libWavelet
A C Library for Parallel Wavelet Transform

Jakob Dalsgaard
c958312

Jakob Østergaard
c958612

July 3rd 1998

Abstract

Recently, the Wavelet Transform has had it's breakthrough, with the appearence of the so-called *Daubechies*. The wavelet theory developed in connection with the intriguing but otherwise not very usefull *Haar* wavelet, could now be used with the *Daubechies* wavelets, suddenly allowed the wavelet theory to move out from it's mathematical hideout, entering the world of real applications. Wavelet Transform is now a hot topic especially in the area of signal processing, thus, the need for an efficient and easy to use Wavelet Transform Library exists.

Contents

Preface

It is the goal of this project to implement an efficient and portable library for performing Forward and Inverse Wavelet Transform in one and two dimensions.

Specifically we are to implement the algorithms known as the Fast Wavelet Transform, and it's Inverse Fast Wavelet Transform. We will provide theese routines in a portable `C` library, hopefully allowing the end user to implement powerfull applications of the Wavelet Transform, whithout spending too much precious time doing parallelization and porting.

The library shall provide the aforementioned routines in both serial, shared memory and massively parallel variants. Providing a common interface to these routines will allow the users to run wavelet transforms on stand-alone workstations, multiprocessor computer systems (both workstations and supercomputers), and cluster systems (clusters of workstations, or massively parallel supercomputers).

We plan to implement the discrete two dimensional wavelet transform, developed by Ole Møller Nielsen for the Fujitsu Vector Parallel computer. The implementation will have to run on at least the HP-9000 computer systems in the DTU G-Databar, and on the SGI Origin 2000 shared memory supercomputer located at UNI-C Aarhus; this again implies that we will have to implement both distributed- and shared memory variants of the transforms.

A performance model will be set up for the transforms wherever possible, and the results held against real world measurements. This is a good way of finding performance problems in the implementation, as well as identifying bottleneck problems in the hardware.

The library is available from the homepage:

`http://www.gbar.dtu.dk/~c958612/libWavelet/`

Lyngby July 3rd 1998
Jakob Østergaard & Jakob Dalsgaard

Chapter 1

Multi Resolution Analysis

The Multi Resolution Analysis (MRA) is based upon four axioms which lead to a nice and useful theory for dealing with wavelets. The notation and order of these axioms is taken from [OMN98, Sec 2.1].

We would like to use wavelets to approximate arbitrary functions. To do this we define approximation spaces V_j ordered as:

axiom a:

$$\{0\} \subset \cdots \subset V_{-1} \subset V_0 \subset V_1 \cdots \subset L^2(\mathbf{R}) \tag{1.1}$$

The closure of all these approximation-spaces is the set of all Lebesgue integrable functions, which is just about any function we would like to approximate:

axiom b:

$$\overline{\bigcup_{j=-\infty}^{\infty} V_j} = L^2(\mathbf{R}) \tag{1.2}$$

In other words; if we approximate a function $f \in L^2(\mathbf{R})$ in the approximation-space V_j then this approximation will converge to f as $j \to \infty$.

We start by defining a basis for the approximation space V_0:

axiom c:

$$\{\phi(x-k)\}_{k\in\mathbf{Z}} \text{ is an orthonormal basis for } V_0 \tag{1.3}$$

i.e. the translations of a function ϕ, which we call the scaling function, by integers, form an orthonormal basis in the basic approximation space V_0. If we have a function $f \in V_0$ then this function can be exactly represented using (1.3).

Through the scaling axiom of our theory we get functions belonging to other approximation spaces:

axiom d:

$$f \in V_j \Leftrightarrow f(2\cdot) \in V_{j+1} \tag{1.4}$$

Hence a function, and therefore also a scaling function, moves to an approximation space of finer granularity by being rescaled by 2. This property will be used to define orthonormal basises for other approximation spaces than V_0.

We can now express a function $f \in L^2(\mathbf{R})$ as a combination of scaling functions in V_j, the higher j, the better approximation. But the big advantage of using wavelets is the wavelet-spaces; these spaces W_j are defined as:

$$V_{j+1} = V_j \oplus W_j \tag{1.5}$$

And using our V_{J_0} as our origin we get:

$$V_J = V_{J_0} \oplus \left(\bigoplus_{j=J_0}^{J-1} W_j \right) \tag{1.6}$$

This equation is actually quite central to the approximation theory. If we have a function $f \in L^2(\mathbf{R})$ which we wish to approximate in the space V_J, we can express the approximation in terms of basis-functions of V_J. But what if this approximation space is too detailed for our use – what if less detail would suffice ?

If we can do with a coarser approximation, we might choose to use the scaling functions that are basis-functions of V_{J_0} and then add levels of details with terms of basis functions of W_j, $j > J_0$ until we reach our desired accuracy.

1.1 Using the axioms

The orthonormality of the basis-functions ϕ (*axiom c*, equation (1.3)) can mathematically be written as:

$$\|\phi\|_2 \equiv \left(\int_{-\infty}^{\infty} |\phi(x)|^2 dx \right)^{1/2} = 1 \tag{1.7}$$

Say that $\phi(x)$ is the scaling function for V_0, using the scaling property, *axiom d*, (1.4) we find a scaling function $\phi_j(x) \in V_j$ which has to be part of an orthonormal basis for V_j

$$\begin{aligned} \int_{-\infty}^{\infty} |\phi(2^j x - k)|^2 dx &= 2^{-j} \int_{-\infty}^{\infty} |\phi(y)|^2 dy \\ &= 2^{-j} \|\phi\|_2^2 \\ &= 2^{-j} \end{aligned}$$

Done with the substitution $y = 2^j x - k$.

This is not good, since we like $\{\phi(2^j x - k)\}_{k \in \mathbf{Z}}$ to be an *orthonormal* basis for V_j, i.e. with a squared norm of 1. Hence we normalize the scaling functions to use it for orthonormal basises in the other approximation spaces:

$$\{2^{j/2}\phi(2^j x - k)\}_{k \in \mathbf{Z}} \text{ is an orthonormal basis for } V_j$$

Furthermore we get that:

$$\{2^{j/2}\psi(2^j x - k)\}_{k \in \mathbf{Z}} \text{ is an orthonormal basis for } W_j$$

To get hold of all these scaling and wavelet functions at different scales and translations we adopt the same naming conventions as in [OMN98]:

$$\begin{aligned} \phi_{j,k}(x) &= 2^{j/2}\phi(2^j x - k) \\ \psi_{j,k}(x) &= 2^{j/2}\psi(2^j x - k) \end{aligned}$$

1.2 Using the functions

It is evident that if we have a function $f \in V_J$ then it can the expressed as a linear combination of a orthonormal basis of V_J, such as:

$$f(x) = \sum_{l=-\infty}^{\infty} c_{J,l}\phi_{J,l}(x), x \in \mathbf{R}$$

Where $c_{J,l}$ is the coefficient of the $(J,l)^{\text{th}}$ scaling function. But since:

$$V_J = V_{J_0} \oplus \left(\bigoplus_{j=J_0}^{J-1} W_j \right)$$

then the very same function could be written as:

$$f(x) = \sum_{l=-\infty}^{\infty} c_{j_0}\phi(x) + \sum_{j=J_0}^{J-1} \sum_{l=-\infty}^{\infty} d_{j,l}\psi_{j,l}(x),\ x \in \mathbf{R}$$

where:

$$\begin{aligned} c_{J_0} &= \int_{-\infty}^{\infty} f(x)\phi_{J_0}(x)dx \qquad (1.8) \\ d_{j,l} &= \int_{-\infty}^{\infty} f(x)\psi_{j,l}(x)dx \end{aligned}$$

This formula is the basis of the wavelet expansion.

1.3 The Dilation Equation

Until now we have not been able to actually draw a wavelet. In this section we will pursue that goal, as well as making a few statements for later user.

Recall that: $V_0 \subset V_1$, then a function expressed in terms of basis-functions in V_0 can also be expressed in terms of basis functions in V_1. In other words: $\phi_{0,0} = \phi(x) \in V_0$ can be expressed in V_1:

$$\phi(x) = \sum_{l=-\infty}^{\infty} a_k\phi_{1,k}(x) = \sqrt{2}\sum_{k=-\infty}^{\infty} a_k\phi(2x-k) \qquad (1.9)$$

The very general dilation equation.

We leave the a_k's to be unknown for a bit more. Since we require compact support, only a rather small number of the a_k's will be non-zero (remember that k is the translation parameter). For the so called Daubechies wavelets, non-zero in the range: $]0; D-1[$ and zero elsewhere, compact support, we get:

$$\phi(x) = \sqrt{2}\sum_{k=0}^{D-1} a_k\phi(2x-k)$$

Which is the dilation equation we will look at.

At first this equation seems very harmless. But one great feature of it is that if we know the a_k's and a number of equidistant numbers of $\phi(x)$, we will be able to calculate $\phi(x)$ at the finer granularity $\phi(2x)$.

Ergo we can now write our scaling function as:

$$\phi(x) = \sqrt{2} \sum_{k=0}^{D-1} a_k \phi(2x - k) \tag{1.10}$$

And since the wavelet function $\psi \in W_0$ and $W_0 \subset V_1$ (Remember that $V_1 = V_0 \oplus W_0$), ψ can now be expanded using $\phi_{1,k}$:

$$\psi(x) = \sqrt{2} \sum_{k=0}^{D-1} b_k \phi(2x - k) \tag{1.11}$$

and trivially we have the filter coefficients:

$$b_k = \int_{-\infty}^{\infty} \psi(x)\phi_{1,k}(x)dx \tag{1.12}$$

In that $V_j \perp W_j$, in this case $V_0 \perp W_0$ then:

$$\int_{-\infty}^{\infty} \psi(x)\phi(x)dx = 0$$

and since (1.10) and (1.12):

$$\begin{aligned}
\int_{-\infty}^{\infty} \phi(x)\psi(x)dx &= 2\int_{-\infty}^{\infty} \sum_{k=0}^{D_1} a_k\phi(2x-k) \sum_{l=0}^{D_1} b_k\phi(2x-l)dx \\
&= \sum_{l,k=0}^{D-1} a_k b_k \int_{-\infty}^{\infty} \phi(y-k)\phi(y-l)dy \\
&= \sum_{l,k=0}^{D-1} a_k b_k = \sum_{k=0}^{D-1} a_k b_k = 0
\end{aligned}$$

Now this relation is surely satisfied for $a_k = b_k = 0$ for $k \in \mathbf{Z}$, this is the trivial case. But the relation is also satisfied for:

$$b_k = (-1)^k a_{D-1-k}, \quad k = 0, 1, \ldots, D-1$$

Finding these filter coefficients a_k, have tricked mathematicians for a long time, and one the greatest breakthroughs was done by Ingrid Daubechies; who used Fourier analysis and conservation of various properties in the frequency domain to find sequences of filter coefficients a_k, that lead to orthonormal basises.

We will not go into further details in finding the coefficients. In our programming we have used precalculated Daubechies filter coefficients for genus 2 to 50.

1.4 Drawing a scaling function

Using the dilation equation (1.9) we are actually able to calculate the function values of the scaling function at dyadic intervals. We will have a look at the Daubechies wavelet of genus 4. Have a look at equation (1.3):

$$\phi(x) = \sqrt{2} \sum_{k=0}^{3} a_k \phi(2x - k) \tag{1.13}$$

Because of the compact support we know that $\phi(0) = \phi(3) = 0$. We get the scaling coefficient from the literature [GS93]:

$$\begin{aligned} a_0 &= \tfrac{1}{4}(1-\sqrt{3}) \qquad & a_1 &= \tfrac{1}{4}(3-\sqrt{3}) \\ a_2 &= \tfrac{1}{4}(1+\sqrt{3}) \qquad & a_3 &= \tfrac{1}{4}(3+\sqrt{3}) \end{aligned}$$

By writing the dilation equation (1.13) matrix-wise we get a solvable system:

$$\begin{bmatrix} \phi(0) \\ \phi(1) \\ \phi(2) \end{bmatrix} = \begin{bmatrix} a_0 & & \\ a_2 & a_1 & \\ & a_3 & a_2 \end{bmatrix}$$

The values of $\phi(x)$ for $x \in \{1, 2\}$ can now be found:

$$\phi(1) = \frac{1}{2}(1+\sqrt{3}) \;\; \phi(2) = \frac{1}{2}(1-\sqrt{3})$$

At all other integers the scaling function is zero, due to the compact support. If we now recall the scaling connectionship between the approximation spaces (1.10) – slightly rewritten to be (the x has been scaled by $1/2$):

$$\phi\left(\frac{x}{2}\right) = \sqrt{2}\sum_{k=0}^{D-1} a_k \phi(x-k)$$

This rewrite tells us that the scaling function can be calculated at discrete points in between the already know integer points. Once these has been calculated, this very same iteration can be applied, thus giving us function values of ϕ at an even finer granularity.

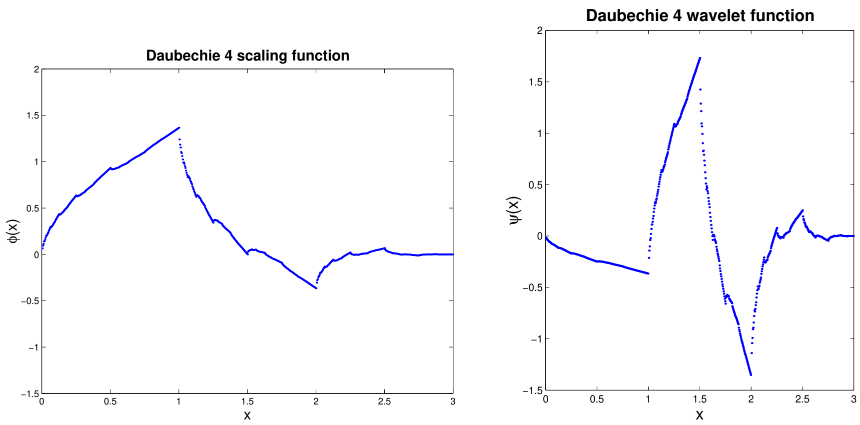

Figure 1.1: D_4 scaling and wavelet functions.

Once we have drawn the scaling function, the wavelet function comes quite natural since its function values can be directly calculated from the scaling function, see equation (1.11).

Chapter 2

Wavelet Expansion

Suppose we have already wavelet-expanded a function $f \in V_j$ as :

$$f(x) = \sum_{l=-\infty}^{\infty} c_{j,l}\phi_{j,l}(x) \tag{2.1}$$

And thus, because of scaling and dilation relationship between approximation spaces 1.5 we can also write the expansion in terms of scaling functions and wavelets:

$$f(x) = \sum_{l=-\infty}^{\infty} c_{j-1,l}\phi_{j-1,l}(x) + \sum_{l=-\infty}^{\infty} d_{j-1,l}\psi_{j-1,l}(x) \tag{2.2}$$

We would now like to get from equation (2.1) to equation (2.2) using some sort of an algorithm. Using the dilation and scaling properties we can do just that. Remember:

$$\begin{aligned}
\phi_{j-1,l}(x) &= 2^{(j-1)/2}\phi(2^{j-1}x - l) && (2.3)\\
&= 2^{j/2}\sum_{k=0}^{D-1} a_k\phi(2^jx - 2l - k) && (2.4)\\
&= \sum_{k=0}^{D-1} a_k\phi_{j,2l+k}(x) && (2.5)
\end{aligned}$$

We have expressed the scaling functions at level $j-1$ as a linear combination of the scaling functions at level j. Furthermore, we can, using equation (1.11) express the wavelet function values as:

$$\psi_{j-1,l}(x) = \sum_{k=0}^{D-1} b_k\phi_{j,2l+k}(x) \tag{2.6}$$

The scaling and wavelet functions are now defined in terms of the former ones. This will give us the new $c_{j-1,l}$ and $d_{j-1,l}$ coefficients as:

$$\begin{aligned}
c_{j-1,l} &= \int_{-\infty}^{\infty} f(x)\sum_{k=0}^{D-1} a_k\phi_{j,2l+k}(x)dx && (2.7)\\
&= \sum_{k=0}^{D-1} a_k\int_{-\infty}^{\infty}\phi_{j,2l+k}(x)dx && (2.8)
\end{aligned}$$

Here we see that the integral is just the calculations to compute the coefficients $c_{j,2l+k}$ (1.8), thus we have:

$$c_{j-1,l} = \sum_{k=0}^{D-1} a_k c_{j,2l+k} \tag{2.9}$$

Furthermore we have:

$$d_{j-1,l} = \sum_{k=0}^{D-1} b_k c_{j,2l+k} \tag{2.10}$$

This mapping from one approximation space onto another is called the Partial Wavelet Transform (PWT) – using the PWT multiple times, the Fast Wavelet Transform (FWT) is obtained.

Chapter 3

The Fast Wavelet Transform

In this chapter we will outline the principles of the Fast Wavelet Transform and describe the algorithm for performing the efficient Fast Wavelet Transform.

3.1 A fast serial 1D FWT

Suppose we have a vector c_4 containing the elements $c_{4,0}, c_{4,1}, \ldots, c_{4,15}$ onto which we would like to apply the Fast Wavelet Transform. Since the vector has 16 elements, we can apply Partial Wavelet Transform to a total depth of $\lambda = 4$ – in which case we end up with just one scaling function coeffient $c_{0,0}$ and 15 wavelet scaling coefficients. In pseudo-code this algorithm, which is just the PWT ((2.9) and (2.10)) is shown in figure 6.1.

3.2 2D Fast Wavelet Transform

Basically the 2D Fast Wavelet Transform is 1D Fast Wavelet Transforms done columnwise then rowwise. The naive rowwise FWT is bad for cache coherency, hence another approach must be taken. The advantage of the columnwise FWT is that data is accessed with a stride 1, giving good cache utilization; this property can however also be applied to the rowwise FWT.

Instead of doing the rowwise FWT's one by one (Figure 3.1, naive approach), the rowwise FWT's are done in steps traversing the rows, the clever approach.

The pseudocode for the 2D Fast Wavelet Transform is shown in figure 6.5.

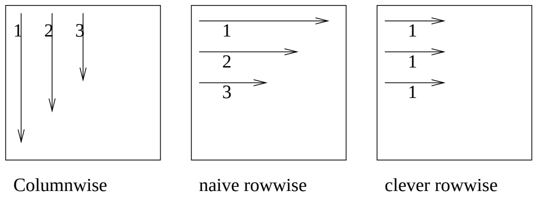

Figure 3.1: Illustration the 1D FWT used in 2D FWT

Chapter 4

Implementation

In this chapter we will discuss some of the aspects of the actual implementation of the wavelet transform routines. Experiences and results from the optimization and parallelization phases will be discussed.

4.1 Optimization of the FWT routines

The ideas and methods concerning performance tuning of the fast wavelet transform outlined in [OMN98] cannot be directly applied to our implementation, as we implement for cache coherent architectures, while [OMN98] describes an implementation on a vector processor machine.

While a vector-processor implementation will focus on pipe-lining, and not care too much about agressive memory usage, we on the other hand do not have to think about pipe-lining, but we take great performance hits on cache misses.

A description of the inner workings of the Intel Pentium Pro and Pentium II CPU cache can be found in [Int97]. Accurate information about CPU cache is invaluable when optimizing data intensive routines.

Optimization on Serial Intel arch.				
Problem	Size	Non-optimized	Optimized	Speed
F1D	2^{10}	2,257 μsec	448 μsec	559%
	2^{12}	10,127 μsec	1,701 μsec	595%
	2^{14}	38,715 μsec	11,583 μsec	334%
	2^{16}	167,985 μsec	49,590 μsec	339%
	2^{18}	691,285 μsec	213,302 μsec	324%
	2^{20}	2,794,424 μsec	870,637 μsec	321%
F2D	$2^{2 \cdot 8}$	331,562 μsec	64,320 μsec	515%
	$2^{2 \cdot 9}$	1,325,565 μsec	316,463 μsec	419%
	$2^{2 \cdot 10}$	5,272,698 μsec	1,120,296 μsec	471%

Optimization on Serial Intel arch.				
Problem	Size	Non-optimized	Optimized	Speedup
I1D	2^{10}	1,545 μsec	548 μsec	282 %
	2^{12}	6,281 μsec	1,930 μsec	325 %
	2^{14}	30,681 μsec	12,743 μsec	241 %
	2^{16}	201,316 μsec	89,496 μsec	225 %
	2^{18}	882,119 μsec	376,350 μsec	234 %
	2^{20}	3,820,707 μsec	1,527,478 μsec	250 %
I2D	$2^{2\cdot 8}$	266,631 μsec	115,897 μsec	230 %
	$2^{2\cdot 9}$	1,096,515 μsec	536,483 μsec	204 %
	$2^{2\cdot 10}$	4,658,427μsec	2,179,913 μsec	214 %

As one can see, the optimizations resulted in dramatic speed improvements for data sets of size 2^{12} and below, and only modest improvements for larger sets. This can be explained very well by the fact that the speed tests where run on an Intel Pentium Pro machine with 256 KB of CPU cache. A problem of size 2^{12} will use two buffers each holding 2^{12} doubles. This will take up $2^{12} \cdot 2 \cdot 8 = 65536$ bytes, which will easily fit into the CPU cache. The next problem of size 2^{14} will take up 262144 bytes, exceeding the CPU cache greatly.

We spent approximately ten man-hours optimizing the two routines. In return we got from well above 200% to almost 500% speedup on the routines. On Intel architecture that is.

4.2 Shared memory considerations

Both the 1D and the 2D routines are parallelized to run on symmetric multiprocessor machines. The 1D routines where only parallelized for the sake of completeness. We cannot expect much gain in performance on these routines, since they generally treat small datasets compared to the 2D routines.

Parallelization affects performance in two directions. It increases performance of the routine, since the computational workload is distributed among several processors. But the processors need to synchronize their datasets at certain points in the algorithms, which introduces some latency.

A good implementation will seek to minimize the number of synchronizations to maximize the overall speedup. However, the performance of the implementation will inherently depend on the efficiency of the hardware and operating system it is running on.

Our implementation of the SMP routines relies on POSIX threads, and in one special case, on a vendor specific (but nonetheless unavoidable) thread-like interface. It uses two synchronization primitives, *mutual exclusion semaphores* and *condition variables* (the vendor-specific interface provided similar functionality). If the routines are run on an operating system with an inefficient implementation of these primitives, the routines cannot be fast. When calling the SMP routines, they will spawn a specified or default number of threads. If the cost of spawning such a thread is high, the routine will suffer from a high "initial cost". This is not as bad as if the synchronization primitives are slow (since they are called numerous times) but it could turn out to be bad enough.

Especially on PC architecture, memory bandwidth is a huge concern. Even if thread synchronization and instantiation where infinitely fast, our overall performance would be limited by the bandwidth of the memory subsystem.

The implementation of the serial routines is optimized for cache efficiency, i.e. accessing as much data as possible with stride 1. The parallel implementation is a rewrite of the serial routines, and thus has the same optimization. This is not a guarantee however, that the processors will not have spend much of the time waiting for data from the main memory. A typical high-end PC has a memory bandwidth of 100 MB/sec, corresponding to 10^7 doubles/sec. If the performance of each processor is $2 \cdot 10^7$ FLOPS which is quite realistic, it is easy to imagine the load of the processor versus that of the memory subsystem.

To round up on our expectations of the symmetric parallel implementation we state the following: *Distributing a job on n processors will never give n times the serial performance.* This is due to

- Delays in the thread synchronization primitives will decrease the performance of a multi threaded program.
- Initial cost of spawning threads will introduce an initial cost of execution.
- Memory bandwidth may limit the speedup severely on some architectures.

4.3 1D FWT

Several approaches are possible, when implementing the 1D parallel FWT. If one chooses to maintain the format of the data in the wavelet domain used in the serial implementation,

$$c_0^3 d_0^3 d_1^3 d_2^3 \ldots$$

the processors must communicate after every iteration. A very good improvement to this approach is, to let every processor perform a complete transform on a part of the domain, returning something like

$$c_0^3 d_0^3 d_0^2 d_1^2 \ldots, c_1^3 d_1^3 d_2^2 d_3^2 \ldots$$

to the user. This is well described in [OMN98].

However, we have chosen to make the parallelization as transparent as possible to the user. Thus, we could not accept the latter data layout. This means, that a parallel implementation of the 1D FWT will be extremely expensive in terms of communication.

We did implement a parallel 1D FWT for shared memory architectures, and it is possible to use it – if called explicitly. However, if one uses the generic 1D FWT routines, they will always run in serial mode, even if run on a parallel architecture.

Chapter 5

Portability

A part of this project was to develop portable code using well-defined standards. With portable code you will be able to develop your own program using our code on a standard workstation and then later on, when the need is there, move your code along with our library routines to a supercomputer. More specifically the goal was to be able to use our library on a normal single processor personal computer as well as on a shared memory multiprocessor computer or a distributed memory parallel computer.

To achieve this, we had to rely on different standard libraries begin available on the platforms we support. In short we utilize:

- Single processor serial execution
- Shared memory multiprocessor computers using threads
- Distributed memory parallel computers using the Message Passing Interface (MPI)

To ensure consistency throughout the development process, we developed a small regression test utility that, when used, would read a script file with some set of challenging test data, along with the pre-computed correct answers. If we introduced an error in any of the transform routines, there would be a large probability that the regression test would fail. It proved well worthwhile spending the time building the utility, as it caught numerous errors in very early stages.

The regression test program also serves a purpose when porting the library to new platforms, as it will allow the future developers to easily verify the integrity of the package, whithout deep understanding of the wavelet internals.

5.1 Serial versions

Not much is said about the single processor serial methods. The programming of these forward and inverse 1D and 2D routines follows the pseudo code in 6.1 and 6 closely. These routines are written in strict ANSI `C` and should pose no problems on an ANSI `C` compiler with standard libraries.

5.2 Shared memory versions

The shared memory multiprocessor versions did however cause some minor problems. Having implemented the forward and inverse routines on our Dual Pentium

Pro Linux system using POSIX threads we cunningly moved on to the Silicon Graphics Origin 2000.

The choice of POSIX threads seemed obvious as this is one of the few standards for shared memory parallel execution under UNIX-like operating systems that is not vendor specific. It is actually implemented under many different operating systems. Besides that, many UNIX systems strive towards POSIX compliance thus buying us cheap portability.

We have found out that when it comes to portability POSIX threads are well supported. The POSIX version of our library has been compiled without trouble on:

Intel PC (running Linux), Sun UltraSPARC (running Solaris), SGI Origin 2000 (running IRIX), and for the fun of it: on a single processor Digital Alpha (running DEC UNIX) – all versions passed the regression test.

The idea on the Intel Linux, SUN UltraSPARC and SGI IRIX multiprocessor systems was that the operating system should distribute our CPU intensive threads onto the processors available - which they did.

Sales material on the SGI web site promised that the POSIX thread implementation on the SGI Origin supported multiple processors, i.e. that our threads would be distributed among the processors [SGIweb]. Thorough investigation, however, revealed that this was *not* the case. The POSIX thread implementation on the SGI Origin cannot distribute the threads, they stay on the same CPU. This was very unsatisfactorily for us because we needed more processors in order to get increased performance compared to the serial version of our program. According to SGI on-line documentation [SGIinsight] there were basically three ways of doing parallel computations on the SGI Origin[1], these three were:

1. Seperate processes (using the `fork()`-function)
2. IRIX lightweight processes (using the `sproc()`-function)
3. POSIX threads

The idea of using seperate processes was useless for our project as these processes do not share the memory, they have to communicate through various forms of Interprocess Communications (IPC) such as files, sockets or signals. The POSIX threads were out of the question as mentioned. Then we were left with the IRIX-specific light weight processes. These processes seemed to fit our needs very well, they share memory and they can be distributed onto several processors.

The documentation in [SGIinsight] on these lightweight processes were however alarmingly sparse; whereas the documentation on IPC and POSIX threads were quite elaborate. Some of the documentation say that care should be taken when using these processes in connection with the IRIX implementation of the MPI-standard as well as with the IRIX auto-parallelizing compilers, as these technologies sport the lightweight processes. Since SGI uses these lightweight processes themselves for doing shared memory parallel computations, we consider them to be the obvious choice for our use.

Moving our POSIX code to the IRIX lightweight processes was fairly easy once we found the functions we needed. The lack of proper documentation was however quite a nuisance. Take the `waitpid()` function for example, it waits for a specific process to either terminate or send some sort of signal. This one is not documented at all – though mentioned various places. We actually found the documentation for a similar function on another UNIX and mixed that information with what we could read from the system header files on the IRIX, to get a proper picture of its behaviour.

[1] Besides using technologies like Parallel Virtual Machine (PVM) and MPI

Sad that efficient programming implies guesswork and reverse engineering like this.

5.3 Distributed parallel versions

In this case we had several computers, called nodes, which had to intercommunicate over a network. To do this we used the Message Passing Interface (MPI) a standard interface we both had used several times before. The MPI version of our library has been compiled and run on HP 9000 C180 nodes, IBM RS/6000 nodes as well as on a Dual Pentium Pro.

Our MPI-version soon proved to be the most portable of the parallel routines. Some of the systems we used required special compilers or compile settings to make the MPI-binaries, but the source code itself required no change at all.

The disadvantage of using MPI is of course that it takes time to transfer information between the cpu's and that the nodes do not necessarily have access to the whole dataset. But these limitations are also what makes MPI usefull; i.e. using dataset too large for one machine by distributing it among n machines.

Chapter 6

The Performance Model

The Fast Wavelet Transform is a linear transform, eg. it should take twice as long time to process twice as much data. At least this is what we would expect from a mathematical point of view.

As we will also mention in the timing section (section 6.1), many factors are involved when it comes to real world performance measurements. Some factors are very dependent on the specific hardware platform on which the transform is run. Theese factors can be put into the following categories:

- Processor speed and capabilities
- Cache size
- Memory bandwidth

Besides from these quite predictable factors (they are all constants), some factors are strictly random. One large factor which we cannot avoid nor predict is that we run on multi-user sytems. This means that usually there will be n processes running on p processors where $n > p$, implying task-switching. Whenever our Wavelet Transform process gets suspended to allow some other process to run on the processor, we will not only loose CPU cycles, but the cache will also be cluttered up. This is bad, and the performance hit we take from task switching can be minimized in a few ways:

- Maximize p (buy more processors)
- Minimize n (run job on dedicated processors)
- Make Wavelet Transform only depend on using a little of the cache

Since we cannot just buy a larger machine, and since the Wavelet Transform is very memory intensive and therefore uses as much cache as possible, we are stuck with the "solution" of running our program somewhere where as few other tasks as possible will interfere. This is usually possible on supercomputer systems, by submitting the job to a queueing system which will run it exclusively on some number of CPUs.

One thing we also choose not the worry about, is the capabilities of the CPUs we run the timings on. Some CPUs are able to increment integer loop counters concurrently with performing the floating point operations needed in the transform, while other CPUs may have to do integer and floating point computations one at a time.

Due to all theese many factors of incertainty, we choose to limit the performance models of the serial routines to only account for:

- The number of FLOPS needed by the transform

- The number of Cache Misses

This will hopefully reflect the actual performance of the algorithms quite well. If it doesn't, it will indicate that the computational work is not the dominant part of the algorithm, wrt. execution time. This will in turn indicate that we have a serious problem with our implementation.

The symmetric parallel routines does not suffer from network limitations. They take a small performance hit when synchronizing, but since this is done very few times during the transform, we choose to neglect this fact. A symmetric architecture could, however, suffer from memory bandwidth limitations. Especially on the PC architecture, it is possible to saturate the memory bus, at times leaving the processors with nothing to do.

During the performance model descriptions, we use the following notations. Where applicable, some typical values for the platform on which we have done performance modelling (the PPro) are supplied. The values where found using the performance measurement tools described in appendix A.5.

Glyph	Meaning	PPro
$\mathcal{T}$	Execution time	—
$\mathcal{M}$	Cache miss cost	$3.48 \cdot 10^{-7}$ sec.
$\mathcal{C}$	Cache line length	32 bytes
$\mathcal{F}$	FLOP cost	$2.56 \cdot 10^{-8}$ sec.
$\mathcal{L}$	Domain length (1D)	—
$\mathcal{L}_{\{1\|2\}}$	Domain dimensions (2D)	—

6.1 A word about timing

When timing programs, many external factors can influence the result, polluting it in a way so that the timings may not reflect very well what performance *could* have been achieved with the program.

The fact that we run all our timings on multi-user systems also means, that our timings will inherently be influenced by external factors, such as other users launching programs, the system running maintenance tasks by itself etc.

Most multiuser systems can be put into a "single user mode", in which the user at the console will be the only user able to run jobs on the system. But single user mode usually implies that the network connections are shut down (rendering MPI usage impossible), and also requires superuser privileges. It has thus not been possible for us, to take the machines into single-user mode when measuring execution times on the Wavelet Library routines.

However, all timings where been done when the systems were loaded minimally (during late nights and early mornings). On some of the systems the timing runs were submitted into a queuing system making sure the program had almost exclusive access to the processors.

6.2 The Forward 1D Transform

The performance model of the forward transform is fairly simple. In pseudo code the algorithm we use looks roughly like seen in figure 6.1.

The pseudo code does not correctly describe a wavelet transform, it does however reflect the work done by the algorithm well. We neglect some work needed to complete the transform, but since this work is insignificant in terms of FLOPS, we take this liberty.

In our model, the 1D Forward Transform execution time will depend on the following parameters:

- The Wavelet Genus D
- The length $\mathcal{L}$ of the input domain
- The cost $\mathcal{M}$ of a cache miss
- The cost $\mathcal{F}$ of one floating point operation
- The length $\mathcal{C}$ of one cache line

From the algorithm in figure 6.1 we can write down our execution time estimate:

$$\boxed{\mathcal{T} = \mathcal{L}(4D\mathcal{F} + D\mathcal{M}/\mathcal{C} + 2\mathcal{M}/\mathcal{C})} \tag{6.1}$$

The first two `for` loops in the pseudocode will approximately generate $\mathcal{L}$ loops over the innermost `for` loop. This is because the first loop loops from $d = 1 \ldots \lambda$ while the second loop loops from $\ell = 0 \ldots 2^d$, and we have

$$\sum_{d=1}^{\lambda-1} 2^d = d^\lambda - 2 \approx \mathcal{L}, \quad \text{since } \lambda = \log_2(\mathcal{L})$$

We assume that we get one cache miss every $\mathcal{C}$'th time, all three places in the pseudo code where we have the possibility of getting a cache miss. This assumption is true if our domain is larger than the cache, which is usually the case.

In figure 6.2 we see our model estimates along with some actual real world timing results. We see that the model is very close to the actual results, when the domain is large. For small domains the model predicts a little faster run-time than we achieve in practice. This must be due to some start-up cost of the implementation we do not take into account in the model.

6.3 The Inverse 1D Transform

The 1D inverse transform is, like the 1D forward, fairly simple to model. Due to the nature of the inverse transform, it is however not as cache efficient as the forward transform. We will see this directly in the model.

```
for depth = λ...1
   for l = 0...2^depth
      for k = 0...genus-1
         c = buffer(2l + k)              ; Eventually a cache miss
         a = a + daub_a(D, k) · c        ; 2 FLOPS
         b = b + daub_b(D, k) · c        ; 2 FLOPS
      end
      outbuffer(l) = a                   ; Eventually a cache miss
      outbuffer(l + 2^depth) = b         ; Eventually a cache miss
   end
end
```

Figure 6.1: Pseudo code for the 1D Forward Transform

The pseudo code for the 1D inverse transform can be found in figure 6.3. As before, we do not show anything that would resemble a correct inverse transform, if implemented, but the pseudo code should reflect the performance characteristics of the implemented transform well.

Again, assuming that our domain is larger than the cache, we can now write down an execution time estimate from the pseudo code in figure 6.3:

$$\boxed{\mathcal{T} = D/2\mathcal{L}(8\mathcal{F} + 4\mathcal{M}/\mathcal{C})} \tag{6.2}$$

We see, that the inverse transform is both heavier in terms of computational work, and in terms of cache misses. We will expect a lower performance of the inverse transform, than we have seen from the forward transform.

In figure 6.4 we see the model results along with some real world timings. This

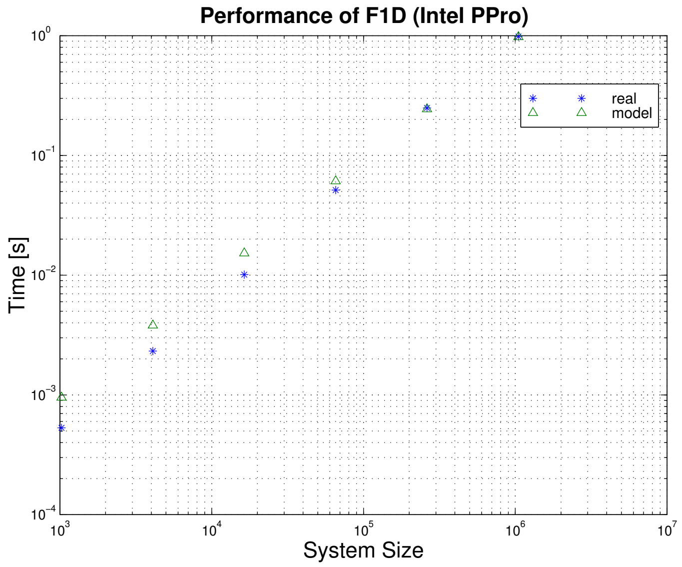

Figure 6.2: The performance model results and the real world timing results. Theese timings are from the Intel Dual PPro

```
for j = 1 ... λ
   for k = 1 ... D − 1
      for i = 0 ... 2^j
         r = i(mod 2^(j−1))
         f1 = buffer(r)                      ; Eventually a cache miss
         f2 = buffer(r + 2^(j−1))            ; Eventually a cache miss
         out(i) += a1 f1 + b1 f2             ; 4 FLOPS + Evt. a cache miss
         out(i + 2^(j−1)) += a2 f1 + b2 f2   ; 4 FLOPS + Evt. a cache miss
      end
   end
end
```

Figure 6.3: Pseudo code for the 1D Inverse Transform

time our model does not fit the real world results quite as well, but there is still a clear resemblance. It is important to note that our initial assumption, that the domain should be larger than the cache, is only fulfilled for the latter three system sizes. The cache on the PPro is 256 KB, and this limit is first exceeded when the domain size goes from 2^{14} to 2^{16} doubles (as one double takes up eight bytes).

6.4 The Forward 2D Transform

The Forward 2D Transform is built upon two passes of multiple 1D Forward Transforms. First, the 1D transform described in section 6.2 is run on every column of the domain. Secondly a variant of the 1D transform is run on every row of the domain. This latter variant of the F1D transform does a good job of maintaining cache coherency, by doing small parts of the 1D transform on every row. As we will see, the row-wise transform is actually identical with tieh column-wise transform, with respect to both computational cost (no surprise there) and to the number of cache misses.

The pseudo-code for the transform is shown in figure 6.5. As usual it does not reflet the transform very well in a mathematical sense, but it does resemble the actual cost well.

We will have to make a few extra assumptions, in order to propose a general performance estimate for the last half of the 2D transform. Since it traverses the domain in a mixed row- and column wise manner, we will depend on more than the usual two cache lines. In fact, we will depend on $2D$ cache lines. If the hardware has less than $2D$ cache lines available, this last half of the 2D transform will do cache thrashing, and it will be inherently slow. However, most systems will greatly satisfy this requirement, since D is usually relatively small (< 50), and the number of cache lines is usually large (the PPro has 8192 cache lines, the supercomputers have a lot more). Our first extra assumption will be, that the number of cache lines is greater than the wavelet genus.

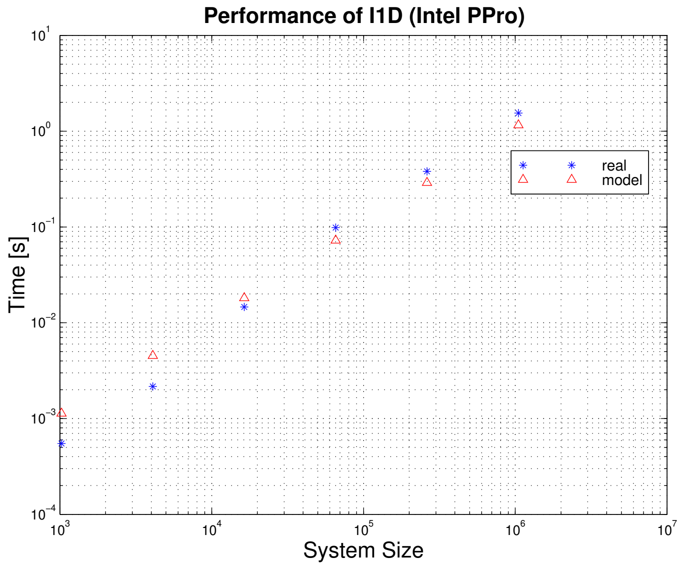

Figure 6.4: The performance model results and the real world timing results. Theese timings are from the Intel Dual PPro

For the sake of simplicity, we furthermore make the assumption, that the leading dimension of the domain ($\mathcal{L}_1$) is larger than the length of a cache line ($\mathcal{C}$).

It is seen, that remembering the two extra assumptions, the last half of the 2D transform is identical to the first half, in terms of performance. Thus, the execution time estimate is:

$$\boxed{\mathcal{T} = 2\mathcal{L}_2\mathcal{L}_1(4D\mathcal{F} + D\mathcal{M}/\mathcal{C} + 2\mathcal{M}/\mathcal{C})} \tag{6.3}$$

We know from section 6.2 that the model for the forward 1D transform lies very close to the real world measurements. Re-using this model should give another close match with reality. However, in figure 6.6 it is seen, that the model and the real world does not agree quite as well as we would have hoped for. In fact the model predicts a roughly 50% longer execution time.

The latter half of the transform does access the memory in another way. It performs more floating-point operations between the cache-misses, but when it misses cache, it takes four misses in a row. Perhaps this can explain the behaviour. We have not investigated this any further, so what the actual cause for this unexpected

```
; Do column-wise transform on every column
for col = 1...L_2
   for depth = λ...1
      for l = 0...2^depth
         for k = 0...genus-1
            c = buffer(2l + k)                  ; Eventually a cache miss
            a = a + daub_a(D, k) · c            ; 2 FLOPS
            b = b + daub_b(D, k) · c            ; 2 FLOPS
         end
         outbuffer(l) = a                       ; Eventually a cache miss
         outbuffer(l + 2^depth) = b             ; Eventually a cache miss
      end
   end
end
; Do row-wise transform on a few columns at a time
; to maintain cache coherency
for depth = λ...1
   for l = 0...2^depth
      for row = 1...L_1
         for k = 0...genus-1
            c_j = buffer(2lL_1 + row + kL_1)    ; Eventually a cache miss
            c = c + daub_a(D, k) · c_j          ; 2 FLOPS
            d = d + daub_b(D, k) · c_j          ; 2 FLOPS
         end
         dout(row + lL_1) = a                   ; Eventually a cache miss
         cout(row + lL_1) = b                   ; Eventually a cache miss
      end
   end
end
```

Figure 6.5: Pseudo code for the 2D Forward Transform

speedup of the implemented transform is, remains a mystery. We must however be pleased, that our implementation performs so much better than estimated.

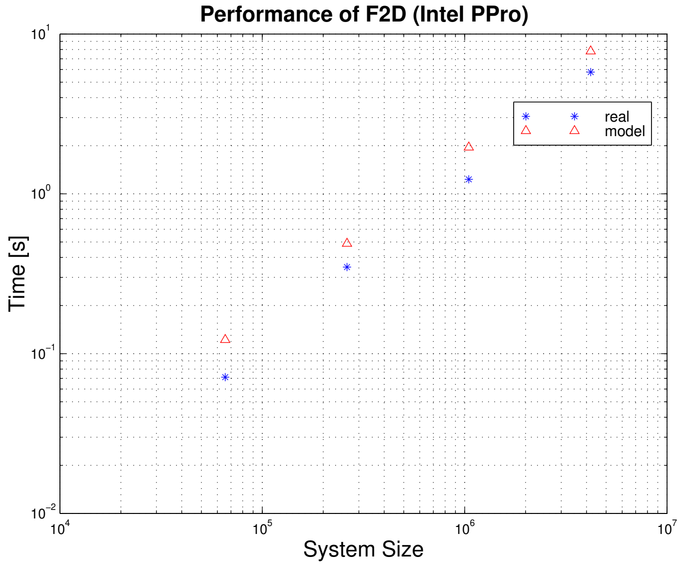

Figure 6.6: The performance model results and the real world timing results. Theese timings are from the Intel Dual PPro

6.5 The Inverse 2D Transform

Like the Forward 2D Transform, the Inverse 2D Transform is built upon multiple inverse 1D transforms, first column-wise, then row-wise.

The pseudo code for the inverse 2D transform is found in figure 6.7. As with the forward 2D transform, it closely resembles what we've seen in the inverse 1D transform, figure 6.3.

We maintain the two extra assumptions from the previous section, saying that the number of cache lines in the processor is greater than the wavelet genus, and that the leading dimension of the domain is greater than the length of a cache line.

As we saw in the case with the forward 2D transform, where it turned out that $\mathcal{T}_{F2D}(\mathcal{L}_1, \mathcal{L}_2) = 2\mathcal{T}_{F1D}(\mathcal{L}_1 \cdot \mathcal{L}_2)$, we see the exact same thing in this case with the inverse 2D transform. The cache miss pattern changes, but the total number of both floating point operations, and of cache misses is preserved. Thus, we can write the execution time estimate of the inverse 2D transform as:

$$\boxed{\mathcal{T} = D\mathcal{L}_1\mathcal{L}_2(8\mathcal{F} + 4\mathcal{M}/\mathcal{C})} \tag{6.4}$$

In figure 6.8 we see the execution time estimates and the corrosponding real world measurements. Unlike seen with the forward 2D transform (figure 6.6), we see a very good correlation between theory and practice here. As we have seen before in the performance models, the larger system sizes seem to give the best fit between the model and the real world, this does not come as a surprise of course.

6.6 The Parallel Forward 2D Transform

The Parallel Forward 2D Transform is very similar to the ordinary forward 2D transform. Since the work is distributed among several processors, each processor will only process a part of the domain. In the first column-wise pass, every processor will process it's own part of the columns in the domain. Then, the processors will communicate the nessecary information to eachother. Second pass of the multiple 1D transform is done as before in a row-wise manner. Finally, the main processor will gather all necessary information to be able to put together the resulting domain.

The transpose step is illustrated in figure 6.9. The data used by the main processor is grayed out. The dark gray part of the domain is the data that is local to this processor throughout the entire 2D transform. This data never needs to be communicated. However, the light-gray data in the left domain needs to be communicated to the other processors, and the light-gray data in the right domain needs to be received from the two other processors.

The pseudo code for the massively parallel forward 2D transform is found in figure 6.10. We use some utility routines to easily show when we do the column-

```
; Do simple multiple column-wise 1D inverse transforms
for col = 1...L_2
   for j = 1...λ
      for k = 1...D − 1
         for i = 0...2^j
            r = i(mod 2^(j−1))
            f_1 = buffer(r)                          ; Eventually a cache miss
            f_2 = buffer(r + 2^(j−1))                ; Eventually a cache miss
            out(i) += a_1 f_1 + b_1 f_2              ; 4 FLOPS + Evt. a cache miss
            out(i + 2^(j−1)) += a_2 f_1 + b_2 f_2    ; 4 FLOPS + Evt. a cache miss
         end
      end
   end
end
; Do row-wise inverse transform, but do a few
; columns at a time, to maintain cache coherency
for d = 1...λ
   for k = 1...D − 1, 2
      for i = 0...2^j
         r = i(mod 2^(j−1))
         for j = 0...L_1
            f_1 = buffer(rL_1 + j)                   ; Eventually a cache miss
            f_2 = buffer((r2^(d−1))L_1)              ; Eventually a cache miss
            out(iL_1 + j) += a_1 f_1 + b_1 f_2       ; 4 FLOPS + Evt. a cache miss
            out((i + 1)L_1 + j) += a_2 f_1 + b_2 f_2 ; 4 FLOPS + Evt. a cache miss
         end
      end
   end
end
```

Figure 6.7: Pseudo code for the 2D Inverse Transform

and row wise multiple 1D transform we have allready analyzed.

One will find, that the performance model of the massively parallel forward 2D transform is very much alike the model of the serial forward 2D transform. Only, we must add a little network latency and a little data transfer delay due to finite network bandwidth.

Our previously defined assumptions about the leading dimension of the domain being larger than the length of a cache line now extends into an assumption that the leading dimension divided by the number of processors, should be larger than the length of a cache line. We have $\mathcal{L}_1/n > \mathcal{C}$. Let's set up the model:

$$\boxed{\begin{aligned} \mathcal{T} &= 2\mathcal{L}_2\mathcal{L}_1(4D\mathcal{F} + D\mathcal{M}/\mathcal{C} + 2\mathcal{M}/\mathcal{C}) \\ &+ 2(n-1)(\mathcal{N} + \mathcal{L}_1\mathcal{L}_2/n^2\mathcal{B}) + (n-1)(\mathcal{N} + \mathcal{L}_1\mathcal{L}_2/n\mathcal{B}) \end{aligned}} \tag{6.5}$$

Note the introduction of two new variables:

- $\mathcal{B}$ The cost of sending one `double` across the network
- $\mathcal{L}$ The network latency.

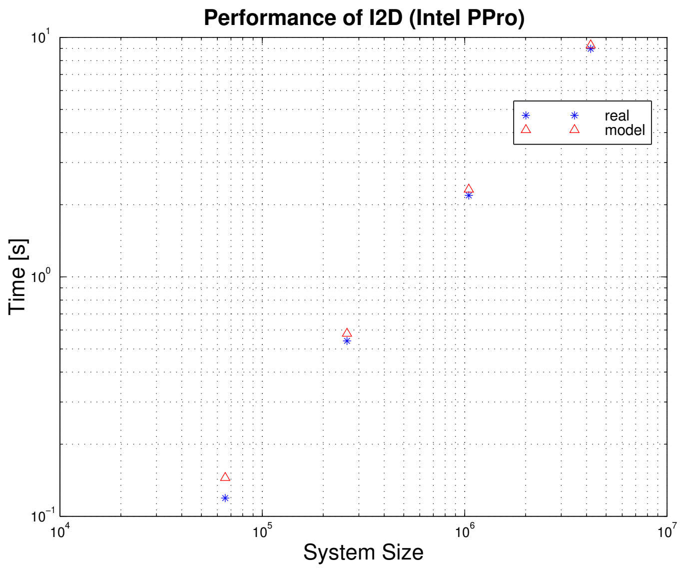

Figure 6.8: The performance model results and the real world timing results. Theese timings are from the Intel Dual PPro

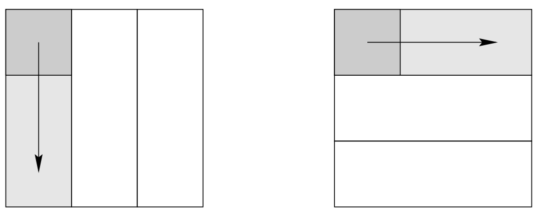

Figure 6.9: Illustration of the transpose operation done between the column-wise and the row-wise multiple 1D transform passes

Because of the heavy and ever changing load in the G databar, and on the G databar network, we where unable to achieve any remotely sane test results. Even performance measurements running over nights, showed chaotic behaviour.

We are still confident that the model proposed, will describe the performance of the massively parallel routines. This is however perhaps only of academic interest, since we have been unable to reproduce any predictable behaviour in real world experiments.

6.7 The SMP routines

The symmetric parallel routines are primarily of interest to those who want to speed up the wavelet computations. Since all processors share the same memory, one will not be able to run any larger systems than with the serial routines. However, in real-time applications (video or sound compression etc.) it may be important to be able to speed up the computations by adding more processors to the computer system.

As seen in figure 6.11, a very good speedup is achievable by using the SMP routines on the Origin 2000 supercomputer. Ofcourse, as one would expect, there is some initial cost of parallelizing the work, and thus the performance gain will increase with the system size. This is seen both in the figure mentioned, but also in more exact numbers in the table just below:

```
; Note that n is the number of processors, while
; id is the id of the current processor.

call column-1D-transform(L1,L2/n)
for dst = 1...n
   if id ≠ dst then
     send-block-to-other(L1/n,L2/n,dst)
     receive-block-from-other(L1/n,L2/n,dst)
   end
end
call row-1D-transform(L1/n,L2)
if id == 1 then
   for src = 2...n
     ; I'm the master, I should receive from slaves
     receive-block-from-other(L1/n,L2,src)
   end
else
   I'm a slave, I should send to master
   send-block-to-other(L1/n,L2,1)
end
```

Figure 6.10: The massively parallel forward 2D transform pseudocode. The parameters given to the routines called from the above code are only meant to illustrate the sizes of the data-sets in question. They do by no means specify what data is treated

Execution time of F2D on Origin 2000			
Size	One CPU	Two CPUs	Four CPUs
$2^{2\cdot 8}$	19,810 μsec	35,396 μsec	33,862 μsec
$2^{2\cdot 9}$	103,896 μsec	144,065 μsec	87,355 μsec
$2^{2\cdot 10}$	521,057 μsec	559,011 μsec	332,170 μsec
$2^{2\cdot 11}$	4,559,520 μsec	3,422,886 μsec	1,643,119 μsec
$2^{2\cdot 12}$	16,834,149 μsec	10,911,799 μsec	5,695,870 μsec

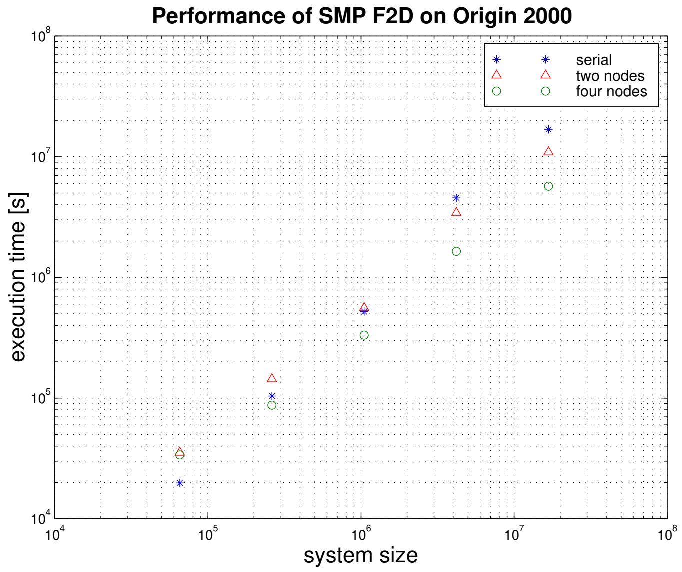

Figure 6.11: Illustration of the achieved performance of the Forward 2D Transform on the Origin 2000

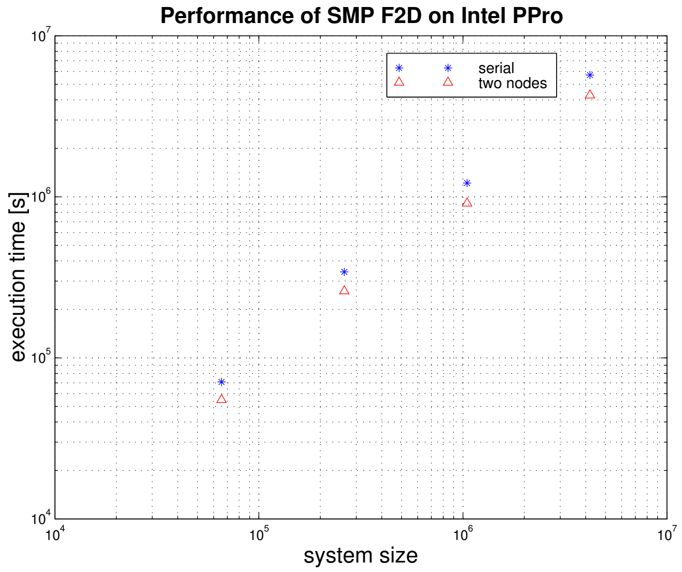

Figure 6.12: Illustration of the achieved performance of the Forward 2D Transform on the Intel PPro

A speedup similar to that seen on the Origin 2000, is not achievable on the Intel PC. This is due to limitations in the PC architecture, especially the design of the memory subsystem. The memory subsystem is simply not capable of supplying the two processors with data. The figures shown in the table below, can also be found in figure 6.12.

Execution time of F2D on Intel PPro		
Size	One CPU	Two CPUs
$2^{2\cdot 8}$	69,733 μsec	54,973 μsec
$2^{2\cdot 9}$	346,463 μsec	259,756 μsec
$2^{2\cdot 10}$	1,211,687 μsec	912,162 μsec
$2^{2\cdot 11}$	5,621,580 μsec	4,274,781 μsec
$2^{2\cdot 12}$	N/A	N/A

The system of size $2^{2\cdot 12}$ could not be treated on the PC, because of memory limitations.

Chapter 7

Function Reference

This chapter is a short reference guide to all routines provided by the libWavelet library.

7.1 General routines

The general routines will use the fastest transform routine possible on whatever hardware they are used on, meaning that on different parallel architectures, the general routines will exploit parallelism in some predefined manner.

If one wishes to use a specific number of parallel tasks, one will have to use the more specific parallel routines described in the following sections.

fwt_init(`void`)

Memory req	
	Initializes the libWavelet subsystem for use.

fwt_finalize(`void`)

Memory req	
	Shuts down the libWavelet environment.

fwt_F1D(`double*` *input*, `int` ℓ, `int` λ, `int` D, `double*` *output*)

Memory req	ℓ doubles
	Forward 1D transform. Performs a simple fast forward wavelet transform on the data set given in *input*.

fwt_F2D(`double*` *input*, `int` ℓ_1, `int` ℓ_2, `int` λ_1, `int` λ_2, `int` D, `double*` *output*)

Memory req	$\ell_1 \cdot \ell_2$ doubles
	Forward 2D transform. Performs a simple two-dimensional fast forward wavelet transform on the dataset given in *input*

fwt_I1D(`double*` *input*, `int` ℓ, `int` λ, `int` D, `double*` *output*)

Memory req	ℓ doubles
	Inverse 1D transform. Performs a simple fast inverse wavelet transform on the data set given in *input*

fwt_I2D(`double*` *input*, `int` ℓ_1, `int` ℓ_2, `int` λ_1, `int` λ_2, `int` D, `double*` *output*)

Memory req	$\ell_1 \cdot \ell_2$ doubles
	Inverse 2D transform. Performs a simple two-dimensional fast inverse wavelet transform on the dataset given in *input*

7.2 Serial routines

These routines always run in serial. Even on parallel architectures. They can be used for small datasets, where the overhead in spawning processes or threads would exceed the gain in speed achieved from the parallelization.

fwt_s_init(`void`)

Memory req	
	Initializes the LIBWAVELET subsystem for use with the serial routines.

fwt_s_finalize(`void`)

Memory req	
	Shuts down the LIBWAVELET environment.

fwt_s_F1D(`double*` *input*, `int` ℓ, `int` λ, `int` D, `double*` *output*)

Memory req	ℓ doubles
	Forward 1D transform. Performs a simple fast forward wavelet transform on the data set given in *input*.

fwt_s_F2D(`double*` *input*, `int` ℓ_1, `int` ℓ_2, `int` λ_1, `int` λ_2, `int` D, `double*` *output*)

Memory req	$\ell_1 \cdot \ell_2$ doubles
	Forward 2D transform. Performs a simple two-dimensional fast forward wavelet transform on the dataset given in *input*

fwt_s_I1D(`double*` *input*, `int` ℓ, `int` λ, `int` D, `double*` *output*)

Memory req	ℓ doubles
	Inverse 1D transform. Performs a simple fast inverse wavelet transform on the data set given in *input*

fwt_s_I2D(`double*` *input*, `int` ℓ_1, `int` ℓ_2, `int` λ_1, `int` λ_2, `int` D, `double*` *output*)

Memory req	$\ell_1 \cdot \ell_2$ doubles
	Inverse 2D transform. Performs a simple two-dimensional fast inverse wavelet transform on the dataset given in *input*

7.3 Symmetric Parallel routines

These routines are written specifically for shared memory multiprocessor architectures like the SGI Origin 2000 or multiprocessor workstations. We supply both 1D and 2D routines, but as mentioned earlier, the 1D routines are supplied for the sake of completeness only. One should not expect any substantial speedup from the 1D routines since they are very synchronization intensive.

fwt_smp_init(`int` *ncpu*)

Memory req	
	Initializes the libWavelet subsystem for use with the symmetric parallel routines, on a *ncpu* processor architecture. Afterwards, calls to the generic routines will run on *ncpu* processors, while specific calls to the symmetric parallel routines will still run on any specified number of processors.

fwt_smp_finalize(`void`)

Memory req	
	Shuts down the libWavelet environment.

fwt_smp_F1D(`double*` *input*, `int` ℓ, `int` λ, `int` D, `double*,output` , `int` *ncpu*)

Memory req	ℓ doubles
	Forward 1D transform. Performs a fast forward wavelet transform on the data set given in *input*. It will distribute the work over *ncpu* processors.

fwt_smp_F2D(`double*` *input*, `int` ℓ_1, `int` ℓ_2, `int` λ_1, `int` λ_2, `int` D, `double*` *output*, `int` *ncpu*)

Memory req	$\ell_1 \cdot \ell_2$ doubles
	Forward 2D transform. Performs a two-dimensional fast forward wavelet transform on the dataset given in *input*. It will distribute the work over *ncpu* processors.

fwt_smp_I1D(`double*` *input*, `int` ℓ, `int` λ, `int` D, `double*` *output*, `int` *ncpu*)

Memory req	ℓ doubles
	Inverse 1D transform. Performs a fast inverse wavelet transform on the data set given in *input*. It will distribute the work over *ncpu* processors.

fwt_smp_I2D(`double*` *input*, `int` ℓ_1, `int` ℓ_2, `int` λ_1, `int` λ_2, `int` D, `double*` *output*, `int` *ncpu*)

Memory req	$\ell_1 \cdot \ell_2$ doubles
	Inverse 2D transform. Performs a two-dimensional fast inverse wavelet transform on the dataset given in *input*. It will distribute the work over *ncpu* processors.

7.4 Massively Parallel routines

These are the routines written for massively parallel architectures, like the IBM SP2 or clusters of workstations. Since the 1D routines are very communication intensive, we do not supply them at all for the distributed memory architectures.

fwt_mpi_init(`int*` *argc*, `char**` *argv*, `MPI_Comm` *comm*)

Memory req	
	Initializes the LIBWAVELET subsystem for use with the massively parallel routines. Calls to generic routines will run on all machines described by the MPI communicator *comm*. The `argc` and `argv` arguments are taken from the `main()` `C` header.

fwt_mpi_finalize(`void`)

Memory req	
	Shuts down the LIBWAVELET environment.

fwt_mpi_F2D(`double*` *input*, `int` ℓ_1, `int` ℓ_2, `int` λ_1, `int` λ_2, `int` D, `double*` *output*, `MPI_Comm` *comm*)

Memory req	$\ell_1 \cdot \ell_2$ doubles per node
	Forward 2D transform. Performs a simple two-dimensional fast forward wavelet transform on the dataset given in *input*. It will distribute the work over all machines in the MPI cluster referred to by *comm*.

fwt_mpi_I2D(`double*` *input*, `int` ℓ_1, `int` ℓ_2, `int` λ_1, `int` λ_2, `int` D, `double*` *output*, `MPI_Comm` *comm*)

Memory req	$\ell_1 \cdot \ell_2$ doubles per node
	Inverse 2D transform. Performs a simple two-dimensional fast inverse wavelet transform on the dataset given in *input*. It will distribute the work over all machines in the MPI cluster referred to by *comm*.

7.5 Distributed data routines

These routines where introduced due to the fact that often parallelization is not needed because of speed, but because of the size of the data handled. On shared memory machines, we gain no capabilities in handling larger data sets by distributing the work among more processors. On distributed memory machines however, we will be able to let each processor process only a part of the entire domain, and having all the processors co-operating, we will be able to process very large domains that would otherwise have been too large for us to process.

Because these routines make no sense on shared memory machines, we only provide an implementation based on MPI, for use on massively parallel architectures. The routines assume that the `fwt_mpi_init()` routine has been called prior to their usage.

fwt_distrib_F2D(`double*` *input*, `unsigned` ℓ_1, `unsigned` ℓ_2, `unsigned` λ_1, `unsigned` λ_2, `unsigned` D, `double*` *output*, `MPI_Comm` *comm*)

Memory req	$\ell_1 \cdot \ell_2$ doubles
	This is the distributed fast forward 2D wavelet transform. It takes a sub-matrix, and cooperatively with other nodes holding other sub-matrices, it performs the 2D transform on the entire matrix.

fwt_distrib_I2D(`double*` *input*, `unsigned` ℓ_1, `unsigned` ℓ_2, `unsigned` λ_1, `unsigned` λ_2, `unsigned` D, `double*` *output*, `MPI_Comm` *comm*)

Memory req	$\ell_1 \cdot \ell_2$ doubles
	This is the distributed fast inverse 2D wavelet transform. It takes a sub-matrix, and cooperatively with other nodes holding other sub-matrices, it performs the 2D transform on the entire matrix.

Chapter 8

Conclusion

The algorithm developed by Ole Møller Nielsen proved efficient even when adapted from vector to cache coherent architectures. Using this a fullfledged library has been implemented and tested on a wide variety of systems:

Developing the serial routines of the library we experienced that hand-optimization gave tremendous speedup. Having spent about equal amounts of time optimizing and later parallelizing the wavelet code, we must conclude that in terms of pure speedup per man-hour the hand-optimization was far superior to the parallelization.

The shared memory version of the library shows good scalability on the Origin2000 platform. Scaling over more processors allows faster transform of large domains. It could be feasible to apply parallel wavelet transform in a real time application. However the speedup gained from adding more processors does not convince us that the money would not be better spent on a faster single processor machine.

The distributed memory version has not shown scalability anywhere near what we have seen on the Origin2000. This was somewhat expected since the network traffic is significant compared to the computational work. The network will simply introduce too much latency in the process to justify a parallelization for reasons of transform speed.

The dedicated distributed transform routines (`fwt_distrib_...`) were designed especially for the one case where a distributed memory computer system is advantageous; the case where the domain to transform is too large to fit into the memory of one computer.

All in all, the wavelet library can serve three purposes: It is *general purpose*, in the sence that it can easily be used on very different systems. It scales well on shared memory supercomputers, allowing fast transforms of large domains. Finally, it allows for transform of exorbitantly large domains, by distributing the domain over a cluster of machines.

It seems that there are two reasons for parallelization: The need for speed, or the need for treating large domains. However, our hand-optimization of the serial wavelet transform routines showed a much greater gain in speed, than we ever saw on the multiprocessor supercomputer moving to more processors.

Jakob Østergaard
c958612

Jakob Dalsgaard
c958213

Appendix A

Machines used

Following here, is a small description of the various machines and clusters we have used during the development, testing and optimization of the wavelet library.

Besides from the machines mentioned here, we have been able to compile and run the wavelet library on architectures like: DEC Alpha (running DEC UNIX), Intel 486 (running RedHat Linux) and a Sun UltraSPARC (running Solaris).

A.1 HP-PA Cluster

A cluster of 12 HP 9000 C180 workstations.

This cluster is the “G Databar” on DTU.

One Node	
CPU	180 MHz HP PA8000
CPU Cache	1 MB
Meas. perf.	35 MFLOPS
RAM	256 MB
Network	100Mbit FDDI

A.2 SGI Origin 2000

A shared memory multiprocessor supercomputer. This machine was kindly made available by UNI-C.

Total System	
CPU	16 × SGI R10000
CPU Cache	16 × 4 MB
Meas. perf.	16 × 48 MFLOPS
RAM	8 GB
Memory bw.	680 MB/sec.

A.3 IBM SP/2

A massively parallel supercomputer. This machine was kindly made available by UNI-C.

The sub-system we used	
CPU	4× IBM Power2 66 MHz
CPU Cache	4× 512 KB
Meas. perf.	? MFLOPS
RAM	256 MB
Memory bw.	? MB/sec.

A.4 Intel Dual PPro

A shared memory multiprocessor PC. This is one of the author's private machines.

Total System	
CPU	2 × Pentium Pro 166 MHz
CPU Cache	2 × 256 KB
Meas. perf.	39 MFLOPS
RAM	256 MB
Memory bw.	100 MB/sec.

A.5 Performance Measurement Tools

In order to model the performance on platforms for which documentation is either sparse or just hasn't been available to us, we developed two usefull tools to measure some essential hardware performance properties.

The performance of the CPU as stated by the vendors is usually a theoretical maximum performance, only guaranteeing what performance you will never reach. These numbers are of no use in a performance model.

Furthermore needed details of the hardware like cache details, memory subsystem details and general processor design are either well hidden or classified. A solution to these culprits was to make a special program that exercises the hardware in a clever way, so that the desired specifications show.

With the supplied `perftest` program one can estimate the following properties of the hardware:

- FPU/Register FLOP performance
- FPU/Cache FLOP performance
- Cache size
- Cache line size
- Cache miss cost

Furthermore the `pperftest` program estimates the following properties of the network:

- Network latency
- Network bandwidth

Appendix B

Build Instructions

This is a short introduction to compiling the wavelet library on different architectures, using different compilers. We put a lot of effort into making the build procedure as simple and painless as possible. A few requirements must however be met:

- The system must have a recent version of GNU Make
- The system must have a compiler capable of compiling ANSI `C`
- The automated `make` process uses the UNIX command `uname` to obtain information about the system. This command must also be available.

If one issues the standard UNIX command `make` standing in one of the two source directories, `./libWavelet/source/` or `./libWavelet/test`, make instructions will be printed.

B.1 Building on a PC

To build the library on a PC running the GNU/Linux operating system, the following scenario could take place:

```
> cd libWavelet/source
> make serial
Building serial version for Linux on i686
gcc -O3 -m486 -Wall -DPOSIX_THREAD -DOS=Linux -DHW=i686
 -c fwt.c -o fwt.o
gcc -O3 -m486 -Wall -DPOSIX_THREAD -DOS=Linux -DHW=i686
 -c daubechies.c -o daubechies.o
gcc -O3 -m486 -Wall -DPOSIX_THREAD -DOS=Linux -DHW=i686
 -c wavelet.c -o wavelet.o
ar rcvs libWavelet.a fwt.o daubechies.o wavelet.o
a - fwt.o
a - daubechies.o
a - wavelet.o
```

Since the build scripts detected we where running on a Linux system, it choose to use the `gcc` compiler which is standard on all Linux systems.

If we want to compile an MPI version, using the MPI compiler script `hcc`, which actually calls an underlying compiler called `egcs`, we would do the following:

```
> make mpi MC=hcc SC=egcs
Building MPI version for Linux on i686
hcc -O6 -mcpu=i686 -malign-double -Wall -DPOSIX_THREAD -DOS=Linux
```

```
 -DHW=i686 -DMPI -c fwt.c -o fwt.o
hcc -O6 -mcpu=i686 -malign-double -Wall -DPOSIX_THREAD -DOS=Linux
 -DHW=i686 -DMPI -c daubechies.c -o daubechies.o
hcc -O6 -mcpu=i686 -malign-double -Wall -DPOSIX_THREAD -DOS=Linux
 -DHW=i686 -DMPI -c wavelet.c -o wavelet.o
hcc -O6 -mcpu=i686 -malign-double -Wall -DPOSIX_THREAD -DOS=Linux
 -DHW=i686 -DMPI -c fwtmpi.c -o fwtmpi.o
hcc -O6 -mcpu=i686 -malign-double -Wall -DPOSIX_THREAD -DOS=Linux
 -DHW=i686 -DMPI -c fwtdistrib.c -o fwtdistrib.o
ar rcvs libWavelet.a fwt.o daubechies.o wavelet.o fwtmpi.o fwtdistrib.o
a - fwt.o
a - daubechies.o
a - wavelet.o
a - fwtmpi.o
a - fwtdistrib.o
```

It's as simple as that !

B.2 Building on the Origin2000

When building the shared memory (SMP) versions of the library, one has several options when it comes to choosing the number of processors the library should be built for.

- One can specify the number with an SMP=n switch to the `make` command
- If one builds the library on either an Origin2000 or a Linux system, the make scripts are able to auto-detect the number of available processors, thus defaulting to this number
- Finally, one can just choose to go with the default of two processors.

If one wishes to build the library for the Origin2000, using four processors, the following scenario would take place:

```
> gmake smp SMP=4
Building SMP version for IRIX64 on IP27
User-specified number of CPUs: 4
cc -Ofast -r10000 -DIRIX_LIGHTWEIGHT_PROCESS -DOS=IRIX64
 -DHW=IP27 -DSMP=4 -c fwt.c -o fwt.o
cc -Ofast -r10000 -DIRIX_LIGHTWEIGHT_PROCESS -DOS=IRIX64
 -DHW=IP27 -DSMP=4 -c daubechies.c -o daubechies.o
cc -Ofast -r10000 -DIRIX_LIGHTWEIGHT_PROCESS -DOS=IRIX64
 -DHW=IP27 -DSMP=4 -c wavelet.c -o wavelet.o
cc -Ofast -r10000 -DIRIX_LIGHTWEIGHT_PROCESS -DOS=IRIX64
 -DHW=IP27 -DSMP=4 -c fwtsmp.c -o fwtsmp.o
ar rcvs libWavelet.a fwt.o daubechies.o wavelet.o fwtsmp.o
a - fwt.o
a - daubechies.o
a - wavelet.o
a - fwtsmp.o
```

As the attentive reader will notice, the GNU make utility on the system in the example above, is installed under the command name `gmake`.

B.3 A sample program

This is an example of a short program using a few of the generic Fast Wavelet Transform library routines. The program will run on single workstations, on cluster machines, and on shared memory machies, without one single change in the code.

```
#include "wavelet.h"
#include "filetools.h"
int main(int argc, char** argv) {
 double * domain;
 unsigned m,n;

 /* Initialize library subsystem */
 fwt_init(&argc,&argv);

 /* Read input image */
 pgm_raw_to_matrix(&domain, &m, &n, "test.pgm");

 /* Transform image to maximum depth, using wavelet genus 4 */
 fwt_F2D(domain, m, n, 0, 0, 4, domain);

 /* Save domain */
 matrix_to_file(domain, m, n, "output.text");

 /* Terminate wavelet environment */
 fwt_finalize();

}
```

The above program reads a `PGM` (portable gray-map) image file, performs a forward wavelet transform on the image data, and saves the resulting domain in a text file, easily readable by MatLab.

To compile the program, one could issue the following command, assuming a serial version of the library, and assuming the library is found in the current directory:

```
>  cc -o test test.c -lWavelet -lm
```

The compile process is heavily dependent on the current platform. A peek into the makefiles supplied with the Wavelet Library should give a hint or two about how to compile programs for different platforms and architectures.

Bibliography

[OMN98] **Wavelets in Scientific Computing**, IMM-PHD-1998-43, *Ole Møller Nielsen*, February 1998, IMM & UNI-C, DTU.

[Int97] **Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide**, Order Number 243192, 1997 Intel Corp.

[GS93] **Wavelet Transforms versus Fourier Transforms**, *Gilbert Strang*, Bulletin of the AMS, April 1993, v.28, n.2, pp288-305

[SGIweb] `http://www.sgi.com/Works/EA/caeforum/Prod/sgi/hardware/servers/products/Irix.html`

[SGIinsight] **SGI Insight** on-line documentation on the SGI Origin 2000, write `insight` on the command line.

[HPC180] **HP C180 Data sheet** `http://www.hp.com/wsg/products/cclstech.html`

[MPI] **The MPI Standard** `http://www.mcs.anl.gov/mpi/mpi-report-1.1/mpi-report.html`

[GLS] **Using MPI** *William Gropp, Ewing Lusk, Anthony Skjellum*, The MIT Press, 1994

[USP] **UNIX Systems Programming (for SVR4)** *David A. Curry* O'Reilly & Associates