs and Christoph W. Ueberhuber, *Frontiers in simulation*, 2002.

[38] R. A. Horn and C. R. Johnson, *Topics in matrix analysis*, Cambridge University Press, Cambridge, 1991.

[39] J. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri, *A Methodology for Designing, Modifying, and Implementing Fourier Transform Algorithms on Various Architectures*, Circuits Systems Signal Process **9** (1990), 449–500.

[40] Jeremy Johnson and Kang Chen, *A Self-Adapting Distributed Memory Package for Fast Signal Transforms*, Proc. International Parallel and Distributed Processing Symposium (IPDPS) (2004), 44a.

[41] S. D. Kaushik, S. Sharma, C.-H. Huang, J. R. Johnson, R. W. Johnson, and P. Sadayappan, *An Algebraic Theory for Modeling Multistage Interconnection Networks*, International Conference on Parallel and Distributed Systems (ICPADS), National Tsing Hua University, Hsinchu, Taiwan, Republic of China, 1992.

[42] Gordon E. Moore, *Cramming more components onto integrated circuits*, Electronics Magazine (19 April 1965).

[43] J. M. F. Moura, J. Johnson, R. W. Johnson, D. Padua, V. Prasanna, M. Püschel, and M. M. Veloso, SPIRAL*: Portable library of optimized signal processing algorithms*, 1998, `http://www.ece.cmu.edu/ spiral`.

[44] José M. F. Moura, Markus Püschel, David Padua, and Jack Dongarra, *Special Issue on Program Generation, Optimization, and Platform Adaptation*, Proceedings of the IEEE **93(2)** (2005).

[45] A. Norton and A. J. Silberger, *Parallelization and performance analysis of the cooley-tukey fft algorithm for shared-memory architectures*, IEEE Trans. Comput. **36** (1987), 581–591.

[46] Robert W. Numrich and John Reid, *Co-Array Fortran for Parallel Programming*, SIGPLAN Fortran Forum **17** (1998), no. 2, 1–31, `http://www.co-array.org/`.

[47] P. Pacheco, *Parallel programming with MPI*, Morgan Kaufmann Publishers, San Francisco, 1997.

[48] M. C. Pease, *An adaptation of the fast fourier transform for parallel processing*, Journal of the ACM **15** (1968), 252–264.

[49] Nikos P. Pitsianis, *The Kronecker Product in Optimization and Fast Transform Generation*, Ph.D. thesis, Department of Computer Science, Cornell University, 1997.

[50] J Pjesivac-Grbovic, T Angskun, G Bosilca, G E Fagg, E Gabriel, and J. Dongarra, *Performance Analysis of MPI Collective Operations*, 19th International Parallel and Distributed Processing, IEEE Computer Society Press, 2005.

[51] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, SPIRAL*: Code Generation for DSP Transforms*, in [44] **93** (2005), no. 2, 232–275.

[52] M. Püschel, B. Singer, J. Xiong, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson, SPIRAL*: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms*, Int'l Journal of High Performance Computing Applications **18** (2004), no. 1, 21–45.

[53] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gačić, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo, SPIRAL*: Code Generation for DSP Transforms*, Proceedings of the IEEE, special issue on Program Generation, Optimization, and Adaptation **93** (2005), no. 2, 232–275.

[54] Jürgen Schmidhuber, *The new AI: General & sound & relevant for physics*, Tech. Report IDSIA-04-03, Version 2.0, November 2003.

[55] B. Singer and M. Veloso, *Stochastic Search for Signal Processing Algorithm Optimization*, Proceedings of the Supercomputing 2001, 2001.

[56] Herb Sutter, *The free lunch is over: A fundamental turn toward concurrency in software*, Dr. Dobb's Journal (2005), http://www.gotw.ca/publications/concurrency-ddj.htm.

[57] C. Temperton, *Fast mixed-radix real fourier transforms*, J. Comput. Phys. **52** (1983), 340–350.

[58] Frank J. Tipler, *Cosmological limits on computation*, International Journal of Theoretical Physics **25** (1986), 617–661.

[59] R. Tolimieri, M. An, and C. Lu, *Algorithms for Discrete Fourier Transforms and Convolution*, 2nd ed., Springer, 1997.

[60] ———, *Mathematics of Multidimensional Fourier Transform Algorithms*, 2nd ed., Springer, 1997.

[61] Sathish S. Vadhiyar, Graham E. Fagg, and Jack Dongarra, *Automatically Tuned Collective Communications*, Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM) (Washington, DC, USA), IEEE Computer Society, 2000, pp. 46–46.

[62] ———, *Performance Modeling for Self Adapting Collective Communications for MPI*, LACSI Symposium 2001, Santa Fe, NM, 2001.

[63] C. Van Loan, *Computational frameworks for the Fast Fourier Transform*, Frontiers in Applied Mathematics, vol. 10, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, 1992.

[64] Yevgen Voronenko and Markus Püschel, *Automatic generation of implementations for DSP transforms on fused multiply-add architectures*, International Conference on Acoustics, Speech, and Signal Processing (ICASSP), 2004.

[65] R. C. Whaley, A. Petitet, and J. J. Dongarra, *Automated empirical optimizations of software and the* ATLAS *project*, Parallel Comput. **27** (2001), 3–35.

[66] J. Xiong, J. Johnson, R. Johnson, and D. Padua, *SPL: A Language and Compiler for DSP Algorithms*, Proceedings of the PLDI 2001, 2001, pp. 298–308.

CURRICULUM VITAE

Name: Andreas Bonelli

Title: Dipl.-Ing.

Date and Place of Birth: January 27th 1978, Vienna, Austria

Nationality: Austria

Home Address: Margaretenstrasse 5/27, A-1040 Vienna, Austria

Affiliation

Institute for Analysis and Scientific Computing
Vienna University of Technology
Wiedner Hauptstrasse 8-10/101, A-1040 Vienna
Phone: +43 1 58801 10164
Fax: +43 1 58801 10196
E-mail: andreas.bonelli@gmail.com

Education

1998	High School Diploma (*Matura*)
1998 – 1999	Military Service
1999 – 2004	Studies in Technical Mathematics at the Vienna University of Technology
2004	Dipl.-Ing. (Technical Mathematics) at the Vienna University of Technology
2004 – 2006	Ph.D. studies

Employment

1997 – 2003	Several Internships at Siemens PSE Vienna
2004 –	Research Assistant at the Institute for Analysis and Scientific Computing (TU Wien), funded by the SFB Aurora

Project Experience

2004 –	Participation in the SFB Aurora