Journal of Universal Computer Science, vol. 3, no. 7 (1997), 821-834
submitted: 18/2/97 , accepted:24/4/97 , appeared: 28/7/97

Stack Filter Design Using a Distributed Parallel Implementation of Genetic Algorithms

Peter E. Undrill
(University of Aberdeen, Scotland, UK
undrill@biomed.abdn.ac.uk)

Kostas Delibasis
(University of Aberdeen, Scotland, UK)

George G. Cameron
(University of Aberdeen, Scotland, UK
george@biomed.abdn.ac.uk)

Abstract: Stack filters are a class of non-linear spatial operators used for the suppression of noise in signals. In this work their design is formulated as an optimisation problem and a method that uses Genetic Algorithms (GAs) to perform the configuration is explained. Because of its computational complexity the process has been implemented as a distributed parallel GA using the Parallel Virtual Machine (PVM) software. We present the results of applying our stack filters to the restoration of magnetic resonance (MR) images corrupted with uniform, un-corellated, noise showing improved statistical performance compared with the median filter and indicating better retention of image details. The efficiency of the parallel implementation is examined, addressing both algorithmic and data decomposition, showing that execution times can be significantly reduced by distributing the task across a network of heterogeneous processors.

Key Words: Distributed Computation, Genetic Algorithms, Stack Filters, Image Processing

Categories: D3.2, G1.0, G1.6, G3, I4

1 Introduction

Stack filters are adaptive, non linear, spatial operators that can perform well in suppressing noise. Their main advantages are their ability to adapt to different types of noise and their generality. Stack filters are supersets of morphological operators and rank order filters. Their main drawback is the computational complexity of their configuration. As will be shown, this complexity, in the general case, is of the order of a double exponential.

1.1 Definitions

Initially we will describe the stacking property and introduce the idea of a positive Boolean function (pBf). A binary sequence $\vec{x}$ possesses the stacking property if:

$$x_i \geq x_j \Leftrightarrow i > j \qquad \text{Eq: 1}$$

An ordered sequence of n binary signals $\vec{x}_i$ possesses the stacking property if it is monotonically decreasing, i.e. $\vec{x}_1 \geq \vec{x}_2 \geq ....\geq \vec{x}_n$. A Boolean function *f* possesses the stacking property if, when applied to an ordered sequence of binary signals that have the stacking property, produces an output that also has the stacking property:

$$\vec{x} \geq \vec{y} \Rightarrow f(\vec{x}) \geq f(\vec{y}) \qquad \text{Eq: 2}$$

It has been shown [Gilbert 54] that a pBf possesses the stacking property if it can be expressed without complements in its input variables and [Nodes and Gallagher 82] and [Muroga 71] showed that for n samples each pBf can be uniquely expressed as a minimum sum of products (MSP) of the input variables:

$$f(\vec{x}) = \sum_{i=1}^{2^n-1} p_i m_i(\vec{x}) \qquad \text{Eq: 3}$$

where

$$m_i(\vec{x}) = \prod_{\text{some } j} x_j \qquad \text{Eq: 4}$$

and p_i is a Boolean variable indicating whether the i^{th} Boolean product contributes to the result of applying function f. The symbols of summation and product in the above equations denote the OR and AND operators respectively.

1.2 Stack Filter Application

A generalised digital filter applies a mathematical process to a collection of one or more input samples producing one or more output values. When applied to a extended sequence of input samples, the domain over which the filter is applied is defined as the region of support (or window) of the filter. The stages of applying a stack filter operator are described by [Wendt et al. 86, Maragos and Schafer 87] as:

(i) An N-point discrete signal (having integer values sampled from $\{0,1,..,M-1\}$ is mapped to the filter window $R(j)$ of width $n = 2r+1$, where r is an integer:

$$\vec{R} = (R(j-r),..,R(j),..,R(j+r)) \qquad \text{Eq: 5}$$

(ii) Each component of $\vec{R}$ can be decomposed into a binary representation $\{0,1\}$ by a threshold operator T_b:

$$T_b(R(j)) = 1 \text{ if } R(j) \geq b \text{ ; else } 0 \qquad \text{Eq: 6}$$

where b has values {1, … M} drawn from the current region of support.

Summing all the resulting decomposed binary sequences produces the original signal. This threshold decomposition is formalised into:

$$\vec{R} = \sum_{b=1}^{M-1} T_b\left(R(j)\right) \text{ for all } R(j) \text{ in } \vec{R} \qquad \text{Eq: 7}$$

(iii) Each of the n-bit binary vectors produced by the application of T_b is independently input to the Boolean function f satisfying the conditions in Eq. 2.

(iv) Finally, all the outputs of the Boolean function are summed, under the convention that true equals 1 and false equals 0, and the resultant scalar F is:

$$F = \sum_{b=1}^{M-1} f\left(T_b\left(\vec{R}\right)\right) \qquad \text{Eq: 8}$$

An example of the above algorithm is given for the very simple case of a 3×1 median filter applied to a one dimensional signal [Fig. 1], verifying that the pBf defining the median filter is of the form:

$$f = x_1x_2 + x_2x_3 + x_3x_1 \qquad \text{Eq: 9}$$

where for each threshold operation x_i is the result for a given position within the region of support of the filter, the products are ANDs and the summations are ORs.

The leftmost column represents the thresholding values and the rightmost column the result of applying the Boolean function to the threshold decomposed signals. For a threshold of 2: $x_1 = 1$, $x_2 = 0$ and $x_3 = 1$, resulting in $f = 1.0 + 0.1 + 1.1 \Rightarrow 0 + 0 + 1 \Rightarrow 1$ under the logic definitions above.

The output of the filter is the arithmetic sum of the individual outputs of the Boolean function for all the binary sequences resulting from the threshold decomposition of the input signal, i.e. (1 + 1 + 0 = 2).

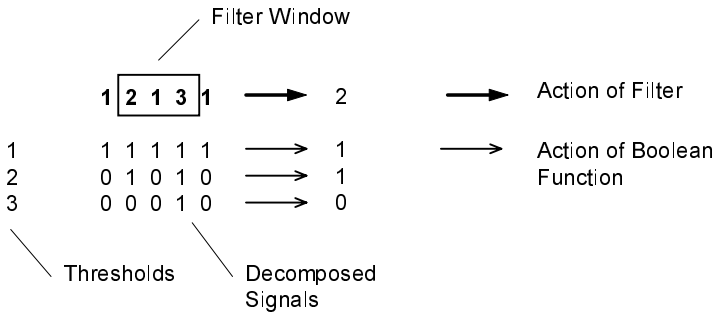

Figure 1: One dimensional median filter expressed as a stack filter sequence

There are 7 discrete combinations x_1; x_2; x_3; x_1x_2; x_1x_3; x_2x_3; x_1x_2,x_3 and any r ($1 \le r \le 7$) of these terms can be selected and combined to form a single pBF. Therefore there are 20 pBfs of 3 independent variables due to the permutation of order and grouping, 7581 pBfs of 5 variables and more than 2^{35} (and therefore stack filters) of 7 variables. For a filter of window size n pixels, [Eq. 3] suggests an upper limit of 2^{2^n-1} possible

stack filters. However a smaller number is suggested as more than one expression, derived from [Eq. 3], can be simplified to the same final function. Taking this into account, a lower limit has been derived which states that the number of stack filters S defined over a window of n pixels is bounded by:

$$2^{2^{n-1}} < S < 2^{2^n-1} \qquad \text{Eq: 10}$$

For a simple two-dimensional 3×3 filter, the number of possible stack filters lies between 2^{256} and 2^{511}.

2 The Objective Function

We will consider the problem of restoring a noise corrupted image. The objective function returns a measure of how close the restored image is to the original noise free image. Several error functions that can be used as objective functions have been suggested by [Kasturi and Walkup 6]. Here we have used the mean absolute error (MAE) [Gabbouj and Coyle 90] between the noise free image I_0 and restored image I_1 defined as:

$$MAE = \frac{1}{n} \sum_{Image} \left| I_0(i,j) - I_1(i,j) \right| \qquad \text{Eq: 11}$$

where *n* is the number of pixels over which the calculation is performed.

3 Configuration Of Stack Filters

3.1 Configuration as an Optimisation Problem

The generalised algorithm used for design is:

(i) given an ideal image and a corresponding image corrupted with a specific type of noise, select a training window,
(ii) configure a stack filter by minimising the objective function over the training window of the ideal and the filtered noisy image.

In previous image interpretation tasks [Delibasis and Undrill 94, Undrill et al. 97, Delibasis et al. 97] we have found Genetic Algorithms (GAs) to be an effective mechanism to resolve large scale optimisation problems, more time-efficient than exhaustive search, with a reported superiority [Hill and Taylor 92] over simulated annealing approaches.

GAs are adaptive methods inspired from the evolution of species which can be used in function optimisation or machine learning problems. The Darwinian theory of evolution provides an interaction between the genetic material of an organism or *individual*, called genotype, and its phenotypic expression (set of observable characteristics), called phenotype. The genotype consists of a population of

chromosomes, each of which consists of a number of *genes*. In a GA based optimisation we encode each of the variables as binary strings (genes), define an objective function which determines our goodness-of-fit and allow principles of genetic evolution, generally limited to selection of the number of offspring, gene-segment crossover and mutation to establish a new chromosome (individual) which when input to the objective function provides the best solution. The concatenation of binary strings defining a set of variables, e.g. an 8-bit string to define a variable with values 0-255, (and in this instance their interaction) is the chromosome which undergoes genetic development. The system is normally primed by a random selection of around 100 chromosomes (often constrained to produce realistic practical instances) and allowed to evolve for at least 100 generations.

3.2 Coding the Boolean Function into a Chromosome

The challenge of stack filter design is determining which pBf to use. [Eq. 3] gives a good idea of how such a coding can be performed. We first construct every possible term of the function. Each of the $2^n - 1$ terms of [Eq. 3] is a product of selected input variables (where n is the number of pixels in the window). The i^{th} term is the product of the input variables x_j for which the j^{th} bit of the i^{th} term equals one. In the median example of [Eq 9], 2^3 - 1 (7) separate combinations of the 3 input variables are possible. In the chosen function the $(i=2)^{th}$ term has $p_i = 1$ and bits 2 and 3 set to 1. Only 3 out of the possible 7 terms has p_i set to 1. Storing these factors in a chromosome is a natural way to proceed as the information is already in binary form.

3.3 Genetic Operators

The implementation of genetic operators is straightforward and very much as described in [Goldberg 89]:

(a) *Selection* determines which individuals will survive and produce offspring and which will perish. It is controlled by *fitness*, determined from the objective function, and can also determine how many offspring are produced. Typically a fitness cut-off is established and larger numbers of offspring, randomly determined, are given to the fittest individuals.

(b) *Crossover* is the where two chromosomes exchange equally sized gene segments. Several different versions of *crossover* were tested, involving one-and two-breakpoints in the chromosome as well as uniform crossover where any bit from a parent chromosome can switch to any position in the offspring with variable exchange probabilities. The results showed a significant difference in convergence speed in favour of uniform crossover, but no differences in the minimisation achieved.

(c) *Mutation* acts upon the chromosome by randomly changing the value of individual bits, with a probability chosen within a range from 0.01 to 0.001. The

encoded mask is mutated by randomly selecting a pair of different valued bits and exchanging them.

(d) Bitwise *local search*, as described in [Goldberg 89], was applied to the best chromosome, every 10 generations, after the 30^{th} generation. In this operation we focus on a single chromosome and by applying mutations see if an even better solution can be established. This is analogous to first allowing the GA to find a regional optimum, then seeing if a nearby local optimum can be found. The reason for this delayed application is that we initiate local search only after the population has converged around a minimum candidate.

3.4 The Overall Algorithm

Our implementation of a system for stack filter design using serial GAs is seen in [Fig. 2], expressed as pseudo-code.

3.5 Computational Complexity

Optimised GA based configuration of a 3×3 stack filter requires a minimum of 5000 objective function evaluations over a 50×50 pixel image training window. Each takes around 0.3 seconds on a SUN Sparcstation 10/41. For our experiments this training window represented 1/25th of the total image to be filtered. The GA has an average of 100 generations with a population of 100 chromosomes, which requires 10,000 function evaluations, hence around 1 hour will be needed for our filters to be established. Once designed the application is simple, being a sequence of thresholding operations.

```
initialise first generation randomly
while(not termination_condition) repeat for each chromosome c
    1. simplify c
    2. construct the Boolean function that c defines
    3. filter the window of the noisy image with the resulting stack filter S
    4. calculate the objective function as an image quality measure
       between the ideal image and the filtered image
    5. if(specific_condition) apply local search starting from c
    6. apply selection algorithm
    7. apply crossover and mutation to produce new_generation
```

Figure 2: Serial GA algorithm

4 Parallelisation Approach

The biological metaphor that has motivated genetic search is that in nature millions of individuals exist in parallel. As expected GAs are highly parallelisable algorithms, well

scaleable as the amount of computation increases. Since the problems attacked by GAs are non-polynomial time optimisations (NP problems) and often highly complex, execution of serial GAs can be very lengthy. The above has encouraged researchers to design parallel implementation of GAs.

Centralised Implementations [Bianchini and Brown 93] follow the philosophy of conventional serial GAs. The *master processor* performs selection and sends pairs of individuals to the *slave processors*. They concurrently apply crossover and mutation operators and calculate the fitness of each individual. The new individuals are sent back to the master processor. Part of the computation (selection) cannot be done in parallel, but for normal problems this does not impose a great overhead since the computation of fitness is much lengthier than any other operation. The main disadvantage of this implementation is the prospect of excessive communication causing a bottleneck at the master processor. Another drawback is the fact that the master processor is idle while the slaves are computing chromosomes' fitness. This can be resolved by having the master processor working on a subset of the population after sending the rest of the chromosomes to the slaves.

Distributed Implementations utilise the organisation of distributed memory architectures. A number of processors run completely independent GAs (with independent populations, operators etc.). Communication happens periodically when processors send to their neighbours their k best individuals which are then inserted into the new populations. This migration allows sub-populations to share genetic material [Whitley and Starkweather 90]. The topology defining interconnections between sub-populations is quite arbitrary.

We have examined both approaches. Allowing independent subsets of genetic material to evolve, with the pooling of the best individuals, [Delibasis 95], found that the use of multiple processors to promote the development of initially separated populations failed to produce any genetic improvement, therefore we are concerned solely with reducing the time to achieve a candidate solution. The centralised implementation approach has therefore been adopted throughout.

4.1 Algorithmic Decomposition

Selection (step 6 of the serial algorithm), and crossover and mutation (step 7) take negligible time compared with the evaluation of the fitness function which involves simplifying the chromosome (step 1), filtering the training image window (step 3) and calculating the MAE by comparing the restored and the original window (step 4). The evaluation is carried out separately for each chromosome in the population allowing parallelisation.

Our specific computing resources consist of a network of inhomogeneous UNIX workstations [Tab. 1]. The Parallel Virtual Machine[Sunderam et al. 94] (PVM) software was used as a communication harness between workstations, utilising a master program that assigns tasks to slave programs running on a single processor and limiting communication to be between slaves and master.

Model	Performance Index	No. of Systems
SLC	0.5	1
IPC	0.6	2
IPX	1.0	2
Classic LX	1.2	1
Sparc 5-70	2.6	1
Sparc 10-41	2.4	1
Sparc 10-51	3.0	2

Table 1:- Network Processor Performance Details

The GA master program performs the following:

- calculates the table of threshold decompositions for the training image window
- if (*first_generation*) create n chromosomes randomly else apply genetic operators to create the *next_generation*
- assign the n fitness evaluation tasks to the k processors and collect the results
- if not(*termination*) goto second step

The assignment algorithm, assuming n tasks and k processors, is shown in [Fig. 3].

```
assign the first k of n tasks to k slave programs
i = k
while(i <= n+k-1) do
{
   listen for result from a slave s
   if(response) despatch result
   {
      set task_index i:= i+1
      if( i<= n) assign task i to slave s
   }
}
```

Figure 3: Task assignment to slave processors

The factors that affect the efficiency of this algorithm are:

(i) the ratio of communication to useful computation performed by the slave program,
(ii) the latency of the network in despatching a message,
(iii) any load imbalance between the slave processors,
(iv) the relative proportions of parallel to serial code.

The first factor is affected by the complexity of the task and the amount of data that needs to be transferred between master and slave. The data packet transferred is the chromosome, along with the identity of the individual within the population. The data that the slaves return to the master is a real number (the value of the fitness function), the individual's identity and the simplified chromosome, therefore the communication intervals are small relative to the computation performed by the slave.

Although the objective function evaluation (the elemental process) can vary by at least a factor of 5 between the fastest and slowest systems [Tab. 1], the parallelisation is efficient in respect of load balancing (third factor) between slave processors since the number of individual tasks n (the population of chromosomes in a generation) is normally much larger than the number of slave programs k. Typical values for n and k are 40-120 and 2-10 respectively. It is this high task-to-processor ratio that allows the incorporation of processors with widely differing powers. As k approaches n the whole process is limited by the slowest slave processor.

4.2 Data Decomposition

The above approach would be sufficient to significantly accelerate a wide range of GA applications The only condition is that the genetic operators themselves take negligible time compared to the fitness function evaluations. Step 5 of the overall algorithm, local search, requires a large number of function evaluations, but the approach of algorithmic decomposition cannot be applied here since each step of the operator depends on the result of the previous step. In the general case this problem cannot be easily resolved but, as in many image processing tasks, we can use data decomposition dividing the image into n equal sections, assigning each section to a different slave. The drawback of data decomposition is that the computation performed by each slave is now much smaller, thus, if the transfer of data per task is the same, the communication overhead will increase.

We initially broadcast to every slave the chromosome to which local search will be applied [Fig. 4], and then transfer the index of the bit of the chromosome that needs to be mutated. The only other data that is transferred is a flag indicating whether the previous mutation was successful and this outcome will be retained, or discarded. At the same time a copy of the chromosome is updated. The distributed algorithm is now complete.

```
! broadcast to all slaves: chromosome, number of equally
sized image sections n !
 previous_fitness:= fitness; keep_previous:= true
 for bit = 1 to all_bits_in_chromosome do {
 ! assign the first k tasks to the k slave programs !
   i = k
     while (i <= n+k-1) do {
       ! listen for a result from a slave s ! {
           i:= i+1
           if (i <=n) then
             assign task i to slave s,
             receive(s,bit,keep_previous)
             fitness:= fitness+result } }
     if fitness better_than previous_fitness then
       keep_previous:= false else
       save recent_mutation to the local copy {
       keep_previous:= true
       discard recent_mutation  }
```

Fig 4: Parallel GA algorithm

In this way the amount of data that needs to be transferred is minimised and the main detrimental factors that remain are the network's latency and load imbalance.

5 Results And Discussion

5.1 Performance of Stack Filters

[Fig. 5] indicates detail from a transverse section MR image of the brain showing part of the cerebellum and brain stem. The original image (256×256 pixels) has been corrupted by adding uniform uncorrelated noise with 20% probability, and filtered images produced by using a 3×3 median and a 3×3 stack filter designed on a arbitrarily placed 50×50 pixel window. We choose the median since this is often the filter-of-choice where uncorrelated noise removal and image-feature retention is required.

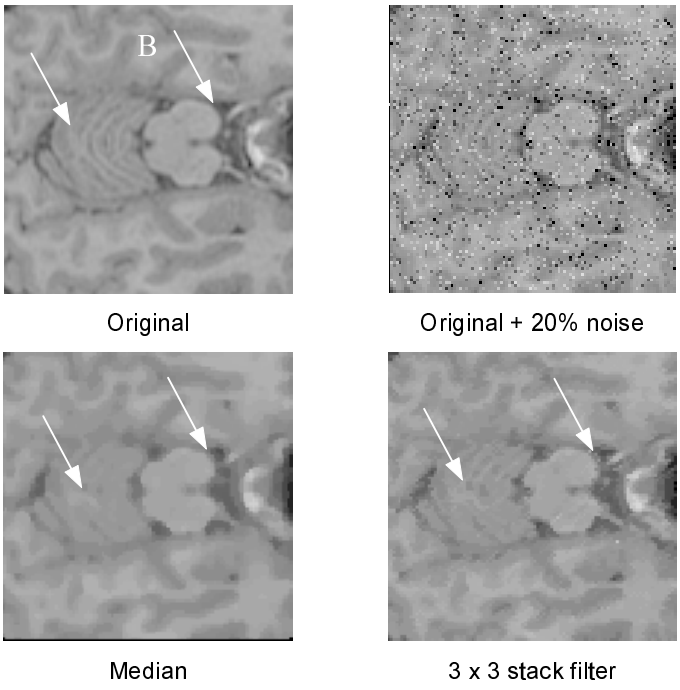

Figure 5: Stack and median filter applied to an MR image of the brain (containing parts of cerebrum, brain stem and cerebellum) corrupted with 20% noise

In [Fig. 6] we compare the performance (in terms of MAE) for different noise probabilities. Stack filters outperform the median filter at all noise levels up to 50%. Since the MAE is a global index and may not be a good indicator of local visual quality, the preservation of fine detail in restored image using a stack filter is shown at

the arrowed points in [Fig. 5]. [Fig 7] shows point B in greater detail.

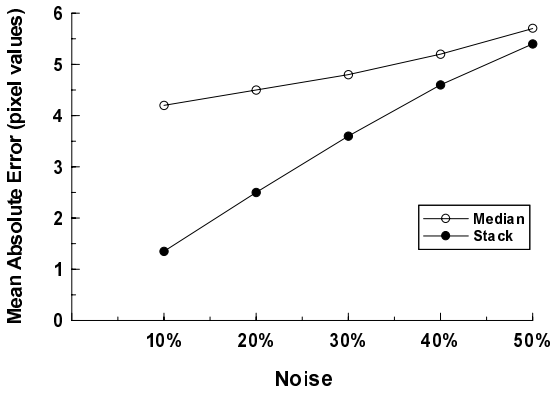

Figure 6: Variation of Mean Absolute Error (MAE) with noise level for stack and median filters (original image contains values 0 - 255)

Fig 7: Comparison of region B (figure 6) by median and stack filter

5.2 Distributing The Design Process Across A Network

We now examine the efficiency of our algorithm using distributed parallel computing. [Fig. 8] shows the speed-up factors for the algorithmic decomposition as a function of computational power. Due to the in-homogeneity of our systems, we define the distributed system's total power in units of a mid-range processor, the SUN SparcStation 10/41. Our reference point is chosen to be a serial implementation on this system.

Ideally the speedup factor should equate to the computational units applied. This is shown by the dotted line in [Fig 8]. As extra processors [Tab. 1], ranked in order of power, are introduced the speed-up factor increases uniformly, but at less than the ideal rate. The communication overhead is independent of the number of workstations

and inhomogeneity of processing power does not affect performance as long as the number of slave processors is much less than the number of tasks (the population of each generation). A gradual levelling off appears when the computational power exceeds 4 units and more slow machines are introduced followed by a down-turn at around 6 units. This is caused by a load imbalance due to the increased number of processors, as well as the influence of the non-parallelisable components of computation.

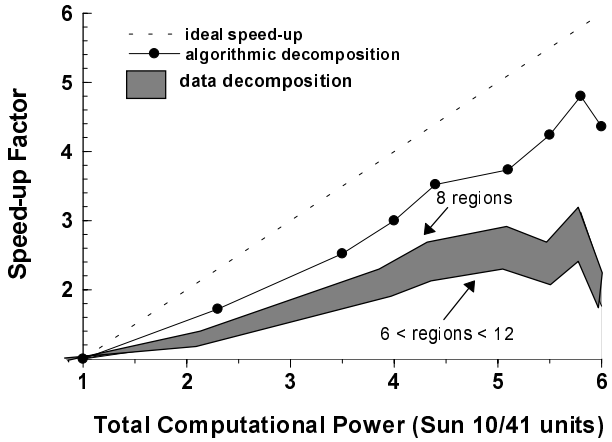

Figure 8: Speed-up performance of the parallel distributed GA

The efficiency of the data decomposition algorithm (used only by local search) is also shown in the cross-hatched section of [Fig. 8]. The initial section of the graph is dominated by the relatively high communication overhead of introducing additional processors. As more processors are added, their inhomogeneity gives rise to load imbalance (in other words the faster computers wait for the slower) and this factor becomes dominant. The speed-up factor reaches a plateau and then decreases. The detailed performance of data decomposition will depend on the number of image regions chosen. A maximum speedup exists when the image is partitioned into 8 equal regions, decreases when fewer are used (greater load imbalance across processors) and also decreases for more than 8 regions as communication overheads become dominant. This is shown by the shaded envelope in [Fig 8] where the upper bound represents 8 equal image regions and the lower bound describes the performance for 12 and 6 regions as explained above.

The algorithm we use has each form of decomposition, in proportions dependent on the GA formulation, therefore the final speed-up [Fig. 8] will lie between the upper algorithmic decomposition curve and the data decomposition curve appropriate to the chosen number of image regions. Since the GA local search is introduced only after a defined period and at intervals thereafter, the final outcome will be dominated by algorithmic decomposition. In our workstation configuration the typical time required for the design of a filter is reduced from 1 hour to around 15 minutes.

6 Conclusions

The design of stack filters for noise suppression using genetic algorithms is described and it is shown that the amount of computation required to configure even small filters is substantial. Since GAs provide a robust method of filter configuration, capable of incorporating arbitrarily complex objective functions, we have developed a system that allows a heterogeneous cluster of workstations, operating in parallel, to achieve this process in acceptable time. Performance improvement factors of up to 4, using 10 systems, are observed. Results are presented for a practical example drawn from medical imaging showing that the stack filter has an improved performance compared to the median filter. The computing network we have used is typical of those used in academic and research communities.

7 References

[Gilbert 54] Gilbert E.: "Lattice-theoretic properties of frontal switching functions", Journal of Mathematical Physics; 33, (1954) 57 - 67.

[Nodes and Gallagher 82] Nodes T. and Gallagher N.: "Median filters: some modifications and their properties"; IEEE Transactions on Acoustics, Speech and Signal Processing, ASSP-30, (1982) 739 - 746.

[Muroga 71] Muroga S.: "Threshold Logic and its Applications"; New York:, Wiley (1971).

[Wendt et al. 86] Wendt P., Coyle E. and Gallagher N.: "Stack filters"; IEEE Transactions on Acoustics, Speech and Signal Processing, ASSP-34, (1986) 11.

[Maragos and Schafer 87] Maragos P. and Schafer W.: "Morphological filters: Part I - Their set theoretic analysis and relation to linear shift invariant filters"; IEEE Transactions on Acoustics, Speech and Signal Processing, ASSP-35, (1987) 1153 - 1169.

[Kasturi and Walkup 86] Kasturi R. and Walkup J.: "Non-linear image restoration in signal dependent noise"; In Advances in Computer Vision and Image Processing', (Ed. Huang T.), Vol. 2, Greenwich, Conn. (1986)

[Gabbouj and Coyle 90] Gabbouj M. and Coyle E.: "Minimum mean absolute error stack filtering with structural constraints and goals"; IEEE Transactions on Acoustics, Speech and Signal Processing, ASSP-38, (1990) 955 - 968.

[Delibasis and Undrill 94] Delibasis K. and Undrill P.: "Anatomical object recognition using deformable geometric models"; Image and Vision Computing, 12, 7, (1994) 423 - 433.

[Undrill et al. 97] Undrill P., Delibasis K. and Cameron G.: "An application of genetic algorithms to geometric model guided interpretation of brain anatomy"; Pattern Recognition, 30, 2 (1997) 217 - 227.

[Delibasis et al. 97] Delibasis K., Undrill P. and Cameron G.: "Designing Texture Filters with Genetic Algorithms : an application to medical images"; Signal Processing, 57, 1 (1997).

[Hill and Taylor 92] Hill A. and Taylor C.: "Model-based image interpretation using genetic algorithms"; Image and Vision Computing, 10, (1992) 295 - 300.

[Goldberg 89] Goldberg D.: "Genetic Algorithms in optimization, search and machine learning"; Addison-Wesley (1989).

[Bianchini and Brown 93] Bianchini R. and Brown C.: "Parallel genetic algorithms on distributed memory architectures"; Technical Report 436, 1993, University of Rochester.

[Whitley and Starkweather 90] Whitley D. and Strakweather T.: "Genitor: a distributed genetic algorithm", Journal of Experimental and Theoretical Artificial Intelligence, 2, (1990) 189 - 214.

[Delibasis 95] Delibasis K.: "Genetic algorithms for medical image analysis"; PhD Thesis, University of Aberdeen (1995).

[Sunderam et al. 94] Sunderam V., Geist G. ,Dongarra J. and Manchek R.: "The PVM concurrent computing system"; Parallel Computing, 20, 4, (1994) 481 - 496.