High Dimensional Wavelet Transforms on MIMD Architectures

Diplomarbeit
zur Erlangung des Diplomgrades
an der Naturwissenschaftlichen Fakultät
der Universität Salzburg

Rade Kutil
Salzburg, 03.2.2000

Contents

List of Figures

List of Pseudo Codes

Chapter 1

Introduction

1.1 What and Why

The goal of signal processing is to find simple methods that enable a scientist or an engineer to perform whatever operations on signals he wants to. An important class of signals are time invariant (stationary) signals. The best way to describe such signals is in terms of frequency. This means that the signal is decomposed into a finite number of oscillating basis functions, and described by the corresponding coefficients. This procedure is the well known Fourier transform. Its advantage is that each linear time-invariant operation (like for instance low pass filtering) can be performed easily in the frequency domain by pointwise modification of frequency coefficients.

But time-invariant signals are not the only signals in this world. Most signals of interest are limited to finite intervals in time or space. Other ways have to be found to deal with such signals. Another problem is that the advantages of frequency based analysis should not be lost.

With this background, approaches like the short time Fourier transform (STFT) and finally the wavelet transform have been developed. Here bounded functions which look like short waves (hence the name) serve as basis functions, whose coefficients describe the frequency behaviour of a signal as it changes in time (or space). Unfortunately, there is the *uncertainty principle* (Heisenberg) which says that there is a limit for time and frequency localization of a signal. To be more precise, if the signal is viewed as concentrated around a point in time and a point in the frequency domain, the product of its variances (in time and frequency respectively) has a minimum of $\frac{1}{4}$. So the better the frequency localization, the worse the time localization – and vice versa. For wavelet analysis this means that there has always to be a tradeoff when choosing the wavelets to work with.

Experimental results show that there is a great potential of entropy decrease, information concentration and feature extraction in transforming signals into the wavelet domain. Therefore, the wavelet transform has proved its usefulness in many applications (just as the Fourier transform did) like: image processing, signal/image compression, feature extraction, classification algorithms, telecommunication applications, numerical mathematics and many more.

Several possible extensions make the wavelet transform even more interesting:

Multidimensional data like images can be decomposed into multidimensional wavelets as base functions.

Wavelet packets convert simple wavelet bases into a variety of combined bases with any desired time and frequency resolution in a very easy way.

Best basis selection is an efficient method to choose the wavelet packet base that fits best into given needs.

Another advantage is the excellent performance of the wavelet transform. The existence of a fast wavelet transform (FWT) has made the wavelet transform attractive for many practical applications, just as the fast Fourier transform (FFT) did. Note that the FFT manages to reduce the computational complexity from $O(n^2)$ to $O(n \log n)$, whereas the FWT reduces the computational complexity from $O(n \log n)$ to linear complexity $O(n)$.

Nevertheless, there are applications where the fast wavelet transform is still not fast enough, as for example realtime video coding. This gives the motivation for this work. The need for the processing of large 3-D (or even higher dimensional) data (like digital video) and the need for high performance lead to the use of general purpose high performance computers or networks of computers. Note that in this work the 3-D case (video) is used as an example for the n-dimensional case and shall give all the ideas for implementation, the 1-D or 2-D case cannot because of reduced complexity.

An enormous work has been done to adapt important algorithms to the most common network topologies that build the multiprocessor core of a high performance computer system, such as star shaped, n-dimensional meshes and hypercube topologies. But nowadays the importance of the type of network topology decreases and seems to be replaced by the interconnection structure, i.e. the technology used for interconnection of processor elements. Therefore, different computer system types are explored as candidates for performing an efficient wavelet transform:

- Distributed memory multicomputers
- Shared memory multiprocessors
- Networks of common single or dual processor computers (clusters)

Also different programming paradigms are tested for performance as well as implementation ease:

- Message passing
  - PVM (parallel virtual machine)
  - MPI (message passing interface)
- Shared memory programming

1.2 History

1.2.1 The Discovery of Wavelets

The main branch of mathematics leading to wavelets began with Joseph Fourier (1807) with his theories of frequency analysis, now referred to as Fourier analysis. After that it took a long time to develop funded theories of functional analysis, before one started to build time and frequency localized basis functions (wavelets). The first mention of a sort of wavelets appeared in an appendix to the thesis of A. Haar (1909). One property of the Haar wavelet is that it has compact support which means that it vanishes outside a finite interval. Unfortunately Haar wavelets are not continuously differentiable which somewhat limits their applications. In the early eighties non-trivial wavelets were discovered (Mallat [36], Meyer, Daubechies [12]) by a fusion of ideas that had been developed independently in engineering, physics and pure mathematics (quadrature mirror filters, pyramid algorithms, multiresolution analysis) (see also [13, 37]). From then on many applications of wavelets have been found in signal analysis and numerical analysis.

1.2.2 Recent Extensions

Wavelet packets [48] represent a generalization of the method of multiresolution decomposition and comprise the entire family of subband coded (tree) decompositions. Whereas in the wavelet case the decomposition is applied recursively to the coarse scale approximation only (leading to the well known (pyramidal) wavelet decomposition tree), in the wavelet packet decomposition the recursive procedure is applied to all subbands (i.e. coarse scale approximation and detail signals) which leads to a complete wavelet packet tree and more flexibility in frequency resolution.

A given data set may be represented by different parts of a wavelet packet tree in many different ways. There are several possibilities how to determine the subbands suited well for an application. The meaning of *suitable* depends on the type of application, e.g. signal/image compression [50], feature extraction [42], classification algorithms [32, 33], telecommunication applications [35], numerical mathematics [38], and many more. The wavelet packet "best basis algorithm" [9] performs an adaptive optimization of the frequency resolution of a complete wavelet packet decomposition tree by selecting the most suitable frequency subbands with respect to an additive cost function. The same algorithm employed with non-additive cost function is denoted "near-best basis algorithm" [44], if the subband structure is restricted to uniform time-frequency resolution the corresponding algorithm is denoted "best level selection" [8].

1.2.3 Applications in Image Processing

In recent years there has been a tremendous increase in the demand for digital imagery. Applications include consumer electronics (Kodak's Photo-CD, HDTV, SHDTV, and Sega's CD-ROM video game), medical imaging (digital radiography), video-conferencing and scientific visualization. The problem inherent to any digital image or digital video system is the large bandwidth required for transmission or storage.

Image and video coding methods that use wavelet transforms have been successful in providing high rates of compression while maintaining good image quality and have generated much interest in the scientific community as competitors to DCT based compression schemes in the context of the MPEG-4 and JPEG2000 standardization process.

In the context of image compression a more advanced technique is to use a framework that includes both rate and distortion, where the best basis subtree which minimizes the global distortion for a given coding budget is searched [41]. Other methods use fixed bases of subbands for similar signals (e.g. fingerprints [22]) or search for good representations with genetic optimization methods [6, 4].

1.3 Related Works

In the context of MPEG-1,2,4 and H.261 several papers have been published describing real-time video coding on general purpose high performance computers [1, 14, 2, 3, 21].

Most video compression algorithms rely on 2-D based schemes employing motion compensation techniques. On the other hand, rate-distortion efficient 3-D algorithms exist which are able to capture temporal redundancies in a more natural way (see e.g. [34, 23, 19, 45] for 3-D wavelet/subband coding). Unfortunately these 3-D algorithms often show prohibitive computational and memory demands (especially for real-time applications). At least, prohibitive for a common microprocessor. A shared memory MIMD architecture seems to be an interesting choice for such an algorithm.

A significant amount of work has been already done on parallel wavelet transform algorithms for all sorts of high performance computers. We find various kinds of suggestions for 1-D and 2-D algorithms on MIMD computers (see e.g. [49, 43, 40, 24, 18, 25] for decomposition only and [31, 11] for algorithms in connection with image compression schemes). On the other hand, the author is not aware of any work (except [39]) focusing especially on 3-D wavelet decomposition and corresponding 3-D wavelet based video compression schemes.

Whereas an enormous amount of work has been already done concerning the development and thorough analysis of parallel algorithms for the fast wavelet transform, only few papers have been devoted to parallel wavelet packet decomposition and its specific features and demands (e.g. approaches for performing the best basis algorithm and the irregular decomposition into such a basis on parallel MIMD [20, 46, 47, 16] and SIMD [15, 5] architectures and application of parallel wavelet packet decomposition in numerics [10, 38]).

As part of this work several articles appeared in conference proceedings [27, 26, 28] and journals [29, 30].

1.4 The Environment of Experimentation

As environment for experimental results, the following computer architectures will be in use:

Cray T3E Located at NIC Jülich. 512 compute nodes type 1200, cycle time 1.66 ns, 1200 MFLOPS/processor.

Message passing library: Native MPI implementation.

Siemens Cluster A hpcLine-Cluster at Paderborn Center for Parallel Computing. 96 Siemens Primergy compute nodes, each with 2 Intel Pentium II, 450 MHz, 512 MByte memory and 4 GByte local hard disk. 96 Dolphin PCI/SCI interfaces with 500 MByte/s SCI-Link bandwidth to form a 2D 12x8 torus with distributed switches.

Message passing library: Native MPI implementation: ScaMPI release 1.9.1.

SGI Powerchallenge SGI Powerchallenge GR (at RIST++, Salzburg Univ.) with 20 MIPS R10000 processors and 2.5 GB shared memory.

Message passing libraries: SGI MPI 3.1.1.0, PVM 3.3.10.

1.5 Road Map

Chapter 2 will introduce the mathematical motivation and details of the wavelet decomposition. Starting with the Fourier transform, whose importance for all parts of science is well known, the need for locally bounded base functions will lead to wavelets. The background (multiresolution analysis) for the fast wavelet transform (FWT) will be explained as well as the (good) algorithmical complexity of the FWT.

Chapter 3 will show, how the 3D-wavelet transform can be implemented on single processor computers. This will serve as a basis to understand the parallel algorithms and introduces optimizations of parts of the problems that will appear again in chapter 4.

Chapter 4 will parallelize step by step the sequential algorithms. Different programming paradigms (message passing, shared memory programming) will be used as well as different data and work distributions. The results (with respect to computational performance) will be compared on several computer platforms.

Chapter 2

Wavelet Analysis

2.1 From FT over STFT to wavelets

Wavelet analysis has to cope with function spaces just as the Fourier analysis. Here we discuss a function space V that forms a Hilbert space with the following operations: V is a Hilbert space over $\mathbb{R}$ with

$$\begin{aligned}\langle f|g\rangle &:= \int_{-\infty}^{+\infty} f(t)g(t)dt \\ (f+g)(t) &:= f(t)+g(t) \\ (\lambda f)(t) &:= \lambda f(t)\end{aligned}$$

The most commonly used function space is $L^2(\mathbb{R}) := \{f : \mathbb{R} \to \mathbb{R} \,|\, \int_{-\infty}^{+\infty} f^2(t)dt < \infty\}$.

The problem is now to find suitable (orthonormal) bases for this space. The most common base is $\{t \to e^{i\omega t} \,|\, \omega \in \mathbb{R}\}$. For this base, computing the coefficients

$$\hat{f}(\omega) = \int_{-\infty}^{+\infty} f(t)e^{-i\omega t}dt$$

is called Fourier transform. Note that for real valued functions $\hat{f}(\omega) = \overline{\hat{f}(-\omega)}$. Although the base functions are not elements of L^2, the reverse transform is given by

$$f(t) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} \hat{f}(\omega)e^{i\omega t}d\omega.$$

For the subspace of τ-periodic functions (or functions with $\mathrm{supp}(f) \subseteq [0, \tau]$) where typically $\tau = 2\pi$ it suffices to limit ω to $2\pi n$ where $n \in \mathbb{Z}$. Then the transform

$$\hat{f}(n) = \int_0^{2\pi} f(t)e^{-i2\pi nt}dt$$

gives the coefficients for the Fourier series

$$f(t) = \frac{1}{2\pi} \sum_{n=-\infty}^{\infty} \hat{f}(n)e^{i2\pi nt}.$$

The Fourier coefficients are very important for frequency based operations. To be more precise, a linear time invariant operator is entirely specified by its eigenvalues $\hat{h}(\omega)$ and $e^{i\omega t}$ are the eigenvectors. Therefore the domain of ω is called frequency domain and $|\hat{f}|$ is the frequency spectrum.

However, there are operations needed that are not time invariant. Because certain properties of signals are localized around a certain point in time, one has to find base functions that are well localized in time domain. On the other hand one does not want to lose the advantages of frequency localization. Heisenberg shows, that there cannot be a base function, that has finite support in time domain as well as in the frequency domain. Therefore, one has to find a compromise.

One method is to use a so called time window g, that is concentrated around zero. By translating g along the time axis, the Fourier base functions can be *windowed* to give the new base functions (time frequency atoms)

$$g_{u,\xi}(t) = g(t-u)e^{i\xi t}.$$

The localization of these functions in time and frequency domain can be controlled by the choice of g and is optimal if g is Gaussian. In this case the $g_{u,\xi}$ are called Gabor functions (Gabor atoms). The transform of a function f into the coefficient space (STFT – short time Fourier transform) is then given by:

$$Sf(u,\xi) = \langle f | g_{u,\xi} \rangle = \int_{-\infty}^{+\infty} f(t)\overline{g_{u,\xi}(t)}dt = \int_{-\infty}^{+\infty} f(t)g(t-u)e^{-i\xi t}.$$

The main disadvantage of this scheme is, that in many practical applications single signals have shorter duration if they have higher frequency components. For this reason, time windows are preferred, which shrink proportionally to the frequency scale of the base functions. It is then possible to retain the shape of the base functions at all translations and dilations. The base functions are then called wavelets. To be more precise, a so called mother wavelet is a function ψ with

$$\int_{-\infty}^{+\infty} \psi(t)dt = 0$$

from which all wavelets can be derived by dilation with a scale parameter s and translation by u:

$$\psi_{u,s}(t) = \frac{1}{\sqrt{s}}\psi\left(\frac{t-u}{s}\right).$$

The wavelet transform is then given by

$$Wf(u,s) = \langle f|\psi_{u,s}\rangle = \int_{-\infty}^{+\infty} f(t)\overline{\frac{1}{\sqrt{s}}\psi\left(\frac{t-u}{s}\right)}dt.$$

In most cases $u \in \mathbb{Z}$ and $s = 2^j$, where $j \in \mathbb{Z}$, so we can redefine

$$\psi_{i,j}(t) = \frac{1}{\sqrt{2^j}}\psi\left(\frac{t-2^j i}{2^j}\right) = \frac{1}{\sqrt{2^j}}\psi\left(\frac{t}{2^j} - i\right).$$

$$Wf(i,j) = \langle f|\psi_{i,j}\rangle$$

2.2 Multiresolution Analysis

The idea of approximating a given function successively by increasing the frequency scale of the time frequency atoms leads to the development of a sequence of subspaces $\{V_j\}_{j\in\mathbb{Z}}$ that give successive approximations of $L^2(\mathbb{R})$. Such a sequence of subspaces (multiresolution analysis) has to satisfy the following conditions:

$$\forall j \in \mathbb{Z}: \quad V_{j+1} \subset V_j$$

$$\forall i \in \mathbb{Z}: \quad f \in V_0 \Leftrightarrow f(\cdot - i) \in \mathbb{Z}$$

$$\forall j \in \mathbb{Z}: \quad f \in V_j \Leftrightarrow f\left(\frac{\cdot}{2}\right) \in V_{j+1}$$

$$\lim_{j \to +\infty} V_j = \{0\}$$

$$\lim_{j \to -\infty} V_j = L^2(\mathbb{R})$$

$$\exists \phi \in V_0 : \quad \{\phi(\cdot - i) \,|\, i \in \mathbb{Z}\} \quad \text{is an orthonormal basis of} \quad V_0$$

Now one can show, that for each multiresolution analysis, there exists an orthonormal basis of wavelets $\{\psi_{i,j}\}$, such that

$$V_{j-1} = V_j \oplus W_j,$$

where $W_j = [\{\psi_{i,j} \,|\, i \in \mathbb{Z}\}]$. Moreover, as ϕ can be constructed out of the $\phi_{-1,j}$ by

$$\phi = \sum_j h_j \phi_{-1,j},$$

where $h_n = \langle \phi | \phi_{-1,j} \rangle$ (because $\{\phi_{-1,j}\}$ is an orthonormal basis of $V_{-1} \supset V_0 \ni \phi$), ψ can be constructed in the same way:

$$\psi = \sum_j g_j \phi_{-1,j}.$$

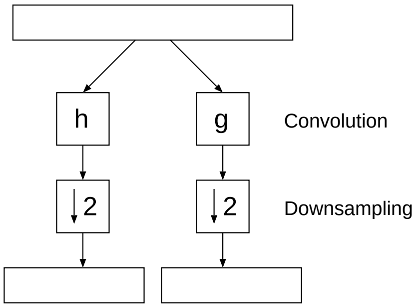
These g_j turn out to be given by

$$g_j = (-1)^j h_{1-j}.$$

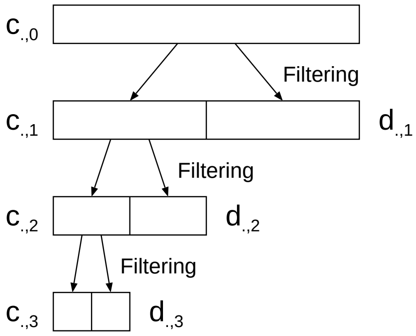
Note, that $\{\phi(\cdot - i) \,|\, i \in \mathbb{Z}\}$ does not have to be an orthonormal basis. If it is a *Riesz basis*, the wavelet transform produces a stable but not perfect reconstruction (inverse transformation).

2.3 Fast Wavelet Transform (FWT)

The multiresolution analysis offers big advantages for practical implementations of the wavelet decomposition in the case, that the wavelet has finite support. Let us first define $c_{i,j} = \langle f | \phi_{i,j} \rangle$ and $d_{i,j} = \langle f | \psi_{i,j} \rangle$, so that the wavelet transform is the mapping $f \mapsto d$. If ϕ and ψ are part of a multi resolution analysis as defined above, the projection of f onto the subspace



(a) Filtering



(b) Decomposition

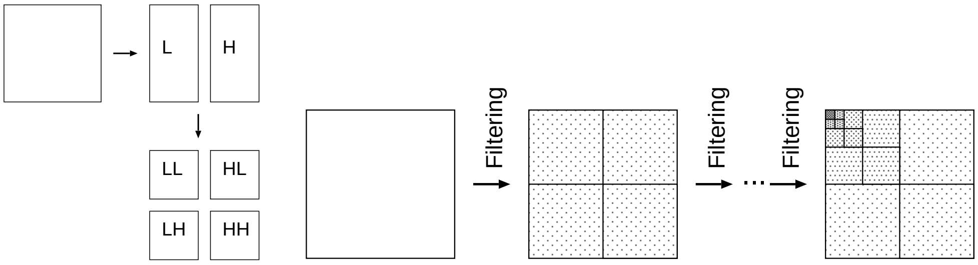
Figure 2.1: The fast wavelet transform

V_j is given by $d_{\cdot,j}$. Because of the properties of multiresolution analysis, the coefficients of the next coarser level $(j+1)$ are given by

$$c_{i,j+1} = \sum_{k\in\mathbb{Z}} c_{2i+k,j} h_k$$
$$d_{i,j+1} = \sum_{k\in\mathbb{Z}} c_{2i+k,j} g_k.$$

The $d_{\cdot,j}$ represent a data sequence, that approximates the function f. The approximation is coarser, if j is higher. The given decomposition step is essentially a convolution of the data sequence $d_{\cdot,j}$ with the sequences h and g followed by a downsampling by 2. The whole decomposition process is then completed, by continuing recursively to decompose the sequence $c_{\cdot,j+1}$ in the same way. This is illustrated in Figure 2.1. The sequences h and g are called low pass and high pass filters respectively, because they *divide* the original *signal* $c_{\cdot,j}$ into a low frequency part $(c_{\cdot,j+1})$ and a high frequency part $(d_{\cdot,j+1})$. Note that $d_{i,j}$ is the coefficient for the wavelet $\psi_{i,j}$, which contains lower frequency components as j increases. Because of their special properties h and g are called quadrature mirror filters (QMF).

If the wavelet ψ has finite support, then also only a finite number l of filter coefficients are non-zero. Therefore one decomposition step of a data sequence of length n requires $2\sum_{i=1}^{n/2}\sum_{k=1}^{l} 1 = nl$ multiplications. In the next decomposition step this number is halfed, leading to a total number of $nl\sum_{j=0}^{\infty} 2^{-j} = 2nl$ multiplications for the whole decomposition, which shows, that the FWT reduces the computational complexity from $O(n\log n)$ to linear complexity $O(n)$. Note that the fast Fourier transform (FFT) manages to reduce the complexity from $O(n^2)$ to $O(n\log n)$.



(a) Filtering (b) Decomposition

Figure 2.2: Fast wavelet transform of 2-D data

2.4 3-D Wavelet Transform

Let us first look at the 2-D case. A 2-D wavelet base for $L^2(\mathbb{R})$ is constructed with the 1-D wavelet ψ and the scaling function ϕ by

$$\begin{aligned} \psi^1(\boldsymbol{x}) &= \psi(x_1)\phi(x_2), \\ \psi^2(\boldsymbol{x}) &= \phi(x_1)\psi(x_2) \quad \text{and} \\ \psi^3(\boldsymbol{x}) &= \psi(x_1)\psi(x_2), \end{aligned}$$

where $\boldsymbol{x} = (x_1, x_2)$ and the derived wavelets

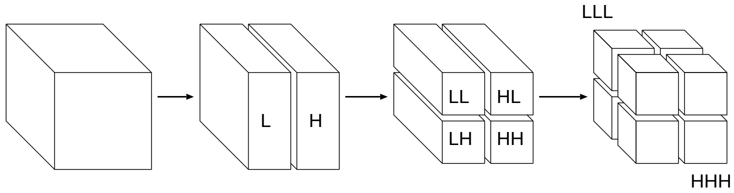
$$\psi^k_{\boldsymbol{i},j}(\boldsymbol{x}) = 2^j \psi^k(2^j \boldsymbol{x} - \boldsymbol{i}) \quad \boldsymbol{i} \in \mathbb{Z}^2.$$

The fast wavelet transform (in terms of QMF) can then be implemented by a columnwise 1-D filtering followed by a rowwise 1-D filtering resulting in four new subbands as shown in Figure 2.2(a). The subband that results from lowpass filtering in both dimensions is called approximation subband and is decomposed in the same way, leading to the pyramidal decomposition structure as shown Figure 2.2(b). Each of the other subbands correspond to the wavelets ψ^1, ψ^2, ψ^3.

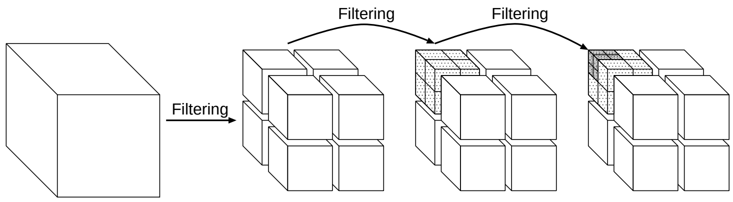
The 3-D case is analogous. There are seven wavelets ψ^k varying in three variables t, x, y. The original data (a cuboid) is filtered in three steps, one for each dimension, as shown in Figure 2.3(a). The approximation subband is decomposed further just as in the 2-D case (see Figure 2.3(b)).

2.5 Wavelet Packets

Wavelet packets are a generalization of wavelets. The idea is based on the following fact: If $\{\theta(\cdot - i) \,|\, i \in \mathbb{Z}\}$ is an orthonormal basis for a space U and



(a) Filtering



(b) Decomposition

Figure 2.3: Fast wavelet transform of 3-D data

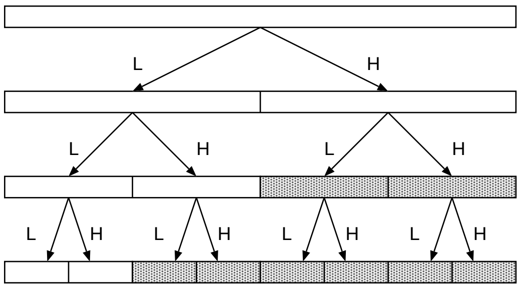


Figure 2.4: Wavelet packet decomposition

g, h a pair of conjugate mirror filters, then

$$\left\{ \sum_{k \in \mathbb{Z}} h_k \theta(\cdot - k - 2i) \,\middle|\, i \in \mathbb{Z} \right\} \cup \left\{ \sum_{k \in \mathbb{Z}} g_k \theta(\cdot - k - 2i) \,\middle|\, i \in \mathbb{Z} \right\}$$

is also an orthonormal basis of U.

This means, that we can also divide the high frequency subbands into two subbands just as with the approximation subband. Because these new subbands can also be divided further we get an entire decomposition tree as shown in Figure 2.4

2.6 Best Basis

As we can now choose, which basis we want to use for the transform, i.e. which subband should be decomposed and which not, we have to find some criterion, to decide this. There are different aspects of the data, that are important for different applications of wavelet transform. So it is best to define an abstract cost function $\mathrm{cost} : L^2(\mathbb{Z}) \to \mathbb{R}$, that determines a measure of (negative) usability for a given data sequence with respect to a given purpose.

The basis can be represented by a decomposition tree T. The vertices of T represent the subbands where the root vertex is the original data sequence s and the leaves are the subbands that form the transformed data c. Now we can define the best basis as the tree T that minimizes $\mathrm{cost}(c)$.

As an example let us look at data compression applications. A cost function in this case should measure the concentration of information, i.e. it should give large values when all but a few coefficients are negligible. A well known example for this is the *entropy*:

$$H(c) = -\sum_i p_i \log p_i, \quad \text{where} \quad p_i = \frac{|c_i|^2}{||c||^2}.$$

Because the number of possible trees is of an order of about $O(2^{n/2})$ in the 1-D case, if the maximum decomposition depth is of order $O(\log_2 n)$, it would mean a lot of work to test all of them to find the best one. Fortunately there is another possibility: If the cost function is additive, i.e. $\mathrm{cost}(c^1 \cup c^2) = \mathrm{cost}(c^1) + \mathrm{cost}(c^2)$, then the branching decision at a certain vertex of the tree depends only on the vertices within the corresponding subtree. To be more precise: The tree has to branch for subband a, if $\mathrm{cost}(a) > \mathrm{bestcost}(a)$, where

$$\mathrm{bestcost}(a) = \begin{cases} \mathrm{cost}(a) & \text{max. decomposition depth reached} \\ \mathrm{detcost}(a) & \mathrm{detcost}(a) < \mathrm{cost}(a) \\ \mathrm{cost}(a) & \mathrm{detcost}(a) \geq \mathrm{cost}(a) \end{cases}$$

and

$$\mathrm{detcost}(a) = \sum_{b \in \mathrm{child}(a)} \mathrm{bestcost}(b).$$

Unfortunately the entropy H from above is not additive, but there exists an additive analogue $\lambda(c) = -\sum_i |c_i|^2 \log |c_i|^2$, which satisfies the same conditions. Note that the same algorithm applied on non additive cost functions is referred to as *near best basis* algorithm.

Chapter 3

Sequential Algorithms

This chapter explains the sequential algorithms that are on the one hand the basis for the development and understanding of the parallel algorithms and on the other hand part of the parallel algorithms, as the type of algorithms investigated here often offer the possibility to distribute the data in a way so that the sequential algorithms work on local data.

Another important aspect of the sequential algorithms is that the parallelization only makes sense, if the sequential algorithms are optimized well enough. Note that the most commonly used and most realistic measure for parallel performance is the speedup compared to the fastest known sequential algorithm (effective speedup).

3.1 Preliminaries

The development of the algorithms leads to frequently used constructs, which should be introduced as abstract formalisms, to make the subsequent pseudo codes shorter and more understandable.

The 3-D array we work on uses indices out of the space $[0 \ldots t_{max} - 1] \times [0 \ldots x_{max} - 1] \times [0 \ldots y_{max} - 1]$. A subband is then defined as a subcuboid with the six characteristic values $(t_{min}, t_{max}, x_{min}, x_{max}, y_{min}, y_{max})$, which specifies the subspace $[t_{min} \ldots t_{max} - 1] \times [x_{min} \ldots x_{max} - 1] \times [y_{min} \ldots y_{max} - 1]$ of indices. This is called *rectangular parallelepiped* or *6-top* in computational geometry. The functions `TRange`, `XRange` and `YRange` are defined to give the range $t_{min} \ldots t_{max} - 1$, $x_{min} \ldots x_{max} - 1$ and $y_{min} \ldots y_{max} - 1$ respectively.

First we have to cope with decomposition trees, where the parent-child relation is very important. Therefore, we define the function `Child (S, ChildNr)` which returns for a subband `S` the subband of `S` that is numbered (`ChildNr`) as shown in Figure 3.1. An element of a subband can be referenced as

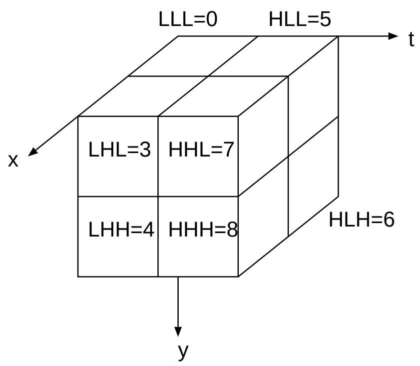


Figure 3.1: Numbering of subbands for reference purposes

`S[t,x,y]`, which gives the element of the array with indices $(t_{min} + t, x_{min} + x, y_{min} + y)$.

After that we can define what is meant by the so called tree walk. If we take a subband S, then all its subbands form a subtree. Experiences show that it is best to walk through this tree in depth first order. This can be done by a recursive procedure call or by use of a stack. As an example, pseudo code 1 shows the implementation of the second variant. The code "BeforeCode" is called, whenever the tree-walk first reaches a certain subband T and the code "AfterCode" is called after the subtree of T is completely processed i.e. before proceeding with the brothers of T.

To store the information about the structure of decomposition trees, one has to define a bittree `BT`, i.e. a map from the subbands to a true/false value, which specifies, if the decomposition tree branches at a specific subband. These values are set by the commands `Branch (BT, S)` and `CutOff (BT, S)` and evaluated by `Branches (BT, S)`.

There are also some procedures necessary to handle the cost measurement for best basis decomposition. `Cost (S)` calculates the cost for a specific Subband `S`. For such cost values addition is supposed to be defined (remember that the cost function should be additive). Likewise a cost value can be set to zero.

3.2 Filtering

The most performance critical parts of an algorithm are the innermost loops. In this case the convolution of the filters with data sequences are the subject of interest. Here the subbands are generated, which will be the objects of abstraction in later investigations.

```
for T in SubTree (S, MaxLevel)
    before: BeforeCode
    after:  AfterCode
```

is defined by:

```
Empty Stack
push (S, 0, 0, 0) onto Stack
while Stack not empty
    pop (T, Action, Level, ChildNr) from Stack
    if Action = 0
        push (T, 1, Level, ChildNr) onto Stack
        perform BeforeCode
        if CutSubtree not called and Level < MaxLevel
            for ChildNr in 1..8
                push (Child (T, ChildNr), 1, Level+1, ChildNr)
                  onto Stack
    if Action = 1
        perform AfterCode
```

Pseudo code 1: Walking through a decomposition tree in depth first order by use of a stack

3.2.1 1D-Filtering

The filtering of 1-D data is a simple thing and is implemented in pseudo code 2, where a subband `S` is filtered by the filters `G` and `H` and downsampled by two. The results reside in the same array `S` – to be more precise: in the first and second half of `S`.

```
Filter1D (S, G, H) :=
    l = length (S)
    f = length (G)
    T = Pad (S)
    for i in 0..l/2-1
        S[i] = 0; S[l/2+i] = 0
        for j in 0..f-1
            S[i]     = S[i]     + T[2i+j] * H[j]
            S[l/2+i] = S[l/2+i] + T[2i+j] * G[j]
```

Pseudo code 2: Filtering of the 1-D array S (of length l) with filters H and G (of length f) and downsampling by 2 resulting in the new 1-D subbands S[0..l/2-1] and S[l/2..l-1]

Padding

A very important point is the function `Pad`, called in pseudo code 2. One can see that `T` is accessed in an index range of $0..l+f-3$, which means that `T` is usually longer than `S`, so a simple copy is not sufficient. This problem comes from the fact that we are using finite data. There are several possibilities to solve this problem:

- Cyclic Padding implies
  `T[0..a-1]=S[l-a..l-1]`,
  `T[a..l+a-1]=S[*]` and
  `T[l+a..l+f-3]=S[0..f-a-3]`,
  where $a \in \{0 \ldots f-2\}$ can shift the data to bring the coefficients to the appropriate position. The problem here is that the transition from the end to the start of the data causes unwanted irregularities with respect to smoothness.

- Symmetric padding with even filter length implies
  `T[0..f/2-2]=S[f/2-2..0]` (reverse!),
  `T[f/2-1..l+f/2-2]=S[*]` and

T[l+f/2-1..l+f-3]=S[l-1..l-f/2+1].

This scheme can only be used with symmetric filters, others would cause a loss of data. The smoothness at the border is improved in the sense that data is continuous but possibly not derivable.

- Symmetric padding with uneven filter length implies
  T[0..(f-1)/2-2]=S[(f-1)/2-1..1],
  T[(f-1)/2-1..l+(f-1)/2-2]=S[*] and
  T[l+(f-1)/2-1..l+f-3]=S[l-2..l-(f-1)/2+1].
  The same arguments as with even filter lengths apply here.

- Using special filters at the border (see [7]). Here the scheme of padding is not used at all.

- Extended Array: Filling in other data assumes that S is also defined outside of the index range $0 \ldots l-1$. Then T can be set to
  T[0..l+f-3]=S[-a..l+f-a-3].
  This is used in the parallelization of the algorithm and has to be combined with the other possibilities.

Reconstruction

The reconstruction is the reverse procedure to the filtering. Pseudo code 3 implements it. The procedure Zero sets the array T to zero, which has the same size as T in Filter1D. The function RePad corresponds to Pad: The parts of T that were duplicated regions of S in Pad have to be added to these regions in RePad.

```
ReFilter1D (S, G, H) :=
    l = length (S)
    f = length (G)
    Zero (T)
    for i in 0..l/2-1
        for j in 0..f-1
            T[2i+j]  = T[2i+j] + S[i] * H[j] + S[l/2+i] * G[j]
    S = RePad (T)
```

Pseudo code 3: Reconstruction of the 1-D array S with filters H and G assuming S[0..l/2-1] and S[l/2..l-1] are the high and low passed filtered subbands

3.2.2 3D-filtering

The 3-D filtering (shown in pseudo code 4) makes use of the 1-D filtering. The procedure `Filter1D` is called for each column in the 3-D array `S` in each of the three dimensions. The result (eight subbands) resides in the eight octants of `S`.

```
Filter3D (S, G, H) :=
    for t in TRange (S)
         for x in XRange (S)
              Filter1D (S[t,x,*], G, H)
         for y in YRange (S)
              Filter1D (S[t,*,y], G, H)
    for x in XRange (S)
         for y in YRange (Y)
              Filter1D (S[*,x,y], G, H)
```

Pseudo code 4: Filtering of the 3-D array S resulting in eight new subbands stored in the octants of S

The reverse `ReFilter3D` can be implemented very easily by simply substituting `Filter1D` with `ReFilter1D` in `Filter3D`.

3.2.3 Reducing Cache Misses

The handling of large 3-D arrays in memory leads to unexpected problems. To speed up memory accesses, each computer system contains components that store frequently used data in smaller but faster memory modules such as caches. Often there are several levels of caches and their structure is mostly a hash with the absolute memory address as key. Usually memory is divided into blocks called *pages*, which are used by dynamic memory allocation to specify which *physical page* belongs to which process. Also a mapping from a *logical page number* to the *physical page* is managed here. This information is kept in a so called *page table* held in memory. In applications with large memory demand the management of such page tables causes similar problems as memory accesses themselves. So there are similar modules for this purpose.

As can be seen in Figure 3.2, not all of the bits in a memory address are used to address the cache. Some high order bits are neglected, assuming that addresses of data accessed within short time do not differ only in high order bits. Unfortunately this is actually what happens within our algorithm. The first loop (`t` loop) is not affected by cache misses since the data is organized frame by frame in the memory. The problem arises when computing the

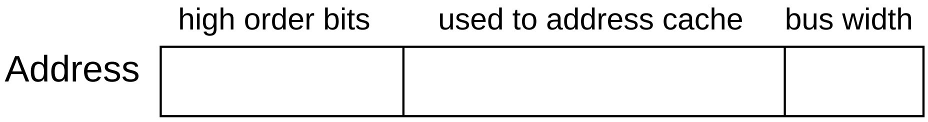


Figure 3.2: The use of a memory address as a hash value in indexing a cache

temporal transform (within the `x` loop) since the data involved is not close in terms of memory location, in contrary these values are separated by a data block of the size of one frame (this is why prefetching techniques do not work properly as well). Moreover, since we operate with power-of-two sized data, these in-between datablocks are as well of power-of-two size which means that the data values required for the temporal transform are loaded into the same cache lines each time when required within the transform stage (which is quite a few times) which produces a significant amount of cache misses in the transform.

A first solution attempt is to insert a short space between the frames to make the address jumps be of a size that is not a power of two. A more sophisticated approach is to completely rewrite the algorithm, as shown in pseudo code 5. It follows the idea to read the input data in a continuous way and to compute immediately each output or intermediate data that depends on the input data.

```
RevConv (Data, Filter, Coeff, Mod) :=
    l = length (Filter) - 1
    for j in 0..f/2-1
        Data[l-j] = Data[l-j] + Coeff * Filter[2*j+Mod]

Filter3D (S, G, H) :=
    set C[*,*] and D[*,*,*] to zero
    for t in 0..t_max+f-1
        for x in 0..x_max+f-1
            for y in 0..y_max/2-1
                b = S[t,x,2*y..2*y+f-1] * G
                RevConv (C[(x-f)/2+1..x/2,y], G, b, x mod 2)
                RevConv (D[(t-f)/2+1..t/2,(x-f)/2+1,y], G,
                         C[(x-f)/2+1,y], t mod 2)
    S = D
```

Pseudo code 5: Optimized Sequential 3-D Wavelet Decomposition

Here `b` is the value of a convolution of the video data with the filter `G` along the y-axis at `y`. Because `y` is the index of the innermost loop the subsequent computation of `b` can reuse the values loaded into cache before. To understand the procedure `RevConv` (reverse convolution), we take a look at the non-optimized algorithm. By setting $i' := 2i + j$, $j' := \frac{j}{2}$ and $Mod := j \bmod 2$, $T[i] = T[i] + S[2i + j]G[j]$ can be written as $T[i'/2 - j'] = T[i'/2 - j'] + S[i']G[2 * j' + Mod]$. This is essentially what `revconv` does: It computes all $T[i'/2 - j']$ for a fixed i' (`Coeff` $= S[i']$). `C` holds the results of convolutions along the x-axis. For given `x` and `y` `C[(x-f)/2+1,y]` is completely computed and can be used to compute `D` in the same way. `D` has to be copied into `S` afterwards to continue with the next decomposition step.

For the sake of better readability the algorithm is a bit oversimplified. Products with `H` are not shown at all. So each time `G` is referenced, the same action has to be applied to `H`. The results have to be kept in separate variables (and arrays) and used in succeeding products. Therefore, two variables `b` and four arrays `C` (with the size of a quarter of a frame) have to be used, producing eight 3-D arrays `D` corresponding to eight subbands. Additionally, setting `C` and `D` to zero can be done within the innermost loop in order to avoid further cache misses.

3.3 Pyramidal Decomposition

```
Decomp3D (S, MaxLevel) :=
    for l in 1..MaxLevel-1
        Filter (S)
        S = Child (S, 1)
```

Pseudo code 6: Pyramidal FWT decomposition

Pseudo code 6 shows the simple implementation of the pyramidal 3-D decomposition. On each level only the approximation subband (`Child (S, 1)`) is decomposed. The composition procedure has to perform `ReFilter` in the reverse order. This is solved in the generalized algorithm in section 3.4.

3.4 Wavelet Packets

Here we introduce a generalization of the wavelet transform, where the choice of the decomposition tree structure is made by an octuple of codes $scheme = (scheme_1, \ldots, scheme_8)$, where $scheme_k \in \{0, 1\}$ means that the subband `child (S, k)` of `S` has to be decomposed only, if $scheme_k = 1$.

That means that for instance $scheme = (1, 0, 0, 0, 0, 0, 0, 0)$ corresponds to pyramidal decomposition and $scheme = (1, 1, 1, 1, 1, 1, 1, 1)$ corresponds to a full wavelet packet decomposition. Pseudo code 7 implements this.

```
PacketDecomp (S, MaxLevel, Scheme) :=
    for T in SubTree (S, MaxLevel - 1)
        before: if Scheme[Child] = 0
                    CutSubTree
                else
                    Filter3D (T)
```

Pseudo code 7: Wavelet packet decomposition

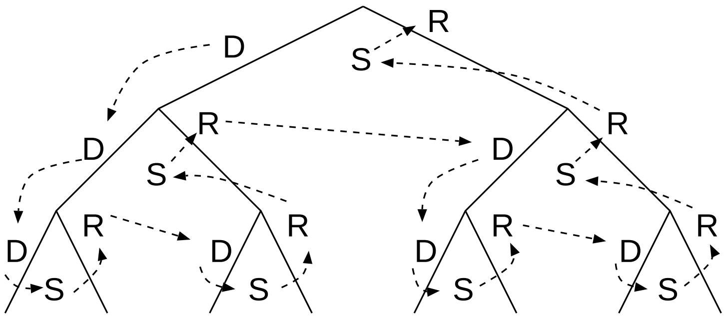
The reverse (synthesis) is performed in pseudo code 8. The only difference is that the Refiltering has to be executed after the subbands have been reconstructed, i.e. after the subtree of `S` has been processed.

```
PacketRecomp (S, MaxLevel, Scheme) :=
    for T in SubTree (S, MaxLevel - 1)
        before: if Scheme[Child] = 0
                    CutSubTree
        after:  if not Scheme[Child] = 0
                    ReFilter3D (T)
```

Pseudo code 8: Wavelet packet synthesis

3.5 Best Basis

A new problem in implementing the best basis algorithm is memory usage. Because the effective decomposition tree is determined after the data has been fully decomposed, there seem to be only two methods to output the subbands that correspond to the best basis: Either we could store all subbands that are generated during the decomposition process and select the proper ones after determining the best basis tree or we could start a second decomposition that stops where the decomposition tree doesn't branch. The first approach requires very much memory (to be more precise: `MaxLevel` times the original memory requirements) and is therefore not usable for video compression. The second possibility obviously increases execution time and causes further complications in parallelizing the code, as we will see in the next chapter. An improvement would be to perform the second run in a reverse way, i.e.



D = Backup subband and Decompose
S = Sum the cost of subbands
R = Restore subband if decomposition does not reduce the cost

Figure 3.3: Best basis decomposition without a second run and minimized memory demand

composition of the required subbands out of the transformed data. This should result in less work, because it is known that subbands on lower levels are more likely to have higher entropy. In addition there would be no need for a reload of data.

Here another strategy is presented. The main idea is to avoid the second run by backing up and restoring subbands and by traversing the decomposition tree in depth first order (see Fig. 3.3). Pseudo Code 9 implements this strategy. In this pseudo code another stack (`CostStack`) is used to store the values of the cost function. This is necessary, because storing all these values in a tree structure could easily lead to a memory demand equal to that of the video data if `MaxLevel` is high. This stack only needs to store all the cost values that have been generated during the tree walk and that have not yet been summed up into a parent subband. Therefore, the memory demand is only of order $O(MaxLevel)$.

In the `before`-part the subband is backed up, so if after processing it turns out that there is no improvement in the decomposition of this subband, then it can be restored in the `after`-part. Then the cost of the current subband is determined and pushed onto the cost stack, because later it has to be compared to the sum of the detail costs (of the child subbands).

In the `after`-part first the sum of the best costs of the child subbands is built, which have cumulated on the stack. Then if this sum is bigger than the cost of the current subband (fetched from the stack), the decomposition tree should not branch at the current subband. This means that the original subband (not decomposed) has to be restored. In the other case a new branch has to be inserted into the bittree `BT`. The smaller cost (best cost) has to be

```
BestBasisDecomp (S, MaxLevel, Scheme) returns DecompTree :=
    Empty (CostStack)
    BT = EmptyTree
    for T in SubTree (S, MaxLevel)
        before: put Cost (T) into CostStack
                if not Leaf
                    Backup (T)
                    Filter (T)
                    if Scheme[ChildNr] = 0
                        CutSubTree
        after:  if not Leaf
                    C = 0
                    for i in 1..8
                        pop D from CostStack
                        C = C + D
                    if Scheme[ChildNr] = 2
                        pop E from CostStack
                        if C < E
                             Restore (T)
                             push C onto CostStack
                        else push E onto CostStack
                             Branch (BT, T)
                    if Scheme[ChildNr] = 1
                        pop E from CostStack
                        push C onto CostStack
                        Branch (BT, T)
```

Pseudo code 9: Decomposition into best basis pushed onto the stack so that the sum on the next lower level subband can be calculated.

Chapter 4

Parallel Algorithms

As we have now seen what the wavelet transform is, which variations exist and how they can be implemented on single processor computers, we have to concentrate now on the parallelization of these algorithms. The parallelization is mostly based on splitting the 3-D array into several pieces and distributing these among the processor elements (PE). This approach is generally called "input driven parallelization".

Sometimes other possibilities offer simpler implementations and/or better performance. Of course the different implementations show different advantages/disadvantages at different computer architectures. The most obvious example for this is shared memory programming compared to message passing. The shared memory approach is **very** easy to implement, but can only be used in shared memory architectures and is somehow dangerous with respect to performance (as we will see).

This chapter will introduce several paradigms and implementations and compare their performance results with regard to this background.

All the experimental results are conducted using 8-tap QMF filters. The speedups are computed as

$$\mathrm{Speedup}_n = \frac{t_s}{t_n}$$

where $\mathrm{Speedup}_n$ is the speedup for the parallel algorithm using n node PE plus 1 host PE for the message passing implementation, t_s is the execution time of the sequential algorithm and t_n is the execution time of the parallel algorithm with n node PE.

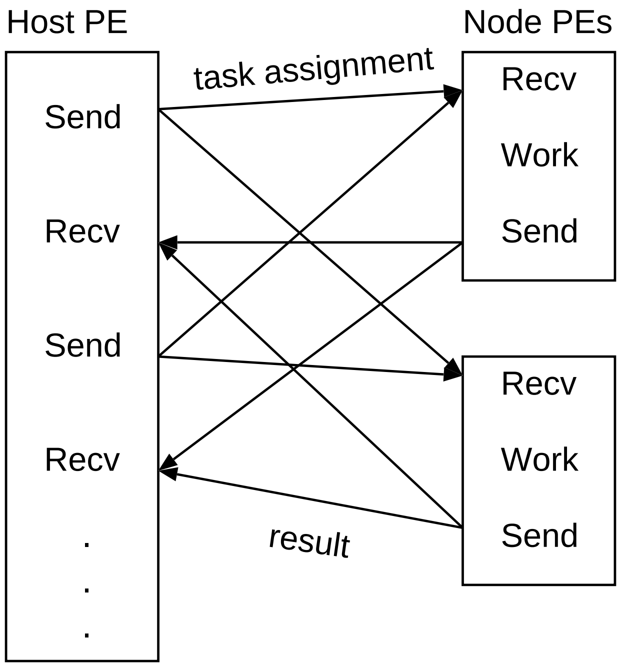


Figure 4.1: The host node programming paradigm

4.1 Message Passing

Message passing is a parallel programming paradigm which makes use of explicit communication. Several processes are running in parallel and each time they have to exchange data, they have to call proper communication routines. These routines are provided by message passing libraries such as PVM and MPI. In this work both libraries were used to get an impression on how much different implementations can affect performance on different computer architectures and what the advantages of different programming interfaces are.

4.1.1 Host-Node Programming

Applying the host-node programming paradigm, one PE is dedicated to distribute the work load among the other PE and to collect the node PEs results. Figure 4.1 visualizes this. In the best case, the work can be distributed in a way such that the node PE do not have to exchange information to proceed with their part of work. Of course this is not always the case – and is not in our application.

The host-node programming paradigm offers the advantage of simplicity and comprehensibility, because one process (host) keeps track of what is going on in the application and controls all other processes (node processes). Although this is not necessary for our purposes, this paradigm corresponds to the communication structure of our problem. The video data – initially read through a single communication port – is distributed among the node PE, where the transformation is done in parallel and the results are collected by the host PE, which outputs it.

4.1.2 Preliminaries

It is not a simple thing to write a pseudo code for a parallel program. So there have to be some formalisms to be defined that are commonly used in message passing algorithms. Here we want to conform to the SPMD (single program multiple data) programming paradigm. So the same algorithm applies to the host as well as the node processes. Whenever some PE should not do the same as the other, the code parts are labeled with `H:` or `N:`, where `H` means that only the host PE performs this part of code and `N` means that only the node processes perform this part of code. Also for each node process the value of `NodeNr` is defined in the range $0 \dots$`#Nodes-1`, where `#Nodes`$= \#PE - 1$. The communication between the PE is expressed with the procedures `Send X to n` and `Recv X from n` where `X` specifies the data to be transferred and `n` is $0 \dots \#PE - 2$ for the node process numbers and `H` for the host process.

As we frequently have to distribute subband data among the PE, there has to be a possibility to split a subband `S` into several pieces. The function `Split (S, k)` gives the k-th part of `#Nodes` possible, where the parts are non overlapping and all `#Nodes` parts entirely fill `S` (i.e. $\{$`Split (S,` k`)` $| k \in 0 \dots$`#Nodes-1`$\}$ build a disjoint partition of `S`). There are several possibilities for such a partitioning. This will be discussed in detail in section 4.1.5.

The procedures `distribute` and `collect` are commonly used here, so pseudo code 10 shows, how these work. Their aim is that the host PE splits data sets into several pieces and sends each of which to a separate PE (or just the reverse operation).

Additionally, we will need to extend the range of a subband to hold also the necessary border data for the parallel filtering: The function `Inflate (S,l,f)` for $S = (t_{min}, t_{max}, x_{min}, x_{max}, y_{min}, y_{max})$ will give the new range $(t_{min} - a_1, t_{max} + b_1, x_{min} - a_2, x_{max} + b_2, y_{min} - a_3, y_{max} + b_3)$ where $a_n + b_n = (2^l - 1)(f - 2)$, if we choose to split the data in the corresponding direction at least one time. (Otherwise no border data has to be exchanged.) `l` specifies the level of decomposition to which the border data should be supplied. `f` denotes the filter length.

```
Distribute (S) :=
    H: for k in 0..#Nodes-1
           Send Split (S, k) to k
    N: Recv Split (S, k) from H

Collect (S) :=
    H: for k in 0..#Nodes-1
           Recv Split (S, k) from k
    N: Send Split (S, k) to H
```

Pseudo code 10: Procedures to distribute and collect data

4.1.3 Filtering

The parallelization of the filtering is an important but not the only way of parallelization of the fast wavelet transform. The main idea is to assume that the subband is distributed so that `Split (S, k)` is located at node PE `k`. Then the 1-D filtering is split so that each node PE processes only that part of the data that belongs to that PE. Combining these together to a 3-D filtering process as we have seen it before produces the eight subbands of `S` distributed just as `S` itself, i.e. `Split(Child (S, c), k)` is located at PE `k`. Pseudo code 11 shows this.

```
FilterMP (S, G, H) :=
    N: T = Split (S, NodeNr)
    N: Filter3D (T, G, H)

ReFiltMP (S, G, H) :=
    N: T = Split (S, NodeNr)
    N: ReFilt3D (T, G, H)
```

Pseudo code 11: The parallel filtering of distributed data

Of course this requires some overlapping data present at each PE, because we have to use the extended-array-version of padding (see the last option of the function `Pad` in section 3.2.1). The next subsection explains, how this data can be gathered.

4.1.4 Border data

A classical problem in parallel wavelet and wavelet packet algorithms is the question of how to handle the exchange of the necessary border data. Recall

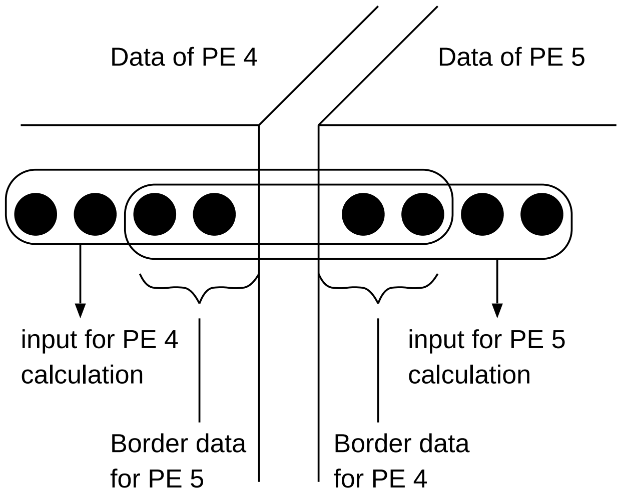


Figure 4.2: Border data

that the need for border data located on adjacent PE is caused by the nature of the QMF filtering process which involves several neighbouring data points in order to compute a single transform coefficient (see Figure 4.2). In the literature two approaches for the boundary problems have been discussed and compared [49, 43]. During the *data swapping* method each PE calculates only its own data and exchanges these results with the appropriate neighbour PE in order to get the necessary border data for the next decomposition level. Employing *redundant data calculation* each PE computes also necessary redundant border data in order to avoid additional communication to obtain this data. It was shown that the decision which method to apply is very important for certain wavelet algorithms (see e.g. [17] for the á trous algorithm and [27] for 3-D pyramidal fast wavelet decomposition).

Here we will combine these methods by performing `RedundancyLevel` steps with redundant data and n subsequent steps with data swapping. `RedundancyLevel=1` means that the host PE sends just that amount of redundant data so that the node PE do not have to exchange border data before the first decompostion step.

The implementation is shown in pseudo code 12. First we have to distribute overlapping data depending on the level of redundancy we choose. This is what `DistributeInflated` does. Like in `Distribute` the host PE sends the parts specified by `Split` to the corresponding node PE, but it also extends these parts by the amount needed as border data for `RedundancyLevel` filtering steps. After that parallel filtering is performed on the extended parts which produces the subbands for the next filtering step including border data

```
DecompMP (S, G, H, MaxLevel, RedundancyLevel) :=
    DistributeInflated (S, RedundancyLevel, length (G))
    for Level in 0..MaxLevel-1
        if Level >= RedundancyLevel
            ExchangeBorderData (S)
        T = Split (S, NodeNr)
        U = Inflate (T, RedundancyLevel - Level, length (G));
        Filter3D (U, G, H)
        for k in 2..8
             Collect (Child (S, k))
        S = Child (S, 1)
    Collect (S)

DistributeInflated (S, RedundancyLevel, FilterLength)
    H: for k in 0..#Nodes-1
           U = Split (S, k)
           V = Inflate (S, RedundancyLevel, FilterLength)
           Send V to k
    N: U = Split (S, NodeNr)
    N: V = Inflate (S, RedundancyLevel, FilterLength)
    N: Recv V from H

ExchangeBorderData (S) :=
    for p in 0..#Nodes-1
        for q in 0..#Nodes-1
            if not p = q
                U = Inflate (Split (S, q), 1, length (G))
                T = Intersect (Split (S, p), U)
                if T not empty
                    Send T from p to q
```

Pseudo code 12: Handling of redundant data and border data exchange for the pyramidal decomposition if `RedundancyLevel` is high enough. If there is no border data left from the previous step, this border data has to be gathered from the neighboured PE. `ExchangeBorderData` performs this by building the intersection between inflated subbands and other subbands. These intersections hold the data that have to be transferred as border data. As this pseudo code implements pyramidal decomposition, only the low pass subband is decomposed in subsequent decomposition steps, so the other seven subbands can be collected by the host PE. Of course, when `MaxLevel` is reached the remaining subband is also collected.

Experimental Results

Figure 4.3 shows, how the choice of border data handling affects the speedup on several architectures (see section 1.4). Data sizes in use are 512, 256 and 256 in t-, x- and y-direction respectively on the Cray T3E and the SGI PowerChallenge and 256 in each direction on the Siemens cluster.

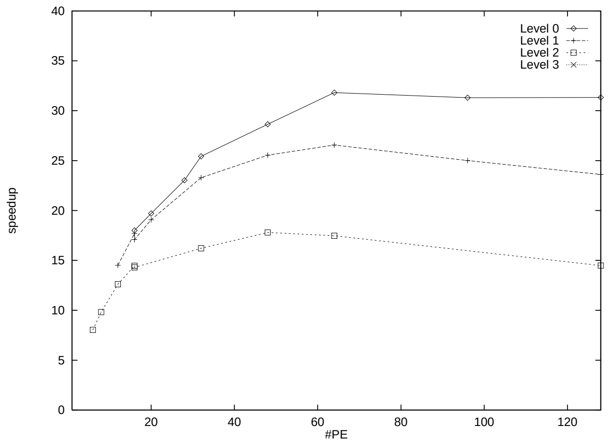
Except for the shared memory architecture a redundancy level of 0 seems to be the best choice. This means, reducing the work for initial data distribution to a minimum which seems to be the bottleneck. One node PE always has to wait until the distribution is complete. This limits the performance to the execution time of this last PE. Note that there is no difference between redundancy level 0 and 1 with respect to computational effort. In the latter case the border data for the first decomposition step is received from the last PE instead of the neighbouring node PE.

This effect is not present (or at least less visible) on the shared memory architecture (c). This can be explained by the fact that the "network structure" underlying the shared memory architecture (i.e. one memory to n nodes) strongly corresponds to the communication structure used in the initial data distribution step (i.e. host to n nodes).

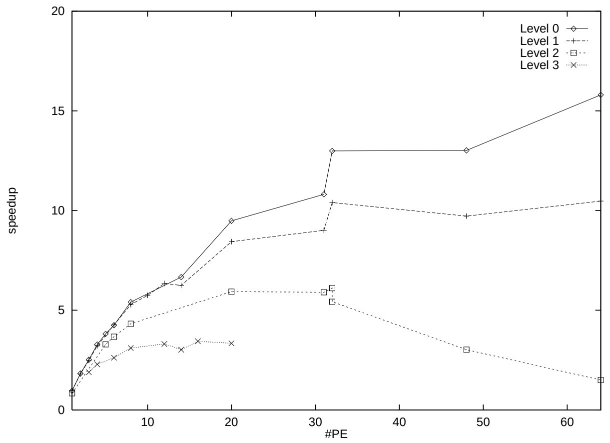
Higher redundancy levels increase the amount of data that has to be distributed in the beginning as well as the computational effort which leads to worse speedups although the exchange of border data is not necessary for the first few decomposition steps.

Another interesting subject is that the speedup increases significantly in Figure 4.3 (b) as $\#PE$ steps from 31 to 32. The reason for this phenomenon is a load balancing problem: A power-of-two-sized data sized data set has to be distributed among an uneven number of PE (31) so that each part is divisible by 2^{MaxLevel} which leads to the situation that some parts are bigger than others. Therefore, the corresponding PE have to process more data. The optimal values for $\#PE$ are thus powers of two.

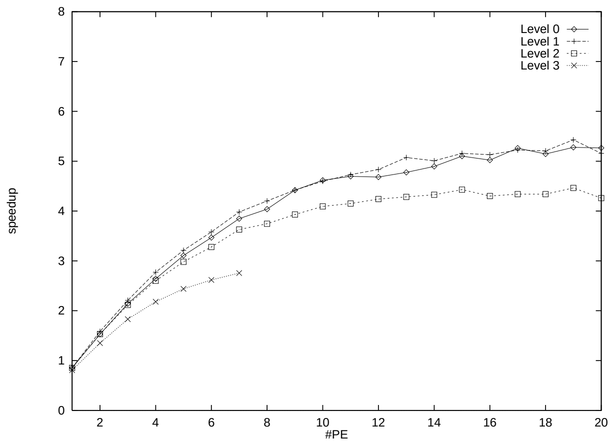
The maximum speedup on the Cray T3E is obviously reached at $\#PE =$



(a) Cray T3E



(b) Siemens Cluster



(c) SGI Powerchallenge

Figure 4.3: Speedups of pyramidal decomposition for different redundancy levels

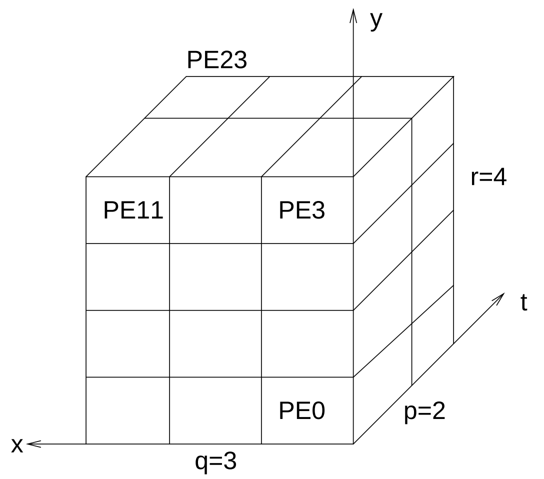


Figure 4.4: Example multidimensional split of video data

64. Above, the communication overhead gets too big. Also it has to be stated that shared memory is the least sensible architecture with respect to different program features, but this can be explained by the smaller network type in use.

4.1.5 Multidimensional Splitting

When computing the 3-D decomposition in parallel one has to decompose the data and distribute it among the PE somehow. In previous works parallelization has been done along time axis [27] or in the spatial domain [39]. In this work we use a technique where it can be chosen how often the video data is split into nearly equal parts in each dimension. In the following p, q and r denote the splitting factors for t, x and y axis respectively (see Figure 4.4). Of course $pqr \leq \#PE$. `Split (S, l)` splits the video data into parts D_{ijk} where $0 \leq i < p$, $0 \leq j < q$, $0 \leq k < r$ and each D_{ijk} is assigned to PE_l where $l = (iq + j)r + k$. No data is assigned to PE_l with $pqr \leq l < \#PE$.

Each D_{ijk} has an approximate size of $\frac{t}{p}\frac{x}{q}\frac{y}{r}$ if t, x, y denote the size of the video data. In a decomposition step overlapping data of a width of $b = f - 2$ (border data) has to be exchanged with neighbouring PE. Thus border data has a size of

$$\left(\frac{t}{p} + b\right)\left(\frac{x}{q} + b\right)\left(\frac{y}{r} + b\right) - \frac{t}{p}\frac{x}{q}\frac{y}{r} =: s(p, q, r).$$

By minimizing $s(p, q, r)$ with the constraint $pqr = \#PE$ using the Lagrange multiplier λ we get

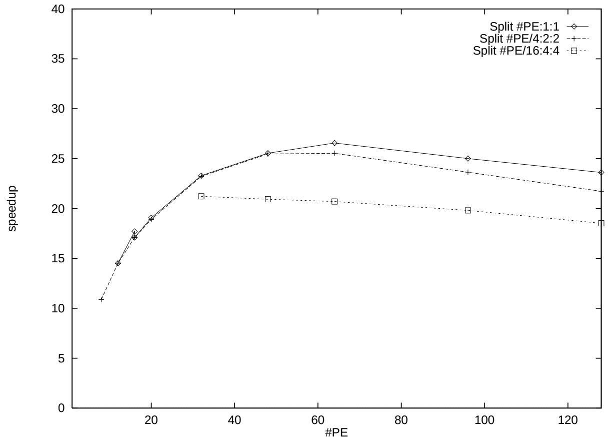
$$
\begin{aligned}
& & \tfrac{\partial}{\partial(p,q,r)}(s(p,q,r) + \lambda(pqr - \#PE)) = 0 \\
\Rightarrow & & \tfrac{xyz}{\#PE} - \tfrac{t}{p}\left(\tfrac{x}{q} + b\right)\left(\tfrac{y}{r} + b\right) + \lambda\#PE = 0 \\
& & \tfrac{xyz}{\#PE} - \left(\tfrac{t}{p} + b\right)\tfrac{x}{q}\left(\tfrac{y}{r} + b\right) + \lambda\#PE = 0 \\
& & \tfrac{xyz}{\#PE} - \left(\tfrac{t}{p} + b\right)\left(\tfrac{x}{q} + b\right)\tfrac{y}{r} + \lambda\#PE = 0 \\
\Rightarrow & \tfrac{t}{p}\left(\tfrac{x}{q} + b\right) = \left(\tfrac{t}{p} + b\right)\tfrac{x}{q} & \text{(by subtracting the first from the second)} \\
\Rightarrow & & \tfrac{p}{q} = \tfrac{t}{x}.
\end{aligned}
$$

$\frac{p}{r} = \frac{t}{y}$ and $\frac{q}{r} = \frac{x}{y}$ follow the same way. This means that the splitting factors should have the same ratios as the data sizes.

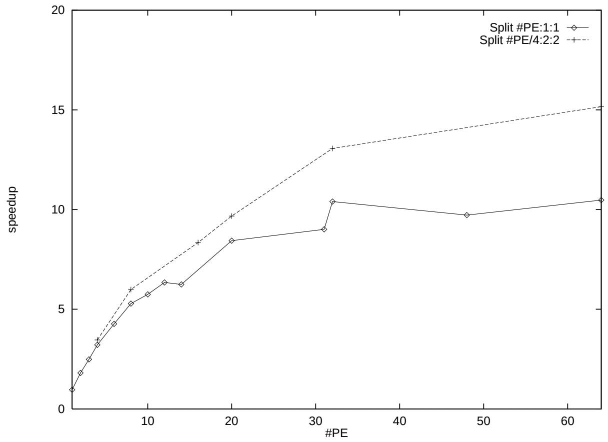
We have to note two facts: First, if either p, q or r is equal to 1 no border data has to be exchanged along the corresponding axis, thus reducing border data. Secondly p, q and r have to be integers. This leads to the problem that possibly $pqr < \#PE$. But $q = r = 1$ always can guarantee $pqr = \#PE$. These two facts are highly relevant for small #PE and can be overcome by writing a short procedure that tests all possible p, q, r and minimizes $s(p,q,r)$ where s considers the smaller border data for $p = 1$, $q = 1$ or $r = 1$ and adds a fraction of $\frac{t}{p}\frac{x}{q}\frac{y}{r}$ to get an estimation for the calculation time of one decomposition step depending on p, q and r. The latter portion has to be weighted according to the speed of processors and communication channels.

Experimental Results

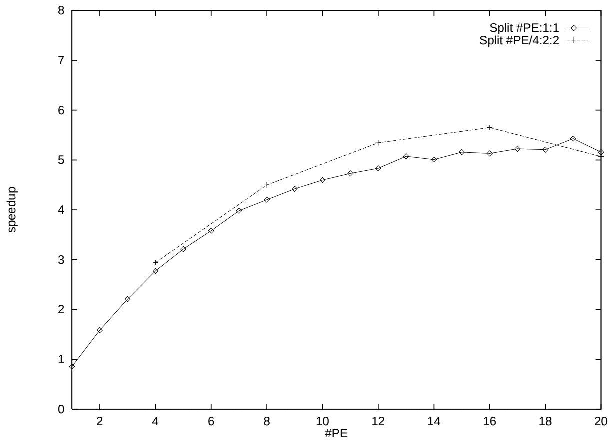
If these theoretical results correspond to the experimental results, one can see in Figure 4.5. Data sizes in use are 512, 256 and 256 in t-, x- and y-direction respectively on the Cray T3E and the SGI Power Challenge and 256 in each direction on the Siemens cluster. Here the notion $p : q : r$ denotes the splitting factors from above (for instance `PE/4:2:2` means $p = \frac{\#PE}{4}, q = 2, r = 2$). Except for the Cray T3E, significantly better results can be observed by splitting along several dimensions. The reason for the fact that this cannot be achieved at the Cray T3E might be the very good performance of the network used for inter-processor communications in the Cray T3E. Here obviously the number of messages sent and the number of communication channels involved has more impact on the communication times as the amount of data that is transferred. The fact that this effect is most significant on the cluster architecture suggests the interpretation that here the bandwidth of the network



(a) Cray T3E



(b) Siemens Cluster



(c) SGI Powerchallenge

Figure 4.5: Speedups of pyramidal decomposition for different types of multidimensional splitting

is a more characteristic value for the network performance than the latency which is the opposite of what one would expect, as the network is less integrated into the system architecture. This again shows that workstation clusters are able to compete against classical multicomputers.

4.1.6 Partitioning Start Data

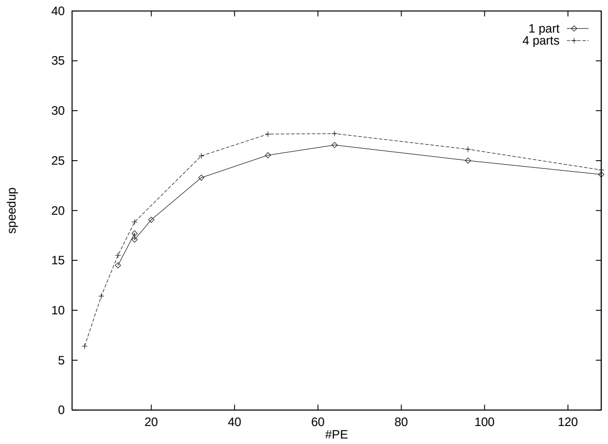
Because the last node PE has to wait very long in the start-up phase for its share of the video data, it consequently starts calculation later. Therefore, the processes are executed very asynchronously. Especially when splitting the data in several dimensions as supposed above the exchange of border data can lead to extensive communication that resynchronizes the processes, whereby time is lost. As a solution data can be sent in several parts. Pseudo code 13 shows a variant of the `DistributeInflated` procedure which implements this idea. `Part (S, p, n)` for $S = (t_{min}, t_{max}, x_{min}, x_{max}, y_{min}, y_{max})$ produces $(t_{min} + (t_{max} - t_{min})p/n, t_{min} + (t_{max} - t_{min})(p+1)/n, x_{min}, x_{max}, y_{min}, y_{max})$ as the p-th of n parts of `S`. After receipt of a part the PE can perform the according part of the first decomposition step and then wait for the next part to be received. The procedure `FilterPart` works the same way as `Filter3D` does, but it takes care that the results are not overwritten by subsequent parts that are received.

```
DistributeInflatedP (S, RedundancyLevel, FilterLength, Parts) :=
    H: for p in 0..Parts-1
           for k in 0..#Nodes-1
               U = Split (S, k)
               V = Inflate (S, RedundancyLevel, FilterLength)
               W = Part (V, p, Parts)
               Send W to k
    N: U = Split (S, NodeNr)
    N: V = Inflate (S, RedundancyLevel, FilterLength)
    N: W = Part (V, p, Parts)
    N: Recv W from H
    N: FilterPart (W)
```

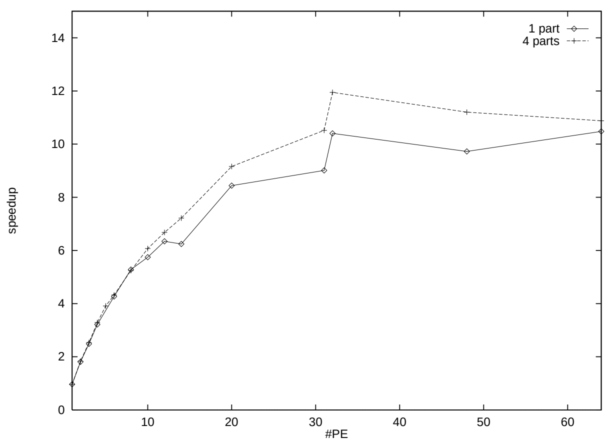
Pseudo code 13: Wavelet packet decomposition

Experimental Results

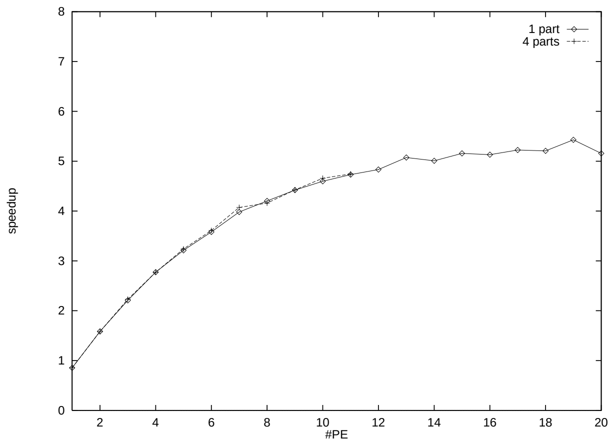
It can be seen in Figure 4.6 that this approach for optimization is able to improve performance. Data sizes in use are 512, 256 and 256 in t-, x- and



(a) Cray T3E

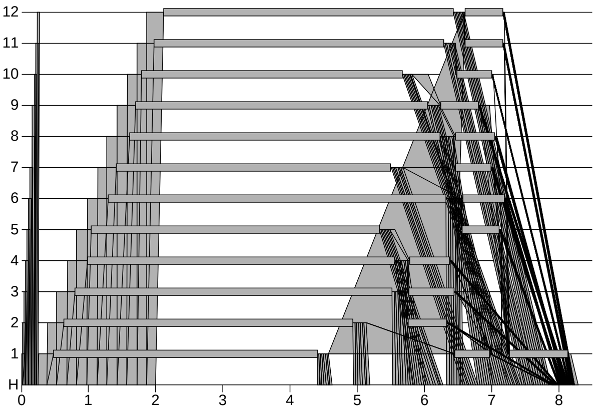


(b) Siemens Cluster

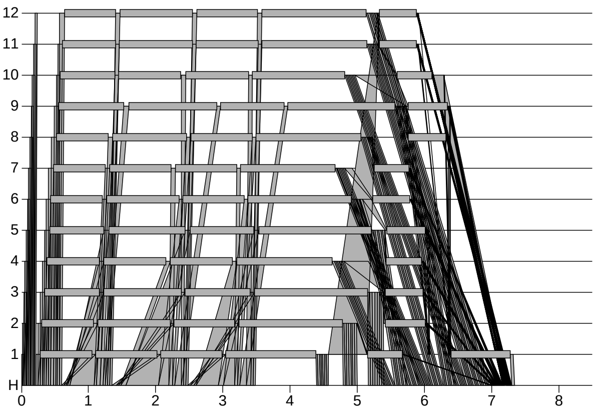


(c) SGI Powerchallenge

Figure 4.6: Speedups of pyramidal decomposition with and without partitioning of start data



(a) without partitioning of start data



(b) with partitioning of start data

Figure 4.7: Execution schemes for pyramidal decomposition on SGI Powerchallenge

y-direction respectively on the Cray T3E and the SGI Powerchallenge and 256 in each direction on the Siemens cluster.

Speedups can be increased by a certain percentage by synchronizing the processes in the specified way. Figure 4.7 illustrates two runs with identical parameter sets with and without partitioning of start data. The lowest horizontal line symbolizes the progression of the host PE in time (denoted H), the other represent the node PE. The horizontal boxes represent calculation phases. Crossed lines and gray parts symbolize the transmission of data from one PE to the other, where the gray parts result from the time that is consumed between the beginning and end of a send or a receive. Time is measured in seconds. Additionally, we observe the significantly different communication structure of the execution (i.e. execution time is reduced although the number of exchanged messages is increased).

As the processes are synchronized enough by splitting the start data into four parts, no further optimization can be achieved.

4.1.7 Wavelet packets

When we move on to wavelet packet decomposition, the problem arises that at higher decomposition levels the size of subbands decreases to ridiculous values and their number increases exponentially. As the width of the border data remains constant, the exchange of border data consumes more and more time, because its size increases and is transmitted within an increasing number of messages. At level l we have 8^l subbands of size $\frac{txy}{8^l}$. For one dimensional splitting the border data size is given by $\frac{xy}{4^l}b$ for one PE and one subband. So the overall border data size is given by $\frac{xyb}{4^l}\#PE8^l = xyb\#PE2^l$. This means that border data size increases exponentially with the decomposition level. But there is a possibility to redistribute the data at a certain level (redistribution level, see also [47, 16]).

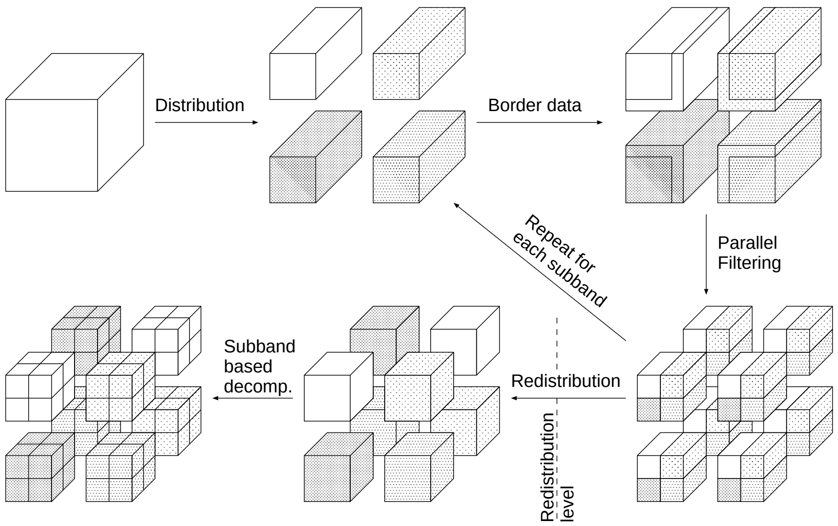


Figure 4.8: Full wavelet packet decomposition – different colors symbolize different PE

Figure 4.8 illustrates what happens from a data distribution point of view. The arrow labeled "Redistribution" shows the most important change from previous schemes. After a certain number of repetitions of the distributed decomposition steps (see arrow labeled "Repeat for each subband") data is redistributed (unicolored boxes) so that no further communication between node PE is necessary. The single subband can then be decomposed using the sequential algorithm ("Subband based decomposition").

Pseudo code 14 shows the implementation. For reasons of simplicity, the handling of border data is not shown. Here the data is decomposed in parallel up to the redistribution level. This means that the data are distributed as we have seen before and decomposed in parallel (including the exchange of border data, if necessary) until the subbands get so small, that the split into a part for each PE and the corresponding exchange of border data cause too much work. So there has some redistribution to be done when there are enough subbands generated to be distributed among the PE. Note that a single subband can be processed by a single PE and there has to be no communication between PE, if other PE process distinct subbands.

This redistribution is performed in a kind of group processing. A group contains as much subbands as there are node PE. Each time a subband at the redistribution level is found to be decomposed further, it is put into a group. When the group is full, there exists a subband for each PE, so load balancing is OK. At this point all the subbands in the group have to be decomposed by a distinct PE using the sequential algorithm. This is, what the procedure `ProcessGroup` does. First each subband has to be redistributed so that it resides on a single subband. Then each PE processes a subband using the sequential algorithm and sends the transformed data back to the master PE. The statement `for (T, k) in Group` denotes a loop with an iteration for each subband `T` of the group which are numbered by `k` from 0 to `#Nodes-1`.

Note that this code can be used as pyramidal decomposition too since the parallel algorithm of the wavelet packet decomposition uses the generalization with the `scheme`-parameter (see pseudo code 14).

Experimental Results

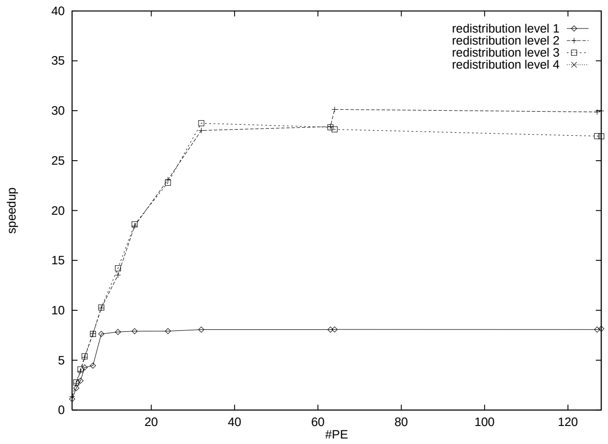
Figure 4.9 shows the speedups for different redistribution levels. Data sizes in use are 256 in each direction on the Cray T3E and the SGI Powerchallenge and 128 in each direction on the Siemens cluster.

One can see clearly that a too low redistribution level limits the number of subbands that are redistributed, leaving some PE without work (load balancing problem). Of course for level one only eight subbands are redistributed, so only eight PE are working after redistribution. Therefore, the speedup cannot exceed the value of eight for the subband based phase (after redistribution) which is the dominant phase. On the other hand, a too high redistribution level increases the communication complexity of the redistribution. This shows that redistribution based on subbands is necessary.

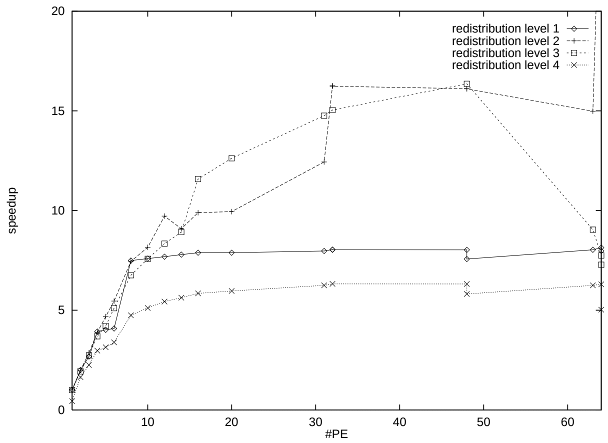
Within Figure 4.10 one can see four different runs with different redis-

```
PacketDecompMP (S, G, H, Scheme, MaxLevel, RedistLevel) :=
    Distribute (S)
    for T in Subtree (S, RedistLevel - 1)
        if Scheme[ChildNr] == 0
            CutSubtree
            Collect (T)
        else
            if not Leaf
                FilterMP (T, G, H)
    for T in Subtree (S, RedistLevel - 1)
        if Scheme[ChildNr] == 0
            CutSubtree
        if Leaf
           Put T into Group
           if Group is full
               Process Group
    Process Group

ProcessGroup :=
    N: for (T, k) in Group
           Redistribute (T, U, k)
    N: Decomp3D (U, G, H, MaxLevel - RedistLevel)
    N: Send U to H
    H: for (T, k) in Group
           Recv T from k
    Empty Group

Redistribute (T, U, DestNode)
    N: Send Split (T, NodeNr) to DestNode
    N: if NodeNr = DestNode
           for k in 0..#Nodes
               Recv Split (U, k) from k
```

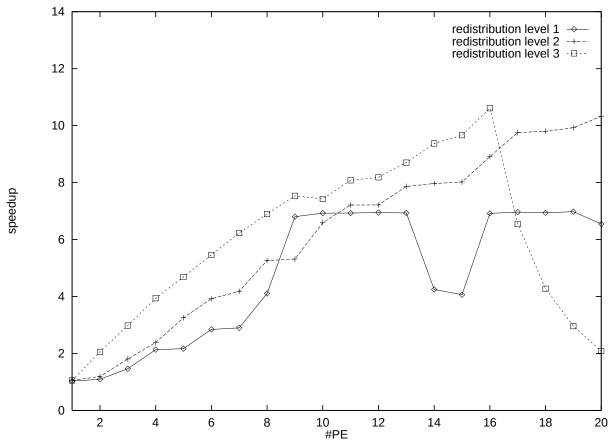
Pseudo code 14: Parallel version of packet decomposition including subband based redistribution



(a) Cray T3E

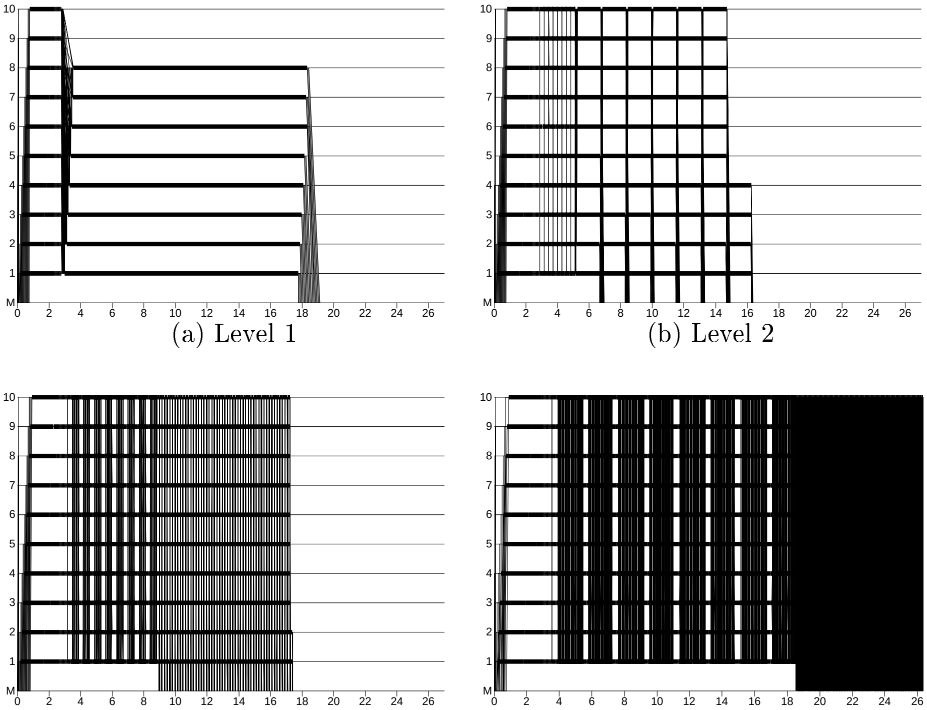


(b) Siemens Cluster



(c) SGI Powerchallenge

Figure 4.9: Speedups for full wavelet packet decomposition for different redistribution levels



(a) Level 1

(b) Level 2

(c) Level 3

(d) Level 4

Figure 4.10: Effect of subband based redistribution

tribution levels. This should illustrate, how load balancing problems and communication complexity affect the overall execution time. For redistribution level one, one can see that some PE are idle. For redistribution level four the disadvantages of excessive message passing are obvious.

4.1.8 Best Basis

The last step is the development of an algorithm that performs the decomposition into the best basis in parallel. Because data has to be redistributed at the redistribution level, the concept of the sequential algorithm cannot be completely adapted to the parallel algorithm. So we choose a mixed approach with three phases:

Phase 1 As visualized in pseudo code 15 in a first phase data are classically distributed (e.g. in equally sized blocks) and completely decomposed up to the redistribution level, using the procedure `PacketDecompMPC`. This procedure acts exactly like `PacketDecompMP`, but it additionally has to calculate the cost of each subband it processes, before it is de-

```
BestBasisDecompMP (S, MaxLevel, Scheme, RedistLevel)
                   returns DecompTree :=
    // Phase 1
    PacketDecompMPC (S, G, H, Scheme, RedistLevel,
                     RedistLevel, Costs)
    // Phase 2
    BT = EmptyTree
    for T in Subtree (S, RedistLevel - 1)
        before: if Scheme[ChildNr] == 0
                    CutSubtree
                else
                    if Leaf
                        Put T into Group
                        if Group is full
                            ProcessGroup
    ProcessGroup
    // Phase 3
    for T in Subtree (S, RedistLevel - 1)
        before: if Scheme[ChildNr] = 0
                    CutSubtree
        after:  if Scheme[ChildNr] = 2
                    CostSum = ZeroCost
                    for k in 1..8
                         CostSum = CostSum + Costs[Child(t)]
                    if Costs[T] < CostSum
                        CutOff (BT, T)
                        ReFiltMP (T)
                    else
                        Costs[T] = CostSum
                        Branch (BT, T)
                else
                    Branch (BT, T)
```

Pseudo code 15: Decomposition into best basis

```
ProcessGroup :=
    N: for (T, k) in Group
           Redistribute (T, U, k)
    N: BT = BestBasisDecomp (U, G, H, MaxLevel - RedistLevel)
    N: for (T, k) in Group
           if k = NodeNr
               for n in 0..#Nodes-1
                   Send Cost(T) to n
           Recv Costs[T] from k
    N: Send U to H
    N: Send BT to H
    H: for (T, k) in Group
           Recv T from k
           Recv SubTree (BT, T) from k
    Empty Group
```

Pseudo code 16: A subroutine of BestBasisDecompMP

composed, and to put this cost into the structure `Costs` (last argument) which is an array (indexed by subbands) located at each PE (not shown in the pseudo code). This is done by a local cost evaluation (`Cost (Split (S, NodeNr))`), followed by a *reduce operation* for summation, i.e. a parallelized summation of values that are distributed among the PE, resulting in the sum located at each PE. Because `MaxLevel` is set to `RedistLevel`, no redistribution of subbands is performed in this phase.

Phase 2 Subsequently, data is redistributed and decomposed using the sequential algorithm `BestBasisDecomp` (including the best basis selection for these subbands). Again this has to be done in groups, to distribute the work correctly among the node PE (see `ProcessGroup` in pseudo code 16. The resulting costs of the subbands (best cost according to best basis) have to be inserted into the `Costs` array. Additionally the bit trees returned by `BestBasisDecomp` which describe the tree structure of the best basis for the subbands at the redistribution level, have to be collected by the host PE and put together into the main tree at the host PE.

Phase 3 Now the final subband structure is generated. We have to cope with the possibility that a subband at a level below the redistribution level is required, so this data have to be regenerated somehow, as it was not backed up for memory reasons. We choose the approach to recompose this data from higher level subbands, because subbands are more likely to be decomposed at these levels. The `Costs` array serves as basis to the calulation of the root part of the decomposition tree (from the root up to the redistribution level). If the tree does not branch for subband `T`, it has to be recomposed in parallel using the `ReFiltMP` procedure.

Note that the exchange of border data is again ignored here for the sake of understandability.

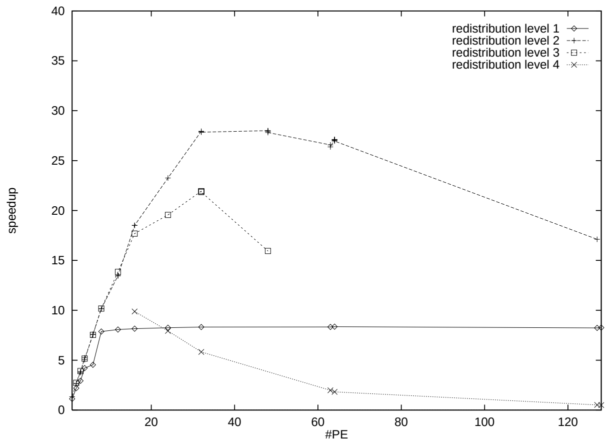
Experimental Results

Figure 4.11 shows the speedups for the best basis decomposition. Data sizes in use are 256 in each direction on the Cray T3E and the SGI Powerchallenge and 128 in each direction on the Siemens cluster.

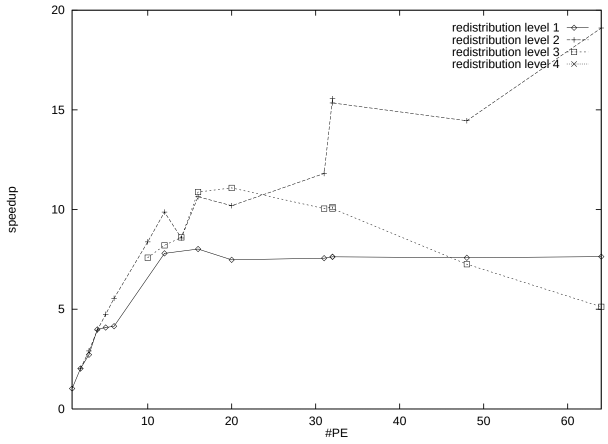
The results are very similar to that with full wavelet packet decomposition. The reason is that the computational demand for each subband is increased in the same way over all decomposition levels, and the only additional communications are the exchange of subband statistics and subtree information which is negligible compared to the exchange of subband data itself.

Effect of Border Data Handling on Best Basis Decomposition

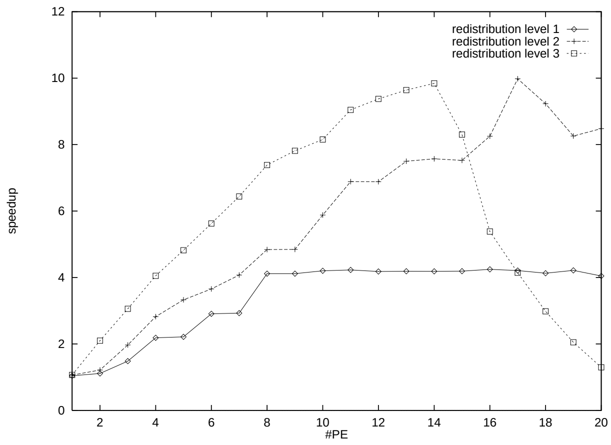
As we have seen, how much the choice of border data handling influences the performance in the pyramidal decomposition case, the question arises, if this behaviour is repeated for best basis decomposition. Figure 4.12 shows that this influence is much smaller than in the pyramidal case. The reason is that in the pyramidal case the entire processing time is dominated by the first decomposition level (where the choice of border handling strategy is crucial) whereas in the wavelet packet case all decomposition levels require roughly the same computational amount. The computations above the redistribution level are not affected by any border problem at all which explains the different behaviour. Only too high redundancy levels can enlarge the execution time of the first few decomposition levels that much so that the overall speedups drops visibly. For the same reason the use of different techniques for multi-dimensional data splitting shows little influence on the speedup behaviour of the algorithm.



(a) Cray T3E

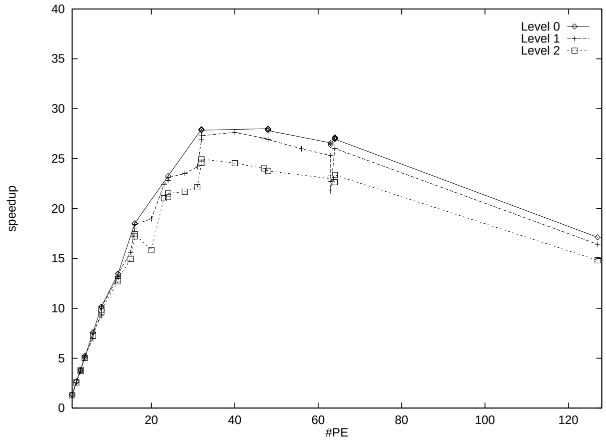


(b) Siemens Cluster

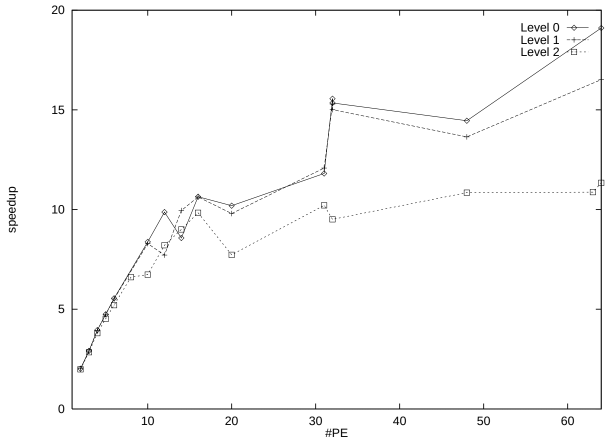


(c) SGI Powerchallenge

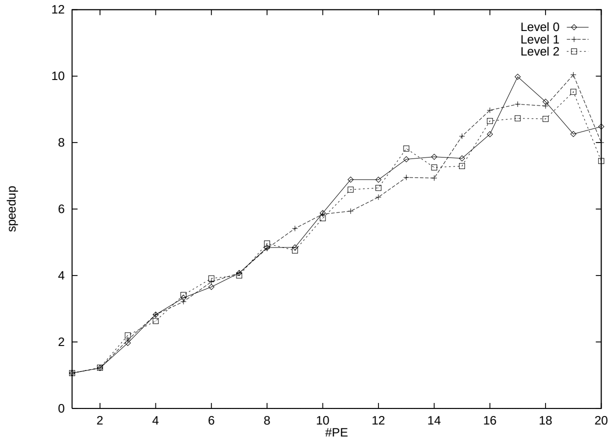
Figure 4.11: Speedups of best basis decomposition for different redistribution levels



(a) Cray T3E



(b) Siemens Cluster



(c) SGI Powerchallenge

Figure 4.12: Speedups of best basis decomposition for different levels of redundancy

4.1.9 The best results

Of course the most important question is which parameters are the ones that give us the optimal performance for the diverse tasks on the different hardware architectures.

Figure 4.13 shows the best results for pyramidal decomposition. The used parameters are as follows:

Cray T3E Redundancy Level 0, one dimensional splitting, no partioning of start data.

Siemens Cluster Redundancy Level 0, multidimensional splitting, partioning of start data.

SGI Powerchallenge Redundancy Level 1, one dimensional splitting, partioning of start data, PVM instead of MPI.

This shows that on different architectures different parameter sets produce the best results. On the Cray T3E the simplest algorithm performs best, whereas on the cluster architecture each modification is able to improve performance.

Figure 4.14 shows the best results for best basis decomposition. Here the used parameters are:

Cray T3E Redistribution level 2, redundancy level 0, one dimensional splitting, no partioning of start data.

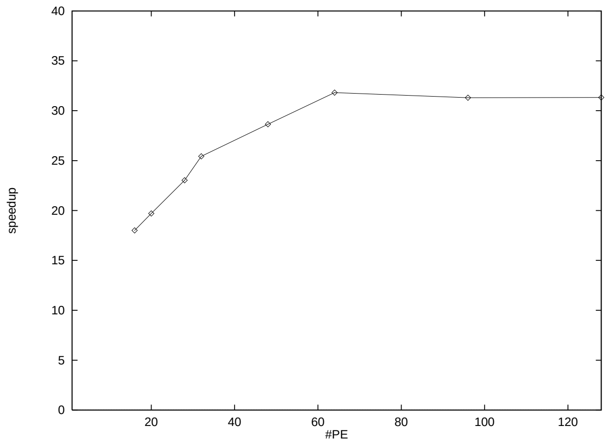
Siemens Cluster Redistribution level 2, redundancy level 0, multidimensional splitting, no partioning of start data.

SGI Powerchallenge Redistribution level 2, redundancy level 2, one dimensional splitting, partioning of start data, PVM instead of MPI.

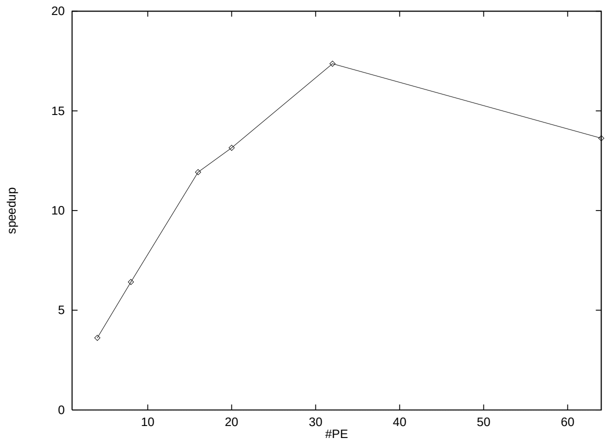
The parameters are about the same as with pyramidal decomposition which is no surprise. But the most important parameter here is the redistribution level. It turns out to be of level 2 on all machines. Of course this is because level 1 would be too low (load balancing) and level 3 would be too complex (8^3 subbands to redistribute).

4.1.10 Scalability

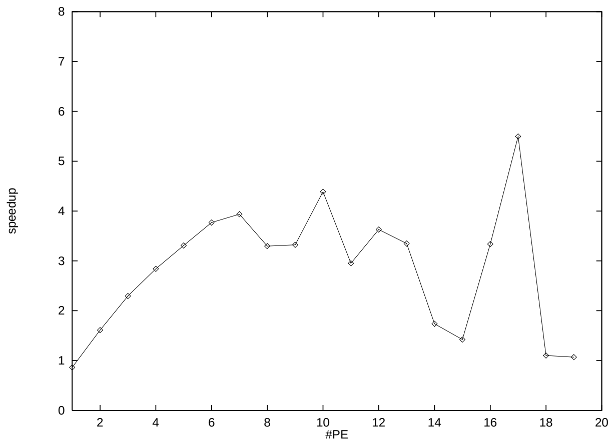
Having now sufficient information about the scalability of our algorithms applied to a fixed data size, we finally investigate the effect of varying the size of the video data in the spatial (just imagine the difference between QCIF



(a) Cray T3E

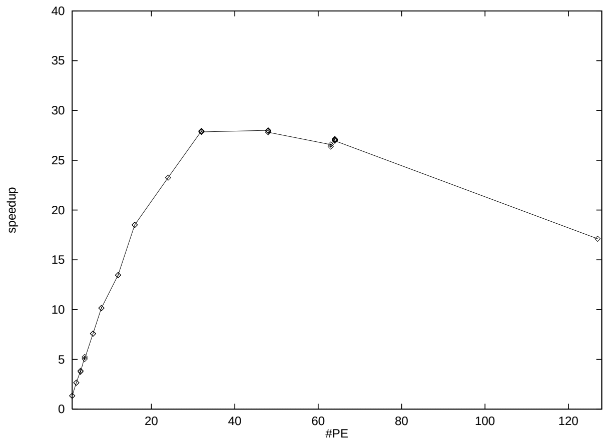


(b) Siemens Cluster

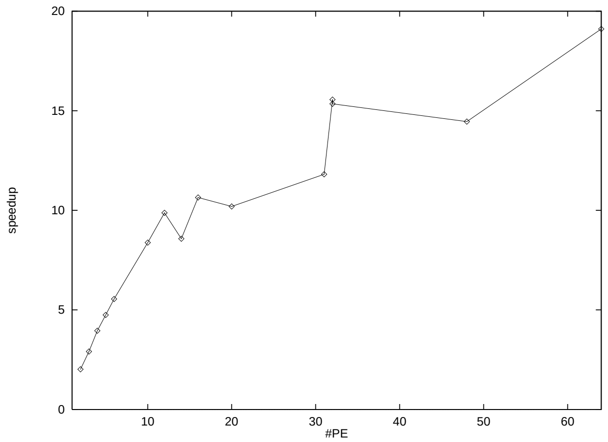


(c) SGI Powerchallenge

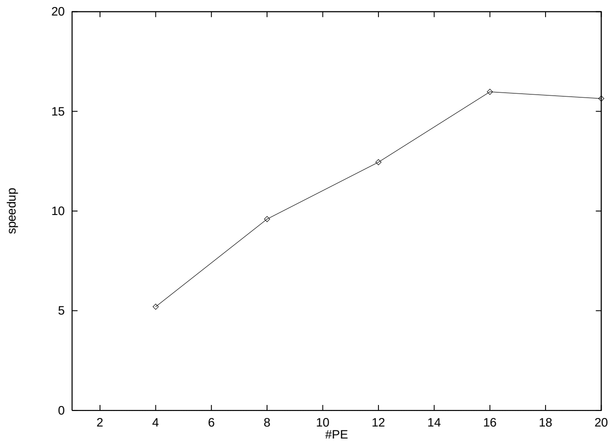
Figure 4.13: Best speedups for pyramidal decomposition



(a) Cray T3E



(b) Siemens Cluster



(c) SGI Powerchallenge

Figure 4.14: Best speedups for best basis decomposition

and SHDTV) as well as in the temporal domain (it might be desirable to use (temporal) smaller blocks in order to keep the coding delay to a minimum in a real-time application).

Figure 4.15 shows execution times of the pyramidal decomposition. Data sizes are kept in cubic shape if possible. The straight line labeled "ideal scalability" is intended to give a comparison to a fictitious program that requires exactly an n-fold execution time to process n-fold data sizes. If the curve for our algorithm is steeper, it means that it performs worse for bigger data size. If the curve is flat, the algorithm performs worse for smaller data size.

One can see that for big enough data sizes the curves are nearly parallel to the ideal scalability. The flat parts for higher #PE and small data sizes are understandable, because communication as well as computation times get very small, so message latency and other communication overhead gets dominant. On the Siemens Cluster the execution times increase rapidly at data size of 128 MB. Obviously this is due to memory problems (although each node owns 512 MB of memory). The author could not find the actual source of this problem.

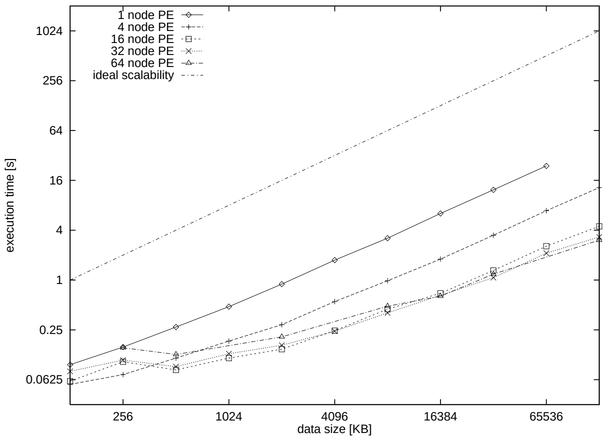
Figure 4.16 shows the equivalent plots for the best basis decomposition. Of course the execution times are higher than in the pyramidal case, but the overall behaviour is the same. Only the effects at small data sizes are more significant, because of more complex communication structures.

4.2 Shared Memory Programming

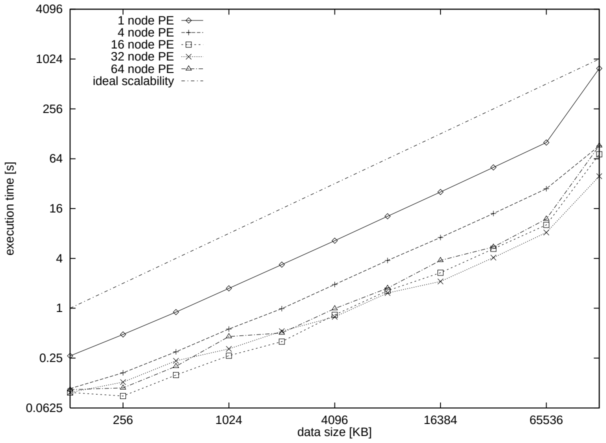
Program development on a shared memory architecture is easily achieved by transforming a sequential algorithm into a parallel one by simply identifying areas of code which are suitable to be run in parallel i.e. in which few dependencies exist among iterations and different iterations access different data. Subsequently, local and shared variables need to be declared (labeled by `local:` and `shared:` here) and parallel compiler directives are inserted (here only `parallel for` is used instead of `for` to distribute the loop iterations uniformly among the PE). In contrast to distributed memory programming there is no need to handle overlapping data regions explicitly.

Communication is handled by hardware. There are two approaches:

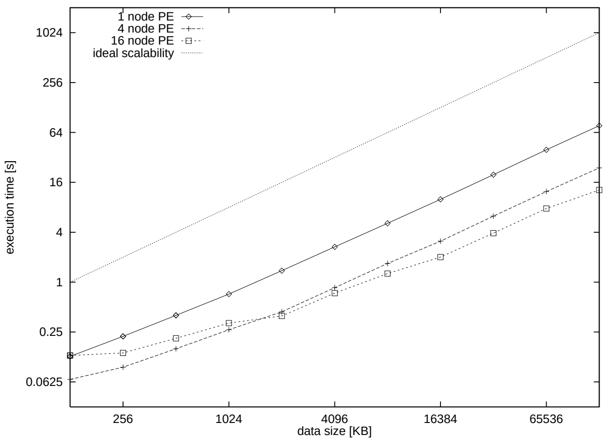
Real shared memory uses a single memory module to which all the PE are connected by a sort of bus. This bus of course turns out to be the bottle neck in most cases. The PE contain caches to reduce the bus activities which can lead to data inconsistency. To avoid this, *write through* caches or *snoopy protocols* can be used.



(a) Cray T3E

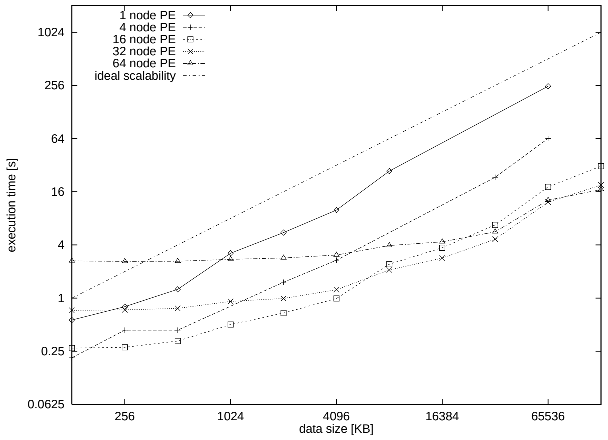


(b) Siemens Cluster

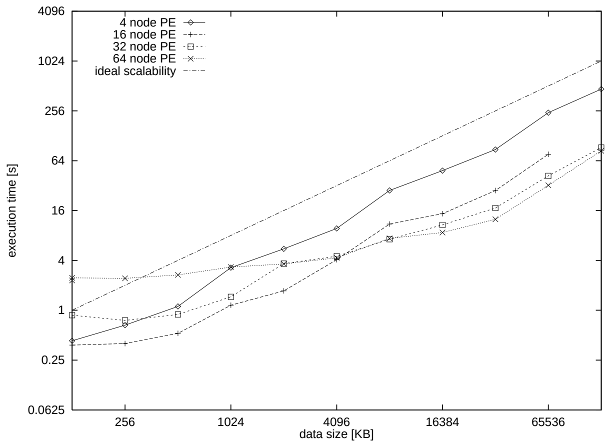


(c) SGI Powerchallenge

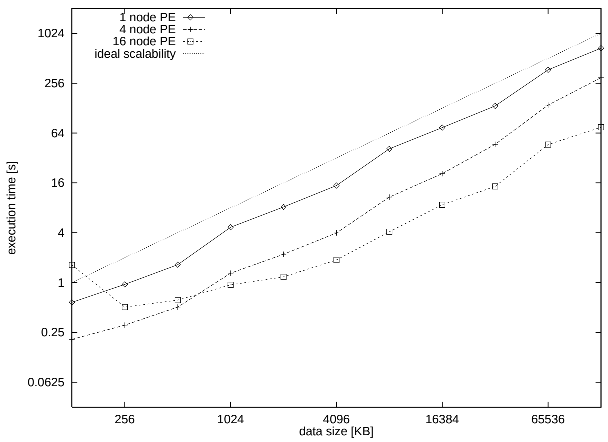
Figure 4.15: Scalability of pyramidal decomposition



(a) Cray T3E



(b) Siemens Cluster



(c) SGI Powerchallenge

Figure 4.16: Scalability of best basis decomposition

Distributed shared memory uses distributed memory, i.e. each PE contains its own memory modules, but each memory block is addressed by a logical global memory address. Whenever a memory element is accessed that does not reside in local memory, the PE have to communicate to transfer the memory element.

4.2.1 A Brute Force Method

The first approach does not modify the sequential code. The best candidates for a parallelizable loop are the outermost loops in the filtering procedure. Pseudo code 17 shows this simple implementation. The inner loops are performed entirely on single PE. There is no data exchange between the PE during the parallel loops, because consecutive iterations access separate data (different frames in the `t`-loop case). A lot of data exchange is performed *between* the two loops, because they are a parallelization along different dimensions which corresponds to a data redistribution. The only shared object is the subband, all other variables are local.

```
FilterShM (S, G, H) :=
    local: t,x,y
    shared: S
    parallel for t in TRange (S)
         for x in XRange (S)
              Filter1D (S[t,x,*], G, H)
         for y in YRange (S)
              Filter1D (S[t,*,y], G, H)
    parallel for x in XRange (S)
         for y in YRange (Y)
              Filter1D (S[*,x,y], G, H)
```

Pseudo code 17: Filtering – parallelized with shared memory programming

The same can be done with the optimized version. Within pseudo code 18 one can see that only one loop has to be parallelized. There are some data dependencies between the loop iterations, but only additions to the shared array `D` (in `RevConv`) which can be resolved by the hardware.

4.2.2 Redistribution

A more sophisticated approach is to emulate the data redistribution, presented in section 4.1.7.

```
FilterShM (S, G, H) :=
    local: C, b, x, y
    shared: S, D
    Distribute: D,1
    set C[*,*] and D[*,*,*] to zero
    parallel for t in 0..t_max+f-1
        for x in 0..x_max+f-1
            for y in 0..y_max/2-1
                b = S[t,x,2*y..2*y+f-1] * G
                RevConv (C[(x-f)/2+1..x/2,y], G, b, x mod 2)
                RevConv (D[(t-f)/2+1..t/2,(x-f)/2+1,y], G,
                             C[(x-f)/2+1,y], t mod 2)
    S = D
```

Pseudo code 18: Optimized filtering – parallelized with shared memory programming

Experimental Results

Figure 4.17 shows the performance of the best basis decomposition when applying the shared memory programming model. The redistribution level means the same as in the message passing case, as the message passing approach is simulated here. Results were conducted on the SGI Powerchallenge. Data sizes in use are 256 in each direction.

One can see that the brute force method hardly produces speedups above 1 and reaches its maximum at $\#PE = 4$. The simulation of the message passing approach can improve the speedups, but they do not exceed a value of 4. The maximum is reached at $\#PE = 9$. An interesting fact is the uniform shape of the speedup curves, with maxima at $\#PE = 9, 14$ and 17. The author cannot imagine a reason for this behaviour.

4.3 Comparisons

4.3.1 Shared memory programming vs message passing

The obvious advantage of shared memory programming is the easiness of implementation. But there has a price to be paid. In this case this means worse performance. Figure 4.18 shows a comparison of speedups for the pyramidal case and the best basis decomposition. The results were conducted

```
BestBasisDecompShm (S, MaxLevel, Scheme, RedistLevel)
 returns DecompTree :=
    // Phase 1
    PacketDecompC (S, RedistLevel, Scheme, Costs)
    // Phase 2
    for T in Subtree (S, RedistLevel - 1)
        before: if Scheme[ChildNr] == 0
                    CutSubtree
                else
                    if Leaf
                        Put T into Group
                        if Group is full
                            ProcessGroup
    ProcessGroup
    // Phase 3
    for T in Subtree (S, RedistLevel - 1)
        before: if Scheme[ChildNr] = 0
                    CutSubtree
        after:  if Scheme[ChildNr] = 2
                    CostSum = ZeroCost
                    for k in 1..8
                         CostSum = CostSum + Costs[Child(t)]
                    if Costs[T] < CostSum
                        CutOff (BT, T)
                        ReFiltShm (T)
                    else
                        Costs[T] = CostSum
                        Branch (BT, T)
                else
                    Branch (BT, T)
    return T

ProcessGroup :=
    parallel for (T, k) in Group
        SubTree (BT, T) = BestBasisDecomp (T, G, H,
                            MaxLevel - RedistLevel, Scheme)
        Costs[T] = Cost (T)
```

Pseudo code 19: Parallel best basis decomposition with shared memory programming – emulating the subband based data redistribution

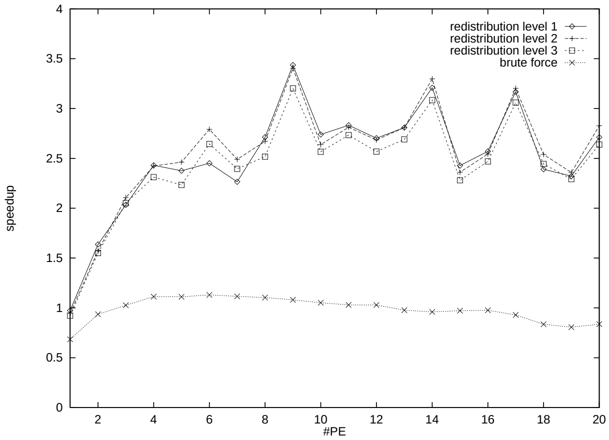


Figure 4.17: Speedups of best basis decomposition using the shared memory programming model

on the SGI Powerchallenge. Data sizes in use are 512, 256 and 256 in t-, x- and y-direction respectively for pyramidal decomposition and 256 in each direction for best basis decomposition. PVM was chosen as message passing library, because it produced the best results.

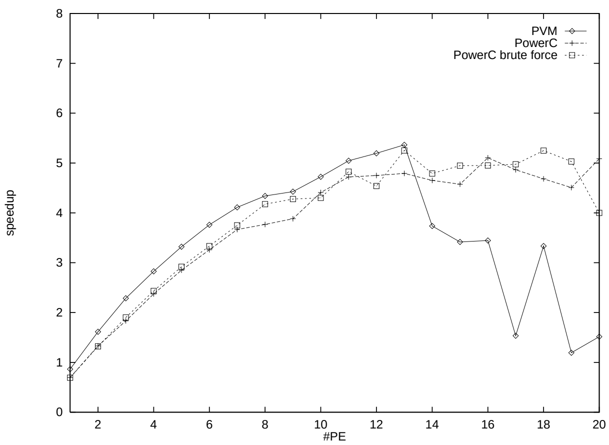
One can see that shared memory programming produces acceptable results for pyramidal decomposition. Moreover, it owns a better stability, so for higher $\#PE$ it performs even better than PVM (but these PVM-problems could possibly be solved).

For best basis decomposition, message passing is vast superior to shared memory programming. Although the speedups can be improved by simulation of the message passing approach, they remain far below the message passing results. The reason is that within best basis decomposition much more data exchange has to be performed between node PE. As it follows from shared memory programming that data exchange is mixed with computation, excessive transfer of big memory blocks results in worse performance.

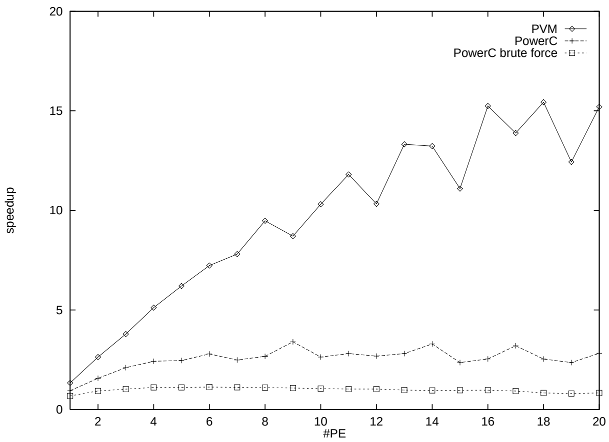
4.3.2 PVM vs MPI

The only platform where two comparable versions of PVM and MPI are available is the SGI PowerChallenge. So we will see now a direct comparison of performance with equal parameter sets for both PVM and MPI.

Within Figure 4.19 (a) PVM shows a slightly better performance for pyramidal decomposition. But the missing stability of PVM lets PVMs performance break down for higher numbers of PE. All experiences with PVM on that architecture show troubles with hangups, data errors and performance

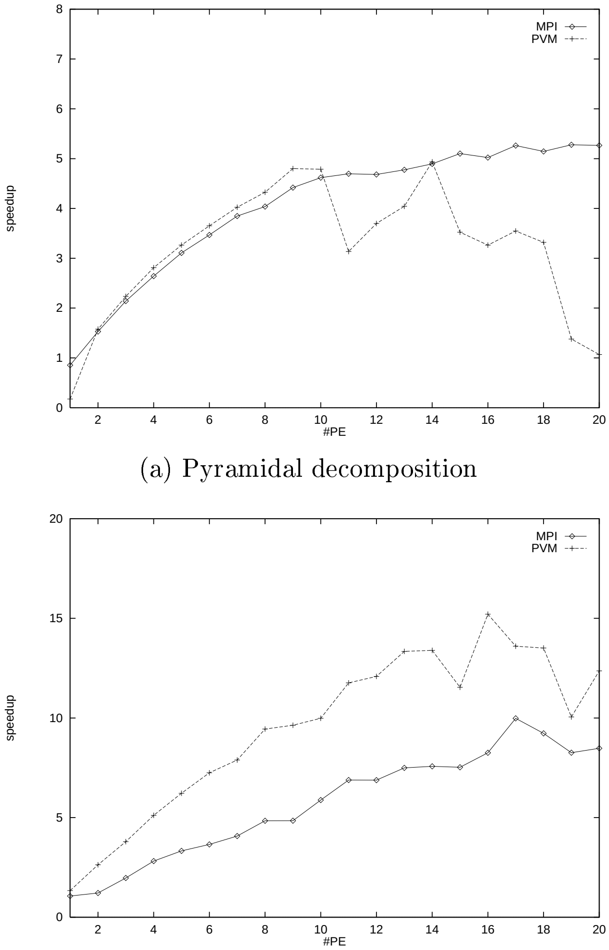


(a) Pyramidal decomposition



(b) Best basis decomposition

Figure 4.18: Comparison of shared memory programming and message passing



(a) Pyramidal decomposition

(b) Best basis decomposition

Figure 4.19: Comparison of PVM and MPI on a shared memory architecture

drops which all need a sure instinct in going around them.

But within Figure 4.19 (b) PVM shows a clear superiority over MPI for best basis decomposition. Although this is very impressing, it has to be stated that PVM wastes a lot of system resources. Large areas of shared memory have to be allocated, to make PVM work and therefore the processes take a long time to start up.

Nevertheless, with MPI there are a lot of sophisticated possibilities to improve performance:

- Definition of dynamic data types for optimal passing of data from the application to MPI. Unfortunately this advantage can not be exploited because of too complex data structures (i.e. subcuboids of a 3-D array) which vary from MPI-call to MPI-call.

- Non blocking communication allows the sending process to proceed, even though not all data have been handed over to MPI. Extra calls to wait for this transfer to complete can be (and are in this implementation) placed right before the parts of code that would overwrite not yet copied data.

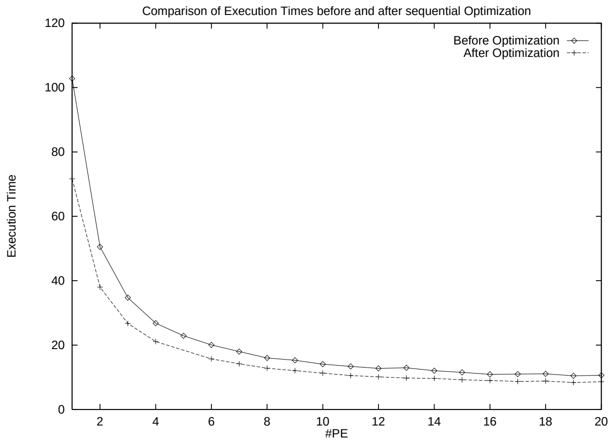
- Collective communication calls can automatically optimize data scatter and gather operations, excessively used in our algorithms. Unfortunately also these cannot be used, because in MPI 1 as well as in MPI 2 standard it is not possible to pass arrays of distinct data types in collective communication calls which would be necessary for our purpose.

As one can see, all these possibilities cannot make MPI perform better than PVM. This suggests that the concerning MPI implementation is not fully matured.

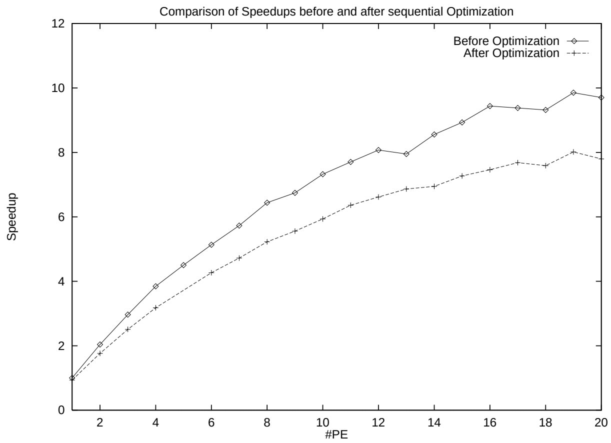
4.3.3 Influence of Sequential Optimization to Parallel Performance

As described in chapter 3, there is a sequential optimization of the filtering algorithm. Of course a change in execution time of the sequential part has some effect on the speedups too.

Figure 4.20 shows the performance improvement we can gain by optimizing the sequential part of the algorithm (applying the message passing algorithm, 2 parallel steps, 1 with redundant data, split along time axis). Of course higher calculation times make communication less important. Therefore, the speedup (computed with respect to the corresponding sequential algorithm) is higher in the non-optimized case.



(a) Execution Time



(b) Speedup

Figure 4.20: Influence of Sequential Optimization

4.4 Miscellaneous Experiences

4.4.1 Performance Troubles on Communication Channel Creation at Siemens Cluster

The Siemens cluster produced very bad speedups for a long time. Repeated testing using automatically generated run diagrams finally showed that most time was lost in the beginning, each time a PE sent some data to another PE for the first time. The guess that MPI had some problems while creating communication channels between PE, led to the following "hack":

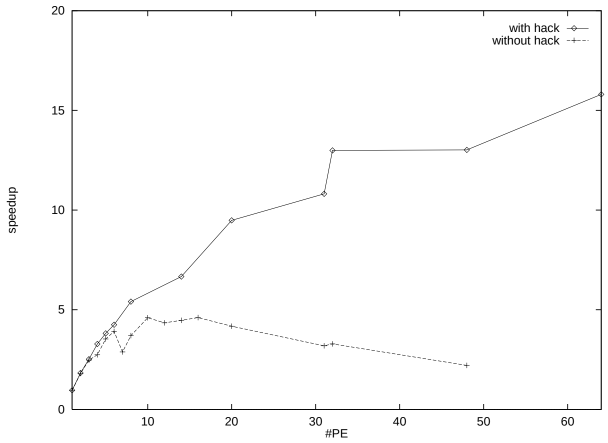
1. Perform an $n : n$ communication
2. Begin time measurement
3. Perform normal wavelet decomposition
4. End time measurement

All the results in this work are conducted with the help of this hack. Figure 4.21 shows a comparison of the speedups achieved with and without the hack.

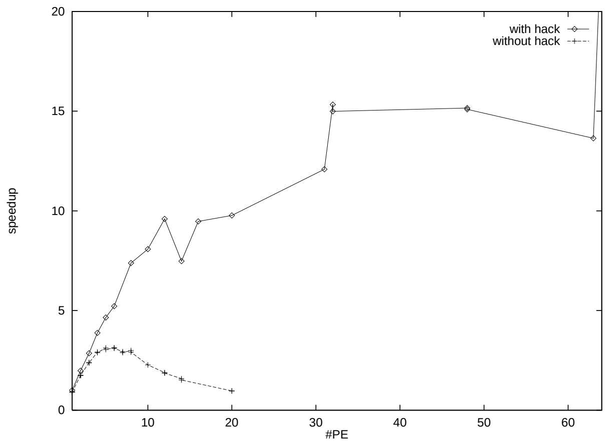
4.4.2 Changing the Operating System and Message Passing Library Version

In section 4.3.3 the influence of sequential optimization on the speedups were investigated. This has actually a longer history. In the first tests (see [27]) a significant improvement of execution time could be achieved by simply adding some dummy data between the frames in the memory (a few bytes) in order to reduce cache misses, as described in section 3.2.3. Moreover, this had a very interesting influence on the speedups: The cache misses produced a congestion of the bus. This effect was even worse, the more PE were involved. So speedups dropped for higher numbers of PE. This can be seen in Figure 4.22 for pyramidal decomposition on the SGI Powerchallenge using PVM.

Unfortunately hardware troubles with the used machine made a system upgrade necessary (now IRIX 6.5 and PVM 3.3.10 (SGI PVM for IRIX Version 3.1.1.0)). Afterwards, this effect could not be observed any more. This is very surprising, because the reason was actually a hardware problem. Instead the program showed a very annoying random speedup behaviour that led to the development of the optimized filtering algorithm in section 3.2.3.



(a) Pyramidal decomposition



(b) Full wavelet packet decomposition

Figure 4.21: Hacking performance problems on Siemens Cluster

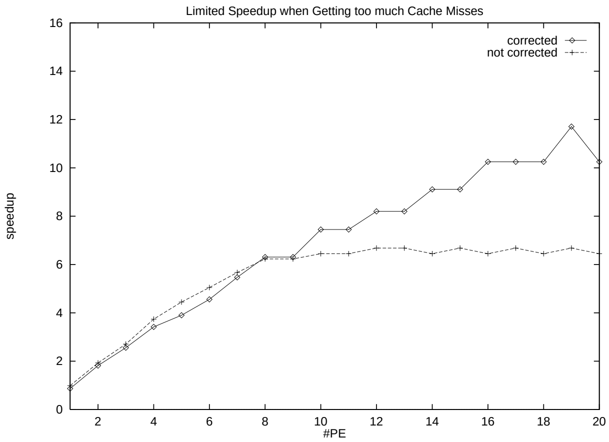


Figure 4.22: Effect of correcting the cache misses problem on speedup

4.4.3 PVM Needs Large Shared Memory Regions

Another example for the difficult and tricky handling of PVM is the environment variable `PVMBUFSIZE` that can be set on the SGI Powerchallenge. PVM makes use of shared memory blocks that are allocated at process startup for each process. Each time one process sends data to another, it copies the data into its own shared memory region and sets a flag in the receivers shared memory region, to signalize that there are data to be fetched by the receiver. If the shared memory region overflows, data is copied into a local buffer. The sender then has to wait until space gets freed from its shared memory region. This can slow down execution times significantly, if the shared memory region is too small. Its size can therefore be set with the environment variable `PVMBUFSIZE`.

Moreover, if messages sent are too big, a single message possibly does not even fit into the shared memory buffer, by what the program tends to hang, because it waits forever for the shared memory buffer to empty which never happens because the receiver is not informed about the presence of data. This situation occurs even more likely if several messages are sent to several PE.

Setting `PVMBUFSIZE` to a very high value can avoid these problems. Unfortunately each process creates a file with the size of `PVMBUFSIZE` in the file system at program startup which can take a long time. All these problems suggest the use of MPI instead of PVM.

Chapter 5

Conclusions

Within this work we have seen that the wavelet transform is a method to transform a (possibly multidimensional) signal or data set into its time-frequency components. Its results are well suited for various image processing applications such as for instance image or video compression. The wavelet transform offers several options:

Pyramidal wavelet transform is the standard signal decomposition with the fastest algorithmical complexity.

Wavelet packets offer a manifold of choices regarding time and frequency resolution. This can improve the results of the application, but requires more computational resources.

Best basis selection offers an fast and efficient method to find the best basis with respect to a given cost function.

We have also seen that it makes sense to parallelize the wavelet transform, because in realtime applications as well as in batch processing of large data sets there is a need for very high performance that cannot be achieved on single processor computers.

So we have used different programming paradigms to parallelize the introduced algorithms:

Message passing turned out to be very flexible in the optimization process. Especially the host node programming paradigm is a good choice for the message passing implementation, because it corresponds to a single input/output interface architecture. Moreover the message passing algorithms gave the best achievable speedups. But on the other hand message passing programs are very difficult to implement and to debug.

Shared memory programming is much easier. These implementations showed comparable speedups for pyramidal decomposition but much worse speedups for wavelet packet decomposition because of the more complex communication structure. Even with emulation of the message passing approach shared memory programming was not able to compete against the message passing implementation.

We have developed several parallel optimizations, which all showed different effectiveness on different computer platforms. Also some of them only are significant for special forms of wavelet transforms (pyramidal, wavelet packets):

Mixed approach for border data handling to find the best suitable compromise between redundant data calculation and data swapping as solution for the border data problem.

Multidimensional splitting to find the initial data distribution that minimizes the need for later inter-processor data exchange.

Partitioning of start data optimizes the startup phase of the pyramidal decomposition, which consumes a big part of the overall execution time.

Subband based redistribution for wavelet packet decomposition to avoid an overload of small messages by organizing data in a way such that no inter processor data exchange is necessary from a certain point in the decomposition process.

Two different communication libraries to implement the message passing algorithms were used and could be compared. Although the same method for communication were used (shared memory blocks) some differences were observed.

MPI is a very complex library of communication calls that offers a big potential for optimization. Its advantages are good availability and portability.

PVM produces slightly better performance, but shows an instable behaviour and bears many technical problems. It turned out to be difficult to handle and not available on most systems. The known advantages of PVM (process management, failure tolerance, interoperability) could not be exploited in this application.

Three different (and quite contrary) system architectures served as a platform for program execution, each of which has its advantages and disadvantages.

Cray T3E is a classical multicomputer. It produced the best speedups (but they did not exceed a value of 30). Although its network is expected to be the most efficient, changes in communication structure (number of messages, numbers of PE involved) exerted the most influence on the performance at this architecture. Nevertheless, high bandwidth is the reason that we got the best results on the Cray T3E.

Cluster is a competitive alternative to classical supercomputers. Todays highly developed network technologies make such architectures powerful and comparatively cheap. We observed good speedups and good scalability.

Shared memory computer is the smallest in this list, because its structure is not suitable for higher numbers of connected processor elements (star shaped communication structure). Therefore speedups are limited anyway. A big advantage of this architecture is that there is a variety of possible programming techniques available, although message passing was the most successful in our application.

Finally we saw that sequential optimization influences the parallel performance, such that normally speedups drop when we manage to reduce computation time.

Acknowledgments

The author wants to thank the NIC Jülich for providing access to its Cray T3E systems, the Paderborn Center for Parallel Computing (PC^2) for providing access to its Siemens Cluster and the Austrian Science Fund FWF who supported this work (project no. P11045-ÖMA).

Pseudo Code Index

Bibliography

[1] S.M. Akramullah, I. Ahmad, and M.L. Liou. A data-parallel approach for real-time MPEG-2 video encoding. *Journal of Parallel and Distributed Computing*, 30:129–146, 1995.

[2] S.M. Akramullah, I. Ahmad, and M.L. Liou. Performance of software-based MPEG-2 video encoder on parallel and distributed systems. *IEEE Transactions on Circuits and Systems for Video Technology*, 7(4):687–695, 1997.

[3] S.M. Akramullah, I. Ahmad, and M.L. Liou. Parallel MPEG-2 encoder on ATM and ethernet-connected workstations. In P. Zinterhof, M. Vajtersic, and A. Uhl, editors, *Parallel Computation. Proceedings of ACPC'99*, volume 1557 of *Lecture Notes on Computer Science*, pages 572–574. Springer-Verlag, 1999.

[4] A. Bruckmann, Th. Schell, and A. Uhl. Evolving subband structures for wavelet packet based image compression using genetic algorithms with non-additive cost functions. In *Proceedings of the International Conference "Wavelets and Multiscale Methods" (IWC'98), Tangier, 1998.* INRIA, Rocquencourt, April 1998. 4 pages.

[5] S. Chawla and D.M. Healy. Fast parallel implementation of the wavelet-packet best-basis algorithm on the MP-2 for real-time MRI. In A.F. Laine, M.A. Unser, and A. Aldroubi, editors, *Wavelet Applications in Signal and Imaging Processing IV*, volume 2825 of *SPIE Proceedings*, pages 409–421, 1996.

[6] C.H. Chu. Genetic algorithm search of multiresolution tree with applications in data compression. In H.H. Szu, editor, *Wavelet Applications*, volume 2242 of *SPIE Proceedings*, pages 950–958, 1994.

[7] A. Cohen and Q. Sun. An arithmetic characterization of the conjugate quadrature filters associated to orthonormal wavelet bases. *SIAM J. Math. Anal.*, 24(5):1355–1360, 1993.

[8] R.R. Coifman, Y. Meyer, S. Quake, and M.V. Wickerhauser. Signal processing and compression with wavelet packets. In Y. Meyer and S. Roques, editors, *Progress in wavelet analysis and applications, Proceedings of the International Conference "Wavelets and Applications", Toulouse, 1992*, pages 77–93. Editions Frontieres, 1993.

[9] R.R. Coifman and M.V. Wickerhauser. Entropy based methods for best basis selection. *IEEE Transactions on Information Theory*, 38(2):719–746, 1992.

[10] S. Corsaro, L. D'Amore, and A. Murli. On the parallel implementation of the fast wavelet packet transform on MIMD distributed memory environments. In P. Zinterhof, M. Vajtersic, and A. Uhl, editors, *Parallel Computation. Proceedings of ACPC'99*, volume 1557 of *Lecture Notes on Computer Science*, pages 357–366. Springer-Verlag, 1999.

[11] C.D. Creusere. Image coding using parallel implementations of the embedded zerotree wavelet algorithm. In B. Vasudev, S. Frans, and S. Panchanathan, editors, *Digital Video Compression: Algorithms and Technologies 1996*, volume 2668 of *SPIE Proceedings*, pages 82–92, 1996.

[12] I. Daubechies. Orthonormal bases of compactly supported wavelets. *Comm. Pure and Appl. Math.*, 41:909–996, 1988.

[13] I. Daubechies. *Ten Lectures on Wavelets.* Number 61 in CBMS-NSF Series in Applied Mathematics. SIAM, Philadelphia, 1992.

[14] A.C. Downton. Generalized approach to parallelising image sequence coding algorithms. *IEE Proc.-Vis. Image Signal Processing*, 141(6):438–445, December 1994.

[15] M. Feil and A. Uhl. Wavelet packet decomposition and best basis selection on massively parallel SIMD arrays. In *Proceedings of the International Conference "Wavelets and Multiscale Methods" (IWC'98), Tangier, 1998*. INRIA, Rocquencourt, April 1998. 4 pages.

[16] M. Feil and A. Uhl. Algorithms and programming paradigms for 2-D wavelet packet decomposition on multicomputers and multiprocessors. In P. Zinterhof, M. Vajtersic, and A. Uhl, editors, *Parallel Computation. Proceedings of ACPC'99*, volume 1557 of *Lecture Notes on Computer Science*, pages 367–376. Springer-Verlag, 1999.

[17] M. Feil and A. Uhl. Real-time image analysis using wavelets: the "à trous" algorithm on MIMD architectures. In D. Sinha, editor, *Real-Time Imaging IV*, volume 3645 of *SPIE Proceedings*, pages 56–65, 1999.

[18] J. Fridman and E.S. Manolakos. On the scalability of 2D discrete wavelet transform algorithms. *Multidimensional Systems and Signal Processing*, 8(1–2):185–217, 1997.

[19] K.H. Goh, J.J. Soraghan, and T.S. Durrani. New 3-D wavelet transform coding algorithm for image sequences. *Electronics Letters*, 29(4):401–402, 1993.

[20] E. Goirand, M.V. Wickerhauser, and M. Farge. A parallel two-dimensional wavelet packet transform and some applications in computing and compression analysis. In R. Motard and B. Joseph, editors, *Applications of Wavelet Transforms in Chemical Engineering*, pages 275–319. Kluwer Academic Publishers Group, 1995.

[21] Y. He, I. Ahmad, and M.L. Liou. Modeling and scheduling for MPEG-4 based video encoder using a cluster of workstations. In P. Zinterhof, M. Vajtersic, and A. Uhl, editors, *Parallel Computation. Proceedings of ACPC'99*, volume 1557 of *Lecture Notes on Computer Science*, pages 306–316. Springer-Verlag, 1999.

[22] T. Hopper. Compression of gray-scale fingerprint images. In H.H. Szu, editor, *Wavelet Applications*, volume 2242 of *SPIE Proceedings*, pages 180–187, 1994.

[23] B.J. Kim and W.A. Pearlman. An embedded wavelet video coder using three-dimensional set partitioning in hierarchical trees (SPIHT). In *Proceedings Data Compression Conference (DCC'97)*, pages 251–259. IEEE Computer Society Press, March 1997.

[24] C. Koc, G. Chen, and C. Chui. Complexity analysis of wavelet signal decomposition and reconstruction. *IEEE Trans. on Aereospace and Electronic Systems*, 30(3):910–918, July 1994.

[25] D. Krishnaswamy and M. Orchard. Parallel algorithm for the two-dimensional discrete wavelet transform. In *Proceedings of the 1994 International Conference on Parallel Processing*, pages III:47–54, 1994.

[26] R. Kutil and A. Uhl. Comparison of PVM and MPI on SGI multiprocessors in a high bandwidth multimedia application. In J. Dongarra,

E. Luque, and T. Margalef, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 6th European PVM/MPI Users Group Meeting*, volume 1697 of *Lecture Notes on Computer Science*, pages 149–156. Springer-Verlag, September 1999.

[27] R. Kutil and A. Uhl. Hardware and software aspects for 3-D wavelet decomposition on shared memory MIMD computers. In P. Zinterhof, M. Vajtersic, and A. Uhl, editors, *Parallel Computation. Proceedings of ACPC'99*, volume 1557 of *Lecture Notes on Computer Science*, pages 347–356. Springer-Verlag, 1999.

[28] R. Kutil and A. Uhl. Parallel adaptive 3-d wavelet analysis for fast and efficient video coding. In E.H. D'Hollander et al., editors, *Proceedings of ParCo 1999*. Imperial College Press, 1999. To appear.

[29] R. Kutil and A. Uhl. Optimization of 3-d wavelet decomposition on multiprocessors. *Journal of Computing and Information Technology (Special Issue on Parallel Numerics and Parallel Computing in Image Processing, Video Processing, and Multimedia)*, 2000. To appear.

[30] R. Kutil and A. Uhl. Parallel adaptive wavelet analysis (Invited Paper). *Future Generation Computer Systems (Special Issue on High Performance Numerical Methods and Applications)*, 2000. To appear.

[31] G. Lafruit and J. Cornelius. Parallelization of the 2D fast wavelet transform with a space-filling curve image scan. In A.G. Tescher, editor, *Applications of Digital Image Processing XVIII*, volume 2564 of *SPIE Proceedings*, pages 470–482, 1995.

[32] A. Laine and J. Fan. Texture classification by wavelet packet signatures. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(15):1186–1191, 1993.

[33] R.E. Learned and A.S. Willsky. A wavelet packet approach to transient signal classification. *Applied and Computational Harmonic Analysis*, 2:265–278, 1995.

[34] A.S. Lewis and G. Knowles. Video compression using 3D wavelet transforms. *Electronics Letters*, 26(6):396–398, 1990.

[35] A.R. Lindsey and J.C. Dill. Digital transceiver implementation for wavelet packet modulation. In H.H. Szu, editor, *Wavelet Applications V*, volume 3391 of *SPIE Proceedings*, pages 255–264. SPIE, 1998.

[36] S. Mallat. Multiresolution approximations and wavelet orthonormal bases of L^2. *Trans. Amer. Math. Soc.*, 315(1):69–87, 1989.

[37] S. Mallat. *A wavelet tour of signal processing.* Academic Press, 1997.

[38] L. Bacchelli Montefusco. Semi-orthogonal wavelet packet bases for parallel least-squares approximation. *Journal of Computational and Applied Mathematics*, 73:191–208, 1996.

[39] H. Nicolas, A. Basso, E. Reusens, and M. Schutz. Parallel implementations of image sequence coding algorithms on the CRAY T3D. Technical Report Supercomputing Review 6, EPFL Lausanne, 1994.

[40] J.N. Patel, A.A. Khokhar, and L.H. Jamieson. Scalability of 2-D wavelet transform algorithms: analytical and experimental results on coarse-grain parallel computers. In *Proceedings of the 1996 IEEE Workshop on VLSI Signal Processing*, pages 376–385, 1996.

[41] K. Ramchandran and M. Vetterli. Best wavelet packet bases in a rate-distortion sense. *IEEE Trans. on Image Process.*, 2(2):160–175, 1993.

[42] N. Saito and R.R. Coifman. Local discriminant bases. In A.F. Laine and M.A. Unser, editors, *Wavelet Applications in Signal and Image Processing II*, volume 2303 of *SPIE Proceedings*, pages 2–14, 1994.

[43] S. Sullivan. Vector and parallel implementations of the wavelet transform. Technical report, Center for Supercomputing Research and Development, University of Illinois, Urbana, 1991.

[44] C. Taswell. Satisficing search algorithms for selecting near-best bases in adaptive tree-structured wavelet transforms. *IEEE Transactions on Signal Processing*, 44(10):2423–2438, 1996.

[45] D. Taubman and A. Zakhor. Multirate 3-D subband coding of video. *IEEE Transactions on Image Processing*, 5(3):572–588, September 1993.

[46] A. Uhl. Adapted wavelet analysis an moderate parallel distributed memory MIMD architectures. In A. Ferreira and J. Rolim, editors, *Parallel Algorithms for Irregulary Structured Problems*, volume 980 of *Lecture Notes in Computer Science*, pages 275–284. Springer, 1995.

[47] A. Uhl. Wavelet packet best basis selection on moderate parallel MIMD architectures. *Parallel Computing*, 22(1):149–158, 1996.

[48] M.V. Wickerhauser. *Adapted wavelet analysis from theory to software.* A.K. Peters, Wellesley, Mass., 1994.

[49] M-L. Woo. Parallel discrete wavelet transform on the Paragon MIMD machine. In R.S. Schreiber et al., editor, *Proceedings of the seventh SIAM conference on parallel processing for scientific computing*, pages 3–8, 1995.

[50] Z. Xiong, K. Ramchandran, and M.T. Orchard. Wavelet packet image coding using space-frequency quantization. *IEEE Transactions on Image Processing*, 7(6):892–898, June 1998.

Curriculum Vitae

Name: Rade Kutil

Date of birth: June 17, 1971

Place of birth: Salzburg, Austria

Education and Employment:

- **1977–1981** Elementary school in Bischofshofen
- **1981–1985** General secondary school in St. Johann
- **1985–1990** Secondary school for electronics and communications engineering
- **1991–1998** Computer engineer at Eisenwerk Sulzau-Werfen
- **1994–2000** Studying informatics at Salzburg University