Parallel FFT With Eden Skeletons*

Jost Berthold, Mischa Dieterle, Oleg Lobachev, and Rita Loogen

Philipps-Universität Marburg, Fachbereich Mathematik und Informatik
Hans Meerwein Straße, D-35032 Marburg, Germany
{berthold,dieterle,lobachev,loogen}@informatik.uni-marburg.de

Abstract. The paper investigates and compares skeleton-based Eden implementations of different FFT-algorithms on workstation clusters with distributed memory. Our experiments show that the basic divide-and-conquer versions suffer from an inherent input distribution and result collection problem. Advanced approaches like calculating FFT using a parallel map-and-transpose skeleton provide more flexibility to overcome these problems. Assuming a distributed access to input data and reorganising computation to return results in a distributed way improves the parallel runtime behaviour.

1 Introduction

The well-known Fourier transform, which describes frequency distribution in a signal, finds diverse applications from pure mathematical applications to real-life scenarios such as digital signal processing. Today's state of the art is the *Fast Fourier Transform* (FFT). Cooley and Tukey [4] were the first to propose an FFT algorithm in 1965 (known as 2-radix FFT) with time complexity $O(n \log n)$. A range of other FFT algorithms have been discovered since then [16].

Developing an efficient parallel distributed-memory implementation of FFT is a great challenge. The manual of the recent 3.2 alpha release of FFTW[1] warns that "distributed-memory parallelism can easily pose an unacceptably high communications overhead for small problems". In the broader context of an implementation for parallel computer algebra algorithms in the parallel Haskell extension Eden [15, 13], we investigate parallelisation strategies for different FFT algorithms. The goal of our work has not been to develop the fastest distributed-memory FFT, but to investigate a skeleton-based parallelisation of FFT. In Eden, skeletons [3, 14, 1] are higher-order functions defining general parallel evaluation schemes. The skeleton approach to parallelisation cleanly separates problem-related and problem-independent issues. This simplifies the parallelisation of algorithms enormously. In essence, FFT algorithms are based on divide-and-conquer strategies. In this paper, we utilize skeletons for two variations of parallel divide-and-conquer evaluations: a *distributed expansion* scheme which unfolds the computation tree dynamically and spawns

* Supported by the DFG grant LO 630-3/1.

[1] Fastest Fourier Transform in the West, http://www.fftw.org/fftw-3.2alpha3-doc/

parallel processes for the evaluation of sub-trees as long as processor elements are available, and a *flat expansion* scheme which unfolds the tree up to a given depth and evaluates all sub-trees at this depth in parallel. Moreover, we present a parallel *map-and-transpose skeleton* for the implementation of more advanced FFT methods. Our skeletons are applicable to a whole *class* of algorithms, those which rely on fixed-branching divide-and-conquer or parallel map-and-transpose schemes.

We analyse the parallel runtime behaviour of various skeleton/algorithm combinations using activity profiles of parallel program executions on networks of workstations, i. e. distributed-memory parallel machines. In addition, we investigate their scalability when increasing the number of processor elements.

Plan of Paper. The following two sections elaborate on divide-and-conquer approaches of parallel FFT (Section 2) and on advanced approaches (Section 3). In each section, we will describe appropriate skeletons for the parallelisation of FFT algorithms and an experimental evaluation of the parallelised algorithms. Section 4 discusses related work, the final section concludes.

2 Divide-and-Conquer FFT

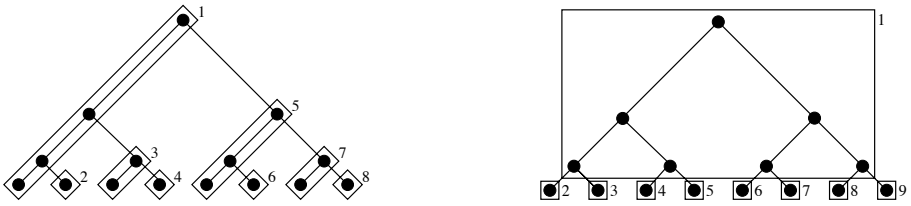
FFT Algorithms. The classic *2-radix FFT algorithm by Cooley and Tukey* divides the input vector xs of length n into two halves, computes their element-wise sum and difference, and multiplies the latter with powers of an n-th primitive root of unity, the *twiddle factors*. The algorithm recursively computes the FFT of these vectors, and combines the results simply by interleaving them element-wise. Recursion ends at singleton vectors which are returned unmodified. This version is called *decimation in frequency*.

An alternative version, called *decimation in time*, essentially consists of the opposite dividing and combining steps. The input vector is split into the sub-vectors with even and odd indices (inverse to the interleaving step above). After evaluating the recursive calls of FFT for the sub-vectors, the more complex combination of the result lists follows. The first and second half of the overall result are defined as element-wise sums and differences including again a multiplication with the twiddle factors.

Divide-and-Conquer Skeletons. The essence of a divide-and-conquer algorithm is to decide whether the problem is trivial and, in this case, to solve it, or else to decompose non-trivial problems into a number of sub-problems, which are solved recursively, and to combine the output. A general skeleton takes parameter functions for this functionality, as shown here:

```
type DivideConquer a b = (a -> Bool) -> (a -> b)      -- trivial? / solve
                       -> (a -> [a]) -> ([b] -> b) -- split / combine
                       -> a -> b                   -- problem / result
```

The resulting structure is a tree of task nodes where child nodes are the sub-problems, the leaves representing trivial tasks.



(a) Binary distributed expansion, depth 3

(b) Binary flat expansion, depth 3

Fig. 1. Divide-and-conquer expansion schemes.

A fundamental Eden skeleton which specifies a general divide-and-conquer algorithm structure can be found in [14]. In [1], we have refined and adapted this skeleton for fixed branching divide-and-conquer algorithms like FFT. Two different basic strategies have been used to unfold a process tree. The *distributed expansion scheme* creates the process tree in a *distributed* fashion: One of the tree branches is processed locally, the others are instantiated as new processes, as long as processor elements are available. This results in a *distributed expansion* of the computation (cf. Fig. 1(a)). Explicit placement of processes is essential to achieve a balanced distribution of processes on the available processor elements. The boxes indicate which tree node are evaluated by the same process. The numbers indicate a possible placement on 8 processor elements (PEs). The corresponding skeleton has the following interface (type):

```
dcN :: (Trans a, Trans b) =>
       Int -> [Int] ->    -- branching degree / processor elements
       DivideConquer a b
```

The Eden type class `Trans` provides internally used communication functions. The first two skeleton parameters determine the fixed branching degree of the underlying divide-and-conquer tree and a list of available processor numbers used for explicit process placement.

In the *flat expansion skeleton*, the main process unfolds the divide-and-conquer tree up to a given depth, usually with more branches than available PEs. The resulting subtrees are then evaluated by parallel processes, the main process combines the results of the sub-processes. This results in a homogeneous *flat expansion* scheme from a single source depicted in Fig. 1(b) for the binary variant. A uniform distribution of the subtrees on processors can be achieved using a farm of worker processes with static or dynamic task distribution. The corresponding skeleton has the following interface (type):

```
dcDM_N :: (Trans a, Trans b) =>
          Int -> Int ->    -- unfolding depth / branching degree
          DivideConquer a b
```

Here the first two skeleton parameters determine the unfolding depth of the underlying divide-and-conquer tree and the fixed branching degree. A detailed

```
-- Parallel 2-radix FFT, decimation in time, with input
-- chunking size, instantiates dcN skeleton
fft2radixTime    :: Int -> [Complex Double] -> [Complex Double]
fft2radixTime c xs
  = chunkDC c chunkL concat
      (dcN 2 [2..noPe]) isSingleton id (unshuffle 2) combine2
```

Fig. 2. Parallelisation by Skeleton Instantiation

explanation of these Eden divide-and-conquer skeletons can be found in [1]. Fig. 2 shows a sample instantiation of the `dcN` skeleton with branching degree 2 and explicit process placement on PEs 2 to `noPe` (the number of available PEs). Input vectors (lists) are chunked into larger pieces to reduce communication costs.

Experimental Results. The following runtime experiments have been performed on a local network of 8 Linux workstations with Core 2 Duo processors and 2 GB RAM connected by Fast Ethernet. The Eden runtime system is instrumented in such a way that a runtime flag activates a tracing mechanism which protocols parallelism-related events like process/thread creation/termination, state changes of machines (i. e. processors), processes and threads, and message sending and receiving. The trace files can then be visualised by the EdenTV tool (Eden Trace Viewer) [2]. The resulting graphics (see e.g. Figure 3) which are best viewed in colour are two-dimensional diagrams. The time scale is on the horizontal axis. The vertical axis shows the machine numbers, on which the processes are placed. For each process, there is a coloured horizontal bar, which shows the process states over time. Green parts (grey) indicate that a thread is working, red parts (dark grey) indicate that all threads of the process are blocked, usually because they are waiting for input, or because the processor is communicating. Yellow areas (light grey) indicate that there are runnable threads but some system activity like e. g. garbage collection is taking place. Data transfer, i. e. messages can be optionally indicated as arrows from the sending to the receiving process.

Our first experiments tested the standard Cooley-Tukey 2-radix FFT algorithm variants decimation in frequency and decimation in time with the distributed expansion and flat expansion skeletons. Figure 3 shows typical traces and the runtimes obtained with the following parameters: input size 2^{20} (double precision complex numbers), chunk size[2] 1500, recursion depth 4 and heap size 1500MB.

The activity profiles in Figure 3 reveal that the flat expansion skeleton leads to a much better runtime behaviour than the distributed expansion skeleton. This is due to the good load balance in the worker processes which start immediately. Note that the skeleton even co-locates one worker process with the main process on machine 1 (lowest bars). The communication overhead is low — only 80 messages were sent in both versions.

The decimation in frequency flat expansion version was the fastest version with 6.92 s. This is due to the fact that the post processing in the master can

[2] The chunk size is only used by the distributed expansion skeleton to reduce the number of messages.

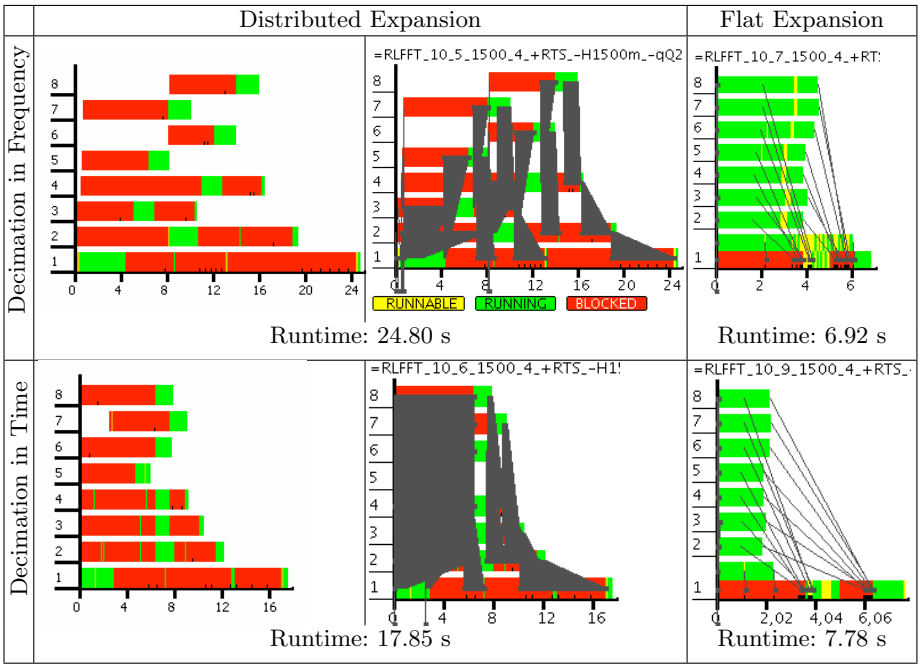


Fig. 3. Traces and runtimes of divide-and-conquer approaches, without/with messages.

be done very fast, because combining the results is a trivial shuffle, while the top level combining phase of the decimation in time version takes almost three quarters of the overall runtime.

With the flat expansion skeleton, we eliminate the input communication, i. e. distributing tasks to the worker processes. Each worker receives the whole unevaluated task specification and evaluates its own part on demand. Contrarily, work distribution is slower with the distributed expansion skeleton because the main process distributes the tasks to all worker processes. These are initially blocked waiting for their tasks and start working at different points in time. This leads to an inhomogeneous runtime behaviour.

3 Advanced Approaches

The parallel divide-and-conquer FFT-implementations show an acceptable performance using few processors, but do not scale well. Therefore we have implemented a more sophisticated algorithm taken from [7] which minimizes data dependencies and provides more fine grained parallelism. The input vector is divided into rows of a matrix with side lengths $l = 2^k$. Thus, the input vector is of length $n = l^2 = 4^k$. The algorithm consists of three phases:

1. preprocessing: permutation of input in bit reverse order, tagging input elements with their position and their segment's length, split into rows.
2. central processing: local `fft3` ∘ a global transpose ∘ local `fft3`

3. postprocessing: concat and remove tags

The key difference between the ordinary sequential FFT and `fft3` is that the latter operates on triples which contain additional information like a position tag. It works with *global* twiddle factors to simulate a contiguous, single-dimensional FFT algorithm. The `divide` step is a trivial split of lists. The combine step needs to be modified using the additional information in the triples. Because of the permuted input, it is possible to perform FFT locally on the available subsets of global lists in a global manner. For more details, see [7].

We have derived a skeleton for the central phase of the above scheme which consists of a composition of parallel `map`s and an intermediate global communication to implement the global transpose. The skeleton has been inspired by the distributable homomorphism skeleton of Gorlatch and Bischof [8]. It can also be used for the distributed-memory FFT algorithms proposed in [17, 10].

The Parallel Map-and-Transpose Skeleton implements the functionality

`(parMap f1)` ∘ `transpose` ∘ `(parMap f2)`.

Defining it with this simple function composition is not appropriate, because all data would be gathered in the main process in between the two parmap phases. This again would provoke too much communication and process creation overhead. Our skeleton `parMapTranspose` includes a distributed transpose phase in between two parallel map evaluations. The skeleton's input is a matrix which will be distributed row cyclic. In our application the functions `f1` and `f2` will be sequential `fft3` invocations. In order to save the costs of input distribution, the parallel maps are executed by direct mapping [12] which means that the matrix is not communicated but transferred unevaluated within the process abstraction's body. The child processes will then evaluate the needed parts locally and demand driven.

The Eden code of the skeleton uses the following Eden constructs. The library function `spawn :: (Trans a, Trans b) => [Process a b] -> [a] -> [b]` creates a list of processes from a list of process abstractions and a list of corresponding process inputs. A process abstraction is a function that will be evaluated by a process. `Process` is the type constructor for Eden Process abstractions, which are created by the function `process :: (a -> b) -> Process a b`. Eden provides the following functions to dynamically define new input channels for processes. The Eden function `createChans :: Int` → `([ChanName a], [a])` creates a list of new (input) channel names. Data (lazily) received via the channels can be accessed in the second component of the result tuple of `createChans`. Channel names can be communicated to other processes which can write into the corresponding channels with the Eden function `multifill :: [ChanName a]` → `[a]` → `b` → `b`, which concurrently passes data via given channels and returns its third argument.

The code of the parallel map-and-transpose skeleton `parMapTranspose` is shown in Figure 4. The distributed `map` functionality is easily defined. Let `np` be the number of available PEs (processing elements). We divide the matrix rows into `np` contiguous blocks using the function `unshuffle`. At the end the final result

```
parMapTranspose :: Int -> ([a] -> [b]) -> ([b] -> [c]) -> [[a]] -> [c]
parMapTranspose np f1 f2 matrix = shuffle res
  where
   myProcs css = spawn [ process (distr2d_f np f1 f2 rows)
                       | rows <- unshuffle np matrix ] css
   (res,chanss) = myProcs $ transpose chanss

distr2d_fs ::  Int -> ([a] -> [b]) -> ([b] -> [c]) ->
               [[a]] -> [ChanName[b]] -> ([[c]],[ChanName [b]])
distr2d_fs np f1 f2 rows theirChanNs
  = let (myChanNs, theirFstRes) = createChans np
        intermediateRes  = map f1 rows
        myFstRes         = unshuffle np $ transpose intermediateRes
        res              = map f2 $ shuffleMatrixFracs theirFstRes
    in  (multifill theirChanNs myFstRes $ res, myChanNs)

-- types of auxiliary functions
-- round robin distribution and combination of list elements
unshuffle :: Int -> [a] -> [[a]]
shuffle   :: [[a]] -> [a]
-- combine n matrix fragments into one matrix
shuffleMatrixFracs :: [[[a]]] -> [[a]]
```

Fig. 4. Parallel map-and-transpose skeleton.

is re-composed using the inverse function `shuffle`. As many processes as available PEs are created using the Eden function `spawn`. Each process applies the function `distr2d_fs np f1 f2` to its portion of rows and the lazily communicated input (a row of `css`). The latter consists of a list of `np` channel names which are used to establish a direct link to all processes: each process can thus send data directly to each other process[3]. Each process evaluates the function `distr2D_fs` which firstly leads to the creation of `np` input channel names `myChanNs` for the corresponding process. These are returned to the parent process in the second component of the result tuple of `distr2d_fs`. The parent process receives a whole matrix `chanss :: [[ChanName a]]` of channel names (`np` channel names from `np` processes), which it transposes before sending them row-wise back to the child processes. Each process receives thus lazily `np` channel names `theirChanNs` for communicating data to all processes. The parallel transposition can thus occur without sending data through the parent process.

After the first `map f1` evaluation, a process locally `unshuffle`s the columns of the result (the locally transposed result rows) into `np` lists. These are sent via the received input channels of the other processes using the function `multifill`. The input for the second `map` phase is received via the initially created own input channels. The column fragments are composed to form rows of the transposed intermediate result matrix. The second `map f2` application produces the final result of the child processes.

[3] To simplify the specification the channel list even contains a channel which will be used by the process to transfer data to itself.

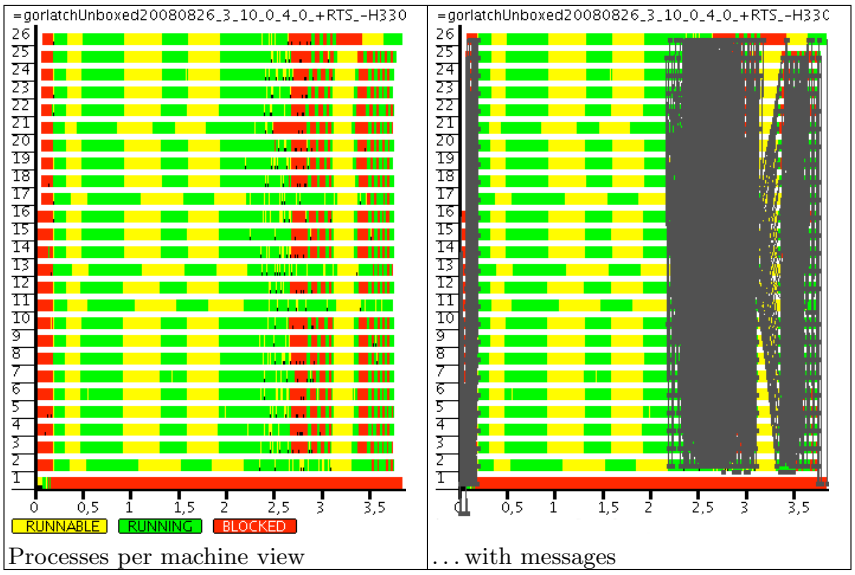


Fig. 5. Trace of parallel FFT using map-and-transpose skeleton (input size 2^{20}, 3.5 seconds on 26 Pentium 4 machines)

Experimental Results. The following traces and runtime measurements have been obtained on a Beowulf cluster at Heriot-Watt-University, Edinburgh, which consists of 32 Intel Pentium 4 SMP processors running at 3 GHz with 512 MB RAM and a Fast Ethernet interconnection. We implemented the FFT version of Gorlatch and Bischof [7] using our map-and-transpose skeleton. Result collection and post processing (a simple `shuffle`) have been omitted leaving the result matrix in a distributed column-wise manner. A runtime trace, again for input size 4^{10}, is depicted in Figure 5. The communication provoked by the distributed transpose phase overlaps the second computation phase, such that stream communication and computation terminate almost at the same time. The first computation phase is dominant because of the preprocessing, in particular the reordering (bit reversal) of the input list and the computation of the twiddle-factors. Noticeable are also the frequent "runnable" phases, which are garbage collections.

Figure 6 shows the runtimes of the parallel map-and-transpose FFT version with and without final result collection (Figure 6, triangle marks) in comparison with the best divide-and-conquer versions (4-radix[4], Flat Expansion, Decimation in Time and Frequency). We have measured these versions on the Beowulf cluster and on our local network of dual-core machines, which are more powerful and have more RAM than the Beowulf nodes. The parallel map-and-transpose versions scale well when increasing the number of processing elements. However, for a small number of PEs it is less efficient than the divide-and-conquer versions discussed in Section 2. Including result collection in the map-and-transpose

[4] 4-radix divides the input into 4 parts instead of two.

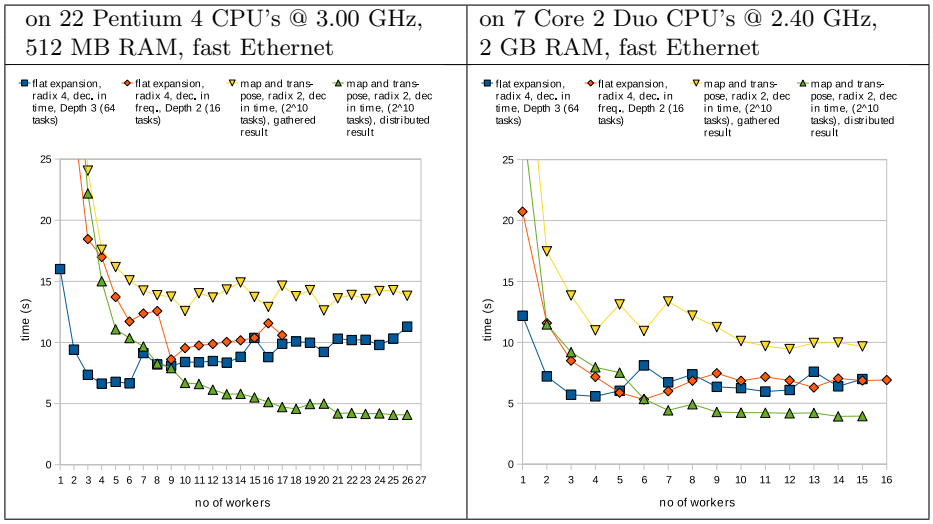


Fig. 6. Runtime and scalability comparison of parallel FFT approaches.

version decreases the performance clearly. The runtime differences of the various versions are less distinct on the powerful dual-core processors than on the Beowulf nodes. The huge performance penalties of the algorithms with a small number of worker processes on the Beowulf are due to more garbage collection rounds, because of the limited memory size.

4 Related Work

A range of parallel FFT implementations have been presented in the past ([6, 5], to mention only a few). The vast majority is tailored for shared-memory systems, see e. g. [9] as an example for a high-level implementation in the functional array language SAC or [1] for experiments with our divide-and-conquer skeletons on multi-core machines. Distributed implementations are mostly based on C+MPI. The distributed MPI-based FFTW implementation [6] is especially tailored for transforming arrays so large that they do not fit into the memory of a single processor. In contrast to these specialised approaches, our work propagates a skeleton-based parallelisation. In his PhD thesis [11], Christoph Herrmann gives a broad overview, classification, and a vast amount of implementation variants for divide-and-conquer, while we have focused on divide-and-conquer schemes with a fixed branching degree. The skeleton-based version of parallel FFT in [8, 7] underlies our parallel map-and-transpose implementation of FFT.

5 Conclusions

The skeleton approach to the parallelisation of FFT provides a high flexibility. In total, six different parallel FFT approaches have been compared, on the basis of three different skeletons: two parallel divide-and-conquer and a parallel

map-and-transpose skeleton. We have achieved an acceptable parallel runtime behaviour with a low parallelisation effort. The most effective techniques to lower the communication overhead have been the use of direct mapping to avoid input communication and leaving the results in a distributed manner to avoid the result communication. When applicable, these techniques substantially improve the efficiency.

Acknowledgements. Thanks go to Thomas Horstmeyer for his comments.

References

1. J. Berthold, M. Dieterle, O. Lobachev, and R. Loogen. Distributed memory programming on many-cores – a case study using Eden divide-&-conquer skeletons. In *ARCS Workshop on Many–Cores*, pages 47–55, Delft, NL, 2009. VDE–Verlag.
2. J. Berthold and R. Loogen. Visualizing Parallel Functional Program Executions: Case Studies with the Eden Trace Viewer. In *Proc. of the Intl. Conf. ParCo 2007 – Parallel Computing: Architectures, Algorithms and Applications*. IOS Press, 2007.
3. M. I. Cole. Algorithmic skeletons: Structured management of parallel computation. In *Research Monographs in Parallel and Distributed Computing*. Pitman, 1989.
4. J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Math. Comput.*, 19:297–301, 1965.
5. P. Dmitruk, L. Wang, W. Matthaeus, R. Zhang, and D. Seckel. Scalable parallel fft for spectral simulations on a beowulf cluster. *Parallel Computing*, 27(14), 2001.
6. M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proc. of the IEEE*, 93(2), 2005.
7. S. Gorlatch. Programming with divide-and-conquer skeletons: A case study of FFT. *J. of Supercomputing*, pages 85–97, 1998.
8. S. Gorlatch and H. Bischof. A generic MPI implementation for a data-parallel skeleton: Formal derivation and application to FFT. *Par. Proc. Let.*, 8(4), 1998.
9. C. Grelck and S.-B. Scholz. Towards an efficient functional implementation of the nas benchmark ft. In *PaCT*, LNCS 2763, pages 230–235. Springer, 2003.
10. S. K. S. Gupta, C.-H. Huang, P. Sadayappan, and R. W. Johnson. Implementing fast Fourier transforms on distributed-memory multiprocessors using data redistributions. *Par. Proc. Let.*, 4(4):477–488, 1994.
11. C. A. Herrmann. *The Skeleton-Based Parallelization of Divide-and-Conquer Recursions*. PhD thesis, Universität Passau, 2000. ISBN 3-89722-556-5.
12. U. Klusik, R. Loogen, and S. Priebe. Controlling Parallelism and Data Distribution in Eden. In *TFP*, volume 2, pages 53–64. Intellect, 2000.
13. O. Lobachev and R. Loogen. Towards an Implementation of a Computer Algebra System in a Functional Language. In *Intelligent Computer Mathematics*, AISC, pages 141–154. Springer LNAI 5144, 2008.
14. R. Loogen, Y. Ortega-Mallén, R. Peña, S. Priebe, and F. Rubio. Parallelism Abstractions in Eden. In F. A. Rabhi and S. Gorlatch, editors, *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003.
15. R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel Functional Programming in Eden. *J. of Functional Programming*, 15(3):431–475, 2005.
16. H. Nussbaumer. *Fast Fourier Transform and Convolution Algorithms*. Springer, Berlin, 1981.
17. M. C. Pease. An adaptation of the fast Fourier transform for parallel processing. *JACM*, 15(2):252–264, April 1962.