ELSEVIER

Parallel Computing 21 (1995) 1837–1848

PARALLEL COMPUTING

Parallelizing the fast wavelet transform

Mats Holmström *

Uppsala University, Department of Scientific Computing, Box 120, S-751 04 Uppsala, Sweden

Received 23 June 1994; revised 26 June 1995

Abstract

Two new algorithms for the Fast Wavelet Transform on parallel computers are presented. They are compared to a previously published algorithm [3,4] and are found to outperform this algorithm for a Connection Machine (CM) Fortran implementation of a two-dimensional wavelet transform on a CM-200 and a CM-5.

Keywords: Fast Wavelet Transform (FWT); Connection Machine (CM); Performance results

1. Introduction

Wavelets are used in many different areas, e.g. image processing and numerical analysis. One important tool when working with wavelets is the Fast Wavelet Transform (FWT). When using wavelet methods on large scale problems the time to execute the FWT's can be prohibitively long, although the FWT has a sequential time complexity of $\mathcal{O}(N)$. One solution is to use massively parallel computers, but we are then faced with the problem of constructing an efficient FWT algorithm for such computers. In this paper three such algorithms are examined.

2. The Fast Wavelet Transform

First we state the algorithm for the periodic FWT

* E-mail: matsh@tdb.uu.se

SSDI 0167-8191(95)00047-X

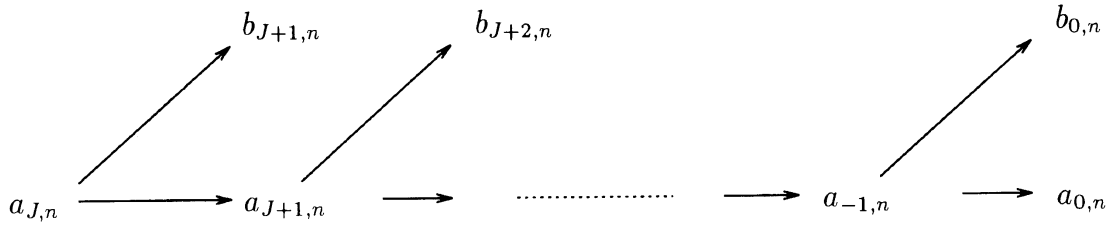

Fig. 1. The pyramid structure of the fast wavelet transform.

$$\begin{cases} a_{j,n} = \sum_{k=h_{min}}^{h_{max}} h_k a_{j-1,2n+k \bmod 2^{-j+1}} & n = 0,1,...,2^{-j}-1, \\ b_{j,n} = \sum_{k=g_{min}}^{g_{max}} g_k a_{j-1,2n+k \bmod 2^{-j+1}} & j = J+1,...,0. \end{cases} \tag{1}$$

Here $a_{j,n}$ are the scaling coefficients and $b_{j,n}$ are the wavelet coefficients. The filter coefficients are h_k and g_k where $h_{min} \le k \le h_{max}$ and $g_{min} \le k \le g_{max}$ respectively. We assume that the filters are finite, i.e., the number of filter coefficients, $|h_{max} - h_{min} + 1|$ and $|g_{max} - g_{min} + 1|$, are finite. The FWT can be viewed as a pyramid scheme with $|J|$ steps as shown in Fig. 1.

To simplify the notation we will from now on only examine one step of this pyramid scheme, to obtain the full algorithm we simply repeat this single step $|J|$-times. We introduce the notation

$$c_n = a_{j-1,n},\ a_n = a_{j,n},\ b_n = b_{j,n} \text{ and } 2^{-j} = N.$$

One step in the periodic FWT can then be written as

$$\begin{cases} a_n = \sum_{k=h_{min}}^{h_{max}} h_k c_{2n+k \bmod 2N} \\ b_n = \sum_{k=g_{min}}^{g_{max}} g_k c_{2n+k \bmod 2N} \end{cases} \quad \text{for } n = 0,1,...,N-1. \tag{2}$$

We see that $\{a_n\}_{n=0}^{N-1}$ and $\{b_n\}_{n=0}^{N-1}$ can be represented by two arrays, a and b, with N elements each, while $\{c_n\}_{n=0}^{2N-1}$ can be represented by one array, c, with $2N$ elements. At the moment we will only examine the forward transform, but for completeness, one step in the inverse periodic FWT can be stated as

$$c_n = \sum_{n|k=\bar{h}_{min}}^{\bar{h}_{max}} \tilde{h}_k a_{\frac{n-k}{2} \bmod N} + \sum_{n|k=\bar{g}_{min}}^{\bar{g}_{max}} \tilde{g}_k b_{\frac{n-k}{2} \bmod N} \quad \text{for } n = 0,1,...,2N-1. \tag{3}$$

Here the notation $n|k$ means that we only sum over those k that have the same parity as n. For orthonormal wavelet bases the filter coefficients for the forward and inverse transform are the same ($h_k = \tilde{h}_k$ and $g_k = \tilde{g}_k$), but this is not true in the general case, e.g. biorthogonal bases [2].

For the one-dimensional transform we assume that we have a linear array of PE's (Processing Elements) which we for the two-dimensional transform will replace by a mesh of PE's. The transform, as stated in Eq. (2), would then involve a lot of global communication, since we have to access elements in c with index around $2i$ for the computation of an element in a or b with index i. Our objective is to construct an algorithm that minimizes this global communication.

2.1 Algorithm 1

One way to localize the computations is to work with temporary arrays $\bar{a}$ and $\bar{b}$, where we store a and b in the elements of $\bar{a}$ and $\bar{b}$ with even index,

$$\bar{a}_{2n} = a_n \quad \textit{and} \quad \bar{b}_{2n} = b_n \quad \text{for } n = 0,\ldots,N-1. \tag{4}$$

We can then restate the original FWT (Eq. 2) as

$$\begin{cases} \bar{a}_n = \sum_{k=h_{min}}^{h_{max}} h_k c_{n+k \bmod 2N} \\ \bar{b}_n = \sum_{k=g_{min}}^{g_{max}} g_k c_{n+k \bmod 2N} \end{cases} \quad \text{for } n = 0,1,\ldots,2N-1, \tag{5}$$

followed by a downsampling of $\bar{a}$ and $\bar{b}$ (Eq. 4). This downsampling of course introduces global communication, but it only has to be done once for each step of the FWT. It should be noted that we are actually doing twice as many arithmetic operations (a.o.) as we need to do since we discard half of the elements in $\bar{a}$ and $\bar{b}$ when we downsample. This algorithm (5) will in the following be denoted Algorithm 1 and is illustrated in Fig. 2.

A possible implementation of the calculation of a in Algorithm 1, using Fortran 90 syntax, would be

```
c_forward=c
c_backward=c
```

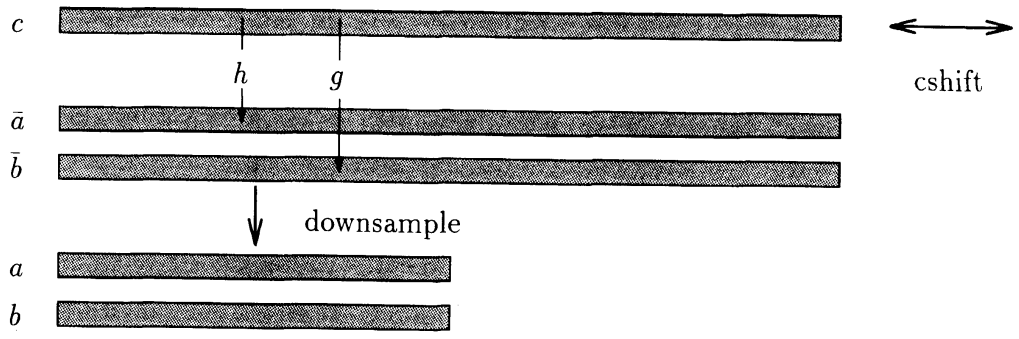

Fig. 2. Illustration of Algorithm 1. Convolution followed by downsampling.

```
i=0
a_bar=h(i)*c
do while (i < hmax)


  i=i+1
  c_forward=cshift(c_forward, 1, 1)
  c_backward=cshift(c_backward, -1, 1)
  a_bar=a_bar+h(i)*c_forward+h(-i)*c_backward


enddo
a=a_bar(0:2*n-1:2)
```

where we have assumed that the filter is symmetric around zero, i.e. $h_{max} = |h_{min}|$. If that is not the case we can insert if-statements that check that $h_{min} \leq i \leq h_{max}$ before shifting and accessing h_i. We can also choose the initial value of i to be $\lfloor (h_{max} - h_{min})/2 \rfloor$ and shift the array c by the same amount before the algorithm starts. So we use two arrays for c, one that is shifted forward and one that is shifted backwards, and we accumulate the results in a_bar along the way. Lastly the downsampling is done and the result is stored in a. This method of convolution before downsampling has been used by Pic and Essafi [4] and Lu [3].

2.2 Algorithm 2

Another approach would be to start by dividing c into two parts, one containing the elements with even index and one part containing the odd elements,

$$c^e_n = c_{2n} \quad \text{and } c^\circ_n = c_{2n+1} \quad \text{for } n = 0,\ldots,N-1. \tag{6}$$

The pattern of global communication has now changed compared to Algorithm 1. Instead of having to downsample two arrays, we have to split one array into an even and an odd part. On the CM-200 the time for this even/odd split is comparable to the time for downsampling, so the cost in terms of communication is halved compared to Algorithm 1. After the even/odd split we calculate a and b as follows

$$\begin{cases} a_n = \sum_{k=h^e_{min}}^{h^e_{max}} h^e_k c^e_{n+k \bmod N} + \sum_{k=h^\circ_{min}}^{h^\circ_{max}} h^\circ_k c^\circ_{n+k \bmod N} \\ b_n = \sum_{k=g^e_{min}}^{g^e_{max}} g^e_k c^e_{n+k \bmod N} + \sum_{k=g^\circ_{min}}^{g^\circ_{max}} g^\circ_k c^\circ_{n+k \bmod N} \end{cases} \quad \text{for } n = 0,1,\ldots,N-1. \tag{7}$$

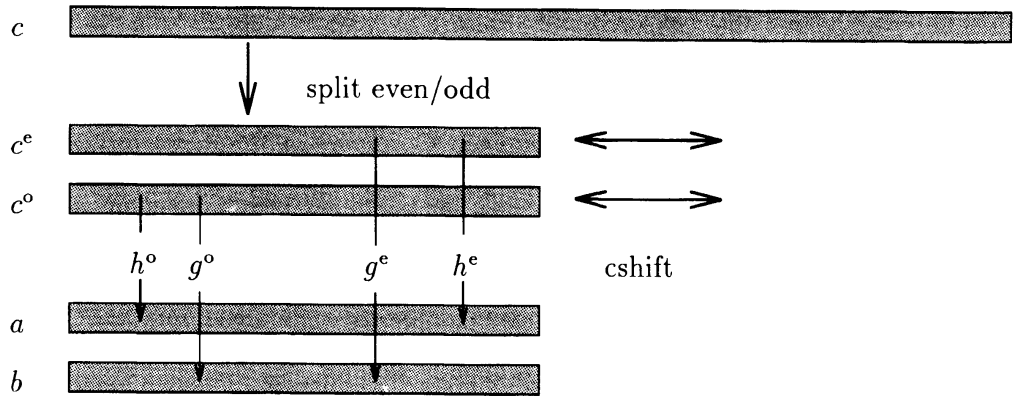

Fig. 3. Illustration of Algorithm 2. Even/odd split followed by convolution.

Note that the filter coefficients also are split into even and odd parts, i.e. $h_k^e = h_{2k}$, $h_k^\circ = h_{2k+1}$, $g_k^e = g_{2k}$ and $g_k^\circ = g_{2k+1}$. This algorithm (6), (7) will in the following be denoted Algorithm 2 and is illustrated in Fig. 3.

To calculate the inverse transform we simply reverse the order of the even/odd split and calculations, c is then given from

$$\begin{cases} c_n^e = \sum_{k=\tilde{h}^e_{min}}^{\tilde{h}^e_{max}} \tilde{h}_k^e a_{n-k \bmod N} + \sum_{k=\tilde{g}^e_{min}}^{\tilde{g}^e_{max}} \tilde{g}_k^e b_{n-k \bmod N} \\ c_n^\circ = \sum_{k=\tilde{h}^\circ_{min}}^{\tilde{h}^\circ_{max}} \tilde{h}_k^\circ a_{n-k \bmod N} + \sum_{k=\tilde{g}^\circ_{min}}^{\tilde{g}^\circ_{max}} \tilde{g}_k^\circ b_{n-k \bmod N} \end{cases} \quad \text{for } n = 0,1\ldots,N-1. \qquad (8)$$

by merging c^e and c° into c according to Eq. (6). So we have a nice symmetry between the forward and inverse transform in Algorithm 2.

A possible implementation of the calculation of a in Algorithm 2, using Fortran 90 syntax, would be

```
c_even_forward=c(0:2n-1:2)
c_even_backward=c_even_forward
c_odd_forward=c(1:2n-1:2)
c_odd_backward=c_odd_forward
i=0
a=h_even(i)*c_even_forward+h_odd(i)*c_odd_forward
do while (i<hmax)


  i=i+1
  c_even_forward=cshift(c_even_forward, 1, 1)
  c_even_backward=cshift(c_even_backward, -1, 1)
  c_odd_forward=cshift(c_odd_forward, 1, 1)
  c_odd_backward=cshift(c_odd_backward, -1,1)
```

```
  a=a+h_even(i)*c_even_forward+h_odd(i)*c_odd_forward
  a=a+h_even(-i)*c_even_backward+h_odd(-i)*c _odd_back-
     ward
enddo.
```

where we, as in Algorithm 1, have assumed that the filter is symmetric around zero. Note that we now shift four arrays, that each is half the length compared to the two arrays in Algorithm 1, and that we do not make any unnecessary computations.

2.3 Algorithm 2 on a MIMD architecture

Here we describe a more general implementation of Algorithm 2. It is well suited for MIMD (Multiple Instruction Multiple Data) architectures. We assume that we have an array of r PE's. Let the array c (length $n = 2^{-J}$), that we want to transform be distributed in a block fashion on the r PE's (assume that $r = 2^R$ and $R < |J| - 1$), so we have n/r elements of c on each PE. We assume that we have symmetric filters of length $2L + 1$. The above assumptions on the number of PE's and the filter lengths are only made to simplify the description of the algorithm, the extension to other cases is obvious. We execute Algorithm 2 on each of the PE's, which mean that we have to receive L elements from each of the two neighbor PE's. This gives us a total communication cost of $2Lr$ nearest-neighbor sends, while the number of a.o. is approximately $8Ln/r$ for each PE, for one FWT step. So the communication cost is independent of the problem size until we have less than L elements of c for each PE.

In a forthcoming paper we will study this algorithm closer and present numerical results for MIMD-machines.

2.4 Algorithm 3

A third option is to avoid the downsampling and the even/odd split altogether and instead operate on arrays of length $2^{|J|}$ at each step of the algorithm. This can be implemented as Algorithm 1 without downsampling. We then have to shift the arrays 2^{-J+j-1} positions at each level, i.e. 1,2,4,8,...,2^{-J-1} positions. If we want all the wavelet coefficients stored in one single array on the standard form,

$$[a_{0,0}\ b_{0,0}\ b_{-1,0}\ b_{-1,1} \ldots\ b_{J+1,N/2-1}] \tag{9}$$

we still would have to introduce global communication, but if we will use the computed coefficients in subsequent calculations it makes perfect sense to keep them distributed over the PE's. In the following we will describe a method for storing the coefficients in such a distributed manner, using only one array, i.e. we device a scheme to store the coefficients that are produced after each step of

Algorithm 1 without downsampling. We have to store the coefficients

$$b_{j,n} \quad \text{for } n = 0,1...,2^{-j} - 1, \quad j = J + 1,...,0, \tag{10}$$

in one array d, of length $2^{|J|}$. This can be accomplished by using the index function

$$r(j,n) = n2^{|J|+j} + 2^{|J|+j-1} - 1. \tag{11}$$

We then have the following storage scheme for the wavelet coefficients

$$b_{j,n} = d(r(j,n)) \quad \text{for } n = 0,1,....,2^{-j} - 1, \quad j = J + 1,...,0. \tag{12}$$

This can be implemented as a shift by $2^{|J|+j}$ positions of the array $\bar{b}$ in Algorithm 1, followed by an assignment to d of *only* those elements in $\bar{b}$ that would be downsampled in Algorithm 1. We do this after each step of the FWT. For the scaling coefficient(s) on the last level of the FWT we use the same index function r, as for the wavelet coefficients

$$c_{j,n} = d(r(j,n+1)) \quad \text{for } n = 0,1...,2^{-j} - 1. \tag{13}$$

As an example, if we have an array of length 16 and perform a full FWT (four steps) on this array, the resulting wavelet coefficients (and the single scaling coefficient) would be stored in d as follows

$$\begin{aligned}[b_{-3,0}\ b_{-2,0}\ b_{-3,1}\ b_{-1,1}\ b_{-3,2}\ b_{-2,1}\ b_{-3,3}\ b_{0,0}\ b_{-3,4}\cdots \\ \cdots\ b_{-2,2}\ b_{-3,5}\ b_{-1,2}\ b_{-3,6}\ b_{-2,3}\ b_{-3,7}\ c_{0,0}].\end{aligned} \tag{14}$$

Note that all communication in this algorithm is done by shifts of arrays. The price we have to pay for this is that we have to do a lot of unnecessary a.o's.

We can also note that if we keep all the intermediate wavelet coefficients after each step of the FWT we have a fast way to compute all shifts of a wavelet expansion as described by Beylkin [1].

3. The two-dimensional periodic FWT

The two-dimensional periodic FWT we use is constructed by considering the tensor product of two one-dimensional multiresolution analyses, i.e. not of the wavelet bases. In the finite case this reduces to applying the one-dimensional periodic FWT (Eq. 1) to the rows and columns of a two-dimensional array. After each step of this recursion the number of elements in each dimension of the two-dimensional array is reduced by a factor of two. So our algorithms for one-dimensional FWT's are easily adopted to the two-dimensional problem, we simply perform the same operations on all columns, and then all rows, in parallel. The extension to higher dimensions is obvious.

4. Numerical experiments

In this section we compare the performance of the three different FWT algorithms on two parallel computers, the CM-200 and the CM-5. The test problem is a three step, two-dimensional, FWT.

4.1 Timings on the CM-200

Unless stated otherwise, all the computations were made in double precision on a CM-200 with 4k bit-processors, i.e. we have 128 FPU's. The filter coefficients used are those introduced by Daubechies [2], which corresponds to a compactly supported wavelet basis with maximum number of vanishing moments. The order (filter length) used is 20, i.e. $|h_{\max} - h_{\min} + 1| = |g_{\max} - g_{\min} + 1| = 20$. The test problem is a three step FWT on a two-dimensional array with N^2 elements. So we stop in the pyramid scheme after three levels, when we have $(N/8)^2$ scaling coefficients left. When we use wavelets to solve partial differential equations, the reason for stopping before reaching the top of the pyramid scheme could be that we want to keep all coefficients on a certain level and above for accuracy reasons, i.e. we set a limit on how coarse the grid can be. The timings for this problem are presented in Fig. 4. The sequential algorithm is an implementation in Fortran 90 on a Sun SPARC-10 and is included for comparison.

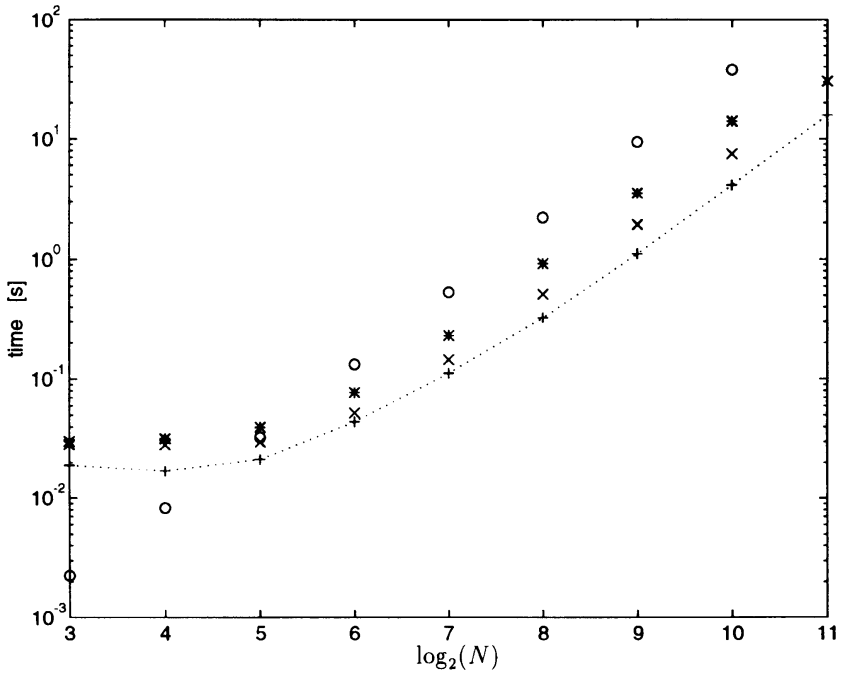

Fig. 4. Comparison of execution times for some two-dimensional FWT algorithms. The FWT is done in three stages on a square with N^2 points. 'o' = a sequential algorithm on a Sun SPARC-10. '$*$' = Algorithm 1, '$\times$' = Algorithm 2 and '$+$' = Algorithm 3. All three algorithms were executed on a CM-200, configured with 4k bit-processors (128 FPU's). (The corresponding timings for a CM-5 are presented in Fig. 6.)

To examine the scalability of Algorithm 3 we run the same problem as above with 4k and 8k bit-processors and compare the timings. The result is presented in Fig. 5, and we can see that the timings for 8k processors are about twice as fast as those for 4k processors (for large enough N), as expected.

In Table 1 we compare the time spent on different CM-operations for the three algorithms. The problem was the same as above with $N = 256$. The *time*-column denotes total time of execution. *CM-total* is the percentage of time that the CM was busy. For all three algorithms the CM was idle only nine percent of the time. The percentage of time spent on global communication on the CM is listed under *send/get*. In our algorithms the global communication (router communication) occurs when we are downsampling arrays with commands such as

```
a=a_bar(0:2*n-1:2)
```

in Algorithm 1. Local communication (grid communication) is listed under *NEWS*, and occurs when we are doing shifts such as

```
c_forward=cshift(c_forward,1,1)
```

in Algorithm 1. Lastly we have *CM-CPU* which is the time spent processing on the PE's.

We can in Table 1 see that, as expected, Algorithms 1 and 2 are dominated by global communication from the downsampling step, while the execution time of

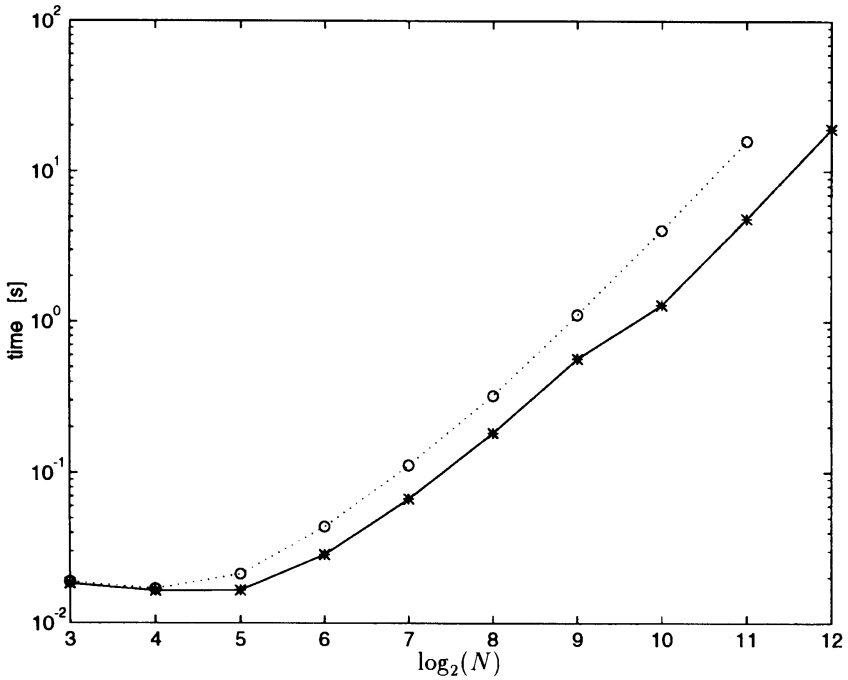

Fig. 5. Comparison of execution times on a CM-200 for a two-dimensional FWT using Algorithm 3 and two different machine configurations. The FWT is done in three stages on a square with N^2 points. 'o' = 4k bit-processors (128 FPU's), '*' = 8k bit-processors (256 FPU's).

Table 1
Percentage of time spent for different operations on the CM as reported by PRISM (a debugger for the CM with timing capabilities). Three step FWT for $N = 256$

		Percentage of time			
	Time [s]	CM total	Send/get	NEWS	CM CPU
Algorithm 1	0.92	91	52	10	29
Algorithm 2	0.51	91	62	9	20
Algorithm 3	0.32	91	1	40	50

Algorithm 3 is evenly split between shifts and CPU-time for the actual computations.

For a comparison of execution times with previously published results we did a two-dimensional, three step, FWT for $N = 1024$. The filters were those of Daubechies, with length six and the computations were now done in single precision. The execution time for Algorithm 1 was 6.1 seconds on a 4k-machine. We can compare this to the execution time reported by Lu [3] on 16k-machine which was between 1 and 2 seconds. If we scale this time to adjust for the larger number of processors, we see that the times are comparable in length. We can also note that Algorithm 3 had an execution time of 1.3 seconds for the same problem with only 4k processors.

Pic and Essafie [4] also provide some timings for Algorithm 1, but we have not made any comparisons with those times since they only perform one step of the FWT, thus avoiding all communication problems.

4.2 Timings on the CM-5

We have also measured the execution times of the three algorithms on a 32 node CM-5. The test problem was the same as for the CM-200, described in detail in the last section. The timings for this problem are presented in Fig. 6.

If we compare the timings on the CM-5 in Fig. 6 with those on the CM-200 in Fig. 4 we note that the same trend is apparent. Algorithm 2 performs slightly better than Algorithm 1, and Algorithm 3 is significantly faster than the other two. Since the CM-5 nodes are connected in a tree structure, this shows that the good performance of Algorithm 3 is not limited to hypercubes.

5. Conclusions

We have constructed two new algorithms (Algorithm 2 and 3) and compared their execution times to a previously published algorithm (Algorithm 1) [3,4]. In all three algorithms there is a trade-off between global communication and doing unnecessary arithmetic operations. Algorithm 1 is suffering from both of these drawbacks and is the slowest of the three algorithms. Algorithm 2 needs global

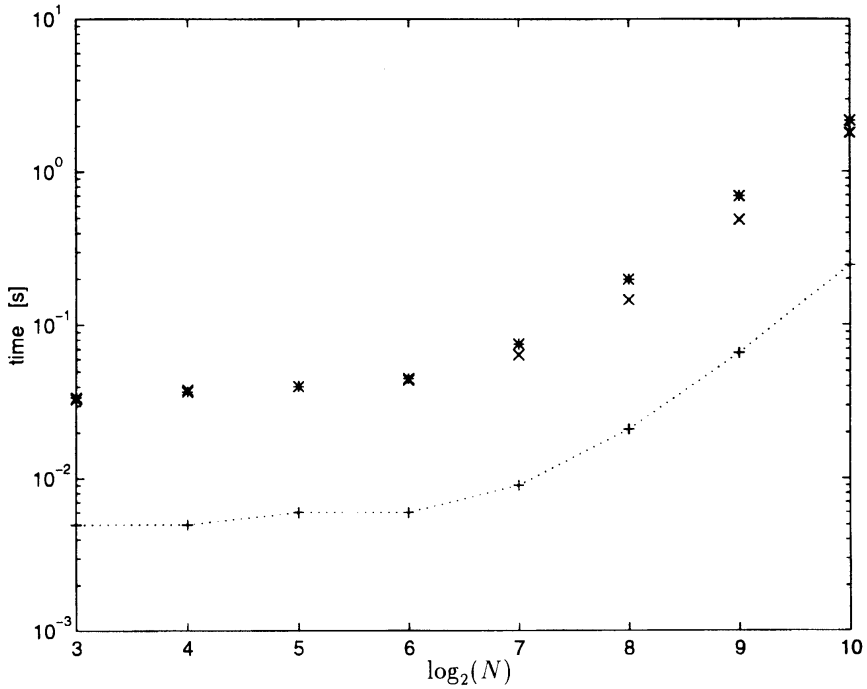

Fig. 6. Comparison of execution times for some two-dimensional FWT algorithms. The FWT is done in three stages on a square with N^2 points. '$*$' = Algorithm 1, '$\times$' = Algorithm 2 and '$+$' = Algorithm 3. All three algorithms were executed on a 32-node CM-5. (The corresponding timings for a CM-200 are presented in Fig. 4.)

communication (although only half as much as Algorithm 1) but does no unnecessary a.o. As a result of that, an advantage of Algorithm 2 is that it is also well suited for sequential execution. Algorithm 3 does a lot of unnecessary a.o. and is dependent on the ability to shift arrays fast in power-of-two steps, but does not introduce any global communication from downsampling or even/odd splits of arrays. Since global communication is expensive on most parallel computers Algorithm 3 seems to be the best choice among the three algorithms for doing FWT's on parallel computers (at least on SIMD machines), which is confirmed by our timings on the CM-200 and the CM-5. An added advantage of Algorithm 3 is that the wavelet coefficients are distributed on the PE's after the FWT computation, which is desirable if we will do further computations using the coefficients, e.g. if the FWT is part of a solver for partial differential equations.

6. Implementation notes

The three algorithms were implemented on the Connection Machine 200 at the Royal Institute of Technology in Stockholm, Sweden. The language was CM Fortran which is essentially a subset of Fortran 90 containing the same array notation as Fortran 90. Some observations were as follows:

- The `cshift` function is very slow on a CM-200 if we shift on array sections, so in Algorithm 1 and 2 we have to do each step of the FWT in a subroutine and

allocate arrays of proper sizes in that subroutine. In this way we shift whole arrays and gain a speedup on the order of two magnitudes.

- On a machine with hypercube topology, such as the CM-200, the `cshift`'s in Algorithm 3 will be fast since two elements whose distance from each other is a power of two will be nearest neighbors on a hypercube.

We also timed the algorithms on a 32 node CM-5 at NCAR (National Center for Atmospheric Research) in Boulder, Colorado. The same CM Fortran codes as for the CM-200 were used, i.e. there were no attempts at machine specific optimizations for the CM-5.

Acknowledgements

I thank Fredrik Hedman at the Royal Institute of Technology in Stockholm, Sweden, and Gary Jensen at NCAR in Boulder, Colorado, for their assistance in using the CM-200 and the CM-5, respectively.

References

[1] G. Beylkin, On the representation of operators in bases of compactly supported wavelets, *SIAM J. Numerical Analysis 6(6) (Dec. 1992) 1716–1740.*

[2] I. Daubechies, *Ten Lectures on Wavelets*, vol. 61 of *CBMS-NSF Regional Conferences Series in Applied Mathematics* (SIAM, Philadelphia, PA, 1992).

[3] J. Lu, Parallelizing Mallat algorithm for 2-d wavelet transforms, *Information Processing Letters 45 (April 1993) 255–259.*

[4] M. M. Pic and H. Essafi, Wavelet transform on Connection Machine and Sympatia 2, *Int. J. Modern Physics C, 4(1) (1993) 97–103.*