Automatic Performance Optimization of the Discrete Fourier Transform on Distributed Memory Computers

Andreas Bonelli[1]*, Franz Franchetti[2], Juergen Lorenz[1], Markus Püschel[2], and Christoph W. Ueberhuber[1]

[1] Institute for Analysis and Scientific Computing
Vienna University of Technology
Wiedner Hauptstrasse 8-10, A-1040 Wien, Austria
{a.bonelli,juergen.lorenz,c.ueberhuber}@tuwien.ac.at
[2] Department of Electrical and Computer Engineering
Carnegie Mellon University
5000 Forbes Ave, Pittsburgh, PA, 15213, USA
{franzf,pueschel}@ece.cmu.edu

Abstract. This paper introduces a formal framework for automatically generating performance optimized implementations of the discrete Fourier transform (DFT) for distributed memory computers. The framework is implemented as part 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, we extend SPIRAL to automatically generate parallelized formulas. Using the same mechanism, we enable the generation of rescaling DFT algorithms, which redistribute the data in intermediate steps to fewer processors to reduce communication overhead. It is a novel feature of these methods that the redistribution steps are merged with the communication steps of the algorithm to avoid additional communication overhead. Among the possible alternative algorithms, SPIRAL's search mechanism now determines the fastest for a given platform, effectively generating adapted code without human intervention. Experiments with DFT MPI programs generated by SPIRAL show performance gains of up to 30% due to rescaling. Further, our generated programs compare favorably with FFTW-MPI 2.1.5.

1 Introduction

For many important numerical problems, current compilers are not able to produce code that is competitive with hand-tuned code in efficiency. To overcome this shortcoming, a number of research efforts have developed novel methods aiming at automatic program generation, optimization, and platform adaptation [17]. Examples include ATLAS for basic linear algebra subroutines (BLAS), FFTW for the discrete Fourier transform (DFT), and SPIRAL for more general linear transforms. These and other approaches address the problem of automatically tuning to single processor platforms. Specifically, one goal is to tune code to a given memory

* This work was supported by the Special Research Program SFB F011 "AURORA" and the Erwin Schrödinger Fellowship of the Austrian Science Fund FWF, and in part by DARPA through the Department of Interior grant NBCH1050009 and by NSF through awards 0234293 and 0325687.

hierarchy. However, with few exceptions, parallelization is still done by hand. The improvement of this situation for the DFT on distributed memory computers is the subject of this paper.

Contributions of this Paper. SPIRAL is a program generation and optimization system for linear transforms including the DFT and many others [19]. SPIRAL supports a wide range of platforms including vector architectures [7, 10] and shared memory platforms [9]. In this paper we extend SPIRAL to generate MPI programs for the DFT. To do this, we identify rewriting rules that enable the automatic parallelization of FFTs given as mathematical formulas. This replaces expensive compiler analysis by simple pattern matching. In addition, we provide rules that rescale the computation to a different number of CPUs during the computation. By integrating these rules in SPIRAL's rewriting system, SPIRAL's automatic search mechanism can find the fastest among alternatives and generate DFT MPI code that is adapted to a given computing platform. We show that the generated programs benefit from rescaling for many sizes and that they compare favorably to FFTW-MPI 2.1.5. Besides performance improvement, the generation of rescaling DFT programs provides greater flexibility to the user in that it decouples initial data distribution and processor use. This flexibility is usually not provided in libraries.

Related Work. The work described in the following addresses the common problem of obtaining fast code for distributed memory platforms by automatically tuning to the platform's characteristics. The approaches range from classical compiler techniques to high level formula manipulation and program generation. The respective application domains range from general linear algebra and linear transforms to more application specific problems like quantum chemistry computations.

A compiler framework for generating MPI code for arbitrarily tiled for-loop nests by performing various loop transformations to gain *inherent* coarse-grained parallelism is presented in [14]. [18] describes the generation of collective communication MPI code by automatically searching for the best algorithm on a given system. Another empirical approach for generating efficient all-to-all communication routines for Ethernet switched clusters is used by [6].

SCALAPACK [3] 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 various architectures using different processor numbers. SCALAPACK requires the user to define the processor configuration and to distribute the matrix data herself.

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

FFTW [11, 12] is a self-adapting DFT library supporting one- and higher-dimensional real and complex input data of arbitrary size. Typically, FFTW is faster than most other publicly available FFT libraries and also compares well to vendor libraries. MPI support, i. e., MPI-FFTW, is available in FFTW 2.1.5 but not in the more recent version 3.1 [13]. FFTW requires the data to be provided in slab decomposition. It then estimates the optimal number of processors to use for a given computation. If this number is different from the number of CPUs the user's program runs on, FFTW requires the user to redistribute prior and after calling FFTW. If other data layouts are required, users often resort to their own custom implementations to increase performance [5, 15]. Experiments [1] show that substantial portions of the runtime are spent on communication between processors. A program generation framework as presented in this paper is a step towards improving this situation in that it enables customization without programming effort.

[16] describes the extension of a sequential self-adapting package for the Walsh-Hadamard transform (WHT) to support 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. In spirit, the approach taken in [16] is similar to the framework developed in this paper.

Synopsis. Section 2 introduces the DFT and the mathematical foundation for representing its fast algorithms. Then we explain the SPIRAL system, which is the platform for our work. In Section 3, we develop the formal framework to generate MPI DFT implementation; an application of this approach to a novel method of rescaling DFT algorithms is illustrated in Section 4. We implemented the framework as extension of SPIRAL and show benchmarks of automatically generated and optimized DFT code in Section 5. The results show that rescaling provides performance gains and that our generated MPI programs compare favorably with FFTW.

2 Background: Discrete Fourier Transform and SPIRAL

Discrete Fourier transform. The discrete Fourier transform (DFT) is the matrix vector multiplication $x \mapsto y = \mathrm{DFT}_n\, x$, where $x, y \in \mathbb{C}^n$ are the input and output, respectively, and DFT_n is the $n \times n$ matrix defined by

$$\mathrm{DFT}_n = [\omega_n^{k\ell} \mid k, \ell = 0, \dots, n-1], \quad \omega_n = e^{2\pi\sqrt{-1}/n}.$$

The famous Cooley-Tukey fast Fourier transform (FFT) can be expressed as a factorization of DFT_n into a product of structured sparse matrices [21], namely, for $n = km$,

$$\mathrm{DFT}_{km} \to (\mathrm{DFT}_k \otimes I_m) T_m^n (I_k \otimes \mathrm{DFT}_m) L_k^n \tag{1}$$

We call (1) a *breakdown rule* since it formally represents a divide and conquer algorithm. This is emphasized by writing $\to$ instead of $=$.

In (1) we used the following notation. The $n \times n$ identity matrix is denoted with I_n; L_k^n is the stride permutation matrix defined by its underlying permutation

$$L_k^n : jm + i \mapsto ik + j, \quad 0 \le i < m,\ 0 \le j < k.$$

It is equivalent to transposing an $m \times k$ matrix stored in row-major order in memory.

Most importantly, the *tensor* or *Kronecker product* of matrices is defined by

$$A \otimes B = [a_{k,\ell} B], \quad \text{for } A = [a_{k,\ell}].$$

Finally, T_m^n is a diagonal matrix, called twiddle matrix, whose exact form can be found in [21].

Recursive computation of the DFT using (1) and other FFTs (in case that n does not decompose) enables the computation of the DFT in $O(n \log(n))$ operations. Note that there is a

SPL construct	code

$y = (A_n B_n)x$	`t[0:1:n-1] = B(x[0:1:n-1]);`
	`y[0:1:n-1] = A(t[0:1:n-1]);`

$y = (I_m \otimes A_n)x$	`for (i=0;i<m;i++)`
	`   y[i*n:1:i*n+n-1] = A(x[i*n:1:i*n+n-1]);`

$y = (A_m \otimes I_n)x$	`for (i=0;i<m;i++)`
	`   y[i:n:i+m-1] = A(x[i:n:i+m-1]);`

$y = L_k^{km} x$	`for (i=0;i<k;i++)`
	`   for (j=0;j<m;j++)`
	`      y[i+k*j]=x[m*i+j];`

Table 1. Compiling SPL into code is done by recursively using the above correspondences. x denotes the input and y the output vector. We use Matlab-like notation: `x[b:s:e]` denotes the subvector of x starting at `b`, ending at `e`, and extracted at stride `s`.

large degree of freedom in recursing, since at each step several factorizations of n may be possible. These recursions have roughly the same operations count but different memory access patterns, which leads to different runtimes when implemented.

SPIRAL. SPIRAL [19, 20] is a program generation and optimization system for linear transforms such as the DFT and many others. Its internal structure is shown in Figure 1.

The user formally specifies a transform she wants to have implemented, e.g., "DFT_{256}". First, SPIRAL recursively applies breakdown rules such as (1) to generate one out of many possible *formulas*, represented in the language SPL (signal processing language), which was informally introduced above. Namely, SPL expresses algorithms as sparse structured matrix factorizations using products, tensor products, and basic matrix such as the identity and permutations. Next, SPIRAL optimizes the structure of the formula using a formula rewriting system (see [4] for an introduction to rewriting systems). The rewriting effectively performs optimizations for the memory hierarchy [8], for vector instructions [10], or for shared memory platforms [9]. The idea is to perform these optimizations at a high level of abstraction (namely on formulas), since they are unpractical at the C code level.

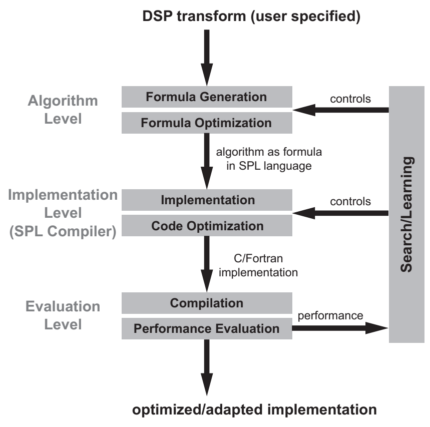

Figure 1. SPIRAL's architecture.

The obtained optimized SPL formula is then translated into C code using a special purpose compiler. This is possible since formulas have a clear interpretation as code. A few simple examples are shown in Table 1. The obtained code is further optimized and then compiled and its runtime measured.

The runtime is fed into a search engine, which drives, in a feedback loop, the formula generation process and the selection of implementation options such as the degree of unrolling.

In doing so, SPIRAL effectively searches for the formula, or algorithm, that runs fastest on the given computing platform. Search strategies include dynamic programming and evolutionary search. Upon termination, the final program is output to the user.

The goal of this paper is to present first steps in extending SPIRAL to generate efficient programs for distributed memory platforms. Similar to the vector code generation and shared memory parallel code generation, we achieve this through a suitably designed extension of SPIRAL's rewriting system and the SPL compiler. This is explained in the next sections.

3 Translating Formulas into MPI Programs

In Section 2 we explained SPIRAL and its theoretical underpinning: the formula language SPL, which enables algorithm generation and optimization at a high level of abstraction. Our goal is to enable SPIRAL to generate efficient MPI implementations. To this end, we now introduce formula constructs that are translated into message passing programs by an extension of the SPL compiler, called MPI-SPL compiler. The MPI-SPL compiler is one major contribution of this paper.

Data distribution. We introduce the tag "par(p)" to express that a formula will be implemented on p processors. We assume that all distributed data vectors are block distributed, i.e., each processors' memory holds one equal sized contiguous chunk of the data vector. For instance, if a formula A_6, representing the computation $y = A_6x$, operates on vectors of 6 data elements which are distributed across 2 processors, we write

$$\left(\begin{array}{c} y_0 \\ y_1 \\ y_2 \\ \hline y_3 \\ y_4 \\ y_5 \end{array}\right) = \underbrace{A_6}_{\text{par}(2)} \left(\begin{array}{c} x_0 \\ x_1 \\ x_2 \\ \hline x_3 \\ x_4 \\ x_5 \end{array}\right).$$

The tag "par(2)" implies that the computation of $y = A_6x$ is distributed across 2 processors. The elements x_0, x_1, x_2 and y_0, y_1, y_2 are stored in the memory of processor 0, while the elements x_3, x_4, x_5 and y_3, y_4, y_5 are stored in the memory of processor 1. We add a horizontal line between vector elements that reside in the local memory of different processors.

In addition, we introduce tags that express data redistribution. The tag "par$(q \leftarrow p)$" expresses that the input vector x is distributed over p processors and the output vector y is distributed over q processors. This implies that the tagged formula does a redistribution from p to q processors during its computation. For instance, we denote a formula A_6 operating on vectors of 6 data elements with the input x distributed across 2 processors and the output y distributed across 3 processors by

$$\left(\begin{array}{c} y_0 \\ y_1 \\ \hline y_2 \\ y_3 \\ \hline y_4 \\ y_5 \end{array}\right) = \underbrace{A_6}_{\text{par}(3 \leftarrow 2)} \left(\begin{array}{c} x_0 \\ x_1 \\ x_2 \\ \hline x_3 \\ x_4 \\ x_5 \end{array}\right). \tag{2}$$

The tag "par$(3 \leftarrow 2)$" implies that the computation of $y = A_6x$ is started on 2 processors and finished on 3 processors, redistributing during computation. The elements x_0, x_1, x_2, and y_0, y_1 are stored in the memory of processor 0, the elements x_3, x_4, x_5, and y_2, y_3 are stored in the memory of processor 1, and the elements y_4, y_5 are stored in the memory of processor 2.

Finally, we introduce the tag "par($p \leftarrow q \leftarrow p$)," which expresses that a formula's input and output are distributed across p processors but the formula internally redistributes to q processors. For instance,

$$y = \underbrace{AB}_{\text{par}(p \leftarrow q \leftarrow p)} x \quad \text{with} \quad \underbrace{AB}_{\text{par}(p \leftarrow q \leftarrow p)} = \underbrace{A}_{\text{par}(p \leftarrow q)} \underbrace{B}_{\text{par}(q \leftarrow p)}$$

has the input x and the output y distributed over p processors, but the output of B (i.e., the input of A) is distributed across q processors.

Parallel computation. The formula construct

$$I_p \otimes A^{m\times n} = \begin{bmatrix} A^{m\times n} & & \\ & \ddots & \\ & & A^{m\times n} \end{bmatrix}, \quad A^{m\times n} \in \mathbb{C}^{m\times n}$$

is a block-diagonal matrix of p blocks of $A^{m\times n}$. The tagged formula

$$y = \underbrace{\left(I_p \otimes A^{m\times n}\right)}_{\text{par}(p)} x \tag{3}$$

expresses a p-way embarrassingly parallel computation. Each $A^{m\times n}$ operates on an independent part of x and y. The vectors $x \in \mathbb{C}^{pn}$ and $y \in \mathbb{C}^{pm}$ are distributed across p processors into p local vectors $x'_i \in \mathbb{C}^n$ and $y'_i \in \mathbb{C}^m$ with $x = x'_0 \oplus \cdots \oplus x'_{p-1}$ and $y = y'_0 \oplus \cdots \oplus y'_{p-1}$; $\oplus$ denotes the stacking of column vectors. All p processors execute the formula $A^{m\times n}$ in parallel computing $y'_i = A^{m\times n} x'_i$. Since it is the same formula in each case, (3) is easily implemented as single program multiple data (SPMD) MPI program.

Similarly, formulas consisting of diagonal matrices,

$$y = \underbrace{D}_{\text{par}(p)} x, \quad D \in \mathbb{C}^{mp\times mp} \text{ diagonal}, \tag{4}$$

can be trivially mapped to MPI programs.

All-to-all communication. Permutations express data reordering. In a distributed address space this reordering translates into explicit communication if the source and target location are in the local memory of different processors. Permutations of the form $P^{mp} \otimes I_n$, where $P^{mp} \in \mathbb{C}^{mp\times mp}$ is a permutation matrix, reorder mp chunks of n consecutive elements where m chunks reside in each processor's memory. This means that up to m messages of length n are to be sent and received per processor. Thus,

$$y = \underbrace{(P^{mp} \otimes I_n)}_{\text{par}(p)} x \tag{5}$$

encodes an all-to-all communication of p processors with message size n and the communication pattern described by P. For instance, when implementing

$$y = \underbrace{\left(L^4_2 \otimes I_2\right)}_{\text{par}(2)} x = \left[\begin{array}{cccc|cccc} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \hline \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 \end{array}\right] \cdot \left(\begin{array}{c} x_0 \\ x_1 \\ x_2 \\ x_3 \\ \hline x_4 \\ x_5 \\ x_6 \\ x_7 \end{array}\right),$$

processor 0 sends the message (x_2, x_3) to processor 1 and processor 1 sends the message (x_4, x_5) to processor 0.

In our example not all chunks of length m become messages. For instance, (x_0, x_1) and (x_6, x_7) stay in the memory of their respective processor. We capture this by decomposing (5) into a *local part* that copies data within the local memory of each processor and a *global part* that must be implemented using message passing. Formally, we decompose P in (5) into a sum of two matrices,

$$P = F + C,$$

and thus

$$P^{mp} \otimes I_n = (F \otimes I_n) + (C \otimes I_n).$$

Each “1” entry in P ends up either in F or C, hence the sum does not incur actual operations.

F contains all “1” entries of P within the block diagonal with blocks of size $m \times m$. It describes the addressing of all data chunks that stay within the local memory of each processor. $F \otimes I_n$ will be implemented as data copying by the respective processor.

C contains all remaining, i.e., off-blockdiagonal “1” entries. It describes the addressing of all data messages that have to be transmitted between two processors. $C \otimes I_n$ will be implemented using one send/receive pair per message.

To make the message addressing explicit, we further factor C as

$$C = SC'G \quad \text{with} \quad C',$$

where C' is a permutation matrix. This factorization is explained next. Assume, that P^{mp} requires kp messages ($k \leq m$). Then $C' \in \mathbb{C}^{kp \times kp}$ is a permutation matrix describing the message addressing. $C'_{i,j} = 1$ implies that message (j mod k) sent by processor $\lfloor j/k \rfloor$ is message (i mod k) received by processor $\lfloor i/k \rfloor$. G is a rectangular block-diagonal matrix of p blocks of size $k \times m$. G assigns k of the m data chunks within each processor’s local memory to one of the k messages to be sent by this processor. S is a rectangular block-diagonal matrix of p blocks of size $m \times k$. It stores the k messages received by each processor at their final location within the local memory of each processor.

Analysis of S, C', and G enables highly optimized implementations like using MPI collective communication functions or implementing $y = \big(P \otimes I_n\big)x$ inplace (vector x and y share the same memory location). For instance, if C' is symmetric and S=G^T (transpose), then $C \otimes I_n$ can be implemented inplace using send-receive-replace operations. The required analysis is implemented using the techniques described in [8]. Details of the analysis are beyond the scope of this paper.

As illustrative example we parallelize L^9_3 for 3 processors. We factor L^9_3 into the local matrix F and the communication matrices S, C', and G:

$$y = \underbrace{L^9_3}_{\text{par}(3)} x = Fx + SC'Gx.$$

The explicit form is shown next and represents the communication addressing pattern:

$$
\left(\begin{array}{c} y_0\\ y_1\\ y_2\\ \hline y_3\\ y_4\\ y_5\\ \hline y_6\\ y_7\\ y_8 \end{array}\right)
=
\left(\begin{array}{c} x_0\\ x_3\\ x_6\\ \hline x_1\\ x_4\\ x_7\\ \hline x_2\\ x_5\\ x_8 \end{array}\right)
=
\underbrace{\left[\begin{array}{ccc|ccc|ccc}
1&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot\\
\cdot&\cdot&\cdot&1&\cdot&\cdot&\cdot&\cdot&\cdot\\
\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&1&\cdot&\cdot\\ \hline
\cdot&1&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot\\
\cdot&\cdot&\cdot&\cdot&1&\cdot&\cdot&\cdot&\cdot\\
\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&1&\cdot\\ \hline
\cdot&\cdot&1&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot\\
\cdot&\cdot&\cdot&\cdot&\cdot&1&\cdot&\cdot&\cdot\\
\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&1
\end{array}\right]}_{\underbrace{L_3^9}_{\text{par}(3,\,\text{mpi})}}
\cdot
\left(\begin{array}{c} x_0\\ x_1\\ x_2\\ \hline x_3\\ x_4\\ x_5\\ \hline x_6\\ x_7\\ x_8 \end{array}\right)
=
\underbrace{\left[\begin{array}{ccc|ccc|ccc}
1&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot\\
\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot\\
\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot\\ \hline
\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot\\
\cdot&\cdot&\cdot&\cdot&1&\cdot&\cdot&\cdot&\cdot\\
\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot\\ \hline
\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot\\
\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot\\
\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&1
\end{array}\right]}_{F}
\cdot
\left(\begin{array}{c} x_0\\ x_1\\ x_2\\ \hline x_3\\ x_4\\ x_5\\ \hline x_6\\ x_7\\ x_8 \end{array}\right)
+
$$

$$
+
\underbrace{\left[\begin{array}{cc|cc|cc}
\cdot&\cdot&\cdot&\cdot&\cdot&\cdot\\
1&\cdot&\cdot&\cdot&\cdot&\cdot\\
\cdot&1&\cdot&\cdot&\cdot&\cdot\\ \hline
\cdot&\cdot&1&\cdot&\cdot&\cdot\\
\cdot&\cdot&\cdot&\cdot&\cdot&\cdot\\
\cdot&\cdot&\cdot&1&\cdot&\cdot\\ \hline
\cdot&\cdot&\cdot&\cdot&1&\cdot\\
\cdot&\cdot&\cdot&\cdot&\cdot&1\\
\cdot&\cdot&\cdot&\cdot&\cdot&\cdot
\end{array}\right]}_{S}
\underbrace{\left[\begin{array}{cc|cc|cc}
\cdot&\cdot&1&\cdot&\cdot&\cdot\\
\cdot&\cdot&\cdot&\cdot&1&\cdot\\ \hline
1&\cdot&\cdot&\cdot&\cdot&\cdot\\
\cdot&\cdot&\cdot&\cdot&\cdot&1\\ \hline
\cdot&1&\cdot&\cdot&\cdot&\cdot\\
\cdot&\cdot&\cdot&1&\cdot&\cdot
\end{array}\right]}_{C'}
\underbrace{\left[\begin{array}{ccc|ccc|ccc}
\cdot&1&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot\\
\cdot&\cdot&1&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot\\ \hline
\cdot&\cdot&\cdot&1&\cdot&\cdot&\cdot&\cdot&\cdot\\
\cdot&\cdot&\cdot&\cdot&\cdot&1&\cdot&\cdot&\cdot\\ \hline
\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&1&\cdot&\cdot\\
\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&1&\cdot
\end{array}\right]}_{G}
\cdot
\left(\begin{array}{c} x_0\\ x_1\\ x_2\\ \hline x_3\\ x_4\\ x_5\\ \hline x_6\\ x_7\\ x_8 \end{array}\right).
$$

The matrix F encodes that x_0 (in processor 0's memory), x_4 (in processor 1's memory), and x_8 (in processor 2's memory) do not require communication and are moved from x to y by their respective processors. The matrix G specifies which elements of the vector x become which message. In our example the data packets are x_1, x_2 (sent by processor 0), x_3, x_5 (sent by processor 1), and x_6, x_7 (sent by processor 2). C' is a 6×6 permutation matrix encoding the send/receive addressing of the data packets. For instance, the entry $C'_{4,1} = 1$ of $C' = [C'_{i,j}]_{i,j}$ describes that message 1 sent by processor 0 (x_2) is message 0 received by processor 2. The matrix S describes the final location of the received data packets. For instance, message 0 received by processor 2 (x_2) will be stored at location y_6. Figure 2 shows the MPI corresponding implementation.

Data redistribution. Formula (2) requires different data distributions for x and y. To capture this, we generalize the idea of all-to-all communication from the previous section to data redistributions. Permutations

$$P^m \otimes I_n \quad \text{with} \quad p, q \mid m,\ P^m \text{ permutation matrix} \in \mathbb{C}^{m\times m} \tag{6}$$

reorder m chunks of data of size n. Thus,

$$y = \underbrace{\left(P^m \otimes I_n\right)}_{\text{par}(q\leftarrow p)} x \tag{7}$$

redistributes data from p to q processors using message size n and with the message addressing encoded in P. We apply again the approach of the last section and decompose P into local

```
int proc[][] = {{1,2}, {0,2}, {0,1}},  // communication pattern
    msg[][]  = {{0,0}, {0,1}, {1,1}},
    SG[][]   = {{1,2}, {3,5}, {6,7}},
    F[]      = {0,1,2};

// parallel function, call by 3 MPI processes simultaneously
void L_9_3(double *yLocal, double *xLocal, int mpirank) {
   // output: yLocal[3], input xLocal[3]; part of x[9] and y[9]
   MPI_Request send[2], recv[2]; int i;
   yLocal[F[mpirank]] = xLocal[F[mpirank]]; // y = Fx +...
   for(i = 0; i < 2; i++){                  //     + SC'Gx
      // nonblocking send
      MPI_Isend(xLocal + SG[mpirank][i],    // source ofs
         1, MPI_DOUBLE,
         proc[mpirank][i],                  // receiving proc
         i,                                 // msg id
         MPI_COMM_WORLD, send + i);
      // nonblocking receive
      MPI_Irecv(yLocal + SG[mpirank][i],    // target ofs
         1, MPI_DOUBLE,
         proc[mpirank][i],                  // sending proc
         msg[mpirank][i],                   // msg id to get
         MPI_COMM_WORLD, recv + i);
   }
   MPI_Waitall(1, recv, MPI_STATUSSES_IGNORE);
}
```

Figure 2. MPI program implementing $y = L^9_3 x$ on 3 processors.

operations and communication to generated MPI code:

$$P = F + C.$$

As an example of a redistribution from 2 to 3 processors consider

$$y = \underbrace{L^6_2}_{\text{par}(3\leftarrow 2)} x = \left(\begin{array}{c} x_0 \\ x_2 \\ \hline x_4 \\ x_1 \\ \hline x_3 \\ x_5 \end{array}\right) = \left[\begin{array}{ccc|ccc} 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \hline \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \hline \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 \end{array}\right] \cdot \left(\begin{array}{c} x_0 \\ x_1 \\ x_2 \\ \hline x_3 \\ x_4 \\ x_5 \end{array}\right).$$

Processor 0 sends the message x_1 to processor 1. x_0 and x_2 stay in the memory of processor 0. Processor 1 sends the message x_3 and x_5 to processor 2 and receives x_1 from processor 0. x_4 stays in in the memory of processor 1.

Parallelization through formula rewriting. Above we introduced formula constructs that can be implemented as parallel computation or as communication. Products of these formulas can be implemented as a sequence of parallel communication and communication steps. This gives rise to the following definition.

Definition 1 (Parallelized formula). *Formulas of the form* (3), (4), (5), (7), *and products of these formulas are called parallelized. Parallelized formulas can be implemented using MPI.*

However, not all formulas are parallelized. For instance, the right-hand side of (1) is not a parallelized formula. Thus, we introduce a set of rewriting rules to use SPIRAL's rewriting system to transform formulas into parallelized formulas. This rule set is summarized in Table 2 and is one of the contributions of this paper. The rule set is designed for the generation of DFT MPI code. Using this rule set, SPIRAL can automatically parallelize formulas for the DFT at a high level of abstraction.

As a small example of the workings of the rewriting system consider

$$y = \underbrace{(I_m \otimes A_n)}_{\text{par}(p)} x. \tag{8}$$

(8) is not parallelized for $m \neq p$. Assuming $p|m$, the application of rule (13) transforms (8) into

$$y = \underbrace{\big(I_p \otimes (I_{m/p} \otimes A_n)\big)}_{\text{par}(p)} x$$

which matches (3) and is thus parallelized in the sense of Definition 1. A more elaborate example showing the parallelization of a DFT_{mn} and rescaling it from p to q processors is given in the next section.

$$\underbrace{AB}_{\text{par}(p)} \to \underbrace{A}_{\text{par}(p)} \underbrace{B}_{\text{par}(p)} \tag{9}$$

$$\underbrace{A}_{\text{par}(p)} \to \underbrace{A}_{\text{par}(p\leftarrow q\leftarrow p)}, \quad q|p \tag{10}$$

$$\underbrace{AB}_{\text{par}(p\leftarrow q\leftarrow p)} \to \underbrace{A}_{\text{par}(p\leftarrow q)} \underbrace{B}_{\text{par}(q\leftarrow p)} \tag{11}$$

$$\underbrace{AB}_{\text{par}(q\leftarrow p)} \to \underbrace{A}_{\text{par}(q)} \underbrace{B}_{\text{par}(q\leftarrow p)} \tag{12}$$

$$\underbrace{I_m \otimes A_n}_{\text{par}(p)} \to \underbrace{I_p \otimes \big(I_m \otimes A_n\big)}_{\text{par}(p)} \tag{13}$$

$$\underbrace{(A_m \otimes I_n)}_{\text{par}(p\leftarrow q)} \to \underbrace{L^{mn}_m}_{\text{par}(p\leftarrow q)} \underbrace{(I_n \otimes A_m) L^{mn}_n}_{\text{par}(q)} \tag{14}$$

$$\underbrace{(A_m \otimes I_n)}_{\text{par}(q\leftarrow p)} \to \underbrace{L^{mn}_m (I_n \otimes A_m)}_{\text{par}(q)} \underbrace{L^{mn}_n}_{\text{par}(q\leftarrow p)} \tag{15}$$

$$\underbrace{L^{mn}_m}_{\text{par}(p)} \to \underbrace{\big(I_p \otimes L^{mn/p}_{m/p}\big)\big(L^{p^2}_p \otimes I_{mn/p^2}\big)\big(I_p \otimes (L^n_p \otimes I_{m/p})\big)}_{\text{par}(p)} \tag{16}$$

$$\underbrace{L^{mn}_m}_{\text{par}(q\leftarrow p)} \to \underbrace{\big(I_q \otimes (I_{p/q} \otimes L^{mn/p}_{m/p})\big)}_{\text{par}(q)} \underbrace{\big(L^{p^2}_p \otimes I_{mn/p^2}\big)}_{\text{par}(q\leftarrow p)} \underbrace{\big(I_p \otimes (L^n_p \otimes I_{m/p})\big)}_{\text{par}(p)} \tag{17}$$

$$\underbrace{L^{mn}_m}_{\text{par}(p\leftarrow q)} \to \underbrace{\big(I_p \otimes L^{mn/p}_{m/p}\big)}_{\text{par}(p)} \underbrace{\big(L^{p^2}_p \otimes I_{mn/p^2}\big)}_{\text{par}(p\leftarrow q)} \underbrace{\big(I_q \otimes (I_{p/q} \otimes L^n_p \otimes I_{m/p})\big)}_{\text{par}(q)} \tag{18}$$

Table 2. Parallelization and rescaling rewriting rules.

4 Rescaling FFTs Using SPIRAL

The framework developed in Section 3 allows us to explore trade-offs between communication and computation. Assume a subroutine computing a DFT on p processors in parallel. Depending on the cost of communication and the speed of processors, computing on $q < p$ processors may speed up the computation. However, the initial and final data distribution on p processors is fixed by the subroutine's interface. In this situation the performance gain by computing on only q processors can easily be lost in the necessary data redistribution from p to q processors before the computation and q to p processors after the computation.

Rescaling. Using the parallelization rules in Table 1 we can systematically derive formulas that internally use less processors than at the beginning and at the end of the computation. Further, the necessary redistribution is performed as part of the communication that has to be done anyway. Thus, these formulas are candidates to speed up the whole computation without changing the subroutine interface. We call this approach *rescaling*.

Specifically, we perform downscaling (to fewer processors) together with the first occurring communication step, while upscaling is performed with the last communication step. Hence, all encapsulated communication steps profit of the reduced communication effort.

After choosing a number of processors to rescale to, there is still to decide which q of the p processors to use for calculation. On machines with non-uniform communication structure (for instance clusters of symmetric multiprocessors) this can be an important choice that strongly influences the achieved performance.

In SPIRAL, the formula rewriting is performed automatically; SPIRAL's search will find a formula, and thus a rescaling strategy that performs fastest on the given platform.

Example: Rescaled DFT. We show the rewriting process that parallelizes a DFT_{mn}, for $p \mid m, n$, across p processors and rescales it to $q \mid p$ processors for the intermediate computation steps. In SPIRAL, this derivation is done automatically. We tag DFT_{mn} for p processors and expand it using rules (1) and (10)–(12):

$$\underbrace{\mathrm{DFT}_{mn}}_{\mathrm{par}(p)} \to \underbrace{(\mathrm{DFT}_m \otimes I_n)}_{\mathrm{par}(p\leftarrow q)} \underbrace{T_n^{mn}}_{\mathrm{par}(q)} \underbrace{(I_m \otimes \mathrm{DFT}_n)}_{\mathrm{par}(q)} \underbrace{L_m^{mn}}_{\mathrm{par}(q\leftarrow p)} .$$

This introduces rescaling to q processors. Next we apply rules (9), (14), and (16)–(18) to formally parallelize:

$$\begin{aligned}
\to & \underbrace{\big(I_p \otimes L_{m/p}^{mn/p}\big)}_{\mathrm{par}(p)} \underbrace{\big(L_p^{p^2} \otimes I_{mn/p^2}\big)}_{\mathrm{par}(p\leftarrow q)} \underbrace{\big(I_q \otimes (I_{p/q} \otimes L_p^n \otimes I_{m/p})\big)}_{\mathrm{par}(q)} \underbrace{\big(I_q \otimes (I_{n/q} \otimes \mathrm{DFT}_m)\big)}_{\mathrm{par}(q)} \\
& \cdot \underbrace{\big(I_q \otimes L_{m/q}^{mn/q}\big)}_{\mathrm{par}(q)} \underbrace{\big(L_q^{q^2} \otimes I_{mn/q^2}\big)}_{\mathrm{par}(q)} \underbrace{\big(I_q \otimes (L_q^n \otimes I_{m/q})\big)}_{\mathrm{par}(q)} \underbrace{T_n^{mn}}_{\mathrm{par}(q)} \underbrace{\big(I_q \otimes (I_{m/q} \otimes \mathrm{DFT}_n)\big)}_{\mathrm{par}(q)} \\
& \cdot \underbrace{\big(I_q \otimes (I_{p/q} \otimes L_{m/p}^{mn/p})\big)}_{\mathrm{par}(q)} \underbrace{\big(L_p^{p^2} \otimes I_{mn/p^2}\big)}_{\mathrm{par}(q\leftarrow p)} \underbrace{\big(I_p \otimes (L_p^n \otimes I_{m/p})\big)}_{\mathrm{par}(p)} .
\end{aligned} \tag{19}$$

Inspection shows that the final expression is parallelized in the sense of Definition 1.

Analysis. The communication and computation cost of (19) depends on the choice of the scaling factor $k = p/q$. Table 3 summarizes the effect of scaling on packet size, number

computation	data volume	packet size	#packets
$O(k)$	$O(1)$	$O(k^2)$	$O(1/k^2)$

Table 3. Effect of rescaling by $k = p/q$ on computation and communication.

of packets, computation cost, and total data to be transmitted as function of k. In essence, scaling down keeps the overall amount of data to be transmitted practically constant while increasing the message size and the computation cost. The best choice of q depends on the relation between the speed of the processor, the communication latency, and the bandwidth.

5 Experimental Results

In this section we evaluate our approach. We first show that rescaling speeds up smaller DFTs. Then we show that our generated DFT programs compare favorably with FFTW.

Benchmark setup. All experiments were done with complex-to-complex double-precision 2-power FFTs. The platform is a cluster of AMD Opteron 250 CPU dual nodes running at 2.4 GHz, connected by a Mellanox InfiniBand high speed network with a theoretical peak of 10 Gb/s and 4 μs latency. All codes were compiled using the GNU C compiler 3.4.4 with the option `-O3` and linked with the mvapich 0.9.5 MPI library. Performance data is given in pseudo Mflop/s computed as $5n \log n/t$, where n is the DFT size and t the runtime in microseconds. This measure is proportional to inverse runtime and hence preserves runtime relationships. Further, it gives an indication of the absolute floating-point performance [12].

Experiment 1: Rescaling. Figures 3 (i)–(iii) show the performance impact of rescaling for the problem sizes 2^{12}, 2^{15}, and 2^{18}. We start with $p = 16$ processors and let SPIRAL generate downscaling programs for $q = 1$, 2, 4, 8, and 16 processors (16 processors implies no downscaling). We compare the performance of the original and the downscaled programs to FFTW-MPI running on 16 processors. Figures 3 (i) and (ii) show a performance peak at $q = 8$ processors. Thus, for the sizes 2^{12} and 2^{15} we gain from downscaling. Figure 3 (iii) shows that $p = 16$ processors are required for the best performance at problem size 2^{18}. For this size, the increased workload per processor overcompensates the gain in communication speed.

On the benchmark platform, rescaling speeds up only for smaller sizes. On machines with slower, higher-latency networks we saw performance gains due to rescaling for larger problem sizes.

Experiment 2: Comparison to FFTW. Figure 4 (ii) shows the speed-up of SPIRAL generated FFT programs run on 16 CPUs *without* downscaling, and *with* optimal downscaling (8 CPUs for small sizes), compared to FFTW-MPI 2.1.5 using 16 CPUs. For sizes up to 2^{17}, downscaling provides significant performance gains. For these sizes SPIRAL generated programs are up to 80% faster than FFTW-MPI. For larger sizes, SPIRAL's performance is comparable to FFTW-MPI.

Figure 4 (i) shows the same experiment, but for 8 CPUs. The optimal downscaling found in this case is to 4 CPUs for small sizes. SPIRAL generated MPI programs are between 1.5 and 2.5 times faster than FFTW-MPI, showing higher relative speed for problems smaller than 2^{16}.

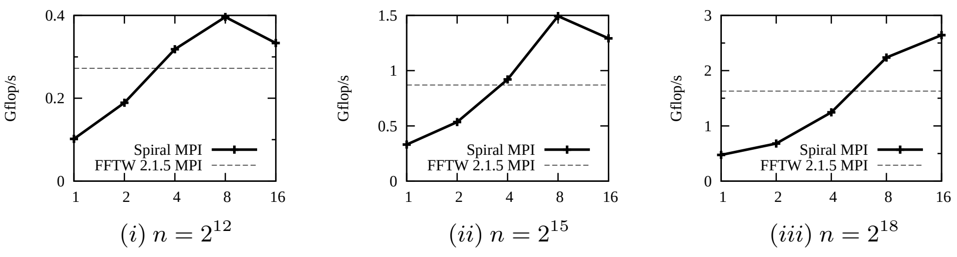

Figure 3. Effect of downscaling from $p = 16$**.** The plots show, for three DFT sizes n, the best performance obtained for different scaling factors $k = p/q$. $p = 16$ and the x-axis is labeled with q. The dashed line is the performance achieved by FFTW. Higher is better.

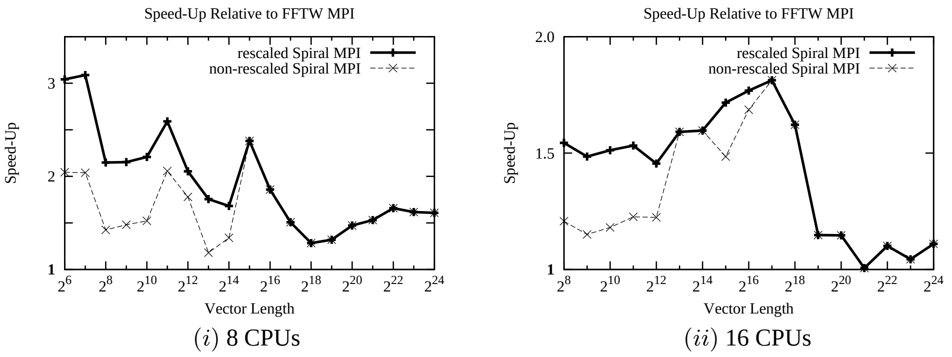

Figure 4. Relative performance. Performance of SPIRAL generated MPI FFT programs without downscaling (dashed), and optimally rescaled (solid), relative to FFTW-MPI 2.1.5. Higher is better.

6 Conclusion

We presented a formal framework for generating efficient MPI algorithms by rewriting formulas representing FFT algorithms. We applied the framework to implement the idea of flexible rescaling and thus enable adaptation to a platform's characteristics. By including the framework into SPIRAL's infrastructure, the entire implementation and adaptation process is automated. It is worth pointing out that we used very similar approaches before to the related problems of vectorization and shared memory parallelization. In fact, all these optimizations are performed using the same infrastructure in SPIRAL. Since our approach is formula based, it is domain-specific but can be generalized to other linear transforms.

Ongoing work aims to enable SPIRAL to optimize the runtime of the DFT including possible data redistributions. This way, the user can specify the desired data layout before and after the computation to interface with his application. As both data distribution and transform are represented on a mathematical level they can be optimized jointly, thus reducing the overhead.

References

1. A. Adelmann, A. Bonelli, W. P. Petersen, and C. W. Ueberhuber. Communication efficiency of parallel 3D FFTs. In *Proc. High Performance Computing for Computational Science (VECPAR)*, volume III, pages 901–907, 2004.

2. G. Baumgartner, A. Auer, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. J. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. Lam, Q. Lu, M. Nooijen, R. M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *In [17]*, pages 276–292, 2005.
3. 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.
4. N. Dershowitz and D. A. Plaisted. Rewriting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 9, pages 535–610. Elsevier, 2001.
5. M. Eleftheriou, B. Fitch, A. Rayshubskiy, T. C. Ward, and R. Germain. Scalable framework for 3D FFTs on the Blue Gene/L supercomputer: Implementation and early performance measurements. *IBM Journal of Research and Development*, 49(2/3):457–464, 2005.
6. A. Faraj and X. Yuan. Automatic generation and tuning of MPI collective communication routines. In *Proc. International Conference on Supercomputing (ICS)*, pages 393–402, 2005.
7. F. Franchetti and M. Püschel. A SIMD vectorizing compiler for digital signal processing algorithms. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, pages 20–26, 2002.
8. F. Franchetti, Y. Voronenko, and M. Püschel. Loop merging for signal transforms. In *Proc. Programming Language Design and Implementation (PLDI)*, pages 315–326, 2005.
9. F. Franchetti, Y. Voronenko, and M. Püschel. FFT program generation for shared memory: SMP and multicore. In *Proc. Supercomputing (SC)*, 2006.
10. F. Franchetti, Y. Voronenko, and M. Püschel. A rewriting system for the vectorization of signal transforms. In *Proc. High Performance Computing for Computational Science (VECPAR)*, 2006. On CD-ROM.
11. M. Frigo. A fast Fourier transform compiler. In *Proc. Programming Language Design and Implementation (PLDI)*, pages 169–180, 1999.
12. M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 3, pages 1381–1384. IEEE, 1998.
13. M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *In [17]*, pages 216–231, 2005.
14. G. Goumas, N. Drosinos, M. Athanasaki, and N. Koziris. Automatic parallel code generation for tiled nested loops. In *Proc. Symposium on Applied Computing (SAC)*, pages 1412–1419. ACM Press, 2004.
15. 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 Blue Gene/L platform using the Qbox code. In *Proc. Supercomputing (SC)*, page 24, 2005.
16. J. Johnson and K. Chen. A self-adapting distributed memory package for fast signal transforms. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, page 44a, 2004.
17. J. M. F. Moura, M. Püschel, D. Padua, and J. Dongarra, editors. *Special Issue on Program Generation, Optimization, and Platform Adaptation*, volume 93(2) of *Proceedings of the IEEE*, 2005.
18. J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. Dongarra. Performance analysis of MPI collective operations. *Cluster Computing Journal, Special Issue on Performance Modeling and Evaluation of Parallel and Distributed Systems*, 2006. Accepted for publication.
19. 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 [17]*, pages 232–275, 2005.
20. Spiral web site. www.spiral.net.
21. C. Van Loan. *Computational Frameworks for the Fast Fourier Transform*, volume 10 of *Frontiers in Applied Mathematics*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, 1992.