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
Tel.: +49-6421-28-21525, Fax: +49-6421-28-25419

Abstract. The notion of Fast Fourier Transformation (FFT) describes a range of efficient algorithms to compute the discrete Fourier transformation, frequency distribution in a signal. FFT plays a major role both for pure mathematical applications and for real-life scenarios such as digital signal processing. The paper investigates and compares skeleton-based parallel Haskell implementations of different FFT-algorithms on workstation clusters with distributed memory. Our experiments show that the original divide-and-conquer versions suffer from an inherent input distribution and result collection problem, because huge amounts of data have to be communicated. Advanced approaches like distributable homomorphism FFT or multidimensional FFT provide more flexibility to overcome these problems. Assuming a distributed access to input data and re-organising computation in such a way that the results can be returned in a distributed way leads to versions with an acceptable parallel runtime behaviour.

Keywords. Algorithmic Skeletons, Parallel Functional Programming, Fast Fourier Transformation

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 [2] 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: besides the Cooley-Tukey, different r-radix, mixed-radix, and multidimensional versions [14].

In the broader context of an implementation for parallel computer algebra algorithms implemented in the parallel Haskell extension Eden [13, 11], we investigate parallelisation strategies for different FFT algorithms. In essence, FFT algorithms are based on divide & conquer strategies, which we implement using suitable algorithmic skeletons [1, 12]. In Eden, such skeletons are higher-order functions defining general parallel evaluation schemes. In this paper, we define skeletons for two variations of parallel divide-and-conquer evaluations: a *distributed expansion* scheme which unfolds the computation tree dynamically and spawns parallel processes for the evaluation of sub-trees as long as processor elements (PEs) 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. In addition, 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 & conquer or parallel map-and-transpose schemes. In this paper we focus on their application for the parallelisation of FFT algorithms. 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 PEs.

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

2 Divide & Conquer FFT

The classic 2*-radix FFT algorithm by Cooley and Tukey* works as follows. The input vector xs of length n (which we shall denote as $\overrightarrow{xs}_{0:n-1}$) is divided into two halves $\overrightarrow{ls} = \overrightarrow{xs}_{0:n/2-1}$ and $\overrightarrow{rs} = \overrightarrow{xs}_{n/2:n-1}$, to compute the element-wise sum ($\overrightarrow{ls}+\overrightarrow{rs}$) and the difference of the two ($\overrightarrow{ls}-\overrightarrow{rs}$). The latter is multiplied with powers $\overrightarrow{ws}$ 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.* The corresponding Haskell code is shown in Figure 1. Vectors are represented as lists. The Prelude function `splitAt :: Int → [a] → ([a],[a])` divides a list at the given position into two parts. The function `transpose :: [[a]] → [[a]]`, defined in the library module `Data.List`, takes a matrix as a list of rows and transforms it into a list of columns. Function `concat :: [[a]] → [a]` flattens a list of lists into a single list. Function `zipWith :: (a → b → c) → [a] → [b] → [c]` combines two lists element-wise using the given operation. It is used to implement basic vector arithmetic by applying standard arithmetic operations element-wise to lists.

```
fftDF     :: [Complex Double] -> [Complex Double]
fftDF [x] =  [x]
fftDF xs  =  shuffle [fftDF (ls @+ rs), fftDF ((ls @- rs)@* ws)]
            where (ls, rs) = splitAt (length xs ‘div‘ 2) xs
                  shuffle  = concat . transpose

(@+)       :: [Complex Double] -> [Complex Double] -> [Complex Double]
(@+) xs ys =  zipWith (+) xs ys
-- @-, @* are defined in the same way
-- ws is the list of powers of an n-th primitive root of unity
```

Fig. 1. A 2-radix FFT Implementation (Decimation in Frequency).

```
fftDT    :: [Complex Double] -> [Complex Double]
fftDT [x] = [x]
fftDT xs  = combine2 [fftDT ls, fftDT rs] where [ls, rs] = unshuffle 2 xs

unshuffle  :: Int -> [a] -> [[a]]
unshuffle n = transpose . (chunk n)

chunk     :: Int -> [a] -> [[a]]
chunk n [] = []
chunk n xs = ys : chunk n zs  where (ys,zs) = splitAt n xs

combine2        :: [[Complex Double]] -> [Complex Double]
combine2 [es,os] = (es@+ts) ++ (es@-ts) where ts = os@*ws
```

Fig. 2. A 2-radix FFT Implementation (Decimation in Time).

Decimation in Time. An alternative version, called *decimation in time*, essentially consists of the opposite dividing and combining steps. The input vector $\overrightarrow{xs}_{(0:n-1)}$ is split by unshuffling elements with even and odd indices into $\overrightarrow{ls} = x_0 : x_2 : x_4 : \ldots$ and $\overrightarrow{rs} = x_1 : x_3 : x_5 \ldots$ (inverse to the interleaving step above). After evaluating the recursive calls of FFT for $\overrightarrow{ls}$ and $\overrightarrow{rs}$, the combination of the result lists is now the more complex step. The first and second half of the overall result are defined as element-wise sums and differences including again a product with the *twiddle factors* $\overrightarrow{ws}$. The Haskell code is shown in Figure 2.

***r*-Radix FFT.** The *r-radix FFT* uses an analogous algorithm structure, but divides into $r = 2^k$ sub-tasks. The resulting task tree is wider. The input length has to be a power of r. Applicability of this approach might suffer from the fact that the distance between two powers of r grows with the value of r. If we change the value of r level-wise, we arrive at a hybrid mixed-radix algorithm.

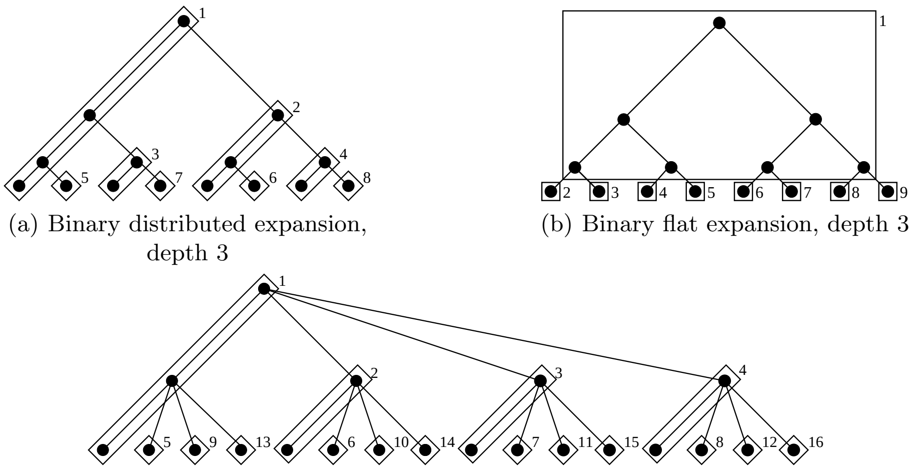
4-Radix FFT divides the input into four parts, where the k-th part begins at position $k\frac{n}{4}$ and k begins at zero. With decimation in time, combining the transformed sequences of length $\frac{n}{4}$ is slightly more complicated than before. First we need to multiply the k-th part with a list of powers of a twiddle factor ω^k, resulting in vectors $\overrightarrow{ys}_k$. The four parts of the output are defined as:

$$\widehat{\overrightarrow{xs}}_0 = (\overrightarrow{ys}_0 + \overrightarrow{ys}_2) + (\overrightarrow{ys}_1 + \overrightarrow{ys}_3) \qquad \widehat{\overrightarrow{xs}}_1 = (\overrightarrow{ys}_0 - \overrightarrow{ys}_2) - i(\overrightarrow{ys}_1 - \overrightarrow{ys}_3)$$
$$\widehat{\overrightarrow{xs}}_2 = (\overrightarrow{ys}_0 + \overrightarrow{ys}_2) - (\overrightarrow{ys}_1 + \overrightarrow{ys}_3) \qquad \widehat{\overrightarrow{xs}}_3 = (\overrightarrow{ys}_0 - \overrightarrow{ys}_2) + i(\overrightarrow{ys}_1 - \overrightarrow{ys}_3).$$

The final output vector is a simple concatenation of the $\overrightarrow{xs}_k$. Optimisation potential of this (more complicated) result combination lies in the repeatedly used sub-expressions computed only once. With decimation in frequency, the splitting step is the more complicated one. We omit details due to space limitations.

2.1 Regular Divide & Conquer Skeletons

The essence of a divide & conquer algorithm is to decide whether the input is trivial and, in this case, to solve it, or else to decompose non-trivial input



(a) Binary distributed expansion, depth 3

(b) Binary flat expansion, depth 3

(c) 4-ary distributed expansion, depth 2

Fig. 3. Divide & conquer expansion schemes.

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                    -- input / result
```

The resulting structure is a tree of task nodes where successor nodes are the sub-problems, the leaves representing trivial tasks. A fundamental Eden skeleton which specifies a general divide & conquer algorithm structure can be found in [12]. In this paper, we refine and adapt this skeleton for the FFT algorithm particularities. The r-radix divide & conquer-based FFT algorithms expose a regular tree structure, i.e. every non-trivial task is split into a *fixed* number of sub-tasks. Therefore, we focus on divide & conquer skeletons that evaluate a regular task tree with a fixed branching degree in parallel.

Two different basic strategies will be used to unfold a process tree. The first option is to create 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 PEs are available. This results in a *distributed expansion* of the computation (depicted in Fig. 3(a) for a binary scheme, and in Fig. 3(c) for a system with four successors per task node). Explicit placement of processes is essential to ensure that no two processes are placed on the same processor element while leaving others unused. The boxes indicate which tree nodes are evaluated by the same process. The numbers indicate a possible placement on 8 and 16 PEs, respectively.

In a second version, the main process unfolds the binary 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

```
dcN :: (Trans a, Trans b) =>
       Int -> [Int] ->    -- branch degree / tickets
       DivideConquer a b
dcN k tickets trivial solve split combine x
   | null tickets = seqDC x
   | trivial x    = solve x
   | otherwise
        = childRes `seq`     -- demand control: first start children
          rnf myRes `seq` rnf localIns `seq` -- then evaluate locally
          combine ( myRes:childRes ++ localRess )
   where
     -- sequential computation
     seqDC x = if trivial x then solve x else combine (map seqDC (split x))
     -- child process generation
     childRes   = spawnAt childTickets childProcs procIns
     childProcs = map (process . rec_dcN) theirTs
     rec_dcN ts = dcN k ts trivial solve split combine
     -- ticket distribution
     (childTickets, restTickets) = splitAt (k-1) tickets
     (myTs: theirTs)             = unshuffle k restTickets
     -- input splitting
     (myIn:theirIn)      = split x
     (procIns, localIns) = splitAt (length childTickets) theirIn
     -- local computations
     myRes     = ticketF myTs myIn
     localRess = map seqDC localIns

-- placed process instantiation (predefined Eden function)
spawnAt :: (Trans a, Trans b) => [Int] -> [Process a b] -> [a] -> [b]
spawnAt places f inputs = ... -- not shown
```

Fig. 4. Distributed expansion divide & conquer skeleton for k-ary task trees.

single source depicted in Fig. 3(b) for the binary variant. A uniform distribution of the subtrees on PEs can be achieved using a farm of worker processes with static or dynamic task distribution.

Distributed Expansion. Figure 4 shows a distributed expansion divide & conquer skeleton for k-ary task trees. Besides the standard parameter functions, the skeleton takes the branching degree, and a ticket list with PE numbers to place newly created processes. The left-most branch of the task tree is solved locally, other branches are instantiated using the Eden function `spawnAt`[1], which instantiates a collection of processes (given as a list) with respective input, on explicitly specified PEs. Results are combined by the `combine` function.

Explicit Placement via Tickets. The ticket list is used to control the placement of newly created processes. First, the PE numbers for placing the immediate child processes are taken from the ticket list. Then, the remaining tickets are distributed to the children in a round-robin manner using the function

[1] The type class `Trans` provides internally used communication functions.

```
fft2radixTime    :: Int -> [Complex Double] -> [Complex Double]
fft2radixTime cs xs
  = chunkDC cs chunkL concat
          (dcN 2 [2..noPe]) isSingleton  id (unshuffle 2) combine2

isSingleton     :: [a] -> Bool
isSingleton [_] =  True;  isSingleton xs  =  False
```

Fig. 5. Parallel 2-radix-FFT, decimation in time, with dcN skeleton.

`unshuffle :: Int → [a] → [[a]]` which unshuffles a given list into as many lists as the first parameter tells (see Figure 2 for a Haskell definition of this function). Child computations will be performed locally when no more tickets are available. The explicit process placement via ticket lists is a simple and flexible way to control the distribution of processes as well as the recursive unfolding of the task tree. If too few tickets are available, computations are performed locally. Duplicate tickets can be used to allocate several child processes on the same PE. The numbers in Figures 3(a) and 3(c) give the PE numbers when the ticket lists `[2..8]` and `[2..16]` are used for placement, respectively.

Demand Control. As Haskell's lazy evaluation would suppress any parallel evaluation, we need to add explicit demand for starting parallel child processes. In the `dcN` skeleton (lines 8 and 9 in Figure 4), we force the creation of the child processes before forcing all local computations. The strategy function `rnf :: NFData a ⇒ a → ()` forces the evaluation of its argument to normal form. The Haskell library function `seq :: a → b → b` evaluates its first argument before returning the second argument. The corresponding infix operator is denoted by `` `seq` ``. Finally, the combination of all results is done using the standard Haskell evaluation strategy.

Instantiation. The skeleton can be easily instantiated to compute different FFT algorithms in parallel. We have used it for 2-radix and 4-radix FFT, decimation in frequency and decimation in time. Figure 5 shows e. g. how the above skeleton can be used to compute the 2-radix FFT, decimation in time, in parallel. Note that list chunking has been used to reduce the communication overhead. Instead of communicating each list element in a single message[2], the lists are divided into chunks of a size `cs` (given as additional integer parameter). Chunking is simply implemented using a wrapper function `chunkDC` (not shown), which wraps all parameter functions with a pair of `unchunk`/`chunk` applications. We use the standard Haskell list function `concat` for unchunking and `chunk` defined in Figure 2 for chunking.

Flat Expansion. The flat expansion skeleton given in Figure 6 is a specialisation of the 'divide & conquer by replicated workers' skeleton (`dcrw`) in [12]. Our skeleton exploits the knowledge that the divide & conquer scheme has a fixed branching degree k. The task tree is unfolded up to a given depth d and each process selects one of the resulting sub-trees (with function `taskNr`). Instead of

[2] Eden communicates lists element-wise as streams

```
dcDM_N :: (Trans a,Trans b) =>
        Int -> Int ->     -- unfolding depth / branching degree
        DivideConquer a b
dcDM_N depth k trivial solve split combine x
  = results `seq` combineLevels k combine results
   where
     -- core computation
     seqDC x = if trivial x then solve x else combine (map seqDC (split x))
     -- child process generation
     results  = shuffle (farm noPe   -- standard task farm skeleton
                        (\ i -> seqDC (tasksNr (digits i d k) split x))
                        [0..k^d-1])

-- select the i-th task from tree with k^d tasks
tasksNr               :: [Int] -> (a -> [a]) -> a -> a
tasksNr []     split x =  x
tasksNr (i:is) split x =  tasksNr is split ((split x)!!i)

digits :: Int -> Int -> Int -> [Int]
digits i d k = reverse (computeDigits  i d k)
  where computeDigits i d k
          | d == 0    = []
          | otherwise = mod i k : (computeDigits (div i k) (d-1) k)

-- combine results level-wise
combineLevels ::  Int -> ([b] -> b) -> [b] -> b
combineLevels _ _       [x] = x
combineLevels k combine rs
 = combineLevels k combine (map combine (chunkL k rs))
```

Fig. 6. Flat expansion divide & conquer skeleton for k-ary task trees.

replicated workers which perform dynamic task distribution we use a simple farm with static task distribution. This is sufficient for regular parallelism, as in the case of FFT. The function `combineLevels` combines the results level-wise. It is important that the *unevaluated* task information is passed to the child processes which select their own parts of it. This *direct mapping* (DM) technique [10] saves the communication to distribute the input to the processes. As all processes are created by the same process, a simple round robin process placement is sufficient.

2.2 Experimental Results

The following run time 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. PEs), processes and threads, and message sending and receiving. The trace files can then be visualised by the EdenTV tool (Eden Trace Viewer). The resulting

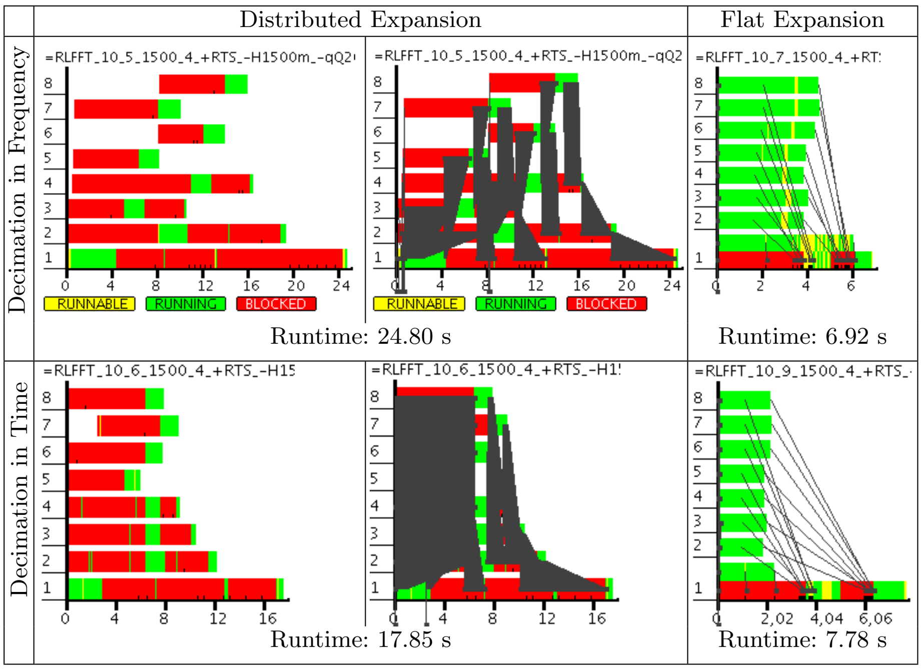


Fig. 7. Traces and runtimes of divide-and-conquer approaches, without/with messages.

graphics which are best viewed in colour are two-dimensional timeline 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 PE 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.

Distributed vs. Flat Expansion. 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 7 shows typical traces and the runtimes obtained with the following parameters: input size 2^{20} (double precision complex numbers), chunk size 1500, recursion depth 4 and heap size 1500MB. Note that the chunk size is only used by the distributed expansion skeleton to reduce the number of messages. Our experiments have shown that chunking is essential for the performance of the distributed expansion skeleton, but varying the chunk size did not have a great impact on the runtimes. Therefore, we have fixed the chunk size to 1500 in the runtime experiments discussed in the following. The recursion depth parameter is only used by the flat expansion scheme. It is not necessary for the distributed expansion scheme, because the latter adapts the recursion depth automatically to the number of available PEs.

The activity profiles in Figure 7 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 master process on machine 1 (lowest bars). The communication overhead is low. This reason is that the input lists are evaluated by the processes themselves. Only the indices of the sub-trees that have to be evaluated by the worker processes and their result lists are communicated, the latter without streaming. With depth 4, there are 16 sub-trees, i. e. 2 sub-trees per worker.

Decimation in frequency was the fastest version with 6.92 s, because the post processing in the master can be done very fast. Combining the results is a trivial shuffle. The top level combining phase of decimation in time takes almost three quarters of the overall runtime. The worker processes finish their evaluations already after 2 seconds while they need 4 seconds with decimation in frequency. Thus, in total less work is done with decimation in time.

With the flat expansion skeleton, we reduce the overhead for the initial communication, i. e. for distributing tasks to the worker processes, because each worker receives the whole task specification and selects its own part on demand. In a realistic setting, each worker would read its input data from files (which would either be distributed to all the nodes in advance, or be accessible via a network file system). What remains is the communication for result collection.

Work distribution is slower with the distributed expansion skeleton. The child processes are inhomogeneous and start working at different points in time. This is due to the communication of inputs to the sub-processes and of results to the parent process, as can be seen from the the activity profiles in Figure 7 which we show without and with messages as the message arrows cover most of the profile. The input distribution/result combination phases outweigh the benefits of the parallel execution. Although the input and output lists with $2^{20} = 1048576$ elements are divided into sub-lists (chunks) with 1500 elements, the total number of sent messages is 2146 in both versions. Note that intermediate lists are also communicated between the different tree levels.

For decimation in time, input division is simple, but result combination is more complex. As child processes can start working earlier and as Eden communicates lists as streams, partial input is already available to the child processes at an early stage. Therefore, decimation in time shows a slightly better runtime with the distributed expansion scheme than decimation in frequency.

2-Radix vs. 4-radix. In theory, the 4-radix algorithm should be faster than 2-radix, as it reduces the number of multiplications and enables to share some sub-results. In Figure 8, we show the runtimes of the four possible combinations of 4-radix in comparison to 2-radix, with the same input size $2^{20} = 4^{10}$. The shape of the trace files is very similar. Due to space limitations, we omit them here. The runtime measurements show that 4-radix behaves much better with

2-radix

	Distr. Exp.	Flat Exp.
Dec. in Freq.	24.80 s	6.92 s
Dec. in Time	17.85 s	7.78 s

4-radix

	Distr. Exp.	Flat Exp.
Dec. in Freq.	12.71 s	5.77 s
Dec. in Time	10.77 s	6.28 s

Fig. 8. Runtimes of 2-radix vs 4-radix on 8 PEs.

the distributed expansion scheme than 2-radix, but the runtimes are bad in comparison to the flat expansion scheme. When using 4-radix with the flat expansion scheme, we get a modest improvement of the overall runtime. The (non-shown) traces reveal that the worker times are almost halved when using 4-radix instead of 2-radix, but the postprocessing in the main process still dominates the overall runtime.

2-radix

	Distr. Exp.	Flat Exp.
Dec. in Freq.	23.08 s	10.68 s
Dec. in Time	18.66 s	8.01 s

4-radix

	Distr. Exp.	Flat Exp.
Dec. in Freq.	12.95 s	6.99 s
Dec. in Time	11,63 s	5,87 s

Fig. 9. Runtimes of 2-radix vs 4-radix on 4 PEs.

Scalability. Unfortunately, our experiments have shown that the different FFT parallelisations we have considered up to now do not scale well with respect to the number of PEs. Figure 9 shows the runtimes of the eight versions on 4 PEs. Comparing them with the runtimes on 8 PEs, speedups are low, in two cases we observe even a slowdown. Note that decimation in time is faster than decimation in frequency on 4 PEs.

3 Advanced Approaches

The parallel divide-and-conquer FFT-implementations show an acceptable performance using few PEs, but they do not scale well. Therefore we implement a more sophisticated algorithm taken from [5] which minimizes data dependencies and provides more fine grained parallelism. The input vector is divided into rows of a $d + 1$-dimensional grid with side lengths $l = 2^k$. In our implementation, we only consider 2-dimensional matrizes ($d = 1$), such that 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[3]
	– tagging input elements with their position and a "‘virtual global"’ length
	– split into rows
2. central processing:	local fft3 $\circ$ global transpose $\circ$ local fft3
3. postprocessing:	shuffle (concat $\circ$ transpose) $\circ$ remove tags

The key difference between the ordinary sequential FFT and `fft3` is that the latter operates on triples comprising the data item, a position tag and the number of already performed combine steps. It works with *global* twiddle factors to

[3] Elements at position i are moved to position ‘bits(i) reversed’.

```
parMapTranspose :: (Trans a, Trans b) =>
                   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)
```

Fig. 10. Parallel map-and-transpose skeleton.

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 first the matrix rows and then its columns. For more details, see [5].

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 distributed transpose. The skeleton has been inspired by the distributable homomorphism skeleton of Gorlatch and Bischof [6]. It can also be used for the distributed-memory FFT algorithms proposed in [15, 8].

3.1 A Parallel Map-and-Transpose Skeleton

The skeleton implements the functionality

`(parMap f1)` $\circ$ `transpose` $\circ$ `(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 overhead. Our skeleton `parMapTranspose` includes a distributed transpose phase in between two parallel map calls. The skeleton's input is a matrix which will be distributed row cyclic. In our application the functions f_i will be sequential `fft3` invocations. To save communication costs, we again use direct mapping [10] to implement the parMaps. This means here 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 in a demand driven way. In our FFT case study, the matrix generation and the preprocessing will be done in this way. The Eden code of the parallel map-and-transpose skeleton is shown in Figure 10. It makes use of Eden's capability to dynamically define new input channels for processes. The Eden function `createChans :: Int` $\rightarrow$ `([ChanName a], [a])` creates a list of new (input) channel names. Data (lazily) received via the channel names 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 :: Trans a ⇒ [ChanName a] → [a] → b → b`. It concurrently passes data via given channel names and returns its third argument.

The distributed `map` functionality is easy to define. Let `np` be the number of available PEs. We divide the matrix rows into `np` contiguous blocks using the function `unshuffle` (defined in Fig. 2). At the end the final result is recomposed using the inverse function `shuffle`. As many processes as available PEs are created using the Eden function `spawn`[4]. 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[5]. Each process evaluates the function `distr2D_fs` which initially leads to the creation of `np` input channels `myChanNs` for the corresponding process. The channel names 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 now occur without sending data through the parent process.

After the first `map f1` evaluation, a process locally unshuffles the columns of the result (the locally transposed result rows) into `np` matrices. 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.

3.2 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 two-dimensional instance of the FFT algorithm developed by Gorlatch and Bischof [5] using our map-and-transpose skeleton. Result collection and post processing (a simple shuffle) can be omitted leaving the result matrix in a distributed column-wise manner. A runtime trace, again for input size 4^{10}, is depicted in Figure 11. 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

[4] `spawn` is the same as `spawnAt` with a default round-robin placement of processes.

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

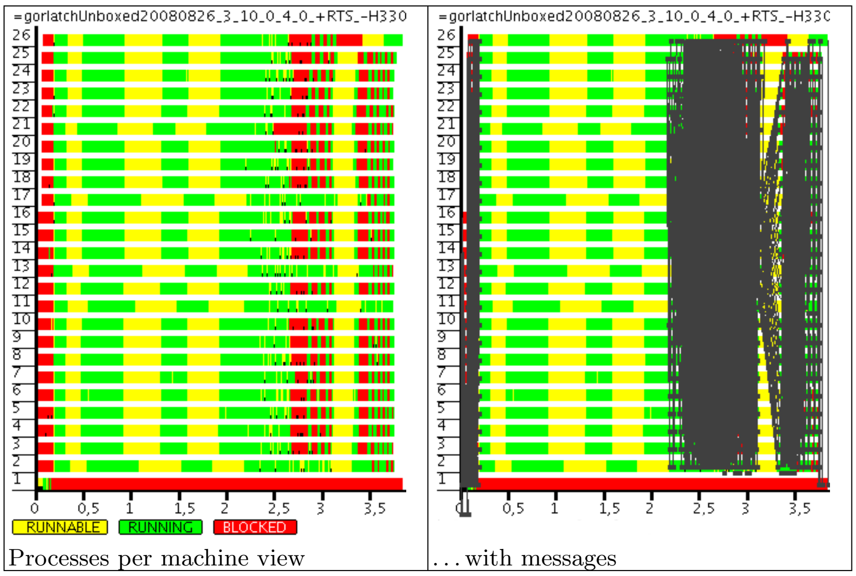


Fig. 11. Trace of parallel FFT using map-and-transpose skeleton .

twiddle-factors. Noticeable are also the frequent "'runnable"' phases, which are garbage collections provoked by the increased memory needs of this FFT-version.

Figure 12 shows the runtimes of the parallel map-and-transpose FFT version with and without final result collection (Figure 12, triangle marks) in comparison with the best divide-and-conquer versions (4-radix, 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 distributed-result version scales 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. A reason for this is its fine-grain parallelism with 2^{10} tasks which is opposite to the coarse-grain parallelism with 16 and 64 tasks, respectively, of the other versions. Moreover, including result collection in the map-and-transpose version substantially decreases its performance. The runtime differences of the various versions are less clear 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.

Multidimensional FFT. We used the above parallel map & transpose skeleton also to implement multidimensional FFT. Multidimensional FFT represents a one-dimensional discrete FFT with input length $n \cdot m$ as a two-dimensional discrete FFT of size $n \times m$. There are many different ways to perform a multidimensional FFT. The simplest case for two dimensions is the *row-column* method. In this case, a one-dimensional FFT is applied to the columns and then

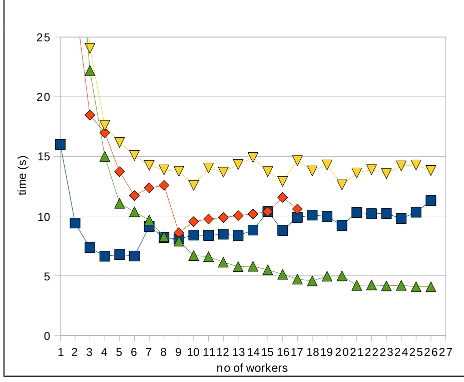
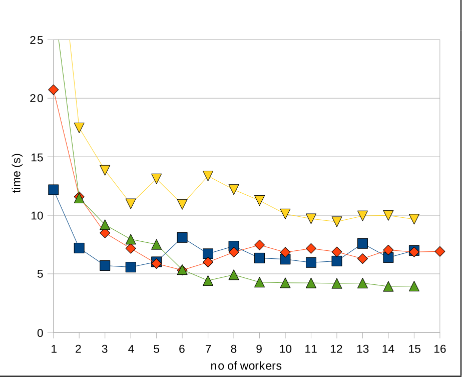
on 27 Pentium 4 CPU's @ 3.00 GHz, 512 MB RAM, fast Ethernet	on 8 Core 2 Duo CPU's @ 2.40 GHz, 2 GB RAM, fast Ethernet
■ flat expansion, radix 4, dec. in time, Depth 3 (64 tasks) ◆ flat expansion, radix 4, dec. in freq., Depth 2 (16 tasks)	▽ map and transpose, radix 2, dec in time, (2^10 tasks), gathered result ▲ map and transpose, radix 2, dec in time, (2^10 tasks), distributed result
	

Fig. 12. Runtime and scalability comparison of parallel FFT approaches.

to the rows of the two dimensional matrix. Our experiments with multidimensional FFT have shown that our skeleton scales equally well and shows a similar runtime behaviour as the instantiation discussed above.

4 Related Work

A range of parallel FFT implementations have been presented in the past ([4, 3], to mention only a few). The vast majority is tailored for shared-memory systems, see e.g. [7] as an example for a high-level implementation in the functional array language SAC. Distributed implementations are mostly based on C+MPI. The distributed MPI-based FFTW, the Fastest Fourier Transform in the West, implementation [4] is especially tailored for transforming arrays so large that they do not fit into the memory of a single PE. In contrast to these specialised approaches, our work propagates a skeleton-based parallelisation. In his PhD thesis [9], Christoph Herrmann gives a broad overview, classification, and a vast amount of implementation variants for divide & conquer, while we have focused on divide-and-conquer schemes with a fixed branching degree. The skeleton-based version of parallel FFT in [6, 5] underlies our parallel map-and-transpose implementation of one-dimensional FFT.

5 Conclusions

Developing an efficient parallel distributed-memory implementation of FFT is a great challenge. The manual of the recent 3.2 alpha release of FFTW[6] warns that "distributed-memory parallelism can easily pose an unacceptably high communications overhead for small problems". This inherent problem of distributed-memory parallel FFT made our investigation difficult. The goal of our work has

[6] `http://www.fftw.org/fftw-3.2alpha3-doc/`

not been to develop the fastest distributed-memory FFT, but to investigate a skeleton-based parallelisation of FFT. The skeleton approach cleanly separates algorithmic (problem-related) and coordinational (problem-independent) issues. The high communication overhead of FFT is clearly problem-related. The skeleton approach provides a high flexibility. In total, six different parallel FFT approaches have been compared, on the basis of three new skeletons: two parallel divide-and-conquer and a parallel map-and-transpose skeleton. The skeletons are widely applicable. They exploit useful general coordination techniques, in particular a flexible ticket mechanism to control the unfolding of a parallel process system and the direct mapping technique to avoid input communication. We have achieved an acceptable parallel runtime behaviour with a low parallelisation effort. The communication overhead could additionally be lowered by leaving the results in a distributed manner to avoid the result communication.

Acknowledgements: We thank Thomas Horstmeyer for supporting our work.

References

1. M. I. Cole. Algorithmic skeletons: Structured management of parallel computation. In *Research Monographs in Parallel and Distributed Computing*. Pitman, 1989.
2. J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Math. Comput.*, 19:297–301, 1965.
3. 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.
4. M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proc. of the IEEE*, 93(2), 2005.
5. S. Gorlatch. Programming with divide-and-conquer skeletons: A case study of FFT. *J. of Supercomputing*, 12(1-2):85–97, 1998.
6. 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.
7. 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.
8. 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.
9. C. A. Herrmann. *The Skeleton-Based Parallelization of Divide-and-Conquer Recursions.* PhD thesis, Universität Passau, 2000. ISBN 3-89722-556-5.
10. U. Klusik, R. Loogen, and S. Priebe. Controlling Parallelism and Data Distribution in Eden. In *TFP*, volume 2, pages 53–64. Intellect, 2000.
11. 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.
12. 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*, pages 71–88. Springer, 2003.
13. R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel Functional Programming in Eden. *J. of Functional Programming*, 15(3):431–475, 2005.
14. H. Nussbaumer. *Fast Fourier Transform and Convolution Algorithms.* Springer, Berlin, 1981.
15. M. C. Pease. An adaptation of the fast fourier transform for parallel processing. *JACM*, 15(2):252–264, April 1962.