

# A Fast Parallel Implementation of the Wavelet Packet Best Basis Algorithm on the MP-2 for Real-Time MRI \*

Sumit Chawla      Dennis M. Healy Jr.  
Departments of Mathematics and Computer Science  
Dartmouth College  
Hanover, NH 03755

## Abstract

Adaptive signal representations such as those determined by best-basis type algorithms have found extensive application in image processing, although their use in real-time applications may be limited by the complexity of the algorithm. In contrast to the wavelet transform which can be computed in  $O(n)$  time, the full wavelet packet expansion required for the standard best basis search takes  $O(n \log n)$  time to compute. In the parallel world, however, both transforms take  $O(\log n)$  to compute when the number of processors equal the number of data elements, making the wavelet packet expansion attractive to implement.

This note describes near real-time performance obtained with a parallel implementation of best basis algorithms for Wavelet Packet bases. The platform for our implementation is a DECMpp 12000/Sx 2000, a parallel machine identical to the MasPar MP-2. The DECMpp is a single instruction, multiple data (SIMD) system; such systems support a data parallel programming model, a model well suited to the task at hand.

We have implemented the 1D and the 2D WPT on this machine and our results show a significant speedup over the sequential counterparts. In the 1D case we almost attain the theoretical speedup, while in the 2D case we increase execution speed by about two orders of magnitude. The current implementation of the 1D transform is limited to signals of length 2048, and the 2D transform is limited to images of size:  $32 \times 32$ ,  $64 \times 64$ , and  $128 \times 128$ . We are currently working on extending our transform to handle signals and images of larger size.

## 1 Introduction

Best-basis algorithms, first described by Coifman and collaborators [1], have found many applications in signal and image processing. These algorithms require the simultaneous representation of a signal in whole family of related bases, followed by a search for the most advantageous of these representations. A standard choice for this family is the collection of wavelet packet bases associated with a particular sub-band filter pair. We will refer to the expansion of a signal in this collection of wavelet packet bases as the (full) wavelet packet transform (WPT).

There is some computational penalty for the simultaneous expansion of a signal in many bases at once, as opposed to picking just one of these bases, such as the wavelet transform, a priori. While the wavelet transform (WT) can be computed in  $O(n)$  time, the WPT takes  $O(n \log n)$  time to compute. In the parallel world however, the latter transform becomes attractive to implement. Consider the ideal case, where the number of processors equals the number of data elements. In the WT at each iteration of the algorithm the data to be processed halves in size resulting in loss of parallelism; this corresponds to an increase in the number of processors sitting idle with each iteration. In contrast, for the WPT the work remains the same at each iteration and no processor sits idle. Hence implementing a WPT transform would be no more expensive than implementing a WT since the otherwise idle processors can now be made to do work. In either case we have  $O(\log n)$  levels to process and  $O(n)$  work at each level – the idle processors in the WT must be accounted for since we are using a SIMD machine. The work at each level can be done in parallel making each transform  $O(\log n)$  to compute; this yields a greater speedup for the WPT over the WT thereby making the former attractive to implement.

In practice, the data is far greater than the number of processors available and the added complexity of the WPT brings with it additional managerial costs, communication, data management, etc; the theoretical

---

\*This work supported in part by ARPA, as administered by the AFOSR under contract DOD F4960-93-1-0567

speedup is therefore rarely achieved. Nevertheless, we have found that a relatively simple parallel implementation of the the WPT produces near real-time performance for signals and images of reasonable size. In this note, we present some details of this implementation and experimental results to indicate its performance.

There has been other work on parallelizing wavelet packet computations; we are aware of two projects, both implemented on multiple instruction, multiple data (MIMD) architectures. In [7] the transform is implemented on a hypercube iPSC/860 for purposes of parallel numerical linear algebra. The goal of this work is to be able to compute the wavelet packet coefficients at a given level of the WP library tree given an input vector; as a result no best basis search is needed. This is done by constructing a special matrix which can then be directly multiplied with the input vector to give the coefficients at the desired level. Although the entire wavelet packet tree decomposition is not computed, it can be modified to do so; however the approach used for their application may not be well-suited for this purpose. In [8] a  $2 - D$  transform is implemented on a cluster of workstations running the Parallel Virtual Machine (PVM) (see for example [5]) a popular software tool that orchestrates a cluster of workstations to function as a single high-performance parallel machine. The idea here is to map the data to the processors, including border data needed for the decomposition, decompose the data, redistribute it and continue until there are as many subbands as there are processors. At this point each processor is assigned a subband and the processors proceed independently. A best basis search algorithm is also included in this implementation.

In contrast, we present a parallel implementation of this transform on the MasPar, a single instruction, multiple data (SIMD) machine. Our implementation is fine-tuned to the processor topology of this machine and makes use of the entire processor set whenever there are more data elements than processors. Our implementation is targeted at a real-time Magnetic Resonance Imaging (MRI) application, and our performance numbers indicate that our implementation will be amenable to this task.

The rest of the paper is organized as follows: we start by indicating some motivation for our work, and follow with a description of the platform for our implementation. Data parallel filtering is presented next, and the mapping of image data onto the mesh of processors is described. Performance numbers are presented next. Finally, an alternate data mapping is proposed, which we believe will provide greater speedup.

## 2 Motivation

Beyond theoretical interest, we have been motivated to consider real time best-basis type algorithms in our work with adaptive Magnetic Resonance Imaging (MRI). This work requires the best-basis type algorithms to be run repeatedly at a rate determined by MRI measurement time; the results presented in this paper indicate this should be possible. A summary of our MRI work follows; for more detail see [6]

MRI has become an essential tool in clinical medicine; however, it can be limited in resolution and speed of image acquisition. Depending on the contrast in the images, acquisition time on a standard scanner can vary from a second to many minutes. Imaging time is largely determined by physical and engineering constraints on the rate at which the scanner takes measurements. The measurement process may be described mathematically as the projection of a function representing tissue properties onto the elements of a basis of a function space; for standard MRI, the Fourier basis is used. Each measurement requires a certain amount of time, and typically many measurements must be made. Most fast imaging methods presently under consideration are concerned with increasing the rate at which these measurements are acquired. Unfortunately these methods can require expensive hardware modifications and can adversely affect contrast and resolution.

We have taken a different approach and consider the possibility of reducing the number of measurements required for certain types of images. Taking advantage of the flexibility of the MRI modality, we adaptively change from measuring with the standard Fourier basis to a basis which incorporates prior knowledge about the imaging task at hand and information from previous measurements. The goal is to obtain the image data in as compact a form as possible. This reduces the total number of projections required and should improve image speed even if the measurement rate remains the same.

We have studied dynamic localization for dealing with motion in MRI. Motion can be a real problem in standard Fourier encoded MRI, with errors due to even localized motion during the measurements being reflected in global reconstruction artifacts. We find that the measurement basis may be modified to the task of adapting the encoding to localized changes in otherwise static anatomy. The reduced encoding requirements attainable using adapted bases enable rapid imaging of a changing object, a requirement for functional imaging of the brain or other organs. A local trigonometric approximate K-L basis adapted to a functional sequence provides full resolution imaging with a small number of coefficients, since the basis

can localize to the region of interest. A real-time best-basis algorithm should permit adaptive tracking of changes in the image, a capability of paramount importance as the features of interest cannot typically be known in advance. See [6] for more particulars and simulations.

### 3 Machine Architecture and Language Considerations

We implemented the full Wavelet Packet Transform on the DECmpp 12000/Sx 2000 [3, 4] (named “cascade” at Dartmouth), a parallel machine identical to the MasPar MP-2. The code was written in the MasPar Application Language, (MPL). This section describes a few particulars of this machine and language.

#### 3.1 Architecture

The DECmpp 12000/Sx 2000 is a single instruction, multiple data (SIMD) system. Such systems support a data parallel programming model, where a single instruction runs on multiple processing elements, each operating on different data elements. This might seem to be an overly constrained programming model in the parallel programming domain, but there are several suitable applications, image processing for example, that are data parallel.

The DECmpp 12000/Sx consists of a console and a data parallel unit (DPU). The console is a uniprocessor running the ULTRIX operating system providing standard I/O devices; modules of singular (non-parallel) code may execute on the console. Cascade’s console is a DECstation 5000/240. The DPU is where all the parallel processing is done. It consists of the Array Control Unit (ACU), the Processor Element (PE) array, and the PE communication system.

The ACU controls the PE array and performs operations on singular (non-parallel) data. Code operating on singular (non-parallel) data within modules of parallel code executes on the ACU.

The PE array consists of the processors and their local memories. On cascade the PE array topology is a mesh of size  $32 \times 64$  (=2048 processors) with each processor having 64K local memory.

The PE communication system provides the mechanism for data movement between PE’s. There are two networks for routing data on the DECmpp 12000/Sx 2000:

- The X-net, which connects each PE to its eight immediate neighbors in the mesh.
- The global router, which supports arbitrary data movement patterns.

The X-net is often faster than the global router for communications between different processors, although its performance degrades with the step-size, which is the distance between the communicating PE’s. For a step-size of one, the X-net outperforms the global router by approximately two orders of magnitude.

The DECmpp 12000/Sx 2000 includes an I/O subsystem that supports both synchronous and asynchronous I/O. We currently do not use this subsystem, although we do intend to experiment with some asynchronous routines when we run some simulations.

#### 3.2 Programming Language

At the start of the project we evaluated the programming languages available, and used performance as the criterion in doing so. We had two choices:

- MasPar Fortran (MPF), which is based on the Fortran 77 ANSI standard with array extensions from the ANSI Fortran 90 proposal.
- MasPar Application Language (MPL), which is an extension of ANSI C with the new data type *plural*, a type for defining parallel data.

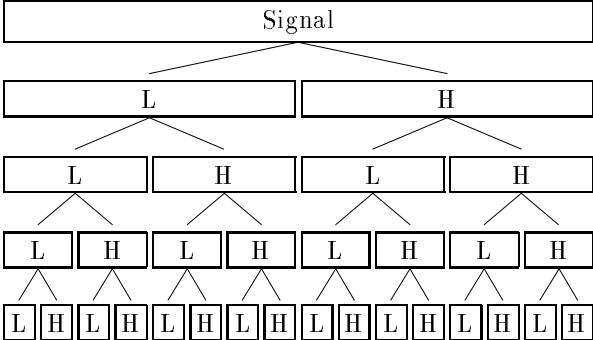
MPF is a high level language in which data movement and virtualization (defined below) get handled automatically by the compiler. While this has an advantage of easing the programming task, it brings with it costs that are hidden from the programmer, and optimizations based on the task at hand are not possible. MPL on the other hand is a lower level language where such issues are under explicit control of the programmer. The programmer is given the freedom to decide what goes where and how, allowing for optimizations on a per task basis. The price one pays for making this choice is that of development time.

Performance was critical to us and we therefore chose MPL as the language to code in. While this added to the complexity of programming, we believe that our performance numbers justify our choice. We do not

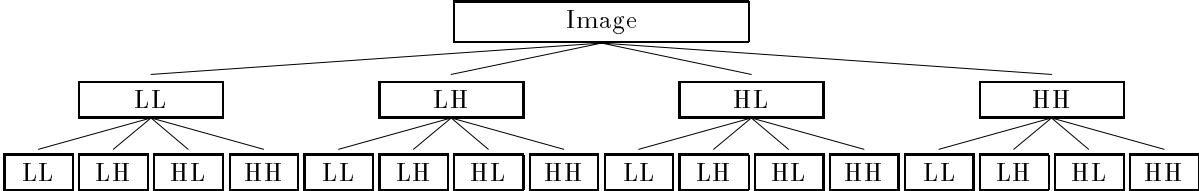
have have a corresponding implementation in MPF and therefore do not have any data to support our claim. It will be interesting to provide a comparison between the two.

## 4 Parallel Implementation of Sub-band Filtering

In a wavelet packet decomposition the signal is convolved with two kernels, one performing low pass filtering and the other high pass filtering. The resulting segments are down-sampled and we iterate on the new segments. Figure 1 illustrates the analysis process for a 1D and a 2D signal. Convolution is central to the WPT, and a key to good performance. In this section we present a fast convolution algorithm specialized to the mesh. In particular, we exploit the speed of the X-net and use it extensively.



(a) Analysis/Synthesis of a 1D signal.



(b) Analysis/Synthesis of a 2D signal.

Figure 1: 1-D and 2-D Wavelet Packet decompositions.

### 4.1 Convolution

In visualizing convolution, one generally imagines a filter mask which is moved along the data; at each position the mask is used to form a weighted sum of the segment of data it currently touches, as indicated in Figure 2. This process is repeated for each position of the mask over the the data. Boundary problems are handled by a periodic extension and/or reflection of the signal.

On the MasPar we can take a somewhat different approach, as in Figure 3. Instead of moving the filter mask along the data, we line up appropriately sized chunks of data in a sequence of processors, and apply the mask in parallel to all of the chunks at once. Specifically, each processor is responsible for computing a single sample of filter output. To compute the filter’s output at this discrete time index, the filter mask must be applied to a corresponding segment of the input data. This segment of the data is stacked into the local memory of the processor responsible for computing the corresponding filter output sample. If the data is initially laid out with one sample per processor, the requisite additional samples are copied onto the stack of each processor using a sequence of left shifts, as in Figure 4. Each shift copies the top of the current stack of a processor onto the top of its left hand neighbor’s stack; this is done in parallel for the entire processor array at once. This process is repeated until the appropriate data segment resides on each processor. Since each shift involves only nearest neighbor communication on the X-net, this is a very efficient approach.

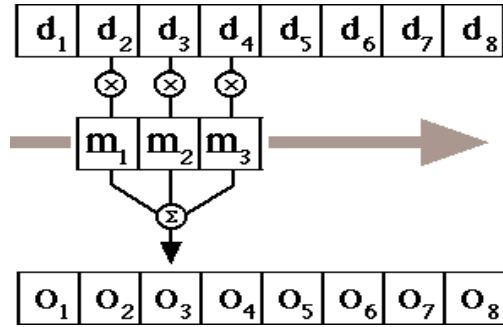


Figure 2: Sequential view of convolution.  $d_i$ 's comprise the input data; Filter mask coefficients  $m_i$  applied to data  $d_i$  produce output samples  $o_i$ .

Once this is done, each processor holds the data segment required for computing its own particular output value. The filter mask is applied simultaneously to all processors, resulting in one sample of output for each processor.

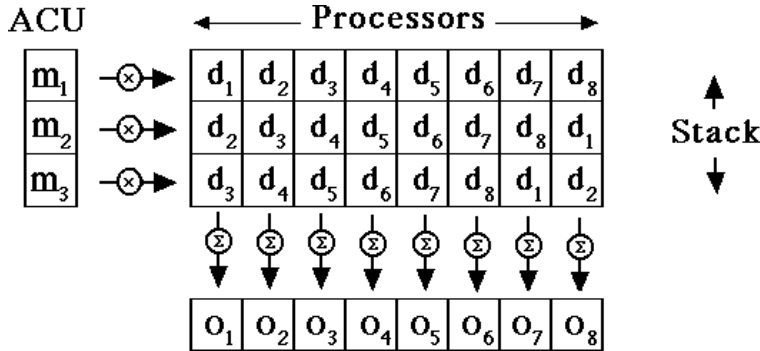


Figure 3: Parallel view of convolution;  $d_i$ 's in each column comprise input data samples needed for one output sample,  $o_i$ . Filter coefficients  $m_i$  reside on the ACU, and are applied in parallel across the rows (processors), followed by summing down the columns (along the stacks of the processors).

In the iterated convolutions required by the wavelet packet transforms, the outputs from one convolution provide the input data for the next stage. This means that the outputs of one stage must be moved among the processors in order to assemble the input data segments for the next stage. This permutation is done in parallel on all the processors using the router; its efficient performance is critical for a fast implementation. The issue is further complicated by the necessity of simultaneously applying *two* filters to the signal in order to obtain both the high and low pass channels of a sub-band split. We address these issues next.

## 4.2 Implementing Sub-band Splitting: Data Movement

In the wavelet packet transform we modify the preceding approach a bit to account for the fact that we simultaneously compute both high and low pass filters and keep decimated output. In practice, decimation is rarely performed; instead, convolution is evaluated at every other sample, once for the low pass filtering and once for the high pass filtering. To perform both these operations in parallel we assign the task of low-pass filtering to the even-numbered processors and the high pass filtering to the odd-numbered processors. Again, each processor is responsible for computing one sample of the appropriate filter output.

Since we are applying two different filters (high and low pass) we can no longer apply the ACU globally to the processor array. Instead, we store the low pass filter mask on each even processor, and the high pass filter mask on each odd processor. These must be applied to the appropriate segment of the input data. In order to do this, a given even numbered processor and its right hand odd numbered neighbor must contain identical data segments. We saw in Figure 4 an efficient sequence of shifts may be used to lay out the

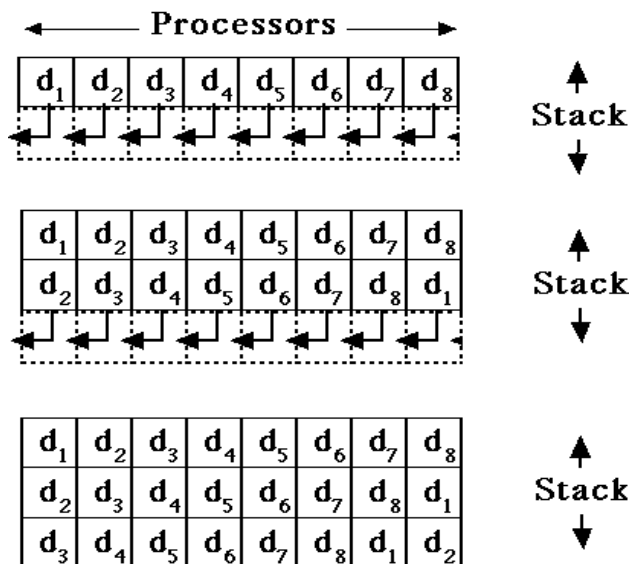


Figure 4: Data movement to set up parallel convolution.  $d_i$ 's comprise the input data initially lined up one per processor. A sequence of shifts is performed, each copying the top of a processor's stack and pushing it onto the top of its neighbor's stack.

input data onto the processors in such a way that the data segments for two neighboring processors differ only two positions; namely the first and last data element. Overwriting the last data element of each odd processor with the first data element of its left hand neighbor results in identical data for each even-odd pair, modulo a rotation. This operation can again be done in parallel on all odd processors, using the X-net for the overwrite. Now with the coefficients of the high pass filter rotated correctly, both the low pass and high pass convolutions can be carried out in parallel, as illustrated in Figure 5. At each step we multiply a filter coefficient, low pass on the even processors and high pass on the odd processors, with the corresponding data element, accumulate the sum, and iterate doing the same for the remaining filter coefficients. Boundary problems are handled using periodization.

Note that the output of the convolution routines results in the low pass and high pass coefficients being alternately placed. We need to group together the low pass and the high pass output samples before we iterate; this requires us to permute the data. Once permuted, the output now becomes the input to the convolution routines and the procedure is repeated. Note that the filter coefficients, once initialized, are not altered.

Permutations are carried out using the global router. The router is generally slower than the X-net, but for permutations where the data elements move large distances on the mesh the advantage shifts to the router. This is the case for the permutations encountered in this phase of the algorithm.

The permutation indices are static in our implementation and can therefore be computed and stored off-line. The disk access time is however slower than the time to re-compute the indices, and therefore we currently re-compute the indices each time we permute the data. We are currently looking at ways to remove this cost and optimize data movement using the router. We are experimenting with the work of [10, 2] to accomplish this.

### 4.3 Advantages of our Approach

The above technique generalizes to 2D signals in an easy manner. Row convolutions are performed in the manner described above, the column convolutions being carried by moving elements from the south; the latter is done by simply changing the direction of the X-net.

With regards to performance, we obtain the following advantages:

1. Each data element in the original signal resides on a processor and the remaining elements are brought to the processor using the X-Net. This is a fast, global operation.

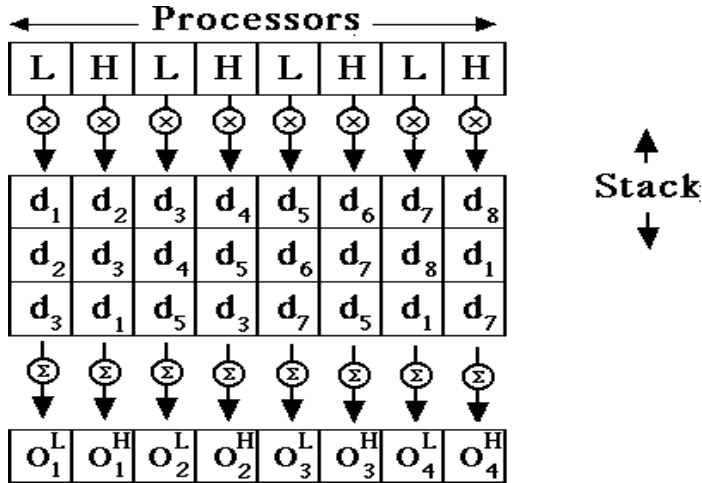


Figure 5: Parallel implementation of sub-band splitting;  $d_i$ 's are data; L's and H's correspond to the coefficients of the low pass and high pass filters. The  $o_{i_s}^L$  and  $o_{i_s}^H$  correspond to low pass and high pass coefficients respectively. In this case, the multiplication is down the columns, not across the rows.

2. The number of X-Net operations are proportional to the length of the filter.
3. The multiplication and accumulation take place within each processor and no communication takes place.

## 5 Virtualization

When an image has more pixels than there are PE's we must provide virtual processors for the excess pixels. This is provided by a mapping which folds the image onto the available processors; each processor is assigned one or more pixels. This process, called virtualization, is crucial to performance.

We implemented a one-dimensional and a two-dimensional transform. We limited the first transform to signals of length 2048, which equals the number of processors on cascade; hence, no mapping was required. We therefore focus only on the two-dimensional transform in this section.

In the current implementation we only compute transforms of images with the following dimension:  $32 \times 32$ ,  $64 \times 64$  and  $128 \times 128$ . Of course, larger sized images, typically  $256 \times 256$  and  $512 \times 512$ , are common in practice and we are extending our application for such instances. We now describe mappings for the image sizes considered.

Figure 6 shows how processors are numbered on the mesh; we will use this convention in succeeding figures and equations to indicate how the images in the size range considered get mapped onto this mesh.

In the case of a  $32 \times 32$  image, Figure 7, we have more processors than data elements and therefore no virtualization is necessary. We simply use the identity map  $I(m, n) \mapsto P(m, n)$ , where picture element  $I(m, n)$  of the image (pixel) gets mapped to processor  $P(m, n)$ . Under this mapping, the image is mapped onto the left half of the mesh.

In the second case of a  $64 \times 64$  image virtualization becomes necessary since we have more data elements than processors. This is implemented by stacking multiple pixel values into the local memory of a single processor element, Figure 8. The processor number and the level in its stack to which a pixel is assigned is determined by the following mapping

$$I(m, n) \mapsto P(i, j, k) = P((m \bmod 32), n, \lfloor \frac{m}{32} \rfloor),$$

where element  $I(m, n)$  of the image gets mapped to processor  $(i, j)$  at stack level  $k$ , and 32 is the number of rows in the mesh, Figure 9. Note here that the number of columns in the image equal the number of columns in the mesh and therefore the identity map may be used for the column index; this is not the case for the

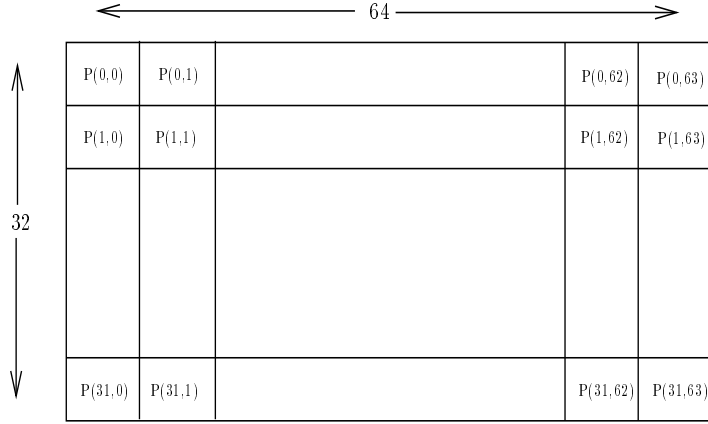


Figure 6: Processor numbering on the mesh

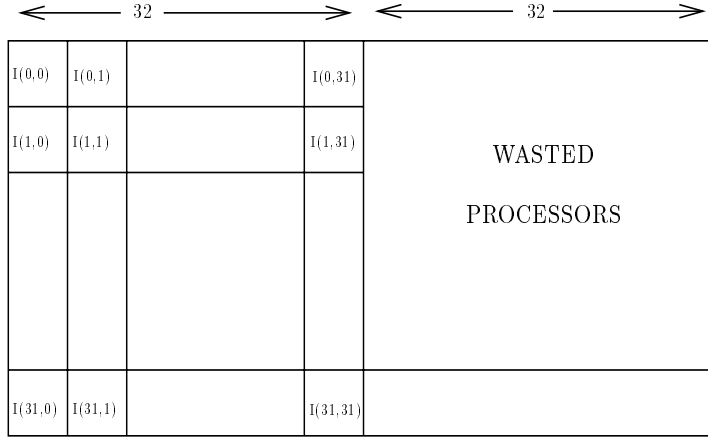


Figure 7: Processor map of a 32 × 32 image

rows where we create a stack layer to provide a virtual mesh to map to. The height of this stack is 2, which gives the load per processor.

In the third case of a 128 × 128 image, Figure 10, the identity map can no longer be used to map the columns index. Both rows and columns now need a map, which results in the following mapping

$$I(m, n) \rightarrow P(i, j, k) = P(((m * 2) \bmod 32) + \lfloor \frac{n}{64} \rfloor, (n \bmod 64), \lfloor \frac{(m * 2)}{32} \rfloor),$$

where element I(m,n) of the image gets mapped to processor (i,j) at stack level k, 32 is the number of rows in the mesh, and 64 is the number of columns in the mesh. Under this mapping a row of the image is split into two halves of length 64, each getting mapped onto two adjacent rows in the mesh. The stack is used to serve the same purpose as before. In this case the depth of the stack is 8, making the load per processor 8 as well.

There are two things to note about the mapping used in these cases. First, as image size increases, the load per processor increases. This is not surprising, since the need for virtual processors increases with size; what is worth noting is that the load per processor is the same for all processors. Second, for each doubling of image size, starting at 64×64, there is a corresponding doubling of the distance between the processors containing the mapped values of adjacent pixels within a column of the original image. This is a cause of concern since we use the X-net to implement necessary communication between these processors and its performance degrades with the increase in distance between the communicating processors. The latter point

Figure 8: Mapping a  $64 \times 64$  MRI scan onto the mesh; the image is split into two halves each of which gets mapped to a layer of the stack.

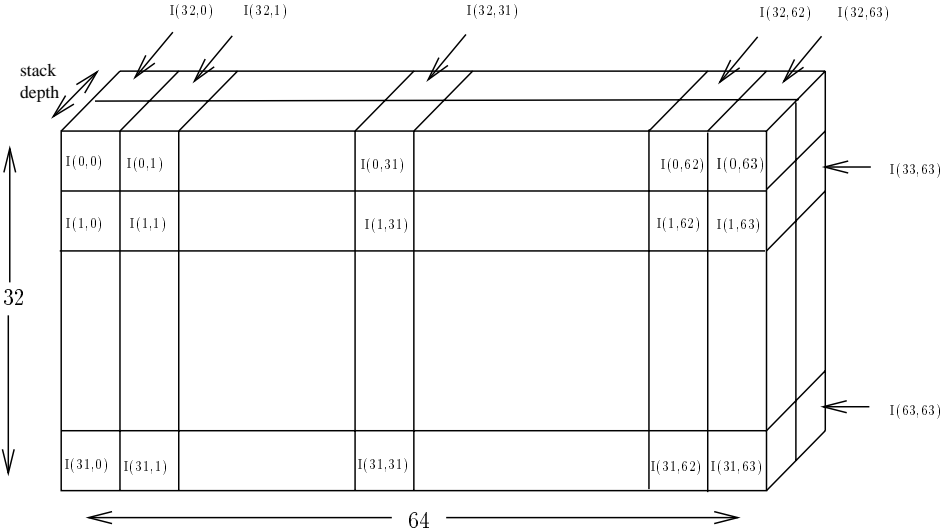


Figure 9: Processor map of a  $64 \times 64$  image.

## 6 Performance

We present representative performance results for 1-D and 2-D full wavelet packet expansions. In the 1-D case, we performed a full wavelet packet expansion of a signal of length  $2K$  using a D4 filter pair. For comparison purposes, we performed the same experiment using a sequential implementation given by Wickerhauser [9]. This code was slightly modified, compiled with gcc using level 3 optimization, and run on a Dec 3000-300LX Alpha-AXP workstation with 96 megabytes memory and 125 MHz clock. Results are given in Table 1. We give there the median timings over ten runs, both with and without the additional binary tree best basis search [9].

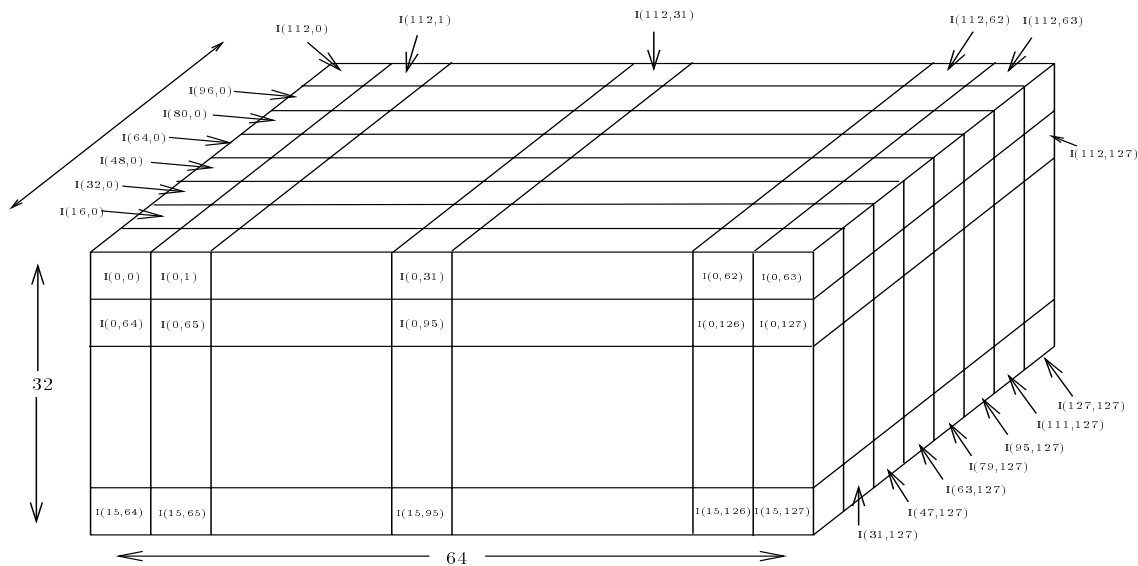


Figure 10: Processor map of a  $128 \times 128$  image

	Sequential (s)	Parallel (s)
With Search	0.198	0.019
Without Search	0.032	0.005

Table 1: Execution time for the 1-D full WPT

We now turn to the 2-D case, and present the execution times for the forward and inverse transform of a  $128 \times 128$  image. The forward transform computes the over-complete WP tree expansion, while the inverse transform acts on a particular WP basis. Two filters are used for the tests: the Haar, a 2 tap filter, and D4, a 4 tap filter.

Table 2 lists the execution time for an entire wavelet packet tree using the two filters. This time includes time spent in the router. We do not present this time as a function of tree depth, since in all applications of this transform a best basis search algorithm is run on the expansion to select a basis suited to the task at hand. A full expansion is therefore always required.

Filter	Execution Time (s)
Haar	0.067
D4	0.081

Table 2: Execution time for the parallel 2-D full WPT

We present two graphs depicting the execution time of the inverse transform, which is dependent on the choice of the WP basis. Figure 11 plots the convolution time as a function of the depth in the WP tree at which we start inverting. We see that, ignoring the time taken at  $depth = 2$ , the plot is almost linear. Moreover, the slope decreases once we get past  $depth = 4$ ; this, and the anomaly at  $depth = 2$ , is the result of our optimizing the convolution routines to the structure of the mesh. The plot also indicates that using D4, which is twice the size of the Haar and hence requires twice as much computation, takes less time than we would expect, which would be twice the time taken to compute the Haar transform.

Figure 12 plots the upsampling time as a function of tree depth. This time is independent of filter length. For the larger filter the cost is small compared to the convolution time, but this is not the case with the Haar, where the two are close. We are looking at ways to get around upsampling by trying some tricks on the MasPar.

## 7 Alternative Mappings

We explored alternative mappings for images of size 128x128, keeping in mind how these mappings would extend to larger images. In this section we present one such mapping.

In the previous mapping when the number of columns in an image exceeded the number of columns in the mesh, we split the row into pieces, the length of each piece equaling the number of columns in the mesh, and mapped the pieces to adjacent rows in the mesh. A new stack layer was created each time we wrapped around. In the new mapping, we use the stack to wrap around the rows and columns of the mesh. See Figure 13.

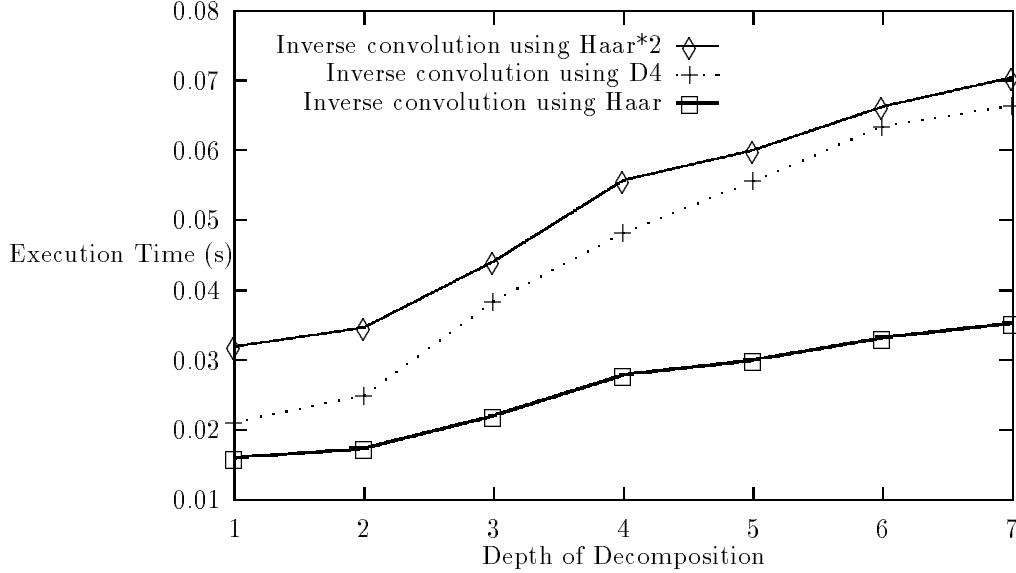


Figure 11: Convolution time in inverse transform as a function of tree depth

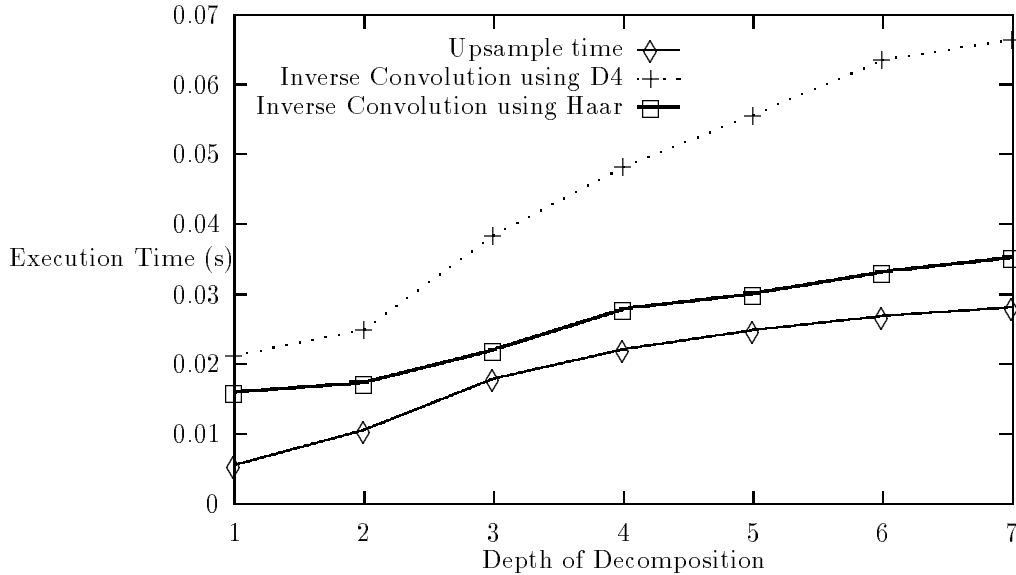


Figure 12: Upsampling time compared to convolution time in inverse transform

Under this mapping

$$I(m, n) \rightarrow P(i, j, k) = P((m \bmod 32), \lfloor \frac{n}{2} \rfloor, (\frac{m}{32}) * (\frac{n}{64}));$$

now a row of an image gets mapped to a row of the mesh.

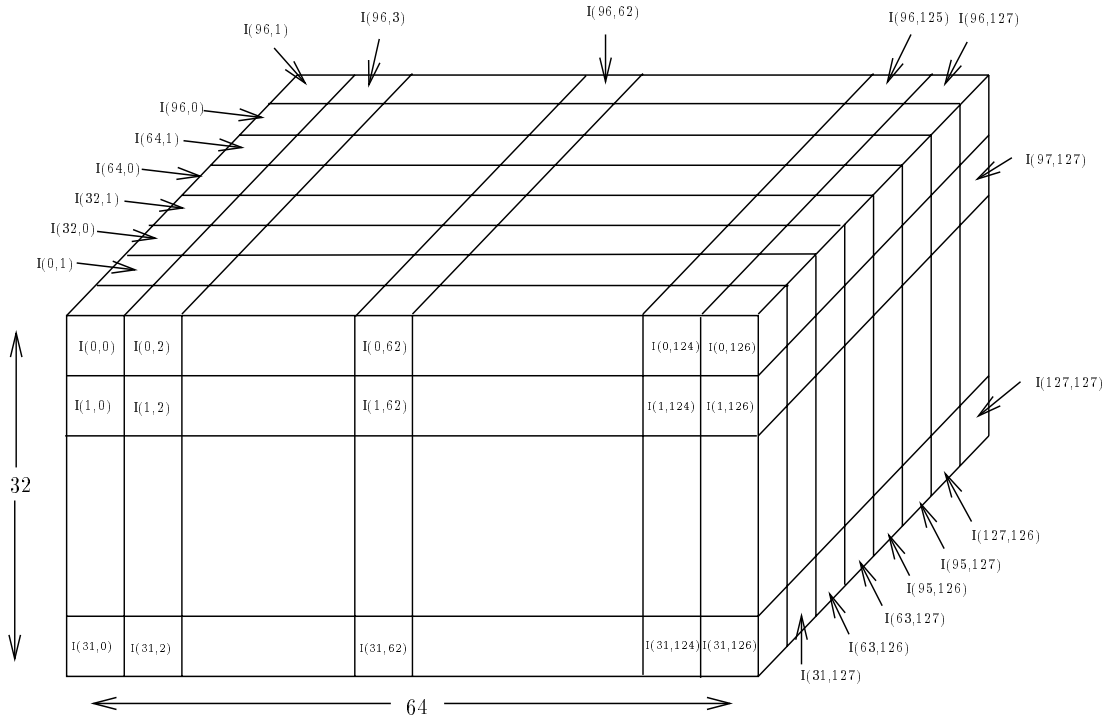


Figure 13: The new processor map for a 128x128 image

Things to note in this mapping:

1. When  $c = 64$  this mapping is identical to the previous one.
2. The load per processor is the same in either mapping.
3. Adjacent data elements within a column in the image get mapped to adjacent processors in the mesh.
4. Adjacent data elements within a row in the image do not get mapped to adjacent processors in the mesh; this proves to be advantageous since for certain adjacent pairs, no communication cost is incurred since they get mapped onto the same processor, while for other pairs the distance remains at most one.
5. This mapping is extensible. For larger sized images, the adjacent elements within a column of the image get mapped to adjacent processors in the mesh, thus keeping the step-size for the X-Net constant at 1. Also, row convolutions get cheaper as the number of columns increase since a larger number of elements get mapped to the same processor, thereby reducing communication.

We believe that this mapping will improve performance for the reasons mentioned above and will scale with image size. Portions of code written for the previous mapping are being re-used.

## 8 Conclusions

We have described an implementation of the wavelet packet transform on the MasPar MP-2. We believe that the execution times are sufficiently low that the implementation can be potentially useful for real-time MRI experiments. We are currently looking at this application in addition to the extensions proposed in the previous sections.

## Acknowledgments

Many thanks to Tom Cormen and Leonard Wisniewski for their help on optimizing the use of the router on the MasPar.

## References

- [1] Coifman, R.R. and M. V. Wickerhauser. Entropy based algorithms for best basis selection. *IEEE Trans. Inf. Theory*, 38:713-718, 1992
- [2] Cormen, T. H. and Bruhl, K., "Don't Be Too Clever: Routing BMMC Permutations on the MasPar MP-2", in *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp288-297, July 1995.
- [3] Digital Equipment Corporation, Maynard, Massachusetts, "DECmpp Programming Language (ANSI Reference Manual", December 1992.
- [4] Digital Equipment Corporation, Maynard, Massachusetts, "DECmpp Programming Language (ANSI Users Guide", December 1992.
- [5] Geist G. A. and Sunderam, V. S., "The PVM System: Supercomputer Level Concurrent Computation on a Heterogeneous Network of Workstations", in *Proceedings of the Sixth Annual Distributed Memory Computer Conference*, pp258-261, 1991.
- [6] D.M. Healy Jr., D. Warner, J. B. Weaver. "Applications of Adapted Wavelet Encoding in Functional Magnetic Resonance Imaging," in *Time-Frequency Methods in the Engineering and Biological Sciences*, M. Akay, ed. IEEE Press, New York. To appear, 1996
- [7] Montefusco, L., "Parallel Numerical Algorithms with Orthonormal Wavelet Packet Bases", in *Wavelet Analysis and its Applications, Volume 5*, pp449-453. C. Chui, L. Montefusco, L. Puccio, editors, Academic Press, 1994,
- [8] Uhl, A., "Adapted wavelet analysis on moderate parallel distributed memory MIMD architectures", in *Proceedings of Workshop on Parallel Algorithms for Irregularly Structured Problems*, pp 275-283, September 1995.
- [9] Wickerhauser, M. *Adapted Wavelet Analysis from Theory to Software* A.K. Peters, Wellesley. 1994.
- [10] Wisniewski, L. , "Efficient Design and Implementation of Permutation Algorithms on the Memory Hierarchy", Dissertation, Dartmouth College Department of Computer Science, March 1996.