THE AUSTRALIAN NATIONAL UNIVERSITY

TR-CS-97-21

A Scalable Parallel 2D Wavelet Transform Algorithm

Ole Møller Nielsen and Markus Hegland

December 1997

Joint Computer Science Technical Report Series

Department of Computer Science
Faculty of Engineering and Information Technology

Computer Sciences Laboratory
Research School of Information Sciences and Engineering

This technical report series is published jointly by the Department of Computer Science, Faculty of Engineering and Information Technology, and the Computer Sciences Laboratory, Research School of Information Sciences and Engineering, The Australian National University.

Please direct correspondence regarding this series to:

Technical Reports
Department of Computer Science
Faculty of Engineering and Information Technology
The Australian National University
Canberra ACT 0200
Australia

or send email to:

`Technical.Reports@cs.anu.edu.au`

A list of technical reports, including some abstracts and copies of some full reports may be found at:

`http://cs.anu.edu.au/techreports/`

Recent reports in this series:

TR-CS-97-20 M. Hegland, S. Roberts, and I. Altas. *Finite element thin plate splines for surface fitting*. November 1997.

TR-CS-97-19 Xun Qu, Jeffrey Xu Yu, and Richard P. Brent. *Implementation of a portable-IP system for mobile TCP/IP*. November 1997.

TR-CS-97-18 Richard P. Brent. *Stability of fast algorithms for structured linear systems*. September 1997.

TR-CS-97-17 Brian Murphy and Richard P. Brent. *On quadratic polynomials for the number field sieve*. August 1997.

TR-CS-97-16 M. Manzur Murshed and Richard P. Brent. *Algorithms for optimal self-simulation of some restricted reconfigurable meshes*. July 1997.

TR-CS-97-15 Peter Strazdins. *Reducing software overheads in parallel linear algebra libraries*. July 1997.

A Scalable Parallel 2D Wavelet Transform Algorithm

Ole Møller Nielsen
uniomni@uni-c.dk

Department of Mathematical Modelling,
Technical University of Denmark and
UNI•C,
Danish Computing Centre for Research and Education,
Lyngby, Denmark

Markus Hegland
Markus.Hegland@anu.edu.au

Computer Sciences Laboratory, RSISE, ANU,
Canberra ACT 0200, Australia

December 16, 1997

Abstract

We present a new parallel 2D wavelet transform algorithm with minimal communication requirements. Data are transmitted between nearest neighbors only and the amount is independent of the problem size as well as the number of processors. An analysis of the theoretical performance shows that our algorithm is highly scalable approaching perfect speedup as the problem size is increased. This performance is realized in practice on the IBM SP2 as well as on the Fujitsu VPP300 where it will form part of the Scientific Software Library.

Keywords: Parallel algorithms, Wavelets, Fujitsu VPP300, IBM SP2

1 Introduction

Wavelets have generated a tremendous interest in both theoretical and applied mathematics over the past few years, and the wavelet transform in particular has proven to be an effective tool for e.g. numerical analysis [1] and image processing [2]. Problems from these areas are typically large and the wavelet transforms can be very time-consuming although the algorithmic complexity is proportional to the problem size. The use of parallel computers is one way of speeding up the wavelet transforms. Most of the previous work on parallel wavelet transforms concentrate on SIMD architectures such as the CM-200 ([6, 8, 9]) or MasPar ([7]), while very little has been done on MIMD architectures ([4]). It is normally not feasible to adapt SIMD algorithms to MIMD architectures because they are typically dependent on the data parallel architecture for which they were written: In [6], for example, the proposed algorithm is dependent on the ability to shift arrays fast in power-of-two steps, and in [9] the algorithm relies on a particular processor topology to be efficient. In [4] the communication pattern of a MIMD parallel wavelet transform algorithm is analyzed and it is shown that the asymmetry in the standard formulation of the wavelet transform leads to poor speedup.

We base our approach on the MIMD philosophy and the architectures we have in mind are distributed memory machines connected with dedicated crossbar networks.

The paper is organized as follows: In Section 2 we introduce the sequential 1D fast wavelet transform, discuss parallelization strategies and derive a performance model. In Section 3 we compare two algorithms for the 2D wavelet transform. The first is traditional in the sense that it divides the data such that a number of sequential 1D wavelet transforms are run in parallel without communication. This comes at the cost of one parallel transposition step. The second algorithm is new. It avoids the parallel transposition step by using results from Section 2. In Section 4 we report the results of implementations on the IBM SP2 and the Fujitsu VPP300 and verify that the new algorithm performs as predicted by the performance model.

2 The 1D Fast Wavelet Transform

Wavelets are basis functions in $L^2(\boldsymbol{R})$ that are particularly interesting because they lead to very good approximations using only a few terms in the wavelet expansions for most reasonable functions. This is a consequence of the fact that wavelets are well localized in both time and frequency and that they can locally yield exact representations of polynomials up to a certain degree. A spin-off from the wavelet theory is the Fast Wavelet Transform (FWT) which is a linear mapping of $\boldsymbol{R}^N$ onto $\boldsymbol{R}^N$ defined recursively by a sequence of fundamental linear mappings. The FWT is used to obtain the coefficients used in the wavelet expansions mentioned above. Because many of these are often small enough to be disregarded without introducing a large error, the FWT forms the basis of efficient data compression algorithms. See [13, 3, 10] for good expositions of wavelet analysis.

Let $N = 2^J$ and $\boldsymbol{c}^J$ be a given vector with elements $\{c_n^J\}_{n=0,1,\ldots,N-1}$. The recurrence formulas for the FWT are

$$c_n^{j-1} = \sum_{l=0}^{D-1} a_l c^j_{\langle l+2n\rangle_{2^j}} \tag{1}$$

$$d_n^{j-1} = \sum_{l=0}^{D-1} b_l c^j_{\langle l+2n\rangle_{2^j}} \tag{2}$$

for $n = 0, 1, \ldots, 2^{j-1} - 1$ and $j = J, J-1, \ldots, J-\lambda+1$. The expression $\langle l+2n\rangle_{2^j}$ denotes the modulus function defined such that $\langle l+2n\rangle_{2^j} \in [0, 2^j - 1]$ for all values of $l \in \boldsymbol{Z}$. λ is an integer between 0 and J that determines how many recursion steps to take, D is an even positive integer denoting the wavelet genus, and the numbers a_l, b_l for $l = 0, 1, \ldots, D-1$ are the wavelet filter coefficients. Each step of the wavelet transform splits a vector into two blocks as illustrated in (3). Here the number of steps is taken to be as large as possible, i.e. $\lambda = J = 4$. Note that once the elements d_n^{j-1} are computed, they are not modified in subsequent steps whereas the elements c_n^{j-1} are replaced at each step.

$$\begin{array}{c}
c_0^4 c_1^4 c_2^4 c_3^4 c_4^4 c_5^4 c_6^4 c_7^4 c_8^4 c_9^4 c_{10}^4 c_{11}^4 c_{12}^4 c_{13}^4 c_{14}^4 c_{15}^4 \\
\downarrow \\
c_0^3 c_1^3 c_2^3 c_3^3 c_4^3 c_5^3 c_6^3 c_7^3 \;\; d_0^3 d_1^3 d_2^3 d_3^3 d_4^3 d_5^3 d_6^3 d_7^3 \\
\downarrow \\
c_0^2 c_1^2 c_2^2 c_3^2 \;\; d_0^2 d_1^2 d_2^2 d_3^2 \;\; d_0^3 d_1^3 d_2^3 d_3^3 d_4^3 d_5^3 d_6^3 d_7^3 \\
\downarrow \\
c_0^1 c_1^1 \;\; d_0^1 d_1^1 \;\; d_0^2 d_1^2 d_2^2 d_3^2 \;\; d_0^3 d_1^3 d_2^3 d_3^3 d_4^3 d_5^3 d_6^3 d_7^3 \\
\downarrow \\
c_0^0 \;\; d_0^0 \;\; d_0^1 d_1^1 \;\; d_0^2 d_1^2 d_2^2 d_3^2 \;\; d_0^3 d_1^3 d_2^3 d_3^3 d_4^3 d_5^3 d_6^3 d_7^3
\end{array} \tag{3}$$

The choice of D and the filter coefficients comes from wavelet theory which is beyond the scope of this paper. Here it suffices to mention that the filters characterize the underlying wavelet uniquely and that they are designed such that elements in the vector $\boldsymbol{c}^{j-1}$ are weighted averages of elements of $\boldsymbol{c}^j$ while elements in

$\boldsymbol{d}^{j-1}$ represent the details lost in the mapping from $\boldsymbol{c}^j$ to $\boldsymbol{c}^{j-1}$. Furthermore the filter coefficients are chosen such that the linear mapping $\{\boldsymbol{c}^J\} \to \{\boldsymbol{c}^{J-1}, \boldsymbol{d}^{J-1}\}$ is invertible. Often it is even orthogonal.

The recursion can be carried out to the coarsest level, i.e. $\lambda = J$ where the overall average is represented by the single coefficient c_0^0. However, in practical applications λ is typically no larger than 5–7. The number of floating point operations needed to perform the 1D wavelet transform of a vector with N elements is

$$4DN\left(1 - \frac{1}{2^\lambda}\right)$$

Since the wavelet transform is a linear mapping of $\boldsymbol{R}^N$ onto $\boldsymbol{R}^N$, it can be represented by an $N \times N$ matrix $\boldsymbol{W}_N$. Let $\boldsymbol{x} = \boldsymbol{c}^J$ and $\boldsymbol{y}$ be the result of the recursion after λ steps, i.e.

$$\boldsymbol{y} = \begin{bmatrix} \boldsymbol{c}^{J-\lambda} \\ \boldsymbol{d}^{J-\lambda} \\ \boldsymbol{d}^{J-\lambda+1} \\ \vdots \\ \boldsymbol{d}^{J-1} \end{bmatrix}$$

then the wavelet transform can be expressed as

$$\boldsymbol{y} = \boldsymbol{W}_N \boldsymbol{x} \quad \text{or} \quad \boldsymbol{y}^T = \boldsymbol{x}^T \boldsymbol{W}_N^T \tag{4}$$

2.1 Parallelization strategies and distribution of data

We will now address the problem of distributing the work needed to compute (1) and (2) on P processors denoted by $p = 0, 1, \ldots, P-1$.

Assume for simplicity that N is a multiple of P and that the initial vector $\boldsymbol{x}$ is distributed such that each processor receives the same number of consecutive elements. This means that processor p holds the elements

$$\{c_n^J\}_n, \; n = p\frac{2^J}{P}, p\frac{2^J}{P} + 1, \ldots, (p+1)\frac{2^J}{P} - 1$$

It is not trivial how to chose the optimal distribution of $\boldsymbol{y}$ and the intermediate vectors. In fact, this choice is crucial to the performance of a parallel FWT.

We consider first the data layout suggested by the serial algorithm as shown in (3). For $P = 2$, $J = 4$, $\lambda = 4$ the situation is as follows

$p = 0$	$p = 1$
$c_0^4 c_1^4 c_2^4 c_3^4 c_4^4 c_5^4 c_6^4 c_7^4$	$c_8^4 c_9^4 c_{10}^4 c_{11}^4 c_{12}^4 c_{13}^4 c_{14}^4 c_{15}^4$
$\downarrow$	$\downarrow$
$c_0^3 c_1^3 c_2^3 c_3^3 c_4^3 c_5^3 c_6^3 c_7^3$	$d_0^3 d_1^3 d_2^3 d_3^3 d_4^3 d_5^3 d_6^3 d_7^3$
$\downarrow$	
$c_0^2 c_1^2 c_2^2 c_3^2 d_0^2 d_1^2 d_2^2 d_3^2$	$d_0^3 d_1^3 d_2^3 d_3^3 d_4^3 d_5^3 d_6^3 d_7^3$
$\downarrow$	
$c_0^1 c_1^1 d_0^1 d_1^1 d_0^2 d_1^2 d_2^2 d_3^2$	$d_0^3 d_1^3 d_2^3 d_3^3 d_4^3 d_5^3 d_6^3 d_7^3$
$\downarrow$	
$c_0^0 d_0^0 d_0^1 d_1^1 d_0^2 d_1^2 d_2^2 d_3^2$	$d_0^3 d_1^3 d_2^3 d_3^3 d_4^3 d_5^3 d_6^3 d_7^3$

It is seen that distributing the results of each transform step evenly across a number of processors will result in a poor load balancing because each step works with one half of the previous vector only. The processors containing parts that are finished early will then be idle in the subsequent transform steps. 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 steps

this communication will take place among the active processors only. This kind of layout was used in [4] where it was observed that optimal load balancing could not be achieved and also in [9] where the global communication was treated by organizing the processors of a connection machine (CM-2) in a pyramid structure.

However, we can obtain perfect load balancing and avoid global communication by introducing another ordering of the resulting and intermediate vectors. The recursion formulas are the same as in (1) and (2) but processor p will now hold the elements

$$\{c_n^{j-1}\}_n \text{ and } \{d_n^{j-1}\}_n$$

where

$$\begin{aligned} n &= p\frac{2^{j-1}}{P}, p\frac{2^{j-1}}{P}+1, \ldots, (p+1)\frac{2^{j-1}}{P}-1 \\ j &= J, J-1, \ldots, J-\lambda+1 \end{aligned} \tag{5}$$

with $N = 2^J$ as previously.
Let for example $P = 2$, $J = 4$, $\lambda = 3$. The situation is now

$p=0$	$p=1$
$c_0^4 c_1^4 c_2^4 c_3^4 c_4^4 c_5^4 c_6^4 c_7^4$	$c_8^4 c_9^4 c_{10}^4 c_{11}^4 c_{12}^4 c_{13}^4 c_{14}^4 c_{15}^4$
$\downarrow$	$\downarrow$
$c_0^3 c_1^3 c_2^3 c_3^3 d_0^3 d_1^3 d_2^3 d_3^3$	$c_4^3 c_5^3 c_6^3 c_7^3 d_4^3 d_5^3 d_6^3 d_7^3$
$\downarrow$	$\downarrow$
$c_0^2 c_1^2 d_0^2 d_1^2 d_0^3 d_1^3 d_2^3 d_3^3$	$c_2^2 c_3^2 d_2^2 d_3^2 d_4^3 d_5^3 d_6^3 d_7^3$
$\downarrow$	$\downarrow$
$c_0^1 d_0^1 d_0^2 d_1^2 d_0^3 d_1^3 d_2^3 d_3^3$	$c_1^1 d_1^1 d_2^2 d_3^2 d_4^3 d_5^3 d_6^3 d_7^3$

Note that this result is a permutation of the result in (3) because each processor essentially performs a *local* wavelet transform of its data. However, the ordering given in (3) is by no means intrinsic to the FWT so this permutation is not a disadvantage at all. Rather, one might argue that 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 the data.

Note also from the example that the local transforms have reached their ultimate form on each processor after only three steps and that it would be infeasible to continue the recursion further (i.e. splitting $\{c_0^1, c_1^1\}$ into $\{c_0^0, d_0^0\}$) because that would lead to load imbalance for the same reasons as for the algorithm mentioned above. This discontinuation of the recursion puts an upper bound on λ:

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

However, it turns out that this bound has to be even more restrictive in order to avoid excessive communication. We will return to this in Section 2.2.

2.2 Communication

Consider the computations done by processor p on a row vector as indicated in Figure 1. The quantities in equation (1) and (2) can be computed without communication provided that the index $l+2n$ does not refer to elements on other processors, i.e.

$$\begin{aligned} l+2n &\leq (p+1)\frac{2^j}{P}-1 \\ n &\leq (p+1)\frac{2^{j-1}}{P}-\frac{l+1}{2} \end{aligned} \tag{6}$$

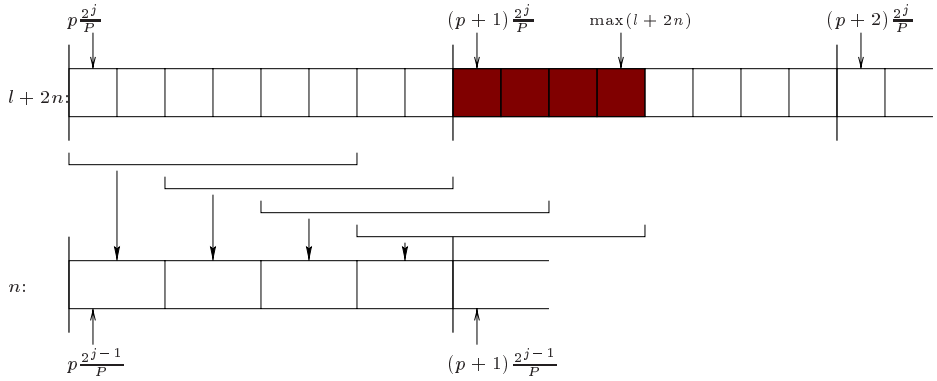

Figure 1: Computations on processor p involve $D-2$ elements from processor $p+1$. Here $D=6$ and $2^j/P=8$. The lines of width D indicate the filters as they are applied for different values of n.

A sufficient condition (independent of l) for this is

$$\boxed{n \leq (p+1)\frac{2^{j-1}}{P} - \frac{D}{2}} \tag{7}$$

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

For a fixed $n > (p+1)2^{j-1}/P - D/2$ computations are still local as long as (6) is fulfilled, i.e. when

$$\boxed{l \leq (p+1)\frac{2^j}{P} - 2n - 1} \tag{8}$$

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 (1), (2), and (5)) is

$$\max(l+2n) = (p+1)\frac{2^j}{P} + D - 3 \tag{9}$$

The smallest value of $l+2n$ for which communication is necessary is also found from (6):

$$(p+1)\frac{2^j}{P} - 1$$

Subtracting this quantity from (9) we find that exactly $D-2$ elements must be communicated to processor p at each step of the FWT as indicated in Figure 1.

2.2.1 A tighter bound on λ

We will not allow situations where processor p needs data from processors other than its right neighbor $\langle p+1 \rangle_P$ so we impose the additional restriction

$$\begin{aligned} \max(l+2n) &\leq (p+2)\frac{2^j}{P} - 1 \\ (p+1)\frac{2^j}{P} + D - 3 &\leq (p+2)\frac{2^j}{P} - 1 \\ D-2 &\leq \frac{2^j}{P} \end{aligned} \tag{10}$$

We then rewrite (10) in terms of the transform depth λ by using the smallest value for j, i.e. $j = \log_2 N - \lambda + 1$

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

from which we obtain the final bound on λ:

$$\boxed{\lambda \leq \log_2 \left(\frac{2N}{(D-2)P} \right)} \tag{11}$$

The bound given in (11) is not as restrictive as it may seem: Firstly, for the applications where a parallel code is called for, one normally has $N >> \max(P, D)$ and, secondly, in most practical wavelet applications one takes λ to be a fixed small number as mentioned in Section 2.

2.3 The multiple 1D FWT

Consider a matrix $\boldsymbol{X} \in \boldsymbol{R}^{M,N}$. Applying the 1D wavelet transform to every row of $\boldsymbol{X}$ is a simple but useful generalization which we will henceforth refer to as the **multiple 1D FWT**. This algorithm corresponds to the expression

$$\boldsymbol{X}\boldsymbol{W}_N^T \tag{12}$$

where $\boldsymbol{W}_N$ is defined as in (4). Since there are M rows, the number of floating point operations needed are $4DMN\left(1 - 1/2^{\lambda}\right)$. We assume that the matrix $\boldsymbol{X}$ is stored such that consecutive elements in each column are located in consecutive positions in memory. Then the multiple 1D FWT can be implemented such that all elements are accessed with stride-one which is the optimal scheme on most computer architectures.

The considerations from the previous section are still valid if we replace single elements with columns: The amount of necessary communication is $M(D-2)$ elements instead of $D-2$, the columns of $\boldsymbol{X}$ are distributed block-wise on the processors, and the transformation of the rows of $\boldsymbol{X}$ involves the recursion formulas corresponding to $\boldsymbol{X}\boldsymbol{W}_N^T$. The recursion formulas now take the form:

$$\begin{aligned} c_{m,n}^{j-1} &= \sum_{l=0}^{D-1} a_l c_{m,\langle l+2n \rangle_{2^j}}^{j} \\ d_{m,n}^{j-1} &= \sum_{l=0}^{D-1} b_l c_{m,\langle l+2n \rangle_{2^j}}^{j} \end{aligned} \qquad \begin{aligned} j &= J, J-1, \ldots, J-\lambda+1 \\ m &= 0, 1, \ldots, M-1 \\ n &= 0, 1, \ldots, 2^{j-1}-1 \end{aligned}$$

2.4 Algorithm

We are now ready to give the algorithm for computing one step of the multiple 1D FWT. The full transform is obtained by repeating this step for all j. The algorithm falls naturally in the following three phases:

1. **Communication phase:** $D-2$ columns of $\boldsymbol{X}$ are copied from the right neighbor as these are sufficient to complete all subsequent computations locally. We denote these columns by the block $\bar{c}_{:,0:D-3}^{j}$.
2. **Fully local phase:** The bulk of the transform is computed, possibly overlapping the communication process.
3. **Partially remote phase:** When the communication has completed, the remaining elements are computed using $\bar{c}_{:,l+2n-2^j/P}^{j}$ whenever $l + 2n > 2^j/P$.

Multiple 1D FWT algorithm (level $j \to j-1$):

$S = 2^j/P$
$p =$"my processor id" $\in [0 : P-1]$

!———————————
! **Communication phase**
!———————————
send $c^j_{:,0:D-3}$ **to** processor $\langle p-1\rangle_P$
receive $\bar{c}^j_{:,0:D-3}$ **from** processor $\langle p+1\rangle_P$

!———————————
! **Fully local phase, cf (7)**
!———————————
for $n = 0 : S/2 - D/2$
$\quad c^{j-1}_{:,n} = \sum_{l=0}^{D-1} a_l c^j_{:,l+2n}$! $\min(l+2n) = 0$
$\quad d^{j-1}_{:,n} = \sum_{l=0}^{D-1} b_l c^j_{:,l+2n}$! $\max(l+2n) = S-1$
end

!———————————————————————————
! **Partially remote phase**
! **communication must be finished at this point**
!———————————————————————————
for $n = S/2 - D/2 + 1 : S/2 - 1$
$\quad$!———————————
$\quad$! **Local part, cf (8)**
$\quad$!———————————
$\quad c^{j-1}_{:,n} = \sum_{l=0}^{S-2n-1} a_l c^j_{:,l+2n}$! $\min(l+2n) = S-D+2$
$\quad d^{j-1}_{:,n} = \sum_{l=0}^{S-2n-1} b_l c^j_{:,l+2n}$! $\max(l+2n) = S-1$

$\quad$!———————————————
$\quad$! **Remote part, use** $\bar{c}^j_{:,1:D-2}$
$\quad$!———————————————
$\quad c^{j-1}_{:,n} = \sum_{l=S-2n}^{D-1} a_l \bar{c}^j_{:,l+2n-S}$! $\min(l+2n) = S$
$\quad d^{j-1}_{:,n} = \sum_{l=S-2n}^{D-1} b_l \bar{c}^j_{:,l+2n-S}$! $\max(l+2n) = S+D-3$
end

2.5 Performance model

The purpose of this section is to focus on the impact that the proposed communication scheme will have on performance with particular regard to speedup and efficiency. We will consider the theoretically best achievable performance of the multiple 1D FWT algorithm.

Recall that (12) can be computed using

$$F(N) = 4DMN\left(1 - \frac{1}{2^\lambda}\right) \tag{13}$$

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

Let t_f be the time it takes to compute one floating point operation on a given computer. Hence, the time needed to compute (12) sequentially is

$$T_0(N) = F(N)t_f \tag{14}$$

and the theoretical sequential performance in terms of floaing point operations per second (flop/s) becomes

$$R_0(N) = \frac{F(N)}{T_0(N)} = t_f^{-1} \tag{15}$$

In our proposed algorithm for computing (12) the amount of double precision numbers that must be communicated between all adjacent neighbors at each step of the wavelet transform is $M(D-2)$ as described in Section 2.3. 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 once the communication has begun. There are λ steps in the wavelet transform so a simple model for the total communication time is

$$C = \lambda(t_l + M(D-2)t_d) \tag{16}$$

Note that C grows linearly with M but that it is independent of the number of processors P as well as 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

$$\boxed{T_P(N) = \frac{T_0(N)}{P} + C}$$

and the performance of the parallel algorithm is

$$R_P(N) = \frac{F(N)}{T_P(N)} \tag{17}$$

The expressions for performance in (15) and (17) lead to a formula for the speedup of the parallel wavelet transform algorithm.

$$S_P(N) = \frac{T_0(N)}{T_P(N)} = \frac{P}{1 + \frac{PC}{T_0(N)}}$$

The **efficiency** of the parallel implementation is defined as the speedup per processor and we have

$$E_P(N) = \frac{S_P(N)}{P} = \frac{1}{1 + \frac{PC}{T_0(N)}} \tag{18}$$

It can be seen from (18) that for N held constant, the efficiency will decrease when the number of processors P is increased.

If we instead keep the amount of work constant on each processor, then we can express the speedup and efficiency, not in terms of how fast a given problem is solved, but in terms of how efficiently the implementation solves a problem which scales with the number of processors. Let N_1 be the constant size of a problem which is solved on one processor regardless of the total number of processors allocated to the algorithm. Then the total problem size becomes $N = PN_1$ and we find from (13) and (14) that $T_0(PN_1) = PT_0(N_1)$ because the computational work of the FWT is linear in N. This means in turn that the efficiency for the scaled problem takes the form

$$E_P(PN_1) = \frac{1}{1 + \frac{PC}{PT_0(N_1)}} = \frac{1}{1 + \frac{C}{T_0(N_1)}}$$

P has been eliminated entirely from the efficiency measure making it constant with respect the number of processors. In other words we have demonstrated that when the computational volume stays constant on each processor this algorithm has the property that the scaled efficiency stays constant with respect to the number of processors. The multiple 1D FWT algorithm is therefore fully scalable.

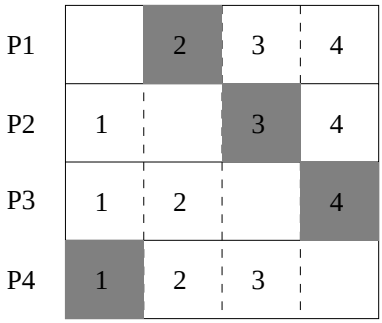

Figure 2: Communication of blocks, first block-diagonal shaded.

3 The 2D wavelet transform

Using the notation introduced in Section 2 the 2D wavelet transform is defined by the matrix product

$$\mathbf{Y} = \mathbf{W}_M \mathbf{X} \mathbf{W}_N^T. \tag{19}$$

The number of floating point operations needed is

$$F(N) = 4DMN\left(2 - \frac{1}{2^{\lambda_M}} - \frac{1}{2^{\lambda_N}}\right) \tag{20}$$

where λ_M and λ_N are the transform depths in the first and second dimensions, respectively.

The expression $\mathbf{X}\mathbf{W}_N^T$ leads to vector operations on vectors of length M and stride-one data access as decribed in Section 2.3. This is not the case for the expression $\mathbf{W}_M\mathbf{X}$, because it consists of a collection of columnwise 1D transforms which do not access the memory as efficiently [11]. We therefore rewrite the matrix product (19) as

$$\boldsymbol{Y}^T = \left(\boldsymbol{X}\boldsymbol{W}_N^T\right)^T \boldsymbol{W}_M^T$$

yielding efficient memory access on each processor at the cost of one transpose step. This algorithm can be implemented on a parallel computer similar to parallel 2D FFTs [5]. The parallel transpose algorithm needed for that is one that moves data in wrapped block diagonals as outlined in the next section.

3.1 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 that 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 column-wise into P blocks, as suggested in Figure 2, then the blocks denoted by i are moved to processor i during the transpose. In total each processor must send $P-1$ blocks and each block contains M/P times N/P elements of $\mathbf{X}$. Hence, following the notation in Section 2.5, we get the model for communication time of a parallel transposition

$$V = (P-1)\left(t_l + \frac{MN}{P^2}t_d\right) \tag{21}$$

Note that V grows linearly with M, N and P (for P large).

3.2 The replicated FWT

The most straightforward way of dividing the 2D FWT algorithm among a number of processors is to parallelize along the first dimension, such that a sequence of 1D row transforms are executed independently

on each processor. This is illustrated in Figure 3. We denote this approach the **replicated FWT** algorithm. Here it is assumed that the matrix $\mathbf{X}$ is distributed such that each processor receives the same number of

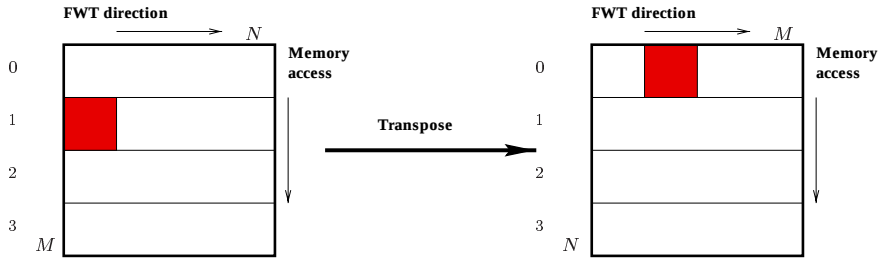

Figure 3: Replicated FWT. The shaded block moves from processor 1 to 0.

consecutive rows of $\mathbf{X}$. The first and the last stages are thus done without any communication. However, the intermediate stage, the transposition, causes a substantial communication overhead as described in Section 3.1. A further disadvantage of this approach is 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 Fujitsu VPP300 as mentioned in section 2.3.

A similar approach was adopted in [7] 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.

We are now ready to derive a performance model for the replicated FWT algorithm. Assume that the transform level is the same in each dimension, i.e. $\lambda = \lambda_M = \lambda_N$. Then we get from (14) and (20) that the sequential execution time is $2T_0(N)$. Hence, using (21), the parallel execution time is found as

$$T_P(N) = \frac{2T_0(N)}{P} + V$$

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

$$S_P(PN_1) = \frac{P}{1 + \frac{V}{2T_0(N_1)}} \tag{22}$$

We will return to this expression in Section 3.4.

3.3 The communication-efficient FWT

In this section we combine the multiple 1D FWT described in Section 2.3 and the replicated FWT idea described in Section 3.2 to get a 2D FWT that combines the best of both worlds. The first stage of the 2D FWT is computed using the multiple 1D FWT, so 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 4.

The main benefit using this approach is that the transpose step is done without 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 Section 2.1 and ordered normally in the other. We call this algorithm the **communication-efficient FWT**.

The performance model for the communication-efficient FWT is a straightforward extension of the multiple 1D FWT because the communication part is the same. We assume as before that $\lambda = \lambda_M = \lambda_N$ and get the theoretical speedup

$$S_P(PN_1) = \frac{P}{1 + \frac{C}{2T_0(N_1)}} \tag{23}$$

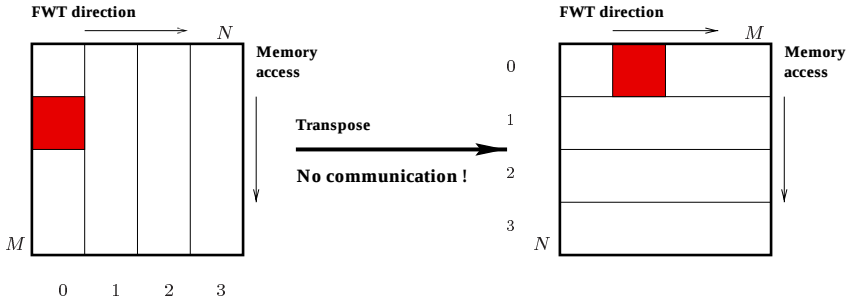

Figure 4: Communication-efficient FWT. Data in shaded block stay on processor 0.

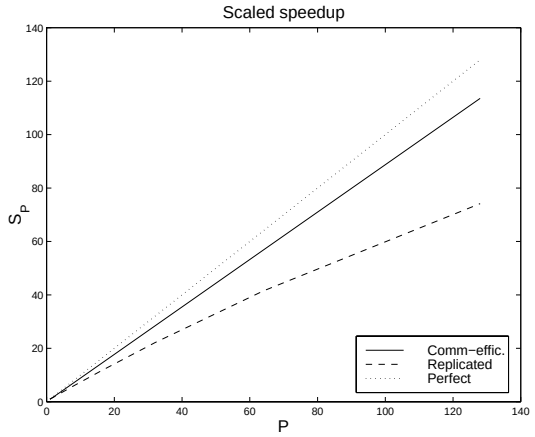

Figure 5: The theoretical scaled speedup of the replicated FWT algorithm and the communication-efficient FWT shown together with the line of perfect speedup. The predicted performances correspond to a problem with $M = 512$, $N = 128$, $D = 12$ and the characteristic parameters were measured on an IBM SP2 to be $t_d = 0.2\ \mu s$, $t_l = 200\ \mu s$, $t_f = 6\ ns$. The performance of the communication-efficient FWT is much closer to the line of perfect speedup than the performance of the replicated FWT and its slope remains constant.

where C is as given in (16).

3.4 Comparison of the two approaches

We can now compare the theoretical performance of the replicated FWT algorithm (22) and our new communication-efficient approach (23) with respect to their dependency on P and N_1, respectively.

In case of the communication-efficient FWT the ratio $C/2/T_0(N_1)$ is constant with respect to P whereas the corresponding ratio for the replicated FWT in (22) goes as $\mathcal{O}(P)$:

$$\frac{V}{2T_0(N_1)} \approx \frac{(P-1)t_l + \frac{P-1}{P^2}MN_1t_d}{8DMN_1} = \mathcal{O}(P)$$

This means that the efficiency of the replicated FWT will deteriorate as P grows while it will stay constant for the communication-efficient FWT. The corresponding speedups are shown in Figure 5.

When P is fixed and the problem size N_1 grows, then $C/2/T_0(N_1)$ goes to zero, which means that the scaled efficiency of the communication-efficient FWT will approach the ideal value 1. For the replicated

P	N	Mflop/s
1	128	169
2	256	301
4	512	596
8	1024	1190
16	2048	2379
32	4196	4747

Table 1: Communication-efficient FWT on the SP2. $N \propto P$, $M = 1024$, $D = 10$.

P	N	Mflop/s
1	512	1278
2	1024	2551
4	2048	5058
8	4096	10186

Table 2: Communication-efficient FWT on the VPP300. $N \propto P$, $M = 512$, $D = 10$.

FWT the corresponding ratio approaches a positive constant as N_1 grows:

$$\frac{V}{2T_0(N_1)} \quad \rightarrow \quad \frac{(P-1)t_d}{8DP^2} \text{ for } N_1 \rightarrow \infty$$

This means that the scaled efficiency is bounded by a constant less than one – no matter how large the problem size. The asymptotic scaled efficiencies of the two algorithms are summarized below

	$P \rightarrow \infty$	$N_1 \rightarrow \infty$
Replicated FWT:	$\frac{1}{1+\mathcal{O}(P)}$	$\frac{1}{1+\frac{(P-1)t_d}{8DP^2}}$
Communication-efficient FWT:	$\frac{1}{1+\frac{C}{2T_0(N_1)}}$	1

4 Numerical experiments

We have implemented the communication-efficient FWT on two different MIMD computer architectures, namely the IBM SP2 and the Fujitsu VPP300. On the SP2 we used MPI for the parallelism whereas the proprietary VPP Fortran was used on the VPP300.

The measured performances on the IBM SP2 are shown in Table 1. It is seen the performance scales well with the number of processors and further that it agrees with the predicted speedup as shown in Figure 6.

On the Fujitsu VPP300 vector computer great care was taken to get good vector performance on one node. The problem with $D = 10$, $M = N = 512$ yields about 58% of the theoretical peak performance which is 2.2 Gflop/s. With increased problem sizes we observed as much as 80% of the peak performance ([12]). The parallel performance is shown in Table 2. We have not estimated the characteristic numbers t_l, t_d, t_f for this machine, but it is nevertheless clear that the performance scales almost perfectly with the number of processors also in this case.

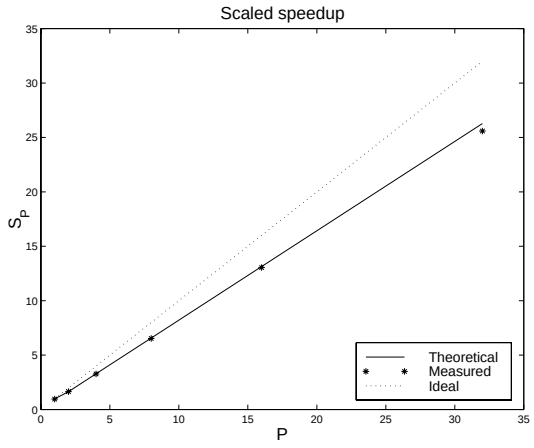

Figure 6: Scaled speedup, communication-efficient FWT (IBM SP2). It is seen from these graphs that the theoretical performance model does give a realistic prediction of the actual performance.

5 Conclusion

We have developed a parallel algorithm for computing the 2D wavelet transform, the communication-efficient FWT.

Our new approach avoids the use of a distributed matrix transpose and performs significantly better than algorithms that require such a transpose. This is due to the fact that the communication volume of a parallel transpose is larger than necessary for computing the 2D FWT. The communication-efficient FWT is optimal in the sense the scaled efficiency is independent of the number of processors and that it approaches 1 as the problem size is increased. Implementations on the Fujitsu VPP300 and the IBM SP2 confirm that our algorithm is highly scalable.

Acknowledgements

The work presented here has partly been done in the Area 4 joint project of ANU and Fujitsu Japan. Some of the resulting software is expected to be included in Fujitsu's SSL II/VPP Scientific Software Library.

References

[1] G. Beylkin, R. Coifman, and V. Rohklin. Wavelets in numerical analysis. In Mary Beth Ruskai et al., editors, *Wavelets and their applications*, pages 181–210. Jones and Bartlett, 1992.

[2] Charles K. Chui. *Wavelets: A Mathematical Tool for Signal Analysis.* SIAM Monographs on Mathematical Modeling and Computation. SIAM, 1997.

[3] Ingrid Daubechies. *Ten Lectures on Wavelets.* SIAM, 1992.

[4] B. K. Das, R. N. Mahapatra, and B. N. Chatterli. Modelling of wavelet transform on multistage interconnection network. In *Proceedings of the Australasian Conference on Parallel and Real-Time Systems*, pages 134–142, Freemantle, Western Australia, 28-29 September 1995.

[5] Markus Hegland. *Real and Complex Fast Fourier Transforms on the Fujitsu VPP500.* Parallel Computing, 22, pp. 539-553, 1996.

[6] Mats Holmström. Parallelizing the fast wavelet transform. *Parallel Computing*, 11(21):1837–1848, April 1995.

[7] Robert Lang and Andrew Spray. The 2d wavelet transform on a massively parallel machine. In *Proceedings of the Australasian Conference on Parallel and Real-Time Systems*, pages 325–332, Freemantle, Western Australia, 28-29 September 1995.

[8] A. Lega, H. Scholl, J.-M. Alimi, A. Bijaoui, and P. Bury. A parallel algorithm for structure detection based on wavelet and segmentation analysis. *Parallel Computing*, 21:265–285, April 1995.

[9] Jian Lu. Parallelizing mallat algorithm for 2-d wavelet transforms. *Information Processing Letters*, 45:255–259, April 1993.

[10] Yves Meyer. *Wavelets: Algorithms and Applications.* SIAM, 1993.

[11] Ole Møller Nielsen and Markus Hegland. *A Two-Dimensional Fast Wavelet Transform for the Fujitsu VP2200.* In preparation

[12] Ole Møller Nielsen, Gavin Mercer, and Markus Hegland. Vector-parallel fast wavelet transforms. In *PCW97 - Parallel Computing Workshop 1997*, Australian National University, Canberra, ACT, 25-26 September 1997.

[13] Gilbert Strang and Truong Nguyen. *Wavelets and Filter Banks.* Wellesley-Cambridge Press, 1996.