Vector and Parallel Wavelet Transforms for the analysis of time-varying signals

Andreas Uhl
Research Institute for Softwaretechnology
University of Salzburg

August 1994

Contents

1 Introduction **2**

2 Vectorization **4**

3 Parallelization **5**

3.1 Parallel Virtual Machine (PVM) 5

3.2 Implementation with PVM and load balancing 6

3.3 Results 7

4 Acknowledgements **8**

5 References **9**

1 Introduction

Wavelet transforms have been extensively used during the last ten years in areas like signal and image processing, mathematics and a wide range of applied sciences. The continous wavelet transform is defined as follows:

$$W_{a,b}(f) = |a|^{-1/2} \int_{-\infty}^{\infty} f(t)\, \psi\left(\frac{t-b}{a}\right)\, dt\,.$$

We always assume $\int_{-\infty}^{\infty} \psi(t)\, dt = 0$. The functions $\psi_{a,b}(t) = |a|^{-1/2}\, \psi\left(\frac{t-b}{a}\right)$ are called 'wavelets', the function $\psi(t)$ is often called 'mother wavelet'.

As a changes, the $\psi_{a,b}$ cover different frequency ranges, changing the parameter b as well allows us to move the time localisation center (which is in $t = b$). So all the wavelets are translated and dilated versions of only one function – the mother wavelet.

In most of the applications the 'fast wavelet transform' (which is based on a multiresolution analysis and was introduced by S. Mallat [7]) is used as the de-facto standard-algorithm. There have been made several attempts to parallelize this algorithm – in general [12] and in particular for image processing purposes [6], [2], [9].

When using this algorithm, the parameters (a, b) are sampled on a so-called 'dyadic' grid in the time -scale plane ($a = 2^j, b = n2^j$). Unfortunatly a wide class of applications requires denser point sets (a, b). For example we mention the detection of coherent structures in time series [3], the extraction of (multi)fractal properties in signals [1], singularity detection and investigation of singularity types [8] and applications in which one wants to produce a 'scalogram' [11].

One possible solution is to have 'M voices per octave' (which means replacing 2^j by $2^{\frac{j}{M}}$) – therefore a fast algorithm exists [10] which is computational M-times as expensive as the standard one. If arbitrary grids are needed this method fails – the continous wavelet transform has to be computed directly, computational even more expensive. Until now no attemps have been made to parallelize or vectorize this procedure, altough the computation of a scalogram corresponding to e.g. 1500 datapoints and $1 \leq a \leq 240$, using the Mexican hat wavelet takes about 30 cpu-minutes on a DEC/AXP 3000/400. We were facing this problem while analyzing highly irregular 'chaotic' 1D data from semiconductor cleanrooms in the context of the project EERP/762/AU-032 with Digital Equipment Cooperation.

Given a real (sampled) signal or a time series $f(t)$, $t = 1, \ldots, n$ the discretized

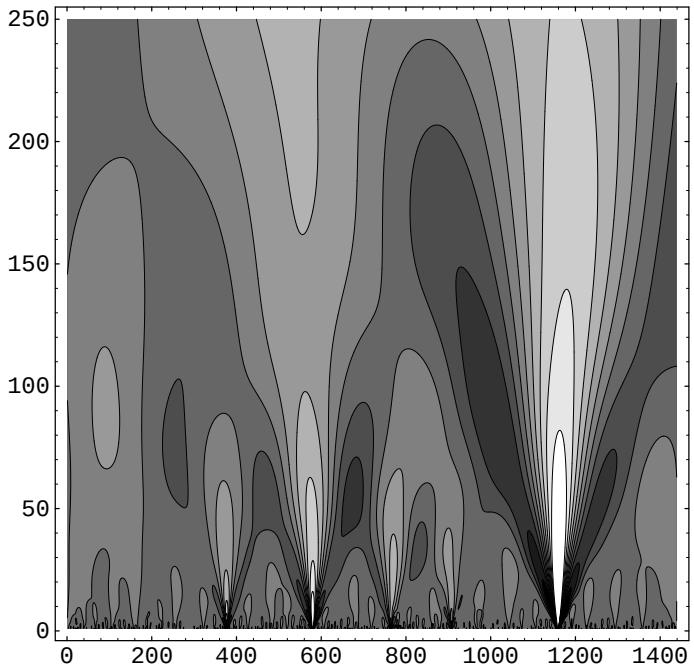

Figure 1: Scalogram of cleanroomdata provided by DEC

version of the continous wavelet transform can be written as

$$W_{a,b}(f) = |a|^{-1/2} \sum_{t=1}^{n} f(t)\, \psi\left(\frac{t-b}{a}\right) .$$

(We do not treat questions of signal extension here, see e.g. [13].) In order to obtain arbitrary grids in the time-scale plane, one has to compute $W_{a,b}(f)$ for arbitrary values of a and b (not necessarily with integer stepsize). A pseudocode for this procedure might look as follows (where `psi()` denotes the wavelet):

Structure of the sequential algorithm

```
for a=1,m                                          /*loop for parameter a*/
        for b=1,k                                  /*loop for parameter b*/
                wt[a][b]=0;
                for t=1,n
                        wt[a][b]=wt[a][b]+f[t]*psi((t-b)/a);
                endfor
                wt[a][b]=1/sqrt(a) * wt[a][b];
```

```
        endfor
endfor
```

In this paper we use the Mexican hat wavelet

$$\psi(x) := (1 - x^2)\, e^{\frac{-x^2}{2}}$$

as a typical example, but we do not exploit any symmetries that result from special properties of this function, since a general procedure is desired. For a specific implementation one should of course take care of these symmetries.

2 Vectorization

The algorithm is implemented on a four processor Convex C3440 of the academic computing center at the university of Linz. The Convex compiler provides several levels of optimization, including a parallelization at the highest level:

no optimization

-O0 basic block scalar optimization

-O1 -O0 plus function wide scalar optimization

-O2 -O1 plus vectorization

-O3 -O2 plus parallelization

The compiler uses the innerst loop (the t–loop) for vectorization and reports a vectorization of 71% when using directly the standard sequential code (algorithm 1, see figure 2). The problem inherent vectorization potential can be exploited better by partitioning the t–loop into 3 loops (algorithm 2): one loop for preprocessing the grid points, one loop for calculating the inner products und one loop for summing up. The first two loops can be fully vectorized and the third one is a vector reduction. Despite of the load–store overhead this results in a better efficiency (see figure 2).

In vector concurrent mode the compiler distributes automatically the t–loop among the processors, the timings were taken in single–user mode to exploit the machines' facilities best. In 'real life' with its unpredictable load situation (in multi–user mode) load balancing is performed automatically by the operating system and lower speed–ups are achieved.

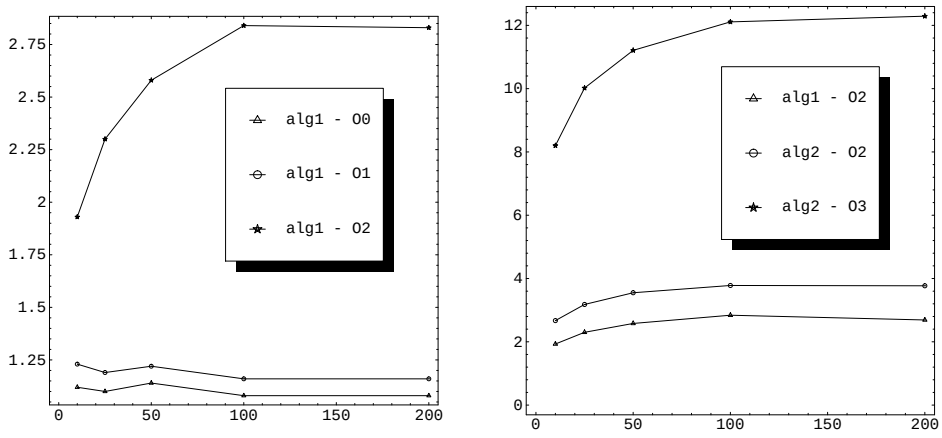

Figure 2: Speedup for algorithm 1 (different optimization) and 2 with increasing problemsize

3 Parallelization

Now a parallel implementation on a workstation cluster with PVM using a coarse grained parallelism is discussed. All the issues treated here can easily be transferred to any moderate parallel MIMD system, moreover PVM is now available for most of the platforms commonly used. The cluster consists of eight DEC/AXP 3000/400 which are connected by FDDI (fiber distributed data interface) and are located at the RIST++.

3.1 Parallel Virtual Machine (PVM)

The Parallel Virtual Machine is a software system that enables a collection of heterogeneous computers to be used as a coherent and flexible parallel computational resource. It is especially suited for the developement and execution of large concurrent or parallel applications that consist of many interacting, but relatively independent components [4]. PVM was developed initially at the Emory University and Oak Ridge National Laboratory.

The individual computers may be shared- or local-memory multiprocessors, vector supercomputers or scalar workstations, that can be interconnected by a variety of networks (ethernet, FDDI, etc.). User programs written in C or Fortran are provided access to PVM through the use of calls to PVM library routines for functions

such as process initiation, message transmission and reception. The PVM system handles message routing, data conversion for incompatible architectures and other tasks that are necessary for operation in a heterogeneous network environment. For more details see [5].

3.2 Implementation with PVM and load balancing

Since the results for the different values of the parameter a are evaluated independently, the outer loop (the a - loop) may be distributed among the processors. On a homogeneous network, where the users disposes 100% of the machines' capacity on each node, this can be done by partitioning the range of a into equal parts for each node. If there are other processes running or a heterogeneous network is used, a processor with high workload or a slower one would cause a delay in the whole computation. With a finer partitioning of the range of a it is possible to react on changes in the systems performance. Therefore the algorithm is implemented using the master–slave paradigm and the asychronous pool of tasks methodology.

A master program is responsible for partitioning the range of a into suitable equalsized tasks and for the distribution of these tasks. (If the grid is not regular, it is also the masterprogram that is responsible for the creation of suitable sized task.) On each processor a slave program receives the task, does the computation and sends the result back to the masterprogram. The tasks are distributed dynamically, which means that whenever a slave has sent back his result to the masterprogram, the master will send another task to this idle slave until all the work is done and the pool of tasks is empty. Moreover each slaveprogram is able to determine the current load situation of the machine it is running on – if the load is too high compared to the size of the tasks, the slaveprogram falls idle for some appropriate time until the load situation is investigated again. Another approach is to use two pools of tasks with different tasksizes, where the loadsituation on the slaveprograms' machine determines if a small or a large task will be executed.

The size of the tasks has to be specified by the user – one must take into account that on the one hand small sized tasks cause less idle time at the end of the computation (especially in a network with a very variing load situation), on the other hand the amount of communication rises the smaller the tasks get.

Structure of the masterprogram

```
enroll in pvm
get size of tasks
```

```
get number of available processors n
invoke n slaves
send global parameters (data, range of a)
send first n tasks to slaves
while (pool of tasks is not empty)
        receive result from a slave
        send next task to that slave
endwhile
terminate slaves
store results
exit
```

Structure of the slaveprogram

```
enroll in pvm
receive global parameters
'infinite loop'
        while (load is too high)
              fall idle
        endwhile
        receive task
        compute result
        send result to master
end loop
```

When there is no further work, the slaveprocess blocks at the receive–statement until it is terminated by the masterprocess.

3.3 Results

We were mainly interested in two questions: would it be possible to achieve good speed-up when using a rather small task size and what would be the influence of the tasksize on the performance using different nodenumbers. All the measurements were taken in single-user mode. This inhibits the small task sizes from showing their real strength).

Even with the smallest tasksize investigated (one task corresponds to one value of parameter a) an almost linear speed-up is achieved (see figure 3), which is not really surprising when using FDDI that makes communication fast. Concerning the influence of the tasksize on the performance it can be concluded that the bigger the

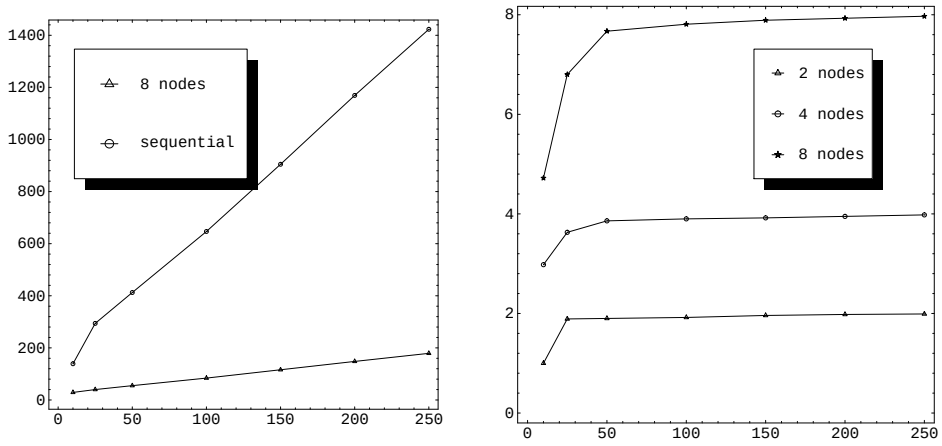

Figure 3: Timing and speedup with small tasksize for increasing problemsize

nodenumber is, the smaller the tasksize may be chosen without losing efficiency (see figure 4).

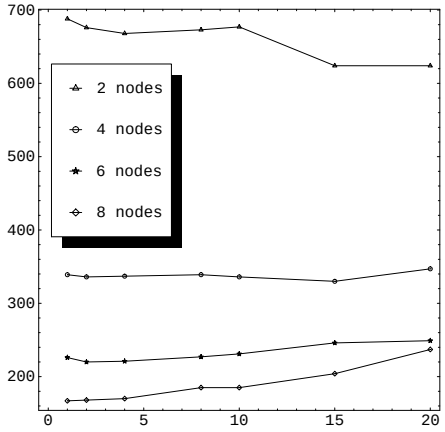

Figure 4: Timings for different tasksizes and fixed problemsize

4 Acknowledgements

We wish to thank Kurt Krenn for his help in enhencing the vector code, Robert Kolmhofer for providing the computing facilities on the Convex and Charly Entacher for providing his Mathematica packages.

References

[1] A. Arneodo, F. Argoul, E. Bacry, J. Elezgaray, E. Freysz, G. Grasseau, J. F. Muzy, and B. Pouligny. Wavelet transform of fractals. In [?], pages 286–354.

[2] H. Essafi and M.M. Pic. Application of parallel computing to wavelet transform. In Y. Meyer and S. Roques, editors, *Progress in wavelet analysis and applications, Proceedings of the International Conference "Wavelets and Applications", Toulouse, 1992*, pages 683–687. Editions Frontieres, 1993.

[3] W. Gao and B.L. Li. Wavelet analysis of coherent structures at the athmosphere-forest interface. *Journal of Applied Meteorology*, 32(11):1717–1725, 1993.

[4] A. Geist. PVM 3 beyond network computing. In J. Volkert, editor, *Parallel Computation*, Lecture Notes in Computer Science 734, pages 194–203. Springer, 1993.

[5] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM 3.0 User's Guide and Reference Manual*. Tennessee, 1993.

[6] T.L. Huntsberger and B.A. Huntsberger. Hypercube algorithm for image decomposition and analysis in the wavelet representation. In Walker and Stout, editors, *Proceedings of the 5th Distributed Memory Conference*, pages 171–175. IEEE Computer Society Press, 1990.

[7] S. Mallat. A theory for multiresolution signal decomposition: The wavelet representation. *IEEE Trans. on Patt. Anal. and Mach. Intell.*, 11(7):674–693, 1989.

[8] S. Mallat and W. L. Hwang. Singularity detection and processing with wavelets. *IEEE Trans. on Inf. Theory*, 38(2):617–643, 1992.

[9] M. Misra and V. K. Prasanna. Parallel computation of 2-D wavelet transforms. In *Proc. of the 11th IAPR Int. Conference on Pattern Recognition*, volume IV, pages 111–114. IEEE Comput. Soc. Press, 1992.

[10] O. Rioul and P. Duhamel. Fast algorithms for discrete and continous wavelet transform. *IEEE Trans. on Inf. Theory*, 38(2):569–586, 1992.

[11] O. Rioul and M. Vetterli. Wavelets and signal processing. *IEEE Signal Proc. Mag.*, pages 14–38, October 1991.

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

[13] Carl Taswell and Kevin C. McGill. Wavelet transform algorithms for finite-duration discrete-time signals. *ACM Transactions on Mathematical Software*, 20(3), 1994. To appear.