Chapter 3

Vectorization and Parallelization of FWT

3.1 The CP200 Vector Parallel Computer

The vector units on the **CP200** computer run with a clock frequency of 142 MHz. They can execute 8 multiplication and 8 additions per clock cycle, leading to a peak performance of 2.272 $Gflops$/processor. The actual performance however depends on many factors such as

- Vector length;
- Memory stride;
- Arithmetic density;
- Ratio of arithmetic operations to load/store operations;
- Type of operations.

A vector processor is designed to perform arithmetic operations on vectors of numbers through hardware pipelining. There is an overhead involved with each vector instruction, so good performance requires very long vector lengths. As with all modern computers, the memory speed up of the **CP200** falls short of the processor speed. To overcome this problem, memory is arranged in banks with consecutive elements spread across the banks. Stride-one access the means that the memory banks have time to recover between the consecutive memory access so that they

are always ready to deliver a piece of data at rate at which it is requested. Furthermore, the **CP200** has a special instruction for this which is faster than any non-uniform memory access. Finally, on computer architectures using cache, stride-one means that all elements in a cache line will be used before it is flushed. In either case, a stride different from one can lead to poor performance because the processor has to wait until the data are ready.

Optimal performance on the **CP200** requires that 8 multiplication and 8 additions occurs every clock cycle. Therefore, any loop containing only addition or multiplication can never run faster than half the peak performance. Also the use of load and store pipes are crucial. The **CP200** has one load and one store pipe, so addition of two vectors say can at best run at $\frac{1}{4}$ of the peak performance because two loads are needed at each iteration. Finally, the type of arithmetic operations is crucial to the performance. For example, a division takes seven cycles on the **CP200**. Taking all these issue into account we shall see that good vector performance requires operations of the form
$for\ n = 0 : N - 1,$
$[\mathbf{y}]_n \leftarrow a * [\mathbf{x}]_n + b$
where a and b are scalars and $\mathbf{x}$ and $\mathbf{y}$ are vectors of length N, with N being large.

3.2 Vectorization of Fast Wavelet Transform

3.2.1 Introduction

Problems involving the FWT are typically large and wavelet transforms can be time consuming even though the algorithmic complexity is proportional to the problem size. The use of high performance computers is one way of speeding the FWT.

In this section and in the next section, we have made an attempt to implement the $1D$ and the $2D$ FWT on a selection of high performance computers, especially the **CP200**. For simplicity, we will assume that $k = 1$ throughout this chapter (see Definition 2.8.1). We have a vector parallel computer **CP200**. This means that it consists of a number of vector processors connected in an efficient network. Good vector performance on the individual processors is therefore crucial to good parallel performance. In this section, we have discussed the implementation and performance of the FWT on one node of **CP200**. In the next section, we discussed the parallelization of FWT and report results on several nodes on the **CP200**.

3.2.2 1 D Fast Wavelet Transform

The basic operations in the $1D$ FWT can be written in the form
$for\ n = 0 : S/2 - 1,$
$[\mathbf{w}]_n \leftarrow [\mathbf{w}]_n + a_l * [\mathbf{x}]_{<l+2n>_S}$
is defined by the recurrence relations (2.32). The arithmetic density as well as the ratio of arithmetic operations to load/store operations are good. However, memory is accessed with stride two because of the inherent double shift in the wavelet transform and indices must be wrapped because of periodicity. Therefore, optimal performance is not expected for the $1D$ FWT. Our implementation of the FWT on one node of the **CP200** yields the performance shown in Table 3.1. We make the following observations from Table-3.1. Firstly, the performance is far

N	F	CPU time (μ s)	R(Mflop/s)
1024	81840	676	121
2048	163760	844	194
4096	327600	1118	293
8192	655280	1790	366
16384	1310640	3260	402
32768	2621360	5650	464
65536	5242800	10180	515
131072	10485680	19311	543
262144	20971440	38130	550
524288	41942960	68646	611

Table 3.1: **Timings of the FWT.** $D = 20$, $N = 2^J$, $J = 10, 11, \cdots, 19$, $\lambda = J$.

from optimal even for the largest value of N. Secondly, the performance improves only slowly as N increases. To understand the latter property, we conduct performance analysis of one step (the PWT) of the recurrence formulas (2.32). Since this is a simple operation on one vector, we assume that computation time in the vector processor follows the model

$$T = t_s + t_v F, \tag{3.1}$$

where F is the number of floating operations, T is the execution time, t_v is the computation time for one floating point operations in the pipelines, and t_s is the start up time. The performance expressed in floating point operations per second is than

$$R = \frac{F}{T}. \tag{3.2}$$

Letting F go to infinity, results in the theoretically optimal performance

$$R_\infty \equiv \lim_{F\to\infty} \frac{F}{T} = \lim_{F\to\infty} \frac{F}{t_s + t_v F} = \frac{1}{t_v}. \tag{3.3}$$

Let $\alpha \in [0,1]$ be the fraction of R_∞ which is achieved for a given problem of size F_α. Then F_α is found from (3.2) with $R = \alpha R_\infty$:

$$\frac{\alpha}{t_v} = \frac{F_\alpha}{t_s + t_v F_\alpha}$$

which has solution

$$F_\alpha = \frac{\alpha}{1-\alpha}\frac{t_s}{t_v}.$$

In particular, for $\alpha = \frac{1}{2}$, we find

$$F_{\frac{1}{2}} = \frac{t_s}{t_v}$$

which is another characteristic performance parameter for the algorithm in question. F_α can now be expressed in terms of $F_{\frac{1}{2}}$ as

$$F_\alpha = F_{\frac{1}{2}}\frac{\alpha}{1-\alpha}.$$

For example, to reach 80% of the maximum performance, a problem size of $F = 4F_{\frac{1}{2}}$ is required and $F = 9F_{\frac{1}{2}}$ is needed to reach 90%. The parameter $F_{\frac{1}{2}}$ can therefore be seen as a measure of how quickly the performance approaches R_∞. A large value of $F_{\frac{1}{2}}$ means that the problem must be very large in order to get good performance. Hence, we wish to $F_{\frac{1}{2}}$ is small as possible.

In order to estimate the characteristic parameters $F_{\frac{1}{2}}$ and R_∞, we use measurements of CPU time in the **CP200**. Table 3.2 shows the timings of the sequence of steps needed to compute the full FWT with $\lambda = J = 19$, $N = 2^J$ and $D = 20$, i.e. the PWT applied successively to vectors of length $S = N, \frac{N}{2}, \cdots, 2$. Using these measurements, we estimate the parameters t_s and t_v to be $t_s = 76\mu s$ and $t_v = 0.00157\mu s$. Consequently,

$$(R_\infty)_{PWT} = 637 \text{ Mflops}$$

and

$$(F_{\frac{1}{2}})_{PWT} = 48222 \text{ operations}. \tag{3.4}$$

It can be verified that the values predicted by the linear model corresponding with those observed in Table 3.2. The execution time of the PWT, thus follows the model

$$T_{PWT}(S) = t_s + t_v F_{PWT}(S)$$

S	F	$T(\mu\ s)$	R(Mflop/s)
524288	20971520	32974	636
262144	10485760	16539	634
131072	5242880	8283	633
65536	2621440	4188	626
32768	1310720	2142	612
16384	655360	1120	586
8192	327680	598	548
4096	163840	344	476
2048	81920	180	455
1024	40960	130	315
512	20480	97	211
256	10240	85	120
128	5120	80	64
64	2560	50	32
32	1280	38	16
16	640	36	8
8	320	32	4
4	160	28	2
2	80	29	1

Table 3.2: **Timings of the PWT.** $D = 20$, $N = 2^J$, $J = 19$, **and** $\lambda = J$.

which can be used to predict the execution time of the FWT to depth λ_N as follows:

$$\begin{aligned} T_{FWT}(N) &= \sum_{i=0}^{\lambda_N - 1} T_{PWT}\left(\frac{N}{2^i}\right) \\ &= \sum_{i=0}^{\lambda_N - 1} t_s + t_v F_{PWT}\left(\frac{N}{2^i}\right) \\ &= t_s \lambda_N + t_v \sum_{i=0}^{\lambda_N - 1} F_{PWT}\left(\frac{N}{2^i}\right) \end{aligned}$$

or, if λ_N assumes its maximal value

$$T_{FWT}(N) = t_s \log_2 N + t_v F_{FWT}(N). \tag{3.5}$$

3.2.3 Multiple $1D$ Fast Wavelet Transform

Consider a matrix $\mathbf{X} \in \mathbf{R}^{M,N}$. We assume that $\mathbf{X}$ is stored by columns so that consecutive elements in each column are located in consecutive positions in memory. Applying the $1D$ FWT to every column of $\mathbf{X}$ leads to inefficient data access and large $F_{1/2}$ as described in the previous section. By applying the FWT to the *rows* of $\mathbf{X}$ instead, one can vectorize over the columns such that all elements will be accessed with stride-one in vectors of length $\mathbf{M}$. We will refer to this procedure as the **multiple** $1D$ **FWT** (MFWT). Applying the MFWT to a matrix $\mathbf{X}$ corresponds the expression

$$\mathbf{X}(\mathbf{W}^{\lambda_N})^T \tag{3.6}$$

where $\mathbf{W}^{\lambda_N}$ is defined as in (2.33). Since there are $\mathbf{M}$ rows, the number of floating point operations needed are

$$F_{MFWT}(M,N) = 4DMN\left(1 - \frac{1}{(2^{\lambda_N})}\right) \tag{3.7}$$

The recurrence formulas now take the form

$$c_{m,n}^{i+1} = \sum_{l=0}^{D-1} a_l c_{m,<l+2n>_{S_i}}^{i} \tag{3.8}$$

$$d_{m,n}^{i+1} = \sum_{l=0}^{D-1} b_l c_{m,<l+2n>_{S_i}}^{i} \tag{3.9}$$

where $i = 0, 1, \cdots, \lambda_N - 1$, $m = 0, 1, \cdots, M-1$, and $n = 0, 1, \cdots, S_{i+1} - 1$. Timings for the MFWT with $\lambda_N = J = 10$, $N = 2^J$, and $D = 20$, where only the vectorized dimension M is varied, are shown in Table 3.3.

M	F	$T(\mu\ s)$	R(Mflop/s)
16	1309440	11240	120
32	2618880	10889	240
64	5237760	11133	474
128	10475520	11474	912
256	20951040	14941	1402
512	41902080	23594	1777
1024	83804160	43711	1919
2048	167608320	84994	1974

Table 3.3: **Timings of the MFWT.** $D = 20$, $N = 2^J$, $\lambda_N = J = 10$, **and** $M = 16, 32, \cdots, 2048$.

We will now derive a performance model for this case. Each step of the MFWT applies a PWT of length S_i to the M rows of $\mathbf{X}$. Hence, by (2.35) the number of flops are $2DS_iM$. Vectorization is achieved by putting m into the innermost loop, so computations on each column can be assumed to follow the linear model for vectorization (3.1). Hence, the execution time for one step of the MFWT (3.8) and (3.9) is $S_i(t_s + 2DMt_v)$ and the execution time for the entire MFWT is

$$\begin{aligned} T_{MFWT} &= \sum_{i=0}^{\lambda_N - 1} \frac{N}{2^i}(t_s + 2DMt_v) \\ &= 2N\left(1 - \frac{1}{2^{\lambda_N}}\right)(t_s + 2DMt_v) \end{aligned} \tag{3.10}$$

The performance is then given by

$$R_{MFWT} = \frac{F_{MFWT}}{T_{MFWT}} = \frac{2DM}{t_s + 2DMt_v} \tag{3.11}$$

and $(R_\infty)_{MFWT} = 1/t_v$ as usual. However, we observe that the performance measure is independent of the depth λ_N. The parameter $M_{1/2}$ is found by solving

$$\frac{1}{2t_v} = \frac{2DM_{1/2}}{t_s + 2DM_{1/2}t_v} \tag{3.12}$$

which has the solution

$$M_{1/2} = \frac{t_s}{2Dt_v} \tag{3.13}$$

Hence,

$$\begin{aligned} (F_{1/2})_{MFWT} &= 4DN\left(1 - \frac{1}{2^{\lambda_N}}\right) M_{1/2} \\ &= 2N\left(1 - \frac{1}{2^{\lambda_N}}\right)\frac{t_s}{t_v} \end{aligned}$$

Using (3.10) and the measurements in Table 3.3, we get the new estimates

$$t_s = 3.73\mu s; \quad t_v = 0.000450\mu s.$$

These estimates are different from those of the $1D$ case and reflect the fact that the MFWT algorithm performs better on the **CP200**. Consequently, we now have

$$(R_\infty)_{MFWT} = 2.222 \quad \text{Gflop/s}$$

and

$$(F_{1/2})_{MFWT} = 16959000 \quad \text{Operations}$$

the latter corresponding to a vector length $M_{1/2} = 208$. These values are close to being optimal on the **CP200** (recall that the peak performance per processor is 2.272 Gflop/s). Finally, since $(F_{1/2}(MFWT))$ grows with N, we note that the MFWT is best for matrices with $M \geq N$.

3.2.4 2D Fast Wavelet Transform

The $2D$ wavelet transform is defined by the matrix product

$$\check{\mathbf{X}} = \mathbf{W}^{\lambda_M} \mathbf{X} \left(\mathbf{W}^{\lambda_M}\right)^T \tag{3.14}$$

The expression $\mathbf{X}\left(\mathbf{W}^{\lambda_N}\right)^T$ leads to vector operations on vectors of length M and stride-one data access as described in Subsection 3.2.3. This is not the case for the expression $\mathbf{W}^{\lambda_M}\mathbf{X}$, because it consists of a collection of columnwise $1D$ transforms which do not access the memory efficiently as described in Subsection 3.2.2. However, (3.14) can be written as

$$\check{\mathbf{X}}^T = \left(\mathbf{X}\left(\mathbf{W}^{\lambda_N}\right)^T\right)^T \left(\mathbf{W}^{\lambda_M}\right)^T \tag{3.15}$$

yielding the efficiency of the multiple $1D$ FWT at the cost of one transpose step. We call this the **split-transpose algorithm**. It consists of the following three stages:

Algorithm: Split-transpose

1. $\mathbf{Z} = \mathbf{X}\left(\mathbf{W}^{\lambda_N}\right)^T$
2. $\mathbf{U} = \mathbf{Z}^T$
3. $\check{\mathbf{X}}^T = \mathbf{U}\left(\mathbf{W}^{\lambda_N}\right)^T$

Transposition can be implemented efficiently on a vector processor by accessing the matrix elements along the diagonals [Heg95], so the $2D$ FWT retains the good vector performance of the MFWT. This is verified by the timings given in Table 3.3.

Disregarding the time for the transposition step, a simple model for the $2D$ FWT execution time is

$$T_{FWT2}(M,N) = T_{MFWT}(M,N) + T_{MFWT}(N,M) \tag{3.16}$$

where T_{MFWT} is given by (3.1).

3.2.5 Conclusion

The FWT has been implemented in the **CP200**. The one dimensional has relatively high value of $N_{1/2}$ and stride-two memory access so the performance is not very good. The $2D$ FWT can be arranged so that these problem are avoided, and a performance of more than 80% of the

M	F	$T(\mu\ s)$	R(Mflop/s)
16	2538240	11903	213
32	5158400	12748	404
64	10398720	14305	726
128	20879360	17686	1180
256	41840640	27167	1540
512	83763200	47686	1756
1024	167608320	91751	1826

Table 3.4: **Timings of the $2D$ FWT (FWT2).** $D = 20$, $N = 2^J$, $\lambda_N = J = 10$, **and** $M = 16.32.\cdots.1024$.

theoretical peak performance is achieved even for relatively small problems. This is fortunate as the $2D$ FWT computing more intensive than the $1D$ FWT and consequently, it justifies the better use of super computers.

3.3 Parallelization of Fast Wavelet Transform

3.3.1 Introduction

With a parallel architecture, the aim is to distribute the work among several processors in order to compute the result faster or to be able to solve larger problems than what is possible with just one processor. Let $T^0(N)$ be the time it takes to compute the FWT with a sequential algorithm on one processor. Ideally,the time needed to compute the same task on P processor is then $T^0(N)/P$. However, there are a number of reasons why this ideal is rarely possible to meet:

- There will normally be some computational overhead in the form of book keeping involved in the parallel in the parallel algorithm. This adds to the execution time.
- If r is the fraction of the time on P processor is bound from below by $(1-r)T^0(N) + rT^0(N)/P$ which is also larger than the ideal. This is known as Amdahl's law.
- The processor might not be assigned the same amount of work. This means that some processor will be ideal while others are doing more than their fair share of the work. In

that case,the parallel execution time will be determined by the processor which is the last to finish. This is known as the problem of good load balancing.

- Processor must communicate information and synchronize in order for the arithmetic to be performed on the correct data and in the correct sequence. This communication and synchronization will delay the computation depending on the amount which is communicated the frequency by which it occurs.

In this section, we will discuss different parallelization strategies for the FWT's with special regard to the effects of load balancing, communication, and synchronization. We will disregard the influence of the first two points since we assume that the parallel overhead is small and that the FWT has no significant unparallelizable part. However, in applications, using the FWT this problem may become significant. Most of the material covered in this section has also appeared in [NH97].

3.3.2 1D Fast Wavelet Transform

We will now address the problem of distributing the work needed to compute the FWT ($\mathbf{y} = \mathbf{W}\mathbf{x}$) as defined in Definition 2.8.1 on P processors denoted by $p = 0, 1, \cdots, P-1$. We assume that the processors are organized in a ring topology such that $\angle p-1\rangle_P$ and $\langle p+1\rangle_P$ are the left and right neighbors of processor p, respectively. Assume also, for simplicity, that N is a multiple of P and that the initial vector x is distributed such that each processor receives the same number of consecutive elements. This means that the processor p holds the element

$$\{c_n^0\}_n, \quad n = p\frac{N}{P}, p\frac{N}{P}+1, \cdots, (p+1)\frac{N}{P}-1. \tag{3.17}$$

A question that is crucial to the performance of a parallel FWT is how to chose the optimal distribution of y and the intermediate vectors.

We consider first the data layout suggested by the sequential algorithm in Definition 2.13.1. This is shown in Table 3.5. It is seen that distributive the results of each transform step evenly across the processors results poor load balancing because each step works with the lower half of the previous vector only. The processors containing parts that are finished early are ideal in the subsequent step. In addition, global communication is required in the first step because every processor must know the values on every other processor in order to compute its own part of the wavelet transform. In subsequent, this communication will take place among the active processors only. This kind of layout was used in [BKDC95] where it was observed that optimal load balancing could not be achieved, and also in [Lu93] where the global communication

$p=0$	$p=1$
$c_0^0 c_1^0 c_2^0 c_3^0 c_4^0 c_5^0 c_6^0 c_7^0$	$c_8^0 c_9^0 c_{10}^0 c_{11}^0 c_{12}^0 c_{13}^0 c_{14}^0 c_{15}^0$
$\downarrow$	$\downarrow$
$c_0^1 c_1^1 c_2^1 c_3^1 c_4^1 c_5^1 c_6^1 c_7^1$	$d_0^1 d_1^1 d_2^1 d_3^1 d_4^1 d_5^1 d_6^1 d_7^1$
$\downarrow$	$\downarrow$
$c_0^2 c_1^2 c_2^2 c_3^2 d_0^2 d_1^2 d_2^2 d_3^2$	$d_0^1 d_1^1 d_2^1 d_3^1 d_4^1 d_5^1 d_6^1 d_7^1$
$\downarrow$	$\downarrow$
$c_0^3 c_1^3 d_0^3 d_1^3 d_0^2 d_1^2 d_2^2 d_3^2$	$d_0^1 d_1^1 d_2^1 d_3^1 d_4^1 d_5^1 d_6^1 d_7^1$

Table 3.5: **Standard data layout results in poor load balancing. Here** $P = 2$, $N = 16$, **and** $\lambda = 3$.

was treated by organizing the processors of a connection machine (CM2) in a pyramid structure.

However, we can obtain perfect load balancing and avoid global communication by introducing another ordering of the intermediate N resulting vectors. This is shown in Table 3.6. Processor

$p=0$	$p=1$
$c_0^0 c_1^0 c_2^0 c_3^0 c_4^0 c_5^0 c_6^0 c_7^0$	$c_8^0 c_9^0 c_{10}^0 c_{11}^0 c_{12}^0 c_{13}^0 c_{14}^0 c_{15}^0$
$\downarrow$	$\downarrow$
$c_0^1 c_1^1 c_2^1 c_3^1 d_0^1 d_1^1 d_2^1 d_3^1$	$c_4^1 c_5^1 c_6^1 c_7^1 d_4^1 d_5^1 d_6^1 d_7^1$
$\downarrow$	$\downarrow$
$c_0^2 c_1^2 d_0^2 d_1^2 d_0^1 d_1^1 d_2^1 d_3^1$	$c_2^2 c_3^2 d_2^2 d_3^2 d_4^1 d_5^1 d_6^1 d_7^1$
$\downarrow$	$\downarrow$
$c_0^3 d_0^3 d_0^2 d_1^2 d_0^1 d_1^1 d_2^1 d_3^1$	$c_1^3 d_1^3 d_2^2 d_3^2 d_4^1 d_5^1 d_6^1 d_7^1$

Table 3.6: **Standard data layout results in poor load balancing. Here** $P = 2$, $N = 16$, **and** $\lambda = 3$.

p will now compute and store the elements $\{c_n^{i+1}\}_n$ and $\{d_n^{i+1}\}_n$ where

$$n = p\frac{N}{P2^{i+1}}, \frac{N}{P2^{i+1}} + 1, \cdots, (p+1)\frac{N}{P2^{i+1}} - 1, \tag{3.18}$$

$i = 0, 1, 2, \cdots, \lambda - 1$.

Let now $S_i^P = S_i/P = N/(P2^i)$. Then the recurrence formula are almost the same as (2.32):

$$\begin{array}{rcl} c_n^{i+1} & = & \sum_{l=0}^{D-1} a_l c^i_{\langle l+2n\rangle_{S_i}} \\ d_n^{i+1} & = & \sum_{l=0}^{D-1} b_l c^i_{\langle l+2n\rangle_{S_i}} \end{array} \tag{3.19}$$

where $i = 0, 1, \cdots, \lambda - 1$ and $n = pS_{i+1}^P, pS_{i+1}^P + 1, \cdots, (p+1)S_{i+1}^P - 1$. The difference lies in the periodic wrapping which is still global, i.e. elements from processor 0 must be copied too processor $P - 1$. However, it turns out that this is just a special case of the general communication pattern for the algorithms.

Note that the layout shown in Table 3.6 is a permutation of the layout shown in Table 3.5 because each processor essentially performs a local wavelet transform of its data. However, the ordering suggested by Table 3.5 and also by equation (2.28) is by no means intrinsic to the FWT so this permutation is not a disadvantage at all. Rather, one might argue as follows:

> *Local transforms reflect better the essence of the wavelet philosophy because all scale information concerning a particular position remains on the same processor.*

This layout is even likely to increase performance for further processing steps (such as compression) because it preserves locality of data.

Note also that the local transforms in this example have reached their ultimate form on each processor after only 3 steps and that it would not be feasible to continue the recursion further (i.e. by letting $\lambda = 4$ and splitting $\{c_0^3, c_1^3\}$ in to $\{c_0^4, d_0^4\}$) because then $N/(P2^\lambda) < 1$, (3.18) no longer holds, and the resulting data distribution would lead to load imbalance as with the algorithm mentioned above. Thus, to maintain good load balancing we must have an upper bound on λ:

$$\lambda \leq \log_2 \left(\frac{N}{P} \right) \tag{3.20}$$

In fact, this bound has to be even more restrictive in order to avoid excessive communication.

Communication

We will now consider the amount of communication required for the parallel $1D$ FWT. Consider the computations done by processor p on a row vector as indicated in Figure-3.1. The quantities in (3.19) can be computed without any communication provided that the index $l + 2n$ does not refer to elements on other processors, i.e.

$$\begin{aligned} l + 2n &\leq (p+1)\frac{N}{P2^i} - 1 \\ n &\leq (p+1)\frac{N}{P2^{i+1}} - \frac{l+1}{2} \end{aligned} \tag{3.21}$$

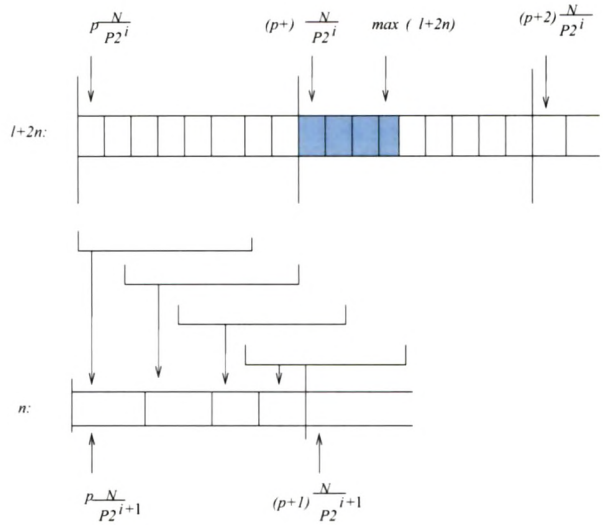

Figure 3.1: **Communications on processor p involve $D-2$ elements from processor $p+1$. Here $D=6$ and $N/(P2^i)=8$. The lines of width D indicate the filters as they are applied for different values of n.**

A sufficient condition (independent of l) for this is

$$n \leq (p+1)\frac{N}{P2^{i+1}} - \frac{D}{2} \tag{3.22}$$

since $l \in [0, D-1]$. We use this criteria to separate the local computations from those that may require communication.

For a fixed $n > (p+1)N/(P2^{i+1}) - D/2$ computations are still local as long as (3.21) is fulfilled, i.e. when

$$l \leq (p+1)\frac{N}{P2^i} - 2n - 1 \tag{3.23}$$

However, when l becomes larger than this, the index $l+2n$ will point to elements residing on a processor located to the right of processor p. The largest value of $l+2n$ (found from (3.19) and (3.18)) is

$$\max(l+2n) = (p+1)\frac{N}{P2^i} + D - 3 \tag{3.24}$$

The largest value of $l+2n$ for which communication is not necessary is

$$(p+1)\frac{N}{P2^i} - 1.$$

Subtracting this quantity from (3.24), we find that exactly $D-2$ elements must be communicated to processor p at each step of the FWT as indicated in Figure-3.1.

A tighter bound on λ

It is a condition for good performance that the communication pattern described above takes place between nearest neighbors only. Therefore, we want to avoid situations where processor p needs data from processors other form its right neighbor $\langle p+1 \rangle_p$ so we impose the additional restriction

$$\begin{aligned} \max(l+2n) &\leq (p+2)\frac{N}{P2^i} - 1 \\ (p+1)\frac{N}{P2^i} + D - 3 &\leq (p+2)\frac{N}{P2^i} - 1 \\ D - 2 &\leq (p+2)\frac{N}{P2^i} \end{aligned} \tag{3.25}$$

Since we want (3.25) to hold for all $i = 0, 1, \cdots, \lambda - 1$, we get

$$D - 2 \leq \frac{N}{P2^{\lambda-1}}$$

from which we obtain the final bound on λ:

$$\lambda \leq \log_2 \left(\frac{2N}{(D-2)P} \right). \tag{3.26}$$

For $N = 256$, $D = 8$, $P = 16$, for example, we find

$$\lambda \leq 5.$$

The bound given in (3.26) is not as restrictive as it may seem: Firstly, for the applications where a parallel code is called for, one normally has $N >> \max(P, D)$, secondly, in most practical wavelet application one takes λ to be a fixed small number, say $4-5$ (see [Str96]), and thirdly, should the need arise for large value of λ, one could use a sequential code for the last step of the FWT as these will not involve large amounts of data.

3.3.3 Multiple $1D$ Fast Wavelet Transform

The considerations from the previous section are still valid if we replace single elements with columns. This is a parallel version of MFWT. Figure-3.2 shows the data layout of the parallel MFWT algorithm.

The amount of necessary communication is now $M(D-2)$ elements instead of $D-2$, the columns of $\mathbf{X}$ are distributed block wise on the processors and the transformations of the rows

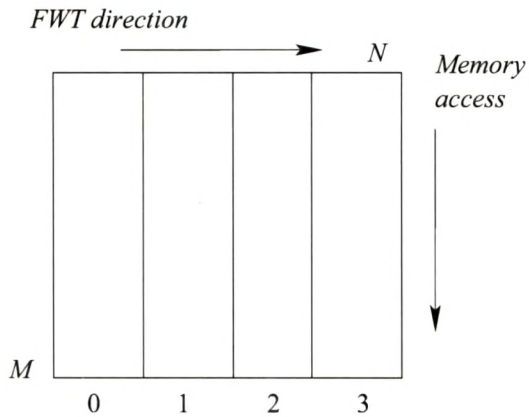

Figure 3.2: **Multiple FWT. Data are distributed column-wise on the processors. The FWT is organized row-wise in order to access data with stride one.**

of $\mathbf{X}$ involves the recursion formula corresponding to $\mathbf{X}\mathbf{W}_N^T$. The recursion formulas take the same form as in Subsection 3.3.3. The only difference from sequential case is that n is now given as in (3.18). We are now ready to give the algorithm for computing one step of multiple $1D$ FWT. The full transform is obtained by repeating this step for $i = 0, 1, \cdots, \lambda - 1$. The algorithm falls naturally in to the following three phases:

1. **Communication phase:** $D - 2$ columns are copied from the right neighbor as these are sufficient to complete all subsequent computations locally. We denote this column by the block $\bar{c}^i_{:,0:D-3}$.

2. **Fully local phase:** The interior of each block is transformed, possibly overlapping the communication process.

3. **Partially remote phase:** When the communication has completed, the remaining elements are computed using $\bar{c}^i_{:,l+2n-N/(p2^i)}$ whenever $l + 2n \geq N/(P2^i)$.

Algorithm: MFWT: level $i \rightarrow i + 1$
$S_i^P = \frac{N}{P2^i}$
$p =$ "my processor id" $\in [0 : P - 1]$

!———————————-
! Communication phase
!———————————-
send $\bar{c}^i_{:,0:D-3}$ **to** processor $\langle p - 1 \rangle_p$

receive $\bar{c}^i_{:,0:D-3}$ **from** processor $\langle p+1 \rangle_p$

!———————————————-

! Fully local phase

!———————————————-

for $n = 0 : S_i^P/2 - D/2$

$c^{i+1}_{:,n} = \sum_{l=0}^{D-1} a_l c^i_{:,l+2n} \quad ! \min(l+2n) = 0$

$d^{i+1}_{:,n} = \sum_{l=0}^{D-1} b_l c^i_{:,l+2n} \quad ! \min(l+2n) = S_i^P - 1$

end

!———————————————————————

! Partially remote phase

! communication must be finished finished at this point

!———————————————————————

for $n = S_i^P - D/2 + 1 : S_i^P/2 - 1$

! –

local part

! –

$c^{i+1}_{:,n} = \sum_{l=0}^{S_i^P-2n-1} a_l c^i_{:,l+2n} \quad ! \min(l+2n) = S_i^P - D + 2$

$d^{i+1}_{:,n} = \sum_{l=0}^{S_i^P-2n-1} b_l c^i_{:,l+2n} \quad ! \min(l+2n) = S_i^P - 1$

! –

Remote part, use $\bar{c}^j_{:,0:D-3}$

! –

$c^{i+1}_{:,n} = c^{i+1}_{:,n} + \sum_{l=S_i^P-2n}^{D-1} a_l \bar{c}^i_{:,l+2n-S_i^P} \quad ! \min(l+2n) = S_i^P$

$d^{i+1}_{:,n} = d^{i+1}_{:,n} + \sum_{l=S_i^P-2n}^{D-1} b_l \bar{c}^i_{:,l+2n-S_i^P} \quad ! \min(l+2n) = S_i^P + D - 3$

end

Performance model for the multiple $1D$ FWT

The purpose of this section is to focus on the impact of the proposed communication scheme on performance with particular regard to speed up and efficiency. We will consider the theoretical based achievable performance of the multiple $1D$ FWT algorithm. Recall that (3.6) can be computed using

$$F_{MFWT}(N) = 4DMN\left(1 - \frac{1}{2^{\lambda_N}}\right) \tag{3.27}$$

floating point operation. We emphasize the dependency on N because it denotes the dimension over which the problem is parallelized.

Let t_f be the average time it takes to compute one floating point operation on a given computer. Hence, the time needed to compute (3.6) sequentially is

$$T^0_{MFWT}(N) = F_{MFWT}(N)t_f \tag{3.28}$$

and the theoretical sequential performance becomes

$$R^0_{MFWT}(N) = \frac{F_{MFWT}(N)}{T^0_{MFWT}(N)} \tag{3.29}$$

In our proposed algorithm for computing (3.6), the amount of double precision numbers that must be communicated between adjacent neighbors at each step of the wavelet transform is $M(D-2)$ as described in Subsection 3.4.3. Let t_l be the time it takes to initiate the communication (latency) and t_d the time it takes to send one double precision number. Since there are λ steps in the wavelet transform, a simple model for the total communication time is

$$C_{MFWT} = \lambda\,(t_l + M(D-2)t_d) \tag{3.30}$$

Note that C_{MFWT} grows linearly with M but that it is independent of the number of processors P as well as the size of the second dimension N!.

Combining the expression for computation time and communication time, we obtain a model describing the total execution time on P processors ($P > 1$) as

$$T^P_{MFWT}(N) = \frac{T^0_{MFWT}(N)}{P} + C_{MFWT} \tag{3.31}$$

The performance of Parallel Algorithm formula for the speed of the MFWT algorithm has been discussed in [Nie98].

3.3.4 2D Fast Wavelet Transform

In this section, we will consider two approaches to parallelize the split algorithm for the $2D$ FWT as described in Section 3.2.4.

The first approach is similar to the way $2D$ FFT's can be parallelized (see [Heg96]) in that it uses the sequential multiple $1D$ FWT and a parallel transpose algorithm: we denote it the **replicated FWT**. The second approach makes use of the parallel $1D$ FWT described in Subsection 3.3.2 to avoid the parallel transposition. We denote this approach as the **communication-efficient FWT**. In both cases, we assume that the transform depth is the same in each dimension, i.e.

$$\lambda = \lambda_M = \lambda_N.$$

Then, we got from (2.39) and (3.7) the sequential execution time for the $2D$ FWT is

$$T^0_{FWT2}(N) = 2T^0_{MFWT}(N) \tag{3.32}$$

3.3.5 Replicated $2D$ Fast Wavelet Transform

The most straightforward way of dividing the work involved in the $2D$ FWT algorithm among a number of processors is to parallelize along the first dimension in $\mathbf{X}$, such that a sequence of $1D$ row transforms are executed independently on each processor. This is illustrated in Figure-3.3. Since we replicate independent row transforms on the processors we denote this approach the replicated FWT(RFWT) algorithm. Here it is assumed that the matrix $\mathbf{X}$ is distributed such that each processor receives the same number of consecutive rows of $\mathbf{X}$. The first and the last stages of Algorithm in Subsection 3.2.4 are thus done without any communication. However, the intermediate stage, the transposition, causes a substantial communication overhead. A further disadvantage of this approach is the fact that it reduces the maximal vector length available for vectorization from M to M/P (and from N to N/P). This is a problem for vector architectures such as the **CP200** as described in Subsection 3.3.3. A similar approach

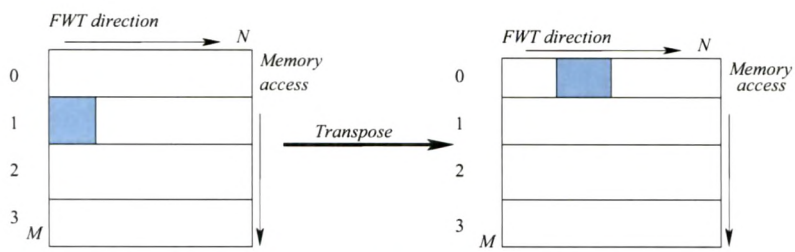

Figure 3.3: **Replicated FWT. The shaded block moves from processor 1 to 0.**

was adopted in [LS95] where a $2D$ FWT was implemented on the MasPar - a data parallel computer with 2048 processors. It was noted that **the transpose operations dominate the computation time** and a speedup of no more than 6 times relative to the best sequential program was achieved. A suitable parallel transpose algorithm needed for the replicated FWT is one that moves data in wrapped block diagonals as outlined in the next section.

Parallel transposition and data distribution

Assume that the rows of the matrix $\mathbf{X}$ are distributed over the processors, such that each processor gets M/P consecutive rows, and the transpose $\mathbf{X}^T$ is distributed such that each

processor gets N/P rows. Imagine that the part of matrix $\mathbf{X}$ that resides on each processor is split columnwise into P blocks, as suggested in Figure-3.4, then the blocks denoted by i are moved to processor i during transpose. In total each processor must send $P1$ blocks and each block contains M/p times N/P elements of $\mathbf{X}$. Hence, following the notation in Subsection 3.4.3, we get the model for communication time of a parallel transposition

$$C_{RFWT} = (P-1)\left(t_l + \frac{MN}{P^2}t_d\right) \tag{3.33}$$

Note that C_{RFWT} grows linearly with M, N and P (for P large).

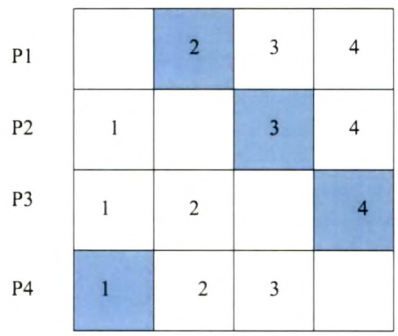

Figure 3.4: **Communication of blocks, first block-diagonal shaded.**

Performance model for the replicated FWT

We are now ready to derive a performance model for the replicated FWT algorithm. Using (3.32) and (3.33), we obtain the parallel execution time as

$$T^P_{RFWT}(N) = \frac{T^0_{FWT2}(N)}{P} + C_{RFWT}$$

and the theoretical speedup for the scaled problem $N = PN_1$ is

$$S^P_{RFWT}(PN_1) = \frac{P}{1 + C_{RFWT}/T^0_{FWT2}(N_1)} \tag{3.34}$$

3.3.6 Communication-efficient FWT

In this section, we combine the multiple $1D$ FWT described in Subsection 3.4.3 and the replicated FWT idea described in Subsection 3.5.1 to get a $2D$ FWT that combines the best of both

worlds. The first stage of Algorithm: Split-transpose given in Section 2.10 is computed using the parallel multiple $1D$ FWT as given in Algorithm of Subsection 3.4.3, so that the consecutive columns of $\mathbf{X}$ must be distributed to the processors. However, the last stage uses the layout from the replicated FWT, i.e. consecutive rows are distributed to the processors. This is illustrated in Figure-3.5 The main benefit using this approach is that the transpose step is done without

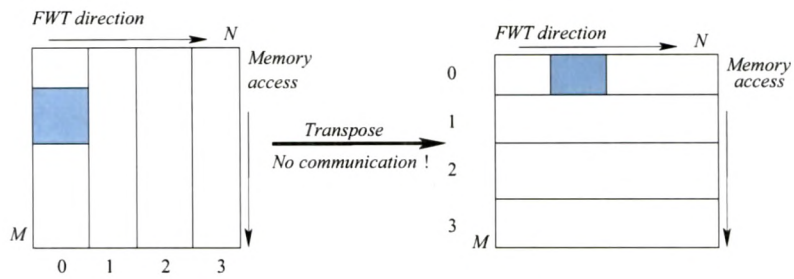

Figure 3.5: **Communication-efficient FWT. Data in shaded block stay on processor 0.**

any communication whatsoever. The only communication required is that of the multiple $1D$ FWT, namely the transmission of $M(D-2)$ elements between nearest neighbors, so most of the data stay on the same processor throughout the computations.The result will therefore be permuted in the N-dimension as described in Subsection 3.4.2 and ordered normally in the other dimension. We call this algorithm the communication-efficient FWT (CFWT).

The performance model for the communication-efficient FWT is a straightforward extension of the MFWT because the communication part is the same, so we get the theoretical speedup

$$S^P_{CFWT}(PN_1) = \frac{P}{1 + C_{MFWT}/T^0_{FWT2}(N_1)} \tag{3.35}$$

where C_{MFWT} and $T^0_{FWT2}(N_1)$ and as given in (3.30) and (3.32) respectively.

3.3.7 Conclusion

We have developed a new parallel algorithm for computing the $2D$ wavelet transform, the communication-efficient FWT. The new approach avoids the use of a distributed matrix transpose and performs significantly better than those algorithms that require such a transpose.