# A generalized FFT algorithm on transputers*

Herman Roebbers
*University of Twente, EL/BSC Dept.,*
*P.O.Box 217, 7500 AE Enschede, The Netherlands*
Peter Welch
*University of Kent at Canterbury, Computing Laboratory,*
*Canterbury, Kent, England, CT2 7NF*
Klaas Wijbrans**
*Van Rietschoten & Houwens, Rotterdam, The Netherlands*

**Abstract.** A generalized algorithm has been derived for the execution of the Cooley-Tukey FFT algorithm on a distributed memory machine. This algorithm is based on an approach that combines a large number of butterfly operations into one large process per processor. The performance can be predicted from theory. The actual algorithm has been implemented on a transputer array, and the performance of the implementation has been measured for various sizes of the complex input vector. It is shown that the algorithm scales linearly with the number of transputers and the problem size.

## 1 Introduction

A commonly used algorithm in scientific engineering is the Fast Fourier Transform[1]. In various fields, such as control theory, system identification, coding theory and signal processing, the Fast Fourier Transform is a valuable tool. Therefore, a lot of effort has been spent in the past in finding efficient implementations of this algorithm.

Most of the implementations in the past made use of fast sequential processors, specialized hardware or dedicated signal processors. The algorithms have been optimized for these kinds of hardware. Some implementations assumed the use of multi-processor shared-memory machines[2].

In this paper, the parallel implementation of the FFT on a distributed-memory machine is considered. With the advent of the transputer, this type of machine has become very cost-effective. Compared to the use of dedicated hardware or signal processors, transputer machines offer greater flexibility and a good cost-performance ratio. Furthermore, a parallel high-level language is provided for the programming of these machines.

The basic element of the FFT is a number of operations on complex data, called the butterfly[4](figure )1. The implementation discussed uses the Decimation In Frequency method. This means that every butterfly consists of one complex addition, one complex subtraction and one complex multiplication. In this case, each butterfly in effect executes a two-point FFT. By combining the butterfly operations in a suitable manner, a $2^N$ point FFT is created.

The butterfly representation of the FFT algorithm[4] (figure 2) is an elegant representation, showing the data-flow and the operations on the data in a graphical manner. This representation is mainly used to determine the order in which the computations have to be performed in a sequential machine. The parallelism that is inherent in the butterfly representation of the FFT is not used in that case.

* This work was performed during the development of material for the COMETT-course on advanced transputer engineering. It has been supported by the ESPRIT Parallel Computing Action proj. nr. 4122

** Working under a research grant from Van Rietschoten & Houwens at the University of Twente, EL/BSC Dept.

On a parallel machine, the butterfly representation can also be used as a process graph. One can look at the butterflies as processes, and at the lines indicating the data-flow in the system as communication channels. In this way an $N$-point FFT algorithm will be decomposed automatically in $\frac{1}{2}N \log_2 N$ parallel processes. To implement this system, only a single butterfly process has to be coded, and the butterfly processes have to be connected together. Walker[3] and Eckelman [5] have used this approach to obtain a (theoretical) indication for the speed-up that can be obtained on a transputer system consisting of T424 transputers.

Section 2 discusses the parallel implementation of the FFT algorithm and it is shown how a universal parallel algorithm can be derived that can be used on different types of message-passing MIMD machines. In section 3 the implementation on a transputer system is given, together with the measurement methods that have been used to determine the performance of the parallel FFT (section 3.2). The results of the measurements for various configurations are given in section 4. The conclusions are in section 5.

## 2 A generalized parallel FFT algorithm

The Fast Fourier Transform is based on the computation of the Fourier Transform by convolving the input data with a number of different frequencies. With a finite number of input samples, the Fourier transform formula can be reduced to its discrete equivalent:

$$F_n = \sum_{k=0}^{N} x_k e^{-\frac{j2\pi kn}{N}}$$

This formula is called the Discrete Fourier Transform (DFT)[1]. The Fast Fourier Transform (FFT) is a special case of the Discrete Fourier Transform. For the FFT, $N$ is chosen as a power of two. Because terms cancel each other during the computation, the total number of computations for the Fourier transform can be reduced to $\frac{1}{2}N \log_2 N$ instead of $N^2$. There is a price for this decrease in the number of computations: some shuffling of the data or complex addressing is necessary to access the data in the correct order.

### *2.1 The butterfly representation*

The butterfly representation is an elegant way of representing the operations on the data and the shuffling of the data. In the butterfly representation of the Fast Fourier Transform, the operations are shown as blocks, and the lines connecting the butterflies represent the data-flow between the blocks. This representation can be used to determine the memory access pattern of the FFT on a sequential machine, or as a process graph on a parallel machine. The shuffling is then performed implicitly by the way the butterfly processes are connected.

Each butterfly has two inputs and two outputs (figure 1). The values at the inputs are called $a$ and $b$, the values at the outputs are called $x$ and $y$. $W$ is the weight factor, and is different for each butterfly. The $x$ and $y$ values are computed according to the equations in figure 1.

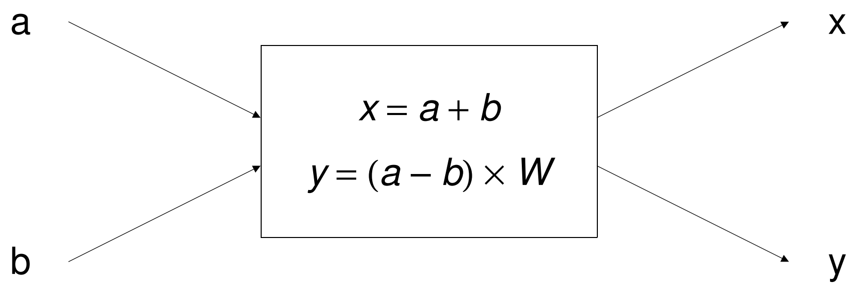


*Figure 1: butterfly operation*

On a parallel machine, each butterfly can be implemented as a parallel process, communicating with the other butterfly processes using communication channels. The shuffling in the parallel implementation is performed implicitly by the way the processes are connected. This is implemented by numbering the processes according to their row and column. The channels are elements of the two-dimensional array $c$. This is shown in figure 2 for an 16-point FFT. The general formulas below have been derived for this interconnection scheme.

$$a_{col,row} = c_{col,row}$$
$$b_{col,row} = c_{col,row + \frac{1}{2}N}$$
$$x_{col,row} = c_{col+1, 2\times row}$$
$$y_{col,row} = c_{col+1, 2\times row+1}$$
$$W_{col,row} = e^{-\frac{2\pi j\,(row\ \&\ (-1 << col))}{N}}$$

& is the symbol for the binary and operation, << is the shift-left operator.

### 2.2 Reducing the local parallelism

The parallel approach has some disadvantages. If all butterfly processes have to be computed in parallel, many processes are needed. However, this either means that a lot of butterfly processes will be running on a single processor, or that many processors have to be used. The internal parallelism and the internal communication cause a lot of overhead. By combining the butterfly processes running on a single processor to a single sequential process the execution can be accelerated. In the remainder of this paper, this sequential process is referred to as the *generalized butterfly*.

Figures 3 and 4 show two different ways of dividing a 16-point FFT over different processor configurations. There are two important parameters for the subdivision of an FFT over a number of processors:

- The number of processors per row (*proc.per.row*)
- The number of processors per column (*proc.per.column*)

The variable *proc.per.row* determines the number of columns that must be executed on each processor. In this paper it is assumed that *proc.per.row* is a divisor of the total number of columns $\log_2 N$.

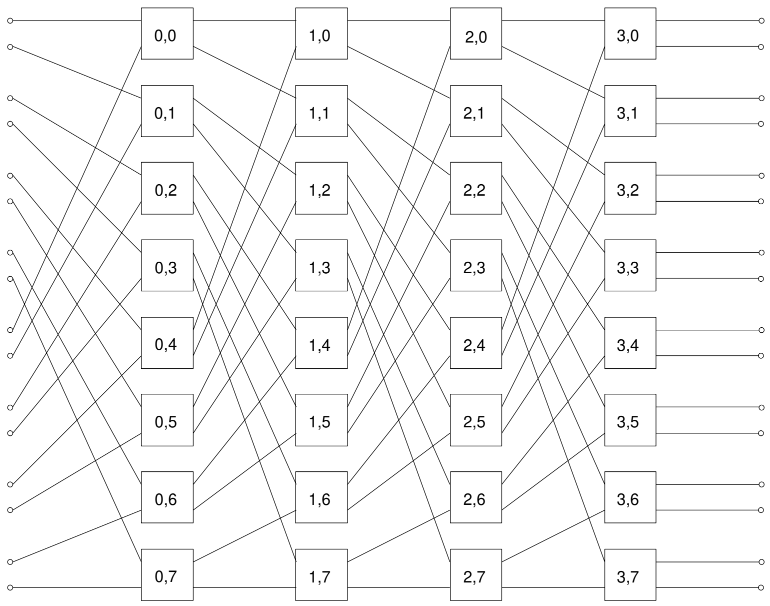


*figure 2: interconnections for 16-point FFT*

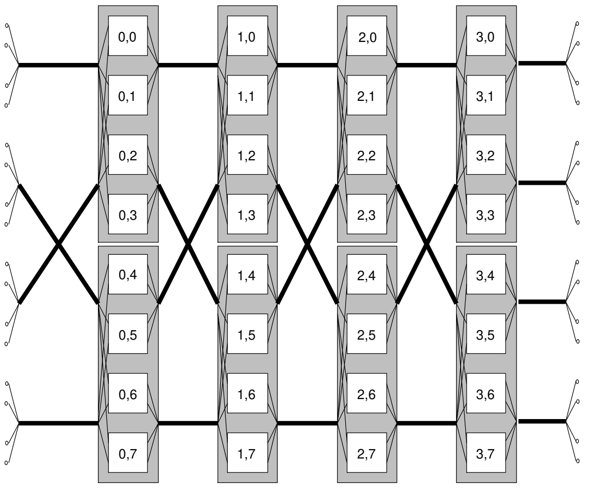


*Figure 3: subdivision over 2 rows and 4 columns*

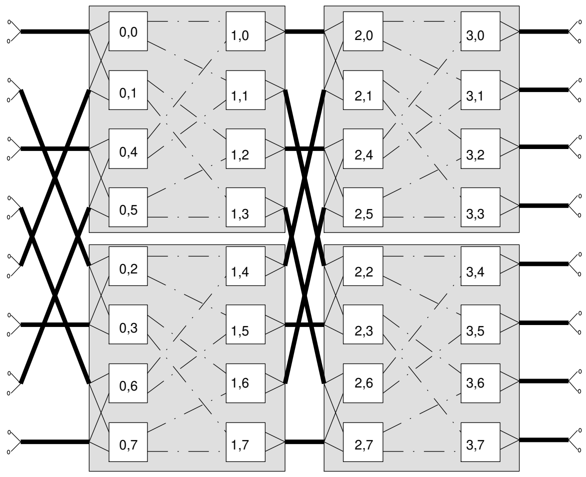


*Figure 4: subdivision over 2 rows and 2 columns*

The variable *proc.per.column* determines the number of butterfly operations that must be executed on each processor for each local column. It is assumed that *proc.per.column* is a divisor of $\frac{1}{2}N$. There are two levels in combining the butterflies:

- The combination of butterflies within a (partial) column into one (partial) column process.
- The combination of (partial) column processes into one large process.

*Combining butterflies into a (partial) column process*

The different butterfly processes in a single column can be combined by executing them sequentially. The combined *a, b, x* and *y* channels receive arrays of $\dfrac{N}{2 \times proc.per.column}$ values. This is shown in figure 3 by the thick lines. Internally, the butterflies are performed on arrays in memory. The shuffling is performed implicitly by using an appropriate addressing scheme. The basic algorithm for the butterfly computation now becomes:

- Receive the input arrays from the previous (partial) column(s)
- Perform the butterfly operation on all corresponding elements of the input arrays
- Send the output arrays to the next (partial) column(s)

By using a double buffering scheme, communication is overlapped with computation. This results in a sequential implementation of the (partial) column process.

The combined butterfly process has the same structure as the original butterfly processes. The main difference is that the number of communications channels is reduced from $N$ to $2 \times proc.per.column$. The amount of data communicated over a channel increases from a single complex number to $\dfrac{N}{2 \times proc.per.column}$ complex numbers. This will result in a decrease of the communication overhead.

*Combining the column processes on a processor*

The partial columns can be combined to the generalized butterfly process. If more than one (partial) column is computed on a single processor, these computations are performed sequentially. This is done by repeatedly executing the butterfly operation on a (partial) column. After the computation of a (partial) column, the output data of this column is used as input data for the next column. However, this requires that for each local column no intermediate results are needed from other processors. This is done by shuffling the butterflies in the FFT (figure 4).

The $m$ local columns in the generalized butterfly process are combined into a $2^m$-point FFT. The processes in each column have to correspond to the processes that are needed to compute this $2^m$-point FFT. This means that the butterflies in a column are no longer numbered consecutively. A bit-permutation algorithm is used to determine the sequence numbers of the butterfly operations in a generalized FFT. This algorithm is outlined below:

First, a rotation operator is defined. In the standard FFT algorithm a bit swap operator is used to determine the weight factors of the butterfly operations. This bit swap operator is generalized to a rotation operator Rot($n$,$m$). Rot($n$,$m$) rotates the bits in a word from position $n$ over $m$ bits. The most significant bit resides at position 0. The behaviour of this operator is shown below.

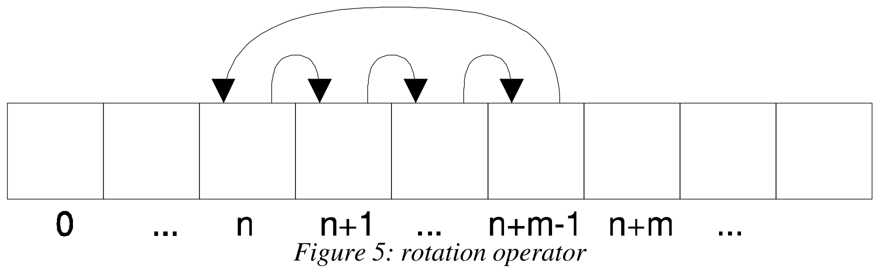


*Figure 5: rotation operator*

The rotation operator is used to determine the sequence number of each butterfly in each local column on a processor. This sequence number is used instead of the row number in the weight formula of section 2.1.

$$W_{col,row} = e^{-\frac{2\pi j\,(sequence_{lcol,row}\ \&\ (-1<<col))}{N}}$$

The sequence number is computed with the following algorithm:

- the sequence number of a butterfly operation in the last column on each processor is equal to the row number of the butterfly operation:
  $sequence_{lcol,row} = row$
- the sequence number at column *lcol* is computed by applying the rotate operator on the sequence number at column *lcol*+1:
  $sequence_{lcol,row} = \mathrm{Rot}(col.per.proc - lcol,\ \log_2 proc.per.column)\ sequence_{lcol+1,row}$

*col.per.proc* is the number of columns per processor. It is equal to $\dfrac{\log_2 N}{proc.per.row}$

*lcol* is the local column number. *lcol* is numbered from 1 to *col.per.proc*

The structure of the generalized butterfly is shown in figure 6. The $a$ and $b$ arrays are received over $2^m$ channels. A single output array is created, that contains the $x$ and $y$ data. This

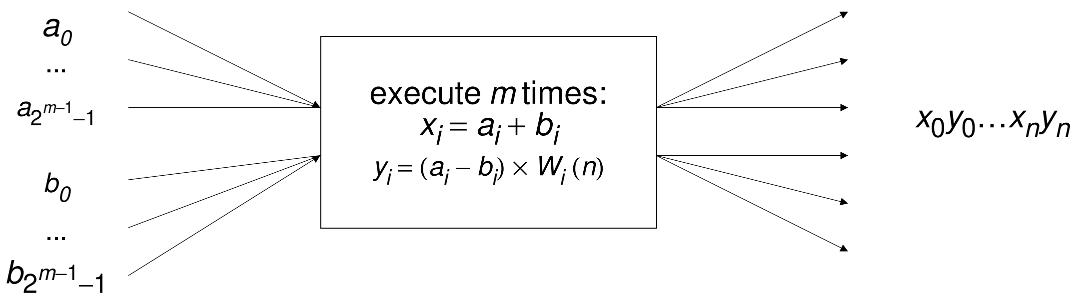


*Figure 6: generalized butterfly operation*

output array is transmitted to the next stage over $2^m$ channels.

*Interconnecting the generalized butterflies*

In the previous section the ordering of the butterfly operations over the processors has been described. The other important algorithm for computing the FFT determines the interconnection structure of the FFT algorithm. The interconnection structure changes if columns are combined. If $m$ columns are combined in a generalized butterfly process, this results in $2^m$ different output channels, each transmitting $\frac{N}{proc.per.column \times 2^m}$ complex numbers.

The interconnection structure is created using the following algorithm:

- Subdivide the output array of the last column of the transmitting stage on each processor into $2^m$ parts. The total output array is then subdivided into $2^m \times$ *proc.per.column* parts. These parts are transmitted on the output channels $o_{0 \ldots 2^m \times proc.per.column - 1}$. Each processor has $2^m$ output channels.
- The processors in the receiving stage each have $2^{m-1}$ *a*-type channels and *b*-type channels. These channels are numbered consecutively from 0 to $2^{m-1}$.
- The channels are connected according to the following formulas:

$$a_i = o_{(2 \times i) \bmod 2^m}$$

$$b_i = o_{(2 \times i + 1) \bmod 2^m}$$

Two examples of the result of this way of interconnecting the generalized butterflies are given in figure 7 and figure 8. The small squares represent the butterfly operations. The numbers inside these squares are the sequence numbers of the butterfly operations. These numbers determine the weight factors.

If the number of processors is less than the number of outputs per processor, several outputs can be multiplexed on the same communication channel. This is especially important if the

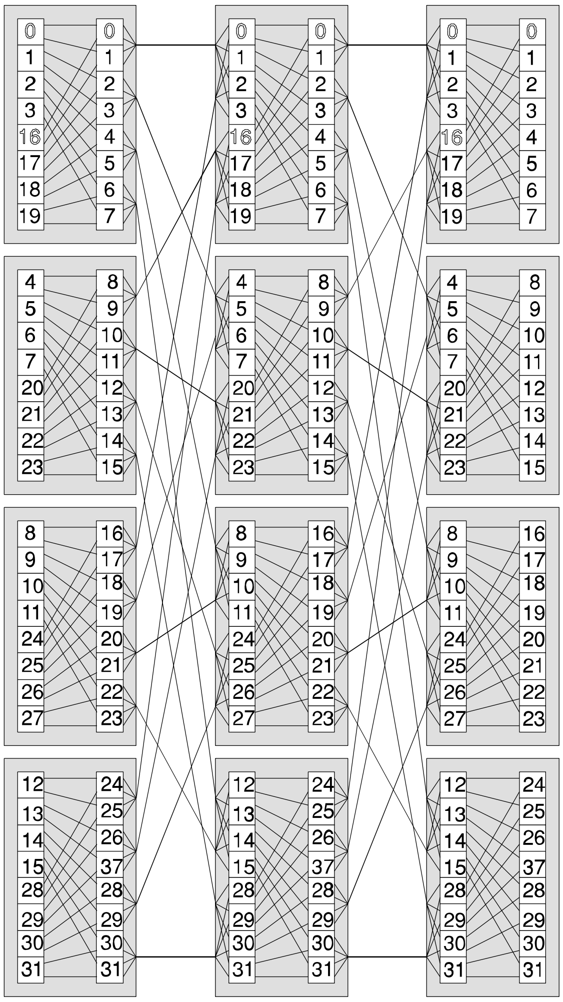

*Figure 7: 64-point FFT on 4 x 3 processor array*

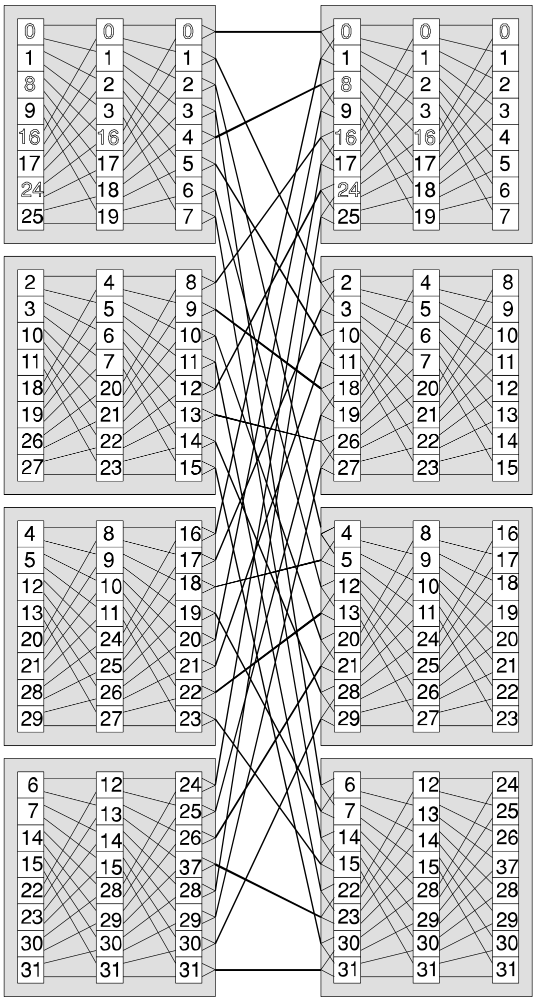

*Figure 8: 64-point FFT on 4 x 2 processor array*

number of communication channels per processor is restricted to a limited number. The outputs that go to the same processor are found using the condition below:

$o_i$ is mapped onto the same channel as $o_{i+proc.per.column}$ if

$$\mathrm{Entier}\left(\frac{i}{2^m}\right) = \mathrm{Entier}\left(\frac{i+proc.per.column}{2^m}\right)$$

## 3 Implementation and measurement method

This section describes the implementation of the algorithm on a transputer system. The fully parallel and the generalized butterfly FFT were implemented on a Meiko system with 48 T800 processors at 25 MHz. The programming language used was Occam[6], because Occam is very efficient and provides good insight in the parallelism. During the implementation, special attention was given to a proper use of the facilities provided by the Occam compiler and the Occam language, resulting in a maximal performance speed-up[7].

Measurements were performed on both implementations, resulting in figures for the overhead and the speed-up. These measurements were performed for different sizes of the FFT.

### *3.1 The FFT generalized butterfly on the transputer*

The generalized butterfly element in principle consists of three parallel processes:

- Receive the input data in input buffer set 1.
- Send the output data from output buffer set 1.
- Operate on input buffer set 2 to create output buffer set 2.

These operations are executed repeatedly on both buffer sets, thus overlapping computation with communication. On a transputer this will result in a large performance benefit. To start the pipeline, some start-up code is required to fill the pipeline. After that, an endless loop is executed, containing the butterfly operation and the sending and receiving of the input and output data. The number of butterflies `lrow` that is computed on this processor is equal to $\frac{1}{2}\frac{N}{proc.per.column}$. This is also the size of the input data vectors. The variable `lcol` is the number of times a butterfly operation has to be executed on the input data.

The basic element of the FFT program is the enhanced butterfly procedure. The unoptimized code for this procedure is shown in figure 10. It is an implementation of an `lrow`×`lcol` butterfly operation.The input and output arrays are contained in the array `c`.The butterfly

```
PROC gen.bfly(...)
  [lcol][lrow][2]REAL32 c1, c2:
  SEQ   -- start up pipeline
    receive(..., c1[0])
    PAR
      receive(..., c2[0])
      butterflies(lcol, c1)
    WHILE TRUE -- pipeline has started
      SEQ
        PAR
          send(c1[lcol], ...)
          receive(..., c1[0])
          butterflies(c2)
        PAR
          send(c2[lcol], ...)
          receive(..., c2[0])
          butterflies(c1)
:
```

*Figure 9: structure of Occam implementation*

```
PROC butterflies([][][2]REAL32 c, w)  -- w is the array with weights
  SEQ i=0 FOR lcol
    SEQ j=0 FOR lrow
      butterfly(c[i][j],
                c[i][lrow+j],
                c[i+1][2*j],
                c[i+1][(2*j)+1],
                 w[i][j])
:
```

*Figure 10: butterfly processing*

procedure operates on these arrays. To do this, the input array is split into two separate arrays. The procedure with the name butterfly performs the butterfly operation as given in section 2.1.

*Weight factors*
The weight factors are computed in advance from the sequence numbers of the butterfly operations and stored in a table. The weight factors table consists of $\frac{1}{2}N \times 2$ real variables per column. This is $\frac{1}{2}N \times 2 \times 4 = 4N$ bytes for a single precision FFT.

*Optimization techniques*
The butterfly implementation shown in figure 10 is not very efficient due to the double indexing of the arrays and the loop overhead. Therefore, an optimized implementation has been made of the basic butterfly procedure. This implementation improves the performance by the use of abbreviations and loop unrolling. The optimization techniques are treated extensively in [7].

*Interconnection problems*
Due to the limited number of links on a transputer, not all possible configurations can be built. There are no problems if the number of columns per processor is 1, or if there are no more than two processors per column. In other cases, some extra shuffling of the butterfly processes is necessary. This extra shuffling results in a more irregular structure.

### *3.2 Measurement method*

The results have been obtained from measurements on the Meiko machine. The test program for the FFT consists of three parts (figure 11):

- A data source, pushing data into the network.
- A data-collector, receiving data from the network.
- The FFT program.

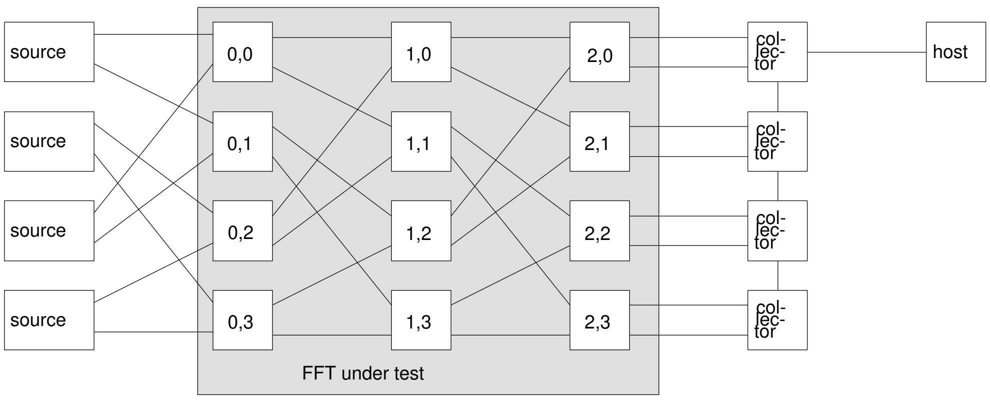


*Figure 11: measurement method*

The measurements have been performed by measuring the time between successive results arriving at the data-collector program.The correctness of the FFT program has been tested by executing the FFT operation on a test vector and comparing the results with the predicted results, and by performing an FFT on a set of data, and an inverse FFT on the result of the FFT. This operation resulted in a reconstruction of the input signal.

## 4 Results

To show the performance of the FFT program, several measurements have been performed.Because FFT programs for different sizes differ widely in the total computation time, the measurements have been scaled with respect to the number of butterfly operations performed. Two different measurements have been performed.

*The time needed per butterfly operation*

The time needed for the computation of an FFT is determined by the time that is needed for a single butterfly operation. Therefore, first the time needed for performing a butterfly operation was measured. On a sequential machine, the time needed for the computation of the total FFT can then be determined with the following formula:

$$totaltime = time\ per\ butterfly \times \tfrac{1}{2} N \log_2 N$$

The time per butterfly has been measured by executing an FFT for different sizes on a single processor. Two different cases can be distinguished:

- the time measured without communication taking place
- the time measured with communication taking place

The results of these measurements are shown in figure 12. The horizontal axis gives the number of butterfly operations on a single processor. The vertical axis shows the time needed per butterfly operation. Four different graphs are shown:

a the fully parallel FFT (section 2.1) without link communication

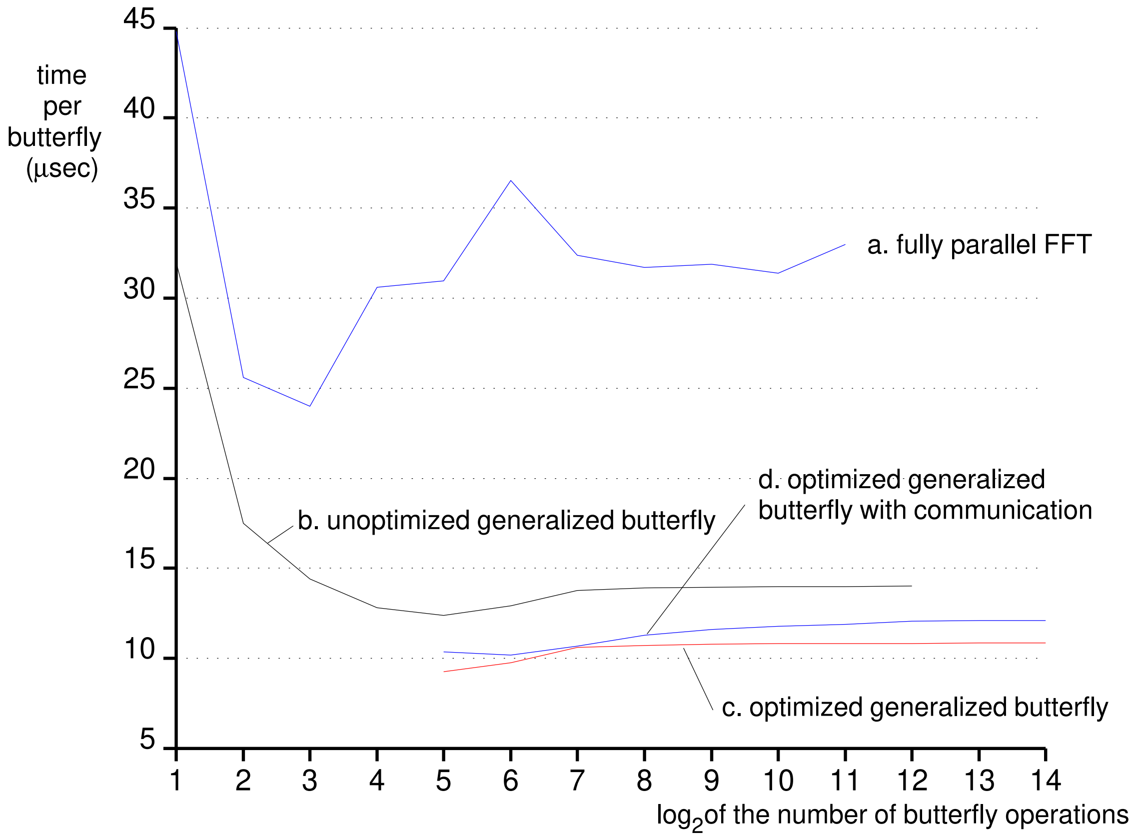


*Figure 12: time per butterfly operation*

b the unoptimized generalized butterfly FFT(section 2.2) without link communication
c the optimized generalized butterfly FFT without link communication
d the optimized generalized butterfly FFT with link communication

From these results it is concluded that the time needed to compute a butterfly is almost constant if $N > 2^7$. The bumps in the graphs occur because more data is placed in the external memory.The graphs show that the optimized generalized butterfly performs significantly better than a fully parallel FFT or an unoptimized generalized butterfly implementation.

The communication causes a fixed overhead of approximately 10%. This means that the time needed per butterfly operation for the multiprocessor generalized butterfly implementation is fixed, and only 10% more than the single processor implementation. Therefore the generalized butterfly FFT is well scaleable. The time needed for executing a generalized butterfly FFT on an array with *proc.per.column* $\times$ *proc.per.row* processors is equal to

$$totaltime = \frac{time\ per\ butterfly\ while\ communicating \times \frac{1}{2} N \log_2 N}{proc.per.column \times proc.per.row}$$

For sufficiently large FFTs the efficiency is equal to

$$efficiency = \frac{time\ per\ butterfly\ without\ communication}{time\ per\ butterfly\ with\ communication} \approx 90\%.$$

### *4.1 Performance as a function of the number of processors*

The second measurement determines the speed-up of the FFT for different processor configurations. The speed-up is defined as:

$$speedup = \frac{time\ needed\ on\ one\ processor}{time\ needed\ on\ n\ processors}$$

Also the efficiency is computed. The efficiency is defined as:

$$efficiency = \frac{speedup}{n}$$

These results have been measured for a 256-point, a 1K and a 4K FFT for the optimized generalized butterfly, and for a 1K FFT for the fully parallel FFT. The speed-up and the efficiency have been measured. The speed-up as a function of the number of processors is shown in figure 13. The efficiency for the different configurations is shown in the table below.

| | optimized generalized butterfly FFT | | | | | | fully parallel FFT |
|---|---|---|---|---|---|---|---|
| #transputers | 256 points, 1 transp. per column | 256 points, 2 transp. per column | 1K points, 1 transp. per column | 1K points, 2 transp. per column | 4K points, 1 transp. per column | 4K points, 2 transp. per column | 1 K points, 1 transp. per column |
| 1 | 1.0 | | 1.0 | | 1.0 | | 1.0 |
| 2 | 0.996 | | 0.991 | | 0.988 | | 0.852 |
| 3 | | | | | 0.937 | | |
| 4 | 0.952 | 0.929 | | 0.939 | 0.933 | 0.933 | |
| 5 | | | 0.937 | | | | 0.699 |
| 6 | | | | | 0.925 | 0.925 | |
| 8 | 0.946 | 0.925 | | | | | |
| 10 | | | 0.930 | 0.915 | | | 0.45 |
| 12 | | | | | 0.899 | 0.899 | |

As can be seen from the table, the fully parallel FFT is less efficient than the generalized

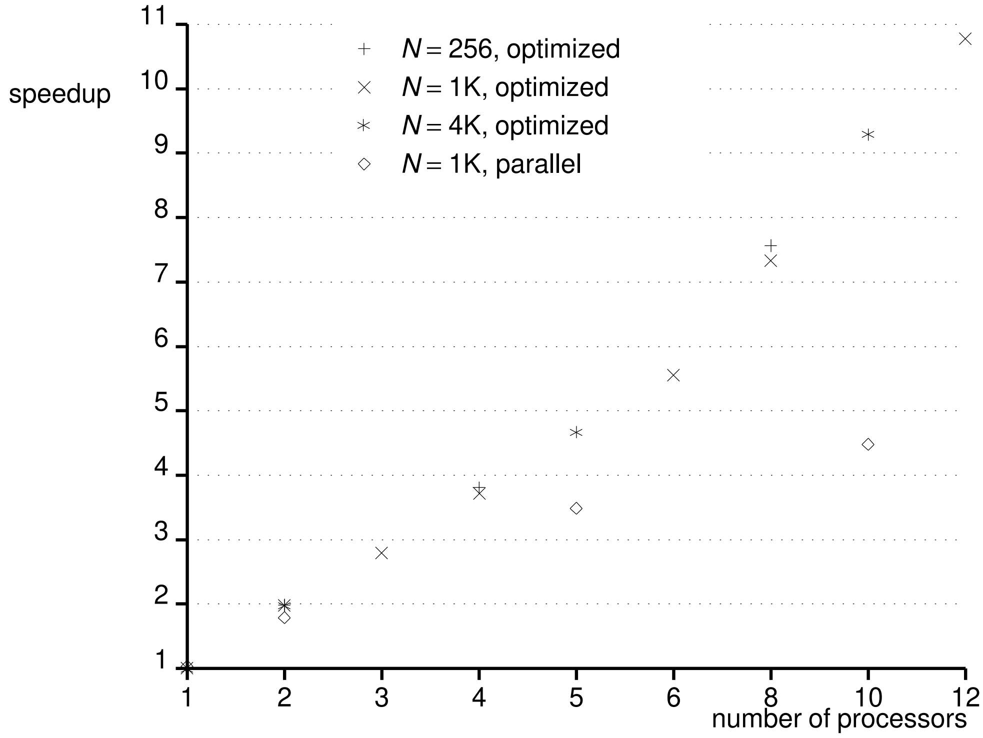


*Figure 13: speed-up for different configurations*

butterfly FFT. In the generalized butterfly FFT, the efficiency decreases only slightly with the number of processors. The decrease in efficiency is caused by the relatively higher communication and loop overhead, because less butterfly operations are performed per transputer. However, the efficiency remains near or above the predicted 90%.

## 5 Conclusions

A generalized butterfly process for performing an $n \times m$ butterfly operation has been derived. This butterfly operation can be executed very efficiently on a single transputer due to the overlap between communications and computations. The computation/communication ratio has been computed for several different values of $n$ and $m$. This ratio is in the order of 10:1 for 20 MBit/s links and a 25 MHz CPU, so CPU and memory speed limit the performance, and not the link speed.

The FFT algorithm based on the generalized butterfly is well scaleable in the size of the FFT. Using the generalized butterfly, it is possible to keep the throughput (and the efficiency) the same while increasing the FFT size.

## 6 References


[1] Cooley J.W. and J.W. Tukey,"An algorithm for the Machine Calculation of Complex Fourier Series", *Math. Comp.,* vol 19, pp. 297-301, 1965

[2] Norton A. and A.J. Silberger, "Parallellization and Performance Analysis of the Cooley-Tukey FFT Algorithm for Shared-Memory Architectures", IEEE Trans. on Computers, Vol C-36(5), May 1987.

[3] Walker P., "Using transputers for optimising cost/performance - examples of how transputers could be used for the FFT", int. report, Inmos Ltd, Bristol, UK, 198?.

[4] Rabiner L.R. and B. Gold, "Theory and Application of Digital Signal processing", Prentice Hall, 1975.

[5] Eckelman P., "Transputer - richtig eingesetzt", Elektronik, Feb 1985.

[6] Inmos, "The Occam-2 programming language reference manual", Prentice-Hall, 1989.

[7] Roebbers H.W., "Advanced Occam 2 and Transputer Engineering", lecture notes for COMETT-course, rep.nr. 90R108, Control Laboratory, Univ. of Twente, 1990.