METHODS FOR SIGNAL FILTERING AND MODELLING AND THEIR PARALLEL DISTRIBUTED COMPUTING IMPLEMENTATION

A Thesis

by

XIAOKUN ZHU

Submitted for the degree of

DOCTOR OF PHILOSOPHY

August 1994

University of Glasgow

Department of Statistics

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

ABSTRACT

Methods for Signal Filtering and Modelling and Their Parallel Distributed Computing Implementation. (August 1994)

Xiaokun Zhu, University of Glasgow

Supervisor: Professor D M Titterington

In this thesis the problem of filtering and modelling one-dimensional discrete signals and implementation of corresponding parallel distributed algorithms will be addressed.

In Chapter 2, the research areas of parallel distributed computing environments, rank-based nonlinear filter and fractal functions are reviewed.

In Chapter 3, an Interactive Parallel Distributed Computing Environment (IPDCE) is implemented based on Parallel Virtual Machine (PVM) and an interactive application development tool, the Tcl language. The approach we use is to provide a Tcl version interface for all procedures of the PVM interface library so that users can utilize any PVM procedure to do their parallel computing interactively.

In Chapter 4, an interactive parallel stack-filtering system is implemented, based on the IPDCE. The user can play with this filtering system in both traditional command mode and modern Graphics User Interface (GUI) mode. In order to reduce the time required to compute a standard stack filter, a new minimum threshold decomposition scheme is introduced and other techniques such as minimizing the number of logical operations and utilizing the CPU bit-fields parallel property are also suggested. In this filtering system the user can select sequential or parallel stack-filtering algorithms. The parallel distributed stack-filtering algorithm is implemented with equal task partitioning and PVM. Two numerical simulations show that the interactive parallel stack-filtering system is efficient for both the sequential and the parallel

filtering algorithms.

In Chapter 5, an extended Iterated Function System (IFS) interpolation method is introduced for modelling a given discrete signal. In order to get the solution of the inverse IFS problem in reasonable time, a suboptimal search algorithm, which estimates first the local self-affine region and then the map parameters is suggested, and the neighbourhood information of a self-affine region is used for enhancing the robustness of this suboptimal algorithm. The parallel distributed version of the inverse IFS algorithm is implemented with equal task partitioning and using a Remote Procedure Call application programming interface library. The numerical simulation results show that the IFS approach achieves a higher signal to noise ratio than does an existing approach based on autoregressive modelling for self-affine and approximately self-affine one-dimensional signals and, when the number of computers is small, the speed-up ratio is almost linear.

In Chapter 6, inverse IFS interpolation is introduced to model self-affine and approximately self-affine one-dimensional signals corrupted by Gaussian noise. Local cross-validation is applied for compromising between the degree of smoothness and fidelity to the data. The parallel distributed version of the inverse algorithm is implemented in Parallel Virtual Machine (PVM) with static optimal task partitioning. A simple computing model is applied which partitions tasks based on only each computer's capability. Several numerical simulation results show that the new IFS inverse algorithm achieves a higher signal to noise ratio than does existing autoregressive modelling for noisy self-affine or approximately self-affine signals. There is little machine idle time relative to computing time in the optimal task partition mode.

In Chapter 7, local IFS interpolation, which realises the IFS limit for self-affine data, is applied to model non self-affine signals. It is difficult, however, to explore the whole parameter space to achieve globally optimal parameter estimation. A two-stage search scheme is suggested to estimate the self-affine region and the associated region parameters so that a suboptimal solution can be obtained in reasonable time.

In the first stage, we calculate the self-affine region under the condition that the associated region length is twice that of the self-affine region. Then the second stage calculates the associated region for each self-affine region using a full search space. In order to combat the performance degradation caused by the the difference of machines capabilities and unpredictable external loads, a dynamic load-balance technique based on a data parallelism scheme is applied in the parallel distributed version of the inverse local IFS algorithm. Some numerical simulations show that our inverse local IFS algorithm works efficiently for several types of one-dimensional signal, and the parallel version with dynamic load balance can automatically ensure that each machine is busy with computing and with low idle time.

To My Parents

ACKNOWLEDGMENTS

I am deeply indebted to my supervisor Professor D.M. Titterington for his invaluable guidance, encouragement and help throughout this work.

I would like to specially thank Dr. B. Cheng and Dr W. Qian for their collaboration, assistance and discussion.

I am grateful to Mr. D Mackay for his generous helping with computer equipments.

The author also wishes to acknowledge financial support from Glasgow University and also partly from the U.K. Government Awards.

Finally, I would like to express my sincerest thanks to my wife, my son, my mother, my father, my mother-in-law and my father-in-law for their love, encouragement and boundless patience over all these years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1. Motivation

In application of data analysis, filtering and modelling are basic and important procedures. As we know, data from the real world include noise which consists of system error and measurement error. The aims of data analysis are to understand the current data received and to use this information to predict the action of future data. Figure 1 illustrates the procedure of data analysis. In the first stage, noisy input data is passed into a filtering block and the noise is smoothed. Then, in the second stage, the filtered data is passed into a modelling block and the model parameters are estimated. We can use these model parameters to predict new data.

In linear filter design, there is simplicity and unifying linear systems theory makes their design and implementation easy. For Gaussian noise the linear filter is optimal, but linear techniques fail if the noise is non-Gaussian, examples of this are impulsive noise, signal dependent noise and nonlinear data degradation. Special linear filters, which were originally used in image filtering applications, cannot cope with nonlinearities of image formation model and cannot take into account the nonlinearities of human vision. As we know human vision is very sensitive to high-frequency information and image edges and image details such as corners and lines, which carry very important information for visual perception, have high-frequency content. Most of the classical linear filters have low-pass characteristics and they tend to blur edges and to destroy lines, edges and other fine image details. These reasons have led researchers, to the use of nonlinear filtering techniques.

Nonlinear filtering techniques emerged at very early stage. However, the bulk of related research has been presented in the past decade. This research area has had a dynamic development. This is indicated by the amount of research presently published and the popularity and widespread use of nonlinear digital processing in a

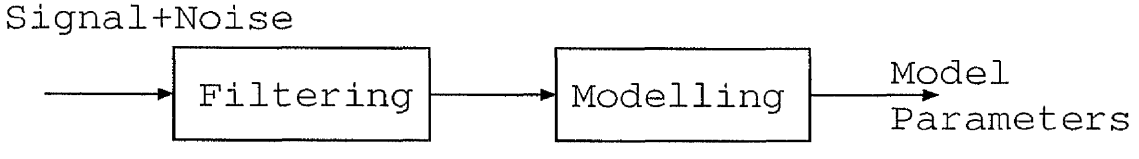

Fig. 1. Data analysis

variety of applications. There are several classes of nonlinear digital signal and image processing techniques [158]:

1. order statistic filters and stack filters;
2. homomorphic filters;
3. polynomial filters;
4. mathematical morphology;
5. neural networks;
6. nonlinear image restoration.

Each class of nonlinear processing technique possesses its own mathematical tools that can provide reasonably good analysis of its performance, but there is not a unifying theory that can encompass all existing nonlinear filters. Recently, mathematical morphology and order statistic filters have been efficiently integrated into one class based on threshold decomposition, although they come from completely different origins. We shall investigate stack filter design in this thesis. The basic tools of the stack filter are threshold decomposition and stacking, which reduce the problem of filtering P-value data to that of filtering binary data and the binary filtering problem is fairly well understood.

There are two traditional methods for modelling discrete signals. One uses polynomial fits and represents the discrete signal by the values of a polynomial evaluated at the sample point. The model parameters are the order of the polynomial, which is usually determined a *priori*, and the coefficients of the polynomial, which are usually estimated in terms of least-squares fit to the given signal values. The other involves fitting an autoregressive moving-average (ARMA) model [41], in which the model parameters are the coefficients of a filter for which the input is white noise and the output is the given signal. However, some signals are self-similar (self-affine) in nature and the basic property of fractal models is that of self-similarity (self-affine) or scale invariance. The best way to model such signals is to use a fractal model: many natural shapes such as coastlines, mountains and clouds are easily described by fractal models.

The terminology *fractal* was first used by the French mathematician Benoit Mandelbrot to describe shapes with fractional dimensions (Latin fractus meaning irregular) [127]. Mandelbrot's fractal geometry provides both a description and a mathematical model for many of the seemingly complex forms and patterns in nature and the sciences. Fractals have blossomed enormously in the past few years and have helped reconnect pure mathematics research with both natural sciences and computing science. Classical geometry provides a first approximation to the structure

of physical objects; it is the language which we use to communicate the designs of technological products and very approximate forms of natural creations. Fractal geometry is an extension of classical geometry. It can be used to make precise models of physical structures from ferns to galaxies. Fractal geometry is a new language. Once you can speak it, you can describe the shape of a cloud as precisely as an architect can describe a house [23]. There are also two fractal approaches to modelling one-dimensional signals. The first is to use fractional Brownian motion (FBM) [127]. However, fractional Brownian motion is defined in a one-dimensional framework and it is very difficult to generalize it to high dimensions. The second way is to use the iterated function systems (IFS) developed by Barnsley and his collaborators. IFS theory has many advantages over FBM: IFS modelling has higher flexibility than FBM modelling; generalization from one dimension to higher dimensions is very natural and easy. We shall apply IFS theory to model one-dimensional signals in this thesis.

From the time when the first generation of computers in the 1950s used electronic valves as their switch components, the computer has been the most basic and powerful tools in data analysis. High-performance computers are increasingly in demand in the areas of structural analysis, weather forecasting, petroleum exploration, fusion energy research, medical diagnosis, aerodynamics simulation, remote sensing, multimedia data processing and communication, military defence, genetic engineering and socioeconomics. Without superpower computers, many of these challenges to advanced human civilization cannot be made within a reasonable time period. The designers always strive to increase the speed of operations. There is a number of possible ways to achieve this. An obvious approach is to improve the technology implemented in the realization of the computer components. The current technology has gone a long way in this direction from the vacuum tube, discrete diodes and transistors, small- and medium-scale integrated (SSI/MSL) devices, to large- and very-large-scale integrated (LSI/VLSI) system, and , the development will continue. There is of course a natural limitation in technology development; no signal can propagate faster than the speed of light. Another approach is to refine the logic design of computer subsystems to achieve higher speed, for instance, to use Carry Look Ahead (CLA) in addition, or the Booth Algorithm for multiplication[92]. Improving algorithms to solve various classes of problems will also lead to higher speed of operations.

There is, however, yet another way of increasing the speed of computation: by performing as many operations as possible simultaneously, concurrently, in parallel, instead of sequentially. In the traditional Von Neumann architecture digital computer [4, 48, 92, 97, 115, 165, 176] operations are performed on a sequential basis.

The CPU fetches an instruction from the memory, decodes it into its registers, fetches operands (if any), executes the operation, and the result is sent from its register to be stored in its memory, in that order. None of these operations is started until the preceding one is completed. A new instruction is fetched only after the execution of previous one is accomplished. There is no time-overlap in the execution of any of elementary operations in the instruction cycle. Each CPU contains just one Arithmetic Logic Unit (ALU), which would perform all of the data processing tasks of the system.

The earliest reference to parallelism in computer design is thought to be in General L F Menabrea's publication in the Bibliothequè Universelle de Genève, October 1842, entitled 'Sketch of the Analytical Engine Invented by Charles Babbage' [111, 139]. In listing the utilities of the analytical engine, he writes:

> Secondly, the economy of time: to convince ourselves of this, we need only recollect that the multiplication of two numbers, consisting each of twenty figures, requires at the very utmost three minutes. Likewise, when a long series of identical computations is to be performed, such as those required for the formation of numerical tables, the machine can be brought into play so as to give several results at the same time, which will greatly abridge the whole amount of the processes.

It does not appear that this ability to perform parallel operation was included in the final design of Babbage's calculating engine; however, it is clear that the idea of using parallelism to improve the performance of a machine had occurred to Babbage over 100 years before technology had advanced to the state that made its implementation possible.

Recently, the major development affecting scientific problem-solving is that of parallel distributed computing. Many scientists are discovering that their computational requirements are best served not by a single, monolithic machine but by a variety of distributed computing resources, linked by high-speed networks. We shall implement our parallel algorithms on this type of parallel computing environment.

1.2. Outline of the thesis

In this thesis, we shall address the three research areas of rank-based nonlinear filters, iterated function system based one-dimensional signal models and parallel distributed algorithm implementation and application.

In Chapter 2, we introduce the advantage of parallel distributed computing relative to traditional parallel computing and compare several popular parallel distributed

computing environments and their point-to-point communication speed. For nonlinear filters, we shall review the basic median-based , and rank-based filters and their extension, namely, stack filters. For fractal models, we shall review the method of constructing a fractal and the approach of fitting a given signal with a fractal model. We also introduce the image compression technique using fractal transform.

In Chapter 3, we introduce the popular parallel distributed computing environment, Parallel Virtual Machine and the interactive application developing tool, Tcl language. We design and implement an interactive parallel distributed computing environment (IPDCE) based on PVM and Tcl language.

In Chapter 4, we present a new minimum threshold decomposition scheme for implementation of a stack filter. In order to reduce the performance time of standard stack filtering we try to minimize the number of logical operations and utilize the CPU bit-fields parallel property. We implement an interactive stack filtering system based on IPDCE, in which we can use traditional command line mode and modern graphics user interface to set filter parameters and select sequential or parallel filtering algorithms.

In Chapter 5, we present an extended Iterated Function System (IFS) interpolation method for modelling a given discrete signal. We suggest a suboptimal search algorithm with robust technique for estimating the map parameters so that we can get a solution in reasonable time. We also implement a parallel distributed version of this inverse algorithm using equal task partitioning and a Remote Procedure Call application programming interface library.

In Chapter 6, we use the robust IFS inverse algorithm with a local cross-validation technique to model the self-affine and approximately self-affine noisy signal corrupted by Gaussian noise. We also implement the parallel distributed version of this inverse algorithm in Parallel Virtual Machine (PVM) with static optimal task partitioning.

In Chapter 7, we apply local IFS, which realises the limit for self-affine data, to model general signals. We suggest a two-stage search scheme to estimate the self-affine region and associated region parameters so that we can get a suboptimal solution in reasonable time. In order to solve the problem of performance degradation caused by the difference of machines capabilities and external loads, we implement a dynamic load balance technique based on a data parallelism scheme.

In Chapter 8, we present the main results and conclusions of this thesis and make suggestions for some further research.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1. Introduction

The past several years have witnessed an ever-increasing acceptance and adoption of parallel distributed computing. In this chapter we review the progress of this research area and compare the key factor, communication speeds, of some popular parallel distributed computing environment. Stack filters are a new general class of nonlinear filters, which includes many particular nonlinear filters such as median-type, order statistics-type and morphological filters. We introduce stack filters' two basic properties, threshold decomposition and the stacking property and we mention ways of extending standard stack filters. Data modelling is the other important research area which this thesis will involve. We present some background knowledge of a new approach, that of fractal-based Iterated Function Systems.

2.2. Parallel Processing and Parallel Distributed Computing

2.2.1. Parallel Processing

First, we give the definition of parallel processing.

Definition 2.1 *Parallel computing [97] is an efficient form of information processing which emphasizes the exploitation of concurrent events in the computing process. Concurrency implies parallelism, simultaneity, and pipelining. Parallel events may occur in multiple resources during the same time interval; simultaneous events may occur at the same time instant; and pipelined events may occur in overlapped time spans. These concurrent events are attainable in a computer system at various processing levels.*

In theory, the speedup that can be achieved by a parallel computer with n identical processors working concurrently on a single problem is at most n times faster than a single processor. In practice, the speedup is much less, since some processors are idle at a given time because of conflicts over memory access or communication paths, inefficient algorithms for exploiting the natural concurrency in the computing problem, or many other reasons. The lower-bound $\log_2 n$ is known as Minsky's conjecture. A more optimistic speedup estimate is upper bounded by $\frac{n}{\ln n}$ as derived below [97].

Consider a computing problem, which can be executed by a uniprocessor in unit time, $T_1 = 1$. Let f_i be the probability of assigning the same problem to the ith processor working equally with an average load $d_i = 1/i$ per processor. Furthermore, assume equal probability of each operating mode using processor i, that is $f_i = 1/n$, for n-operating modes: $i = 1, 2, \cdots, n$. The average time required to solve the problem on an n-processor system is given below, where the summation represents n operating modes.

$$T_n = \sum_{i=1}^{n} f_i \cdot d_i = \frac{\sum_{i=1}^{n} \frac{1}{i}}{n} \tag{2.1}$$

The average speedup S is obtained as the ratio of $T_1 = 1$ to T_n; that is,

$$S = \frac{T_1}{T_n} = \frac{n}{\sum_{i=1}^{n} \frac{1}{n}} \leq \frac{n}{\ln n} \tag{2.2}$$

Hockney and Jesshope [92] suggested a structure taxonomy involving sequential computers, parallel computers and multicomputer systems. A taxonomy for MIMD computers [92, 178] is given in the Figure 2 taken from [92].

2.2.2. Distributed Parallel Computing

Two developments [56] promise to revolutionize scientific problem solving. The first is the development of massively parallel computers. Massively parallel systems offer the enormous computational power needed for solving grand challenge problems. Unfortunately, software development has not kept pace with hardware advances. In order to exploit fully the power of these massively parallel systems, new programming paradigms, languages, scheduling and partitioning techniques, and algorithms are needed.

The second major development affecting scientific problem solving is that of parallel distributed computing. Many scientists are discovering that their computational requirements are best served not by a single, monolithic machine but by a variety of distributed computing resources, linked by high-speed networks.

Parallel Distributed Computing, also called heterogeneous concurrent computing [68, 174], is gaining increasing acceptance as an alternative or complementary paradigm to multiprocessor-based parallel processing as well as to conventional supercomputing. While algorithmic and programming aspects of heterogeneous concurrent computing are similar to their parallel processing counterparts, system issues, partitioning and performance aspects are significantly different.

Definition 2.2 *The term parallel distributed computing, also called heterogeneous concurrent computing, refers to the simultaneous execution of the components of a*

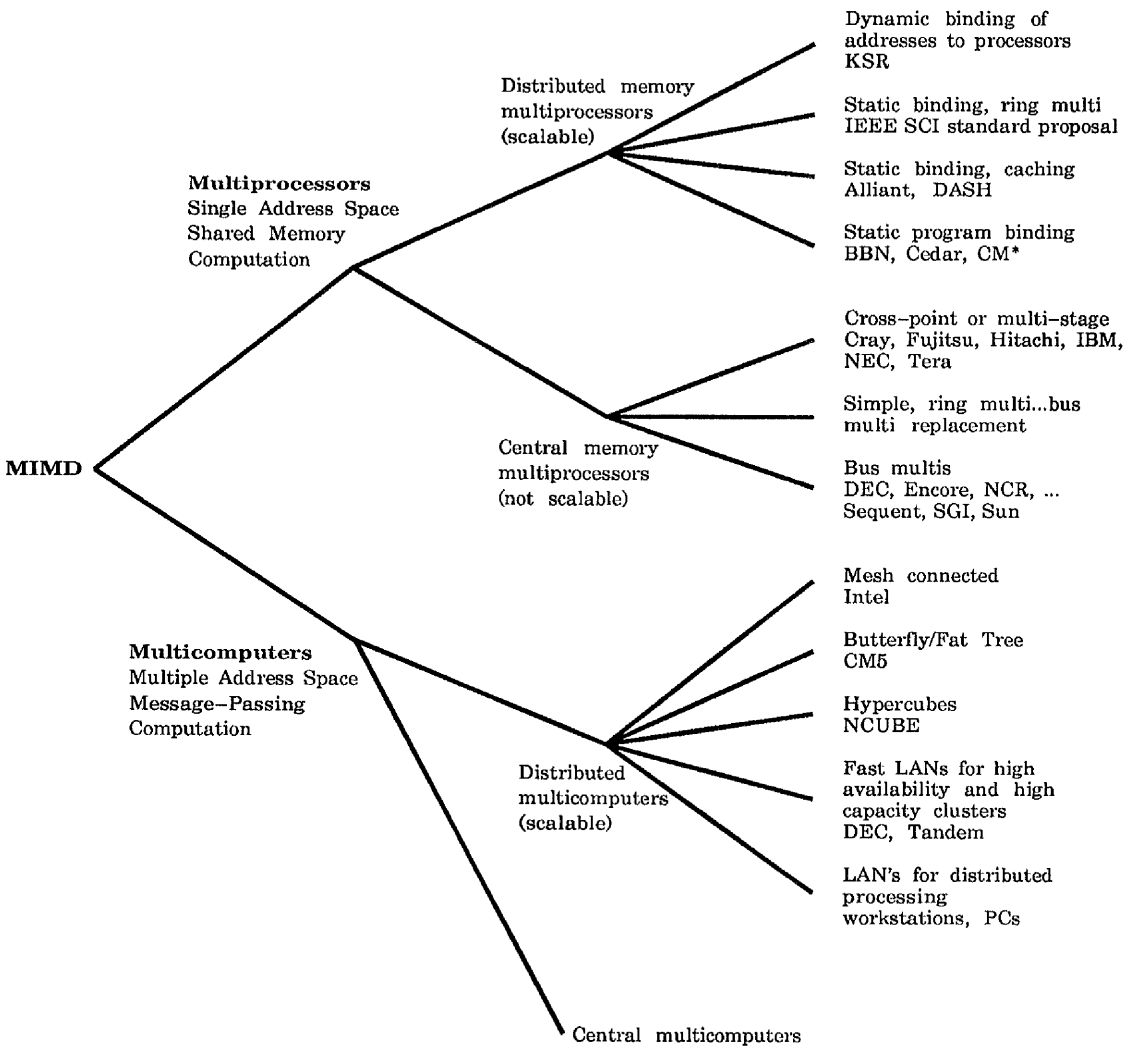

Fig. 2. Taxonomy of MIMD Computers[92].

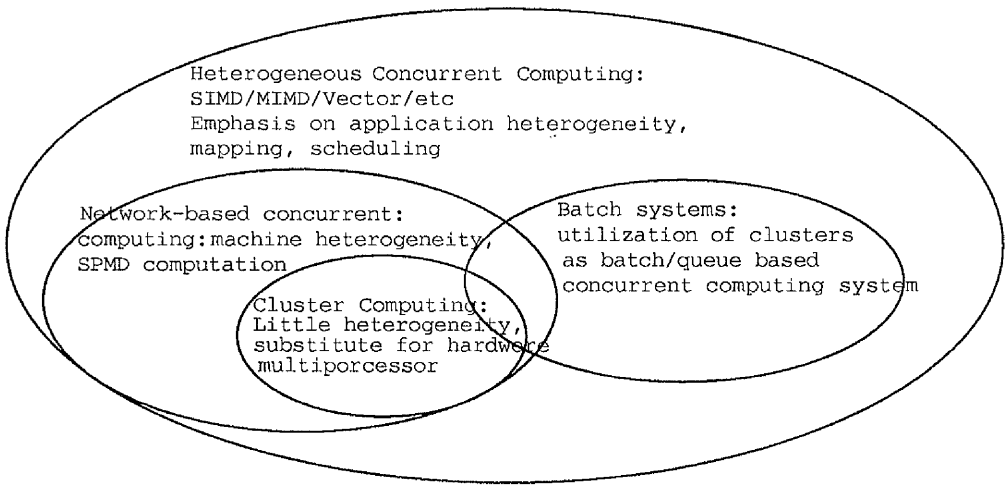

Fig. 3. Heterogeneous, Network, and Cluster Concurrent Computing[174]

single application on multiple processing elements which are loosely coupled, physically and logically independent, and heterogeneous.

These characteristics distinguish heterogeneous concurrent computing from traditional parallel processing, normally performed on homogeneous, tightly coupled platforms which possess some degree of physical independence but which are logically coherent.

It is worthwhile to note [174] that parallel distributed computing is a superset of similar methodologies referred to as network computing and cluster computing. While the nomenclature is as yet informal, network computing may be considered equivalent to heterogeneous computing, but with rather less emphasis on application heterogeneity, mapping, and task partitioning aspects. Cluster computing is even more restrictive, in that it generally refers to usually identical workstation clusters that are used as a substitute for hardware multiprocessors. Figure 3 depicts the relationship between various concurrent computing paradigms.

2.2.3. Evaluation of Network and Other Hardware Sources

During the last decade there has been an exponential growth in networked computing resources. This fact is reflected by the growth in registered systems connected to the Internet. The most recent status report from the Network Information Systems Center [121] summarizes this growth (see Figure 4). Over 725,000 hosts have been connected via approximately 17,000 domains in just ten years! The rapid growth of networked computers has been accompanied by an astonishing increase of computational power by these network attached computers. Microprocessors have doubled in

performance approximately every eighteen months during the last decade and they continue to increase in performance at a much greater rate than supercomputers (see Figure 5) [30]. Cheong [45] summarized the five key technology areas which drive high performance scientific computing: microprocessors, networks, backplane buses, semiconductor main memory, and magnetic fixed disk. Cheong provided the following appraisal of each technology.

- Since 1985 the performance of CMOS-based microprocessors has quadrupled every three years, or at the rate of 60% every year. Clock speeds alone have evolved from 200 kHz in 1971 to 50 MHz in 1991.
- Local area networks have improved by a factor of 10 every decade. In 1980 Ethernet operated at 10 M-bits/sec. In 1990 FDDI operated at 100 M-bits/sec. Early prototypes indicate that G-bits/sec networks will be commercially available by 2000.
- Computer backplane buses have improved by a factor of 10 every decade. Digital's Unibus operated at 2 M-bits/sec in 1970. Motorola's VME bus operated at 20 M-bits/sec in 1980. In 1990 several buses operated at 200 M-bits/sec.
- Semiconductor memory chips have quadrupled in capacity every three years (annual rate of 60%) since 1972. The chronology on the number of bits per chip follows. 1K (1972), 4K (1975), 16K (1978), 64K (1981), 256K (1984), 1M (1987), 4M (1990).
- Magnetic disk storage has evolved from a density of 1K bits per square inch (1957) to 1G bits per square inch in 1990 (annual rate of 26% per year, or doubling every three years).

This combined performance growth indicates that significant computational capability is available and interconnected.

Some of the infrastructure requirements of heterogeneous concurrent computing are listed below [85, 112, 144, 178].

- High bandwidth networks to support communications requirements (e.g. 100-800 M-bits/sec per host).
- Low latency communication mechanisms (e.g. 100-500 microsecond between hosts).
- Good scaling characteristics (e.g. 10-1000 hosts).
- Support for high-bandwidth multi-cast communications.

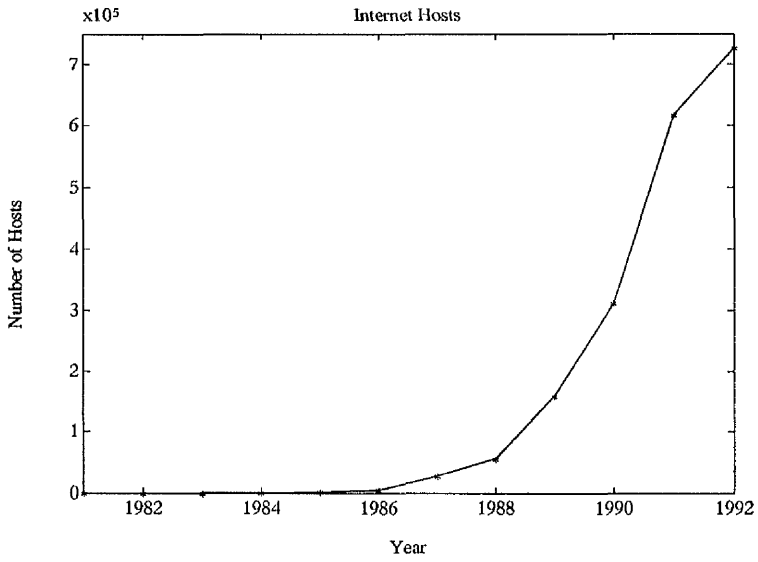

Fig. 4. Internet Host Growth in Last Decade[121].

- Capability to recover automatically from network and node failures (e.g. fault tolerant).
- Standard low-level primitives for communications, synchronization, and scheduling across architectures.
- Heterogeneous remote procedure calls that hide architecture, protocol and system differences.
- Real-time performance monitors.
- Reliable production batch job scheduler.
- Distributed application development tools.
- Support for traditional high level languages for heterogeneous computing.
- Applications which are capable of exploiting workstation clusters.
- New system administration tools to address system management issues for distributed computing resources.
- Development of standards which protect software investments.

2.2.4. The Advantages and Limitations of Parallel Distributed Computing

Parallel distributed computing offers several advantages: [56] By using existing hardware the cost of this computing can be very low. Performance can be optimized

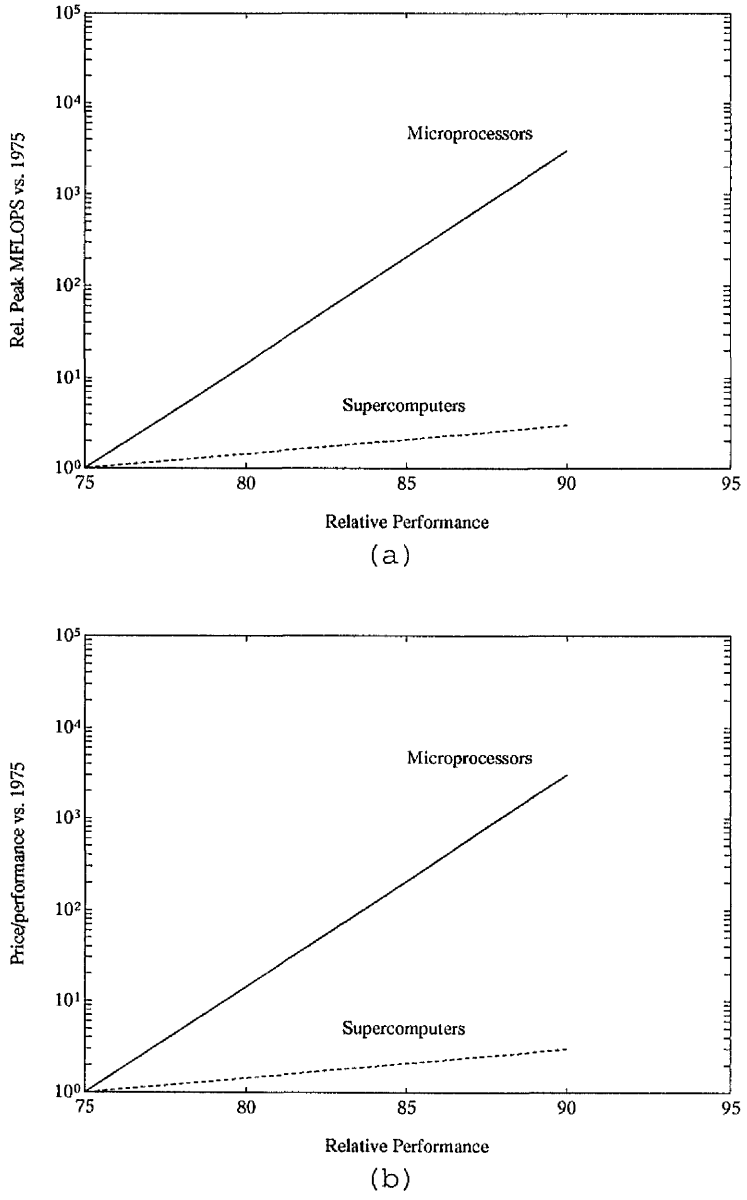

Fig. 5. Improvement of Microprocessors vs. Supercomputers[121]

by assigning each individual task to the most appropriate architecture. Parallel distributed computing also offers the potential for partitioning a computing task along lines of service functions. Typically, parallel distributed computing environments possess a variety of capabilities; the ability to execute subtasks of a computation on the processor most suited to a particular function both enhances performance and utilization. Another advantage in network-based concurrent computing is the ready availability of development and debugging tools, and the potential fault tolerance of the network and the processing elements. Typically, systems that operate on loosely coupled networks permit the direct use of editors, compilers, and debuggers that are available on individual machines. These individual machines are quite stable, and substantial expertise in their use is readily available. These factors translate into reduced development and debugging time and effort for the user, and reduced contention for resources and possibly more effective implementations of the application. Yet another attractive feature of loosely coupled computing environments is the potential for user-level or program-level fault tolerance that can be implemented with little effort either in the application or in the underlying operating system. Most multiprocessors do not support such a facility; hardware or software failures in one of the processing elements often lead to a complete crash.

One of the obvious limitations of clusters [178] is created by the relatively slow network interconnection hardware. The interface employed will depend on the bandwidth requirements, latency requirements, distance limitations and budget constraints. Ethernet is the most commonly implemented network and transmits information at 10 M-bits/sec. Many dedicated clusters are interconnected by more expensive technologies to overcome the limitations induced by the speed of Ethernet. The most common alternatives to Ethernet are Fiber Distributed Data Interface (FDDI) and IBM's Serial Optical Channel Converter (SOCC).

2.2.5. Several Popular Parallel Distributed Computing Environments

Linda [77, 78] is a concurrent programming model that was developed by Yale University. The primary concept in Linda is that of a "tuple-space", an abstraction via which cooperating processes communicate. The central theme of Linda has been proposed as an alternative paradigm to the two traditional methods of parallel processing, viz. those based on shared memory and on message passing. The tuple-space concept is essentially an abstraction of distributed shared memory, with one important difference (tuple-spaces are associative), and several minor distinctions (destructive and non-destructive reads, and different coherency semantics are possible). Applications use the Linda model by embedding explicitly, within cooperating sequential

programs, constructs that manipulate (insert/retrieve tuples) the tuple space. From the application point of view Linda is a set of programming language extensions for facilitating parallel programming. There have been serveral commercial implementations of Linda. C-Linda from Scientific Computing Associates Incorporated is one of most popular Linda systems. POSYBL is a public domain version of Linda developed at the University of Crete. POSYBL is one of the first public domain Linda programming environments. It is also one of the best since it is the only public domain Linda system that supports a distributed tuple space rather than a centralized tuple server. A major difference between POSYBL and the commercially supported versions of Linda is the fact that POSYBL is implemented strictly in terms of a library and therefore cannot utilize the optimizations possible with the compiler-based Linda system. However, the performance of POSYBL, is still high enough to make the system quite useful.

P4 is a library of macros and subroutines developed at Argonne National Laboratory for programming a variety of parallel machines. The P4 system [40, 43, 124] supported both the shared memory model (based on monitors) and the distributed-memory model (using message-passing). For the shared-memory model of parallel computation, P4 provides a set of primitives from which monitors can be constructed, as well as a set of useful monitors. For the distributed-memory model, P4 provides typed send and receive operations, and creation of processes according to a text file describing group and process structure. P4 is intended to be portable, simple to install and use, and efficient.

TCGMSG [89] (Theoretical Chemistry Group Message passing system) is a simple message passing system that has risen to a position of prominence among computational chemists. It is very efficient with communication taking place over direct, point-to-point TCP/IP sockets.

PVM (Parallel Virtual Machine) [56, 74, 75, 76, 84, 132, 173, 174, 175] was developed at Oak Ridge National Laboratory and Emory University and is a software package which allows the utilization of a heterogeneous network of parallel and serial computers as a single computational resource. Facilities for spawning, communication, and synchronization are supported. PVM has been widely accepted by hardware vendors(Cray, Convex, SGI, HP, etc.) and therefore has spawned several related development efforts. Table I summarizes some of the projects related to PVM.

MPI [67] stands for Message Passing Interface. The goal of MPI, simply stated, is to develop a widely used standard for writing message-passing programs. As such the interface should establish a practical, portable, efficient, and flexible standard for message passing. The main advantages of establishing a message-passing standard

Product	Function
DoPVM	Distributed object PVM
FT-PVM	Fault Tolerant PVM
PVM++	Message passing object oriented PVM
HeNCE	Graphical front-end to PVM
Xab	Run time monitoring and debug of PVM program

Table I. PVM related systems

are portability and ease-of-use. In a distributed memory communication environment in which the higher level routines and/or abstractions are build upon lower level message passing routines the benefits of standardization are particularly apparent. Furthermore, the definition of a message passing standard provides vendors with a clearly defined base set of routines that they can implement efficiently, or in some cases provide hardware support for, thereby enhancing scalability.

2.2.6. Comparison of Several Parallel Distributed Computing Environments

Douglas and others [57, 185], present experiments comparing the communication times for a number of different network programming environments on two isolated SUN SPARC-station 1 workstations.

With TCGMSG, point to point TCP sockets are established between every pair of nodes. This is done when the program is initiated and these sockets are not reclaimed in the course of the calculation. We call this approach the static TCP socket system. The static TCP socket systems method can run into trouble scaling up to large numbers of nodes since the number of open file descriptors per node grows as the twice the number of nodes. PVM and P4 both use dynamic TCP sockets and PVM also provides daemon communication. This means they establish a socket between two communicating nodes at run time when they first communicate with each other. This method has the advantage that it will scale better on a large set of nodes as long as none of the processors runs out of file descriptors (as in the static TCP socket case). One disadvantage of dynamic TCP relative to static TCP is that the first communication is significantly slower than subsequent communications.

Table II shows clear and consistent performance differences for message ranging in size from 100 bytes to one megabyte. TCGMSG was significantly faster for all message sizes. P4 and PVM and C-Linda (in that order) represent a middle range in performance. Finally, POSYBL was the slowest system and even failed for the largest message size. It is clear that the management of message buffers at either end of the communication plays a major role in the overall communication performance. This follows from the fact that systems using identical network protocols (TCGMSG, P4,

Bytes in	Message passing				Virtual shared memory	
message	TCGMSG	P4	PVM[1]	PVM[2]	C-Linda	POSYBL
100	0.0556	0.0408	0.0350	0.0142	0.0254	0.0126
400	0.1632	0.1538	0.1194	0.0494	0.0880	0.0454
1000	0.3390	0.3174	0.2174	0.1082	0.1834	0.1030
4000	0.6350	0.5194	0.4520	0.1856	0.3792	0.2622
10000	0.8548	0.6098	0.4706	0.2794	0.3732	0.3110
40000	1.0012	0.6482	0.5432	0.3246	0.4736	0.2930
100000	0.9920	0.6492	0.5614	0.3418	0.5140	0.1586
400000	1.0074	0.6594	0.5784	0.3578	0.5364	0.0944
1000000	1.0112	0.6600	0.5748	0.3538	0.5388	—

Table II. Average data transfer rates for the two node studies[31]. All rates are in megabytes per second. 1 use direct TCP communication and 2 use daemon-based communication.

and PVM) displayed very different results.

It is important to note that two node, point-to-point communication tests are a very simple way to compare programming environments. More complicated communication patterns found in actual applications are essential to make a fair and complete comparison.

Simple communication tests indicated that the increase in efficiency was of the order of a factor of 30% for daemon communication and only about 60% for direct TCP communication under optimal conditions for the PVM environment. Some of this degradation was caused by another facet of the PVM message passing mechanism – that of requiring separate buffer initialization, and packing calls before a message may be sent. This latter characteristic is necessitated by the desire to support heterogeneity, both in terms of message contents and because sending and receiving processors might utilize different data representations. However, in practice, most messages are of homogeneous content, i.e. most messages carry a single data type, that too from a single data area or array. Further, architecture trends follow standard data representation formats – most modern computers utilize identical representations, and those that do not, usually differ in either word lengths or byte ordering only.

Based on the reasoning above, White et al [185] devised an alternative message passing mechanism for the PVM system. This enhancement is based on a multi-party protocol architecture where one-to-one, one-to-many, and many-to-many communication are implemented robustly on pairwise connections. From the programming interface point of view, the new message passing scheme, accessible via the pvm_fsend() and pvm_frecv() calls, permit the direct transfer of user program data without requiring buffer initialization and packing. However, data conversion can still be included

Platform	Throughput (Kb/sec)			
Msg. size	1 byte	100 bytes	10kB	1MB
Daemon	0.06	12.88	263.41	358.48
Fsend	0.49	81.79	358.48	1003.87
TTCP	0.65	130.45	965.04	1125.24

Table III. Point-to-point communication bandwidth in PVM[108]

if communicating between different architectures, thus retaining data heterogeneity but not heterogeneity of message content.

The pvm_fsend() and pvm_frecv() library was implemented and tested on a variety of environments and networks. Table III indicates the performance of this communication scheme for simple point-to-point data transfer, for a variety of message sizes, for the SPARC-station 1 + Ethernet cluster. Also shown, for reference, are the corresponding values for daemon-based PVM communication, and for a stand-alone benchmarking program, viz. TTCP.

From the table it can be observed that the enhanced communication scheme delivers throughput several times as much as the daemon based communication. However, it also indicates that, except for large messages, even the enhanced communication mechanism delivers only a fraction of the throughput actually attainable by software as indicated by the reference TTCP numbers which, incidentally, are of the order of 70–95% of the theoretical maxima.

2.3. Order Statistic Filters and Stack Filters

2.3.1. Median-type Filters

Since their introduction in the early 1970's [177], the standard median (SM) filter has had widespread application in both signal and image processing as an alterative to linear filters. The theory of SM is that of order statistics [53, 157]. Order statistics have played an important role in statistical data analysis and especially in the robust analysis of data contaminated with outlying observations, called outliers [53, 86]. One of the most important applications of order statistics is in the robust estimation of parameters [86, 114]. The median is a prominent example of a robust estimator.

Let $X_1, X_2, \cdots, X_n$ be random variables. If they are arranged in ascending order of magnitude, $X_{(1)} \leq X_{(2)} \leq \cdots \leq X_{(n)}$, $X_{(i)}$ is called the ith-order statistic.The maximum and the minimum of X_i, $i = 1, \cdots, n$ are denoted by $X_{(n)}$, $X_{(1)}$. A very

important order statistic is the *median*, $med(X_i)$, given by

$$med(X_i) = \begin{cases} X_{p+1} & \text{if } n = 2p+1 \\ (X_p + X_{p+1})/2 & \text{otherwise.} \end{cases} \tag{2.3}$$

The one-dimensional median filter of size n, where $n = 2p+1$, is defined by

$$y_i = med(X_{i-p}, \cdots, X_i, \cdots, X_{i+p}),\ i \in Z \tag{2.4}$$

where Z denotes the set of integers.

The two-dimensional median filter of size $n \times m$, where $n = 2p+1$, $m = 2s+1$, $\{X_{ij}\}$, $i, j \in Z^2$, is defined by $y_{ij} = med(X_{i+p,j+s};\ (p,s) \in A),\ (i,j) \in Z^2$ where the set $A \subset Z^2$ is the filter window.

Median filters can be described in terms of statistical analysis and deterministic analysis. Based on statistical analysis [53, 86], median filters perform well for *long-tailed* noise distributions (e.g. Laplacian noise), whereas their performance is poor for *short-tailed* noise distributions (e.g. uniform noise). This fact suggests that the median filter is efficient at removing impulsive noise. The good performance of the median filter for *long-tailed* distributions is explained by the fact that it minimizes the L^1 norm [86, 157]: $\sum_{i=1}^{n} |x_i - T_n| \rightarrow min$, where T_n is the estimator based on random variables $X_1, \cdots, X_n$. From the equation 2.3.1, the median is the maximum likelihood estimate (MLE) of location for the Laplacian distribution: $f(x) = \frac{1}{2}e^{|x-\mu|}$ In general the median filter performance is compared to the performance of the *moving average or mean* filter: $y_i = \frac{1}{n}\sum_{j=i-p}^{j+p} x_j$. which is essentially a "moving" arithmetic mean. The arithmetic mean is the MLE of location for the Gaussian distribution and it minimizes the L^2 norm.

The median is a B-robust operator since its influence function is bounded provided f is bounded away from zero at the median [86]: $IF(x; med, F) = \frac{1}{2f(F^{-1}(1/2))} sign(x - F^{-1}(\frac{1}{2}))$. Therefore, a single outlier (e.g. impulse) can have no effect on its performance, even if its magnitude is very large or very small. However, the influence function of the arithmetic mean for the Gaussian distribution is given by [86]: $IF(x; \vec{x}, F) = x$ and it is unbounded. Therefore, the moving average filter is very susceptible to impulses.

Edge information is very important for human perception. Edges, by definition, contain high frequencies. Although both median and mean filters are low-pass filters, the median filter tends to preserve edge sharpness [14, 38, 193], owing to its robustness properties, while the mean filter smooths them. The median filter not only smooths noise in homogeneous image regions, but it also tends to produce regions of constant

or nearly constant intensity [37]. Usually, they are either linear patches or blotches. These effects are undesirable because they are perceived as lines or contours which do not exist in the original image.

In the deterministic analysis of median filters, the basic problem is that of finding signals, called roots or fixed points, which are invariant under median filtering [8, 70, 179]. There are several problems related to the median filters' roots:

- determination of the shape of a signal which is a root of a one- or two-dimensional median filter.
- construction and counting of the number of a median's roots.
- the rate of convergence of a non-root signal to a root after successive passes through the median.

These three problems form the subject of the deterministic analysis of median filters [6, 13, 61, 58, 71, 184]

There are several modifications and extensions of the standard median.

Separable Median Filter: [147, 148] This aims at reduction of the computational complexity for median filter computation. A separable two-dimensional median of size n results from two successive applications of a one-dimensional median filter of length n along rows and then along columns of an image (or vice versa):

$$\begin{aligned} y_{ij} &= med(z_{i,j-p}, \cdots, z_{ij}, \cdots, z_{i,j+p}) \\ z_{ij} &= med(x_{i-p,j}, \cdots, x_{ij}, \cdots, x_{i+p,j}) \end{aligned} \tag{2.5}$$

The main advantage is its low computational complexity in comparison with that of the non-separable median filter, since it sorts n numbers two times, whereas the non-separable $n \times n$ median sorts n^2 numbers.

Recursive Median Filter: This is defined as

$$y_i = med(y_{i-p}, \cdots, y_{i-1}, x_i, \cdots, x_{i+p}). \tag{2.6}$$

Its output tends to be much more correlated than that of the standard median filter. Recursive median filters have higher immunity to impulsive noise than have non-recursive median filters [7, 36]

Weighted Median Filters (WMF): This is defined as [105]

$$y_i = med(w_{-p} \diamond x_{i-p}, \cdots, w_p \diamond x_{i+p}) \tag{2.7}$$

where $w \diamond x$ denotes duplication of x w times: $w \diamond x = x, \cdots, x$ $(w\ times)$ It is closely related to the FIR filter of the form $y_i = \frac{\sum_{j=-p}^{p} w_j x_{i+j}}{\sum_{j=-p}^{p} w_j}$. Brownrigg [42], Yli-Harja [192],

and Ko [110] analyze the performance of the weighted median filter. They have shown that the WMF can outperform the standard median filter [110]. There is a connection between stack filters and weighted median filters [192] which can be used to derive the statistical and deterministic properties of WMF.

Max-Median filters and Multistage Median Filters: These aim at preserving the structural and spatial neighborhood information which could be destroyed by the ordering process. The max-median filter is defined by [11]:

$$y_{ij} = max(z_1, z_2, z_3, z_4), \tag{2.8}$$

where

$$\begin{aligned} z_1 &= med(x_{i,j-p}, \cdots, x_{ij}, \cdots, x_{i,j+v}) \\ z_2 &= med(x_{i-p,j}, \cdots, x_{ij}, \cdots, x_{i+p,j}) \\ z_3 &= med(x_{i+p,j-p}, \cdots, x_{ij}, \cdots, x_{i-v,j+v}) \\ z_4 &= med(x_{i-p,j-p}, \cdots, x_{ij}, \cdots, x_{i+v,j+v}). \end{aligned}$$

Its performance can be improved considerably if the median operator is used to replace the max operator in equation 2.8. The resulting filter belongs to the multistage median filters:

$$y_{ij} = med(med(z_1, z_2, x_{ij}), med(z_3, z_4, x_{ij}), x_{ij}). \tag{2.9}$$

Multistage median filters can preserve details in horizontal, diagonal, and vertical directions since they use sub-filters that have regions of support along these directions. [10]

Median Hybrid Filters: This aims also at preserving the spatial information of an image by using linear filter substructure. It is a combination of linear filters and median filters and has the following definition :

$$y_i = med(\Phi_1(x_i), \cdots, \Phi_M(x_i)), \tag{2.10}$$

where the filters $\Phi_j(x_i), j = 1, \cdots, M$ are linear FIR or IIR filters. Heinonen [90], Astola [13] analyse the performance of median hybrid filters. An extended family of FIR hybrid median filters with good transient response are presented in [186].

2.3.2. Order Statistic Filters

The class of order statistic filters includes a large number of nonlinear filters. The L filter (also called the order statistic filter) is an important generalization of the

median which can be defined as:

$$y_i = \sum_{j=1}^{n} a_j x_{(j)}, \tag{2.11}$$

where $x_{(j)}$, $j = 1, \cdots, n$ are the order statistics of $x_{i-p}, \cdots, x_{i+p}$. The moving average, median, rth ranked-order, and α-trimmed mean are special cases if the coefficients a_j, $j = 1, \cdots, n$ are defined appropriately.

Ranked-order filters [91] are very similar to median filters which are straightforward applications of order statistics in filtering. An *rth ranked-order* filter can be defined as:

$$y_i = rth\ order\ statistic\ of \{x_{i-p}, \cdots, x_i, \cdots, x_{i+p}\} \tag{2.12}$$

Weight order statistic filter [192] is a general weight median filter which can be defined as

$$y_i = rth\ order\ statistic\ of \{w_{-p} \diamond x_{i-p}, \cdots, w_p \diamond x_{i+p}\} \tag{2.13}$$

where $w \diamond x$ denotes duplication of x w times: $w \diamond x = x, \cdots, x$ $(w\ times)$

The *α-Trimmed Mean Filter* [32] is good compromise between the moving average filter, which is good at suppressing additive white Gaussian noise, and the median filter, which is good at suppressing impulses and preserving edges. It satisfies:

$$y_i = \frac{1}{n(1-2\alpha)} \sum_{j=\lfloor \alpha n \rfloor + 1}^{n - \lfloor \alpha n \rfloor} x_{(j)}, \tag{2.14}$$

where $\lfloor \alpha n \rfloor$ is the integer part of αn. The α-trimmed mean filter rejects the smaller and the larger data based on the coefficient α, $0 \leq \alpha \leq 0.5$. If $\alpha = 0$, no data are rejected, which is equivalent to the moving average filter. If α is close to 0.5, all data but the median are rejected.

L filters are based on the theory of robust L estimators [86, 39]. The filter coefficients, a_j, $j = 1, \cdots, n$, can be chosen to satisfy an optimality criterion that is related to the probability distribution of the input noise. Structural constraints can be incorporated in the optimization function in order to design filters that are sensitive to local signal structure [141]. The deterministic properties of the L filters and relation to linear filters are discussed in [120]. The ability of the L filter to have optimal coefficients for a variety of input distributions makes it suitable for a large number of application. Another advantage of the L filter over the median filter is that it has no streaking effect. However, the L filter involves greater computational complexity than the median filter.

The *R filter* is another nonlinear filter which is based on R estimators [86, 73].

The most important R filter is the *Wilcoxon* filter [51, 72]:

$$y_i = med\{\frac{x_{(j)} + x_{(k)}}{2},\ 1 \leq j \leq k \leq n\}. \tag{2.15}$$

Wilcoxon filters have been proved to be effective in suppressing additive Gaussian noise, but they do not preserve edges well. If the sum in the equation (2.15) is restricted to a maximum distance $j - k < D$, the *modified Wilcoxon filter* can be expressed as [72] $y_i = med\{\frac{x_{(j)}+x_{(k)}}{2},\ 1 \leq j \leq k \leq n, k - j < D\}$. This modified Wilcoxon filter has better edge preservation properties than the standard Wilcoxon filter. However, a disadvantage of the Wilcoxon filter is its computational complexity. It requires $n(n+1)/2$ additions and the ordering of $n + n(n+1)/2$ items . A fast algorithm for the Wilcoxon filter is suggested in [113].

2.3.3. Stack Filters and Threshold Decomposition

Stack filters form an extension of the class of *order statistics filters*. This includes, but is not limited to, median-type filters, weight order statistic filters [192], and all compositions of morphological filters composed of opening and closing operations[88, 131].

Stack filters[183] originate from two fundamental properties of the median filter, the weak superposition property known as *threshold decomposition*[62, 63] and the ordering property[142] called the *stacking property* in [183].

Let x_i be an M-valued signal: $x_i \in \{0, 1, \cdots, M-1\}$, for which there are the $M-1$ thresholds: $\{1, 2, \cdots, M-1\}$. The signal x_i can be decomposed into $M-1$ binary valued signals x_i^l, $l = 1, 2, \cdots, M-1$, using the functions $T_l(x_i)$:

$$x_i^l = T_l(x_i) = \begin{cases} 1 & \text{if } x_i \geq l \\ 0 & \text{otherwise.} \end{cases} \tag{2.16}$$

These $M-1$ binary valued signals can be filtered independently.

A Boolean function $f(\cdot)$ operating on a binary vector of length n is said to possess the *stacking property* if the binary output signal y_i^l at time i consists of a column of 1's having a column of 0's on top. The filters satisfying the stacking property are called *stack filters* which can be defined as:

$$y_i = S_f(\vec{x}_i) = \sum_{l=1}^{M-1} y_i^l = S_f(\sum_{l=1}^{M-1} T_l(\vec{x}_i)) = \sum_{l=1}^{M-1} S_f(T_l(\vec{x}_i)) = \sum_{l=1}^{M-1} f(\vec{x}_i^l) \tag{2.17}$$

where vectors $\vec{x}_i = [x_{i-p}, \cdots, x_i, \cdots, x_{i+p}]$, and $\vec{x}_i^l = [x_{i-p}^l, \cdots, x_i^l, \cdots, x_{i+p}^l]$.

The Boolean function $f(\cdot)$ determines the properties of the stack filter. Gilbert [81]

showed that a necessary and sufficient condition for a Boolean function to satisfy the *stack property* is that it contains no complements, $\bar{x}_i$, of the input variables x_i. The stackable functions are also called *positive Boolean functions.*

Although positive Boolean functions provide a large class of filtering operations, we are interested in examining an even larger class of filters. This larger class is obtained by allowing the output of a Boolean function to be randomized [49].

Let the 2^b possible binary sequences of length b be ordered in some fashion as $\vec{x}_1, \vec{x}_2, \cdots, \vec{x}_{2^b}$. A window width b *randomizing Boolean function* $\mathcal{B}(\cdot)$ is defined by the vector $\vec{P}_{\mathcal{B}}$ with 2^b elements, in which the ith element is

$$P_{\mathcal{B}}(1|\vec{x}_i) = \Pr(\mathcal{B} \text{ produces output } 1 \mid \vec{x}_i \text{ is in the window corresponding to } \mathcal{B}), \tag{2.18}$$

where $i = 1, 2, \cdots, 2^b$. Also define $P_{\mathcal{B}}(0|\vec{x}) = 1 - P_{\mathcal{B}}(1|\vec{x})$.

A randomizing Boolean function $\mathcal{B}(\cdot)$ is said to possess the *probabilistic stacking property* if and only if

$$E\left(\mathcal{B}(\vec{x})\right) \geq E\left(\mathcal{B}(\vec{y})\right) \quad \textit{whenever } \vec{x} \geq \vec{y}, \tag{2.19}$$

where $E(\cdot)$ is the expectation operator as defined on the appropriate probability space.

The addition of randomization allows the Boolean function's expected output for a given binary input sequence to be any real number in [0, 1]. This allows average output of the filter to be the same as the deterministic output of many well-known filters. For example, linear filters with nonnegative weights on the bits in the window can be realized as randomizing Boolean functions satisfying the probabilistic stacking property [49].

There is another way to extend stack filters which leads to so-called *generalized stack* (GS) filters [116], which allows different logical operators on different levels of the threshold decomposition architecture.

Let x_m be a $(2I + 1) \times n$ binary array at threshold level m. The ordered set of $M - 1$ Boolean functions $\{f^1(\cdot), \cdots, f^{M-1}(\cdot)\}$ is called a *stacking set* of Boolean functions if

$$f^{m+1}(x^{m+1}) \leq f^m(x^m), \quad m = 1, 2, \cdots, M - 2. \tag{2.20}$$

A window width N, M-value *generalized stack filter* $F_{gs}(\cdot)$ is a stacking set of $M - 1$ Boolean functions. The operation of this filter on the input $\vec{x}$ is defined as follows:

$$F_{gs}(\vec{x}) = \sum_{m=1}^{M-1} f^m(x^m). \tag{2.21}$$

Great advances have been made recently in the design of optimal and adaptive stack filters and generalized stack filters [49, 50, 69, 116, 117, 118, 119, 194]. Both an estimation approach and a structural approach have been developed [50]. The estimation approach employs the *minimum absolute error* (MAE) criterion because of its robustness [49]. Optimal stack filters and generalized stack filters based on the MAE criterion can be found via linear programming (LP) [49, 116, 194]. The computational complexities of the algorithms are very high since the number of variables and constraints in the LP procedure grows exponentially with the window width of the filters. An improved method is to use the *adaptive stack filter* [117]. This approach alleviates the modelling of the signal and noise by taking a part of the input signal to train the stack filter. The advantage of this algorithm is that only simple arithmetic operations are required. The disadvantage of the algorithm is that the number of variables still grows exponentially with the increase of the window width. The other disadvantage is that the convergence speed of the algorithm of the adaptive stack filters is very slow.

A new design method is suggested in [119] based on threshold decomposition and Bayesian decision theory. The maximum number of unknown variables is 2^N for an N-length stack filter [117]. If there are some constraints on the positive Boolean functions, any positive Boolean function can be equivalent to a corresponding threshold logic function [192] in which the number of variables is $N+1$. More importantly, any linear adaptive algorithm can be applied to the new optimization problem. Most of them have a remarkable higher convergence rate than that of Lin's algorithm [119]. It is worth noting that the new adaptive algorithm does not generally give optimal stack filters under the MAE criterion.

Neural filters [118] have been suggested as a way of solving the problem of optimal generalized stack filter design. The neural network representation enables the stack filter to be implemented using sorting operations in the real domain. This reduces the amount of computation since the complexity of implementing stack filters in the binary domain increases exponentially with the word length. Two classes of neural filters have been defined [118], hard neural filters and soft-neural filters. The hard neural filters are defined by a set of neural networks in which the activation functions are unit step functions. If they satisfy the stacking property, the hard neural filters reduce to GS filters. Soft neural filters are defined by neural networks whose activation functions are sigmoidal.The universal approximation property of neural networks [94] suggests that soft neural filters can approximate all filters defined by linear and nonlinear continuous functions such as linear FIR filters and micro-statistic filters [9]. Moreover, soft neural filters can also approximate the hard ones. Two

adaptive neural filtering algorithms, the adaptive least mean absolute error (LMA) algorithm and the adaptive least mean square error (LMS) algorithm, are used for finding optimal neural filters under the MAE and MSE criteria, respectively [118]. Hard neural filters and soft neural filters can be trained using these two algorithms.

2.4. Fractals, Iterated Function Systems and Inverse Fractal Transformations

There are three popular ways to construct a fractal scene. The first is to use L-systems [161] to models fractal botanical models. The second is to use fractional Brownian motion (fBm). The third is to use the Iterated Function System (IFS) developed by Barnsley and his collaborators [21, 23]. L-systems and fractional Brownian motion are limited models. L-systems are only suitable for botanical graphics; fBm is defined in a one-dimensional framework and it is very difficult to generalize it to higher dimensions. Fractal techniques based on iterated function systems are the most flexible generalization from one dimension to higher dimensions is very natural and easy, and highly complex spatial information can be derived from temporal iteration that is governed by only a small set of parameters.

2.4.1. Iterated Function Systems

In Barnsley's IFS theory, a deterministic and random iterated function and system can be defined as the following :

Definition 2.3 *The Hausdorff distance between sets K and L, $K, L \in X$, can be defined as*

$$h(K,L) = \max\{\max\{d(x,K) : x \in L\}, \max\{d(y,L) : y \in K\}\}, \tag{2.22}$$

where $d(x,K)$ is the distance from x to K, $d(x,K) = \min\{d(x,y) : y \in K\}$, and $d(y,L)$ is the distance from y to L, $d(y,L) = \min\{d(y,x) : x \in L\}$.

Definition 2.4 *A deterministic iterated function system (IFS) is an N-tuple $(w_1, w_2, \cdots, w_N)$ of maps from a compact metric space (X,h) into itself, where h is Hausdorff distance. A map $w : X \to X$ is called a contraction iff there exists a constant $c \in \mathbf{R}$ with $0 \leq c < 1$ and $h(w(x), w(y)) \leq ch(x,y)$, $\forall x, y \in X$. The smallest c with this property is called the Lipschitz constant of w and is denoted by* $\mathrm{Lip}(w)$. *A deterministic IFS consists of contractions $w_1, w_2, \cdots, w_N$.*

Hutchinson [96] proved, that if $W(K) = \bigcup_{i=1}^{N} w_i(K)$, W is a contraction with respect to h, with $\mathrm{Lip}(W) \leq \max(\mathrm{Lip}(w_1), \cdots, \mathrm{Lip}(w_N))$, and has a unique fixed point

A in X. The fixed point A of W is called the *attractor* of the IFS$(w_1, w_2, \cdots, w_N)$.

Definition 2.5 *A random iterated function system IFS$(w_1, w_2, \cdots, w_N$, $p_1, p_2, \cdots, p_N)$ consists of Lipschitz map functions w_i in compact metric space (X, h) with probability p_i, $i = 1, 2, \cdots, N$ and $\sum_{i=1}^{N} p_i = 1$. Choose $x_0 \in X$ and then choose, recursively and independently,*

$$x_n \in \{w_1(x_{n-1}), w_2(x_{n-1}), \cdots, w_N(x_{n-1})\}, \quad n = 1, 2, \cdots \tag{2.23}$$

where the probability of the event $x_n = w_i(x_{n-1})$ is p_i.

Thus it defines a discrete-time Markov process $\{Z_n,\ n = 1, 2, \cdots, \}$ [21, 23].

$$P(Z_n \in B | Z_{n-1} = x_{n-1}, \cdots, Z_0 = x_0) = P(x_{n-1}, B), \tag{2.24}$$

where

$$P(x, B) = \sum_{i=1}^{N} p_i \delta_{w_i x}(B) \tag{2.25}$$

is the probability of transfer from $x \in X$ to the Borel set B in X, where δ_x is the Dirac measure concentrated at x. Barnsley [23] proved that if $\mu \in P(X)$, the set of Borel probability measures on X, is a stationary initial distribution and maps $(w_1, w_2, \cdots)$ are Lipschitz, the process $\{Z_n\}$ converges in distribution to μ.

For the fractal interpolation problem, let

$$Y_i = f(X_i), \quad i = 1, 2, \cdots, N, \tag{2.26}$$

where $\{Y_i\}$ are data, $\{X_i\}$ are interpolating points with $X_1 < X_2 < \cdots < X_N$ and f is an unknown function which displays some sort of self-similarity under magnification. Define a graph $G = \{(X_i, Y_i), i = 1, \cdots, N\}$. Barnsley [18, 19] suggested finding an IFS whose attractor approximates this graph G and which would give an estimate, $\overline{f}$, for f. The basic structure is that the maps w_i are affine transformations with the special structure

$$w_j \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a_j & 0 \\ c_j & d_j \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e_j \\ f_j \end{pmatrix}, \tag{2.27}$$

Berger [34] uses random IFS and affine transformations to show how refinement methods for smooth curve generation can be carried out efficiently. The applications include Béziear curves, splines, wavelets and various interpolants. Barnsley et al [26] have shown that it is possible to design the interpolation such that $\overline{f}$ is in $C^l[X_1, X_N]$, i.e., $\overline{f}$ has lth continuous derivative on $[X_1, X_N]$, where l is any non-negative integer. Since any lower-dimensional function f can be regarded as a projection of a

high-dimensional function, the graph G also can be considered as a projection of another graph in higher dimensions. Barnsley et al [22] have considered finding an IFS whose attractor approximates this high-dimensional graph, and from this they obtain an interpolant for f, by projection of this high-dimensional attractor, which will not limit f to be self-similar. They called it *hidden variable fractal interpolation.* Geronimo et al [79] extended IFS interpolation to two-dimensional fractal surfaces. Their algorithm allows the construction of these surfaces over polygonal regions with arbitrary interpolation points.

The recurrent iterated function system (RIFS) [24, 44], also called local iterated function system (LIFS), generalizes iterated function systems. The flexibility of RIFS permits the construction of more general sets and measures which do not have to exhibit the strict self-similarity of the IFS case. RIFS can be defined as follows :

Definition 2.6 *Let (X_j, d_j) be compact metric spaces, $j \in \{1, 2, \cdots, N\}$,and let (H_j, h_j) denote the associated metric spaces of nonempty compact subsets which use the Hausdorff metrics. Let there be defined maps $w_{ij} : H_j \to H_i$, $\forall (i,j) \in I$, where I is some set of pairs of indices with the property that $I(i) = \{j | (i,j) \in I\} \neq \emptyset\}$ and $h_i(w_{ij}(A), w_{ij}(B)) \leq s_{ij} h_j(A,B)$, for some s_{ij}, $\forall (i,j) \in I$ and $\forall A, B \in H_j$. Then when $s < 1$ there is a unique element $A = (A_1, \cdots, A_N) \in \tilde{H}$ such that*

$$A_i = \bigcup_{j \in I(i)} w_{ij}(A_j), \quad \text{for } i = 1, 2, \cdots, N, \tag{2.28}$$

i.e., $W(A) = A$, where $\tilde{H}$ consists of a stack of planes $K1, K2, \cdots, K_N$ with a point in $\tilde{H}$ being the N-tuple of one image in each plane and

$$W(A_1, \cdots, A_N) = \left(\bigcup_{j \in I(1)} w_1 j(A_j), \cdots, \bigcup_{j \in I(N)} w_{Nj}(A_j) \right). \tag{2.29}$$

Barnsley's Collage theorem [25, 19] tells us that, in order to control the closeness between the attractor A and the data set K under the Hausdorff distance, it is sufficient to control the closeness between K and $W(K)$ which is obtained by one-step iteration ahead of K by W under the Hausdorff distance. Here is the Collage theorem for IFS.

Theorem 2.1 *Suppose IFS$(w_1, w_2, \cdots, w_N)$ has an attractor A on a compact metric space (X, h). Let $K \subset X$ and $W : K \to K$, where $W(K) = w_1(K) \cup \cdots \cup w_N(K)$ and Lips$(W) = s$. If $h(W(K), L) < \varepsilon$ then*

$$h(A, K) < \frac{\varepsilon}{1 - \varepsilon}. \tag{2.30}$$

For a RIFS, there exists a corresponding collage theorem [24]. Φien et al [154] present a new collage theorem holding for a certain class of affine mappings called Affine Blockwise Averaging maps, which operate on the space of discrete signals and are suitable for the orthogonalized version of Jacquin's algorithm [103], introduced in Φien and Lepsoy [155]. The theorem provides a better bound on the distance between the original image and the attractor, by considering in the estimate norms of collage errors at successively coarser resolutions. Baharav et al [16] proposed a fast decoding algorithm based on a hierarchical interpretation of the IFS-code which can reduce the computation time by more than an order of magnitude.

2.4.2. Fitting Data with Fractional Brownian Motion

An important class of fractal signals is $1/f$ processes [109], which exhibit rich behaviour well suited to modelling a wide range of one-dimensional natural phenomena. $1/f$ processes are a class of random processes of which average spectral density is proportional to the inverse of frequency $1/f$ and can be characterized by an inherent scale invariance and persistent long-term correlation structure. In contrast to the well-studied family of ARIMA process, $1/f$ processes have received relatively little attention in the transitional signal processing literature. This has been due, at least in part , to the mathematical intractability of fractal processes. However, $1/f$ fractal signal representations in terms of orthonormal wavelet bases have been suggested recently [65, 163, 191] that considerably simplify the analysis of these processes.

A popular example of the $1/f$ processes is that of fractional Brownian motion (fBm) [128], which is a generalization of normal Brownian motion. The fBm $B_H(t)$ is a zero mean non-stationary Gaussian random process with the covariance function

$$r_{B_h}(t,s) = \frac{\sigma^2}{2}(|t|^{2H} + |s|^{2H} - |t-s|^{2H}) + o(|t|), \tag{2.31}$$

where the parameters σ^2 and $0 < H < 1$ characterize the process. The parameter H controls the "roughness" of the fBm such that an individual realization of the process has a fractal dimension [127] $D = 2 - H$. The H parameter also controls the shape of the average spectral density defined as

$$S(f) = \frac{c}{|f|^{r_b}} \tag{2.32}$$

where $r_b = 2H + 1$. As a result, the fBm serves as a good model for $1/f$ processes where $1 < r_b < 3$.

Wornell [191, 190, 189] suggests a new algorithm which uses the discrete wavelet transform [164, 126, 52] to derive an approximate maximum likelihood estimator

when a $1/f$ process is embedded in white Gaussian noise. Wornell's algorithm needs the wavelet coefficients of a Karhunen-Loève-like expansion for $1/f$ noise [188] to be uncorrelated over scale and time. Kaplan et al [109] improve Wornell's algorithm by using Haar based wavelets [126]. The coefficients of the new algorithm are weakly correlated and have a variance that is exponentially related to scale. Theoretical analysis and numerical simulation of Kaplan's algorithm indicate that it improves the accuracy of estimating H for moderate data length of the fBm; for longer lengths, both algorithms can find a very good estimate; for short data length of the fBm with additive noise, both algorithms are unreliable. The problem of how to improve a wavelet based fractal estimator for short data length is still open.

2.4.3. Inverse Problems of the Iterated Function Systems

As usual, inverse problems are hard, and potentially ill-posed. In a typical inverse problem in fractal construction, a single phenomenon is given, and must be reproduced in terms of some of its characteristics, or in the whole, by a fractal approximation. There are two forms [180] of inverse IFS problem; *Measure:* given a target (normalized Borel) measure ν, find an IFS whose invariant measure μ approximates ν as closely as possible (in terms of the Hutchinson metric); *Geometric:* given a target set S, find an IFS whose attractor A approximates S as closely as possible in geometry (in terms of the Hausdorff metric).

The inverse problem for measure can be defined as follows [21, 25] :

Definition 2.7 *Given a probability measure λ on K, where K is a compact metric space, find an IFS and associated probabilities p for which the p-balanced measure μ is close to λ (in the weak $*$ topology).*

Consider an IFS $\{K,\ w_i : i = 1, 2, \cdots, N\}$ where $K \subset \mathbf{C}$ and $w_i(z) = s_i z + b_i,\ i = 1, 2, \cdots, N$, with $s_i, b_i \in \mathbf{C},\ 0 \leq |s_i| < 1$. Then the moments $M_n = \int_k z^n d\mu(z)\ \ n = 0, 1, \cdots$ can be calculated. This follows from the stationarity condition $\mu(\mathbf{B}) = \int_k P(x, \mathbf{B}) d\mu(x)$, where $\mathbf{B}$ is Borel subset of K. The recursive formula is

$$M_n = \left(1 - \sum_{i=1}^{N} p_i s_i^n\right)^{-1} \sum_{i=1}^{N} \sum_{j=0}^{n-1} \binom{n}{j} s_i^j b_i^{n-j} p_i M_j. \tag{2.33}$$

The M_n values can be computed starting from $M_0 = 1$. In particular, we have available the reverse procedure, that of *matching* a finite number of moments, $M_n = g_n$, where $g_n = \int z^n d\lambda(z)$, $n = 1, 2, \cdots, M$, to get the IFS parameters [25, 2, 129, 130, 87]. However, because of the problems associated with the nonlinearity of the equations, the scheme is found to be extremely unstable [181, 180], and hence useless

from a practical viewpoint. Moreover, the complexity of this approach increases enormously in the two-dimensional case. Vrscay [181, 180] suggested a new method for minimizing a "Euclidean distances" between moments M_n and g_n. For a fixed number, N, of IFS maps, and M, the number of moments M_i to be "matched", the objective function to be minimized was the sum of the squared Euclidean distance in "moment space",

$$D_M^N(\pi) = \sum_{i=1}^{M}(g_i - M_i)^2. \tag{2.34}$$

Vrscay suggests using a Genetic Algorithm [93, 82] to minimize the above function.

For an inverse IFS problem of the geometric type, Withers [187] suggested applying Newton's method on the parameter space of the IFS to solve the problem of fitting a given linear function in the L^q norm with a function generated by an IFS. Walach [182] utilized a fixed-length *yardstick* to traverse the entire data to construct a piecewise linear for approximating a curve in order to compress an image. The compression rate is near 16:1.

Modern fractal image compression led to the creation of the concepts and mathematical results of iterated function systems. Barnsley and Sloan advertised in popular science magazines the incredible power of IFS for compressing colour images at compression rates of over 10000 : 1 [28]. In 1989 Jacquin proposed a fully automated algorithm (called as block-based image coding) for fractal image compression [100] which was based on local affine transformations, and was also called Recurrent IFS. He suggested an approach for partitioning a monochrome image into non-overlapping square pixel blocks, called *range blocks* (R_i), and larger square pixel blocks, called *domain blocks* (D_i), sorted into a set of categories such as shade blocks, edge blocks and midrange blocks, following classification [162]. For each range block, a domain block of the same category is searched such that its grey level under a local strictly contractive affine mapping (τ_i) minimizes its distance to the original block in the root-mean-square sense. Each affine mapping is is composed of a *geometric* part (S_i) which shrinks the domain block down to the size of a range block by pixel averaging, and a *massif* part (S_i) that transforms the obtained block by shuffling (J_i), scaling (α_0), with quantized parameters and addition of a constant grey-tone block (Δg). The general form is [104] :

$$T \circ S(\mu_{\neg D}) = \alpha_0 J(S(\mu_{\neg D})) + \Delta g \tag{2.35}$$

This scheme is in many aspects related to vector quantization (VQ) [80], with which it shares the idea of using a codebook providing a library for the selection of the domain blocks. However, the codebook in fractal compression is only a "virtual" one

since the domain blocks are not stored but are taken from the image itself, thereby exploiting the redundancy of the information present in the image.

Jacquin's papers provide a good starting point for further research and extensions in many possible directions. Mazel et al [134] use IFS and local IFS to represent discrete-time sequences. Beaumont [31] extends it to process sequences of video images, and Cochran et al [46] extend it to volumetric data, where the basic elements of the partition are three-dimensional blocks. The results of fractal volume compressions compare well against similar techniques based on vector quantization. Jacobs et al [99] conducted a thorough study to determine the optimal number of bits for the uniform quantization of α and Δg.

Φien et al [155, 156] express the item Δg in a three-dimensional subspace $\Delta g = \sum_{k=1}^{3} \alpha_k A_k$, where the α_k are real coefficients and $\{A_1, A_2, A_3\}$ are the fixed basis vectors. Φein et al first Gram-Schmidt orthogonalize the fixed-basis vectors, in effect decoupling the optimization of the scalar α_0 and the constant Δg. Saupe [166, 167] suggest another method, that of multi-dimensional nearest neighbour search, which runs in logarithmic time, to replace the common sequential search for a best match of image portion, which runs in linear time. Monro et al [137, 138] propose to express Δg with high order items as $\Delta g = \sum_{k=1}^{3} a_k x^k + \sum_{k=1}^{3} b_k y^k + t_0$. The parameters are optimally determined by applying a least square criterion. The authors report a significant increase in image quality by including these higher order items and, in particular, with the consequence that no searching procedure for domain blocks, which is the main factor leading to long encoding times with block-based fractal image coding, is needed. Barthel et al [29] propose an extension to linear scale transformation which applies a high order transformation in the frequency domain. Bit-rate reductions are higher than those achieved by "spatial-domain" fractal coding schemes. Fisher and Jacobs [60, 99] use a quad-tree, rectangular and triangular automatic partition of the range blocks in order to improve image fidelity. Another triangular partition scheme [54, 55] is provided by the triangular in a Delaunay tessellation [160], which permits an adaptive partition of the image support. Davoine etal [54, 55] show an improvement in the visual quality of reconstructed images, computing times and compression ratios.

CHAPTER 3

DESIGN OF INTERACTIVE PARALLEL DISTRIBUTED COMPUTING ENVIRONMENT

3.1. Introduction

A Parallel Virtual Machine (**PVM**) system is a software infrastructure that permits connection of heterogeneous Unix computers to be used as a unified general and flexible, message-passing, concurrent parallel computational resource. In this chapter, we describe the construction of PVM version 3, the principle of program design under PVM, the approach of dynamic monitoring and ways to debug PVM programs. Later in Chapter 4, we shall build parallel distributed algorithms with PVM system.

Interactive applications need a powerful general-purpose command language. We introduce such a powerful and embeddable command language, Tcl. Tk then extends the core Tcl facilities with additional commands for building user interfaces so that you can construct Motif-like interfaces by writing Tcl scripts instead of C code based on Tk. It raises the level of X-Windows programming and results in application development that is 5-10 times faster.

The combination of interactive and parallel processing will lead a new and useful application area, especially for visual science data, image analysis/processing and multimedia applications. We implement this composition based on a parallel distributed environment, PVM, and the interactive development tool, Tcl. The approach we use is to provide a Tcl version interface for all procedures in the PVM C library so that users can call any PVM procedure to do their parallel computing interactively. In order to solve the problem of there being no binary-type data in Tcl, we use Tcl's general-purpose hash table to define a kind of object, GBOX, in which to hold any binary data. Several Tcl procedures are implemented to do tasks related to GBOX.

3.2. The Method of Program Design Under A Parallel Virtual Machine

3.2.1. Construction of A Parallel Virtual Machine

Under PVM [74, 75, 76, 173, 175], a user-defined collection of serial, multi-processor, and vector super computers appears as one large distributed-memory computer, known as a virtual machine as shown in Figure 6. PVM is a public domain, full source code availability software system, of which the current version is 3. With source code, users can easily port PVM to any other new computer platform and

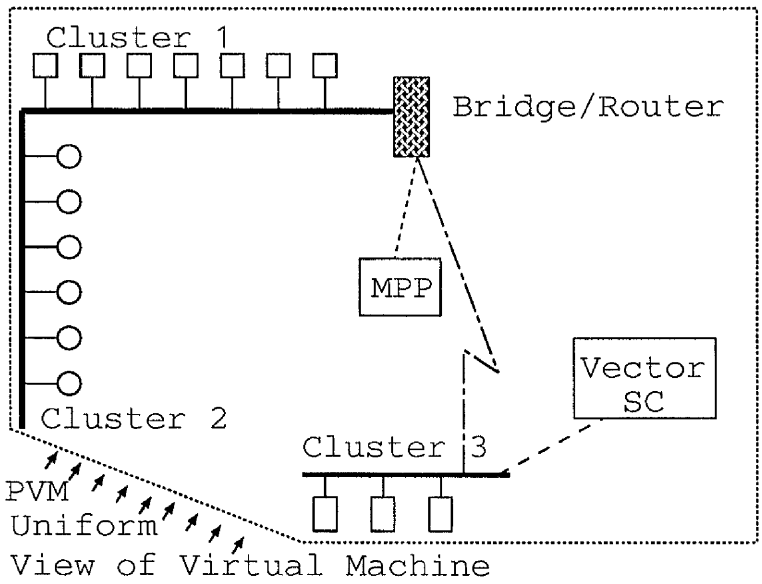

Fig. 6. PVM Architectural Overview[175]

improve the speed of message-pass with new network protocol.

Definition 3.1 *PVM computing environment is composed of user programs $\mathcal{U}$, system daemon $\mathcal{D}$, and interface library $\mathcal{I}$.*

Definition 3.2 *$\mathcal{U}$ is a standard single instruction single data-flow program, which consists of user data structure, C or Fortran control-flow statements, and explicit call-$\mathcal{I}$ statements.*

Figure 7 illustrates the **PVM** computing environment.

The PVM system software is composed of two parts. As explained in [75], the first part is a daemon $\mathcal{D}$, called pvmd3, that resides on all the computers making up the virtual machine. Pvmd3 is designed so any user with a valid login can install this daemon on a machine. When a user wants to run a PVM application, he executes pvmd3 on one of the computers which in turn starts up pvmd3 on each of the computers making up the user-defined virtual machine. The PVM application can then be started from a Unix prompt on any of these computers. Multiple users can configure overlapping virtual machines, and each user can execute several PVM applications simultaneously.

The second part of the system is a library of PVM interface $\mathcal{I}$ routines, libpvm3.a for C language or libfpvm3.a for Fortran 77. This library contains user-callable routines for message-passing, spawning processes, coordinating tasks, and modifying the virtual machine. Application programs must be linked with this library in order to use PVM.

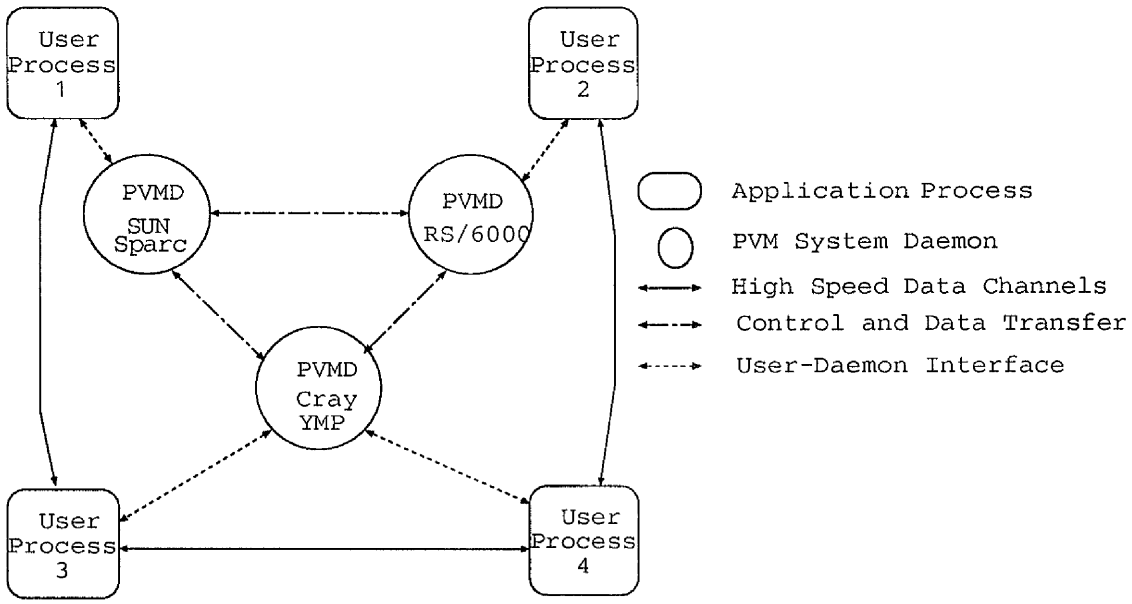

Fig. 7. PVM Computing Environment

Definition 3.3 *$\mathcal{I}$ consists of a C library and a Fortran library. Multi-data flow will appear when $\mathcal{I}$ is called since send-and-receive message are asynchronous.*

Definition 3.4 *Under a PVM, an application $\mathcal{A}$ is made up of a set of instances.*

An *instance* of an application subtask or *component* (realized as a *process*) [169], is the unit of computational abstraction in the PVM system. Each *process* is an executing *instance* of an application *component*, where a *component* is a domain-specific module amenable to single program multi-data flow (**SPMD**) execution. All *processes* that enrol in PVM are represented by an integer task identifier (*tid*). The *tid* is the primary and most efficient method of identifying processes in PVM. Since *tids* must be unique across the entire virtual machine, they are supplied by the local *pvmd* and are not user-chosen. PVM contains several routines that return *tid* values so that the user application can identify other *processes* in the system. An illustrative example of this computing model is shown in Figure 8.

Definition 3.5 *In a message-pass model, processes are created by the programmer explicitly; they communicate explicitly and may send data repeatedly to other processes.*

Instances communicate via the use of message-pass models; each message may contain data of several types. These message segments are built by provided library routines in a machine independent manner. Message exchange is asynchronous, in

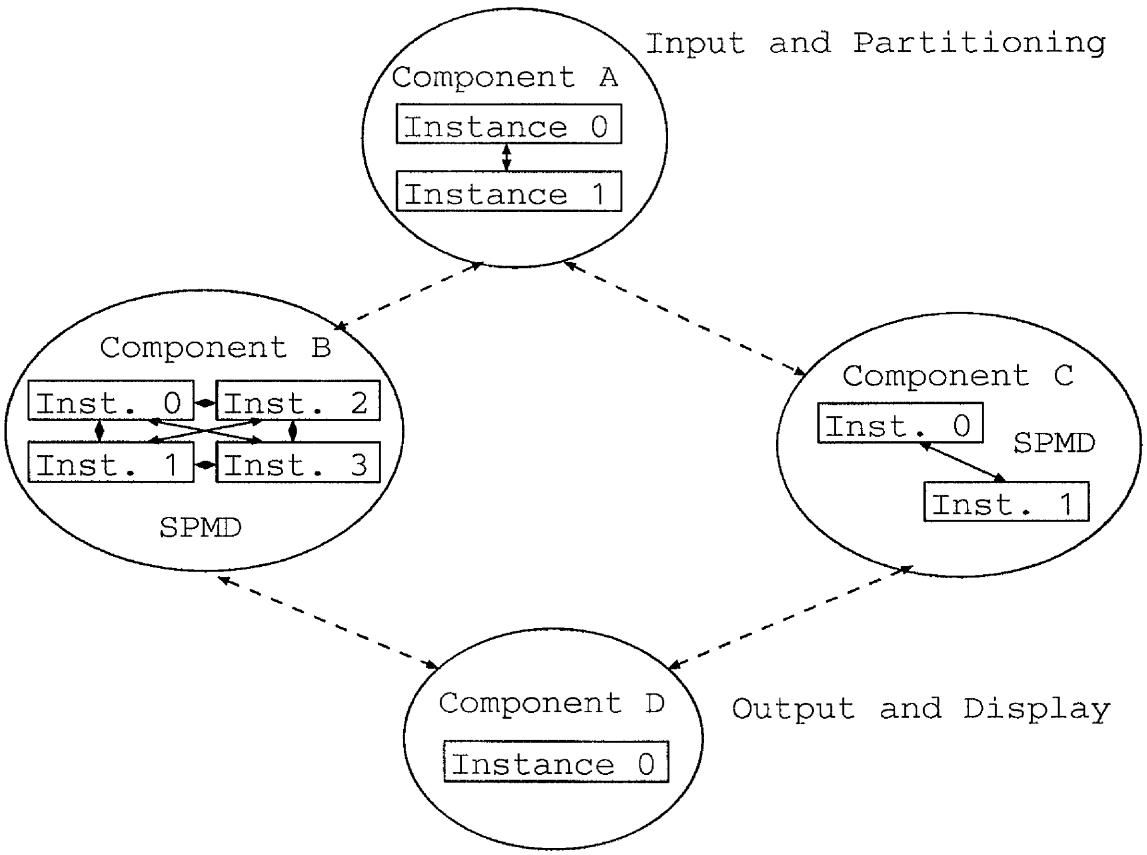

Fig. 8. PVM Concurrent Computational Model[169]

that a sending process may continue execution prior to physical message reception by the destination process. The PVM model guarantees that message order is preserved. If task 1 sends message A to task 2, and then sends message B to task 2, message A will arrive at task 2 before message B. The model assumes that any instance can send a message to any other PVM task, and that there is no limit to the size or number of such messages. While all hosts have physical memory limitations the communication model does not restrict itself to a particular machine's limitation and assumes that sufficient memory is available.

3.2.2. PVM User Interface Library

The following are a summary of the functions provided by PVM version 3 [74, 84].

- PVM supplies process control routines that enable a user process to become a PVM task, to become a normal process again, to spawn a new process, and to terminate other processes.
- PVM supplies dynamic configuration routines to add or delete hosts from the virtual machine, to start the system daemon, and to halt whole virtual machine.
- PVM supplies information request routines to find out information about the virtual machine configuration and active PVM tasks.
- PVM provides two methods of signalling other PVM tasks. One method sends a Unix signal to another task. The second method notifies a set of tasks about an event by sending them a message with a user-specified tag that the application can check for.
- If a host fails, PVM will automatically detect it and delete the host from the virtual machine. The status of hosts can be requested by the application. It is still the responsibility of the application developer to make his application tolerant of host failure. PVM makes no attempt to automatically recover tasks that are killed because of a host failure.
- PVM provides routines for packing and sending messages and unpacking messages between tasks.
- The PVM communication model provides asynchronous blocking send, asynchronous blocking receive, and non-blocking receive functions. In addition to these point-to-point communication functions the model supports broadcast to a set of tasks and to a user-defined group of tasks. Wildcard can be specified in the receive for the source and label allowing either or both of these contexts

to be ignored. A routine can be called to return information about received messages.

- The user can define multi buffers in PVM version 3. Message buffers are allocated dynamically so that the maximum size messages that can be sent or received is limited only by the amount of available memory on a given host.
- Dynamic process groups are implemented on top of PVM. In this implementation, a process can belong to multiple groups that can change dynamically at any time during a computation. Routines are provided for tasks to join and leave a named group. Tasks can also request information about other group members.

3.2.3. Developing a Good PVM Application

Application programs view PVM as a general and flexible parallel computing resource that supports a message-passing model of computation. This resource may be accessed at three different levels [56] :

1. The transparent mode, in which tasks are automatically executed on the most appropriate host (general the least loaded computer).
2. The architecture-dependent mode in which the user may indicate specific architectures on which particular tasks are to be executed.
3. The low-level mode in which a particular host may be specified.

Such layering permits flexibility while retaining the ability to exploit particular strengths of individual machines on the network.

Application programs under PVM may possess arbitrary control and dependency structures. In addition, any process may communicate and/or synchronize with any other. This allows for the most general form of multi-instruction multi-data flow (**MIMD**) parallel computation, but in practice-mode concurrent applications are more structured. Two typical structures are the Single Program Multi-Data (SPMD) model, in which all processes are identical, and the master/slave model, also known as server/clients, in which a set of computational slave processes performs work for one or more master processes.

There are no limitations to the programming paradigm a PVM user may choose. Any specific control and dependent structure may be implemented under the PVM system by appropriate use of PVM constructs. On the other hand there are certain considerations [175] of which the application developer should be aware when programming any message passing system.

The first consideration is task granularity. This is typically measured as a ratio of the number of bytes received by a process to the number of floating point operations a process performs. The tradeoff is the larger the granularity the higher the speedup but often there is a reduction in the available parallelism as well.

The second consideration is the number of messages sent. The number of bytes received may be sent in many small messages or in a few large messages. Using a few large messages can reduce the total message start-up time. There are cases where small messages can be overlapped with other computations so that their overhead is masked.

Definition 3.6 *Functional parallelism: There are different algorithmic subcomponents of the computation in each processor.*

Definition 3.7 *Data parallelism: the data are partitioned and distributed to all the processors; algorithmic subcomponents of the computation which are often similar are performed for each part of data and information is passed between processes until the problem is solved.*

A third consideration is whether the application is better suitable to functional parallelism or data parallelism. For example, a vector supercomputer may solve a part of a problem suitable for vectorization, a multiprocessor may solve another part of the problem that is suited to parallelization, and a graphics workstation may be used to visualize the generated data in real time. Each machine performs different functions (possibly on the same data). Of course in PVM both models can be mixed in a hybrid that exploits the strengths of each machine.

There are additional considerations about networking for the application developer if he wishes to run his parallel application over a network of machines. His parallel program will be sharing the network with other users. This multiuser, multitasking environment affects both the communication and computational performance of his program in complex ways.

First, there is different computational power on each machine in the configuration. Second, there are the effects of long message latency across the network. Third, the computational performance and effective network bandwidth are dynamically changing as other users share these resources. Many of these network considerations are taken care of by incorporation of some form of load balancing into a parallel application.

Definition 3.8 *An application A is a 4-tuple $\{P, G, f, e\}$. where P is a set of n processors; $G = (\Gamma, \Delta)$ is an undirected graph; Γ is a set of l processes; Δ is a set of*

undirected edges corresponding to communication between processes; $f : \Gamma \times P \rightarrow T_0$ *is a function such that* $f(\gamma, p)$ *returns the cost required to compute task* $\gamma \in \Gamma$ *on processor* $p \in P$*;* $e : \Delta \times \Delta \rightarrow T_0$ *is a function returning the cost associated with communication between processes if they are mapped to different processors. The load balancing is to minimize the global cost [149] of* $\sum f(\gamma, p) + \sum e(\gamma, \delta)$.

In a multiuser network environment load balancing is the single most important performance enhancer. There are many load balancing schemes for parallel programs. We will describe the two most common schemes used in network computing [74].

The simplest method is *static load balancing.* In this method the problem is divided up, and tasks are assigned to processors only once. The data partitioning may occur off-line before the job is started, or the partitioning may occur as an early step in an application. The size of the tasks or the number of tasks assigned to a given machine can be varied to account for the different computational powers of the machines. Since all the tasks can be active from the beginning, they can communicate and coordinate with one another. On a lightly loaded network, static load balancing can be quite effective.

When the computational loads are varying, a *dynamic load balance* scheme is required. The most popular method is called the Pool of Tasks paradigm. It is typically implemented in a master/slave program where the master program creates and holds the pool and farms out tasks to slave programs as they fall idle. The pool is usually implemented as a queue and if the tasks vary in size then the larger tasks are placed near the head of the queue. With this method all the slave processes are kept busy as long as there are tasks left in the pool.

3.2.4. Monitoring and debugging a PVM Application

In general, debugging parallel programs is much more difficult than debugging serial programs. Not only are there more processes running simultaneously, but their interaction can also cause errors. While PVM provides a solid programming base, it does not provide the user with many tools for analyzing or debugging PVM programs. Xab (X-window Analysis and deBugging) [33] is a tool for the visual (X-based) analysis and debugging of PVM programs. Xab gives the user direct feedback as to what PVM functions his program is performing. In its simplest form, this feedback is displayed in a X-window. Xab uses PVM to monitor PVM programs. This makes Xab very portable but it leads to interesting issues of how to make Xab compatible with the programs it monitors.

Xab consists of three main components, a user library, a monitoring program and an X window front end. The user library provides instrumented versions of the PVM

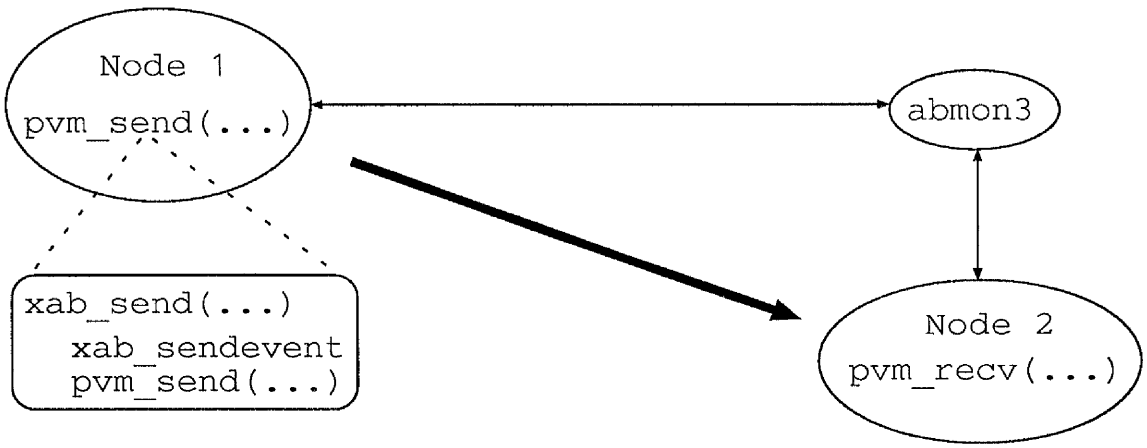

Fig. 9. Node 1 task is calling pvm_send to send a message to node 2 task. Node 1's pvm_send actually translates into an xab_send. The xab_send sends an event message to abmon3 and then performs the actual pvm_send on behalf of the program.

calls. The monitoring program runs as a PVM process and gathers monitor events in the form of PVM messages. The Xab front end displays information graphically about PVM processes and messages. The approach of real time monitoring is particularly apropos in a heterogeneous multiprogramming environment. Monitoring can help give the user insight into how a program is behaving in such an environment.

The Xab routines perform the normal PVM functions for the user but they also send PVM messages to a special monitoring process, called *abmon*, illustrated in Figure 9. The abmon process receives event messages from the instruction of PVM calls, and formats them into human readable form. The formatted event messages can either be written to a file or sent to the Xab display front end program.

3.3. Use of Tcl to Develop Interactive Application

A general-purpose programmable command language amplifies the power of software by allowing users to write programs in the command language in order to extend the software's built-in facilities. Among the best-known examples of powerful command languages are those of the unix *shell* [108] and *Emacs* editor [171].

Nowadays it is possible and easy to develop interactive applications on a personal workstation. Unfortunately, few of today's interactive applications have the power of the *shell* command languages. Where good command languages exist, they tend to be tied to specific programs. Each new interactive application requires a new command language to be developed. In most cases application programmers do not

have time or inclination to implement a general-purpose facility, so the resulting command languages tend to have insufficient power and clumsy syntax [150].

Tcl stands for "Tool Command Language" [150, 151, 152] which is general-purpose, embeddable, and powerful. It consists of a simple Tcl shell application called *tclsh* and a library package that programs can use as the basis for their command languages.

Tcl implements an interpreter for a simple programming language that provides variables, procedures, control constructs like *if* and *for*, arithmetic expressions, lists, strings and other features. Tcl also allows applications to extend the generic command set with application-specific commands. An application need only implement a few basic Tcl commands related to the application; when these are combined with the Tcl library a fully-programmable command language results.

3.3.1. Tcl language Syntax

Definition 3.9 *The Syntax of Tcl language is defined by Backus et al Form (BNF) [15] as the following :*

<Tcl-script>	::= <Tcl-command> <C-separator> <Tcl-command> ···
<C-separator>	::= ';' | 'newline-key'
<Tcl-command>	::= <Field> <Separator> <Field> <Separator> ···
<Separator>	::= 'space' | 'tab'
<Field>	::= <Word> | <L-syntactic-construct> <Word> | <Word> <R-syntactic-construct>
<L-syntactic-construct>	::= '[' | '{'
<R-syntactic-construct>	::= ']' | '}'
<Word>	::= <Command> | <Argument> | <Comment>
<Command>	::= <Built-in-command> | <Application-specific-command> | 'proc'
<Argument>	::= ASCII-string | '$' ASCII-string | ASCII-string '\special-character' ASCII-string
<Comment>	::= '#' ASCII-string

The angular brackets (<>) delimit meta-linguistic terms and the vertical bars (|) separate alternatives (read as 'or'). The double-colon equals (::=) is to be read as 'may be'.

Tcl's basic syntax is similar to that of the unix shell: a command consist of one or more fields separated by spaces or tabs. Unlike the unix shell, each Tcl command returns a string result, or the empty string if a return value isn't appropriate. There are four additional syntactic constructs in Tcl, which give the language a Lisp-like [1] flavor.

The following examples summarize a few of the key features of Tcl:

Example 1

```
set a 934
put a; set a b
```

Simple Tcl commands consist of words separated by white space. The first word is a command name (here is 'set' and the additional words are arguments for the command (here is '934'). 'set a 934' means that variable 'a' is set to a new value 934. 'put a' means that character 'a' is displayed on the screen. Commands are separated by semi-colons or newlines. □

Example 2

```
set msg "Hello, world"
set x {a b {x1 x2}}
```

Double-quotes or nested curly braces may be used to delimit complex arguments in Tcl commands. Each of the above commands has three fields in all. If an argument is enclosed in braces then the contents of the braces are passed to the command without any further interpretation (newline and semi-colons are not command separators and the substitutions described in Examples 3-5 are not performed). If an argument is enclosed in quotes, then the substitutions in Examples 3-5 are performed on its contents. □

Example 3

```
print $msg
if $i < 2 {set j 27}
```

Dollar signs invoke variable substitution in Tcl commands: the dollar sign and variable name will be replaced with the value of the variable in the argument passed to the command. □

Example 4

```
print [list q r $x]
set msg [format "x is %s" $x]
```

Tcl commands may contain other commands enclosed in brackets. When this occurs, the nested command is executed and its result is substituted into the argument of the enclosing command, replacing the bracketed command. □

Example 5

```
set msg "{ and [ are special"
print Hello1\n
```

Backslashes prevent special interpretation of characters like braces and brackets in Tcl commands. Backslashes can also be used to insert control characters into commands, as in the second command above. □

Tcl evaluates a command in two steps [152] : parsing and execution, as shown in Figure 10. In the parsing step the Tcl interpreter divides the command up into

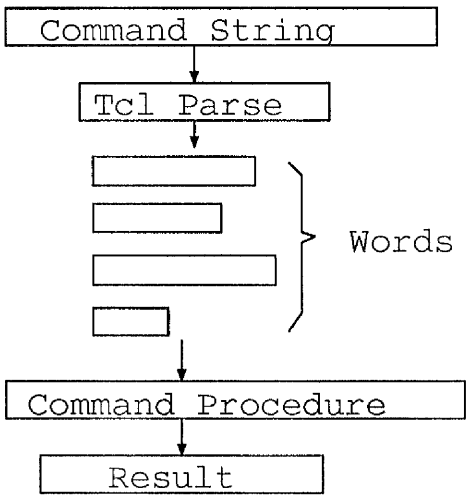

Fig. 10. Tcl Command Execute Flow

words and performs substitutions. Parsing is done in exactly the same way for every command. During the parsing step the Tcl interpreter does not apply any meaning to the values of the words. Tcl just performs a set of simple string operations such as replacing the characters "$a" with the string stored in variable *a*; Tcl does not know or care whether *a* or the resulting word is a number or anything else.

In the executing step meaning is applied to the words of the command. Tcl treats the first words as a command name, checking to see if the command is defined and locating a command procedure to carry out its function. If the command is defined then the Tcl interpreter invokes its command procedure, passing all of the words of the command to the command procedure.

3.3.2. Tcl Data Type

There is only one type of data in Tcl [152]: strings. All commands, arguments to commands, results returned by commands, and variable values are ASCII strings. The use of strings throughout Tcl makes it easy to pass information back and forth between Tcl library procedures and C code in the enclosing application.

Although everything in Tcl is a string, many commands expect their string arguments to have particular formats. There are three particularly common formats for strings [150]: list, expressions and commands.

1. A list is just a string containing one or more fields separated by white space, similar to a command. For example, the string

```
dog cat {horse cow mule} bear
```

is a list with three elements.

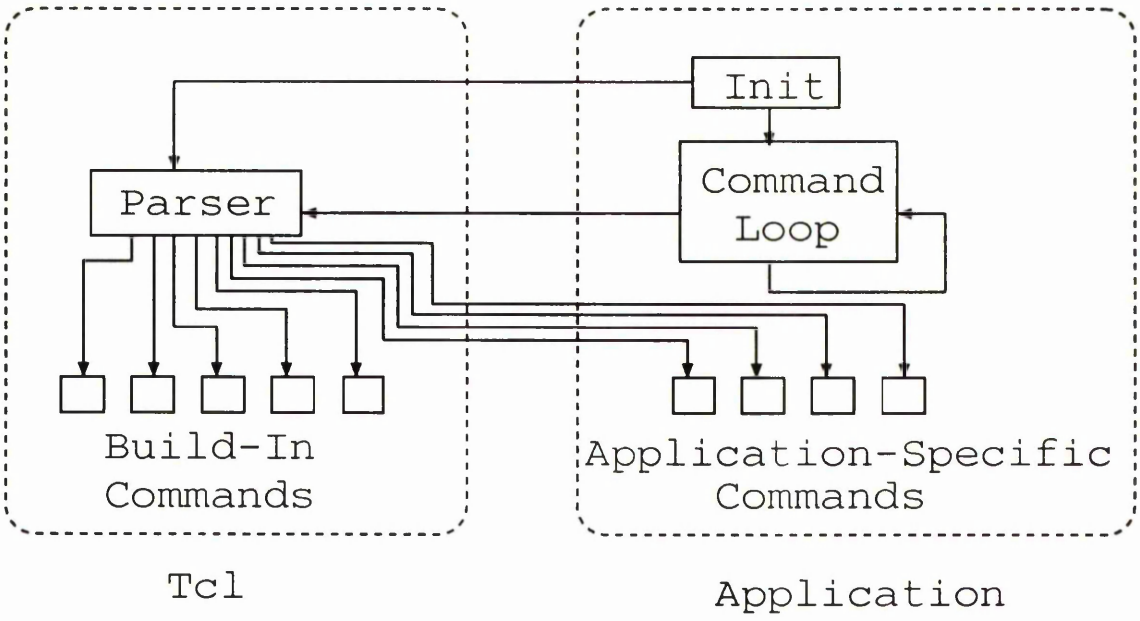

Fig. 11. Tcl Embeddable Structure

2. The second common form for a string is a numeric expression. Tcl expressions have the same operators and precedence as expressions in C.

3. The third common form for a string is as commands (or sequences of commands). Arguments of this form are used in Tcl commands that implement control structures. For example, consider the following command:

   Example 6

```
if { $a < $b } {
   set tmp $a
   set a $b
   set b $tmp
 }
```

   The *if* command receives two arguments here, each of which is delimited by curly braces. *If* is a built-in command that evaluates its first argument as an expression; if the result is non-zero, *if* executes its second argument as a Tcl command.

3.3.3. Embedding An Application into Tcl

Tcl is an embedded language [150]. It is a library that is designed to be linked together with C applications as shown in Figure 11. The main loop of the application generates Tcl commands. This could happen in any of serveral ways, depending on

the application. One way is to read commands from standard input; this results in a shell-like program. Another way, used by Tk, is to associate Tcl commands with X events such as button presses or keystrokes; when an X event occurs, the corresponding commands are executed. When the application has generated a Tcl command it passes it to a Tcl library procedure for evalution. The Tcl interpreter parses the command, performs the substitutions described in Examples 2-5, uses the first word of the command to locate a command procedure for the command, and then calls the command procedure to actually execute the command. The command procedure carries out its function and returns a string result, which the Tcl interpreter returns back to the calling code in the application.

The Tcl library includes several built-in commands that implement the generic facilities [152] such as variables and looping. Additional command procedures may be provided by each application. The application registers its own specific commands by passing their names and command procedures to Tcl. This information is used later by the Tcl interpreter when it evaluates command strings. Application-specific and built-in commands have exactly the same structure; they are indistinguishable except that built-in commands are registered automatically and users may expect them to be present in all applications. New commands may be created and deleted at any time while an application executes.

The most important aspects of Tcl are the simplicity of the language and the simplicity of its interface to C programs. The language simplicity makes Tcl easy to learn; the interface simplicity makes it easy to use Tcl in applications, easy to write new Tcl commands, and easy to use Tcl to compose primitives written in C.

3.3.4. Tk — Extending Tcl into X11 Window System

Tk is a new toolkit for the X11 window system [168]. Like other X11 toolkits such as Xt [12], Tk consists of a set of C library procedures intended to simplify the task of constructing windowing applications. The Tk library procedures, like those of other toolkits, serve two general purposes [151]: framework and convenience. First, they provide a framework that allows applications to be built out of many small interface elements called *widgets* (e.g. buttons, scrollbars, menus, etc.). The toolkit's framework makes it possible to design widgets independently, compose them into interesting applications, and re-use them in many different situations without re-design. The second purpose of the toolkit is to provide ready-made solutions for the most common needs of windowing applications. For example, Tk includes a set of commonly used widgets plus procedures to make it easy to build new widgets. Using Tk, it is possible to build many interesting windowing applications by plugging

together existing widgets. Many other applications can be built by constructing one or two new widget types and combining them with Tk's existing widgets.

Although Tk's overall purpose is similar to that of other toolkits, its implementation has the unusual property that it is based around the Tcl command language. The Tcl interfaces allow the look and feel of an application to be queried and modified at any point in the application's execution. They also allow new interface elements, or even new applications, to be created dynamically just by writing Tcl script. C code is needed only for creating new widget types or data structures.

Each widget/window has a textual name [151] that is used to refer to it in Tcl commands. Window names are similar to the hierarchical path names used to name files in Unix, except that "." is used as the separator character instead of "/". The name "." refers to the topmost window in the hierarchy, which is called the main window.

Tk applications are controlled by two kinds of Tcl scripts [152]: an initialization script and event handlers. The initialization script is executed when the application starts up. It creates the application's user interface, loads the application's data structures, and performs any other initialization needed by the application. Once initialization is complete, the application enters an X event loop to wait for user interactions. Whenever an interesting X event occurs, such as the user invoking a menu entry or moving the mouse, a Tcl script is invoked to process that event. These scripts are called event handlers; they can invoke application specific Tcl commands, modify the user interface, or do many other things.

Wish is the simplest possible Tk application. The only Tcl commands it contains are the Tcl built-ins and the additional commands provided by Tk. The following is the famous "Hello, world" example.'

Example 7

```
button .b -text "Hello, world!" -command "destroy ."
pack .b
```

Type the above commands to *wish* and the produced application is shown in Figure 12.

Tk provides four main groups of Tcl commands; they create widgets, arrange widgets on the screen, communicate with existing widgets, and interconnect widgets within and between applications. Whenever a new widget is created Tk also creates a new Tcl command whose name is the same as the widget's name. This command is called a widget command, and the set of all widget commands (one for each widget in the application) constitutes the third major group of Tk's commands.

The most important feature [151] of Tk is that it allows different applications to work together in powerful ways. Tk provides a remote-procedure-call-like facility

Fig. 12. Tk Implementation of the Example "Hello, World"

called *send* that allows any Tk-based application to invoke Tcl commands in any other Tk-based application. Send takes two arguments: the name of an application and a Tcl command. This facility encourages the development of lots of small specialized tools that can be programmed with send to work together in interesting ways. The tools could be developed and maintained independently, and yet be used in many different ways. It could result in much richer and more powerful interactive environments than we have today.

3.4. Design Interactive Parallel Distributed Computing Environment

In PVM version 3, there are the C language and Fortran 77 interface libraries. In order to give PVM the facility of interactive application, we do not want to invent a new wheel, we just use the powerful tool language Tcl and bind Tcl with PVM C language interface library, called as Interactive Parallel Distributed Computing Environment (IPDCE).

There are two ways of implementation. One is only to use basic Tcl and the other is to use Tcl+Tk as shown in Figure 13. The advantage of the first is that we do not need to use an X environment which is very large in size and slow in some old machines. We can do interactive parallel computing character-based applications. The advantage of the second is we can use Graphics User Interface (GUI) under X window, which is the de facto standard of GUI. With Tk you can develop GUI beautifully for your interactive parallel computing application.

In binding Tcl with the PVM C language interface library, the first consideration is the names of library procedures. We use the same approach to change the character '_' to 'I' in the name of its corresponding PVM C library procedure, as in the example

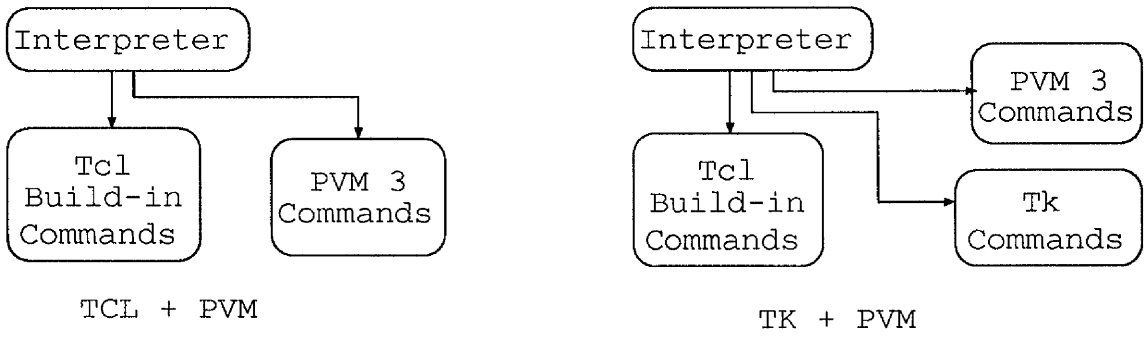

Fig. 13. Bind Tcl or Tk with PVM

PVM C procedure name	Tcl PVM procedure name
pvm_mytid	pvmImytid
pvm_send	pvmIsend

The second consideration involves the arguments of library procedures. As in the C language, we use the value-pass method in IPDCE. As we know, we can simulate the reference-pass method by a pointer in C language, but we do not use the approach in IPDCE. Instead of a pointer, we use the LIST structure as argument to pass input values into a procedure and return a new LIST structure from the procedure to return result values.

Example 8

```
int info pvm_config( int *nhost, int *narch, struct hostinfo
                     **hostp );              in PVM C library
RETLIST pvmIconfig;                          in IPDCE library
```

where RETLIST = info nhost narch hostlist; info, nhost, narch are integer; hostlist is host-name list. □

In the C version of the procedure, the arguments *nhost* and *narch* are pointers and *hostp* is the pointer's pointer. They are used to pass result value.

Each Tcl command is represented by a command procedure written in C. The interface to a command procedure is defined by the *Tcl_CmdProc* procedure prototype:

```
typedef int Tcl_CmdProc(ClientData clientData, Tcl_Interp *interp,
                        int argc, char *argv[] );
```

Each command procedure takes four arguments. The first, *clientData*, is useful when the command is associated with object-oriented style programming. The second, *interp*, is the interpreter in which the command was invoked. The third and

fourth arguments have the same meaning as the *argc* and *argv* arguments in a C main program: *argc* specifies the total number of words in the Tcl command and *argv* is an array of pointers to the values of the words. A command procedure returns two values. One is an integer completion code (e.g. *TCL_OK* or *TCL_ERROR*) and the other is a result string or error message in *interp− > result*.

The following is the command procedure for a new command called *PVM_mytid* which enrols your process in PVM.

Example 9

```
int PVM_mytid(clientData, interp, argc, argv)
ClientData clientData; Tcl_Interp *interp; int argc; char *argv[];
{
    register int tid;
    if( argc != 1 )
    {   interp->result = "wrong # args"; return TCL_ERROR;  }
    if( (tid=((PVM_XAB3) ? xab_mytid() : pvm_mytid()) ) < 0 )
    {   sprintf(interp->result,"%d",tid); return TCL_ERROR; }
    else
    {   sprintf(interp->result,"%d",tid); return TCL_OK;    }
}
```

In order for a command procedure to be invoked by Tcl, we must register the new command by calling *Tcl_CreateCommand*. For example

Example 10

```
Tcl_CreateCommand(interp, "pvmImytid", PVM_mytid,(ClientData *)NULL,
                  (Tcl_CmdDeleteProc *)NULL );
```

The first argument to *Tcl_CreateCommand* identifies the interpreter in which the command will be used. The second argument specifies the name for the command and the third argument specifies its command procedure. The fourth and fifth arguments are related to *ClientData*, which is not used in IPDCE design. *Tcl_CreateCommand* will create a new command for *interp* named *pvmImytid*. Whenever *pvmImytid* is invoked in *interp*, Tcl will call *PVM_mytid* to carry out its function. After the above call to *Tcl_createCommand*, *pvmImytid* can be used in TCL script just like any other command.

Following the approach described above, we can design the whole PVM 3 Tcl interface from its C library. Here are all the Tcl command names:

Process Control: pvmImytid, pvmIexit, pvmIspawn, pvmIkill.

Information: pvmIparent, pvmIpstat, pvmIpvmImstat, pvmIconfig, pvmItasks, pvmIgetopt, pvmItidtohost.

Dynamic Configuration: pvmIaddhosts, pvmIdelhosts, pvmIhalt, pvmIstart_pvmd.

Signalling: pvmIsendsig, pvmInotify.

Error Messages: pvmIperror, pvmIserror.

Message Buffers: pvmImkbuf, pvmIinitsend, pvmIfreebuf, pvmIgetrbuf, pvmIgetsbuf, pvmIsetsbuf, pvmIsetrbuf.

Packing Data: pvmIpkbyte, pvmIpkcplx, pvmIpkdcplx, pvmIpkdouble, pvmIpkfloat, pvmIpkint, pvmIpklong, pvmIshort, pvmIpkstr

Sending and Receiving Data: pvmIsend, pvmImcast, pvmInrecv, pvmIrecv, pvmIprobe, pvmIbufinfo, pvmIrecvf, pvmIadvise, pvmIfvend, pvmIvrecv, pvmIvbufinfo.

Unpacking Data: pvmIupkbyte, pvmIupkcplx, pvmIupkdcplx, pvmIupkdouble, pvmIupkfloat, pvmIupkint, pvmIupklong, pvmIupkshort, pvmIupkstr.

Dynamic Group: pvmIjoingroup, pvmIlvgroup, pvmIgettid, pvmIgetinst, pvmIgetgsize, pvmIbarier, pvmIbcast.

We also provide the support of Xab version 3 in IPDCE. Besides the original function of Xab 3, users can dynamically set on or off the Xab. There are two new Tcl commands "xabIon" for starting the Xab and "xabIoff" for ending the Xab.

One problem in IPDCE is that you cannot use binary type data directly since only string type data is officially supported in Tcl. Actually in all packing-data and unpacking-data commands the input from Tcl script are string type; for example, "-13.5e10" is a string expression of a floating point number. If the size of data which requires transfer to another machine is small, we can use the string type to communicate with the other machine. But if the size of data is large, we cannot use the string type since there are very high overheads compared with string expressions and original binary expressions. For example, "-13.5e10" needs 9 bytes as a string which also includes a end mark '\0' of C string style, but the original expression only uses 4 bytes if it is floating-point type or 8 bytes if it is double-precision type.

Although Tcl does not allow a user to define any new data type, it provides an object-oriented style data-save structure, the hash table. A hash table is a collection of entries, where each entry consists of a key and a value. No two entries have the same key. Given a key, a hash table can very quickly locate its entry and hence the associated value. Tcl exports its general-purpose hash table facilities through a set of C library procedures so that applications can use them. In Tcl's hash table, the values for hash table entries are items of type ClientData, which are large enough to hold either an integer or a pointer.

In order to solve the problem of no binary type in IPDCE, we define a new C structure as

```
typedef struct GBox_ {  unsigned long int total_size, cur_size,
                                          view_pos, width;
                char *b_array;      } GBOX;
```

where *b_array* is a binary character array, *total_size* is the total size of the *b_array*, *cur_size* is the current size of *b_array*, *view_pos* is the current position of view point and *width* is the width of two-dimensional data array. Several Tcl commands are implemented for processing a GBOX data structure. *gbIcreate* produces a new GBOX object, *gbIdestroy* destroys a GBOX object, *gbIstate* states the status of a GBOX object, *gbIpush* pushes a set of string type data into a GBOX object, *gbIpop* pops a set of string type data from a GBOX object, *gbIview* displays value of a part of a GOBX object, *gbIseek* seeks a new view position in a GBOX object, *gbIfread* creates a new GBOX and reads data from a file into the GBOX, *gbIfwrite* writes data of a GBOX into a file and destroys the GBOX object.

The first thing to do is define a new hash table. For example,

Example 11

```
Tcl_HashTable GBoxTable;
...
Tcl_InitHashTable(&GBoxTable, TCL_STRING_KEYS);
```

The second stage is to create an entry with a given key, and *Tcl_SetHashValue* sets the value associated with the entry. For example,

Example 12

```
int GBox_create(clientData, interp, argc, argv)
 ...
     do { sprintf(interp->result, "gbox%d", id);       id++;
       entryP = Tcl_CreateHashEntry(&GBoxTable, interp->result,
                                    &new);
     }   while( !new);
     if( (gbp = (GBOX *)malloc(sizeof(GBOX))) == NULL )
     { interp->result = "wrong # no mem space"; return TCL_ERROR; }
     Tcl_SetHashValue(entryP, gbp);
     ...
     return TCL_OK;
 }
```

The third stage is to find an entry with the procedure *Tcl_FindHashEntry*. *Tcl_FindHashEntry* is typically used to find an object given its name. For example,

Example 13

```
int GBox_destroy(clientData, interp, argc, argv)
...
```

```
    for( i=1; i<argc; i++ )
    {   entryP = Tcl_FindHashEntry(&GBoxTable, argv[i]);
        if( entryP == NULL )                    continue;
        gbp = (GBOX *)Tcl_GetHashValue(entryP);
        Tcl_DeleteHashEntry(entryP);
        free(gbp->array);                    free(gbp);
      }
    return TCL_OK;
}
```

The last stage is to delete an entry with the procedure *Tcl_DeleteHashEntry*, as shown in the above example.

GBOX can be read/written from/to file with *gbIfread/gbIwrite* commands. The format of the file is ppm [159] for two-dimensional data and modified ppm [1] for one-dimensional data.

[1] See Appendix A

CHAPTER 4

THE STACK FILTERS, MINIMUM THRESHOLD DECOMPOSITION AND INTERACTIVE STACK FILTERING SYSTEM

4.1. Introduction

In this chapter we present a new procedure which uses minimum threshold decomposition and the positive Boolean function to realize stack filtering. In order to reduce the time complexity of stack filters, we try to minimize the number of logical operations and use the CPU bit-fields parallel method to do stack filtering. A full parallel algorithm based on the new procedure and the data parallel scheme has been implemented. Under the Interactive Parallel Distributed Computing Environment (IPDCE) we develop a powerful, Interactive Stack Filtering System, which provides beautiful Graphics User Interface (GUI), one- and two-dimensional stack filtering procedures and convenient selection of series and parallel algorithms. We apply two numeric examples to the stack filter and the results show that the interactive parallel stack filtering system is efficient for both sequential and parallel filtering algorithm.

4.2. Stack Filters Based on Threshold Decomposition

Consider a signal $\vec{X} = (X_1, \cdots, X_N)$, where each $X_i \in \{0, 1, \cdots, 2^{M-1}\}$. The threshold decomposition property of $\vec{X}$ can be defined [62, 63] by

$$x_i^l = T_l(X_i) = \begin{cases} 1 & \text{if } X_i \geq l \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

$$\text{for } l = 1, 2, \cdots, 2^{M-1}. \tag{4.2}$$

Given two binary signals, $\vec{u}$ and $\vec{v}$, a property called the *stacking property* holds between $\vec{u}$ and $\vec{v}$ if and only if $u_k \geq v_k$ for all k. Suppose $\vec{u}$ and $\vec{v}$ are filtered with a binary filter, of window width b, defined by a Boolean function $f : \{0,1\}^b \rightarrow \{0,1\}$. The binary filter f is said to possess the stacking property if and only if

$$f(\vec{u}) \geq f(\vec{v}). \tag{4.3}$$

In other words, if the binary output signals are piled on top of one another according to their threshold level, the result is a column of 0's piled on top of a column of 1's.

x_1	x_2	x_3	$x_1x_2 + x_1x_3 + x_2x_3$	$\text{med}(x_1, x_2, x_3)$
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	1
1	0	0	0	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

Table IV. Detailed explanation of the MSP form of the PBF for the third-order binary median filter

It has been shown[81] that a necessary and sufficient condition for a Boolean function to satisfy the stacking property is that the function be a Positive Boolean Function PBF, i.e., no complement of any of the input variables must appear in the minimum-sum-of products (MSP) form of the function. For example, the third-order binary median is

$$f(x_1, x_2, x_3) = x_1x_2 + x_1x_3 + x_2x_3, \tag{4.4}$$

where multiplication denotes logical **AND** and addition denotes logical **OR**. The details of this are listed in Table IV. The function (4.4) is a PBF, but the following example is not,

$$f_1(x_1, x_2, x_3) = \bar{x}_1x_2 + x_3x_1, \tag{4.5}$$

since $\bar{x}_1x_2$ includes the complement, $\bar{x}_1$ of variable x_1.

Definition 4.1 *A stack filter $f_S(\cdot)$ of window width $b = 2p+1$ is based on a b-variable positive Boolean function PBF $f(\cdot) : \{0,1\}^b \rightarrow \{0,1\}$ operating on the binary signals. The output of the stack filter is obtained by adding all the binary outputs:*

$$\begin{aligned} f_S(\vec{X}_i) &= S_f\left(\sum_{l=1}^{M} T_l(\vec{X}_i)\right) = \sum_{l=1}^{M} S_f\left(T_l(\vec{X}_i)\right) \\ &= \sum_{l=1}^{M} f\left(T_l(\vec{X}_i)\right) = \sum_{l=1}^{M} f\left(\vec{x}_i^l\right) \qquad (4.6) \\ &= \max\{l \mid f(x_i^l) = 1, l \in \{1, 2, \cdots, 2^{M-1}\}\}, \qquad (4.7) \end{aligned}$$

where $\vec{X}_i = \{X_{i-p}, \cdots, X_i, \cdots, X_{i+p}\}$.

The above equations reveal that the stack filtering algorithm is composed of three steps. The first is that of threshold decomposition, the second involves Boolean function logic operations, and the third involves accumulation of all the binary results generated within the second step; equivalently, this is a search for the highest level at

which the Boolean logical output, generated within the second step, is 1. The general stack filter algorithm can be expressed as follows:

Algorithm 4.1 : Original stack filter algorithm

1. For each signal X_i, apply threshold decomposition equation (4.2) to get the binary signal x_i^l.
2. For each binary signal, apply the PBF $f_S(\vec{X}_i)$ to get the binary output value r_i^l.
3. For all binary outputs for position i, accumulate them to get the stack filter output $f_S(\vec{X}_i)$.

The computational complexity of this algorithm is very high since the number $(2^M - 1)$ of threshold decompositions grows exponentially with the number bits (M) associated with the signal value. Lin et al [119] use weighted order statistic filters based on threshold logic instead of stack filters since there is equivalence between linear separable Boolean functions and threshold logic. Kar [107] suggested an algorithm which transforms a given sequence to equivalent-rank-preserving sequences through bit manipulation. This reduces the problem of finding a rank-order selection for a k-bit-long number to finding out k rank-order selections for '1'-bit-long numbers.

4.3. Minimum Threshold Decomposition of Signal

In order to speed up the stack filtering algorithm, the first thing we should do is to reduce the number of threshold decomposition levels, since the comparison operation that underlies threshold decomposition is rather slow, relative to logic and arithmetic operations. From the theory of data retrieval, the minimum operation time for picking out a particular one of a set of N values, is $\log_2 N$. The search scheme is known as binary search in which at each step a midpoint value is examined to find out in which direction to continue.

Based on the stacking property of the output of stack filters, we can define the search procedure as follows :

Definition 4.2 *For an M-bits input signal, search each binary output value r^j from the set of $\{r^{M-1}2^{M-1}, r^{M-2}2^{M-2}, \cdots, r^1 2, r^0\}$ using a binary searching scheme, where r^j, $j = 0, 1, \cdots, M-1$, are binary and $r^j = 0$ means that the next threshold level will decrease and $r^j = 1$ means that the next threshold level will increase. The output of the stack filter is $\sum_{j=0}^{M-1} r^j 2^j$.*

In company with the above output searching, we can define a new Minimum Threshold Decomposition (**MTD**) which also uses a binary search method to determine a new threshold decomposition value.

Definition 4.3 *For an M-bits input signal, there are M level threshold decompositions. They are*

$$\begin{aligned} T^{M-1} &= 2^{M-1} \\ T^j &= \sum_{l=j+1}^{M-1} r^l 2^l + 2^j, \quad j = M-2, \cdots, 0. \end{aligned} \tag{4.8}$$

In order to present the new method of minimum threshold decomposition, let us describe a simple example of a median filter. Consider a 3 third-order median filter, where each datum belongs to the set $\{0, 1, 2, 3\}$, and $M = 2$. Suppose that the data in the window are 1, (2), 0 and that the current filtering position is at 2, as indicated by the parentheses. Initially, the threshold decomposition level is $2^{M-1} = 2$, the mid-range of [0,4). Applying threshold decomposition using logical **AND** with the datum written as $(10)_2$ in binary form, we get binary data 0, (1), 0. The result of median filtering at this level is $r^1 = 0$, which means the next direction of search is to the lower half of the range. The new threshold is set to $r^1 2^1 + 2^0 = 1$. Applying threshold decomposition using logical **AND** with datum $(01)_2$ we get the binary input data 1, (0), 0. The central binary value is not obtained correctly directly from the logical AND operation since the original datum, 2, is greater than threshold 1 for ever. The correct binary data are 1, (1), 0. We therefore need a logical variable to record each state that will subsequently always be greater than the threshold. Similarly we need a variable to record each state that will subsequently always be less than the threshold. For this level the median filter's result is 1. Finally, the maximum threshold level is 1 and its binary median filter result is 1. Thus, the output of the median filter in this position is 1.

In each MTD level, define two new logic variable 'lt' and 'gt' to record the MTD state of the current datum; $lt = 1$ & $gt = 0$ means the datum is less than the threshold for ever, $gt = 1$ & $lt = 0$ means the datum is greater than the threshold for ever, and $lt = 0$ & $gt = 0$ means that the current threshold decomposing value only relates to the current datum.

Definition 4.4 *For each MTD level j, $j = M-1, M-2, \cdots, 0$, the variables lt^j and gt^j are*

$$\begin{aligned} lt^{M-1} &= gt^{M-1} = 0 \\ \overline{lt}^j &= gt^{j+1} \mid (\overline{lt}^{j+1} \;\&\; (\bar{r}^j \mid x_i^j)) \\ gt^j &= \overline{lt}^{j+1} \;\&\; (gt^{j+1} \mid (\bar{r}^j \;\&\; x_i^j)), \; j = M-2, \cdots, 0, \end{aligned} \tag{4.9}$$

where we use notation of C language, "|" means logical **OR**, *"&" means logical* **AND**.

Definition 4.5 *The MTD is*

$$x_i^j = \overline{lt}^{j+1} \ \& \ (b_i^j \mid gt^{j+1}), \tag{4.10}$$

where the input signal X_i *is* $b_i^{M-1}b_i^{M-2}\cdots b_i^0$, *and* $b_i^j, j = M-1,\cdots,1,0$ *are binary.*

The new algorithm based on MTD can be expressed as follows :

Algorithm 4.2 : Stack filter algorithm based on minimum threshold decomposition.

1. For each signal X_i, apply MTD equation (4.10) to get x_i^j.
2. For each x_i^j, apply the PBF $f_S(\vec{X}_i)$ to get binary output value r_i^l.
3. Calculate the new states of variables lt^{j-1} and gt^{j-1} using equation (4.9) for the next threshold decomposition level.
4. After finishing M-level MTD, calculate the output of the stack filter at position i by $\sum_{j=0}^{M-1} r^j$.

Theorem 4.1 *Given inputs* $\{X_i\}$, $i = 0,1,\cdots,N$ *and a positive Boolean function* $f_S(\cdot)$, *the above algorithm produces the same result as the original stack filter.*

Proof : Since the difference between algorithm 4.1 and algorithm 4.2 is the threshold decomposition procedure, we need only to prove that the new MTD procedure works.

Let $X_i = b^{M-1}b^{M-2}\cdots b^1b^0$, and let lt^k, gt^k be variables to express the state of threshold decomposition of current datum X_i. Suppose in threshold decomposition level $M-1, M-2,\cdots,j+1$, variable $lt^k = gt^k = 0$, $k = M-1,\cdots,j+1$. Now in new level j, the variables will change their states.

Case 1: Variable $lt^j = 1$. Since in level $j+1$ the variables $gt^{j+1} = 0$ and $lt^{j+1} = 0$, equation (4.9) can be simplified as

$$lt^j = r^j \ \& \ \bar{x}_i^j. \tag{4.11}$$

Because $lt^j = 1$, we have $r^j = 1$ and $x_i^j = 0$. $x_i^j = 0$ means that the threshold decomposition value is zero. We can express it in the original form (4.2)

$$X_i = b^{M-1}2^{M-1} + b^{M-2}2^{M-2} + \cdots + b^0 < T^j. \tag{4.12}$$

According to definition 4.3, $r^j = 1$ means that the next threshold value T^{j-1} increases. Since $T^{j-1} = T^{j+1} + 2^j + 2^{j-1}$, we have

$$X_i = b^{M-1}b^{M-2}\cdots b^jb^{j-1}\cdots b^0 < T^j < T^{j-1}. \tag{4.13}$$

The current datum X_i will be less than threshold value T^k, $k = j-1, j-2, \cdots, 1, 0$ for ever. This proves that the MTD scheme works in this case.

Case 2: Variable $gt^j = 1$. Since in level $j+1$ the variables $gt^{j+1} = 0$ and $lt^{j+1} = 0$, equation (4.9) can be simplified as

$$gt^j = \bar{r}^j \ \& \ x_i^j. \tag{4.14}$$

Because $gt^j = 1$, we have $r^j = 0$ and $x_i^j = 1$. $x_i^j = 1$ means that the threshold decomposition value is one. We can express it in the original form (4.2)

$$X_i = b^{M-1}2^{M-1} + b^{M-2}2^{M-2} + \cdots + b^0 > T^j. \tag{4.15}$$

According to definition 4.3, $r^j = 0$ means that the next threshold value T^{j-1} decreases. Since $T^{j-1} = T^{j+1} + 2^{j-1}$ we have

$$X_i = b^{M-1}b^{M-2}\cdots b^j b^{j-1}\cdots b^0 > T^j > T^{j-1}. \tag{4.16}$$

The current datum X_i will be more than threshold value T^k, $k = j-1, j-2, \cdots, 1, 0$ for ever. This proves that the MTD scheme works in this case too. □

4.4. The Positive Boolean Function and its Minimum Logical Operations Formula

We know that a positive Boolean function (PBF) has a unique minimum sum-of-products (MSP) form [140], but the number of logical operations associated with this form increases very quickly. For example, the PBF for the third-order median filter is

$$f_{med}(x_1, x_2, x_3) = x_1x_2 + x_1x_3 + x_2x_3, \tag{4.17}$$

which represents only $3 * 1 + 2 = 5$ logical operations. For the fifth-order median filter, the PBF is

$$\begin{aligned} f_{med}(x_1, x_2, x_3, x_4, x_5) = \quad & x_1x_2x_3 + x_1x_2x_4 + x_1x_2x_5 + x_1x_3x_4 + x_1x_3x_5 + x_1x_4x_5 + \\ & x_2x_3x_4 + x_2x_3x_5 + x_2x_4x_5 + x_3x_4x_5 \end{aligned} \tag{4.18}$$

and the number of logical operations increases to $10*2+9 = 29$. In general, we state the following theorem.

Theorem 4.2 *For the $(2N+1)$th order binary median filter, the MSP form of its PBF is composed of $\binom{2N+1}{N}$ items and the number of logical operations involved is $N * \binom{2N+1}{N} + \binom{2N+1}{N} - 1$.*

Proof: We know that there are two states, 1 and 0, for each binary datum. From the definition of the median, the condition for the median of a set of $2N+1$ data to be 1 is equivalent to the condition that there are at least $N+1$ 1's in the set of data. The identities of a set of $N+1$ data can be written in the Boolean logical way as $x_{p_1}x_{p_2}\cdots x_{p_{N+1}}$, and these aggregate to form a Sum of Products (SP). The total number of "products" in such an SP is clearly $\binom{2N+1}{N+1} = \binom{2N+1}{N}$. We notice that if any one of the components of the SP is 1, the median is 1, so the Boolean function of the $2N+1$ dot binary median filter can be expressed, using logical **OR**, as an SP.

Each term in the SP involves N logical **AND** operations, and there are $\binom{2N+1}{N}$ such terms, linked by logical **OR** operations. The total number of logical operations within the Boolean function is therefore $N * \binom{2N+1}{N} + \binom{2N+1}{N} - 1$

□

In order to reduce the number of logical operations, we can rewrite the MSP in another way which identifies the minimum number of logical operations. First, let us consider some simple examples. For the third-order median filter, the PBF of Equation (4.17) can be rewritten as

$$f_{med}(x_1, x_2, x_3) = x_1(x_2 + x_3) + x_2x_3. \tag{4.19}$$

Obviously, the number of logical operations is 4. For a fifth-order median filter, the PBF of Equation (4.18) can be rewritten as

$$\begin{aligned} f_{med}(x_1, x_2, x_3, x_4, x_5) &= x_1(x_2(x_3 + x_4 + x_5) + x_3(x_4 + x_5) + \\ &\quad x_4x_5) + x_2(x_3(x_4 + x_5) + x_4x_5) + x_3x_4x_5 \end{aligned} \tag{4.20}$$

We conjecture that, in order to achieve the minimum number of logical operations, we must adopt the following scheme of evaluation:

$$\begin{aligned} s &= x_4x_5 \\ m &= x_4 + x_5 \\ M_0 &= x_3m + s \\ M_1 &= x_2(x_3 + s) + m_0 \\ f_{med}(x_1, x_2, x_3, x_4, x_5) &= x_1M_1 + x_2M_0 + x_3m. \end{aligned} \tag{4.21}$$

From this scheme, the number of the Boolean logical operations is seen to be 12 compared with 29 using the basic MSP expression.

For the general $(2N+1)$th order median filter, we can similarly rewrite the PBF, as in the following theorem.

Theorem 4.3 *The PBF of a $(2N+1)$th order binary median filter can be expressed as*

$$
\begin{aligned}
f_{med}(x_1, x_2, \cdots, x_{2N+1}) = x_1(\ & x_2(\cdots(\ x_{N-1}[\ x_N(x_{N+1} + x_{N+2} + \cdots + x_{2N+1}) + \\
& \qquad x_{N+1}(x_{N+2} + x_{N+3} + \cdots + x_{2N+1} + \\
& \qquad \vdots \\
& \qquad x_{2N}x_{2N+1}] + \\
& \quad x_N[\ x_{N+1}(x_{N+2} + x_{N+3} + \cdots + x_{2N+1}) + \\
& \qquad x_{N+2}(x_{N+3} + x_{N+4} + \cdots + x_{2N+1} + \\
& \qquad \vdots \\
& \qquad x_{2N}x_{2N+1}] + \\
& \quad \vdots \\
& \quad x_{2N-1}x_{2N}x_{2N+1}) + \\
& \vdots \\
& x_{N+2}x_{N+3}\cdots x_{2N+1}) + \\
x_2(\ & x_3(\cdots(\ x_N[\ x_{N+1}(x_{N+2} + x_{N+3} + \cdots + x_{2N+1}) + \\
& \qquad x_{N+2}(x_{N+3} + x_{N+4} + \cdots + x_{2N+1} + \\
& \qquad \vdots \\
& \qquad x_{2N}x_{2N+1}] + \\
& \quad \vdots \\
& \quad x_{2N-1}x_{2N}x_{2N+1}) + \\
& \vdots \\
& x_{N+2}x_{N+3}\cdots x_{2N+1}) + \\
\vdots & \\
x_N(\ & x_{N+1}(\ \cdots(\ x_{2N-2}(\ x_{2N-1}(x_{2N} + x_{2N+1}) + \\
& \qquad x_{2N}x_{2N+1}) + \\
& \quad x_{2N-1}x_{2N}x_{2N+1}) + \\
& \cdots) + \\
& x_{N+2}x_{N+3}\cdots x_{2N}x_{2N+1}) + \\
x_{N+1}x_{N+2}\cdots & x_{2N+1}.
\end{aligned}
\tag{4.22}
$$

Proof: Within formula (4.22), none of items is repeated so that, if we prove that the number of items in the formula is equal to $\binom{2N+1}{N}$, as required by Theorem 4.2, we can conclude that we have proved this theorem. We use the method of induction.

Step 1. In the inner-most layer of parentheses of the first row of the formula, $x_{N+2} + x_{N+3} + \cdots + x_{2N+1}$ involves $N+1$ items. In the next layer, the number of items is

$$(N+1) + N + \cdots + 2 + 1 = \frac{1}{2}(N+1)(N+2). \tag{4.23}$$

In the third layer, the number of items is

$$\begin{aligned}
& (\frac{1}{2}(N+1)(N+2)) + (\frac{1}{2}(N)(N+1)) + \cdots + 3 + 1 \\
= \; & [\frac{1}{3*2}(N+1)(N+2)(N+3) - \frac{1}{3*2}(N)(N+1)(N+2)] \\
& + [\frac{1}{3*2}(N)(N+1)(N+2) - \frac{1}{3*2}(N-1)(N)(N+1)] \\
& + \cdots + [\frac{1}{3*2}(2*3*4) - \frac{1}{2*3}(1*2*3)] + [\frac{1}{2*3}(1*2*3) - 0] \\
= \; & \frac{1}{3*2}(N+1)(N+2)(N+3).
\end{aligned} \tag{4.24}$$

Step 2. Suppose that, in the pth layer of parentheses, the number of items is

$$\frac{1}{p*(p-1)*\cdots*3*2}(N+1)(N+2)\cdots(N+p). \tag{4.25}$$

Step 3. In the next layer of parentheses, the number of items is

$$\begin{aligned}
& [\frac{1}{p*(p-1)*\cdots*3*2}(N+1)(N+2)\cdots(N+p)] + \\
& [\frac{1}{(p-1)*(p-2)*\cdots*2}(N+1)(N+2)\cdots(N+p-1)] + \cdots + 1 \\
= \; & [\frac{1}{(p+1)!}(N+1)\cdots(N+p)(N+p+1) - \frac{1}{(p+1)!}(N)\cdots(N+p-1)(N+p)] + \\
& [\frac{1}{(p+1)!}(N)\cdots(N+p-1)(N+p) - \frac{1}{(p+1)!}(N-1)\cdots(N+p-2)(N+p-1)] \\
& + \cdots + 1 \\
= \; & \frac{1}{(p+1)*p*(p-1)*\cdots*3*2}(N+1)(N+2)\cdots(N+p)(N+p+1).
\end{aligned} \tag{4.26}$$

This completes the induction proof. Thus the total number of items in the formula (4.22), corresponding to $p = N+1$, is

$$\begin{aligned}
& \frac{1}{(N+1)*(N)*\cdots*3*2}(N+1)(N+2)\cdots(2N)(2N+1) \\
= \; & \frac{(2N+1)!}{(N)!(N+1)!}
\end{aligned}$$

$$= \begin{pmatrix} 2N+1 \\ N \end{pmatrix}. \tag{4.27}$$

□

Formula (4.22) defines the scheme which we believe achieves the minimum number of logical operations through the following parallel algorithm.

Algorithm 4.3 : Evaluating the PBF with the minimum number of operations.
Input: the input binary data set $(c_1, c_2, \cdots, c_{2N+1})$.
Output: the output binary result r.

1. $M = c_{2N}$ & $c_{2N+1}, S = c_{2N} \mid c_{2N+1}$.
2. $T_1 = (c_{2N-1}$ & $S) \mid M$.
3. **FOR**$(i = 2, \cdots, N$) BEGIN
4. $\quad M = M$ & c_{2N-i+1}.
5. $\quad T_i = (c_{2N-i}$ & $T_{i-1}) \mid M$. END.
6. **FOR**($i = 1, \cdots, N-1$) BEGIN
7. $\quad S = S \mid c_{2N-i}$.
8. $\quad T_1 = (c_{2N-i-1}$ & $S) \mid T_1$.
9. $\quad$ **FOR**($j = 2, \cdots, N$)
10. $\quad\quad T_j = (c_{2N-i-j}$ & $T_{j-1}) \mid T_j$. END.
11. $r = T_N$.
12. End algorithm.

In steps 1 to 5, the algorithm calculates the last two terms of Formula (4.22) and from step 6 to the end, the algorithm calculates the other $N-1$ terms. In each iteration, indicated by variable i, one item can be calculated. The required number of logical operations is derived in the following theorem.

Theorem 4.4 *The number of logical operations required to implement a $(2N+1)$th order binary median filter based on formula (4.22) is $2N(N+1)$.*

Proof: In steps 1 to 7 in the algorithm, the number of logical operations is

$$1 + 1 + 2 + (N-1) * (1+2) = 3N + 1. \tag{4.28}$$

In steps 8 to 14, the number of logical operations is

$$(N-1)*(1+2+(N-1)*2) = (N-1)(2N+1). \tag{4.29}$$

Thus the total number is

$$3N+1+(N-1)(2N+1) = 2N^2+2N = 2N(N+1). \tag{4.30}$$

We can pursue the aim of minimising the number of operations for a median Boolean function in the context of any other PBF. For example, consider the PBF for cascading weighted median filters,

$$\begin{aligned} f_{WM}(x_{i-5},\cdots,x_i) &= x_{i-5}x_{i-4}x_{i-2}+x_{i-5}x_{i-3}x_{i-1}+x_{i-4}x_{i-3}x_{i-1}+x_{i-3}x_{i-2}+x_{i-2}x_{i-1} \\ &= x_{i-5}x_{i-4}x_{i-2}+x_{i-3}x_{i-1}(x_{i-5}+x_{i-4})+x_{i-2}(x_{i-3}+x_{i-1}) \end{aligned} \tag{4.31}$$

4.5. Bit-Parallel Structure and a Data-Parallelism Stack Filtering Algorithm

Bit-field parallel arithmetic is the most basic parallel processing mechanism and can be implemented in a computer with the facility of static random-access memories from which all the bits of a word can be read conveniently in parallel, can execute arithmetic instructions on all bits, and then can write all bits back to memories.

The bit-field width of the processor of a modern computer is not less than 16 and is often 32. In serial mode with a bit field of width 32, a binary datum occupies only 1 bit and the other 31 bits are wasted. We need therefore to develop a method of utilizing this structure effectively. The natural parallel approach is for each bit in the bit-field to hold a binary datum and then each logical or arithmetic operation is applied to all binary data in the bit-field in parallel. However, all of the original binary data are in serial mode so that we need to assemble them in the parallel structure and then, after accomplishing the filtering operation, we also need to disassemble or restore the parallel data into the original serial structure.

The easiest assembly procedure is to fill each bit field directly with 32 items of binary data. It takes $8*94 = 752$ time units to accomplish this assembly task, which can be expressed as

$$\begin{aligned} &(d_0\&\mathbf{MASK}_j) \mid (d_1\&\mathbf{MASK}_j \ll 1) \mid \cdots \mid (d_{31}\&\mathbf{MASK}_j \ll 31) \\ &j = 1,2,\cdots,8, \end{aligned} \tag{4.32}$$

where d_i, $i = 0,1,\cdots,31$ are the original data, $d_i\&\mathbf{MASK}_j$ gives a binary datum and, in C language notation, the logical operation "$\ll n$" means logical left shift by

n bits and "$\gg n$" means logical right shift by n bits. Each d_i belongs to the set $(0, 1, \cdots, 255)$.

Utilizing the equivalence of one 32-bit integer with a four character array, we can divide the assembly procedure into two steps which together involve fewer operations than directly assembly. In stage one, we assemble 4 data in byte bound and repeat this 8 times. In stage two we combine the 8 bytes into one bit-field. The whole procedure can be expressed as follows.

Algorithm 4.4 : The two-stage assembly of data
Input: The original serial data set $(d_1, d_2, \cdots, d_{32})$.
Output: The original parallel data set $(c^1, c^2, \cdots, c^8)$.

1. **FOR**$(i = 0;\ \ i \leq 7;\ \ i++)$ BEGIN
2. **FOR**$(j = 0;\ \ j < 3;\ \ j++)$
3. $w_i[j] = d_{4*i+j}$. END
4. $c^1 = (w_0 \& (80808080)_2) \gg 7\ |(w_1 \& (80808080)_2) \gg 6\ | \cdots | w_8 \& (80808080)_2)$.
5. $c^2 = (w_0 \& (40404040)_2) \gg 6\ | \cdots | (w_8 \& (40404040)_2)) \ll 1$.
6. $c^3 = (w_0 \& (20202020)_2) \gg 5\ | \cdots | (w_8 \& (20202020)_2)) \ll 2$.
7. $c^4 = (w_0 \& (10101010)_2) \gg 4\ | \cdots | (w_8 \& (10101010)_2)) \ll 3$.
8. $c^5 = (w_0 \& (08080808)_2) \gg 3\ | \cdots | (w_8 \& (08080808)_2)) \ll 4$.
9. $c^6 = (w_0 \& (04040404)_2) \gg 2\ | \cdots | (w_8 \& (80808080)_2)) \ll 5$.
10. $c^7 = (w_0 \& (02020202)_2) \gg 1\ | w_1 \& (02020202)_2\ | \cdots |\ (w_8 \& (02020202)_2))\ \ll 6$.
11. $c^8 = w_0 \& (01010101)_2)\ |\ (w_1 \& (01010101)_2) \ll 1\ | \cdots |\ (w_8 \& (01010101)_2)) \ll 7$.
12. End algorithm.

In the algorithm, 32 original data are assembled into 8 parallel data. The superscripts of the set c^i correspond to the levels of minimum threshold decomposition. The following theorem gives the time requirements of this procedure.

Theorem 4.5 *The two-stage assembly procedure requires 208 time units.*

Proof: From steps 1 to 5 of the algorithm, it obviously takes 32 time units to complete the first stage task and from steps 6 to 13, it takes $8 * 22 = 176$ time units to finish the operation, giving a total time of

$$32 + 176 = 208 \ \text{ units.} \tag{4.33}$$

□

Using a similar procedure, we present the following algorithm for the disassembly procedure.

Algorithm 4.5 : The two-stage disassembly of data
Input: The parallel filtered data set $(r^0, r^1, \cdots, r^7)$ and $\mathbf{MASK}_i$, $i = 0, 1, \cdots, 7$: $(80808080)_2, (40404040)_2, (20202020)_2, (10101010)_2, (08080808)_2, (04040404)_2, (02020202)_2, (01010101)_2$
Output: The output data set $(o_1, o_2, \cdots, o_{32})$.

1. **FOR**($i = 0$; $i \leq 7$; $i++$) BEGIN
2. $w_i = 0$.
3. **FOR**($j = 0$; $j \leq 7$; $j++$)
4. $w_i = w_i \mid ((r_j \ \& \ \mathbf{MASK}_i) \gg j)$. END.
5. **FOR**($i = 0$; $i \leq 7$; $i++$) BEGIN
6. **FOR**($j = 0$; $j < 3$; $j++$)
7. $o_{4*i+j} = w_i[j]$. END.
8. End algorithm

The time requirement of the disassembly procedure is presented in the following theorem.

Theorem 4.6 *The disassembly procedure requires 232 time units.*

Proof: From steps 1 to 6 of the algorithm, it takes $8 * (1 + 8 * 3) = 200$ time units to complete the first stage and from steps 6 to 11, it takes 32 time units for the second stage, giving a total of

$$200 + 32 = 232 \ \text{ units.} \tag{4.34}$$

□

The assembly procedure is used to convert data from serial to parallel form, which is necessary for obtaining the median filter by minimum threshold decomposition, and the disassembly procedure is used to convert the data resulting from the median filter into general serial data. The above procedures for both assembly and disassembly are time consuming.

The bits-field parallel algorithm for the stack filter by minimum threshold decomposition can be expressed as follows.

Algorithm 4.6 : One-dimensional stack filter algorithm with the minimum threshold decomposition

Input: Length of the original data is L, each datum belongs to $(0, 1, \cdots, M-1)$, and the window width of the median filter is $2N+1$.

1. Input original serial data and divide them into 32 parts.
2. Apply the first stage of assembly to all 32 parts of data.
3. Initialise variables lt^j and gt^j.
4. REPEAT point filtering **From** 1 **To** L **Do** steps 5 to 11
5. **REPEAT** each level of the minimum threshold decomposition **From** $\log_2 M$ **To** 1 **Do** steps 6 - 11
6. Apply $\mathbf{LOAD}_2$ to the next datum point, giving c^i_{2N+1}.
7. Use equation (4.10) to accomplish the minimum threshold decomposition of $c^i_1, c^i_2, \cdots, c^i_{2N+1}$.
8. Use the minimum logic operations formula which is similar to that for evaluating the PBF in Algorithm 4.3 for median filters. $f_S(c^i_1, c^i_2, \cdots, c^i_{2N+1})$, and get the result r^i.
9. Update $c^i_2, c^i_3, \cdots, c^i_{2N+1}$ to $c^{i+1}_1, c^{i+1}_2, \cdots, c^{i+1}_{2N}$.
10. Carry out $\mathbf{SAVE}_1$ for filtering output r^i.
11. Apply equation (4.9) to update variables lt_i and gt_i for all data in the window.
12. Carry out stage two of disassembly, defined by steps 7 to 12 of algorithm 3.
13. End algorithm.

In the above algorithm, $\mathbf{LOAD}_2$ means applying steps 6 to 14 of algorithm 1 to accomplish stage two of assembly and $\mathbf{SAVE}_1$ means applying steps 1 to 6 of algorithm 2 to accomplish stage one of disassembly.

In the two-dimensional case, there are many possible shapes for the filter window (square, cross, diamond, etc) as shown in Figure 14. Although the scheme for stack filtering in two-dimensions is different from that in one-dimension, the evaluation of the PBF of the median filter in algorithm 1 still obtains. The two-dimensional stack filters has very similar structure to that of its one-dimensional algorithm. Whether the logical variables belong to one-dimension or two-dimensions, their action in the algorithm is the same, involving only the logical **TRUE/1** or **FALSE/0**. For example,

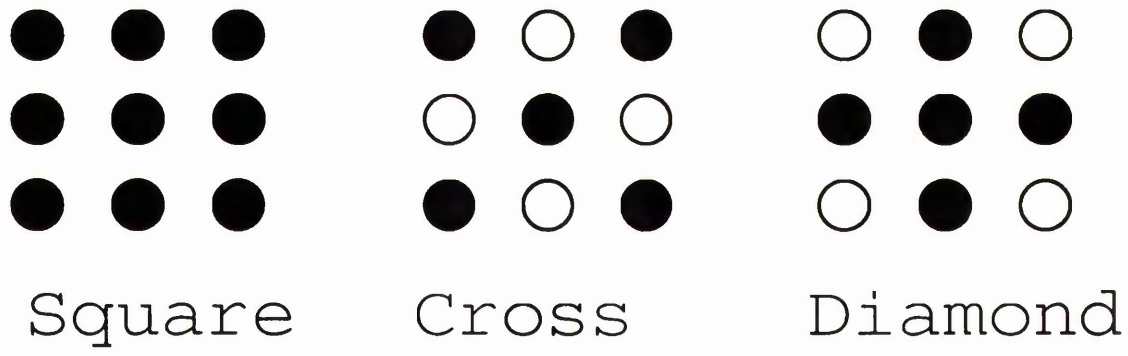

Fig. 14. The Shape of Windows of Two-dimensional Stack Filters

for the 3×3 square-window median filter, the PBF can be expressed as

$$f_{med}\begin{pmatrix} x_{11}, & x_{12}, & x_{13}, \\ x_{21}, & x_{22}, & x_{23}, \\ x_{31}, & x_{32}, & x_{33}, \end{pmatrix} = f_{med}(x_{11}, x_{12}, x_{13}, x_{21}, x_{22}, x_{23}, x_{31}, x_{32}, x_{33}). \tag{4.35}$$

For a ninth-order one-dimensional median filter, the PBF can be expressed as $f_{med}(x_1, x_2, \cdots, x_9)$. Both PBFs have the same form if we replace x_1 by x_{11}, x_2 by x_{12}, x_3 by x_{13}, x_4 by x_{21}, x_5 by x_{22}, x_6 by x_{23}, x_7 by x_{31}, x_8 by x_{32}, x_9 by x_{33}.

The real parallel algorithm of stack filters can be expressed in a PVM environment. We use the data parallelism scheme and master/slave models, described in Section 3-1. We simply partition the original one-dimensional input data into K parts, such as

$$\{X_i\},\ (i = 0, 1, \cdots, L-1) = \bigcup_{t=1}^{K} P_t, \tag{4.36}$$

where $P_t = \{X_{tK+u}\} + B$, $u = 0, 1, \cdots, \frac{L}{K} - 1$. For two-dimensional data, we partition it into $K \times K$ parts, such as

$$\{X_{ij}\},\ (i, j = 0, 1, \cdots, N-1) = \bigcup_{t,q \in [0,K)} P_{tq}, \tag{4.37}$$

where $P_{tq} = \{X_{tK+u,qK+v}\} + B$, $u, v = 0, 1, \cdots, \frac{L}{K} - 1$. In both one- and two-dimensional cases, there is some data , B, in border regions where a processor must share the points which have been assigned to a neighboring processor, as shown in Figure 15.

The width of the shared data can be controlled by the new filtering parameter $MaxWidth$ in the following algorithm.

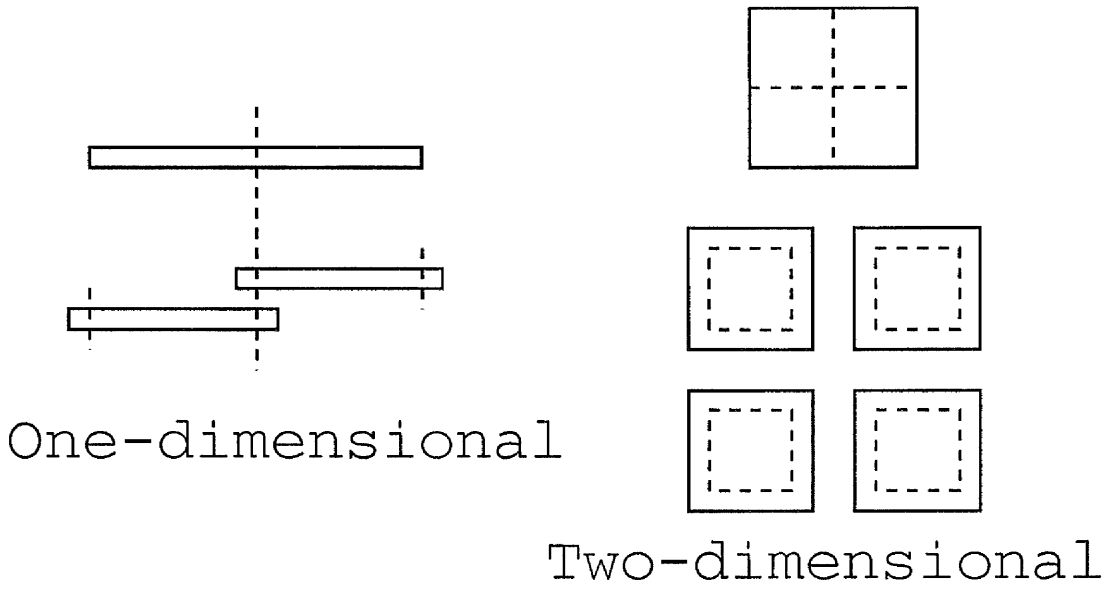

Fig. 15. Data partitions of One- and Two-dimensional Parallel Stack Filters

Algorithm 4.7 : Parallel stack filter algorithm with the data parallelism scheme. Algorithm for the master processor :

Input: Original data length L, Number of subtasks K, the filtering parameters (window width $2N+1$, PBF $f_S(\cdot)$, $MaxWidth$).

1. Partition data according to equation (4.36) or (4.37).
2. Send each subtask to corresponding slaves.
3. Send stack filter parameters ($2N+1$, $f_S(\cdot)$, $MaxWidth$) to its slaves.
4. Receive the computation results from each slave.
5. Combine all parts as the stack filter's output.
6. End of the master algorithm.

Algorithm for the slave processors : **Input:** sub-data set length $\frac{L}{K}$, the filtering parameters ($2N+1$, $f_S(\cdot)$, $MaxWidth$).

1. Receive one sub-data set.
2. Receive stack filter parameters.
3. Apply algorithm 4.6 to do stack filtering.
4. Send the part result back to its master.
5. End of the slave algorithm.

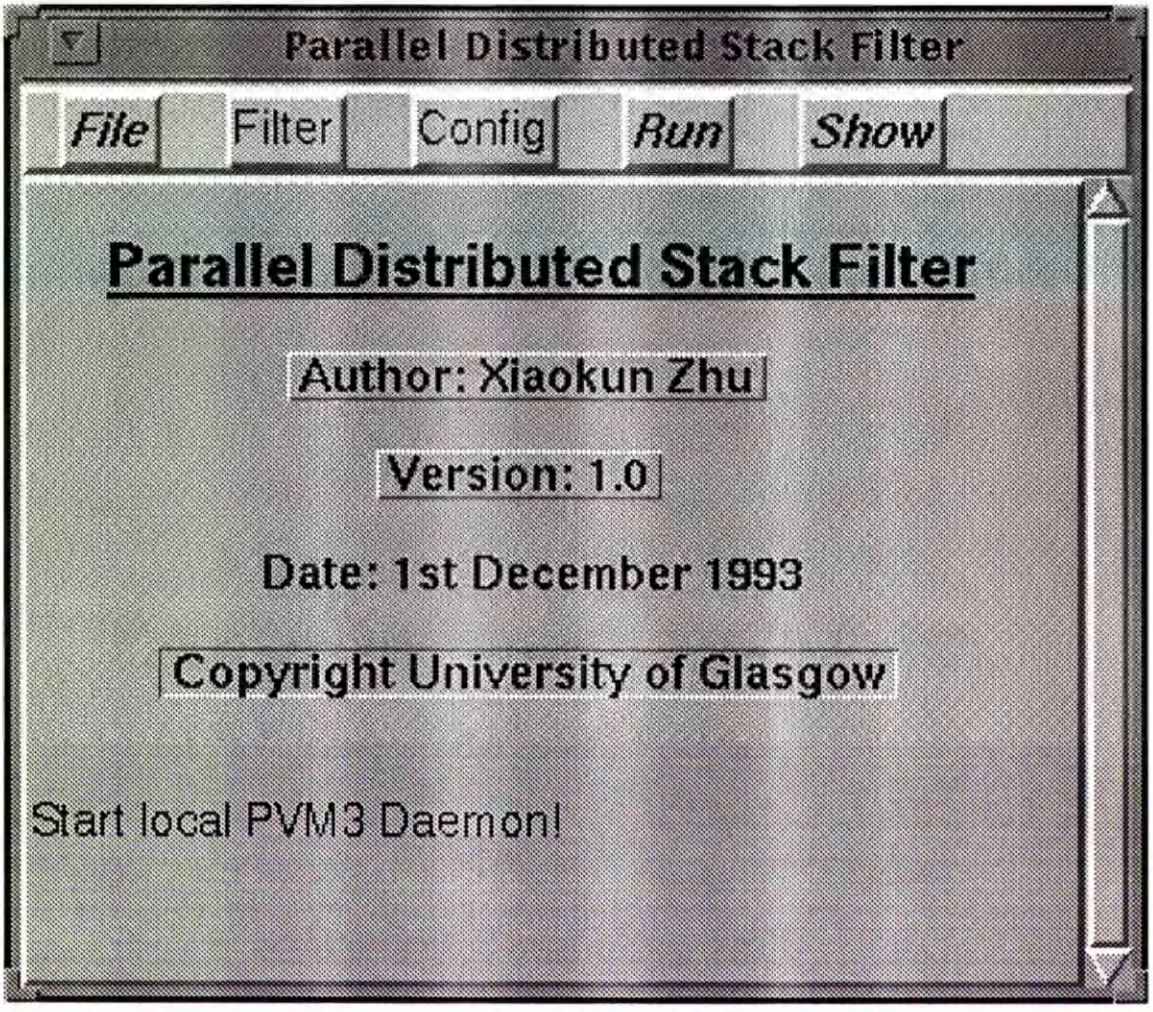

Fig. 16. Interactive Parallel Distributed Stack Filtering System

4.6. Implementation of Interactive Stack Filtering System

In this section we use IPDCE (Interactive Parallel Distributed Computing Environment) to design a Interactive Stack Filtering System (ISFS). IPDCE, as shown in 16, provides both parallel computing, interactive processing and Graphic User Interface development facilities.

Our aims are to provide the following capabilities in the ISFS:

- set any input/output file names.
- modify any stack filter parameters ($MaxWidth$, window width $2N+1$, PBF).
- permit the use of serial or parallel algorithms.
- modify any parallel processing parameters.
- visualise the original and filtered data.

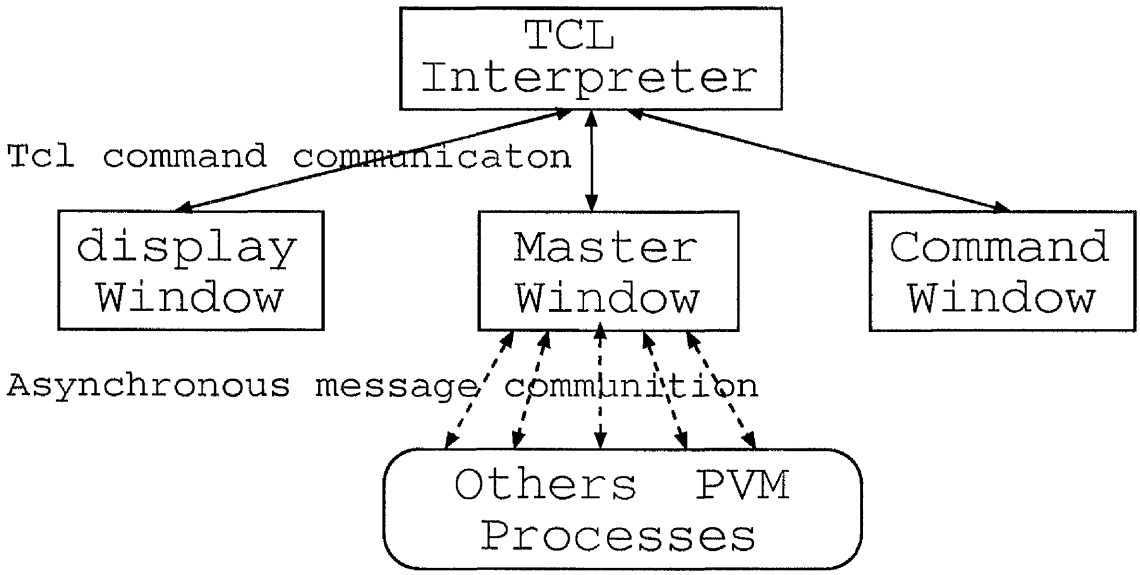

Fig. 17. The Structure of Interactive Stack Filtering System

- provide both GUI and command line mode.

The structure of the ISFS is illustrated in Figure 17. The ISFS consists of Display/Command window, a menu *File*, a menu *Filter*, a menu *Config*, a menu *Run* and a menu *Show*. There are two kinds of mode, command line and menu, to process a user request. The default mode of ISFS is menu. In order to switch to command line mode, the user clicks on the menu item *File/Command*. The command *sfexit* transfers ISFS back to menu mode.

In command line mode, the user can use any Tcl/Tk commands and any IPDCE commands. For example, the commands for starting and ending XAB, *xabIon* and *xabIoff*, are only used in command line mode.

In menu *File* there are four menu items, *Load, Save, Command, Quit. Load* will display a dialog window to ask the user to input a new original data file name as shown in Figure 18. The parameters of Size and Dimension will be obtained from the input file. *Save* will display a dialog to ask the user give a output file name. *Command* switches to command mode. *Quit* will quit the ISFS.

Menu *Filter* is a dialog window as shown in Figure 19. The user can set the stack filter's parameters in it. These parameters include window width, maximum width, filtering commands (PBF) and two additional parameters for the two-dimensional filter.

Menu *Config* is also a dialog window as shown in Figure 20. The user can choose to use the serial or the parallel algorithm, can partition his task into K subtasks with

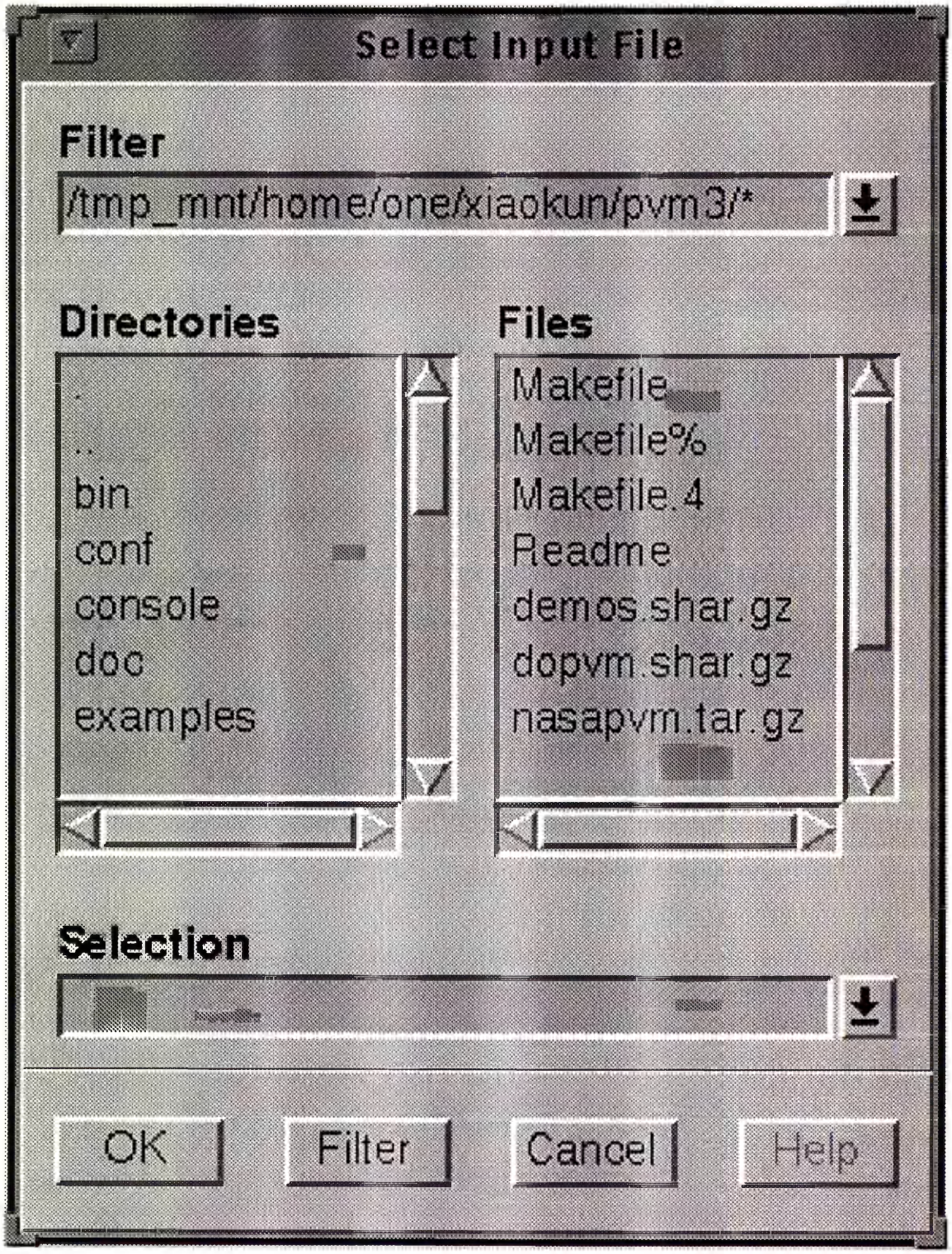

Fig. 18. Dialog Window of Select an Input File Name

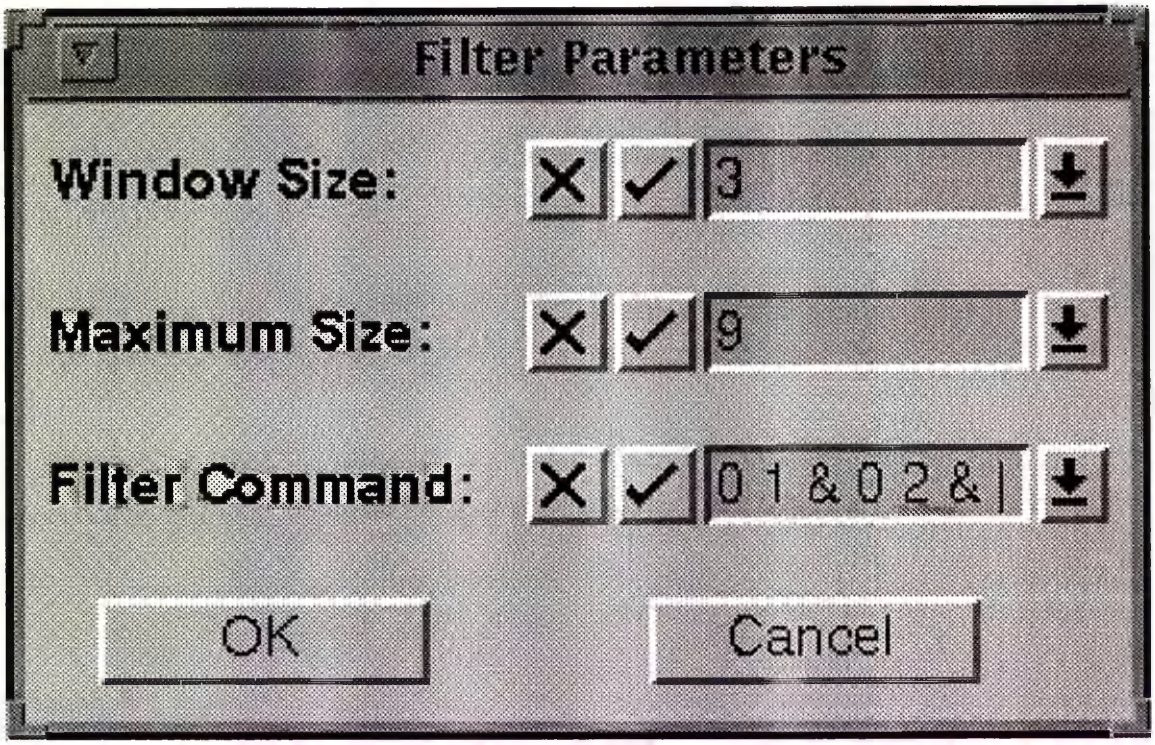

Fig. 19. Dialog Window of Select Filter's Parameters

the data parallelism scheme, and can dynamically add and delete *host* of PVM.

In menu *Run* there are two menu items, *Start, End.* The menu act as Master window in Figure 17. Item *Start* will begin the parallel algorithm 4.7 and item *End* will end the algorithm, of which the function is to destroy all slave processes built from the master process in PVM. Item *Start* can be called many times, and each time the user can redefine new filtering parameters from the menu *Filter*

In menu *Show* there are three menu items, *Show parameters*, *Show input data* and *Show output data.* Item *Show parameters* displays current filtering parameters and parallel relative configuration in the Display/Command window. Item *Show input data* displays the original input in a special window. For one-dimensional data, Howlett's graph widget [95] is applied as shown in Figure 21. For two-dimensional data, the Mackerras's photo widget [125] is applied as shown in Figure 22.

4.7. Numerical Examples

In this section, we give two numerical simulations to illustrate the performance of Parallel Distributed Stack Filter Systems (PDSFS). In the first example, we use PDSFS to process one-dimensional data which are corrupted by additive Gaussian white noise and impulsive noise. In order to quantitatively compare the performance abilities of

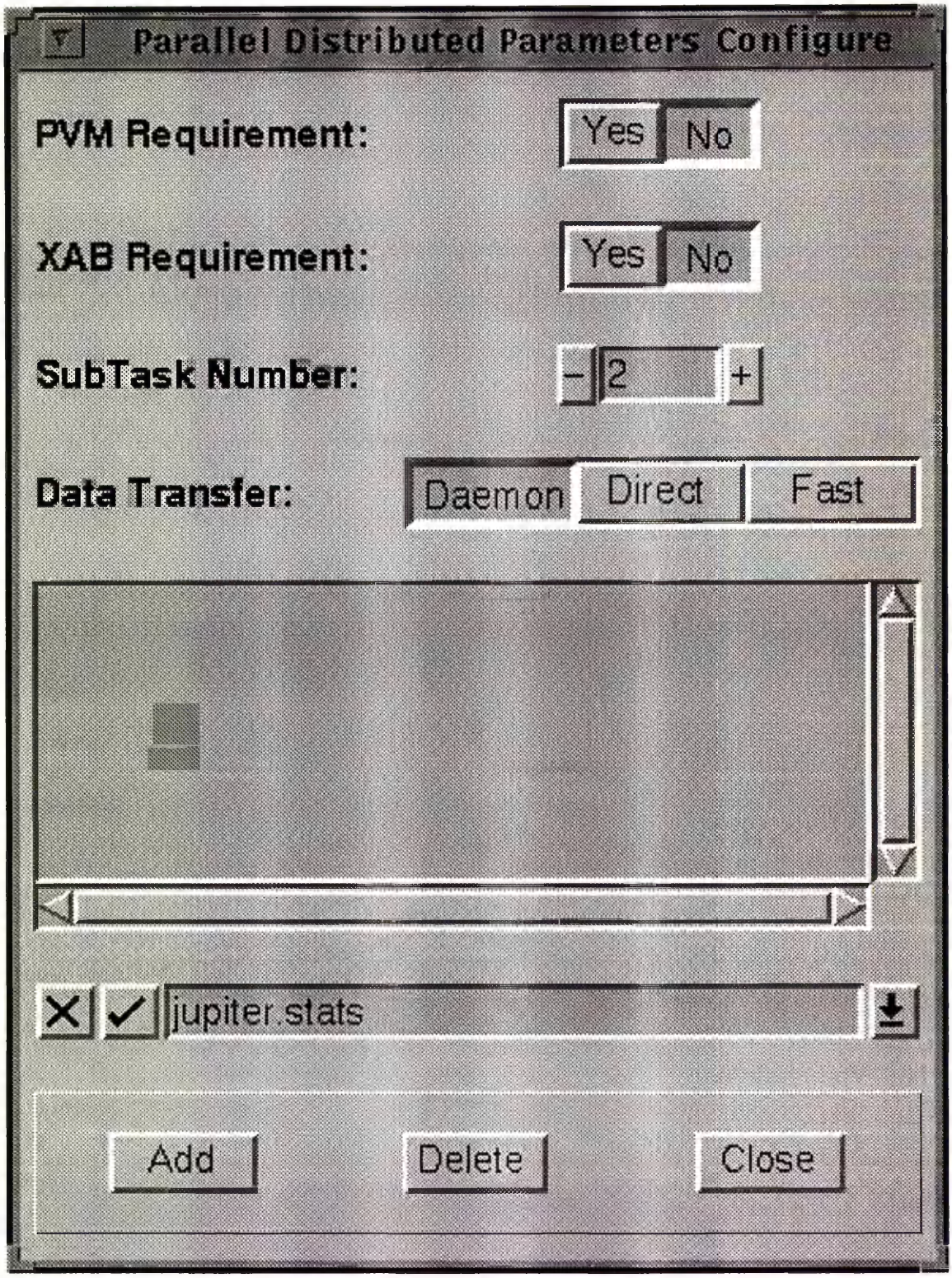

Fig. 20. Dialog Window of Select the Network and PVM Parameters

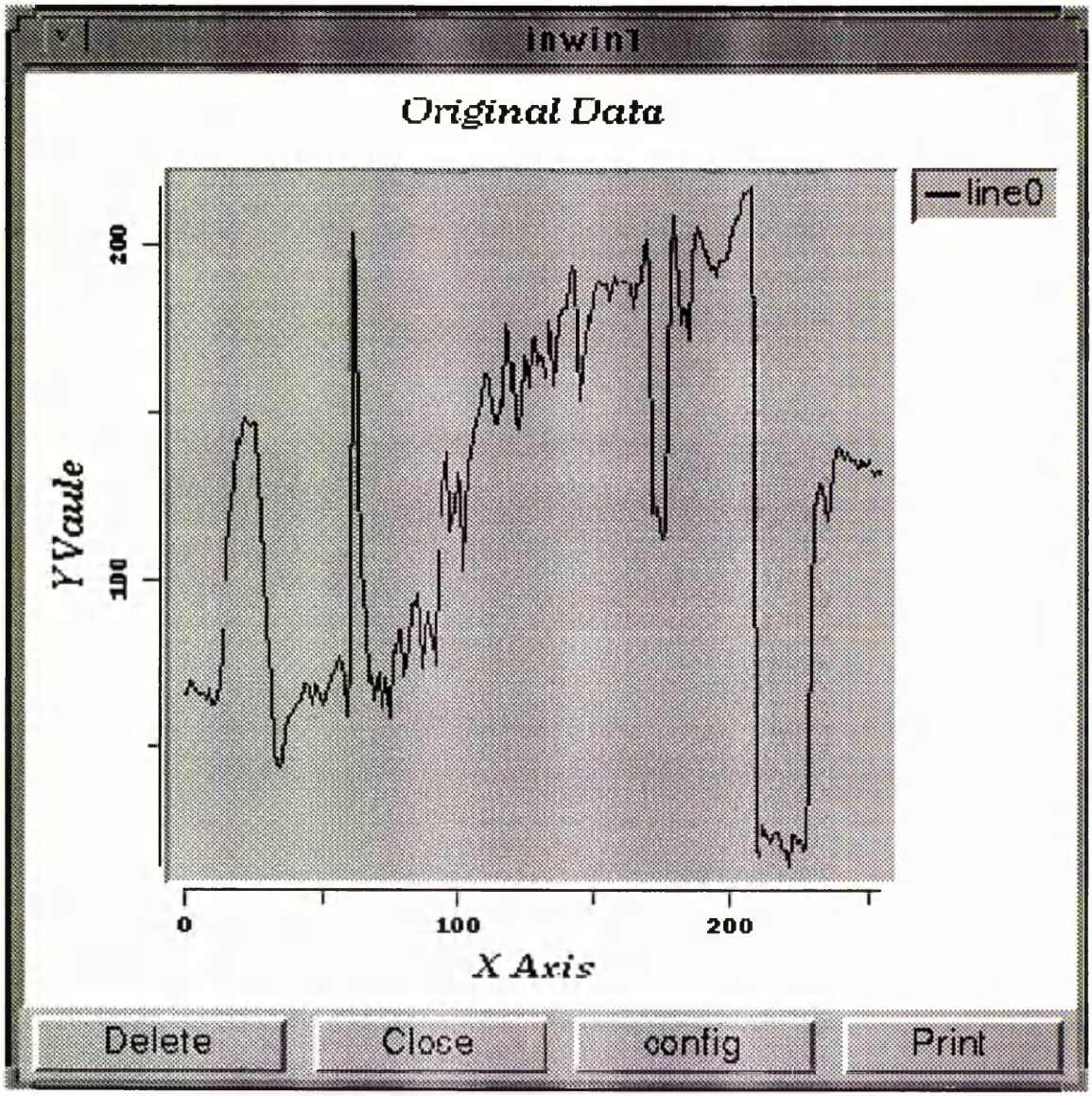

Fig. 21. One-dimensional Data Display Window

Fig. 22. Two-dimensional Data Display Window

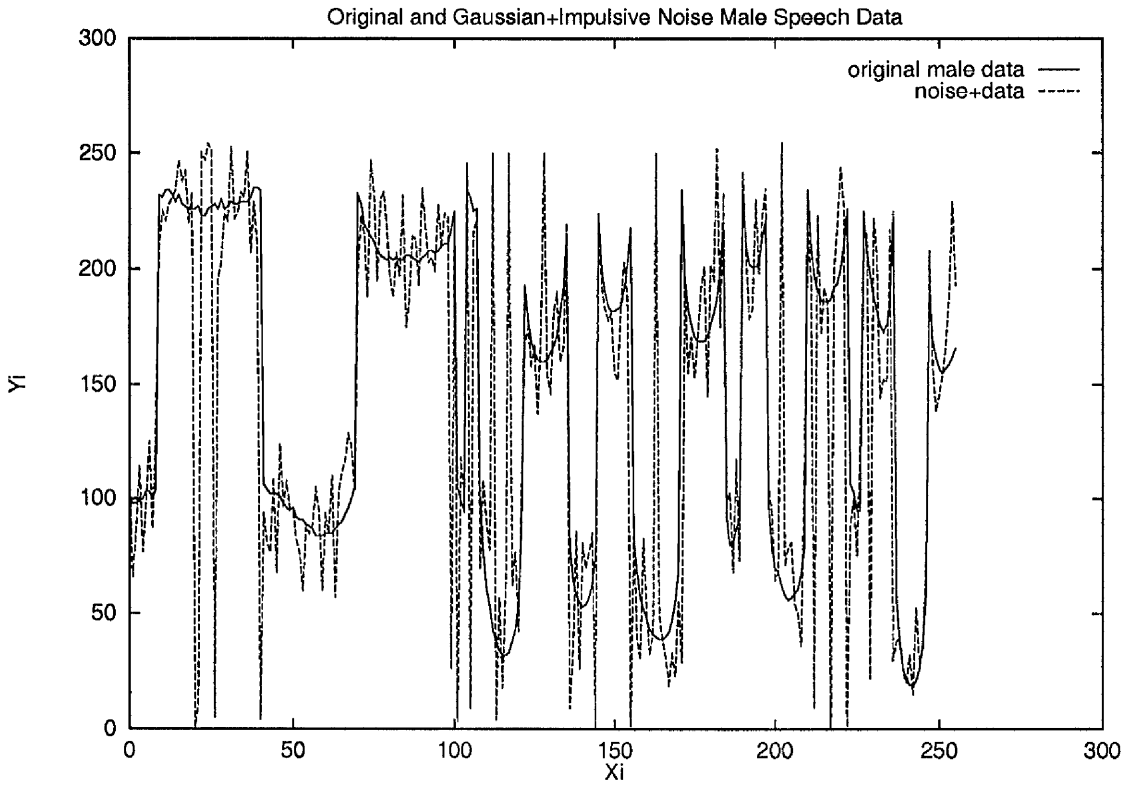

Fig. 23. Male speech signal corrupted by Gaussian noise with $\mu = 0$ and $\sigma = 10$ and impulsive noise with occurrence probability $p = 0.1$

several filters used here, we define the Normalized Mean Square Error (*NMSE*) between the original, noisy input and the filtered output data as the following:

Definition 4.6

$$NMSE = \frac{\sum_{i=0}^{N}[Y(i) - S(i)]^2}{\sum_{i=0}^{N}[X(i) - S(i)]^2} \tag{4.38}$$

where $S(i), X(i), Y(i)$ *are the original signal, the input signal and the filtered output signal respectively.*

Example 4.1 The original input signal is a piece of male speech as shown in Figure 23, of dimension 256 and with magnitude belonging to [0, 255]. We suppose the original signal is corrupted by Gaussian white noise with zero mean and standard deviation $\sigma = 10.0$ and by impulsive noise with the probability $p = 0.1$ of an impulse occurring at any given point, where the impulse magnitude can be 0 or 255 with the equal probability.

We use two types of filters to smooth this signal. One is the standard median (SM) filter and the positive Boolean function (*PBF*) of the binary fifth-order SM is

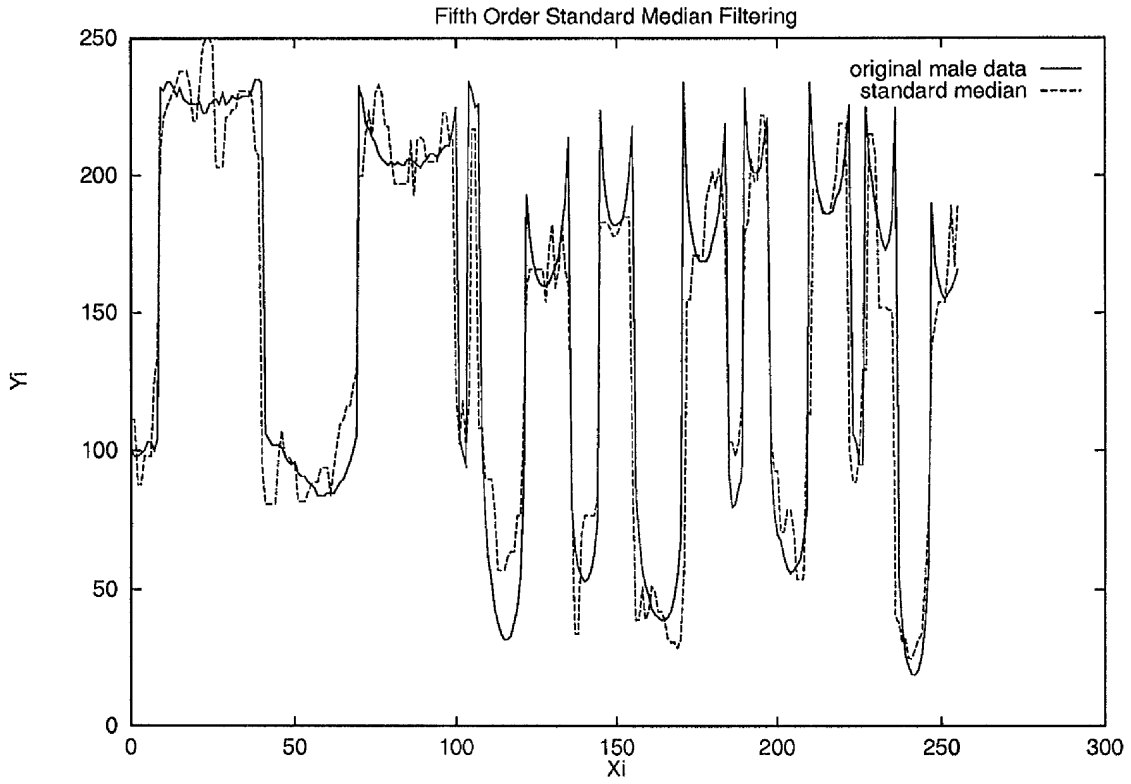

Fig. 24. Fifth-order SM filter for male speech signal corrupted by Gaussian and impulsive noise

given by

$$f_{med}(x_1, x_2, x_3, x_4, x_5) = x_1(x_2(x_3 + x_4 + x_5) + x_3(x_4 + x_5) + x_4x_5) + x_2(x_3(x_4 + x_5) + x_4x_5) + x_3x_4x_5. \quad (4.39)$$

The other is a weighted order statistic filter. We only use the special version corresponding to the central weighted median (CWM) filter. The *PBF* of the binary weight 2 fifth-order CWM is given by

$$f_{cwm}(x_1, x_2, x_3, x_4, x_5) = x_3(x_1 + x_2 + x_4 + x_5) + x_1x_2(x_4 + x_5) + x_4x_5(x_1 + x_2). \quad (4.40)$$

Figures 24 and 25 show the results of filtering the noisy signal in Figure 23 with the fifth-order SM filter and the weight 2 fifth-order CWM filter respectively. Table V summarizes the NMSE of the SM and CWM filters. Comparison of these NMSE indicates that the CWM filter performs slightly better than the SM filter.

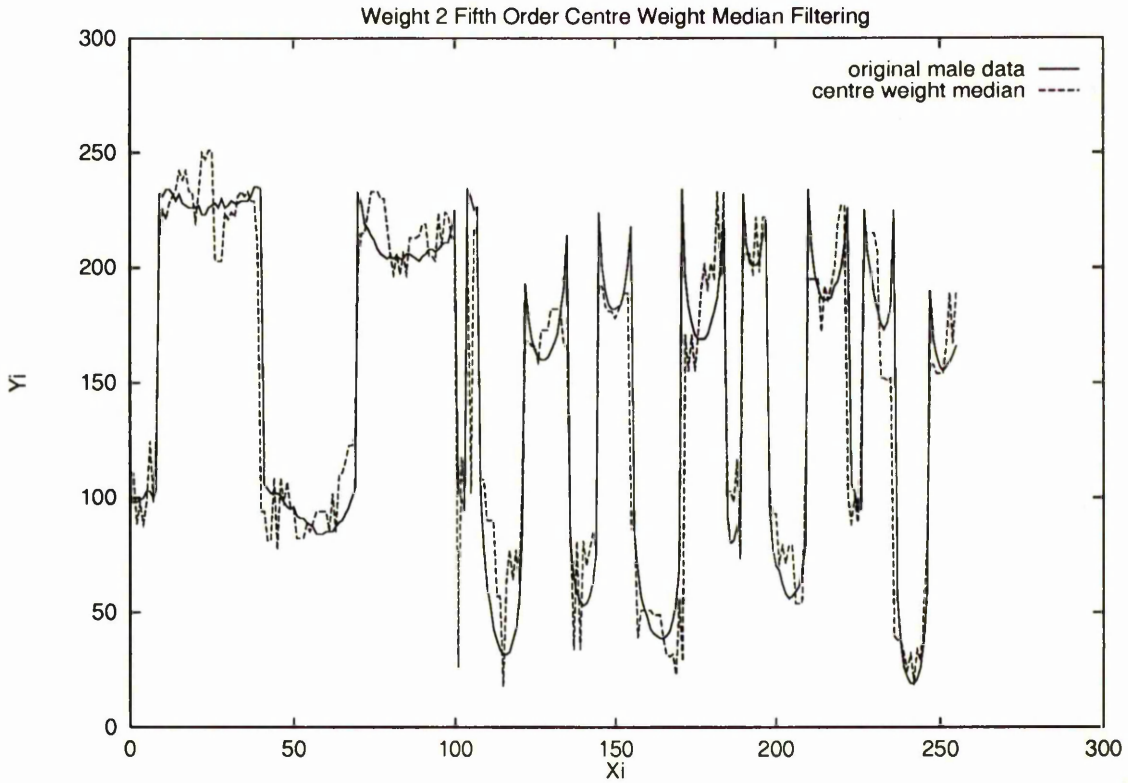

Fig. 25. Weight 2 fifth-order CWM filter for male speech signal corrupted by Gaussian and impulsive noise

Table V. Normalised Mean Square Error for male speech data corrupted by Gaussian and impulsive noise with SM and CWM filters

	NMSE
central weight median	0.26
standard median	0.3

Table VI. Normalized Mean Square Error for lena test image corrupted by Gaussian and impulsive noise with two-dimensional SM and CWM filters

	NMSE
central weight median	0.22
standard median	0.34

In the second example, we use PDSFS to smooth two-dimensional images which are also corrupted by additive Gaussian white noise and impulsive noise. The version of two-dimensional Normalized Mean Square Error (*NMSE*) can be defined as

Definition 4.7

$$NMSE = \frac{\sum_{i=0}^{N}\sum_{j=0}^{N}[Y(i,j) - S(i,j)]^2}{\sum_{i=0}^{N}\sum_{j=0}^{N}[X(i,j) - S(i,j)]^2}. \tag{4.41}$$

where $S(i,j), X(i,j), Y(i,j)$ are the original image, the noise corrupted input image and the filtered output image respectively.

In addition, we shall present the image showing the differences between the original and the filtered images. These images provide information about both the detail-preservation and noise-suppression characteristics of filters. In the difference image, a zero difference is shown as a black pixel and a difference of 255 is shown as a white pixel.

Example 4.2 The original input image is the standard test image, "lena", which consists of 256×256 pixels with eight bits of resolution. The original noise-free image is shown in Figure 22. The noise corrupted image was generated by adding zero mean Gaussian noise of standard deviation 20 and impulsive noise with occurrence probability 0.2. We evaluated two types of filters, two-dimensional SM and CWM, under PDSFS. The PBF of the 3×3 square-window SM have been given in equation (4.35) and the PBF of 3×3 square-window weight 3 CWM is given by

$$f_{cwm}\begin{pmatrix} x_{11}, & x_{12}, & x_{13}, \\ x_{21}, & x_{22}, & x_{23}, \\ x_{31}, & x_{32}, & x_{33}, \end{pmatrix} = f_{med5}(x_{11}, x_{12}, x_{13}, x_{21}, x_{23}, x_{31}, x_{32}, x_{33}) + f_{med3}(x_{22}, x_{11}, x_{12}, x_{13}, x_{21}, x_{23}, x_{31}, x_{32}, x_{33}) \tag{4.42}$$

where $f_{med5}(\cdot)$ is the fifth-order binary SM filter and $f_{med3}(\cdot)$ is the third-order binary SM filter.

Table VI summarizes the the NMSE of the two-dimensional SM and CWM filters. Figure 28 and Figure 27 show the results of filtering the noisy image in Figure 26 with two-dimensional window 3×3 SM filter and weight 3 window 3×3 CWM

Fig. 26. 256x256 lena test image corrupted by Gaussian noise with $\mu = 0$ and $\sigma = 20$ and impulsive noise with occurrence probability $p = 0.2$

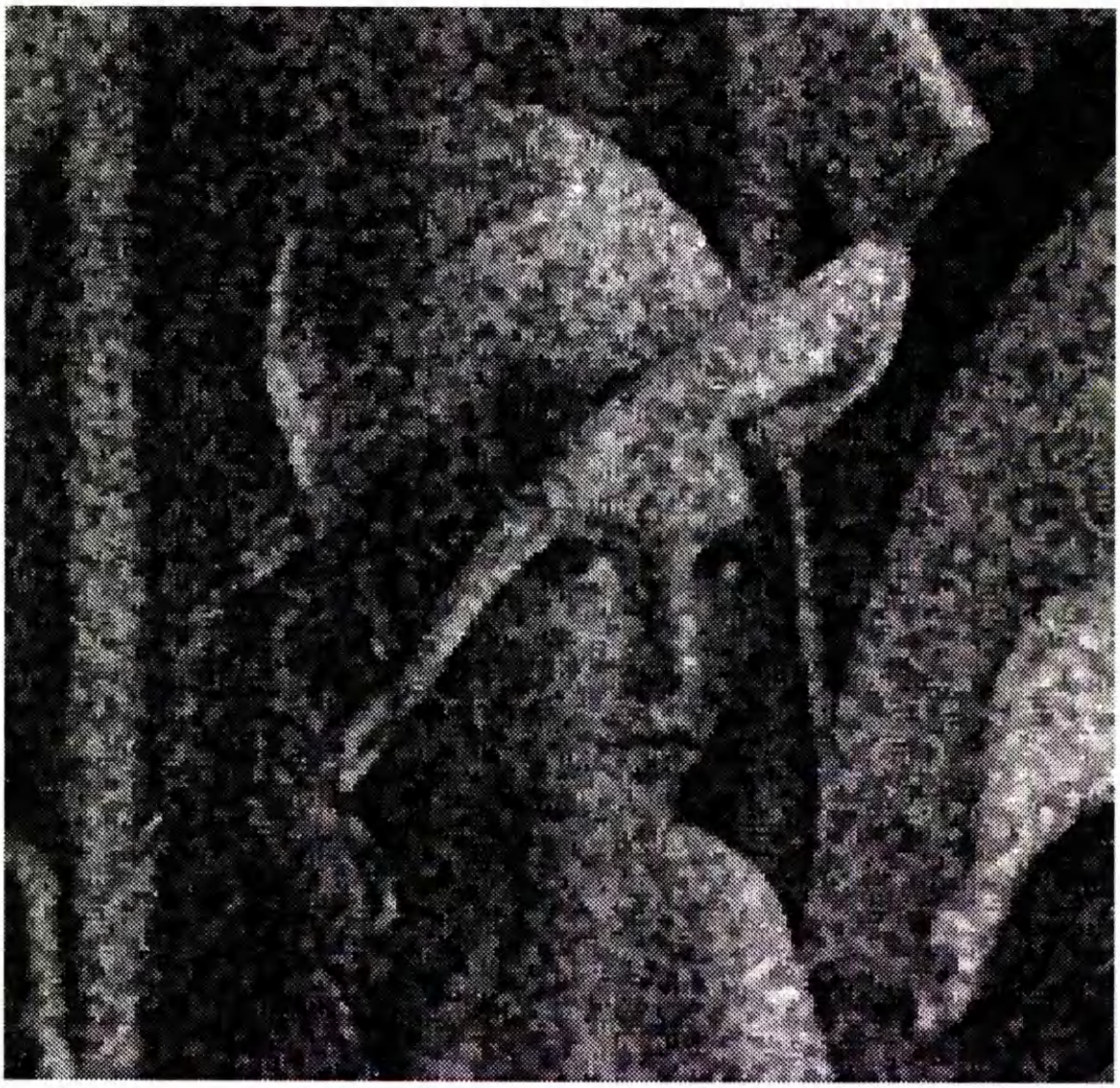

Fig. 27. Two-dimensional weight 3 window 3×3 CWM filter for lena image corrupted by Gaussian and impulsive noise

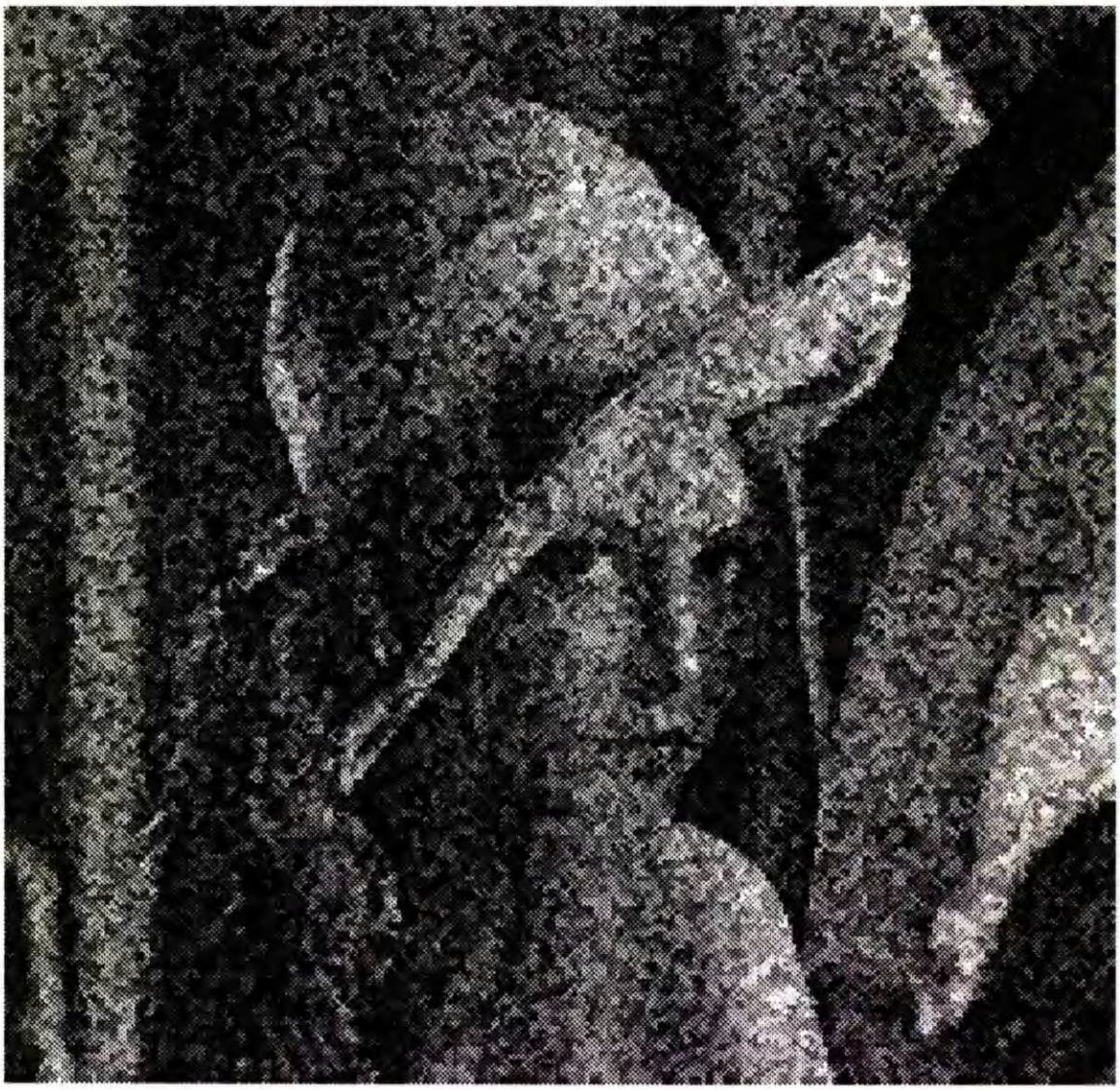

Fig. 28. Two-dimensional window 3×3 SM filter for lena image corrupted by Gaussian and impulsive noise

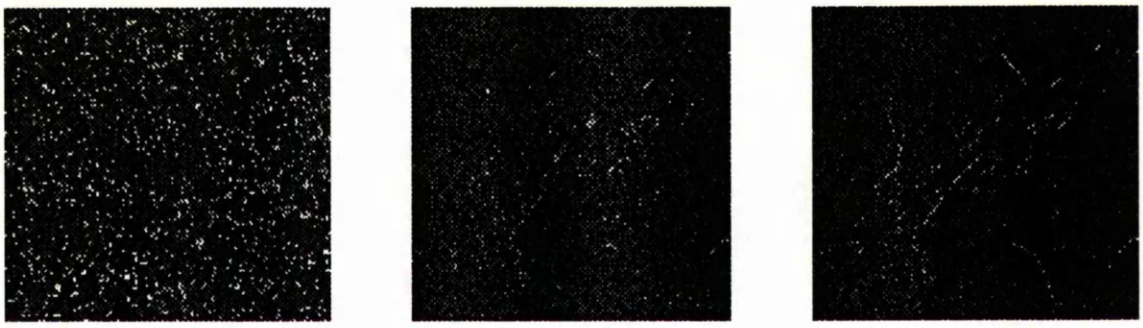

Fig. 29. Difference images, (a) Fig 26 - original, noise free image, (b) Fig 27 - original, noise free image, (c) Fig 28 - original, noise free image

Table VII. Execution Times (milli-seconds) and Communication Times (milli-seconds) of two-dimensional SM and CWM filters for lena image

Computers	SM filter		CWM filter	
	Exec. Time	Comm. Time	Exec. Time	Comm. Time
1	3500	0	3900	0
2	2310	476	2460	453
3	1820	699	1950	687
4	1610	917	1715	905

filter respectively. Figure 29 shows the difference between the original and the noisy and filtered images. It is seen from the above difference images that the SM filter caused more blur than the CWN filter. Table VI tells us that the noise-suppression characteristic of the SM filter looks poorer than the one of the CWM filter.

We distributed the filter algorithms among several computers under PDSFS in order to compute the results for Example 4.2. Table VII summarizes the filter executing time and communication time. Figure 30 shows the total time of the parallel distributed filtering algorithms. The computers of the parallel algorithms used are SUN Sparc ELC, IPC, Sparc 10, and SUN 470. Based on Figure 30 and Table VII we find that the parallel speed-up is good when the number of computers is small, for example, two or three. It is worthy of note that the main part of PDSFS was written using the TCL/TK language. The TCL/TK language is an interpretative language, which means that some execution degradation will appear compared with a compiler such as C language.

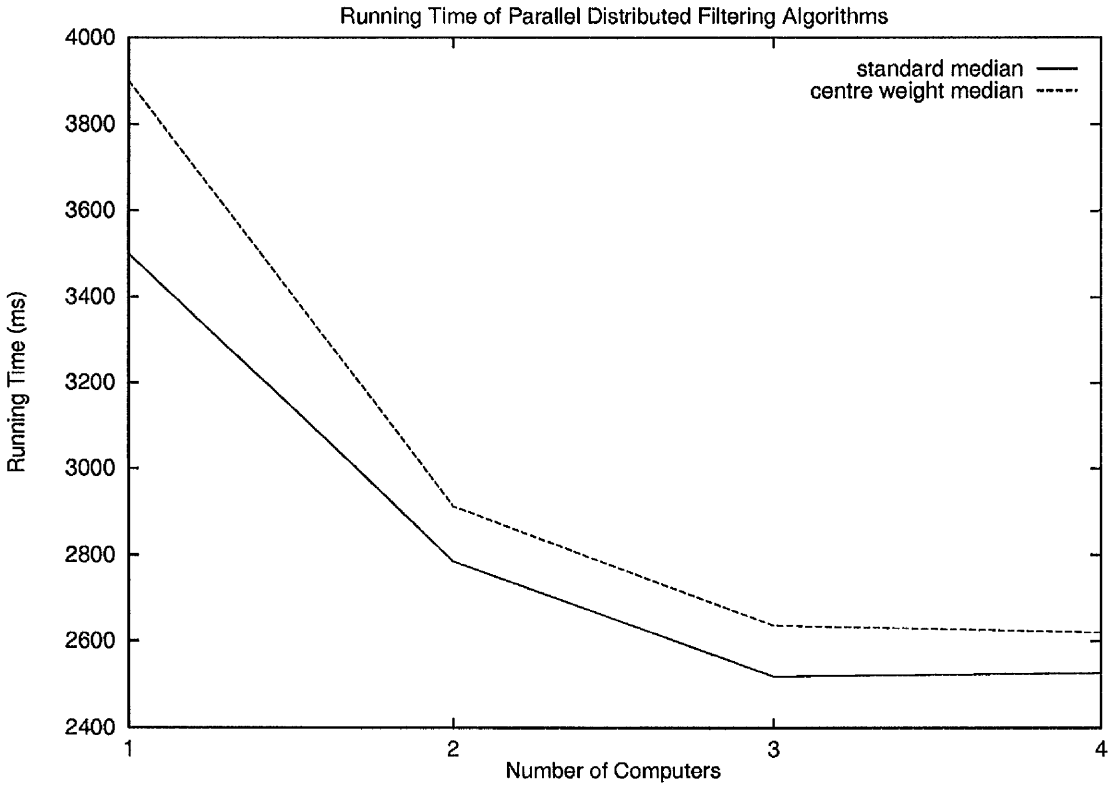

Fig. 30. Running time of parallel distributed filtering algorithms for lena image

CHAPTER 5

AN ITERATED FUNCTION SYSTEM MODEL OF ONE-DIMENSIONAL DISCRETE SIGNAL

5.1. Introduction

In this Chapter we present an extended Iterated Function System (IFS) interpolation method for modelling for a given discrete signal. In order to reduce the computing complexity we introduce a suboptimal search algorithm with a robust technique for estimating the IFS affine map parameters. Simulation results show that the IFS approach achieves a higher signal to noise ratio than does an existing approach based on autoregressive modelling. We also exploit the power of a computer network in implementing a full parallel distributed algorithm for the suboptimal search using an Remote Procedure Call (RPC) scheme. The simulation results show that the speed-up rate is almost proportional to the number of computers.

5.2. The Construction of an IFS Model for a Given Signal

5.2.1. Background of IFS Theory

In a deterministic fractal model with IFS, a one-dimensional signal , also known as a time series, $\{(x_i, y_i) : i = 0, 1, \cdots, N;\ x_i < x_{i+1},\ |x_i - x_j| \leq N, \forall i, j, y_i \in \mathbf{R}^1\}$ is divided into M parts by contractive maps. Each part is self-affine to the whole signal, known as the self-affine region. The end-points of each component will be denoted by (u_j, v_j), $j = 0, 1, \cdots, M$ and, in particular, $(u_0, u_0) = (x_0, y_0)$ and $(u_M, v_M) = (x_N, y_N)$. In order to simplify notation, we define a vector $P = \{i_j : j = 0, 1, 2, \cdots, M\}$ so that, for each i_j, (x_{i_j}, y_{i_j}) is an end-point. Throughout we shall restrict our attention to affine transformations[20, 22, 26], and we therefore define the contraction map w_j by

$$w_j \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a_j & 0 \\ c_j & d_j \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e_j \\ f_j \end{pmatrix} \quad j = 1, 2, \cdots, M, \tag{5.1}$$

where $a_j > 0$, which means that, for the region $[i_{j-1},\ i_j]$, w_j maps $(x_0,\ y_0)$ to $(x_{i_{j-1}},\ y_{i_{j-1}})$ and $(x_N,\ y_N)$ to $(x_{i_j},\ y_{i_j}]$. The affine maps described above are often known as IFS interpolation[26], and the end-points are known as interpolation points. In Equation (5.1) the parameter d_j is known as the contraction factor for

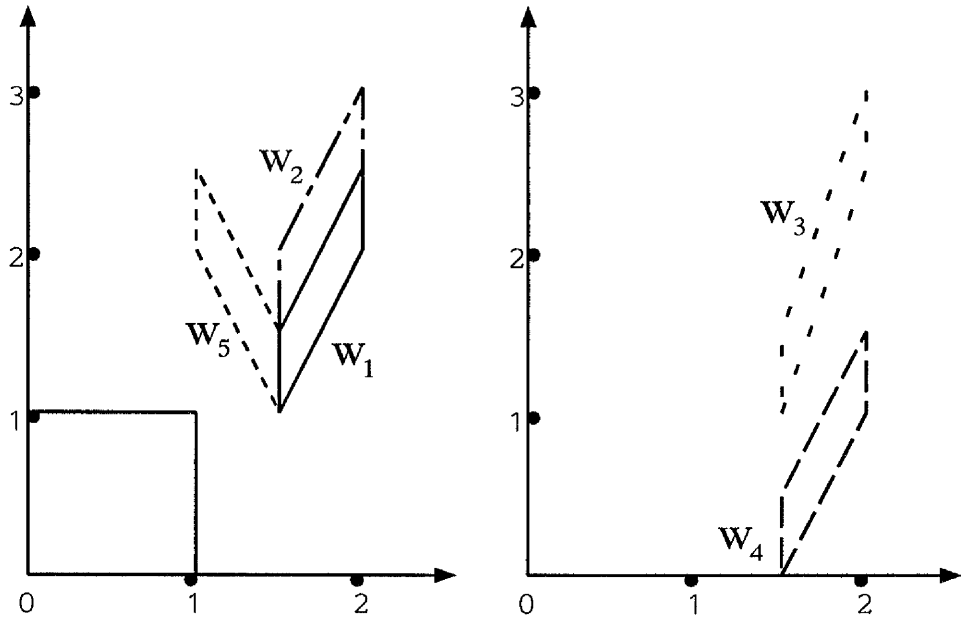

Fig. 31. Affine transformations w_1, w_2, w_3, w_4 applied to the unit square.

map j, and it must satisfy $|d_j| < 1$. With IFS interpolation the self-affine region is described as

$$\hat{R} = \{[i_{j-1}, i_j] : \ j = 0, 1, \cdots, M\}. \tag{5.2}$$

It is obvious that the maps are just touching, which means that overlap occurs only at interpolation points.

In this thesis we extend the idea and define what we shall call extended IFS interpolation. We construct a new self-affine region, based on each interpolation point (x_{i_j}, y_{i_j}) and its consecutive point (x_{i_j+1}, y_{i_j+1}) to construct the new set of self-affine regions for all j, except that the cases $j = 1$ and $j = M$ are treated differently.

$$\begin{aligned} R \ = \ & \{[i_{j-1}+1, i_j]\}, j = 2, 3, \cdots, M-1 \ \text{ and} \\ & \{[0, i_1]\}, \ \{[i_{M-1}+1, N]\}. \end{aligned} \tag{5.3}$$

For map parameter a_j, we also extend its range to $-1 < a_j < 1$ so that w_j maps (x_0, y_0) to $(x_{i_{j-1}+1}, \ y_{i_{j-1}+1})$ and (x_N, y_N) to $(x_{i_j}, \ y_{i_j})$, or (x_0, y_0) to $(x_{i_j}, \ y_{i_j})$ and (x_N, y_N) to $(x_{i_{j-1}+1}, \ y_{i_{j-1}+1})$, depending on the sign of a_j. Obviously, extended IFS interpolation involves a totally disconnected map.

We use examples to show the geometric properties of the map parameters in Figure 31. Suppose we have a unit square, the bottom left-hand corner of which is

located at $(0,0)$, and we use the following affine transformations to map the square:

$$w_1\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \frac{1}{2} & 0 \\ 1 & \frac{1}{2} \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} \frac{3}{2} \\ 1 \end{pmatrix}$$

$$w_2\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \frac{1}{2} & 0 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} \frac{3}{2} \\ 1 \end{pmatrix}$$

$$w_3\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \frac{1}{2} & 0 \\ \frac{3}{2} & \frac{1}{2} \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} \frac{3}{2} \\ 1 \end{pmatrix}$$

$$w_4\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \frac{1}{2} & 0 \\ 1 & \frac{1}{2} \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} \frac{3}{2} \\ 0 \end{pmatrix}$$

$$w_5\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} -\frac{1}{2} & 0 \\ 1 & \frac{1}{2} \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} \frac{3}{2} \\ 1 \end{pmatrix}.$$

In the case of maps w_2 and w_1 we see how the parameters d_j control the vertical contraction. Maps w_3 and w_1 illustrate how the parameters c_j control the rotation, and maps w_4 and w_1 illustrate how the parameters e_j control the translation. Maps w_5 and w_1 illustrate how the sign of a_j controls the mirror transform. Thus, extended IFS interpolation should provide a flexible fitting procedure.

The following Collage Theorem [20] gives a bound on the fidelity of a given signal with the IFS attractor.

Theorem 5.1 *Let (X,h) be a complete metric space, let L be a given function (signal) and let $\epsilon > 0$ be given. Choose an IFS $\{X;\ w_1, w_2, \cdots, w_M\}$ with contraction factor $\lambda = \max\{\lambda_j;\ j = 1,2,\cdots,M\}$ so that*

$$h(L, \bigcup_{j=1}^{M} w_j(L)) \leq \epsilon, \tag{5.4}$$

where h is the Hausdorff metric. Then

$$h(L,A) \leq \frac{\epsilon}{1-\epsilon}, \tag{5.5}$$

where A is the attractor of the IFS.

The Hausdorff metric is defined as follows,

Definition 5.1

$$\begin{aligned} h(A,B) &= \max\{\max\{\min\{d(x,y);\ y\in B\}:\ x\in A\}, \\ &\quad \max\{\min\{d(y,x);\ x\in A\}:\ y\in B\}\}, \end{aligned} \tag{5.6}$$

where A, B are sets of points and $d(\cdot,\cdot)$ is the distance between points.

Note that the Collage Theorem does not provide a procedure for constructing a map. It only provides us with a way of assessing the goodness-of-fit of an IFS without computing its attractor. From definition 5.1 we see that the Hausdorff metric involves heavy calculation[170]. In practice we can use an approximation method in place of precise computation to save the calculation time. The approximation method is to calculate the Hausdorff metric for each local neighbourhood, not for the whole space, and this was done in all the following numerical work.

Definition 5.2 *The local neighbourhood c of x_i can be expressed as $(x_i - c, x_i + c)$ where $c \ll N$ is called the local neighbourhood width and $x_i - c,\ x_i + c \in [0, N]$.*

Definition 5.3 *The approximate Hausdorff metric of a one-dimensional discrete signal is defined by*

$$\begin{aligned} h_a(A,B) &= \max\{\max\{\min\{d(x_i,x_j);\ x_j\in(x_i-c,x_i+c)\}:\ x_i\in A\}, \\ &\quad \max\{\min\{d(x_j,x_i); x_i\in(x_j-c,x_j+c)\}:\ x_j\in B\}\}, \end{aligned} \tag{5.7}$$

where A, B are subsets of R^1.

5.2.2. Estimation of affine transformation parameters

We essentially have an inverse problem: given a signal L, find an IFS for which L is the approximation of the IFS attractor. The main problem is to estimate the self-affine region, which is also determined by the index vector P of the interpolation points. Once we have estimated P, we can compute the parameters of the affine transformation, a_j, c_j, d_j, e_j, f_j, as follows.

Suppose we have a map w_j so that $w_j:\ [0,N] \Rightarrow (i_{j-1}, i_j]$, and $i_{j-1}, i_j \in P$. Then we have

$$\begin{aligned} a_j &= \frac{x_{i_j} - x_{i_{j-1}+1}}{x_N - x_0} \\ e_j &= \frac{x_N x_{i_j} - x_0 x_{i_{j-1}+1}}{x_N - x_0}. \end{aligned} \tag{5.8}$$

Since the map is a contraction in the x-axis direction, we should allow for the approximate calculation in which discrete data on a larger interval along the x-axis is mapped into a smaller interval. In practice, the method is to average y-values of points which are mapped into each destination point.

Define the set A_p by

$$A_p = \{j : \ p = \text{int}(a_j x_j + e_j)\}, \tag{5.9}$$

and let

$$\begin{aligned} \overline{x_p} &= \frac{\sum_{j \in A_p} x_j}{\text{number of points included in set } A_p}, \\ \overline{y_p} &= \frac{\sum_{j \in A_p} y_j}{\text{number of points included in set } A_p}. \end{aligned} \tag{5.10}$$

The least-squares estimates of c_j, d_j, and f_j are the minimizers of

$$E_j = \sum_{p \in (i_{j-1}, i_j]} (c_j \bar{x}_p + d_j \bar{y}_p + f_j - y_p)^2 \quad j = 1, 2, \cdots, M. \tag{5.11}$$

The stationarity equations are

$$\begin{bmatrix} \sum \bar{x}_p^2 & \sum \bar{x}_p \bar{y}_p & \sum \bar{x}_p \\ \sum \bar{x}_p \bar{y}_p & \sum \bar{y}_p^2 & \sum \bar{y}_p \\ \sum \bar{x}_p & \sum \bar{y}_p & \sum 1 \end{bmatrix} \begin{bmatrix} c_j \\ d_j \\ f_j \end{bmatrix} = \begin{bmatrix} \sum \bar{x}_p y_p \\ \sum \bar{y}_p y_p \\ \sum y_p \end{bmatrix}, \tag{5.12}$$

and we can solve (5.12) easily.

In other cases to be considered, where we have w_j such that $w_j : \ (x_0, y_0) \Rightarrow (x_{i_j}, \ y_{i_j})$ and $(x_N, y_N) \Rightarrow (x_{i_{j-1}+1}, \ y_{i_{j-1}+1})$, we need only interchange x_{i_j} and $x_{i_{j-1}+1}$ in Equation (5.8). The Equations (5.12) are unchanged.

5.2.3. Suboptimal Algorithm for the Inverse Extended IFS Interpolation Problem

In order to choose interpolation points optimally, we have to minimize the objective function

$$\min h(L, \sum_{i=1}^{M} w_j(L)), \tag{5.13}$$

where L is the given one-dimensional discrete signal, and w_j is determined by the index vector P of interpolation points and by equation (5.12). This is a global optimization problem. As a result of the required scale of computation, there is no acceptable algorithm for obtaining the globally optimal solution. However, there is a method, based on local search, which achieves an acceptable solution, as justified empirically

in the following simulation section. The method is based on the following remark.

> First we note that M is known implicitly once P is determined. Secondly, each i_j of P is an integer that satisfies $0 \leq i_j \leq N$, and $i_1 < i_2 < \cdots < i_{M-1}$, so that we can first search for i_1, then i_2, and so on.

We modify the global objective function (5.13) to the local objective function

$$\begin{aligned} \min \quad & h(L(R_j), w_j(L)) \text{ or } \min h(L(i_{j-1}, i_j], w_j(L)), \\ & j = 1, 2, \cdots, M, \ \textit{sequentially}, \end{aligned} \tag{5.14}$$

where $L(R_j)$ and $L(i_{j-1}, i_j]$ are the data which belong to the self-affine region R_j, each w_j maps L into region R_j, and the self-affine region R_j is defined by equation (5.3). The corresponding inverse algorithm can be described as follows.

Algorithm 5.1. Inverse Extended IFS Interpolation Algorithm.

INPUT: $(x_0, y_0), \cdots, (x_N, y_N)$ and W.
OUTPUT: the number M and the IFS maps w_j, $j = 1, 2, \cdots, M$.

1. Initialize interpolation point indices $i_0 = 0$, $i_M = N$ for $a_j > 0$ and $i_0 = N$, $i_M = 0$ for $a_j < 0$. For the index i_1 of the other interpolation point of self-affine region R_1, set the limits $[s, e]$ of the search space, where the integer s and e satisfy $x_s = x_0 + W$ and $x_e = x_N - W$.
2. Estimate the interpolation point index i_j, $j = 1, 2, \cdots, M-1$, and construct the self-affine region R_j using equation (5.3) for both $a_j > 0$ and $a_j < 0$.
   - 2.1 For each element c in $[s, e]$ construct the temporary interpolation region $\{(x_{i_{j-1}+1}, y_{i_{j-1}+1}), (x_c, y_c)\}$. Apply equations (5.8) and (5.12) to estimate the parameters of the map w_j, compute the approximate Hausdorff metric, and store the Hausdorff error in BUFFER[c].
   - 2.2 Choose, as the candidate interpolation index, i_j in $[s, e]$ such that BUFFER[i_j] is minimum.
   - 2.3 Choose the minimum from the $a_j > 0$ case and the $a_j < 0$ case and determine the sign of a_j.
3. If $i_{j-1} \neq 0$, construct the self-affine region $\bar{R}_j = (i_{j-1}, i_M]$ and estimate the parameters of the map $\bar{w}_j$.
4. If $i_{j-1} \neq 0$ and if $h(L(i_{j-1}, i_M], \bar{w}_j(L)) < h(L(i_{i-1}, i_j], w_j(L))$, discard the candidate interpolation point index i_j. Exit from the algorithm.

5. Accept the candidate interpolation point index i_j and identify the new interpolation point (x_{i_j}, y_{i_j}). Construct the self-affine region $R_j = \{(x_{i_{j-1}+1}, y_{i_{j-1}+1}), (x_{i_j}, y_{i_j})\}$.
6. Update the search limit $s = i_j + W + 1$.
7. If $e \leq s$, exit from the algorithm.
8. Return to step 2.

In the algorithm described above, BUFFER is a one-dimensional array, and W is a input constant, known as the minimum self-affine region width, which is not required when modelling a continuous signal. However, when we try to build a model for a discrete signal, W is required since a region containing only two points is self-affine to any signal and the fit is perfect. W has some influence on the value taken by M. The larger W we use, the smaller is the resulting M. The choice of a specific value is, in practice, not a sensitive one. In the following simulation section, we chose $W = 9$. There are two ways of exiting from the algorithm. One is at step 7 and occurs if no further self-affine region wider than W exists. The other is at step 4 and occurs if inclusion of another interpolation point will increase the error of fitting the given signal.

Algorithm 5.1 emphasizes the fidelity of fitting the given signal. In order to emphasize data compression, we can revise step (2.2) so as to satisfy some prescribed tolerance in the choice of the value from BUFFER that allows a larger region width.

5.2.4. Enhancement of the Robustness of the Inverse Algorithm

Algorithm 5.1 is naturally sensitive to the given signal, as seen later in the examples in Section 5.4. In the estimation of each self-affine region, the point that minimizes Hausdorff error may be not a valid interpolation point. Suppose, for example, that $(i_{j-1}, i_j]$, with $i_{j-1}, i_j \in P$, is a valid self-affine region. It is possible that there exists a point i_p, where $i_p \in P$ and $i_{j-1} < i_p < i_j$, such that $h(L(i_{j-1}, i_p], \bar{w}_i(L)) < h(L(i_{j-1}, i_j], w_j(L))$. This may occur, in particular, if the given signal is approximately self-affine or non self-affine. One way to avoid this is to use instead the"next best" as a minimizing point i_s. At the next step, and after computing the self-affine region based on point i_j, choose the point i_s also as one of the interpolation points, compute a new self-affine region based on the point i_s and locate the new minimum error point $\bar{i}_{j+1}$. If

$$h(L(i_{j-1}, i_j], w_j(L)) + h(L(i_j, i_{j+1}), w_{i+1}(L)) > h(L(i_{j-1}, i_s], \bar{w}_i(L)) + h(i_s, \bar{i}_{j+1}], \bar{w}_{i+1}(L)), \tag{5.15}$$

then discard the interpolation point indices i_j, i_{j+1}, replacing them by $i_s, \bar{i}_{j+1}$. The robustness of the method follows from the fact that, if i_j is not a valid interpolation index but i_s is, the self-affine region $(i_j, i_{j+1}]$ based on the point index i_j produces larger Hausdorff error, and the self-affine region $(i_s, \bar{i}_{j+1}]$ based on the index i_s will keep the Hausdorff error at a low level since it is a valid self-affine region. Thus inequality (5.15) is true if the Hausdorff error of region $(i_j, i_{j+1}]$ is large enough.

The robust algorithm can be described as follows.

Algorithm 5.2. Robust Inverse Extended IFS Interpolation Algorithm

INPUT: $(x_0, y_0), \cdots, (x_N, y_N)$ and W.
OUTPUT: the number M and the IFS maps w_j, $j = 1, 2, \cdots, M$.

1. Step 1 is the same as for Algorithm 5.1.
2. Step 2 is the same as for Algorithm 5.1.
   - 2.1 Step (2.1) is the same as for (2.1), (2.2), and(2.3) of Algorithm 5.1.
   - 2.2 Choose, as the possible alternative index, i_s from $(i_j, e]$ such that BUFFER$[i_s]$ is minimum.
   - 2.3 If $i_{j-1} \neq 0$ and $i_{s-1} + W < e$ then
     - 2.3.1. Set new limits of the search space $(i_{s-1}, e]$ for both the cases $\bar{a}_j > 0$ and $\bar{a}_j < 0$.
     - 2.3.2. For each element $\bar{c}$ in $(i_{s-1}, e]$ construct the temporary region $\{(x_{i_{s-1}+1}, y_{i_{s-1}+1}), (x_{\bar{c}}, y_{\bar{c}})\}$. Apply equations (5.8) and (5.12) to estimate the parameters of map $\bar{w}_j$, compute the approximate Hausdorff metric, and store the Hausdorff error in BUFFER$[\bar{c}]$.
     - 2.3.3. Choose, as the candidate interpolation index, $\bar{i}_j$ in $(i_{s-1}, e]$ such that BUFFER$[\bar{i}_j]$ is minimum.
     - 2.3.4. Choose the minimum from the $\bar{a}_j > 0$ case and the $\bar{a}_j < 0$ case and determine the sign of $\bar{a}_j$.
     - 2.3.5. Choose, as the possible alternative index, i_s in $(\bar{i}_j, e]$ such that BUFFER$[i_s]$ is minimum.
     - 2.3.6. If $h(L(i_{j-2}, i_{j-1}], w_{j-1}(L)) + h(L(i_{j-1}, i_j), w_j(L)) > h(L(i_{j-2}, i_{s-1}], \bar{w}_{j-1}(L)) + h(i_{s-1}, \bar{i}_j], \bar{w}_j(L))$, then set $i_{j-1} = i_{s-1}$ and $i_j = i_s$.
3. Steps 3-8 are the same as for Algorithm 5.1.

Algorithm 5.2 is similar in structure to the search algorithm in [134]. However, in algorithm 5.2 we use extended IFS interpolation to get a better fit and we store the possible alternative indices to enhance the robustness of the algorithm.

5.3. Distributed Parallel Computing for the IFS Model of a Given Signal

5.3.1. Distributed Parallel Computing Based on Remote Procedure Call (RPC)

The essence of distributed parallel computing (DPC) is that many autonomous general computers, connected by a communications medium of which the most popular is *Ethernet*, cooperate in dealing with a single computing task. Each computer has its own independent memory, processor and ability to communicate.

The basic method of DPC is the client-server model. A single server works for clients who have special computational demands. After completing one client's task, the server waits for the next.

One way to convert a sequential algorithm into a DPC algorithm is as follows.

- Select a basic subtask as the server task in order that the number of servers can be determined and assign to each server one client.
- Use one computer as a control unit to manage communication among clients and servers and to synthesize the data resulting from the different servers.

This multi-clients-multi-servers model is shown in Figure 32.

Remote Procedure Call (RPC) is a high-level message-passing paradigm which allows network applications to be developed by way of specialized kinds of procedure calls designed to hide the details of the underlying networking mechanisms. The net effect of programming with RPC is that programs are designed to run within a client/server network model. With RPC, the client makes a procedural call which sends requests to the server as necessary and it then awaits the result from the server. When these requests arrive, the server calls a dispatch routine, performs whatever service is requested, sends back the reply, and the procedural call returns to the client as shown in Figure 33.

RPC uses XDR (eXternal Data Representation) routines to convert procedure arguments and results into network format and vice-visa. Each RPC procedure is uniquely defined by a program number, version number, and procedure number. The program number and version number specify a group of related remote procedures, each of which has a different procedure number.

The details of programming in applications of RPC can be tedious. One of the more difficult areas is writing XDR routines. Fortunately, the compiler *rpcgen* exists to help programmers write RPC applications simply and directly. It accepts a remote program interface definition written in a language, called RPC language[136], which is similar to C; see subsection 5.3.2. However, it only supports a one-client-one-server model, and we have to use a text editor to modify it for our multi-clients-multi-servers

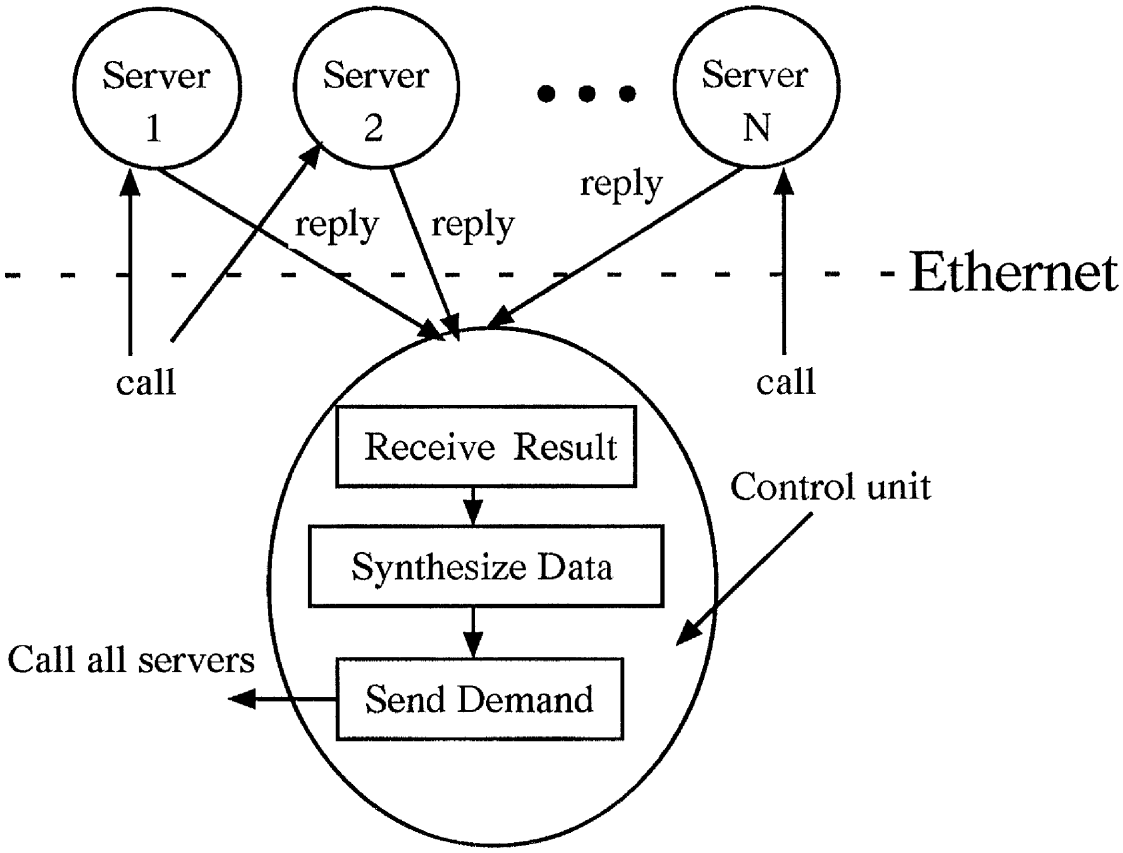

Fig. 32. Distributed Parallel Computing Model of multi-clients-multi-servers.

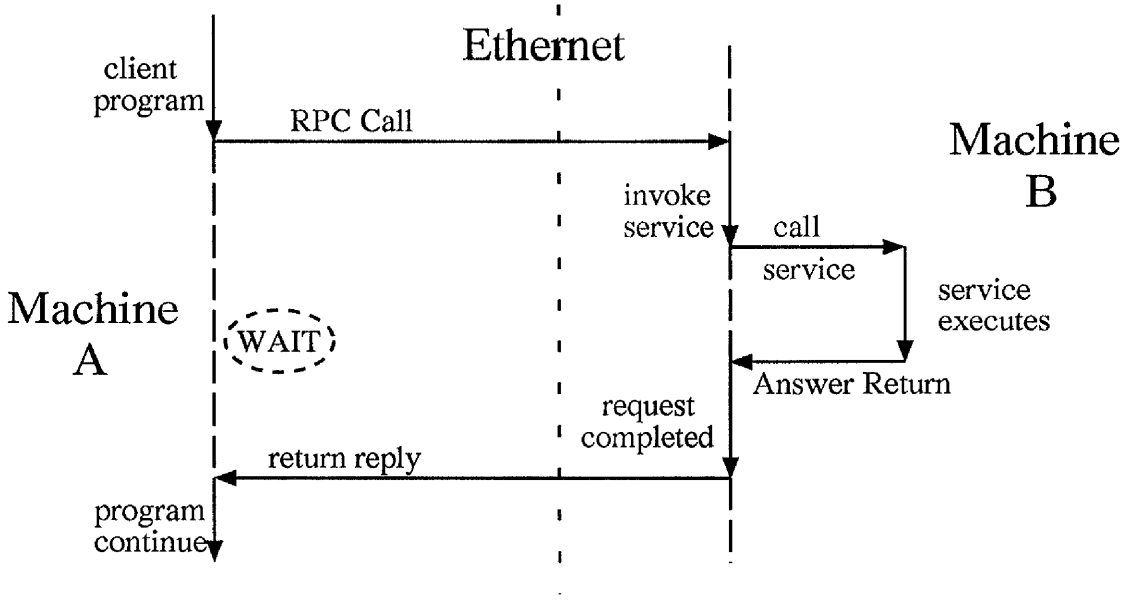

Fig. 33. RPC programming model.

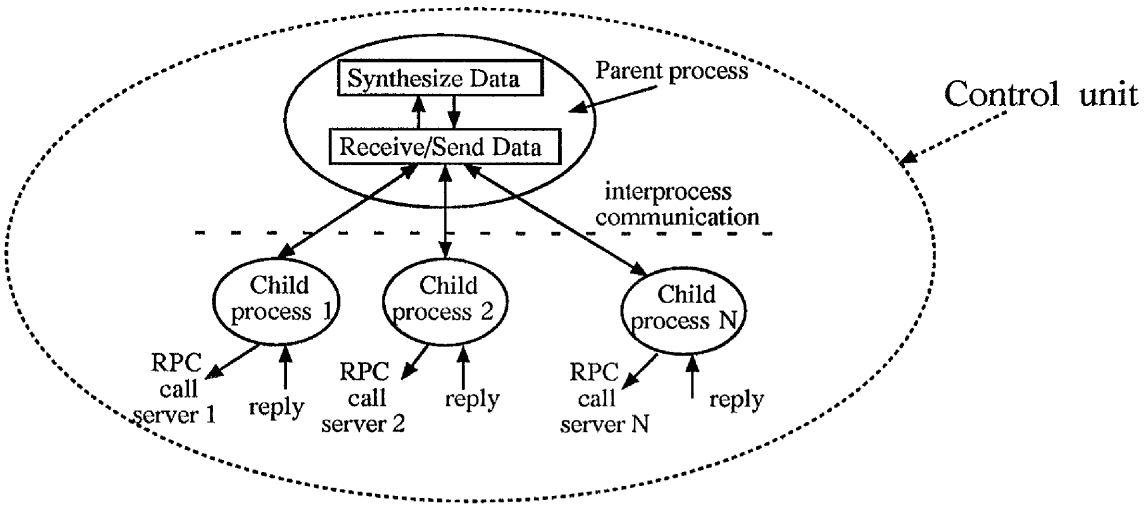

Fig. 34. Control unit's parent- and child-process

model. The output of **rpcgen** includes client routines, a server skeleton, XDR filter routines for both procedure parameters and results, and a header file that contains common definitions.

In normal **RPC**, clients send a call and wait for the server to reply to the effect that the call has succeeded. This implies that clients do not compute while servers are processing a call, which also means that clients cannot work in parallel in one computer simultaneously. We should utilize the UNIX concurrent process ability. First, we can build multi-child processes. Each child-processor runs a RPC client associated with the special server, and the parent processor processes the data synthesis and interprocessor communication as shown in Figure 34

The system call "fork()" in UNIX can build a child process, which returns zero in the child process and returns non-zero, which is the child process identifier, in the parent. The simplest but slowest method of interprocess communication is through *file*. Alternatively one might use *pipe* and *named pipe*, which employ the basic stream model used for file input/output. A more advanced method is *Message*. A *message* queue identifier *msqid* is a unique positive integer created by the "msgget()" system call. Each *msqid* has a message queue and data structure associated with it. The system call "msgctl()" can destroy a message *msqid*. The system call "msgsnd()" can send a message to other processes and the system call "msgrcv()" can receive special kinds of message.

5.3.2. Implementment of the Distributed Parallel Algorithm

In order to utilize the powerful ability of multi-computer processors, a server task should be a massive floating-point calculation task and not just logical decision-making or I/O processing. In Algorithm 5.2, the massive calculation comes from step 2.1 and step 2.4.2, which use equations (5.8) and (5.12) to compute the parameters of the affine map. Since communication among computers is a slower operation than that of calculation, we use larger task granularity, see subsection 3.1.3, to reduce the quantity of communication operations in order to construct a good distributed parallel algorithm.

We select the procedure of estimation of the affine map parameters in some interval as the server task according to the number of computers connected by Ethernet and each server processes the same length of search space. Suppose we have three computers. We choose one as the control unit and the other two as servers. In order to keep the control unit busy in computing, we have to allocate some computing task to it. For example we may assign the search region $[s,\ k_c \cdot e]$ to the control unit. One server deals with the search region $[k_c \cdot e + 1,\ \frac{k_c \cdot e+e+1}{2}]$, and other server deals with the search region $[\frac{k_c \cdot e+e+3}{2}, e]$. The constant k_c controls the distribution of the tasks between the control unit and the servers.

When each demand is sent, the only information needed by the servers is the latest interpolation index i_{j-1}. Given the constant W we can easily construct the search region as $[i_{j-1} + W + 1,\ N - W]$. The results from the server include the minimum Hausdorff error and its associated index and the next to minimum Hausdorff error and its associated index. The parallel protocol written in the **rpcgen** language is as follows:

```
/* priei.x: Parallel Robust Inverse Extended IFS
                          Interpolation Protocol */
/* define a variable named "poserr" */
typedef struct int_float  poserr;
/* data structure of */
struct int_float { int pos0; int pos1; int mirror;
 float err0;  float err1; };       /* computing result */
program PRIEIPROG {
    version PRIEIVERS {
       /* The following is a RPC procedure, named SEIFSP,
          argument type is integer,
          and return value is a struct named "poserr"  */
          poserr SEIFSP(int) = 1;  /* procedure number */
    } = 1;                         /* version number   */
} = 0x20000999;                    /* program number   */
```

The suboptimal search algorithm from the server's point of view is as follows.

Algorithm 5.3. Server's Contribution to the Robust Inverse Extended IFS Interpolation Algorithm

INPUT: $(x_0, y_0), \cdots, (x_N, y_N)$, the number of servers, SV_n, the constant, W, and the constant, k_c. The latest interpolation index, i_{j-1}.
OUTPUT: the position of the minimum and the next to minimum Hausdorff error and their indices.

1. Initialize RPC server program.
2. Repeat until the condition of new RPC demand coming is TRUE. Then call the service procedure SEIFSP.
   - 2.1 In SEIFSP, set up the search region of the servers: $s = i_{j-1} + W + k_c\frac{(N-i_{j-1}-2W)}{(SV_n+k_c)} + 1$, $e = s + \frac{(N-2W)}{(SV_n+k_c)}$.
   - 2.2 For each integer c in $[s, e]$ construct the temporary self-affine region $\{(x_{i_{j-1}+1}, y_{i_{j-1}+1}), (x_c, y_c)\}$. Apply equations (5.8) and (5.12) to estimate the parameters of the map w_j, compute the approximate Hausdorff metric, and store the Hausdorff error in BUFFER[c] for both $a_j > 0$ and $a_j < 0$.
   - 2.3 Choose the i_j from $[s, e]$ for which BUFFER[i_j] is minimum as the candidate interpolation index.
   - 2.4 Choose the minimum from the $a_j > 0$ case and the $a_j < 0$ case and determine the sign of a_j. Store the minimum error and the index to return **struct** "*poserr*".
   - 2.5 Choose the i_s from $(i_j, e]$ for which BUFFER[i_s] is the minimum value as the next-to-minimum error index. Store the error and index to return **struct** "*poserr*".
3. Answer the RPC and return the computing result **struct** "*poserr*".
4. Return to step 2.

The suboptimal search algorithm for the clients consists of parent and child algorithms. The parent algorithm is described as follows.

Algorithm 5.4. Client's Contribution to Robust Inverse Extended IFS Interpolation Parent Algorithm.

INPUT: $(x_0, y_0), \cdots, (x_N, y_N)$, the number of servers, SV_n, the constant, W, the constant, k_c, the servers name.
OUTPUT: the number M and the IFS maps w_j, $j = 1, 2, \cdots, M$.

1. Initialize the **message** structure and build the child process. Initialize the interpolation point indices $i_0 = 0$ and $i_M = N$. For the other interpolation point index i_1 of the self-affine region R_1, set the limit of the search region $[s, e]$, where integer s and e satisfy $x_s = x_0 + W$ and $x_e = x_N - W$.

2. Broadcast the index $i_{j-1}, j = 1, 2, \cdots, M-1$, to all child processes.
3. As in step 2.1 - step 2.3 of algorithm 2, to compute the minimum index $\bar{i}_j$ and the next to minimum index $\bar{i}_s$ in the search space $[s,\ k_c \cdot e]$.
4. Receive the results from other child processes.
   - 4.1 Set the message size, state, and message identifier.
   - 4.2 Apply system call "msgrcv()" to receive messages.
   - 4.3 If no message is received after five attempts, suspend the parent process.
5. Compare all results and choose the minimum and the next-to-minimum Hausdorff errors and the interpolation point indices i_j, i_s.
6. If $i_{j-1} \neq 0$ and $i_{s-1} + W < e$ then :
   - 6.1 Set new limits for the search region $(i_{s-1}, e]$.
   - 6.2 Step (6.2) is the same as Step 2.
   - 6.3 As in step 2.3.2 - step 2.3.4 of algorithm 2, compute the minimum and the next to minimum interpolation point indices $\hat{i}_j$, $\hat{i}_s$ in the search space $[s,\ k_c \cdot e]$.
   - 6.4 Step (6.4) is the same as Step 4.
   - 6.5 Compare all results and choose the minimum and the next-to-minimum Hausdorff error and interpolation point index $\bar{i}_j, \bar{i}_s$.
   - 6.6 Choose the minimum from the $a_j > 0$ case and the $a_j < 0$ case and determine the sign of a_j.
   - 6.7 If $h(L(i_{j-2}, i_{j-1}], w_{j-1}(L)) + h(L(i_{j-1}, i_j), w_j(L)) > h(L(i_{j-2}, i_{s-1}], \bar{w}_{j-1}(L)) + h((i_{s-1}, \bar{i}_j], \bar{w}_j(L))$, then set $i_{j-1} = i_{s-1}$ and $i_j = i_s$.
7. Step 7-11 are the same as Steps 3-8 in Algorithm 5.1.

All child-process algorithms are the same. We describe one as follows.

Algorithm 5.5. Client's Contribution to the Robust Inverse Extended IFS Interpolation Child Algorithm

INPUT: Server name and the latest interpolation point index i_{j-1}.
OUTPUT: The minimum and the next to minimum point indices and their Hausdorff errors.

1. Initialize the client side of the RPC.

2. Repeat indefinitely the following.

   2.1 Set the message size, state, and message identifier.

   2.2 Repeatedly apply the system call "msgrcv()" until i_{j-1} is received.

   2.3 Execute the RPC procedure to the corresponding server and await its return.

   2.4 Set the message size, state, and message identifier.

   2.5 Apply system call "msgsnd()" to send the RPC result.

   2.6 Awaken the parent process to receive the result.

5.4. Numerical Simulation of Iterated Function System Model

We use four numerical examples to test algorithm 5.1 and algorithm 5.2 with $W = 9$. For strictly self-affine data generated by a self-affine IFS map, the standard algorithm (algorithm 5.1) and the robust algorithm (algorithm 5.2) produced the same result, matching the original data. For approximately self-affine data, however, they gave different results. The "approximate" data are produced by subsampling strictly self-affine data. For example, we sample every other point in a 512-point strictly self-affine data set to produce an approximately self-affine data set of length 256.

Example 5.1 is based on approximately self-affine data produced by large contraction factors d_j and sparse interpolation points with sample rate 50%. We find a solution by trying $M = 2, 3, 4, \cdots$. The best result is obtained with $M = 5$ with the robust algorithm (algorithm 5.2) and $M = 12$ with the standard algorithm (algorithm 5.1) as shown in Figure 35 (top picture) and Table VIII (top). Note that the layout of Table VIII is such that, to save space, each row includes two self-affine regions. The interpolation points' indices are listed in the first column of Table VIII. If the first index is greater than the second one, it means that the parameter a_j of the affine map w_j is negative. In Table VIII the last column is the signal-to-noise ratio (**SNR**), defined by

$$\mathbf{SNR} = -10 \times log\left(\frac{(\text{Original Data - Produced Data})^2}{(\text{Original Data})^2}\right).$$

In the results for the robust algorithm, the interpolation points' indices are the same as in the original, and the affine map parameters d_j are almost the same. The **SNR**, indicating fidelity of fit to the data, are about 15db, which is acceptable. In the results from the standard algorithm, the number of interpolation points is more than the number from the original, which does not represent good data compression. The map parameters d_j are not the same as with the original, and the fidelity of fit to the

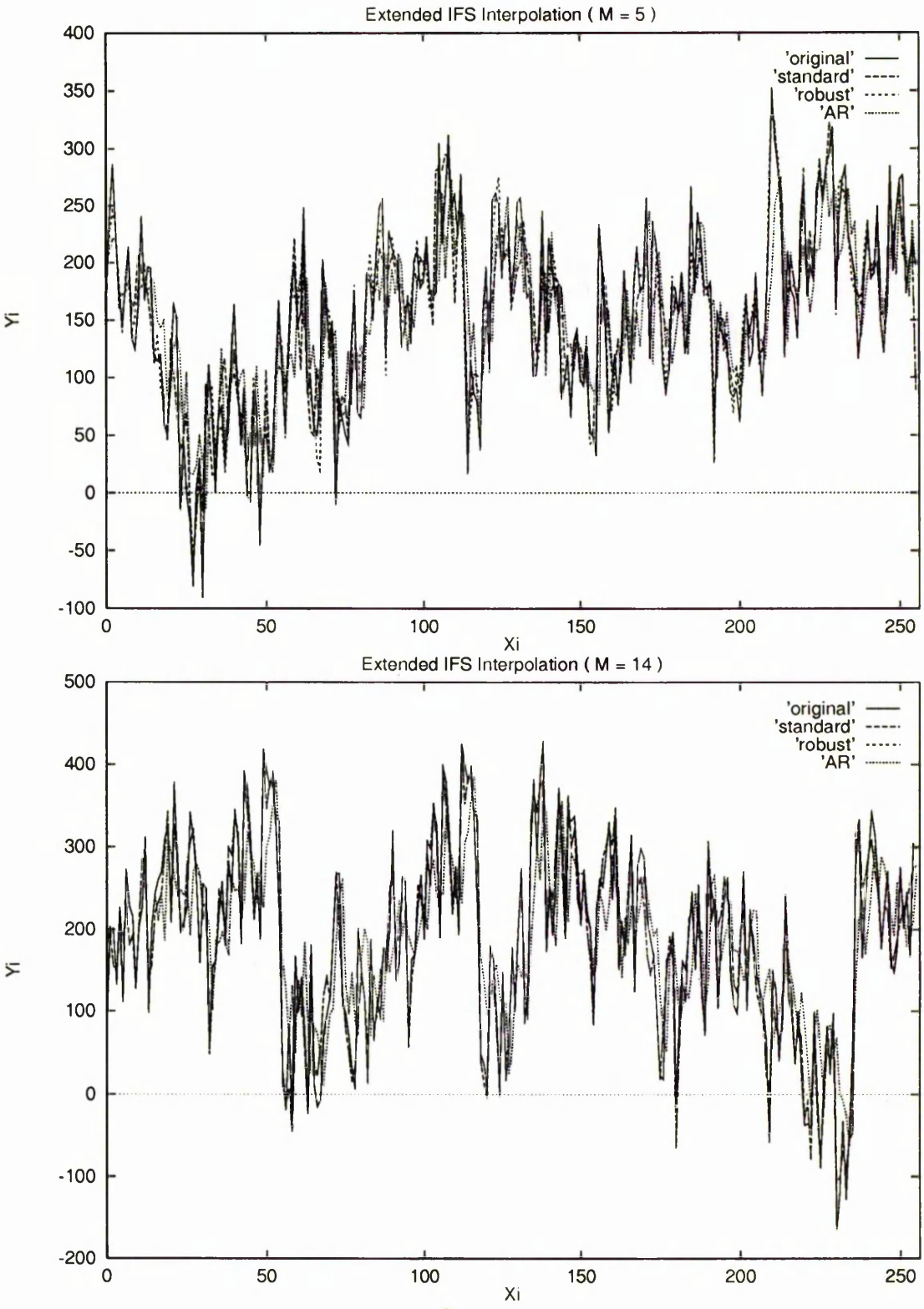

Fig. 35. Inverse IFS Interpolation with $M = 5$ (top) and $M = 14$ (bottom) with Large d_j Approximately Self-affine Data (50% sample).

Table VIII. Original and Calculated IFS Interpolation Point Indices, Map parameters, Hausdorff Error, Signal-to-noise Ratio of Large d_j for Approximately Self-affine Data

	Index	Map Param.	H	SNR	Index	Map Parame.	H	SNR
Orig.	22,0,	.15 .92 5.8			67,23	-.17 -.89 224.0		
	68,113	.24 .9 5.3			114,209	.35 -.93 246.9		
	210,255	-.29 .98 148.5						
Robust	22,0	.18 .71 29.3	57.6	15.5	67,23	-.13 -.95 226.9	46.6	12.6
	68,113	.2 .87 12.6	56.9	17.7	114,209	.36 -.94 245.8	68.3	15.9
	210,255	-.58 .95 150.7	47.3	19.3				
Stand.	9,0	.52 -.7 233.1	59.8	15.4	19,10	.66 -.26 103.9	27.1	13.8
	29,20	.4 .74 -124.0	46.6	5.2	30,40	.75 -.93 108.1	54.6	6.1
	50,41	-.06 .71 -58.4	28.5	5.2	61,51	-.29 -.77 267.9	56.9	14.6
	62,81	-.57 .97 27.3	67.0	8.7	104,82	.22 -.39 219.0	72.7	15.9
	115,209	.22 -.7 228.8	73.8	11.6	210,255	-.58 .95 150.7	52.8	16.0
Orig.	0,11	.16 .89 8.0			29,12	.06 .84 97.7		
	54,30	-.46 .92 150.0			55,75	.28 -.95 216.2		
	76,92	.37 -.83 231.6			117,93	-.44 .92 155.8		
	118,133	.3 -.85 219.1			134,157	-.21 .91 123.6		
	158,178	-.48 .87 118.5			193,179	-.26 .94 34.4		
	194,211	-.28 .89 21.0			235,212	.53 -.91 125.4		
	236,246	.03 .9 81.3			255,247	-.46 .81 141.9		
Robust and Stand.	0,11	.19 .52 64.7	48.0	14.8	29,12	-.07 .76 121.0	5.5	16.2
	54,30	-.52 .78 191.4	46.0	21.6	55,75	.5 -.8 172.2	45.9	13.0
	76,92	.57 -.63 167.7	6.6	13.6	117,93	-.51 .78 197.4	45.9	21.7
	118,133	.45 -.72 178.2	71.9	13.7	134,157	-.42 .73 192.4	51.4	18.5
	158,178	-.68 .74 157.5	48.5	14.6	193,179	-.4 .97 29.5	59.6	16.9
	194,211	-.52 .68 91.8	6.6	13.5	235,212	.56 -.82 100.8	56.7	13.2
	236,246	-.24 .56 194.4	72.3	17.2	255,247	-.51 .08 262.3	6.5	14.9

Table IX. Original and Calculated IFS Interpolation Points Indices, Map parameters, Hausdorff Error, Signal-to-noise Ratio of Small d_j for Approximately Self-affine Data

	Index	Map Param.	H	SNR	Index	Map Param.	H	SNR
Orig.	22,0	.32 .08 67.2			67,23	-.34 -.04 161.9		
	68,113	.41 .05 67.4			114,209	.19 -.16 190.7		
	210,255	-.38 .04 217.1						
Robust and Stand.	22,0	.33 .05 69.9	1.3	42.7	67,22	-.33 -.05 161.7	1.0	44.8
	68,113	.41 .06 67.2	.9	45.7	114,209	.19 -.14 188.9	1.3	48.7
	210,255	-.37 .04 215.9	1.3	47.9				
Orig.	0,11	.32 .09 66.4			29,12	.21 .07 153.9		
	54,30	-.29 .12 208.2			55,75	.1 -.05 151.7		
	76,92	.21 -.03 173.2			117,93	-.27 .05 219.4		
	118,133	.14 -.05 160.7			134,157	-.05 .11 182.0		
	158,178	-.32 .07 176.9			193,179	-.08 .04 100.1		
	194,211	-.12 .09 79.4			235,212	.36 -.11 67.0		
	236,246	.19 .1 139.7			247,255	-.33 .13 191.5		
Robust and Stand.	0,11	.31 .07 67.9	2.2	39.7	29,12	.2 .07 155.8	4.4	4.5
	54,30	-.29 .11 209.0	1.8	45.0	55,75	.1 -.04 149.8	1.7	44.4
	76,92	.2 -.02 172.5	1.1	47.8	117,93	-.27 .05 219.3	1.1	47.5
	118,133	.14 -.04 158.8	1.3	45.3	134,157	-.06 .09 186.3	3.0	43.3
	158,178	-.32 .06 177.7	1.9	41.8	193,179	-.07 .04 100.6	1.0	42.0
	194,211	-.13 .07 82.9	3.0	35.0	212,235	.36 -.1 65.5	2.1	39.1
	236,246	.16 .7 150.2	3.6	39.4	247,255	-.32 .05 199.9	3.9	36.3

data is not good in the (20,29) and (40,51) regions, in which the **SNR** is only about 5db.

Example 5.2 involves approximately self-affine data produced by large contraction factors d_j and dense interpolation points, with sample rate 50%. We try to search for a solution using $M = 2, 3, 4, \cdots$. The best result is obtained with $M = 14$ for both the standard and robust algorithms in Figure 35 (bottom picture) and Table VIII (bottom). In the results the number of interpolation points is the same as for the original, and the map parameters d_j are nearly equal to those in the original. The measures of fidelity of fit to the data are close to 13db, which is acceptable.

Example 5.3 involves approximately self-affine data produced by small contraction factors d_j and sparse interpolation points with sample rate 50%. We try to search for a solution using $M = 2, 3, 4, \cdots$. The best result is obtained with $M = 5$ for both the standard and robust algorithms; see Figure 36 (top picture) and Table IX (top). In the results the number of interpolation points is the same as in the original and the parameters of the map d_j are close to those in the original. The measures of fidelity of fit to the data, are about 45db, which is very good.

Example 5.4 involves approximately self-affine data produced by small contrac-

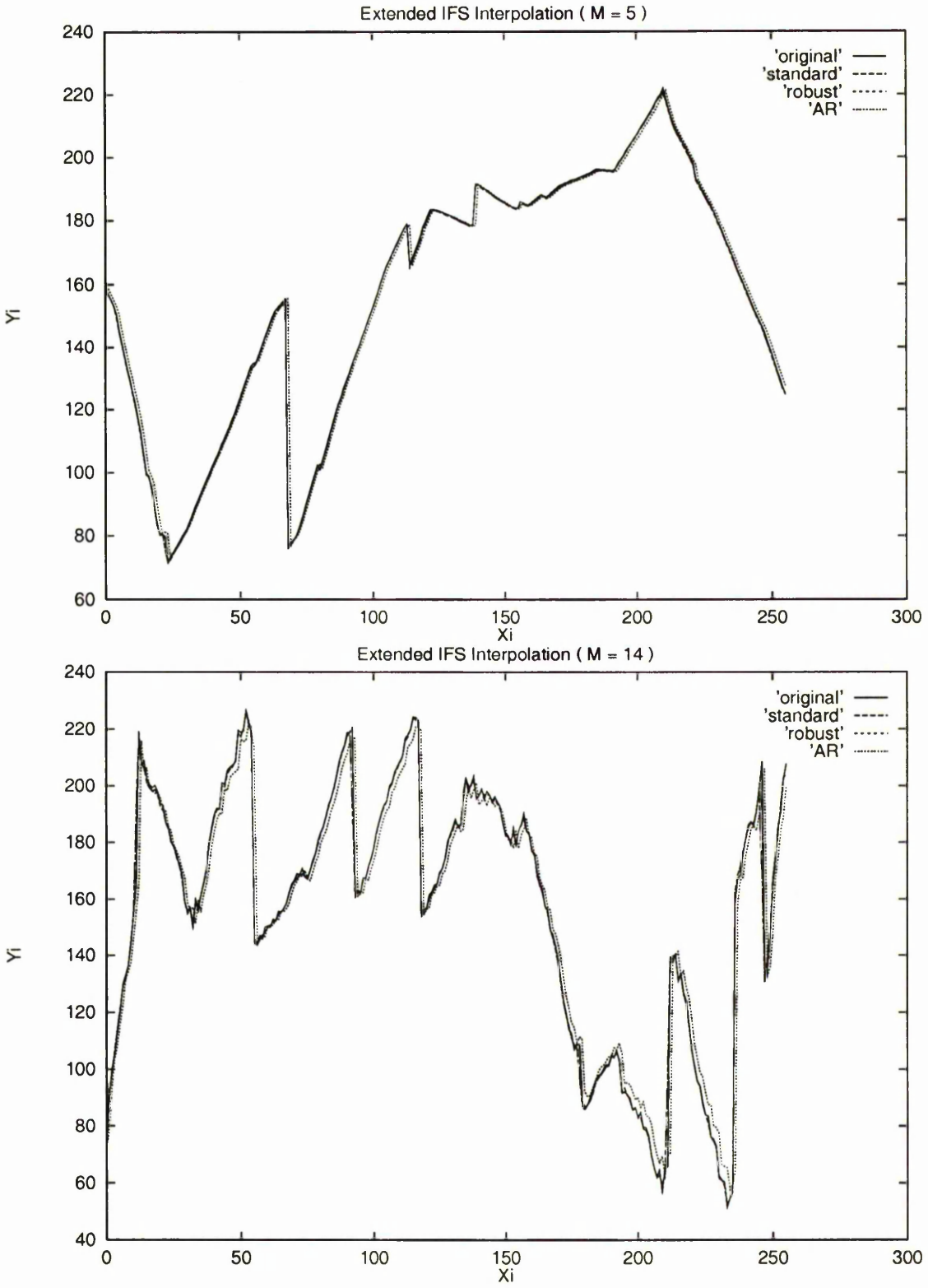

Fig. 36. Inverse IFS Interpolation of $M = 5$ (top) and $M = 14$ (bottom) with small d_j Approximately Self-affine Data (sampled at 50%).

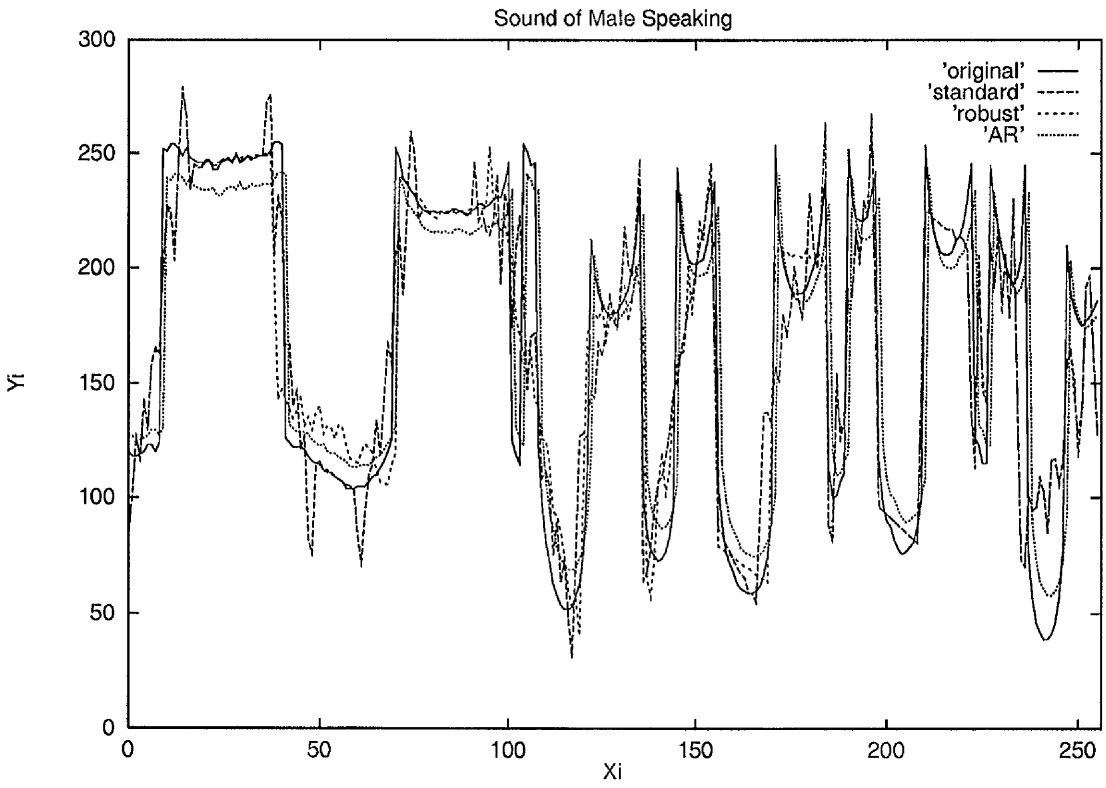

Fig. 37. Estimated IFS fitted curves for male speaking data.

tion factors d_j and dense interpolation points with sample rate 50%. We search for a solution using $M = 2, 3, 4, \cdots$. The best result occurs with $M = 14$ for both the standard and robust algorithms; see Figure 36 (bottom picture) and Table IX (bottom). The number of interpolation points is the same as for the original and the parameters of the map d_j are nearly equal to those of the original. The measures of fidelity of fit to the data are about 35db, which is very good. Since the parameters of the IFS maps are almost the same as the original one in Example 5.3 and 5.4, the curves produced by the standard and robust algorithm are almost coincident with the original data in Figure 36.

In comparing these results, note that the larger the vertical contraction factors d_j are, the larger are the resulting Hausdorff errors and the smaller are the **SNR**. The closer to self-affinity the given signal is, the less are the Hausdorff errors of the extended IFS interpolation model. The larger the distance between two consecutive interpolation points, the more likely it is that the standard algorithm does not converge. When the standard suboptimal search algorithm does not converge, the robust suboptimal search algorithm can converge to the best solution.

We now consider data from male speech in Example 5.5 and apply the standard

Table X. Calculated IFS Interpolation Points Indices, Map parameters, Hausdorff Error, Signal-to-noise Ratio for Male Speech, Non Self-affine Data

	Index	Map Param.	H	SNR	Index	Map Param.	H	SNR
S	16,0	-.69 .38 203.1	52.1	16.4	17,35	.01 -.02 249.2	1.9	47.4
t	36,48	-.69 .43 196.1	43.5	16.3	49,60	-.05 .01 113.8	1.7	39.8
a	76,61	-.67 .35 189.2	48.2	16.0	89,77	.004 .01 223.2	1.9	47.8
n	101,90	.12 .4 141.5	64.2	19.2	102,118	-.67 .38 146.5	73.6	8.6
d	135,119	-.44 -.25 268.8	47.5	15.6	136,155	.71 -.07 71.4	75.4	13.4
a	167,156	.15 .03 47.2	14.1	2.1	184,168	-.44 -.33 29.8	75.5	13.9
r	185,197	.63 -.46 170.0	42.9	16.5	209,198	.08 .0007 79.6	2.0	18.9
d	221,210	.06 .03 205.2	27.1	24.7	222, 234	.19 -.68 268.4	45.3	14.6
	255,235	-.33 .45 90.3	142.5	7.14				
	16,0	-.69 .38 203.1	52.1	16.4	17,38	.02 -.01 245.4	4.8	42.6
R	70,39	.13 -.1 126.9	111.44	1.5	71,81	-.09 -.02 246.1	4.5	39.5
o	82,94	.01 .01 222.2	2.3	45.0	95,120	-.87 -.14 278.3	81.4	1.6
b	136,121	-.04 .18 157.7	68.1	15.2	137,155	.71 -.25 103.2	82.5	14.1
u	170,156	.07 .005 62.1	23.8	15.6	184,171	.02 .04 198.1	45.1	2.6
s	185,197	.63 -.46 170.0	42.7	16.5	209,198	.07 .0007 79.6	2.0	18.9
t	221,210	.06 .03 205.2	27.1	24.7	222, 234	.19 -.68 268.4	42.3	14.6
	255,235	-.33 .45 90.3	142.5	7.14				

and robust algorithms to estimate the parameters of the map. The results are shown in Figure 37 and Table X. We search for the solution using $M = 2, 3, 4, \cdots$. The best result comes from $M = 17$ for the standard algorithm and $M = 15$ for the robust algorithm. The fit obtained from the robust algorithm is better than that from the standard algorithm since there are indices [102,118] from the standard algorithm between which the **SNR** is less than 10db.

In order to create a comparison with other techniques, we used autoregression (**AR**) models [41] to fit the above examples. An AR process with non-zero mean μ

Table XI. AutoRegression Model Parameters Estimation with Yule-Walker Equations for the Five Examples

Example	AR order	Mean μ	Variance σ^2	AR coefficients $(a_1, a_2, \cdots, a_p)$
1	14	153.37	2975.37	0.55 0.05 0.15 0.06 -0.06 -0.08 0.17 0.04 -0.0033 -0.098 0.049 -0.066 -0.16 0.26
2	16	185.87	7039.92	0.15 0.12 -0.032 0.02 -0.08 0.048 0.042 0.19 -0.038 -0.13 -0.11 0.24 -0.059 -0.017 -0.12
3	1	156.59	34.56	0.99
4	1	157.21	206.38	0.95
5	1	166.66	1332.32	0.85

Table XII. Signal-to-Noise Ratios from the Various Methods

Example	Standard **SNR**	Robust **SNR**	AR **SNR**
1	13.11	17.25	10.53
2	15.73	15.73	8.88
3	47.97	47.97	29.34
4	31.28	31.28	22.02
5	14.25	14.27	13.95

can be expressed in term of the recursive equation[98]:

$$x_t = a_1 x_{t-1} + a_2 x_{t-2} + \cdots + a_p x_{t-p} + \epsilon_t, \tag{5.16}$$

where ϵ_t is a white noise process with zero mean and finite variance σ^2 and the order of the AR process is p. Table XI shows the results where AR models are fitted to the examples, using the S Plus command, ar.yw, based on the Yule-Walker algorithm.

The **SNR** achieved by applying the inverse extended IFS interpolation standard and robust algorithms, and the AR model to the examples are listed in Table XII. The IFS algorithms achieve uniformly higher SNR than the AR model in examples 1, 2, 3, and 4. In the case of the audio signal there is little difference.

By applying the parallel distributed algorithm to the same numerical examples we obtained the same map parameters, but the running time obviously decreased. The numerical values are listed in Table XIII where the first column is the number of computers. The computers of the parallel algorithms used are SUN Sparc ELC, IPC, Sparc 2, and SUN 470. There are many factors to influence the running time, such as the number of processors in each computer, the CPU speed of each computer, etc. The results listed in Table XIII and Figure 38 are averages taken from several tests.

The time complexity of Algorithm 5.1 relative to N, the size of the data set, is not linear. We see that the speed-up ratio based on using two computers, relative to the case of a single computer, is approximately three. The time spent on communication increases quickly especially if the number of computers exceeds 3, so that the improvement in the time spent on calculating the self-affine region cannot cover the increase in communication time. The best distributed parallel computing proposal for these examples is to use two or three computers.

The algorithms have been applied to various examples of approximately self-affine signals selected on the basis of their self-affine parameters, i.e., the number of maps and the vertical contraction factors. The simulation results show that the robust algorithm is strong enough to converge to the true result when the standard method does not. Real data on male speech have also been used to test the algorithms, and the results show that the robust algorithm achieved better fidelity to data than did

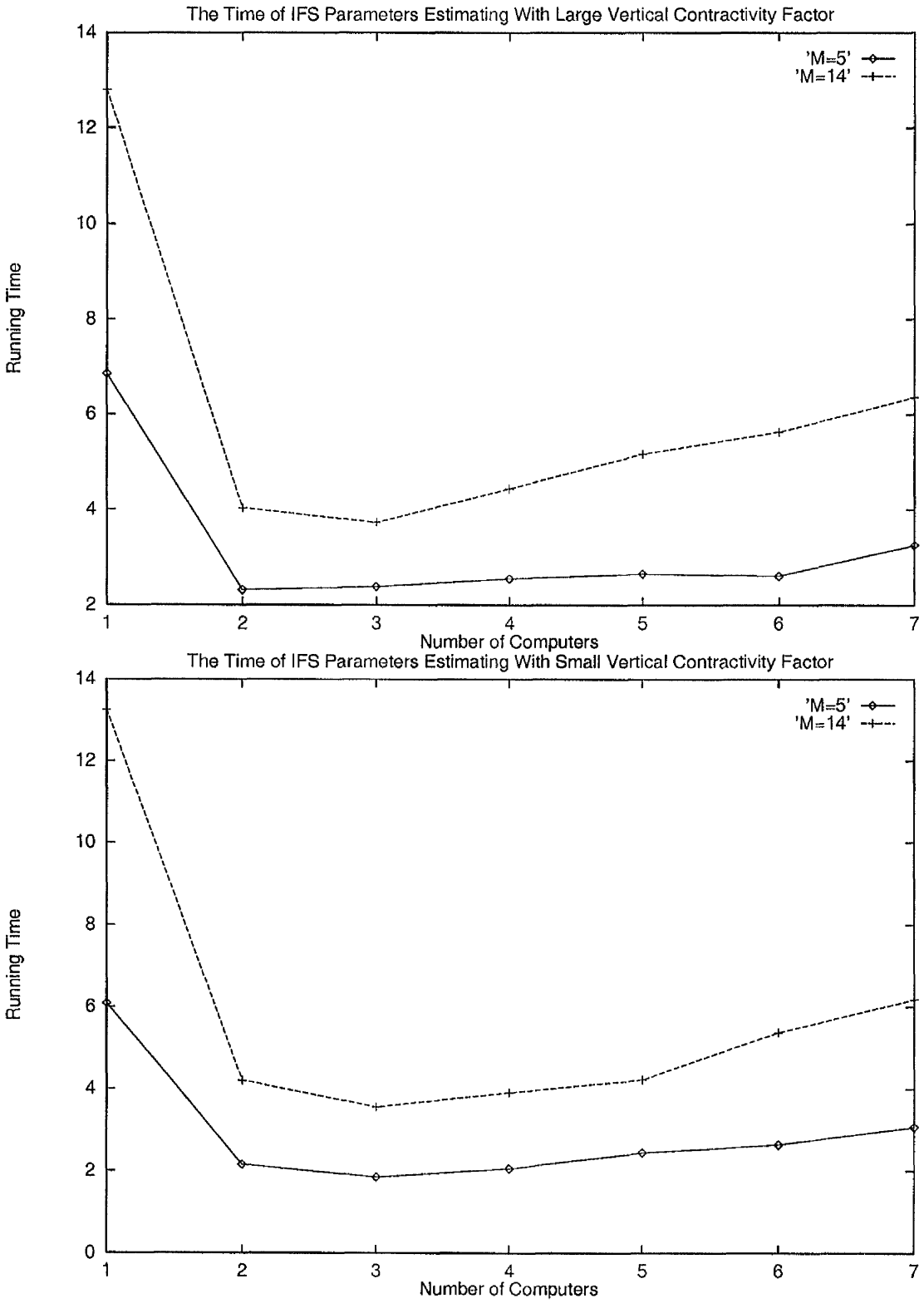

Fig. 38. Running Time for Estimating IFS Parameters for Approximately Self-affine Data (50% sample) with Large d_j (top diagram) and with Small d_j (bottom diagram).

Table XIII. Running Time (Seconds) for Estimating IFS Parameters

No.	Example 1 $M = 14, d_i$ large		Example 2 $M = 5, d_i$ large		Example 3 $M = 14, d_i$ small		example 4 $M = 5, d_i$ small	
	Compu.	Comm.	Compu.	Comm.	Compu.	Comm.	Compu.	Comm.
1	12.8	0	6.85	0	13.25	0	6.08	0
2	4.0	0.03	2.23	0.08	4.17	0.03	2.13	0.03
3	2.35	1.38	1.23	0.82	2.28	1.28	1.13	0.72
4	1.77	2.67	0.97	2.55	1.53	2.38	0.83	1.23
5	1.4	3.77	0.78	1.87	1.27	2.95	0.6	1.85
6	1.15	4.48	0.63	1.98	1.15	4.22	0.57	2.08
7	1.08	5.28	0.6	2.65	1.05	5.13	0.55	2.53

the standard algorithm.

In an empirical comparison, we have shown that the inverse extended IFS interpolation methods can achieve noticeably higher **SNR** than the popular AR model method in the case of approximately self-affine signals.

CHAPTER 6

ITERATED FUNCTION SYSTEM (IFS) SMOOTHING OF ONE-DIMENSIONAL DISCRETE SIGNALS BASED ON LOCAL CROSS-VALIDATION

6.1. Introduction

In this chapter, self-affine and approximately self-affine data corrupted by Gaussian noise are modelled with a robust IFS inverse algorithm and a local cross-validation technique. The local cross-validation is applied to compromise between smoothness and fidelity to the data. The parallel distributed version of the algorithm is implemented in Parallel Virtual Machine (PVM) with optimal task partition. Since the quantity of communication is small in this parallel algorithm a simplifying task partition model can be applied which is only concerned with each computer's speed. Several numerical simulation results show that the new IFS inverse algorithm achieves a higher signal to noise ratio than does autoregressive modelling. There is little machine idle time relative to total computing time in optimal task partitioning mode.

6.2. An Inverse IFS Algorithm Based on Local Cross-Validation

In Chapter 5 we explored the method for constructing an IFS model for a noise-free one-dimensional signal which is self-affine or approximately self-affine. If the input signal is corrupted by noise, the model we used in Chapter 5 will fail to achieve a good fit to the original signal. In order to solve this problem, some smoothing technique must be applied.

Recall the definition of IFS interpolation, that A is the graph of a continuous function $\hat{f} : [x_0,\ x_N] \rightarrow R$ which interpolates the data $\{(x_1, y_1), ..., (x_N, y_N)\}$. That is,

$$A = \{(x,\ \hat{f}(x)) :\ x \in [x_0,\ x_N]\}, \tag{6.1}$$

such that

$$\hat{f}(x_{i_j}) = f(x_{i_j}) = y_{i_j},\quad for\ j = 1, 2, ..., M. \tag{6.2}$$

Generally, however, we have the fractal interpolation model if the original signal is corrupted by noise:

$$\hat{f}(x_i) = f(x_i) + \epsilon_i,\quad for\ i = 1, 2, ..., n, \tag{6.3}$$

where ϵ_i are independent identically distributed errors.

In Section 5.2, we explained that Barnsley's linear fractal interpolating function $\hat{f}$ is a real-valued function of unknown parameter vectors R, defined in Equation (5.3), parameter vectors D which are the set of affine transform parameters $\{a_j, c_j, d_j, e_j, f_j\}$ defined in Equations (5.8) and (5.12), and the integer parameter M. Since d_j is the most important parameter among all the parameters of an affine transform, we shall only deal with the d_j in the following discussion. Therefore, $\hat{f}(x) = \hat{f}(R, D, M, x)$ is the output from the attractor of the IFS $\{R,\ w_0, ..., w_{M-1}\}$ based on the data $\{(x_1, y_1), \cdots, (x_N, y_N)\}$. The problem therefore becomes that of how to estimate the parameter vectors R, D and the integer parameter M. One possible approach is to minimize the residual sum of squares (RSS)

$$RSS(R, D, M) \ = \ \sum_{i=1}^{n} (y_i \ - \ \hat{f}(R, D, M, x_i))^2. \tag{6.4}$$

In Chapter 5 we tried to minimise a similar Hausdorff distance instead of the L^2 distance. However, $RSS(R, D, M)$ is a decreasing function of M, and M controls the degree of smoothness. The fewer affine transforms there are in an IFS, the higher is the degree of smoothness but the less is the fidelity to the data. The more affine transforms there are in an IFS, the lower is the smoothness but the better is the fidelity to the data. Thus, minimization of $RSS(R, D, m)$ will lead to over-fitting and is not the best approach. The problem is analogous to that of the identification of an auto-regression $AR(p)$ model, for which Akaike's AIC criterion [3], cross-validation [172] and other methods have been used as a means of penalizing the complexity of the fitted model.

The idea of $leave - one - out$ cross-validation is applied here. For $1 \leq i \leq n$, we define the *leave-one-out* data set by

$$S_{\backslash i} \ = \ \{(x_1, y_1), \cdots, (x_{i-1}, y_{i-1}), (x_{i+1}, y_{i+1}), \cdots, (x_N, y_N)\}. \tag{6.5}$$

Based on $S_{\backslash i}$, we compute parameter vectors R, D and the integer parameter M, and obtain an output, $\hat{f}_{\backslash i}$ say, from the attractor of the IFS $\{R^2,\ w_0, ..., w_{M-1}\}$ for $i = 1, 2, \cdots, N$. The cross-validation function is defined by

$$CV(R, D, M) \ = \ \sum_{i=1}^{n} (y_i \ - \ \hat{f}_{\backslash i}(R, D, M, x_i))^2. \tag{6.6}$$

For the data sets in Figure 39, $M = 2$, $R = \{0, i_1\}$, and $D = \{d_0, d_1\}$. Thus, if M is specified, $CV(i_1, d_0, d_1)$ is a real-valued function of three variables on $R \times D$. In Figures 40 and 41, we give plots of projections in the interpolation points subspace

R and the contraction factor subspace D.

We know that in general $CV(R, D, M)$ is a very high dimensional function and is not strictly convex, as shown by Figure 40. Any global search method in high dimensions is computationally highly demanding. However, the function $CV(R, D, M)$ has a special structure such that we can use a low-dimensional search algorithm to find a suboptimal solution. First, we note that M is known implicitly once R is determined. Secondly, R is an integer-valued vector such that each i_j in R satisfies $1 \leq i_j \leq N$, and there is an ordering among the elements of R, i.e. $i_1 < i_2 < \cdots < i_{M-1}$. Since each w_j contracts the data points $(x_1, y_1), \cdots, (x_N, y_N)$ into the region between the left-end interpolation point (x_{i_j}, y_{i_j}) and the right-end interpolation point $(x_{i_{j+1}}, y_{i_{j+1}})$, $\hat{f}_{\setminus i}$ is a function of the contraction factor d_j only, i.e., $\hat{f}_{\setminus i}(D, x) = \hat{f}_{\setminus i}(d_j, x)$ on $[x_{i_j}, x_{i_{j+1}}]$. Thus the cross-validation function $CV(R, D, M)$ can be expressed as a sum of local cross-validation functions:

$$CV(R, D, M) = \sum_{j=0}^{M-1} \sum_{i=i_j}^{i_{j+1}-1} (Y_i - \hat{f}_{\setminus i}(d_j, X_i))^2 = \sum_{j=0}^{M-1} CV_j(i_j, i_{j+1}, d_j), \quad (6.7)$$

where $i_0 = 1$ and $i_M = N$. Thus the minimum of $CV(R, D, M)$ is achieved if and only if each of $CV_j(i_j, i_{j+1}, d_j)$ achieves its minimum and given correct choices for i_j and i_{j+1}.

In order to enhance the robustness of the local cross-validation algorithm, we use a technique similar to that in Section 5.2 as shown in Figure 42, where P_{j-1}, P_j, P_{j_s}, P_{j+1}, $P_{j'_{s+1}}$ are interpolation points. For each new interpolation region R_{j+1}, we calculate the new best interpolation point P_{j+1} and use the next best point P_{j_s} of the last interpolation region R_{j_s} to calculate the new next best region $R_{j'_{s+1}}$. If

$$CV_{i_j} + CV_{i_{j+1}} > CV_{i_s} + CV_{i'_{s+1}}, \quad (6.8)$$

then discard the interpolation point indices i_j, i_{j+1}, replacing them by i_s, i'_{s+1}.

We propose the following algorithm in which we minimize the $CV_j(i_j, i_{j+1}, d_j)$ consecutively.

Algorithm 6.1 Robust Inverse IFS Interpolation Algorithm Based on Cross-Validation

INPUT: $(x_1, x_1), \cdots, (x_N, y_N)$ and W, which controls the minimal distance between two consecutive interpolation points along the x direction in the algorithm.
OUTPUT: P, M and D.

1. j=0.

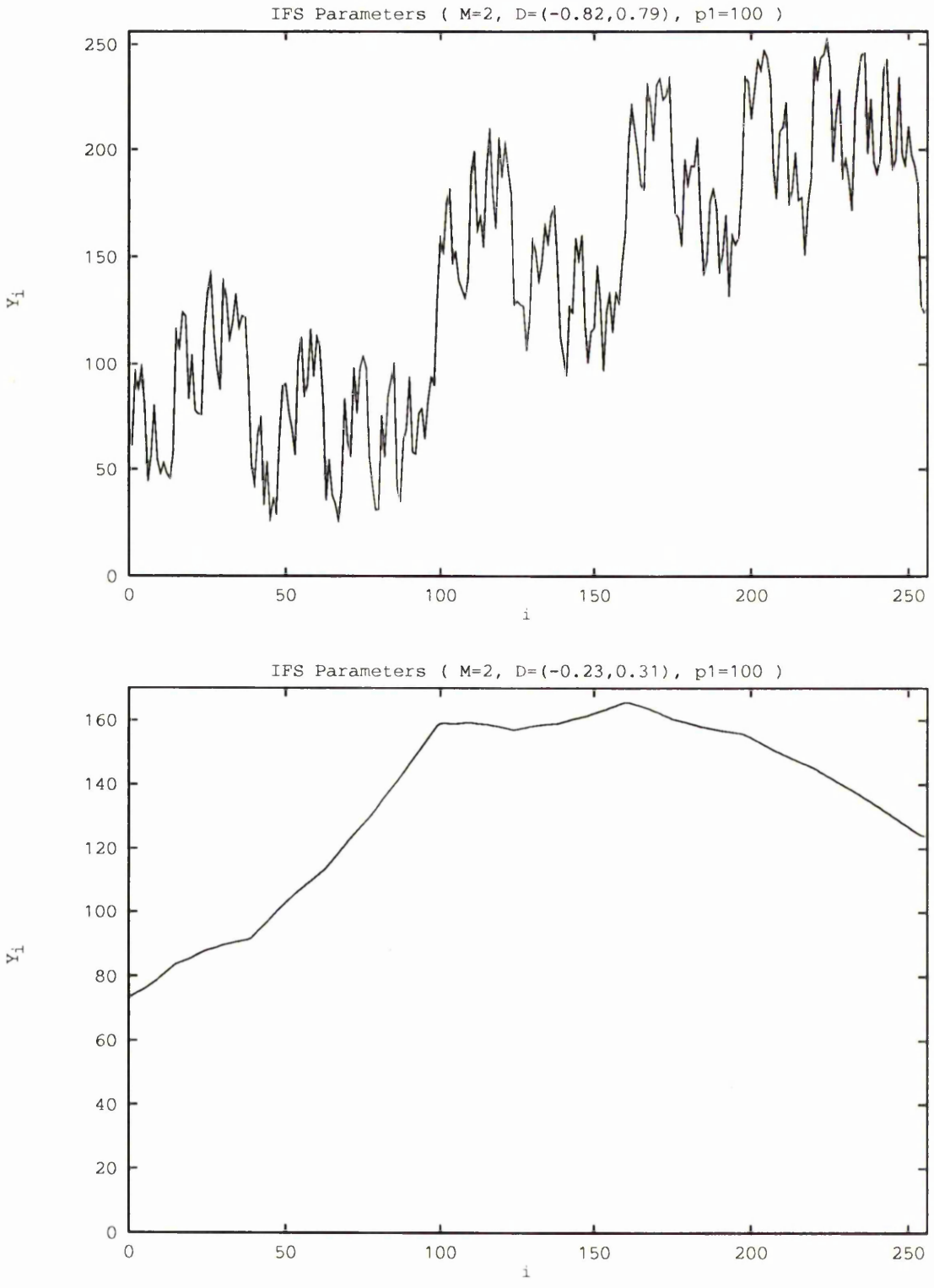

Fig. 39. Self-affine data generated by deterministic IFS. For the top picture, the contraction factors are $d_0 = -0.82$ and $d_1 = 0.79$. For the bottom picture, the contraction factors are $d_0 = -0.23$ and $d_1 = 0.31$.

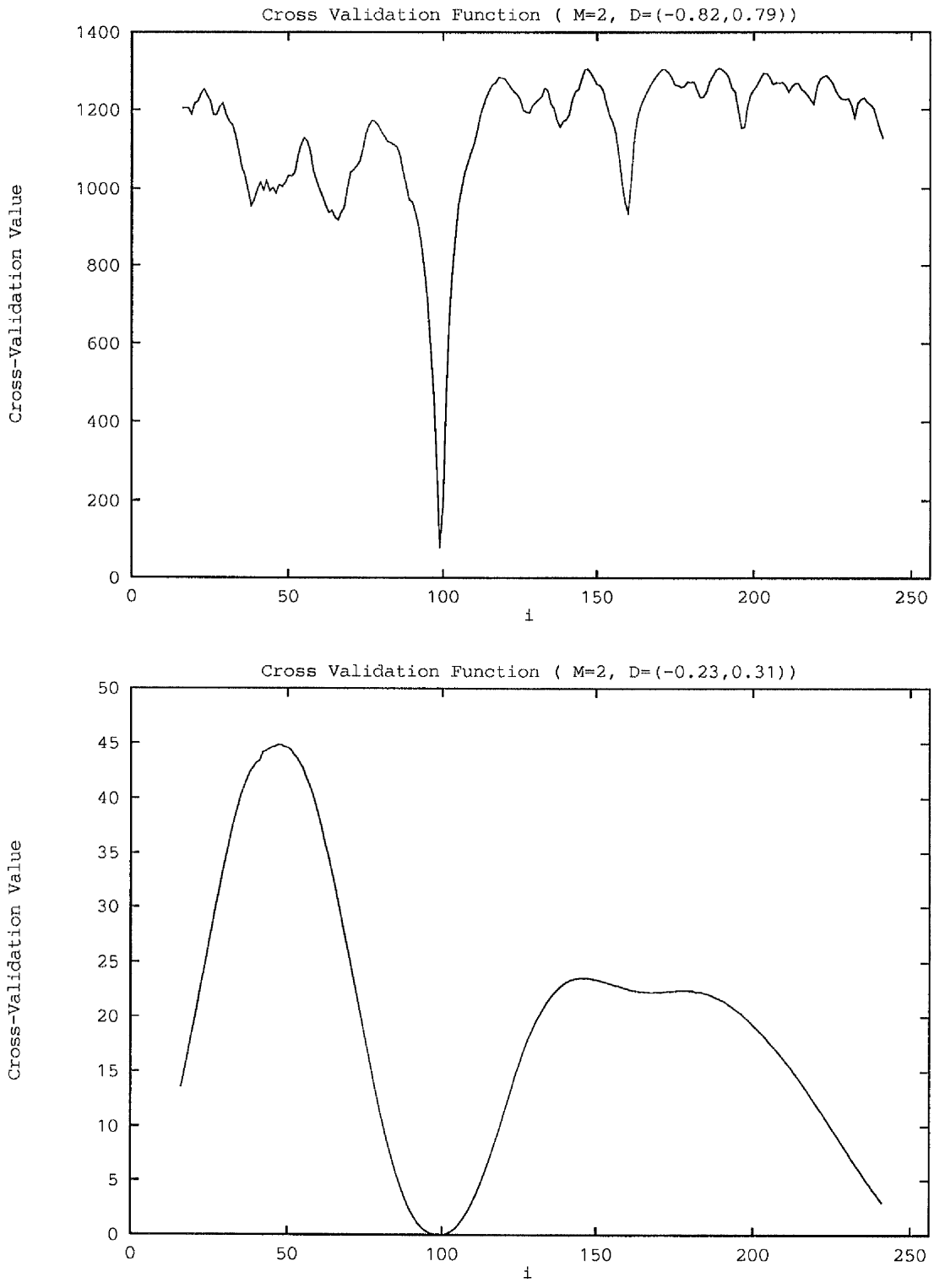

Fig. 40. Projection of the $CV(i_1, d_0, d_1)$ function on the interpolation point subspace R for fixed contraction factors D. In the top picture $D = (-0.82, 0.79)$ and in the bottom picture $D = (-0.23, 0.31)$.

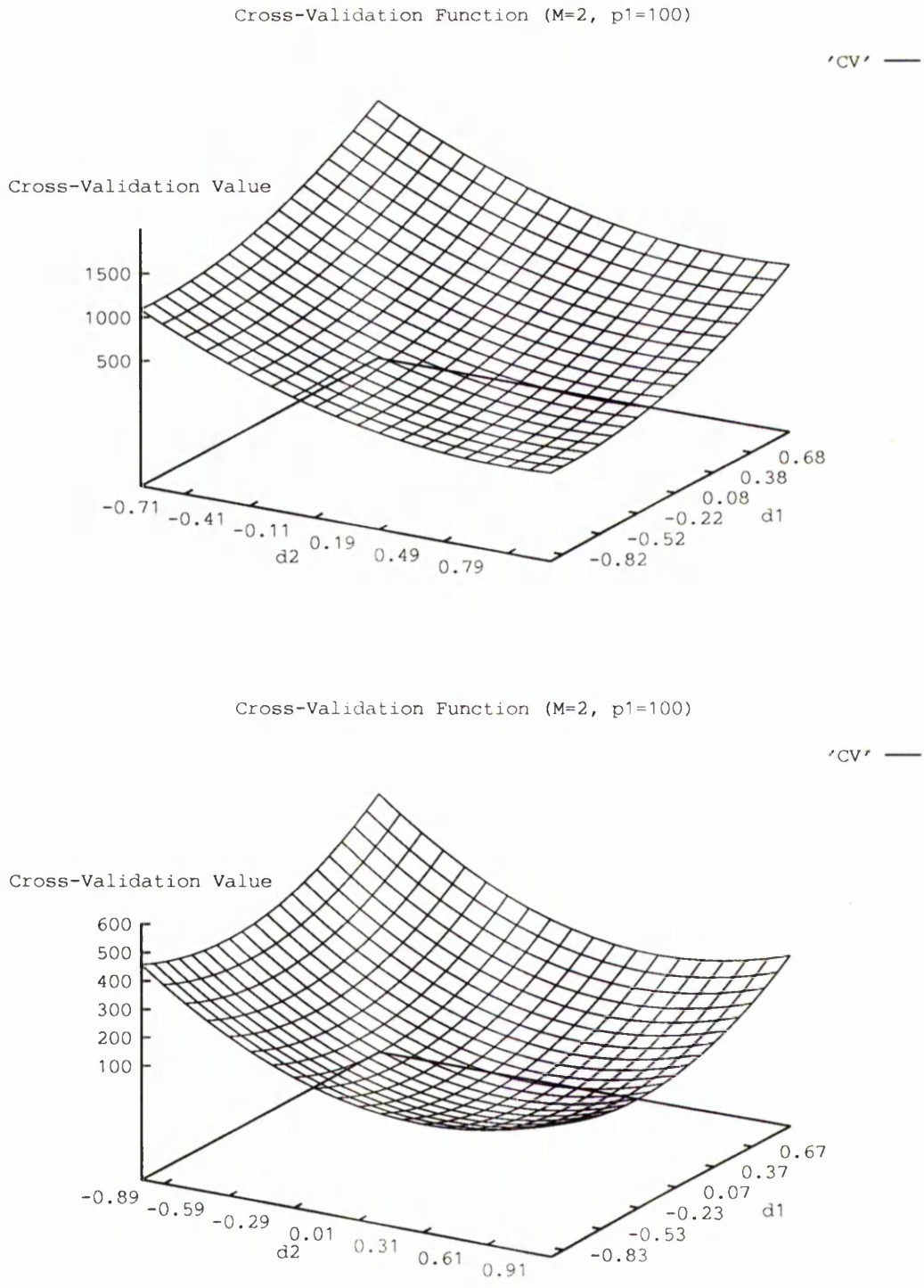

Fig. 41. Projection of the $CV(i_1, d_1, d_2)$ function on the contraction factor subspace D for fixed $R = \{0, 100\}$ in both pictures. For fixed $R = \{0, 100\}$, the minimum of CV appears at $(-0.82, 0.79)$ in the top picture and at $(-0.23, 0.31)$ in the bottom picture.

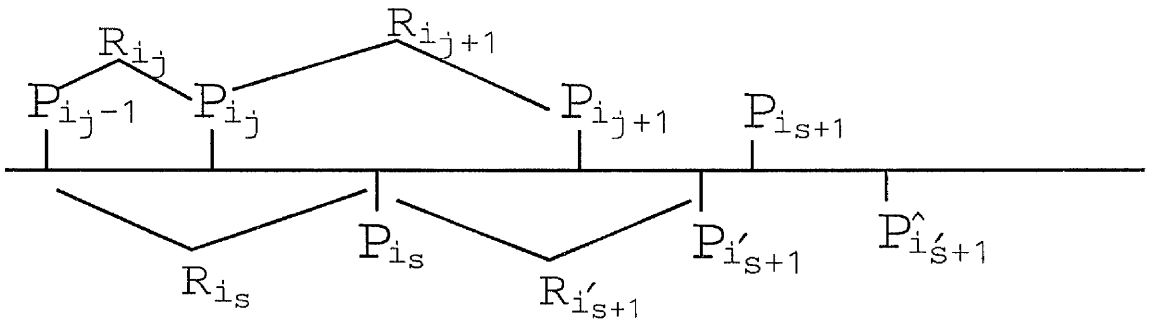

Fig. 42. Robustness modification of local cross-validation algorithm

2. j=j+1.

   2.1 Set a search interval $[s,\ e]$ for i_j, where integers s and e satisfy $x_s = x_1 + W$, $x_e = x_N - W$.

   2.2 For each element c in $[s, e]$, let (x_c, y_c) be the right-end interpolation point of the map w_j. The left-end interpolation point is $(x_{i_{j-1}+1}, y_{i_{j-1}+1})$ which has already been determined. Minimizing $CV_j(i_j, c, d_j)$ gives an estimate, $\hat{d}_j$, say, of d_j. Store $CV_j(i_j, c, \hat{d}_j)$ in a one-dimensional array $BUFFER[c]$.

   2.3 Choose, as a candidate of the index for the jth interpolation point from the $[s, e]$, the integer i_j such that $BUFFER[i_j]$ is minimal among the values in $BUFFER[c]$ for which $c \in [s, e]$. Then determine the next best i_s, which cross-validation value is next minimum.

3. If $i_{j-1} \neq 0$ and $i_{s-1} + W < e$ then

   3.1 Set new limits of the search interval $(i_{s-1}, e]$.

   3.2 In a similar way to (2.2) and (2.3), calculate indices i'_s, $\hat{i}'_s$ and corresponding cross-validation values $CV_{i_{s-1}}$, $CV_{i'_s}$.

   3.3 If $CV_{i_{j-1}} + CV_{i_j} > CV_{i_{s-1}} + CV_{i'_s}$ then set $i_{j-1} = i_{s-1}$ and $i_j = i_s$.

4. If $CV_j(i_j, e, \hat{d}_j) < CV_j(i_{j-1}, i_j, \hat{d}_j)$ then discard the candidate index i_j and exit from the algorithm.

5. Accept i_j as the jth interpolation index. Update the search limit to $s = i_j + W$.

6. If $e \leq s$ then exit from the algorithm.

7. Goto step 2.

8. Finally, when the algorithm stops, let $M = j + 1$.

There are two kinds of exit condition in the algorithm. One is at step 6 and occurs if no further interpolation point exists. The other is at step 4 and occurs if a further interpolation point will increase the error of fitting the given function.

6.3. Parallel Distributed Algorithm Based on Static Task Partition

In Chapter 5 we used Remote Procedure Call (RPC) library to implement our parallel distributed algorithms. RPC is a fundamental approach to interprocess communication based on the simple concept known as the procedure call. However, RPC does not provide machine configuration and process management functions which are necessary for an integrated Parallel Distributed Computing (PDC) environment. In chapter 5 we used some Unix system calls to implement these functions, but this implementation has not been optimized and it only applies to a special platform, SUN. For example, in Figure 34, we need to create multi-child processes. The overhead is high to maintain these processes.

Parallel Virtual Machine (PVM) is an integrated PDC environment, almost that of Unix machine and a dedicated Multi-processor machine can use it. It means that the algorithm you design for a special platform such as a SUN can also be used on any other platform which supports PVM.

The primary objective in PDC is that of faster execution by using multiple processing elements that work cooperatively on a single problem. There are several factors, ranging from inherent non-parallelism in the algorithm to the overheads of communication and synchronization among the multiple processors, which influence the efficiency in speeding up computations. In network-based environments, there are also external influences, since both the network and the processors may be in use by other applications in general.

In our situation, the quantity of communication is small, as we shall indicate in the following. Therefore, we can ignore the difference in communication overheads among the machines used for parallel computing and we only consider the computing speed of these machines.

As in Algorithm 5.2, the intensive computation requirement for Algorithm 6.1 comes from step (2.2) and step (3.2). We can partition the computation requirement of steps (2.2) and (3.2) into K sub-tasks if there are K computers which are available for us to use. The scale $scale_i$ of each sub-task is determined by the computing speed of the corresponding computer. We can get these speed parameters by running a benchmark program. In order to drive these sub-tasks, we need only the left interpolation point of the current interpolation region and the search interval $[s_i, e_i]$, if we have preloaded the parameter N and whole set of data into each sub-task. The

parameters of s_i and e_i can be determined by the speed parameters $scale_i$, using

$$s_i = e_{i-1} + 1 \tag{6.9}$$

$$e_i = s_i + (e - s)scale_i,\ i = 1, 2, \cdots, K, \tag{6.10}$$

where $e_0 = 0$ and $[s, e]$ is the current search interval of the sequential algorithm 6.1.

The parallel algorithm based on PVM and static task partitioning can be expressed as follows:

Algorithm 6.2 Master Part of Robust Inverse IFS Interpolation Parallel Algorithm Based on Cross-Validation, PVM and Static Task Partitioning.

INPUT: $(x_1, x_1), \cdots, (x_N, y_N)$ and W, which controls the minimal distance between two consecutive interpolation points along the X direction in the algorithm.
OUTPUT: P, M and D.

1. Register this process to PVM, *pvm_mytid*(); Create K PVM slave tasks, *pvm_spawn*(); Initialize the data structure, $j = 0$;
2. $j = j + 1$.
   - 2.1 Set a search interval $[s,\ e]$ for i_j, where integers s and e satisfy $x_s = x_1 + W$, $x_e = x_N - W$.
   - 2.2 Apply Equation (6.10) to calculate each search interval $[s_i, e_i]$; Pack this data, *pvm_pkint*(); Send them to each slave task, *pvm_send*();
   - 2.3 Collect from each, in return, the best and next best index interpolation point, *pvm_recv*(); Unpack them, *pvm_upkint*() and *pvm_upkfloat*(); Choose the best one as a candidate for the index for the jth interpolation point. Then determine the next best one, i_s.
3. If $i_{j-1} \neq 0$ and $i_{s-1} + W < e$ then
   - 3.1 Set new limits of the search interval $(i_{s-1}, e]$.
   - 3.2 On similar lines to steps (2.2) and (2.3), calculate indices i'_s, $\hat{i}'_s$ and the corresponding cross-validation values $CV_{i_{s-1}}$, $CV_{i'_s}$.
   - 3.3 If $CV_{i_{j-1}} + CV_{i_j} > CV_{i_{s-1}} + CV_{i'_s}$ then set $i_{j-1} = i_{s-1}$ and $i_j = i_s$.
4. If $CV_j(i_j, e, \hat{d}_j) < CV_j(i_{j-1}, i_j, \hat{d}_j)$ then discard the candidate index i_j and exit from the algorithm.
5. Accept i_j as the jth interpolation index. Update the search limit to $s = i_j + W$.

6. If $e \leq s$ then exit from the algorithm.
7. Goto step 2.
8. Finally, when the algorithm stops, set $M = j+1$; kill all slave tasks, *pvm_kill*(); quit from PVM, *pvm_exit*();

Algorithm 6.3 Slave Part of Robust Inverse IFS Interpolation Parallel Algorithm Based on Cross-Validation, PVM and Static Task Partitioning.

INPUT: $(x_1, x_1), \cdots, (x_N, y_N)$
OUTPUT: the best and next best indices and cross-validation value of interpolation points.

1. Register this process to PVM, *pvm_mytid*(); Initialize data structure;
2. Wait for receipt of the new index $i_{j-1}+1$ of the left interpolation point and the search interval $[s_i, e_i]$, *pvm_recv*();
3. Unpack this new data, *pvm_upint*();
4. For each element c in $[s_i, e_i]$, let (x_c, y_c) be the right-end interpolation point of the map w_j. The left-end interpolation point is $(x_{i_{j-1}+1}, y_{i_{j-1}+1})$, which has already been determined. Minimizing $CV_j(i_j, c, d_j)$ gives an estimate, $\hat{d}_j$ say, of d_j. Store $CV_j(i_j, c, \hat{d}_j)$ in a one-dimensional array $BUFFER[c]$.
5. Choose, as a candidate for the index for the jth interpolation point from the $[s_i, e_i]$, the integer i_j such that $BUFFER[i_j]$ is minimal among the values in $BUFFER[c]$ for which $c \in [s_i, e_i]$. Then determine the next best i_s, which gives the next smallest minimum of the cross-validation function.
6. Pack the best and next best indices and cross-validation values of the interpolation points, *pvm_pkint*(), *pvm_pkfloat*(); Send them to the master task, *pvm_send*();
7. Goto step 2.
8. Finally, when the slave is killed by the master, quit from PVM, *pvm_exit*().

6.4. Numerical Simulation

In this section there are two features of interest. First, we wish to test the efficiency of the inverse algorithm for the problem of identifying an IFS in terms of accurate estimates of the system's parameters P, D, and M. Secondly, since the IFS is concerned with the fractal interpolation problem, we want to see how the inverse algorithm compromises between the two contradictory aims of the degree of smoothness and

Table XIV. Original and calculated map parameters, local CV values, and Hausdorff distances for the strictly self-affine data with sample size 256

Original		Calculation		CV	H.
P	D	P	D	Value	Distance
0,39	0.87	0,39	0.87	0.19	0.62
40,73	-0.83	40,73	-0.83	0.25	0.6
74,115	-0.92	74,115	-0.92	0.27	0.57
116,177	0.85	116,177	0.85	0.3	0.7
178,255	0.91	178,255	0.91	0.26	0.73
0,39	0.16	0,39	0.16	0.19	0.59
40,73	-0.09	40,73	-0.09	0.26	0.48
74,115	-0.24	74,115	-0.24	0.27	0.5
116,177	0.13	116,177	0.13	0.27	0.6
178,255	0.22	178,255	0.22	0.23	0.55

fidelity to the data and also to compare with the fit of auto-regression models for smooth data to see the capacity for noise suppression.

In Example 6.1 and 6.2, for strictly self-affine data, we chose two data sets for which the map parameter Ds are different but the Ps are the same. One corresponds to large contraction factors whose absolute values are near to 1, whereas the other has small contraction factors whose absolute values are near to zero. To be specific $M = 5$, $D_1 = (0.87, -0.83, -0.92, 0.85, 0.91)$, $D_2 = (0.16, -0.09, -0.24, 0.13, 0.22)$ and $P = (0, 39, 73, 115, 177, 255)$. The results obtained from the inverse algorithm are reported in Table XIV. Algorithm 6.1 was used to search for solutions with $M = 2, 3, 4, 5, 6 \cdots$. The best solutions for both examples are found at $M = 5$ and the estimated parameters P and D coincide with the original P and D. In Figure 43 we plot the fractal interpolation on the basis of the estimated parameters of the IFS. The fidelity to the given strictly self-affine data is of course very good. The small local cross-validation values and Hausdorff distances in Table XIV are caused by computational error in the inverse algorithm since we use integer operations to replace floating point operations in order to reduce the computing time.

Next, in Examples 6.3 and 6.4 we again used the same strictly self-affine data as in Examples 6.1 and 6.2 used, respectively. The noise corrupted signal was generated by adding zero mean Gaussian noise of standard deviation $\sigma = 10.0$. The results are reported in Table XV and Figure 44. The optimal choice for the number of affine maps was the correct one of $M = 7$ for Example 6.3 and $M = 3$ for Example 6.4. The estimated P do not coincide with the originals and the estimated contraction factors in D also differed in both Examples 6.3 and 6.4. The new noise-corrupted input data are not self-affine, because of the Gaussian noise. However we can find from Figure

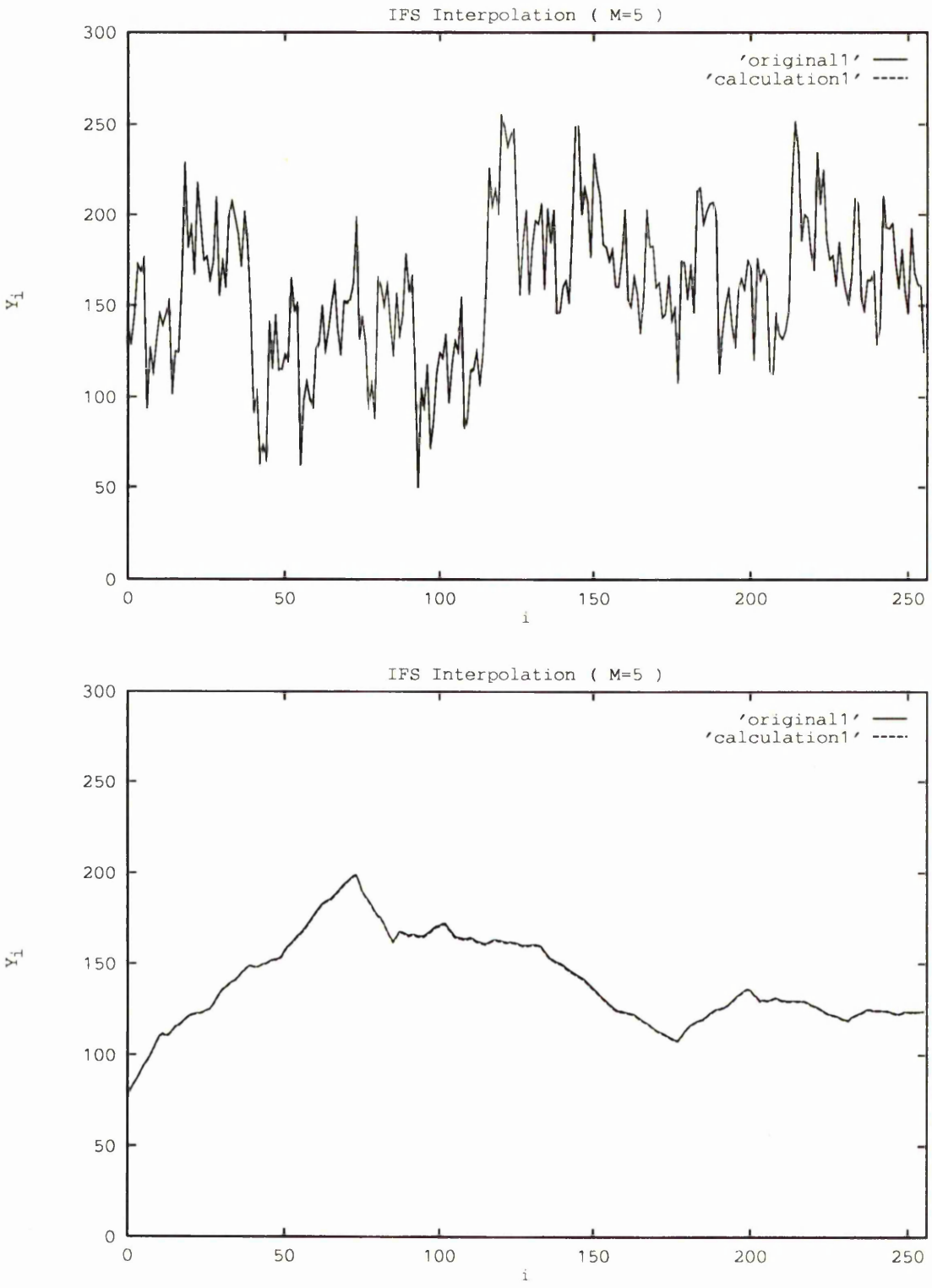

Fig. 43. Fractal interpolation ($M = 5$) for strictly self-affine data with large D (top picture) and small D (bottom picture).

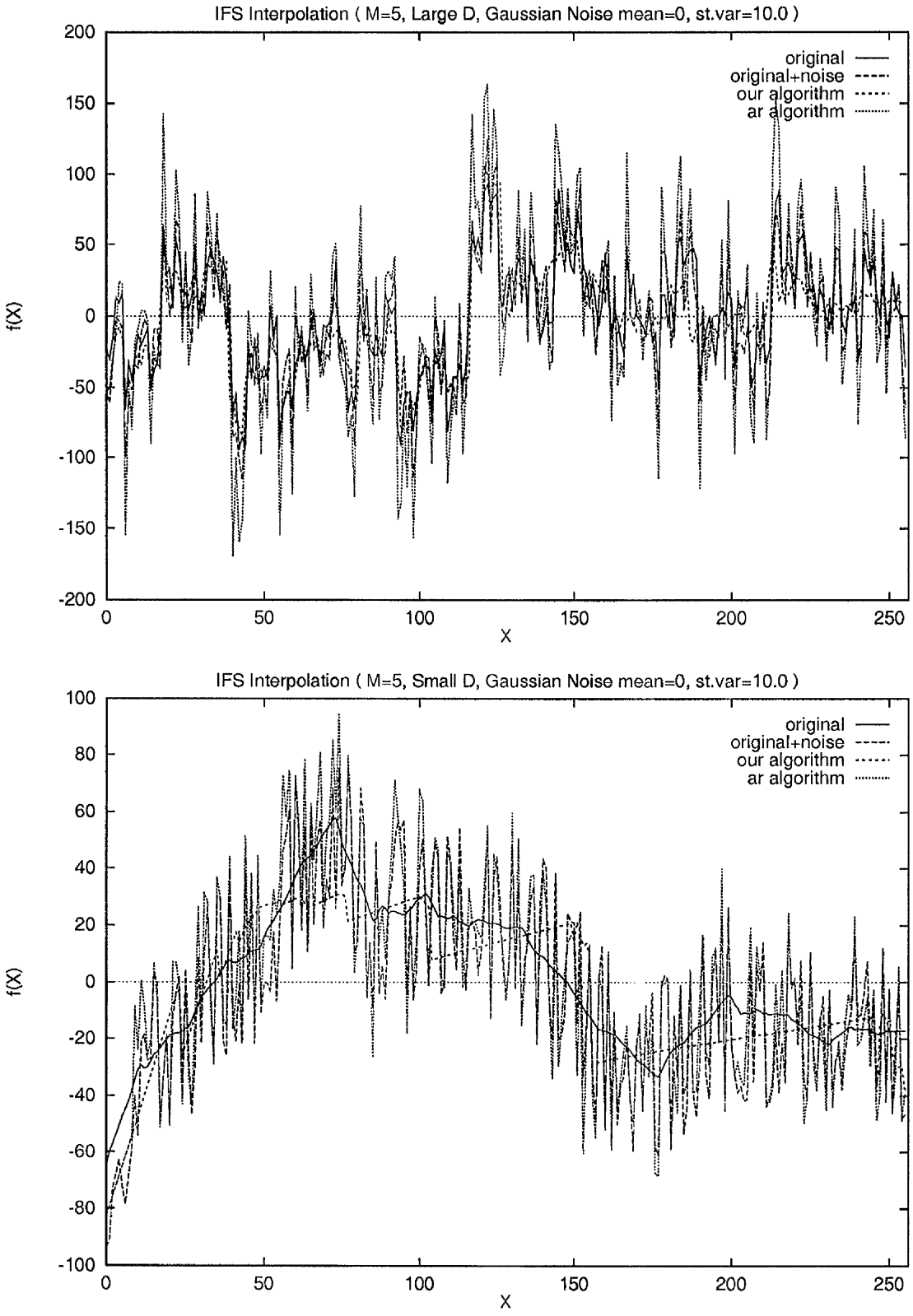

Fig. 44. Fractal interpolation for strictly self-affine data with a large D (top picture) and a small D (bottom picture) and additional Gaussian noise with zero mean and standard deviation $\sigma = 10.0$.

Table XV. Original and calculated map parameters, local CV values, and Hausdorff distances for the strictly self-affine data with Gaussian noise, mean=0, $\sigma = 10.0$

Original		Calculation		CV	H.
P	D	P	D	Value	Distance
0,39	0.87	0,38	0.67	14.08	31.62
40,73	-0.83	39,74	-0.55	19.99	30.13
74,115	-0.92	75,116	-0.64	22.52	40.55
116,177	0.85	117,126	0.99	16.08	39.09
178,255	0.91	127, 151	-0.26	21.62	55.47
		152,174	-0.38	23.39	58.21
		175, 255	0.4	25.56	67.4
0,39	0.16	0,23	-0.003	10.47	33.28
40,73	-0.09	24,155	0.46	20.84	39.35
74,115	-0.24	156, 245	0.006	18.7	46.33
116,177	0.13				
178,255	0.22				

44 that our algorithm gives a better compromise than the auto-regression model does between smoothness and fidelity to the data.

Finally in Examples 6.5 and 6.6 we applied our algorithm to process fractional Brownian motion data which are also corrupted by Gaussian noise with zero mean and standard deviation 10.0. A fractional Brownian motion, $V_H(t)$, is a single-valued function of one variable, t(usually time) and $H > 0$. Its increments $V_H(t_2) - V_H(t_1)$ have a Gaussian distribution. $V_H(t)$ exhibits a statistical scaling property in that, if the time scale t is changed by a factor r, then the increments $\Delta V_H(t)$ change by a factor r^H. We generated the fractional Brownian motion data by the spatial method with displaced interpolated points [153].

Noise-free fractional Brownian motion also provides approximately self-affine data. We chose two data sets of FBM on which to test the inverse algorithm. They had different parameters, ($H = 0.8$, $r = 0.2$) and ($H = 0.5$, $r = 0.4$). The simulation results are reported in Table XVI and Figure 45. The optimal choice for the number of affine maps was $M = 7$ for Example 6.5 and $M = 12$ for Example 6.6. We find from Figure 45 that our algorithm gives a better compromise than the auto-regression model does between the degree of smoothness and fidelity to the data.

All auto-regression models used in this section are described in Equation (5.16). Table XVII shows the results where AR models are fitted to examples 6.3, 6.4, 6.5 and 6.6, using the S Plus command *ar.yw*, based on the Yule-Walker algorithm.

By applying the parallel distributed algorithm 6.2 to all examples we obtained the same map parameters, but the running time obviously decreased. The computers used

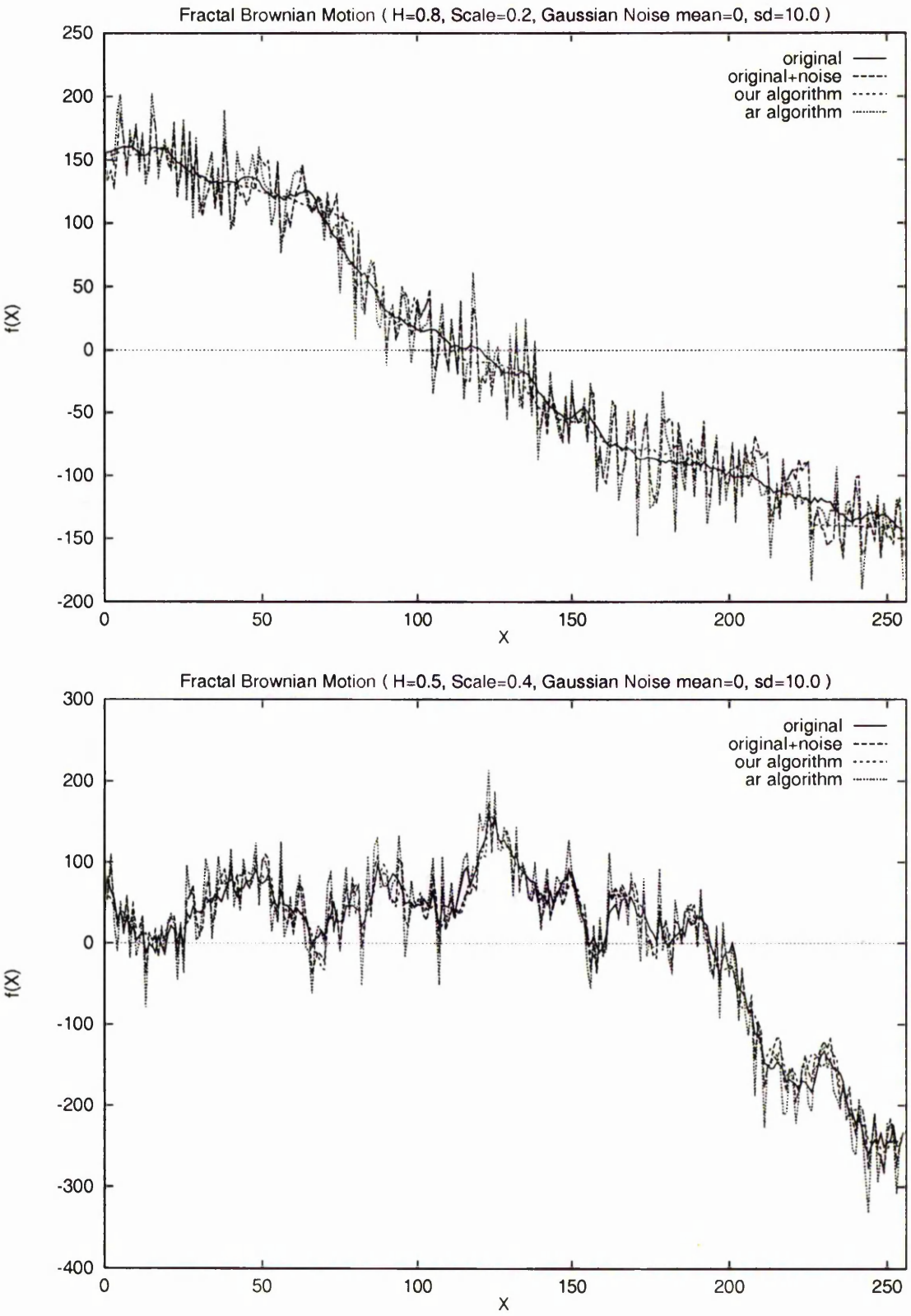

Fig. 45. Fractional Brownian Motions and Their IFS Interpolation Expressions. H=0.8, Scale=0.2 (top diagram) and H=0.5, Scale=0.4 (bottom diagram)

Table XVI. Calculated map parameters M, D, P, local CV values, and Hausdorff distances H for fractional Brownian motion corrupted by Gaussian noise with zero mean and standard deviation 10.0

Calculation		CV	H.
P	D	Value	Distance
0,21	0.08	9.04	28.82
22, 79	-0.09	15.89	37.06
80, 104	0.54	13.37	39.0
105, 202	0.41	17.55	40.21
203, 212	-0.14	10.68	13.79
213, 225	-0.23	8.05	8.31
225, 255	0.002	10.83	40.3
0,22	-0.16	10.01	40.14
23, 53	0.14	21.31	34.28
54, 70	0.17	21.12	43.46
71, 81	-0.09	19.44	40.88
82, 93	0.35	16.53	20.14
94, 122	-0.31	18.51	36.73
123, 151	-0.26	13.7	19.97
152, 161	-0.24	17.71	20.01
162, 177	0.23	11.78	31.46
178, 188	-0.2	17.25	31.46
189, 234	-0.4	18.97	37.99
235, 255	-0.17	10.98	33.53

Table XVII. Auto-Regression Model Parameters Estimation with Yule-Walker Equations for Examples

Ex.	AR order	Mean μ	Var. σ^2	AR coefficients $(a_1, a_2, \cdots, a_p)$	AR SNR	IFS SNR
6.3	4	-2.89	1192.01	0.45 0.002, 0.14, 0.15	1.0	4.64
6.4	9	-2.89	610.48	0.19 0.18 0.17 0.29 -0.006 -0.16 0.06 -0.0002 0.16	0.63	7.13
6.5	3	-0.43	765.16	0.53 0.24 0.23	12.86	19.64
6.6	2	0.69	1112.41	0.68 0.28	10.51	16.45

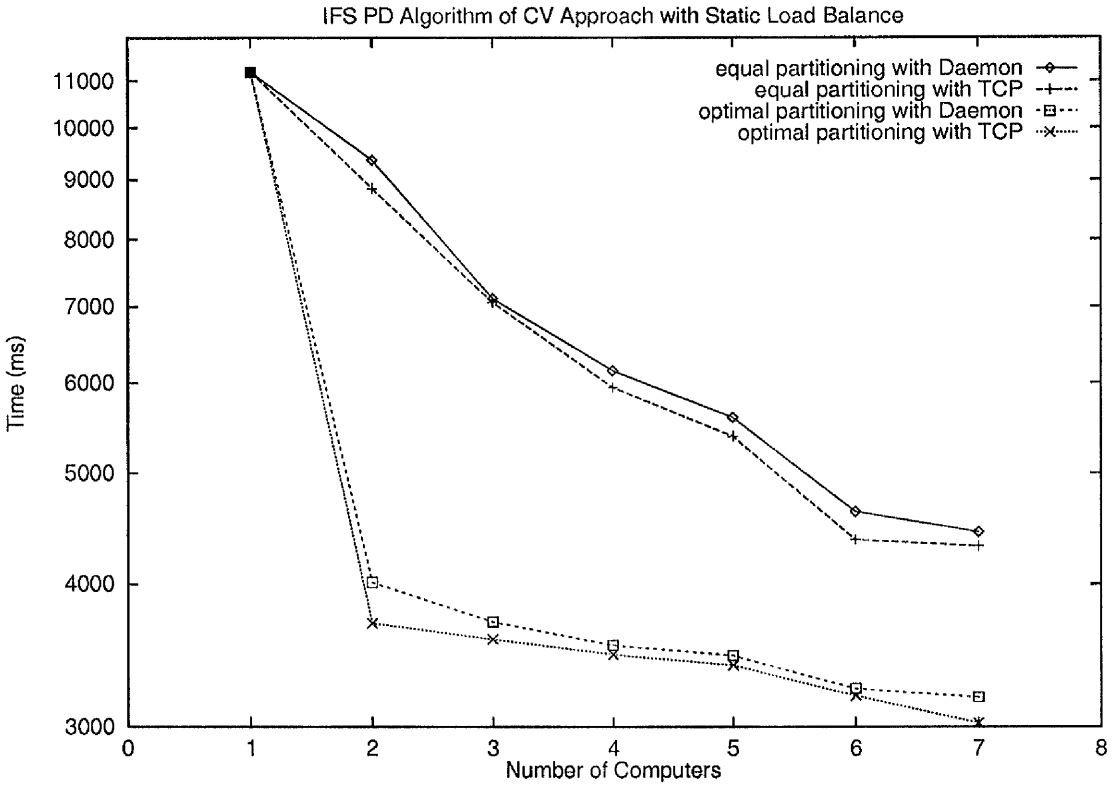

Fig. 46. Total time for Example 6.3 using PVM

in the parallel algorithms are SUN Sparc ELC, IPC, Sparc 10 and SUN 470. Figure 46 shows the total time (computing + communicating + idle) for Example 6.1. There are four curves in the Figure. Two of them use a Daemon-based communication scheme and others use a TCP-based communication scheme. In the PVM environment, the TCP-based mode provides a more efficient communication path than the Daemon mode so that we can obtain improvement in total time. Comparing Figure 38 and Figure 46 we can conclude that PVM is better than RPC for parallel distributed computing applications, since both algorithms have similar structure but the RPC approach fails to improve when the number of computers reaches four, while the PVM approach continues improving until the number of computers reaches seven.

Task partitioning is a very important issue in parallel distributed computing. We shows this by providing results for optimal task partitioning and equal task partitioning in Table XVIII. We note one second to five seconds improvement in total time in Table XVIII.

More detailed comparison is shown in Figure 47 and Table XIX. The height of each box in Figure 47 indicates the scale of each sub-task. In the case of equal task partitioning, the fastest computer incurs high idle time while awaiting the new

Table XVIII. Total times (milli-seconds) for Example 6.3 using PVM Daemon and TCP communication with equal and optimal task partitioning

No	Equal (Daemon)	Optimal (Daemon)	Equal (TCP)	Optimal (TCP)
1	11200	11200	11200	11200
2	9364	4015	8848	3699
3	7119	3710	7063	3584
4	6154	3541	5945	3477
5	5594	3470	5386	3402
6	4626	3243	4372	3197
7	4441	3187	4317	3023

Table XIX. Task Partitioning and Load Balance for Example 6.3 with PVM TCP Communication Mode and Seven Computers

Computer Name	Scale of Sub-task	Computing Time	Comm. Time	Idle Time
1	0.15	1140	30	2720
2	0.14	3652	16	174
3	0.14	734	18	3148
4	0.14	2719	17	1162
5	0.14	3153	13	675
6	0.14	2620	18	1178
7	0.14	2113	21	1703
1	0.112	1780	20	850
2	0.091	2062	15	311
3	0.505	1818	21	884
4	0.075	1562	16	878
5	0.065	2077	16	403
6	0.072	2076	12	466
7	0.08	2119	16	462

message. This is the case, for example, with No. 3 computer in Table XIX, and it wastes computing resource. However, in optimal task partitioning, all computers have low idle time and keep busy in computing, as we expect.

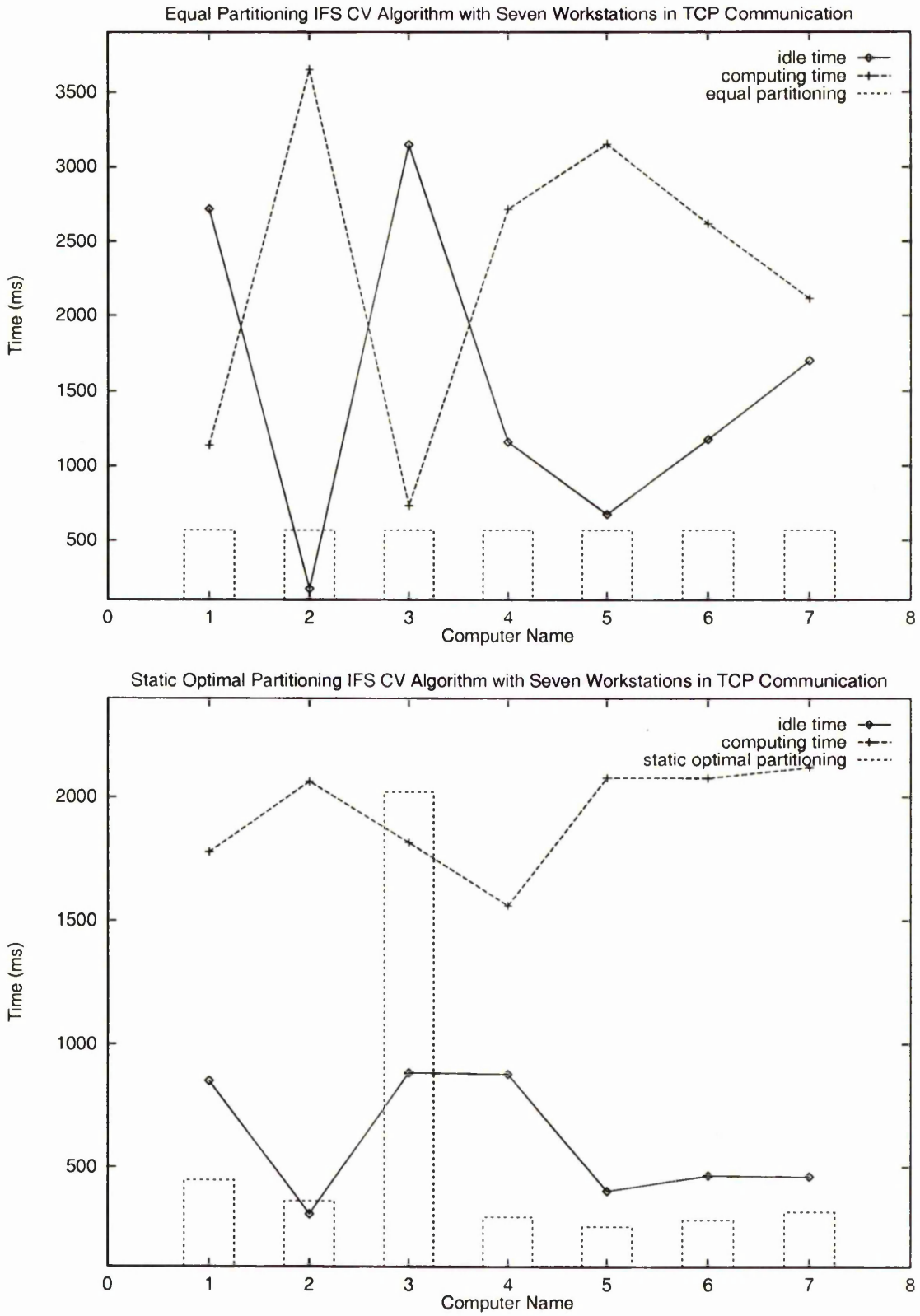

Fig. 47. Task Partitioning and Load Balance for Example 6.3 with PVM TCP Communication Mode and Seven Computers, Equal Partitioning (top diagram) and Optimal Partitioning (bottom diagram)

CHAPTER 7

USING INVERSE LOCAL ITERATED FUNCTION SYSTEMS (IFS) TO MODEL ONE DIMENSIONAL DISCRETE SIGNALS

7.1. Introduction

Local IFS realise the IFS limit if data are self-affine and are suitable for modelling non self-affine signals. However it is difficult to explore the whole parameter space to achieve globally optimal parameter estimation. We present a two-stage search scheme to estimate the parameters of local IFS in this chapter so that we can get a suboptimal solution in a reasonable time. In network-based parallel computing, most performance degradation involve load imbalance caused by the difference of machines capability and external load. We apply a dynamic load balance technique to overcome the problem. Some numerical simulation indicates that our inverse local IFS algorithm works well for serval types one-dimensional signal and the parallel version, with dynamic load balance, can automatically have each machine busy with computing and with low idle time.

7.2. Inverse Local IFS Theory and Algorithm

As we have shown in the last two chapters, IFS interpolation is a viable method for modelling a given one-dimensional signal if it is a self-affine or approximately self-affine discrete sequence. Most signals, however, are not approximately self-affine. A sinusoid, for example, is neither self-affine nor approximately self-affine. A local IFS approach may be appropriate for modelling general signals.

A general LIFS can be defined as follows:

Definition 7.1 *Let (X, d) be a compact metric space. Let $w_i : R_i \to X$ be a local contraction mapping on (X, d), with contractibility factor s_i, for $i = 1, 2, \cdots, M$, where M is a finite positive integer. Then $\{w_i : R_i \to X : i = 1, 2, \cdots, M\}$ is called a local iterated function system. The number $s = \max\{s_i : i = 1, 2, \cdots, M\}$ is called the contractibility factor of the LIFS.*

A one-dimensional LIFS interpolation can be defined as :

Definition 7.2 *A one-dimensional signal, $\{(x_i, y_i) : i = 0, 1, \cdots, N;\ x_i < x_{i+1},\ |x_i - x_j| \leq N, \forall i, j, y_i \in \mathbf{R}^1\}$ is divided into M regions R_j by contractive maps w_j.*

$$R_j = \{[i_{j-1} + 1, i_j]\}, j = 2, 3, \cdots, M - 1,$$

$$\begin{aligned} R_0 &= \{[0, i_1]\}, \\ R_M &= \{[i_{M-1}+1, N]\}, \end{aligned} \tag{7.1}$$

where $\{(x_{i_{j-1}+1}, y_{i_{j-1}+1}), (x_{i_j}, y_{i_j})\}$ *are terminal points, also known as interpolation points. Each region is self-affine for an associated region,* $\underline{R}_j$,

$$\begin{aligned} \underline{R}_j &= \{[\underline{i}_{jl}, \underline{i}_{jr}]\}, j = 2, 3, \cdots, M-1, \\ \underline{R}_0 &= \{[\underline{i}_{0l}, \underline{i}_{0r}]\}, \\ \underline{R}_M &= \{\underline{i}_{Ml}, \underline{i}_{Mr}]\}, \end{aligned} \tag{7.2}$$

where $\{(x_{\underline{i}_{jl}}, y_{\underline{i}_{jl}}), (x_{\underline{i}_{jr}}, y_{\underline{i}_{jr}})\}$ *are terminal points of the associated region* $\underline{R}_j$. *The affine map* w_j *is the same as in Equation (5.1).*

Among the affine map parameters, d_j must satisfy $|d_j| < 1$ so that it guarantees that w_j is a contraction map. The parameter a_j can be located in $(-1, 1)$. If $a_j > 0$ it means that, for the region $(i_{j-1},\ i_j]$, and associated region $[\underline{i}_{jl},\ \underline{i}_{jr}]$, w_j maps $(x_{\underline{i}_{jl}}, y_{\underline{i}_{jl}})$ to $(x_{i_{j-1}+1},\ y_{i_{j-1}+1})$ and $(x_{\underline{i}_{jr}}, y_{\underline{i}_{jr}})$ to $(x_{i_j},\ y_{i_j})$. If $a_j < 0$, it means that, for the region $(i_{j-1},\ i_j]$, and associated region $[\underline{i}_{jl},\ \underline{i}_{jr}]$, w_j maps $(x_{\underline{i}_{jl}}, y_{\underline{i}_{jl}})$ to $(x_{i_j},\ y_{i_j})$ and $(x_{\underline{i}_{jr}},\ y_{\underline{i}_{jr}})$ to $(x_{i_{j-1}+1},\ y_{i_{j-1}+1})$.

Comparing this with the definition of an IFS in Chapter 5, we can find that the difference between an IFS and a local IFS is the associated region $\underline{R}_j$. We have only one associated region $[0, N]$ in IFS, but, we have M associated regions in local IFS. We can get new affine map parameters estimation formulae by modifying the corresponding equations.

For map parameters a_j, e_j, we have

$$\begin{aligned} a_j &= \frac{x_{i_j} - x_{i_{j-1}+1}}{x_{\underline{i}_{jr}} - x_{\underline{i}_{jl}}} \\ e_j &= \frac{x_{\underline{i}_{jr}} x_{i_j} - x_{\underline{i}_{jl}} x_{i_{j-1}+1}}{x_{\underline{i}_{jr}} - x_{\underline{i}_{jl}}}, \end{aligned} \tag{7.3}$$

if w_j maps $[\underline{i}_{jl},\ \underline{i}_{jr}] \Rightarrow (i_{j-1}, i_j]$. The map parameters c_j, d_j, f_j can be obtained from Equation (5.12). In other cases w_j maps $[\underline{i}_{jr},\ \underline{i}_{jl}] \Rightarrow (i_{j-1},\ i_j]$, and we need only interchange i_j and i_{j-1} in Equation (7.3).

The inverse local IFS can be defined as the following optimal problem:

$$\min h(L, \sum_{j=1}^{M} w_j(\underline{L}_j)), \tag{7.4}$$

where L is the input signal and $\underline{L}_j$ is the input signal of the associated region $\underline{R}_j$.

The corresponding sub-optimal problem is given:

$$\sum_{j=1}^{M} \min h(L_j, w_j(\underline{L}_j)), \tag{7.5}$$

where L_j is the input signal of the self-affine region R_j and the unknown parameters are the right interpolation point index i_j of self-affine region R_j, associated region $\underline{R}_j$ indices $[\underline{i}_{jl}, \underline{i}_{jr}]$ and the map w_j parameters $(a_j, c_j, d_j, e_j f_j)$. We need to search for all of these unknown parameters within this search space. Even for this sub-optimal problem, the search space is still too large to explore, where $i_j \in [0, N]$, $\underline{i}_{jl} \in [0, N]$ and $\underline{i}_{jr} \in [0, N]$, limited by the condition $x_{\underline{i}_{jr}} - x_{\underline{i}_{jl}} > x_{i_j} - x_{i_{j-1}}$. We need further to simplify the sub-optimal problem in order that we can solve it in a reasonable time.

We suggest a two-stage search scheme :

First we suppose that the associated region length is twice the length of the self-affine region, that is,

$$x_{\underline{i}_{jr}} - x_{\underline{i}_{jl}} = 2 \times (x_{i_j} - x_{i_{j-1}}). \tag{7.6}$$

We search for estimation of the parameters i_j, $\underline{i}_{jl}, \underline{i}_{jr}$ in this sub-space.

Second We receive all self-affine regions R_j, $j = 1, \cdots, M$. Then, for each self-affine R_j, we search the corresponding associated region in the full search space, that is,

$$x_{\underline{i}_{jr}} - x_{\underline{i}_{jl}} > x_{i_j} - x_{i_{j-1}},\ x_{\underline{i}_{jr}},\ x_{\underline{i}_{jl}} \in [0, N]. \tag{7.7}$$

We also use information about the neighbouring self-affine region to enhance the robustness of the inverse local IFS algorithm, as we have done in Chapter 5. Actually for each self-affine region we need to calculate the next best candidate interpolation point index i_s to accompany with i_j. Then when we search for the next self-affine region, we have to calculate two possible self-affine regions, one, $R_{i_{j+1}}$, based on i_j and other, $R_{i'_{s+1}}$, based on i_s. If

$$H(R_{i_j}) + H(R_{i_{j+1}}) > H(R_{i_s}) + H(R_{i'_{s+1}}), \tag{7.8}$$

then we discard the interpolation point indices i_j, i_{j+1}, replacing them by i_s, i'_{s+1}.

The inverse local IFS algorithm consists of three algorithms. The first is the core of sub-space search (7.6).

Algorithm 7.1. Inverse Local IFS Interpolation Algorithm (Estimation of Map Parameters and Hausdorff Distance for One Self-affine Region).

INPUT: $(x_0, y_0), \cdots, (x_N, y_N)$, self-affine region indices $(i_{j-1},\ i_j)$ and associated region width *Awid*.

OUTPUT: the best and next best associated regions $(\underline{i}_{jl}, \underline{i}_{jr})$, $(\underline{i}_{sl}, \underline{i}_{sr})$ and Hausdorff distances $H(R_{i_j})$, $H(R_{i_s})$ and w_j parameters.

1. Initialize $H(R_{i_{jl}})$ and $H(R_{i_{sl}})$
2. For each $\underline{i}_{jr} \in [Awid, N]$ do steps 3 – 5.
3. Get $\underline{i}_{jl} = \underline{i}_{jr} - Awid$; calculate the map w_j parameters $(a_j, c_j, d_j, e_j, f_j)$ for self-affine region (i_{j-1}, i_j) with associated region $(\underline{i}_{jl}, \underline{i}_{jr})$ and the Hausdorff distance $H_{(\underline{i}_{jr})}$.
4. If $H_{(\underline{i}_{jr})} < H(R_{i_{jl}})$, then record the i_{jr} as the new best one and update $H(R_{i_{jl}}) = H_{(\underline{i}_{jr})}$.
5. Otherwise, if $H_{(\underline{i}_{jr})} < H(R_{i_{sl}})$, then record the i_{jr} as the new next best one and update $H(R_{i_{sl}}) = H_{(\underline{i}_{jr})}$.
6. Finally, output the best and next best associated regions, defined by $(\underline{i}_{jl}, \underline{i}_{jr})$, $(\underline{i}_{sl}, \underline{i}_{sr})$ and Hausdorff distances $H(R_{i_j})$, $H(R_{i_s})$ and w_j parameters.

For stage one of the search, we describe the algorithm as follows:

Algorithm 7.2. Inverse local IFS Interpolation Algorithm (Stage One Search for Self-affine Regions).

INPUT: $(x_0, y_0), \cdots, (x_N, y_N)$ and W.
OUTPUT: the number M and Self-affine Regions R_j, $j = 1, 2, \cdots, M$.

1. Initialize interpolation point indices $i_0 = 0$, $i_M = N$ and $j = 0$.
2. $j = j + 1$.

   2.1 Set a search interval $[s,\ e]$ for i_j, where integers s and e satisfy $x_s = x_{i_{j-1}} + W$, $x_e = x_N - W$.

   2.2 For each element c in $(s, e]$, do steps 2.3 – 2.6

   2.3 Let (x_c, y_c) be the right-end interpolation point of the map w_j. The left-end interpolation point is $(x_{i_{j-1}+1}, y_{i_{j-1}+1})$, which has already been determined.

   2.4 Apply Algorithm 7.1 to get the best and next best candidate associated regions, defined by $(\underline{i}_{jl}, \underline{i}_{jr})$, $(\underline{i}_{sl}, \underline{i}_{sr})$, and the Hausdorff distances $H(R_{i_j})$, $H(R_{i_s})$.

   2.5 If $H_{(\underline{i}_{jl})} < H(R_{i_j})$, then record i_j as the new best one and update $H(R_{i_j}) = H_{(\underline{i}_{jl})}$.

   2.6 Otherwise, if $H_{(\underline{i}_{sl})} < H(R_{i_s})$, then record i_s as the new best one and updating $H(R_{i_s}) = H_{(\underline{i}_{sl})}$.

3. If $i_{j-1} \neq 0$ and $i_{s-1} + W < e$ then

   3.1 Set new limits of the search interval $(i_{s-1}, e]$.

   3.2 As in step (2.2), calculate the best and next best indices i'_s, i'_{s+1} and the Hausdorff distances $H(R_{i_{s-1}})$, $H(R_{i'_s})$.

   3.3 If $H(R_{i_{j-1}}) + H(R_{i_j}) > H(R_{i_{s-1}}) + H(R_{i'_s})$ then set $i_{j-1} = i_{s-1}$ and $i_j = i_s$.

4. Accept i_j as the *j-th* interpolation index. Update the search limit to $s = i_j + W$.

5. If $e \leq s$ then exit from the algorithm.

6. Goto step 2.

7. Finally, output $M = j + 1$ and all self-affine regions R_j, $j = 1, \cdots, M$.

For stage two of the search, the algorithm is as follows:

Algorithm 7.3. Inverse local IFS Interpolation Algorithm (Stage Two Search for Associated Region).

INPUT: $(x_0, y_0), \cdots, (x_N, y_N)$ and M and all self-affine regions R_j, $j = 1, \cdots, M$.
OUTPUT: Associated Region $\underline{R}_j$, $j = 1, 2, \cdots, M$.

1. For each self-affine region R_j, $j = 1, 2, \cdots, M$, do step 2 – 6.

2. Set up a search interval $[s, e]$ as $(i_{j-1}, i_j]$.

3. For each new $Awid \in [1.5 \times (x_{i_j} - x_{i_{j-1}}), N]$, do steps 4 - 6.

4. Apply Algorithm 7.1 to get the best and next best candidate associated regions, $(\underline{i}_{jl}, \underline{i}_{jr})$, $(\underline{i}_{sl}, \underline{i}_{sr})$, and the Hausdorff distances $H(R_{i_j})$ and $H(R_{i_s})$.

5. If $H_{(\underline{i}_{jl})} < H(R_{i_j})$, then recording the i_j as new best one and updating $H(R_{i_j}) = H_{(\underline{i}_{jl})}$.

6. Finally, output the associated region $(\underline{i}_{jl}, \underline{i}_{lr})$ which corresponds to $H(R_{i_j})$.

7.3. Parallel Distributed Inverse Local IFS Algorithm Based on PVM and Dynamic Load Balance

In Chapter 6 we explored the parallel distributed algorithm with static optimal task partition. This static load balance model supposes that the work-stations have no any external job appearing in the task executing period. However, in a real environment, there are external influences, since, in general, both the network and the processors may be in use by other applications. In a network-based computing environment,

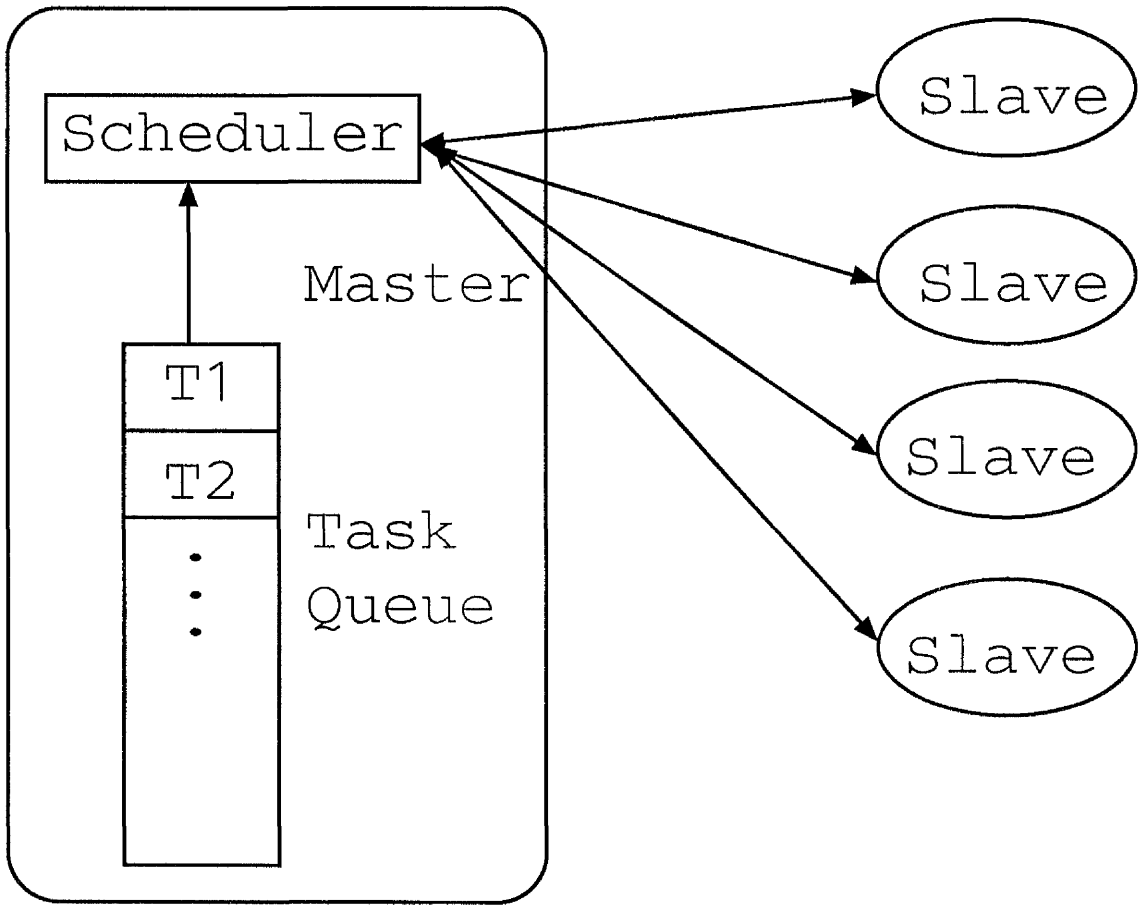

Fig. 48. Schematic for Dynamic Load Balance Application.

load imbalance, caused by disparities in machine capabilities as well as by external loads, emerges as a primary cause of lowered overall performance. A good parallel distributed algorithm should combat this imbalance.

In order to match the dynamic varied computing resource, we need to apply a dynamic task load scheduler. The dynamic load scheme is illustrated in Figure 48. The scheme requires that the whole task can be partitioned into completely independent and the same portions, a slave algorithm is applied to each, and partial results are combined using simple combination schemes. The scheduler keeps each slave under observation. When it finds any slave idle, it tries to get a new sub-task from the task queue and loads the sub-task into the idle slave.

The general dynamic load balance algorithm can be given as follows:

Algorithm 7.4. General Dynamic Load Balance Algorithm.

INPUT: slave number K, sub-task generation algorithm and data collected algorithm.

OUTPUT: results from data collected algorithm.

1. For each slave $p \in [1, K]$, do steps 2 – 3
2. Get a new sub-task by applying the sub-task generated algorithm.
3. Load this sub-task to slave p by pack sub-task, *pvm_pkint*(), and sending it to the slave, *pvm_send*().
4. Check, if slave p has finished its job, by *pvm_recv*(). Then:
   - 4.1 Apply the data collection algorithm.
   - 4.2 Get a new sub-task by applying the sub-task generated algorithm.
   - 4.3 If the sub-task generation algorithm fails to generate a new sub-task, then goto step 5.
   - 4.4 Load this sub-task into slave p by packing the sub-task, *pvm_pkint*(), and sending it to the slave, *pvm_send*().
5. Wait for the other slaves to finish their jobs, by *pvm_recv*().
6. Apply the data collection algorithm.
7. Finally, output the results.

In the above algorithm, we need two external algorithms (one is for sub-task generated and other is for data collection), since each application may have a different sub-task generation method and a different data collection scheme.

In our parallel local IFS algorithm, we have two choice for sub-task generated. One is for steps 4–7 of Algorithm 7.1, when task granularity[1] is small. The total throughput of communication for estimating a candidate self-affine is $16 \times (N - Awid)$ bytes. Other sub-task is for the whole Algorithm 7.1, when task granularity is medium. The total throughput of communication for estimating a candidate self-affine is 16 bytes. Because of the very low speed of communication relative to the speed of the workstations, we shall use the medium task granularity method. The corresponding sub-task generation for stage one and stage two are given as follows:

Algorithm 7.5. Sub-task Generation for Stage One of the Search.

INPUT: left interpolation point index i_{j-1}, right point search interval $[s,\ e]$ and the associated region width *Awid*.
OUTPUT: one sub-task which includes the candidate self-affine region defined by (i_{j-1}, i_j) and the associated region width *Awid*.

[1]See page 38 for more.

1. Choose one right interpolation point index i_j from the search interval $(s, e]$.
2. If there is no new i_j, then output that a new sub-task cannot be generated.
3. Finally, output one sub-task which includes the candidate self-affine region index (i_{j-1}, i_j) and the associated region width $Awid$.

Algorithm 7.6. Sub-task Generation for Stage Two of the Search.

INPUT: self-affine region indices (i_{j-1}, i_j) and the associated region width search interval $[Awid, N]$.
OUTPUT: one sub-task which includes the self-affine region indices (i_{j-1}, i_j) and the associated region width $Awid_j$.

1. Choose one associated region width $Awid_j$ from the search interval $[Awid, N]$.
2. If there is no new $Awid_j$, then output that a new sub-task cannot be generated.
3. Finally, output one sub-task which includes self-affine region indices i_{j-1}, i_j and the associated region width $Awid_j$.

Both stages one and two of the search use the same data collection algorithm, which is given as follows:

Algorithm 7.7. Date Collection Algorithm.

INPUT: candidate right interpolation point index i_c, Hausdorff distance $H(R_c)$ the and associated region $(\underline{i}_{cl}, \underline{i}_{cr})$.
OUTPUT: best and next best right interpolation point indices i_j, i_s, Hausdorff distances $H(R_j)$, $H(R_s)$ and the associated regions $(\underline{i}_{jl}, \underline{i}_{jr})$, $(\underline{s}_{jl}, \underline{s}_{jr})$.

1. Initialize $H(R_{i_j})$ and $H(R_{i_s})$
2. Unpack data by $pvm\_upkint()$ and $pvm\_upkfloat()$.
3. If $H(i_c) < H(R_{i_j})$, then record i_c as the new best right interpolation point index, record the associated region $(\underline{i}_{jl}, \underline{i}_{jr}) = (\underline{i}_{cl}, \underline{i}_{cr})$ and update $H(R_{i_j}) = H_(\underline{i}_c)$.
4. Otherwise, if $H_(i_c) < H(R_{i_s})$, record i_c as the new next best right interpolation point index, record the associated region $(\underline{i}_{sl}, \underline{i}_{sr}) = (\underline{i}_{cl}, \underline{i}_{cr})$ and update $H(R_{i_s}) = H_(\underline{i}_c)$.
5. Finally, output the best and next best right interpolation point indices i_j, i_s, the Hausdorff distances $H(R_j)$, $H(R_s)$ and the associated regions $(\underline{i}_{jl}, \underline{i}_{jr})$, $(\underline{i}_{sl}, \underline{i}_{sr})$.

The parallel algorithm based on PVM and dynamic load balance can be expressed as follows:

Algorithm 7.8. Parallel Inverse Local IFS Interpolation Algorithm (Stage One Search for Self-affine Region), Master Part.

INPUT: $(x_0, y_0), \cdots, (x_N, y_N)$ and W.
OUTPUT: the number M and Self-affine Regions R_j, $j = 1, 2, \cdots, M$.

1. Initialize interpolation point indices $i_0 = 0$, $i_M = N$ and $j = 0$; register to PVM by *pvm_mytid*().
2. $j = j + 1$.
   2.1 Set up a search interval $[s,\ e]$ for i_j, where integers s and e satisfy $x_s = x_{i_{j-1}} + W$, $x_e = x_N - W$.
   2.2 Apply dynamic load Algorithm 7.4 to calculate i_j, i_s and $H(R_j)$.
3. If $i_{j-1} \neq 0$ and $i_{s-1} + W < e$ then
   3.1 Set new limits of the search interval $(i_{s-1}, e]$.
   3.2 Apply dynamic load Algorithm 7.4 to calculate i'_s, i'_{s+1} and $H(R_{i'_s})$.
   3.3 If $H(R_{i_{j-1}}) + H(R_{i_j}) > H(R_{i_{s-1}}) + H(R_{i'_s})$ then set $i_{j-1} = i_{s-1}$ and $i_j = i_s$.
4. Accept i_j as the *j-th* interpolation index. Update the search limit to $s = i_j + W$.
5. If $e \leq s$ then exit from the algorithm.
6. Goto step 2.
7. Finally, output $M = j + 1$ and all self-affine regions R_j, $j = 1, \cdots, M$.

Algorithm 7.9. Parallel Inverse Local IFS Interpolation Algorithm (Stage Two Search for Associated Regions) Master Part.

INPUT: $(x_0, y_0), \cdots, (x_N, y_N)$ and $M = j - 1$ and all self-affine regions R_j, $j = 1, \cdots, M$.
OUTPUT: Associated Regions $\underline{R}_j$, $j = 1, 2, \cdots, M$.

1. For each self-affine region R_j, $j = 1, 2, \cdots, M$, do steps 2 – 4.
2. Set a search interval $[s,\ e]$ as $(i_{j-1}, i_j]$.
3. Apply dynamic load Algorithm 7.4 to calculate i_j, i_s, $H(R_{i_j})$ and the associated regions $(\underline{i}_{jl}, \underline{i}_{jr})$ and $(\underline{i}_{sl}, \underline{i}_{sr})$.
4. Finally, output the associated regions $(\underline{i}_{jl}, \underline{i}_{jr})$ which correspond to the $H(R_{i_j})$.

Algorithm 7.10. Parallel Inverse Local IFS Interpolation Algorithm, Slave Part.

INPUT: $(x_0, y_0), \cdots, (x_N, y_N)$, self-affine region indices (i_{j-1}, i_j) and the associated region width $Awid$.
OUTPUT: the best and next best associated region indices $(\underline{i}_{jl}, \underline{i}_{jr})$, $(\underline{i}_{sl}, \underline{i}_{sr})$ and Hausdorff distances $H(R_{i_j})$ and $H(R_{i_s})$.

1. Register this process to PVM, *pvm_mytid*(); Initialize $H(R_{i_{jl}})$ and $H(R_{i_{sl}})$;
2. Wait for receipt of the new index i_{j-1} of the left interpolation point and their search interval $[s_i, e_i]$, by *pvm_recv*();
3. Unpack this new data, by *pvm_upint*();
4. For each $\underline{i}_{jr} \in [Awid, N]$ do steps 2 – 9.
5. Get $\underline{i}_{jl} = \underline{i}_{jr} - Awid$; Calculate the map w_j parameters $(a_j, c_j, d_j, e_j, f_j)$ for self-affine region indices (i_{j-1}, i_j) with the associated regions $(\underline{i}_{jl}, \underline{i}_{jr})$ and the Hausdorff distance $H_{(\underline{i}_{jr})}$.
6. If $H_{(\underline{i}_{jr})} < H(R_{i_{jl}})$, then record i_{jr} as the new best one and update $H(R_{i_{jl}}) = H_{(\underline{i}_{jr})}$.
7. Otherwise, if $H_{(\underline{i}_{jr})} < H(R_{i_{sl}})$, record i_{jr} as the new next best one and update $H(R_{i_{sl}}) = H_{(\underline{i}_{jr})}$.
8. Finally, pack the best and next best candidate associated region defined by $(\underline{i}_{jl}, \underline{i}_{jr})$, $(\underline{i}_{sl}, \underline{i}_{sr})$ and Hausdorff distances $H(R_{i_j})$ and $H(R_{i_s})$ with *pvm_pkint*(), *pvm_pkfloat*(); Send them to the master task, *pvm_send*().
9. Go to step 2.

7.4. Numerical Simulation

In this section, we first present a variety of non self-affine one-dimensional signal types, modelled with local IFS interpolation. Second, we want to see how the input constant W, which controls the minimal distance between two consecutive interpolation points along the X direction in the inverse LIFS algorithm, can change the compression ratio. Third, we distribute our computing task in a network environment and test the speed-up ratio.

In Example 7.1, we sample the sinusoid function $128\sin(2\pi x/255)$ in the interval $[0, 255]$ to get the discrete signal of length 256. We apply inverse LIFS algorithm with $W = 32$ and find that the best result is obtained when $M = 7$. The results are listed

Table XX. Local IFS calculated self-affine region (S.R) indices, associated region (A.R) indices, map parameters and Hausdorff distances for a Sinusoid Signal $128\sin(2\pi x/255)$

S.R. Index	A.R. Index	Map Param.	H	SNR
0,35	126, 179	1.06 -0.33 -131.55	0.02	40.6
36, 68	32, 80	0.34 0.44 48.51	0.03	45.83
69, 103	54, 105	-0.6 0.42 107.5	0.04	45.7
149, 104	92, 160	1.28 -0.26 -156.98	0.81	35.7
181, 150	22, 69	1.77 -0.4 -136.79	0.55	44.0
220, 182	156, 213	-0.27 0.45 -18.61	0.07	45.7
255, 221	78, 129	-2.67 -0.31 246.68	1.2	38.4

Table XXI. Signal Noise/Ratio of Local IFS and IFS

	Example 7.1		Example 7.2		Example 7.3	
	LIFS	IFS	LIFS	IFS	LIFS	IFS
SNR	45.98	31.05	4.05	2.25	24.36	21.91
H	1.62	5.46	203.35	244.45	16.89	29.97

in Table XX. In order to compare with the inverse IFS algorithm, we also illustrate the both results in Figure 49. The total signal/noise ratios from both algorithms are listed in TableXXI.

In Example 7.2, we use a real-world male speech signal of length 256. We apply the inverse LIFS algorithm with $W = 9$ and find that the best results is obtained when $M = 20$. The results are listed in Table XXII. For comparison with the inverse IFS algorithm, we illustrate both results in Figure 50. The total signal/noise ratios of both algorithm are listed in Table XXI.

In Example 7.3, we use a fractional Brownian motion signal of length 256 generated by the method used in Section 6.4 of length 256. We apply the inverse LIFS algorithm with $W = 9$ and find that the best result is obtained when $M = 21$. The results are listed in Table XXIII. For comparison with the inverse IFS algorithm, we illustrate the both results in Figure 51. The total signal/noise ratios of both algorithms are listed in Table XXI.

In these examples, we find that the local IFS approach fits the data better than the IFS does. The SNR improvement is 14DB for smooth data and 2DB for rough data.

The constant W influences the compression ratio. We simply define the compression ratio as $R = \frac{9M}{4N}$ since our original input signal uses single precision, which takes four bytes, we can use three bytes describe the self-affine and associated region and six bytes for the map parameters c_i, d_i, f_i. We choose $W = 32$, $W = 48$ and

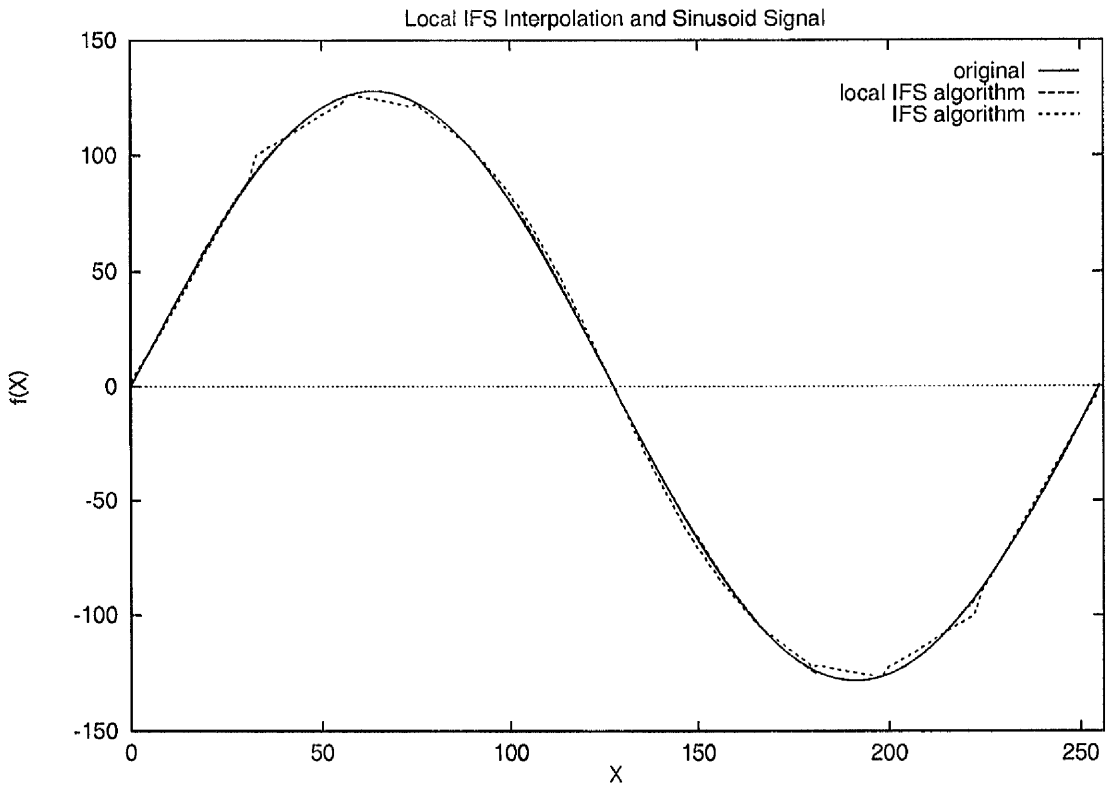

Fig. 49. Local IFS Modelling of the Sinusoid Signal $128\sin(2\pi x/255)$

Table XXII. Local IFS calculated self-affine region (S.R) indices, associated region (A.R) indices, map parameters and Hausdorff distances for a Male Speech Signal

S.R. Index	A.R. Index	Map Param.	H	SNR
0,9	22, 42	0.81 -0.99 342.52	2.4	41.1
10, 27	56, 122	-0.13 -0.02 262.64	3.0	42.3
41, 28	21, 42	2.05 0.08 157.2	94.6	18.3
42, 56	38, 217	-0.1 0.02 125.85	1.8	43.0
71, 57	37, 65	-0.02 0.99 -0.57	3.6	35.9
72, 80	48, 64	-0.71 0.54 211.43	0.7	53.8
81, 91	180, 226	0.002 -0.02 227.13	0.9	48.4
103, 92	66, 83	0.63 0.94 -34.55	2.64	43.1
113, 104	48, 81	3.23 0.9 -205.	2.8	41.0
114, 132	57, 88	0.46 0.98 -73.94	26.0	26.4
142, 133	152, 166	16.33 0.36 -2513.3	41.8	12.4
154, 143	6, 25	-5.28 0.19 229.24	64.9	14.1
170, 155	40, 70	1.88 0.95 -142.64	24.6	19.9
179, 171	109, 122	2.26 0.3 -83.88	1.7	46.7
189, 180	202, 217	-1.86 0.91 414.0	4.3	37.5
206, 190	158, 182	2.33 0.85 -351.83	9.7	32.1
207, 222	4, 32	0.2 0.99 -35.88	37.2	23.1
223, 234	61, 86	-1.92 0.97 135.58	6.1	33.9
244, 235	157, 173	2.8 0.94 -473.42	3.4	35.7
255, 245	123, 138	-1.3 0.96 165.9	9.7	31.7

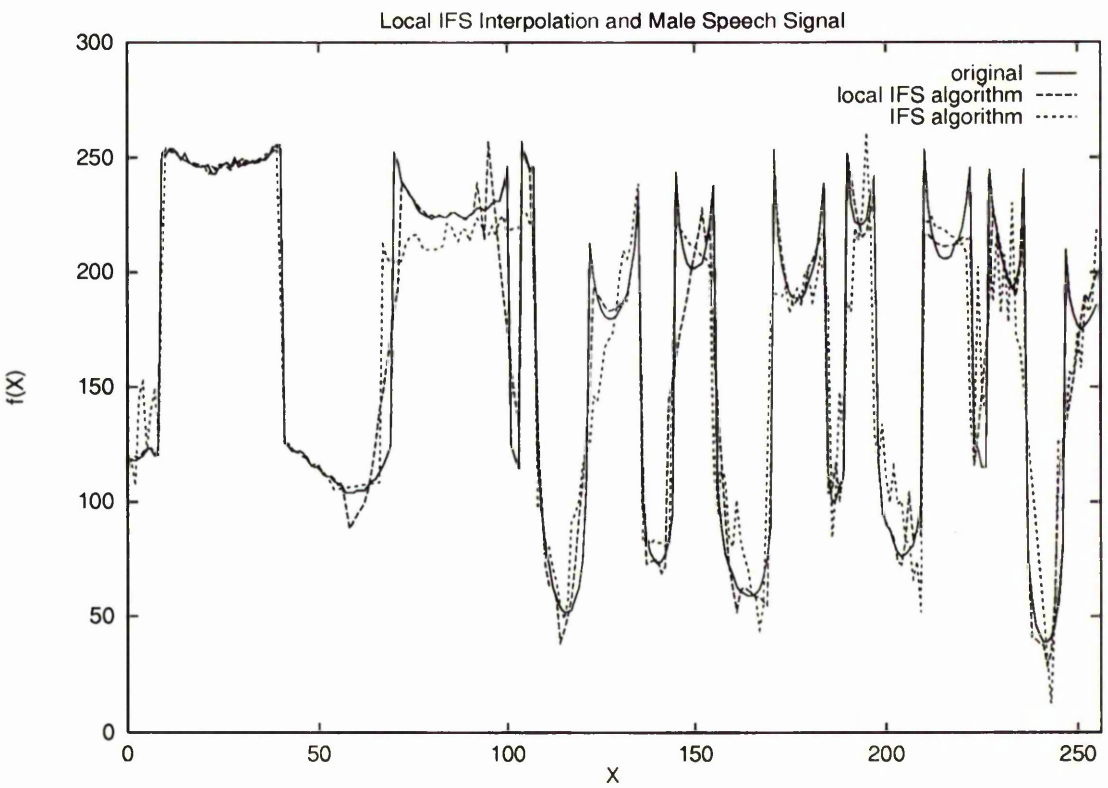

Fig. 50. Local IFS Modelling of a Male Speech Signal

Table XXIII. Local IFS calculated self-affine region (S.R) indices, associated region (A.R) indices, map parameters and Hausdorff distances for a Fractional Brownian Motion Signal (H=0.5, Scale=0.4)

S.R. Index	A.R. Index	Map Param.	H	SNR
14, 0	19, 48	5.68 -0.99 -116.08	13.6	15.1
15, 24	0, 114	0.12 0.06 -7.4	21.8	0.8
25, 40	121, 145	5.87 0.85 -813.55	6.2	21.2
52, 41	129, 231	0.15 0.13 50.94	9.5	24.8
53, 62	37, 52	0.01 -0.89 112.75	5.5	25.1
72, 63	104, 159	-0.28 -0.23 70.02	14.5	7.1
83, 73	152, 178	0.13 0.1 12.61	11.7	12.0
94, 84	133, 169	-1.31 -0.46 297.19	7.3	28.2
95, 104	92, 130	-0.79 0.23 120.39	4.77	27.19
105, 114	102, 205	-0.44 -0.45 134.78	6.7	21.76
124, 115	198, 217	4.84 0.97 -807.1	8.8	28.5
125, 140	21, 44	-5.42 0.53 251.46	6.9	29.1
141, 150	174, 192	1.5 0.77 -219.67	2.4	33.8
151, 160	57, 198	-0.69 -0.33 124.08	2.9	27.4
186, 161	121, 211	1.3 0.38 -200.8	9.1	13.6
187, 195	14, 56	-2.38 0.81 69.61	2.54	27.4
205, 196	143, 226	1.79 0.42 -352.01	17.9	9.9
206, 215	37, 69	-2.95 -0.24 44.13	10.9	22.23
216, 226	14, 126	-0.16 -0.07 -154.22	9.4	29.8
227, 241	12, 57	-2.26 0.25 -103.12	26.6	20.9
242, 255	110, 203	-0.38 -0.31 -170.01	16.4	31.7

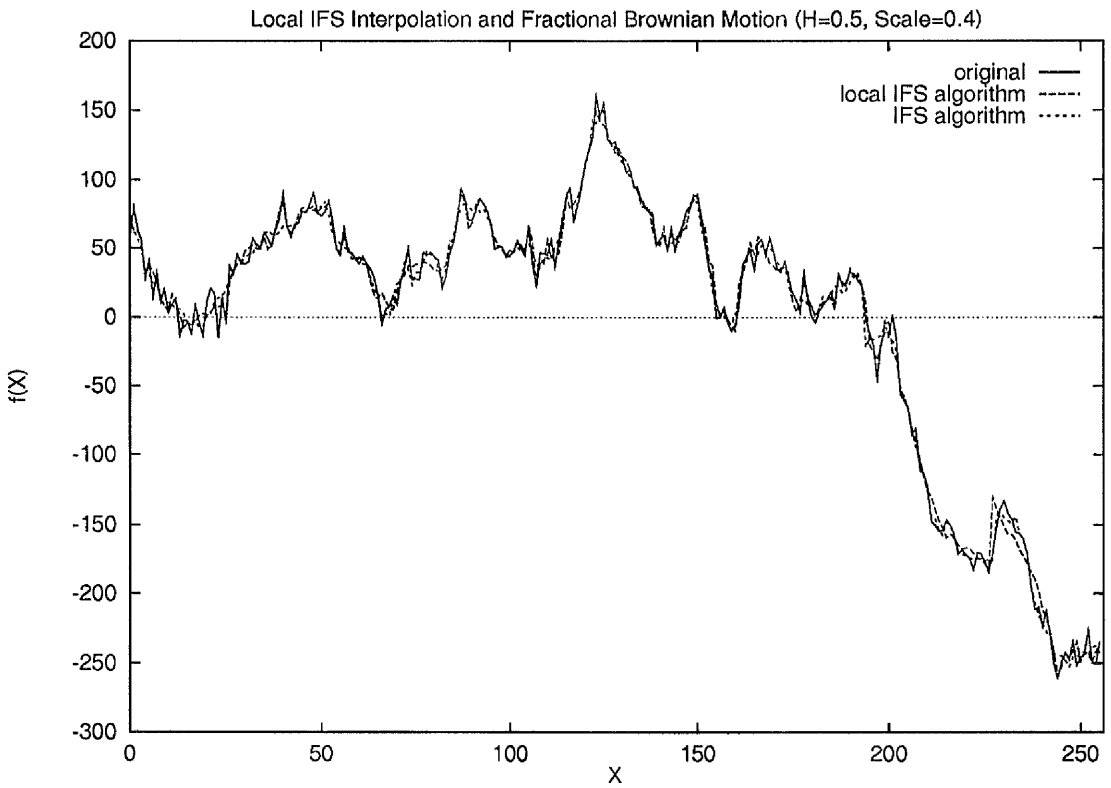

Fig. 51. Local IFS Modelling a Fractional Brownian Motion (H=0.5, Scale=0.4)

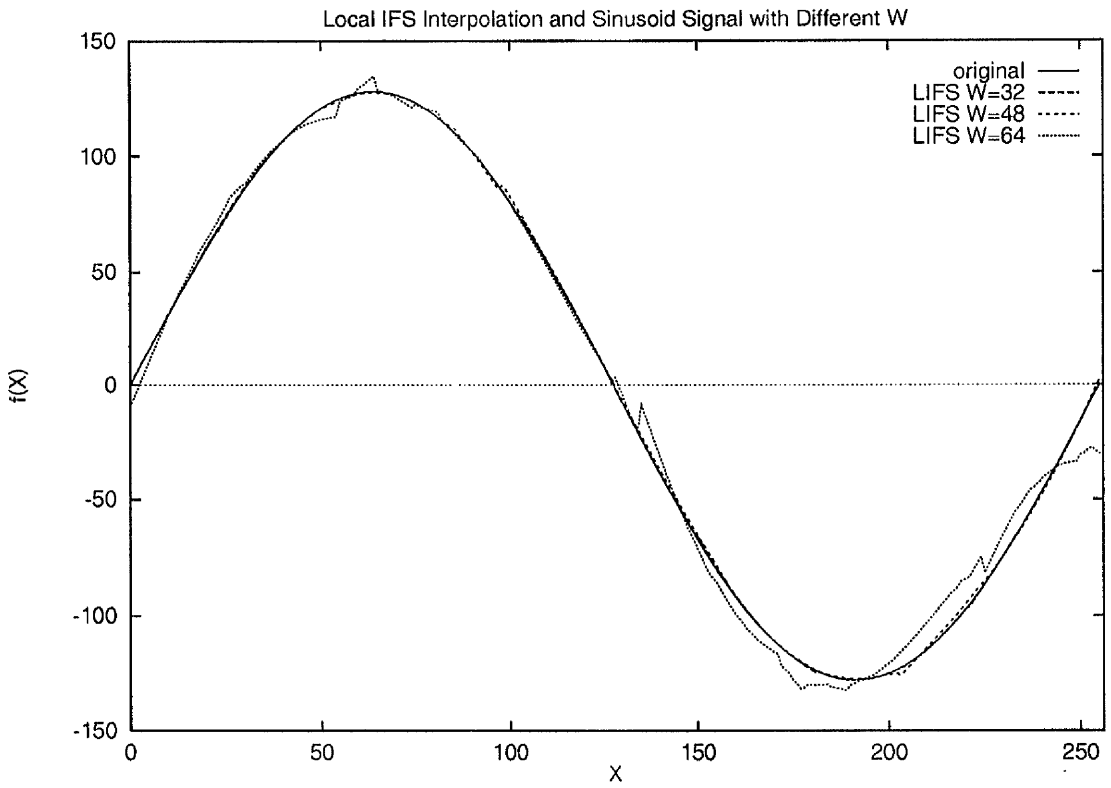

Fig. 52. Local IFS Model of a Sinusoid Signal $128\sin(2\pi x/255)$ with the different W values

Table XXIV. Local IFS Model of a Sinusoid Signal $128\sin(2\pi x/255)$ with the different W values

W	M	H	SNR	R
32	7	1.62	45.98	0.0615
48	5	3.43	38.17	0.0439
64	3	29.22	22.5	0.0264

$W = 64$ to test the influence. The results are shown in Figure 52 and Table XXIV. With $W = 32$ and $W = 48$ we get a fit to the data. but with $W = 64$ the fit is not good, although, the compression ratio is high.

To test our dynamic load balance technique, we set up a PVM configuration with three SUN clusters as shown in Figure 53. The first of these is the Department of Statistics SUN cluster which includes nine SUN work-stations (Sparc 10, ELC, IPC and SUN 470). The second is the Statistic Lab SUN cluster, which includes three SUN Sparc 10. The third is the Computing Service SUN cluster which includes two SUN Sparc 10. The first and second clusters are connected with Ethernet and the third is connected with FDDI. All machine use SUN OS 4.1.x.

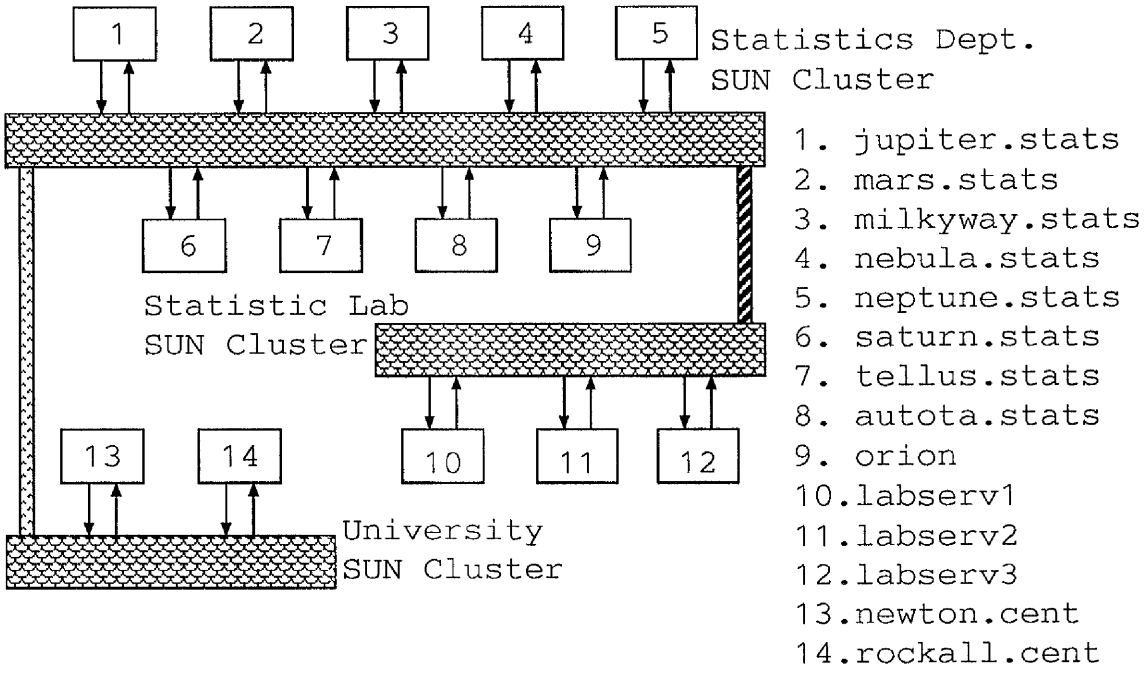

Fig. 53. Work-station Configure for PVM

Figure 54 shows the total time (computing + communication + idle) for example 7.2. There are four curves in the Figure. Two of them use a Daemon-based communication scheme and others use a TCP-based communication scheme. In the PVM environment, the TCP-based mode provides a more efficient communication path than the Daemon mode so that we can get some improvement in total time. From Table XXV we see that we get a good speed-up ratio with the dynamic load balance technique even if the number of computers is fourteen, compared with static load balance where the number of computers is seven.

More detail comparison is shown in Figure 55 and Table XXVI. The height of each box in Figure 55 indicates the scale of each sub-task. In the equal-task-load case, the fastest computers incur high idle time waiting for a new message; see, for example, computers 3, 8, 10, 11, 13, 14 computers in Table XXVI. This wastes the computing resource. In the dynamic load case, however, all computers have low idle time and keep busy in computing, as expected. We also find that Nos. 13 and 14 incur large task load. Both computers are fast Sparc 10s in cluster 3 which is not connected to the local network. The large task load indicates that the network delay across the campus is small and that the network is suitable for this type of parallel distributed computing application.

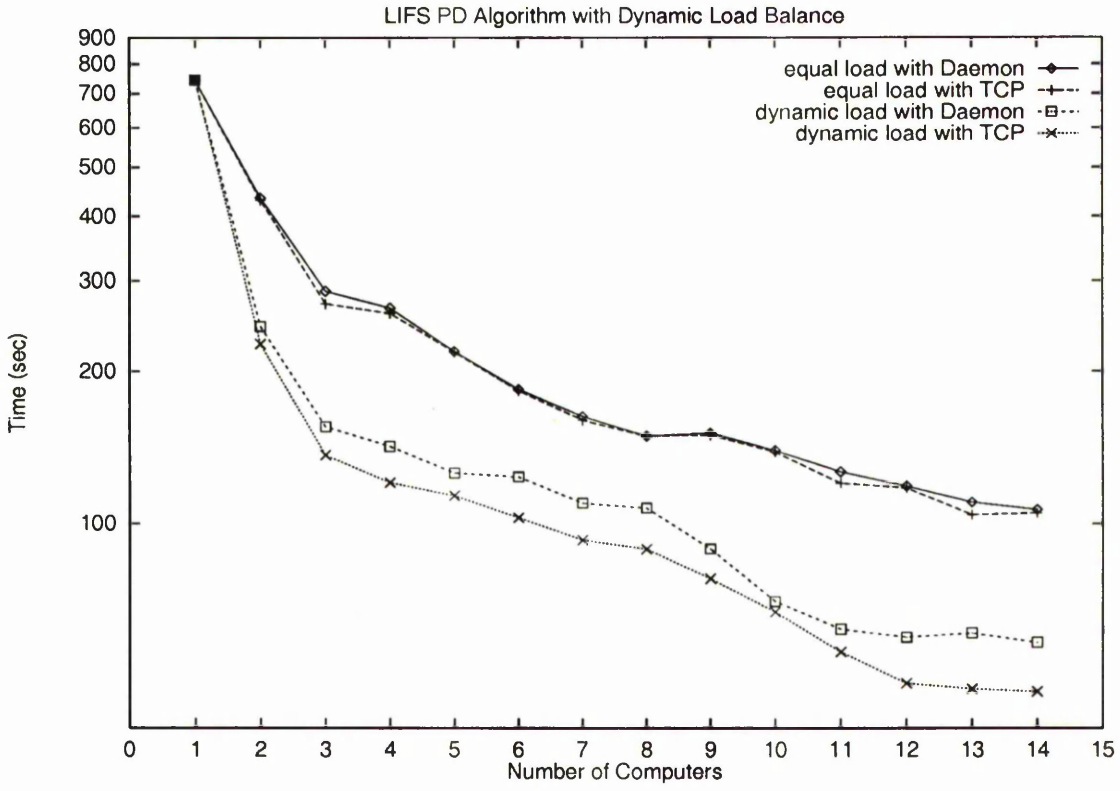

Fig. 54. Total time for example 7.2 using PVM

Table XXV. Total times (seconds) for Example 7.2 using PVM daemon and TCP communication with equal and dynamic task load

No	Equal (Daemon)	Dynamic (Daemon)	Equal (TCP)	Dynamic (TCP)
1	744.02	744.02	744.02	744.02
2	434.89	430.22	244.91	226.17
3	286.73	271.13	155.44	136.72
4	265.96	259.76	142.25	120.77
5	219.0	218.01	126.21	113.81
6	184.33	183.14	123.95	102.77
7	163.07	160.24	109.98	92.98
8	149.21	148.91	107.56	89.24
9	151.11	149.9	89.29	78.21
10	139.82	138.83	70.64	67.41
11	126.95	120.48	62.46	56.44
12	118.94	117.83	60.27	48.99
13	110.44	104.36	61.37	47.8
14	106.81	105.1	58.88	47.19

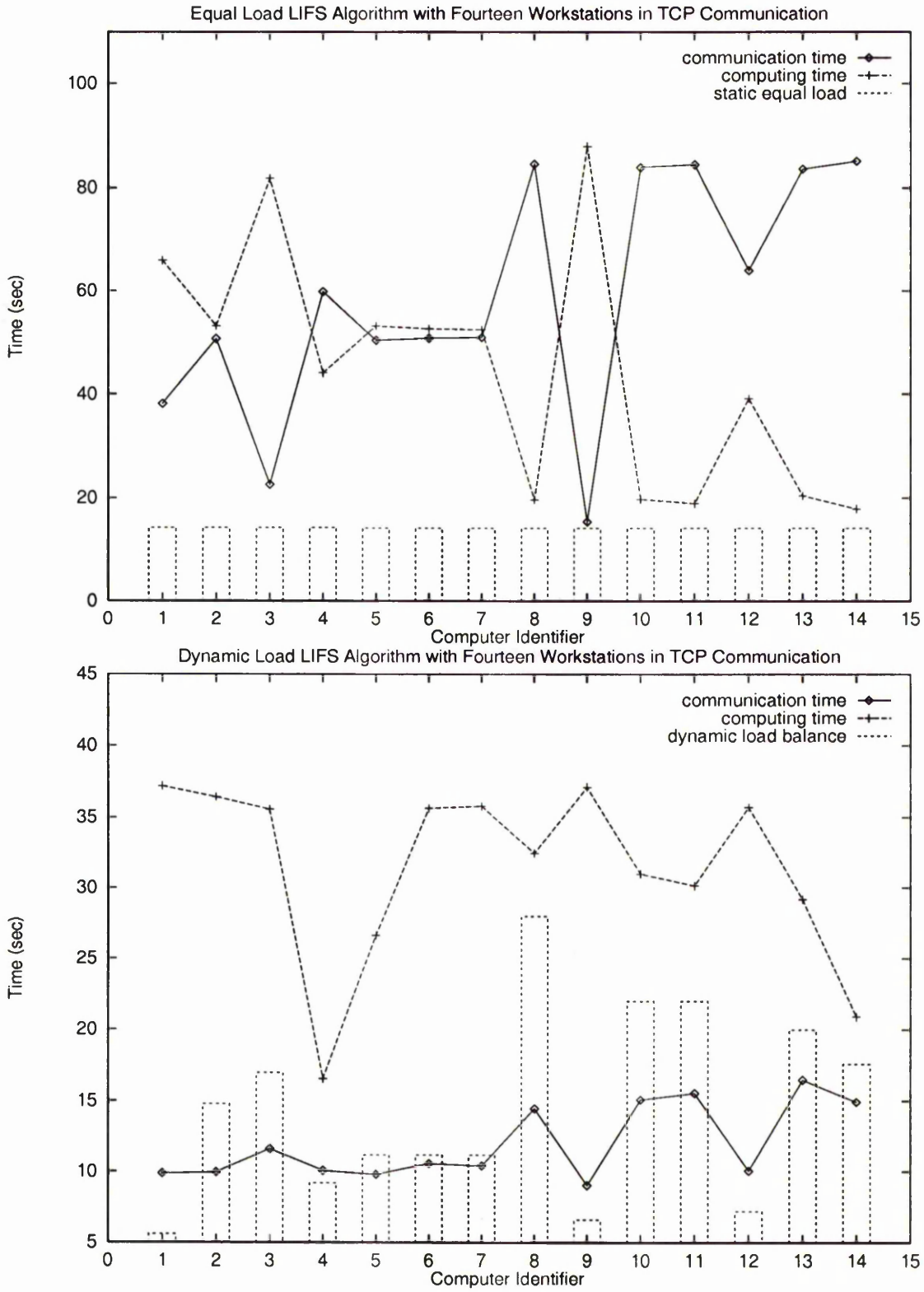

Fig. 55. Dynamic Load Balance for Example 7.2 with PVM TCP Communication Mode and Fourteen Computers, Equal Load (top diagram) and Dynamic Load (bottom diagram)

Table XXVI. Task Partitioning and Load Balance for Example 7.2 with PVM TCP Communication Mode and Fourteen Computers

Computer Name	Scale of Sub-task	Computing Time	Comm. Time	Idle Time
1	0.0714	1140	30	2720
2	0.0714	65.64	0.38	137.78
3	0.0714	52.41	0.47	49.78
4	0.0714	22.51	0.87	80.99
5	0.0714	43.98	0.36	59.51
6	0.0714	52.23	0.45	50.02
7	0.0714	51.65	0.44	50.44
8	0.0714	51.44	0.4	50.62
9	0.0714	19.49	0.28	84.29
9	0.0714	87.68	0.48	14.95
10	0.0714	19.54	0.27	83.5
11	0.0714	18.68	0.26	84.21
12	0.0714	38.88	0.35	63.67
13	0.0714	20.16	0.31	82.32
14	0.0714	17.71	0.22	84.89
1	0.0284	36.99	0.22	9.67
2	0.0854	35.39	0.77	9.86
3	0.0555	35.4	0.34	9.63
4	0.056	36.41	0.23	9.84
5	0.0557	35.41	0.35	9.46
6	0.0558	34.53	0.34	10.23
7	0.0558	34.77	0.32	10.09
8	0.14	32.22	0.51	13.97
9	0.033	36.86	0.23	8.8
10	0.11	30.72	0.41	14.61
11	0.11	29.89	0.41	15.16
12	0.037	35.52	0.51	9.55
13	0.1	28.95	0.35	16.15
14	0.081	30.59	0.28	14.72

CHAPTER 8

CONCLUSION AND DISCUSSION

8.1. Main Results

This thesis concentrates mainly on stack filtering, fractal modelling of one-dimensional discrete data and their implementation using parallel distributed algorithm.

The combination of interactive and parallel processing will lead to a new and useful application area, especially for visual science data, image analysis/processing and multimedia applications. We implemented this combination based on a parallel distributed computing environment, PVM, and the interactive application development tool, Tcl in Chapter 3. Tcl is an embeddable interpreter language and directly supports the user's extension. The approach we use is to provide a Tcl's interface for all procedures of the PVM interface library so that users can utilize any PVM procedure to do their parallel computing interactively.

In Chapter 4, we implement an interactive parallel stack filtering system based on the Interactive Parallel Distributed Computing Environment. In order to reduce the performance time of the standard stack filter, we suggest a new minimum threshold decomposition scheme, we try to minimize the number of logical operations and we utilize the CPU bit-fields parallel method to do stack filtering. We also use equal task partitioning to implement a full parallel distributed filtering algorithm on PVM. We apply the parallel stack filter to two numeric examples and the results show that the interactive parallel stack-filtering system is efficient for both sequential and parallel filtering algorithm.

In Chapter 5, we present an extended Iterated Function System (IFS) interpolation method for modelling a given discrete signal. This inverse IFS problem is a global optimal problem and there is no acceptable algorithm for obtaining the solution in reasonable time. We suggest a suboptimal search algorithm which first estimates the local self-affine region and then the map parameters, and neighbouring information for a self-affine region is used for enhancing the robustness of this suboptimal algorithm. We also implement a parallel distributed version of this algorithm using equal task partitioning and a Remote Procedure Call library. The simulation results show that the IFS approach achieves a higher signal to noise ratio than does an existing approach based on autoregressive modelling for self-affine and approximately signals, and, when the number of computers is small, the speed-up ratio is almost linear.

In Chapter 6, we use the robust IFS inverse algorithm with a local cross-validation

technique to model self-affine and approximately self-affine signals corrupted by Gaussian noise. The local cross-validation is used to compromise between the degree of smoothness and fidelity to the data. We implement the parallel distributed version of the algorithm in Parallel Virtual Machine (PVM) with optimal task partitioning. We use a simple computing model and partition tasks based only on each computer's capability. Several numerical simulation results show that the new IFS inverse algorithm achieves a higher signal to noise ratio than does autoregressive modelling for noisy self-affine or approximately self-affine signal. There is little machine idle time relative to computing time in the optimal task partitioning mode.

In Chapter 7, we apply local IFS to model non self-affine signals. The local IFS realises the IFS limit for self-affine data and is suitable for modelling general signals. However it is difficult to explore the whole parameter space to get globally optimal parameter estimates. We suggest a two-stage search scheme to estimate the self-affine region and the associated region parameters, so that we can get a suboptimal solution in a reasonable time. In the first stage, we suppose that the associated region length is twice the length of the self-affine region and we can calculate all self-affine region parameters. Then in the second stage, for each self-affine region, we search for corresponding associated region parameters from the full search space. In a network-based parallel computing environment, most performance degradation is load imbalance caused by the different machines capabilities and the external loads. We apply dynamic load balance technique based on data parallelism scheme to overcome the problem. Some numerical simulation show that our inverse local IFS algorithm works efficiently for several types of one-dimensional signals, and the parallel version with dynamic load balance can automatically have each machine busy with computing and with low idle times.

8.2. Discussion and Suggestion

In chapter 4, we cannot use normal RPC mode, in which a client sends a call and waits for the server to reply to the effect that the call has succeeded. In order to get real parallel-task sending, we use several UNIX system calls to implement multi-process communication and management, but this implementation has not been optimized. PVM is a parallel distributed computing environment. It not only provides a point-to-point communication scheme, but also provide process management and many other facilities. It is better to implement a parallel distributed algorithm on PVM than on RPC.

In a network-based parallel computing environment, we need some load balancing technique to combat imbalance. We investigated static and dynamic load balance

methods and found that dynamic load balance based on an data parallelism scheme is suitable for our algorithm and achieves better results. However dynamic load balancing requires that each sub-task be the same. We need to arrange this, if possible, so that more computing tasks can benefit from this load balance.

For parallel stack filtering, we can use the dynamic load balancing technique to enhance the parallel algorithm. Also, an adaptive stack-filtering algorithm can be implemented on the interactive stack filtering system. For modelling general noisy signals, we can implement local inverse IFS algorithms with the local cross-validation technique. Another possible research topic for local IFS which we do not address in this thesis is that of fractal compression. For a compression problem, our aim is to find the minimum number of self-affine region subject to a given error limit. The problem of compression is still open.

APPENDIX A

FUNCTION PROTOCOLS OF INTERACTIVE PARALLEL DISTRIBUTED COMPUTING ENIRONMENT

A.1. Binding the PVM User Interface Library with Tcl Language

We define here all protocols of functions of Tcl-based PVM user interface library. Most of contents come from reference manual pages of PVM 3.2.

NAME: *pvmIaddhosts* – Adds one or more hosts to the virtual machine.

SYNOPSIS: *pvmIaddhosts* hosts N.

PARAMETERS: hosts – *LIST* returning the host names.
N – the number of the hosts.

RETURN: *LIST* of info, $host_1$-start-code,$\cdots$,$host_N$-start-code.
info – integer status code. info < 0 indicates an error.
$host_i$-start-code – integer returning the start code of the host i.

NAME: *pvmIadvise* – Advises PVM to use direct task-to-task routing (TCP) or not.

SYNOPSIS: *pvmIadvise* route.

PARAMETERS: route – integer advising PVM to set up direct task-to-task (TCP) links.
PvmDontRoute (1) $\rightarrow$ don't allow direct links to this task.
PvmAllowDirect (2) $\rightarrow$ allow but don't request direct links.
PvmRouteDirect (3) $\rightarrow$ request direct links.

RETURN: info – integer status code. info < 0 indicates an error.

NAME: *pvmIbufinfo* – Returns information about the requested message buffer.

SYNOPSIS: *pvmIbufinfo* bufid.

PARAMETERS: bufid – integer specifying a particular message buffer identifier.

RETURN: *LIST* of info, bytes, msgtag, tid.
info – integer status code. info < 0 indicates an error.
bytes – integer returning the length in bytes of the entire message.
msgtag – integer returning the actual message label.
tid – integer returning the source of the message.

NAME: *pvmIconfig* – Return information about the present virtual machine configuration.

SYNOPSIS: *pvmIconfig*

PARAMETERS:

RETURN: *LIST* of info, nhost, narch, hostlist.
info – integer status code. info < 0 indicates an error.
nhost – integer returning the number of hosts (pvmds) in the virtual machine.
narch – interger returning the number of different data formats being used.
hostlist – *LIST* of hi_tid, hi_name, hi_mtu, hi_speed.
hi_tid – pvmd's task ID; hi_name – pvmd's name;
hi_mtu – pvmd's architecture; hi_speed – pvmd's relative speed.

NAME: *pvmIdelhost* – Deletes one or more hosts from the virtual machine.

SYNOPSIS: *pvmIdelhost* hostnames N.

PARAMETERS: hostnames – *LIST* returning the host names,
N – integer returning the number of hosts.

RETURN: *LIST* of info, $host_1$-error-code, $\cdots$, $host_N$-error-code.
If any value less than zero, the corresponding error appears.

NAME: *pvmIexit* – Tells the local pvmd that this process is leaving PVM.

SYNOPSIS: *pvmIexit*

PARAMETERS:

RETURN: info – integer status code. info < 0 indicates an error.

NAME: *pvmIfreebuf* – Disposes of a message buffer.

SYNOPSIS: *pvmIfreebuf* bufid

PARAMETERS: bufid – integer message buffer identifier.

RETURN: info – integer status code. info < 0 indicates an error.

NAME: *pvmIgetopt* – Returns the value of various PVM library options.

SYNOPSIS: *pvmIgetopt* what.

PARAMETERS: what – integer defining what option is being selected. See also *pvmIsetopt*.

RETURN: val – integer returning the value of the option.

NAME: *pvmIgetrbuf* – Returns the message buffer identifier for the active receive buffer.

SYNOPSIS: *pvmIgetrbuf*

PARAMETERS:

RETURN: bufid – integer returning message buffer identifier for the active receive buffer.

NAME: *pvmIgetsbuf* – Returns the message buffer identifier for the active send buffer.

SYNOPSIS: *pvmIgetsbuf*

PARAMETERS:

RETURN: bufid – integer returning message buffer identifier for the active send buffer.

NAME: *pvmIhalt* – Shuts down the entire PVM system.

SYNOPSIS: *pvmIhalt*

PARAMETERS:

RETURN: info – integer status code. info < 0 indicates an error.

NAME: *pvmIinitsend* – Clear default send buffer and specfy message encoding.

SYNOPSIS: *pvmIinitsend* encoding

PARAMETERS: encoding – integer specify the next message's encoding scheme.
PvmDataDefault (0) $\rightarrow$ XDR if heterogeneous;
PvmDataRaw (1) $\rightarrow$ no encoding;
PvmDataInPlace (2) $\rightarrow$ data left in place.

RETURN: bufid – integer returned containing the message buffer identifier and bufid < 0 indicate an error.

NAME: *pvmIkill* – Terminates a specified PVM process.

SYNOPSIS: *pvmIkill* tid

PARAMETERS: tid – integer task identifier of the PVM process to be killed (not yourself).

RETURN: info – integer status code. info < 0 indicates an error.

NAME: *pvmImcast* – Multicasts the data in the active message buffer to a set of tasks.

SYNOPSIS: *pvmImcast* tids N msgtag

PARAMETERS: tids – integer *LIST* containing the task IDs of the tasks to be sent to.
N – integer specifying the number of tasks to be sent to.
msgtag– integer message tag (*geq*0) supplied by the user.

RETURN: info – integer status code. info < 0 indicates an error.

NAME: *pvmImstat* – Returns the status of a host in the virtual machine.

SYNOPSIS: *pvmImstat* hostname

PARAMETERS: hostname – string specifying the host name.

RETURN: mstat – integer returning machine status, *PvmOk, PvmNoHost, PvmHostFail.*

NAME: *pvmImytid* – Enrols this process into PVM on its first call and returns the tid of the process on every call.

SYNOPSIS: *pvmImytid*

PARAMETERS:

RETURN: tid – integer returning task identifier of the calling PVM process.

NAME: *pvmInotify* – Notify a set of tasks about some event.

SYNOPSIS: *pvmInotify* what msgtag ntask tids

PARAMETERS: what – integer identifier of what event should trigger the notification, *PvmTaskExit, PvmHostDelete, PvmHostAdd.*
msgtag – integer message tag to be used in notification.
ntask – integer specifying the length of the tids list.
tids – integer *LIST* specifying the task IDs to be notified.

RETURN: info – integer status code. info < 0 indicates an error.

NAME: *pvmInrecv* – Non-block receive.

SYNOPSIS: *pvmInrecv* tid msgtag

PARAMETERS: tid – integer task identifier of sending process supplied by the user, a -1 matching any tid (wildcard).
msgtag – integer message tag supplied by the user, -1 matching any message tag.

RETURN: bufid – integer returning the value of the new active receive buffer identifier and bufid < 0 indicate an error.

NAME: *pvmIpk* – Pack the active message buffer with a list of prescribed data.

SYNOPSIS: *pvmIpkbyte* bytelist nitem stride;
pvmIpkshort shortlist nitem stride;
pvmIpkint intlist nitem stride;
pvmIpkdouble doublelist nitem stride;
pvmIpkfloat floatlist nitem stride;
pvmIpkstr strname.

PARAMETERS: nitem – the total number of items to be packed.
stride – The stride to be used when packing the items.
bytelist – bytes *LIST* to be packed.
shortlist – short integers *LIST* to be packed.
intlist – integers *LIST* to be packed.
doublelist – double precision real *LIST* to be packed.
floatlist – single precision real *LIST* to be packed.
strname – character string name to be packed.

RETURN: info – integer status code. info < 0 indicates an error.

NAME: *pvmIparent* – Returns the tid of the process that spawned the calling process.

SYNOPSIS: *pvmIparent*

PARAMETERS:

RETURN: tid – integer returning the task identifier of the parent of the calling process.

NAME: *pvmIperror* – Prints the error status of the last PVM call.

SYNOPSIS: *pvmIperror* msg

PARAMETERS: msg – character string supplied by the user which will be prepended to the error message of the last PVM call.

RETURN: info – integer status code. info < 0 indicates an error.

NAME: *pvmIprobe* – Check if message has arrived.

SYNOPSIS: *pvmIprobe* tid msgtag.

PARAMETERS: tid – integer task identifier of sending process supplied by the user.
msgtag – integer message tag supplied by ther user.

RETURN: bufid – integer returning the value of the new active receive buffer identifier and bufid < 0 indicate an error.

NAME: *pvmIpstat* – Returns the status of the specified PVM process.

SYNOPSIS: *pvmIpstat* tid.

PARAMETERS: tid – integer task identifier of the PVM process in question.

RETURN: status – integer returns the status of the PVM process identified by tid, *PvmOk, PvmNoTask , PvmBadParam.*

NAME: *pvmIrecv* – Blocks until a message with specified message tag has arrived from the specified source and places it in a new active receive buffer.

SYNOPSIS: *pvmIrecv* tid msgtag

PARAMETERS: tid – integer task identifier of sending process supplied by the user.
msgtag – integer message tag supplied by the user.

RETURN: bufid – integer returning the value of the new active receive buffer identifier and bufid < 0 indicate an error.

NAME: *pvmIsend* – Immediately sends the data in the active message buffer.

SYNOPSIS: *pvmIsend* tid msgtag

PARAMETERS: tid – integer task identifier of destination process.
msgtag – integer message tag supplied by the user.

RETURN: info – integer status code. info < 0 indicates an error.

NAME: *pvmIsendsig* – Sends a signal to another PVM process.

SYNOPSIS: *pvmIsendsig* tid signum.

PARAMETERS: tid – integer task identifier of PVM process to receive the signal.
signum – integer signal number.

RETURN: info – integer status code. info < 0 indicates an error.

NAME: *pvmIserror* – Sets automatic error message printing on or off for subsequent PVM calls by this process.

SYNOPSIS: *pvmIserror* set.

PARAMETERS: set – integer defining whether detection is to be turned on (1) or off (2).

RETURN: oldset – integer defining the previous setting of pvmIserror.

NAME: *pvmIsetopt* – Sets various PVM library options.

SYNOPSIS: *pvmIsetopt* what vat.

PARAMETERS: what – Integer defining what is being set. Options include: PvmRoute(1), PvmDebugMask(2), PvmAutoErr(3), PvmOutputTid(4), PvmTraceTid(6), PvmTraceCode(7), PvmFragSize(8).
val – integer specifying new setting of option.

RETURN: oldval – integer returning the previous setting of the option.

NAME: *pvmIsetrbuf* – Switches the active receive buffer and saves the previous buffer.

SYNOPSIS: *pvmIsetrbuf* bufid.

PARAMETERS: bufid – integer specifying the message buffer identifier for the new active receive buffer.

RETURN: oldbuf – integer returning the message buffer identifier for the previous active receive buffer.

NAME: *pvmIsetsbuf* – Switches the active send buffer.

SYNOPSIS: *pvmIsetsbuf* bufid.

PARAMETERS: bufid – integer the message buffer identifier for the new active send buffer.

RETURN: oldbuf – integer returning the message buffer identifier for the previous active send buffer.

NAME: *pvmIspawn* – Starts new PVM process.

SYNOPSIS: *pvmIspawn* task argv flag where ntask

PARAMETERS: task – character string containing the executable file name of the PVM process to be started.
argv – *LIST* of arguments to the executable with the end of the *LIST* by NULL (-1).
flag – integer specifying spawn options, *PvmTaskDefault(0), PvmTaskHost(1), PvmTaskArch(2), PvmTaskDebug(3), PvmTaskTrace(4).*
where – character string specifying where to start the PVM process, which depending on the value of flag.
N – integer specifying the number of copies of the executable to start up.

RETURN: *LIST* of numt, $host_1$-tid,$\cdots$,$host_N$-tid.
numt – integer returning the actual number of tasks started.
$host_i$-tid – integer returning the task identifier of new process. Value < 0 indicate an error.

NAME: *pvmIstart_pvmd* – Starts new PVM daemon.

SYNOPSIS: *pvmIstart_pvmd* argv argv block

PARAMETERS: argc – number of arguments in argv.
argv – *LIST* of arguments to the executable with the end of the *LIST* by NULL (-1).
block – integer specifying whether to block until startup complete or return immediately.

RETURN: info – integer status code. info < 0 indicates an error.

NAME: *pvmItasks* – Returns information about the tasks running on the virtual machine.

SYNOPSIS: *pvmItasks* where

PARAMETERS: where – integer specifying what stasks to return information about. 0 for all the tasks on the virtual machine; pvmd tid for all tasks on a given host; tid for a specific task.

RETURN: *LIST* of info, ntask, tasklist.
info – integer status code. info < 0 indicates an error.
ntask – integer returning the number of tasks being reported on.
tasklist – *LIST* of ti_tid, ti_ptid, ti_host, ti_flag, ti_a.out.
ti_tid – its task ID; ti_ptid – parent tid; ti_host – pvmd task ID;
ti_flag – status flag (waiting for a message, waiting for the pvmd, running);
ti_a.out – the name of this task's executable file.

NAME: *pvmItidtohost* – Returns the host of the psecified PVM process.

SYNOPSIS: *pvmItidtohost* tid.

PARAMETERS: tid – integer task identifier of the PVM process in question.

RETURN: dtid – integer returns the tid of the host's pvmd or a negative value if an error.

NAME: *pvmIupk* – Unpack the active message buffer into arrays of prescribed data type.

SYNOPSIS: *pvmIupkbyte* bytelist nitem stride;
pvmIupkshort shortlist nitem stride;
pvmIupkint intlist nitem stride;
pvmIupkdouble doublelist nitem stride;
pvmIupkfloat floatlist nitem stride;
pvmIupkstr strname.

PARAMETERS: nitem – the total number of items to be unpacked.
stride – The stride to be used when unpacking the items.
bytelist – bytes *LIST* unpacked.
shortlist – short integers *LIST* unpacked.
intlist – integers *LIST* unpacked.
doublelist – double precision real *LIST* unpacked.
floatlist – single precision real *LIST* unpacked.
strname – character string name unpacked.

RETURN: info – integer status code. info < 0 indicates an error.

NAME: *xabIon* – Start monitoring and debugging PVM with Xab.

SYNOPSIS: *xabIon*

PARAMETERS:

RETURN:

NAME: *xabIoff* – End monitoring and debugging PVM with Xab.

SYNOPSIS: *xabIoff*

PARAMETERS:

RETURN:

NAME: *xabIshowEvents* – Selects event types to be displayed.

SYNOPSIS: *xabIshowEvents* flags.

PARAMETERS: flags – integer specifying the events to be displayed in *abmon*. *XAB_NONE, XAB_SENDREC, XAB_PACK, XAB_INFO, XAB_CONTROL, XAB_DYNAMIC, XAB_GROUP, XAB_SIGNAL, XAB_BUFFER, XAB_ERROR, XAB_ALL, XAB_COMMON.*

RETURN:

NAME: *xabIbufEvents* – Sets the event buffering of a user process.

SYNOPSIS: *xabIbufEvents* num.

PARAMETERS: num – integer specifying the buffer size. A user process will store num events before sending them to *abmon*

RETURN: .

A.2. General Binary Data (GBOX) Processing Functions

NAME: *gbIcreate* – Create a new GBOX data structure and a hash table item.

SYNOPSIS: *gbIcreate*

PARAMETERS:

RETURN: gboxname – returnning the new GBOX string name.

NAME: *gbIdestroy* – Destroy a old GBOX data structure and a hash table item.

SYNOPSIS: *gbIdestroy* $gbox_1 \cdots gbox_N$

PARAMETERS: $gbox_i$ – GBOX string name to be destroyed.

RETURN:

NAME: *gbIpush* – Push new data into GBOX.

SYNOPSIS: *gbIpush* gboxname mode data packstate.

PARAMETERS: gboxname – GBOX string name.
mode – integer specifying the data mode; 0 for character, 1 for short integer, 2 for integer, 3 for long integer, 4 for single precision real, 5 for double precision real, 6 for unsigned chararcter.
data – *LIST* of data with ASCII expression.
packstate – integer specifying the packing state; 0 for raw data, 1 for others.

RETURN: nitem – integer returning the number of new data to be pushed.

NAME: *gbIpop* – Pop data from GBOX.

SYNOPSIS: *gbIpush* gboxname mode nitems packstate.

PARAMETERS: gboxname – GBOX string name.
mode – integer specifying the data mode; See also *gbIpush*.
nitem – integer returning the number of new data to be poped.
packstate – see also *gbIpush*.

RETURN: data – *LIST* of data with ASCII expression.

NAME: *gbIstate* – State the internal structure of GBOX.

SYNOPSIS: *gbIdisplay* gboxname.

PARAMETERS: gboxname – GBOX string name.

RETURN: result – *LIST* of total_size, cur_size, view_pos.
total_size – integer specifying the total size of the GBOX buffer;
cur_size – integer specifying the current data size;
view_pos – integer specifying the current position of view point.

NAME: *gbIview* – View the contents of GBOX.

SYNOPSIS: *gbIview* gboxname mode nitem packstate.

PARAMETERS: gboxname – GBOX string name.
mode – integer specifying the data mode; See also *gbIpush*.
nitem – integer returning the number of data to be viewed.
packstate – see also *gbIpush*.

RETURN: data – *LIST* of data with ASCII expression.

NAME: *gbIseek* – Move GBOX view_pos to new position.

SYNOPSIS: *gbIseek* gboxname unit pos mode.

PARAMETERS: gboxname – GBOX string name.
unit – string name specifying the data unit of size;
pos – integer specifying the new position;
mode – integer specifying the direction of moving; (0 from the starting, 1 from the current, 2 from the ending position).

RETURN: info – integer status code. info < 0 indicates an error.

NAME: *gbIfread* – Read data from a file.

SYNOPSIS: *gbIfread* filename

PARAMETERS: filename – character string file name.

RETURN: gboxfile – returning *LIST* of GBOX file structure.
magic – integer identifying the file type, see also [159];
width – integer specifying the width of data file;
height – integer specifying the height of data file;
maxval/type – integer specify maximum value of two-dimensional data or one-dimensional data type (see also *gbIpush*);
gboxname – GBOX string name.

NAME: *gbIfwrite* – Write data into a file.

SYNOPSIS: *gbIfwrite* filename gboxfile

PARAMETERS: filename – character string file name.
gboxfile – *LIST* of GBOX file structure, see also *gbIfread*.

RETURN:

Definition A.1 *modified PPM [159] file structure for one-dimensional data:*

- *magic:* integer number for identifying the file type, P10 for ASCII, P11 for binary data.
- *white-space*: (blanks, TABs, CRs, LFs).
- *width:* formatted as ASCII characters in decimal.
- *white-space.*
- height: fixed to 1 for the one-dimensional data. again in ASCII decimal.
- *white-space.*
- *type :* maximum color-component value, again in ASCII decimal.
- *white-space.*
- *array:* $width * height$ data array.

REFERENCES

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[2] S. Abenda. Inverse problem for one-dimensional fractal measures via iterated function systems and the moment method. *Inverse Problems*, 6:885–896, 1990.

[3] H. Akaike. Statistical prediction identification. *ann Inst. Statist. Math.*, 22:203–217, 1970.

[4] George S. Almas and Allan Gottlieb. *Highly Parallel Computing.* The Benjamin/Cummings Publishing Company, Inc., 1989.

[5] L. F. Anson. Fractal image compression. *BYTE*, pages 195–202, Oct 1993.

[6] G. Arce and N. C. Gallagher. State description for the root-signal set of median filters. *IEEE Trans. Acoust., Speech, Signal Process.*, 30:894–902, 1982.

[7] G. Arce and N. C. Gallagher. Stochastic analysis for the recursive median filter process. *IEEE Trans. Acoust., Speech, Signal Process.*, 34:669–679, 1988.

[8] G. R. Arce, N. C. Gallagher, and T. A. Nodes. *Median Filters: Theory for One- and Two-dimensional Filters.* Advances in Computer Vision and Image Processing. JAI Press, T.S.Huang, ed. edition, 1986.

[9] G.R. Arce. Microstatistics in signal decomposition and the optimal filtering problem. *IEEE Trans. Signal Process.*, 40:2669–2683, 1992.

[10] G.R. Arce and R.E. Foster. Detail preserving ranked-order based filters for image processing. *IEEE Trans. Acoust., Speech, Signal Process.*, 37:83–98, 1989.

[11] G.R. Arce and M.P. McLoughlin. Theoretical analysis of the max/median filter. *IEEE Trans. Acoust., Speech, Signal Process.*, 35:960–69, 1987.

[12] P. Asente, R. Swick, and J. McCormack. *X window System Toolkit: The Complete Programmer's Guide and Specification.* Digital Press, 1990.

[13] J. Astola, P. Heinonen, and Y. Neuvo. On root structure of median and median-type filters. *IEEE Trans. Acoust., Speech, Signal Process.*, 35:1199–1201, 1987.

[14] E. Ataman, V. K. Aatre, and K. M. Wong. Some statistical properties of median filters. *IEEE Trans. Acoust., Speech, Signal Process.*, 29:1073–1075, 1981.

[15] J.W. Backus, J. Bauer, F.L. Green, C. Katz, J. McCarthy, P. Naur, A.J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J.H. Wegstein, A. van Wijngaarden, and M. Woodger. Report on the algorithmic language algol60. *Numer. Math.*, 2:106–136, 1960.

[16] Z. Baharav, D. Malah, and E.D. Karnin. Hierarchical interpretation of fractal image coding and its applications to fast decoding. In *Int. Conf. on Digital Signal Processing*, Cyprus, 1993. ftp site: ftp.informatik.uni-freiburg.de:/papers/fractal/BaM*.

[17] Z. Baharav, D. Malah, and E.D. Karnin. Hierarchical interpretation of fractal image coding and its application to fast decoding. In *Intl. Conf. on Digital Signal Processing*, 1993. ftp site: ftp.informatik.uni-freiburg.de:/papers/fractal/BaM*.

[18] M.F. Barnsley. Fractal functions and interpolation. *Constr. Approx.*, 2:303–329, 1986.

[19] M.F. Barnsley. *Fractal Everywhere.* Academic Press, New York, 1988.

[20] M.F. Barnsley. *Fractals Everywhere.* New York: Academic, 1988.

[21] M.F. Barnsley and S.G. Demko. Iterated function systems and the global construction of fractals. *Proc. R. Soc. London A*, 399:243–275, 1985.

[22] M.F. Barnsley, J. Elton, and P. Massopust. Hidden variable fractal interpolation functions. *SIAM J. Math. Anal.*, 20:1221–1242, 1989.

[23] M.F. Barnsley and J.H. Elton. A new class of Markov processes for image encoding. *Adv. appl. Prob.*, 20:14–33, 1988.

[24] M.F. Barnsley, J.H. Elton, and D.P. Hardin. Recurrent iterated function systems. *Constr. Approx.*, 5:3–31, 1989.

[25] M.F. Barnsley, V. Ervin, D. Hardin, and J. Lancaster. Solution of an inverse problem for fractals and other sets. *Proc. Natl. Acad. Sci. USA*, 83:1975–1977, 1986.

[26] M.F. Barnsley and A.N. Harrington. The calculus of fractal interpolation functions. *J. Approx. Theory*, 57:14–34, 1989.

[27] M.F. Barnsley and L.P. Hurd. *Fractal Image Compression.* AK Peters, Ltd., 1993.

[28] M.F. Barnsley and A.D. Sloan. A better way to compress images. *BYTE*, pages 215–223, Jan. 1988.

[29] K.U. Barthel and T. Voyè. Adaptive fractal image coding in the frequency domain. In *Proc. of Int. Workshop on Image Process.*, pages 20–22, Budapest, Hungary, 1994. ftp site: ftp.informatik.uni-freiburg.de:/papers/fractal/BaV*.

[30] BBN. Parallel computing, past, present and future. Technical report, BBN Advanced Computers Inc., Cambridge, MA, November 1990.

[31] J.M. Beaumont. Image data compression using fractal techniques. *BT Technol J.*, 9:93–109, 1991.

[32] J.B. Bednar and T.L. Watt. Alpha-trimmed means and their relationship to the median filters. *IEEE Trans. Acoust., Speech, Signal Process.*, 32:145–153, 1987.

[33] A.L. Beguelin. Xab: A tool for minitoring pvm programs. Technical report, School of Computer Science, Carnegie Mellon University, ftp Site: dao.nectar.cs.cmu.edu (128.2.205.73) /afs/cs.cmu.edu/project/nectar-adamb/ftp, 1992.

[34] M.A. Berger. Random affine iterated function systems: Curve generation and wavelets. *SIAM Review*, 30:713–747, 1981.

[35] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation Numerical Methods.* Prentice Hall, Inc., 1989.

[36] W.W. Boles, M. Kanewski, and M. Simaan. Recursive two-dimensional median filtering algorithms for fast image root extraction. *IEEE Trans. Circuit Syst.*, 35:1323–1326, 1988.

[37] A. C. Bovik. Streaking in median filtered images. *IEEE Trans. Acoust., Speech, Signal Process.*, 35:493–503, 1987.

[38] A. C. Bovik, T. S. Huang, and D. C. Munson. The effect of median filtering on edge estimation and detection. *IEEE Trans. Pattern Anal. Machine Intell.*, 9:181–194, 1987.

[39] A.C. Bovik and T.S. Huang. A generalization of median filtering using linear combinations of order statistics. *IEEE Trans. Acoust., Speech, Signal Process.*, 31:1342–1349, 1983.

[40] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors.* Holt, Rinehart, and Winston, 1987.

[41] P.J. Brockwell and R.A. David. *Time series: Theory and Method.* Springer-Verlag, second edition edition, 1991.

[42] D.R.K. Brownrigg. Weighted median filters. *Commun. Ass. Comput. Mach.*, 27:807–818, 1984.

[43] Ralph Butler and Ewing Lusk. User's guide to the p4 programming system. Technical Report ANL-92/17, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, October 1992.

[44] C. Cabrelli, U. Molter, and R. Vrscay. *Recurrent Iterated Function Systems: Invariant Measures, A Collage Theorem and Moment Relations*, pages 71–80. Fractals in the Fundamental and Applied Sciences. Elsevier Science Publishers B.V. (North-Holland), peitgen, h.-o. and henriques, j.m. and penedo, l.f. edition, 1991.

[45] Fah-Chun Cheong. *OASIS: An agent-oriented programming language for heterogeneous distributed environment.* PhD thesis, The University of Michigan, 1992. School of Computer Science and Engineering.

[46] W.O Cochran, J.C. Hart, and P.J. Flynn. Fractal volume compression. Technical report, Washington State University, School of EECS, ftp site: ftp.informatik.uni-freiburg.de: /papers/fractal/Guide*, 1994.

[47] W.O. Cochran, J.C. Hart, and P.J. Flynn. Fractal volume compression. Technical report, Washington State University, School of EECS, ftp site: ftp.informatik.uni-freiburg.de: /papers/fractal/Guide*, 1994.

[48] Bruno Codenotti and Mauro Leoncini. *Introduction to Parallel Processing.* Addison-wesley Publishing Company, 1992.

[49] E.J. Coyle and J.H. Lin. Stack filters and the mean absolute error criterion. *IEEE Trans. Acoust., Speech, Signal Process.*, 36:1244–1254, 1988.

[50] E.J. Coyle, J.H. Lin, and M. Gabbouj. Optimal stack filtering and the estimation and structural approaches to image processing. *IEEE Trans. Acoust., Speech, Signal Process.*, 38:955–968, 1990.

[51] R.J. Crinon. The wilcoxon filter: A robust filtering scheme. In *Proc. IEEE Symp. Circuits and Systems*, 1985.

[52] I. Daubechies. The wavelets transform, time-frequency localization and signal analysis. *IEEE Trans. Inform. Theory*, 36:961–1005, 1990.

[53] H.A. David. *Order Statistics.* New York: Wiley, 1981.

[54] F. Davoine, E. Bertin, and J.M. Chassert. From rigidity to adaptive tessellations for fractal image compression: comparative studies. In *IEEE 8th Workshop on Image and Multi-dimensional Signal Process.*, Cannes, 1993. ftp site: ftp.informatik.uni-freiburg.de:/papers/fractal/Dab*.

[55] F. Davoine and J.M. Chassert. Adaptive delaunay triangulation for attractor image coding. In *12th Int. Conf. on Pattern Recognition*, Jerusalem, 1994. ftp site: ftp.informatik.uni-freiburg.de:/papers/fractal/Dab*.

[56] J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Integrated pvm framework supports heterogeneous network computing. Technical report, Oak ridge National Laboratory and University of Tennessee, ftp Site: ftp.mathcs.emory.edu /pub/vss, January 1993.

[57] C. C. Douglas, T. G. Mattson, and M. H. Schultz. Parallel programming systems for workstation clusters. Technical Report TR-975, Yale University Department of Computer Science, ftp site: ftp.cs.yale.edu /pub/TR, 1993.

[58] D. EBerly, H. Longbotham, and J. Aragon. Complete classification of roots to one-dimensional median and rank-order filters. *IEEE Trans. Signal Process.*, 39:197–200, 1991.

[59] Y. Fisher. *A Discussion of Fractal Image Compression*, pages 903–919. Chaos and Fractals. Springer Verlag, peitgen, h.o. and jurgens, h. and saupe, d. eds. edition, 1992.

[60] Y. Fisher. *Fractal Image Compression.* ACM SIGGRAPH. Prusinkiewicz, p. (ed) edition, 1992. Course Notes, ftp site: legendre.ucsd.edu: /pub/Reasearch/Fisher.

[61] J. P. Fitch, E. J. Coyle, and N. C. Gallagher. Root properties and convergence rates for median filters. *IEEE Trans. Acoust., Speech, Signal Process.*, 33:230–240, 1985.

[62] J.P. Fitch, E.J. Coyle, and N.C. Gallagher. Median filtering by threshold decomposition. *IEEE Trans. Acoust., Speech, Signal Process.*, 32:1183–1189, 1984.

[63] J.P. Fitch, E.J. Coyle, and N.C. Gallagher. Threshold decomposition of multidimensional rank order operators. *IEEE Trans. Circuits Syst.*, 32:445–450, 1985.

[64] P. Flandrin. On the spectrum of fractional brownian motions. *IEEE Trans. Information Theory*, 35:197–199, 1989.

[65] P. Flandrin. Wavelet analysis and synthesis of fractional brownian motion. *IEEE Trans. Information Theory*, 38:910–917, 1992.

[66] J. D. Foley and A. Van Dam. *Fundamentals of Interactive Computer Graphics.* Addison-wesley Publishing Company, Inc., 1982.

[67] Message Passing Interface Forum. Mpi: A message-passing interface standard, April 1994. ftp site: netlib2.cs.utk.edu /mpi/draft-final.ps.

[68] R. F. Freund and H. J. Siegel. Heterogeneous processing. *Computer*, 26(6):13–17, June 1993.

[69] M. Gabbouj and E.J. Coyle. Minimum mean absolute error stack filtering with structural constraints and goals. *IEEE Trans. Acoust., Speech, Signal Process.*, 38:955–968, 1990.

[70] N. C. Gallagher and G. L. Wise. A theoretical analysis of the properties of the median filter. *IEEE Trans. Acoust., Speech, Signal Process.*, 29:1135–1141, 1981.

[71] Z.J Gan and M. Mao. Two convergence theorems on the deterministic properties of median filters. *IEEE Trans. Signal Process.*, 39:1689–1691, 1991.

[72] P.P. Gandhi and S.A. Kassam. Performance of some rank filters for edge preserving smoothing. In *Proc. IEEE Symp. Circuits and Systems*, pages 264–267, 1987.

[73] P.P. Gandhi, I. song, and S.A. Kassam. Nonlinear smoothing filters based on rank estimates of location. *IEEE Trans. Acoust., Speech, Signal Process.*, 37:1359–1379, 1989.

[74] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM 3 User's Guide and Reference Manual*, version 3.1 edition, 1993.

[75] G. Geist and V. Sunderam. Network-based concurrent computing on the pvm system. *Concurrency: Practice and Experience*, 4(4):293–311, June 1992.

[76] G. A. Geist and V. S Sunderamm. The evolution of the pvm concurrent computing system. In *38th Annual IEEE Computer Soc Int. Computer Conf. (COMPCON Spring 93)*, pages 549–557, 1993.

[77] D. Gelernter. Generative communications in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[78] David Gelernter. Multiple tuple spaces in Linda. In *PARLE 89*, pages 20–27. Springer-Verlag, June 1989. Volume 366 of Lecture Notes in Computer Sciences.

[79] J.S. Geronimo and D. Hardin. Fractal interpolation surfaces and a related 2-d multiresolution analysis. *J. of Math. Analysis and App.*, 2:561–586, 1993.

[80] A. Gersho and R.M. Gray. *Vector Quantization and Signal Compression.* Kluwer Acad. Press, 1991.

[81] E.N. Gilbert. Lattice-theoretic properties of frontal switching functions. *J. Math. Phys.*, 33:57–67, 1954.

[82] D. Goldberg. *Genetic algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, 1989.

[83] S. Graf. Barnsley's scheme for the fractal encoding of images. *J. of Complexity*, 8:72–78, 1992.

[84] B. K. Grant and A. Skjellum. The pvm systems:an in-depth analysis and documenting study. Technical report, Lawrence Livermore National Laboratory, Numerical Mathematics Group, Livermore, CA 94550, September 1992.

[85] Andrew S. Grimshaw. Meta-systems: an approach combining parallel processing and heterogeneous distributed computing systems. In *Proceedings 1992 Workshop on Heterogeneous Parallel Processing, International Parallel Processing Symposium*, 1992.

[86] F. Hampel, E. Roncheti, P. Rousseevw, and W. Stahel. *Robust Statistics: an approach Based on Influence Functions.* New York: Wiley, 1986.

[87] C.R. Handy and G. Mantica. Inverse problems in fractal construction: Moment method solution. *Physica D*, 43:17–36, 1990.

[88] R.M. Haralick, S.R. Sternberg, and X. Zhuang. Image analysis using mathematical morphology. *IEEE Trans. Pattern Anal. Machine Intell.*, 9:532–550, 1987.

[89] R. J. Harrison. Portable tools and applications for parallel computers. *International Journal of Quantum Chemistry*, 40:847–863, 1991.

[90] P. Heinonen and Y. Neuvo. Fir-median hybrid filters. *IEEE Trans. Acoust., Speech, Signal Process.*, 35:832–838, 1987.

[91] G. Heygster. Rank filters in digital image processing. *Computer Vision, Graphics and Image Processing*, 19:148–164, 1982.

[92] Roger W. Hockney and C. R. Jesshope. *Parallel Computer 2: Architecture, Programming and Algorithms.* ZOP Publishing Ltd, second edition, 1988.

[93] J. Holland. *Adaptation in Natural and Artificial Systems.* Univ. Mich. Press, 1975.

[94] K.M. Hornik. Approximation capabilities of multilayer feedforward networks are universal. *Neural Network*, pages 251–257, 1991.

[95] G. Howlett. Blt toolkit library based on tk toolkit, 1994. ftp site: harbor.ecn.purdue.edu /pub/tcl/extensions/BLT*.

[96] J. Hutchinson. Fractal and self-similarity. *Indiana Univ. J.*, 30:713–747, 1981.

[97] Kai Hwang and Fayé A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill Book Company, 1984.

[98] Statistical Sciences Inc. S-plus user's manual, 1991. Manual.

[99] E.W. Jacobs, Y. Fisher, and R.D. Boss. Image compression: A stud of the iterated transform method. *Signal Process.*, 29:251–263, 1992.

[100] A.E. Jacquin. *A Fractal Theory of Iterated Markov Operators with Applications to Digital Image Coding.* Phd thesis, Georgia Institute of Technology, 1989.

[101] A.E. Jacquin. Fractal image coding based on a theory of iterated contractive image transformations. In *Proc. SPIE's Visual Communications and Image Processing*, pages 227–239, 1990.

[102] A.E. Jacquin. A novel fractal block-coding technique for digital images. In *Proc. ICASSP*, pages 2225–2228, 1990.

[103] A.E. Jacquin. Image coding based on a fractal theory of iterated contractive image transformations. *IEEE Trans. on Image Processing*, 1:18–30, 1992.

[104] A.E. Jacquin. Fractal image coding: A review. *Proc. of the IEEE*, 81:1451–1465, 1993.

[105] B. I. Justusson. *Median Filter: Statistical Properties.* Two-Dimensional Digital Signal Processing II. Springer Verlag, T.S.Huang, ed. edition, 1981.

[106] L.M. Kaplan and C.-C. J. Kuo. Fractal estimation from noisy data via discrete fractional gaussian noise (DFGN) and the haar basis. *IEEE Trans. Signal Process.*, 41:3554–3563, 1993.

[107] B. K. Kar. A new algorithm for order statistic and sorting. *IEEE Trans. Signal Process.*, 41:2688–2694, 1993.

[108] B.W. Kernighan and D.M. Ritchie. *The UNIX Programming Environment.* Prentice Hall, Inc., 1984.

[109] M.S. Keshner. 1/f noise. *Proc. of the IEEE*, 70:212–218, 1982.

[110] S.J. Ko and Y.H. Lee. Center weighted median filters and their applications to image enhancement. *IEEE Trans. Circuits Syst.*, 38:984–993, 1991.

[111] D. J. Kuck. A survey of parallel machine organization and programming. *Comput. Surv.*, 9:29–59, 1977.

[112] H. T. Kung, Robert Sansom, Steven Schlick, Peter Steenkiste, Matthieu Arnould, Fracois J. Bitz, Fred Christianson, Eric C. Cooper, Onat Menzilcioglu, Denise Ombres, and Brian Zill. Network-based multicomputers: an emerging parallel architecture. In *Proceedings Supercomputing 91*, pages 664–673, November 1991.

[113] J.H. LEE and J.S. Kao. A fast algorithm for two-dimensional wilcoxon filtering. In *Proc. IEEE Symp. Circuits and Systems*, pages 268–271, 1987.

[114] E. L. Lehmann. *Theory of Point Estimation.* New York: Wiley, 1983.

[115] F. Thomson Leighton. *Introduction To Parallel Algorithms and Architectures: ARRAY.TRESS.HYPERCUBES.* Morgan Kaufmann Publishers, Inc., 1992.

[116] J.H. Lin and E.J. Coyle. Minimum mean absolute error estimation over the class of generalized stack filters. *IEEE Trans. Acoust., Speech, Signal Process.*, 38:663–678, 1990.

[117] J.H. Lin, T.M. Sellke, and E.J. Coyle. Adaptive stack filtering under the mean absolute error criterion. *IEEE Trans. Acoust., Speech, Signal Process.*, 38:938–954, 1990.

[118] Y. Lin, J. Astola, and Y. Neuvo. A new class of nonlinear filters–neural filters. *IEEE Trans. Acoust., Speech, Signal Process.*, 38:663–678, 1990.

[119] Y. Lin, J. Astola, and Y. Neuvo. Adaptive stack filtering with application to image processing. *IEEE Trans. Acoust., Speech, Signal Process.*, 41:162–184, 1993.

[120] H.G. Longbotham and A.C. Bovik. Theory of order statistic filters and their relationships to linear fir filters. *IEEE Trans. Acoust., Speech, Signal Process.*, 37:257–287, 1989.

[121] M. Lottor. Internet growth (1981-1991). Request for Comment 1296, Network Information Systems Center, SRI International, January 1992.

[122] M.P. Loughlin and G.R. Arce. Deterministic properties of the recursive separable median filter. *IEEE Trans. Acoust., Speech, Signal Process.*, 35:98–106, 1987.

[123] T. Lundahl, W.J. Ohley, S.M. Kay, and R. Siffert. Fractional brownian motion: A maximum likelihood estimator and its application to image texture. *IEEE Trans. Medical Imaging*, MI-5:152–161, 1986.

[124] Rusty Lusk and Ralph Butler. Portable parallel programming with p4. In *Proceedings of the Workshop on Cluster Computing*, Tallahassee, FL, December 1992. Supercomputing Computations Research Institute, Florida State University. Proceedings available via anonymous ftp from ftp.scri.fsu.edu in directory pub/parallel-workshop.92.

[125] P. Mackerras. photo: Tk widget for image display, 1994. Dept. of Computer Science, The Australian National University, ftp site: harbor.ecn.purdue.edu /pub/tcl/extensions/photo*.

[126] S.G. Mallat. A theory for multiresolution signal decomposition: The wavelets representation. *IEEE Trans. Pattern Anal. Machine Intell.*, 11:674–693, 1989.

[127] B. Mandelbrot. *The Fractal Geometry of Naure.* Freeman, San Francisco,, 1982.

[128] B. Mandelbrot and J.W.V. Ness. Fractional brownian motions, fractional noises and applications. *SIAM Rev.*, 10:422–437, 1968.

[129] G. Mantica. Chaotic optimization and the construction of fractals: Solution of an inverse problem. *Complex System*, 3:37–62, 1989.

[130] G. Mantica. *Techniques for solving Inverse Fractal Problems*, pages 255–268. Fractals in the Fundamental and Applies Sciences. Elsevier Science Publisher B.V. (North-Holland), peitgen, h.-o. and henriques, j.m. and penedo, l.f. edition, 1991.

[131] P. Maragos and R.W. Schafer. Morphological system for multidimensional signal processing. *Proc IEEE*, 78:690–710, 1989.

[132] A. Matrone, P. Schiano, and V. Puoti. Linda and pvm – a comparison between 2 environments for parallel programming. *Parallel Computing*, 19(8):949–957, 1993.

[133] D. J. Mayhew. *Principles and Guidelines in software User Interface Design.* Prentice Hall, Inc., 1992.

[134] D.S. Mazel and M.H. Hayes. Using iterated function systems to model discrete sequences. *IEEE Trans. on Signal Processing*, 40:1724–1734, 1992.

[135] D. R. McNeil. *Interactive Data Analysis.* John Wiley & Sons, Inc., 1977.

[136] SUN Microsystems. Network programming guide, 1990. Manual.

[137] D.M. Monro and F. Dudbridge. Fractal block coding of images. *Electron. Lett.*, 28:1053–1055, 1992.

[138] D.M. Monro and F. Dudbridge. Fractal block coding of images. *Electron. Lett.*, 29:362–363, 1993.

[139] P. Morrison and E. Morrison. *Charles Babbage and his Calculating Engines*, page 244. New York: Dover, 1961.

[140] S. Muroga. *threshold Logic and Its Applications.* New York: wiley Interscience, 1971.

[141] L. Naaman and A.C. Bovik. Least squares order statistic filters for signal restoration. *IEEE Trans. Circut Syst.*, 38:244–257, 1991.

[142] Y. Nakagawa and A. Rosenfeld. A note on the use of local min and max operations in digital picture processing. *IEEE Trans. Syst., Man, Cybern.*, 8:632–635, 1978.

[143] J. Neejarvi and Y. Neuvo. sinusoidal and pulse responses of the fir-median hybrid filters. *IEEE Trans. Circuits Syst.*, 37:1552–1556, 1990.

[144] Dan Nessett and Jim Rathkopf. Computing on heterogeneous supercomputer clusters. In *Proceedings of the Workshop on Cluster Computing*, Tallahassee, FL, December 1992. Supercomputing Computations Research Institute, Florida State University. Proceedings ftp site: ftp.scri.fsu.edu.

[145] T.A. Node and N.C. Gallagher. Median filters: some modifications and their properties. *IEEE Trans. Acoust., Speech, Signal Process.*, 30:739–746, 1983.

[146] T.A. Node and N.C. Gallagher. Two-dimensional root structure and convergence properties of the separable median filter. *IEEE Trans. Acoust., Speech, Signal Process.*, 31:1350–1365, 1983.

[147] T.A. Nodes and N.C. Gallagher. Median some modifications and their properties. *IEEE Trans. Acoust., Speech, Signal Process.*, 30:739–746, 1982.

[148] T.A. Nodes and N.C. Gallagher. Median some modifications and their properties. *IEEE Trans. Acoust., Speech, Signal Process.*, 31:1350–1365, 1983.

[149] M.G. Norman and P. Thanisch. Models of machinese and computation for mapping in multicomputers. *ACM Computing Surveys*, 25(3):263–302, 1993.

[150] J. K. Ousterhout. Tcl: An embeddable command language. In *Proc. USENIX Winter Conference*, pages 133–146, 1990.

[151] J. K. Ousterhout. An X11 toolkit based on the Tcl language. In *Proc. USENIX Winter Conference*, pages 109–115, 1991.

[152] J. K Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley Publishing Company, Inc., 1993.

[153] H.O. Peitgen. *The Science of Fractal Images.* Springer-Verlag, New York, 1988.

[154] G.E. Φien, Z. Baharav, S Lepsφy, D. Malah, and E. Karnin. A new improved collage theorem with applications to multiresolution fractal image coding. In *Proc. ICASSP*, 1994. ftp site: ftp.informatik.uni-freiburg.de:/papers/fractal/BaM*.

[155] G.E. Φien, S Lepsφy, and T.A. Ramstad. A inner product space approach to image coding by contractive transformations. In *Proc. ICASSP*, pages 2773–2776, 1991.

[156] G.E. Φien, S Lepsφy, and T.A. Ramstad. Reducing the complexity of a fractal-based image coder. In *Proc. of Eur. signal Proc. Conf.*, pages 1353–1356, 1992.

[157] I. Pitas and A. N. Venetsanopoulos. *Nonlinear Digital Filters: Principles and Applications.* Boston, MA: Kluwer Academic, 1990.

[158] Ioannis Pitas and A. N. Venetsanopoulos. Order statistics in digital image processing. *Proceedings of the IEEE*, 80(12):1892–1921, 1992.

[159] J. Poskanzer. Portable pixmap format. UNIX 'man page' manual, 1991.

[160] J.P. Preparata and M.I.S. Shamos. *Computational Geometry, an Introduction.* Springer-Verlag, 1988.

[161] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants.* Springer-Verlag, New York, 1990.

[162] B. Ramamurthi and A. Gersho. Classified vector quantization of images. *IEEE Trans. on Commun.*, 34:1105–1115, 1986.

[163] J. Ramanathan and O. Zeirouni. On the wavelet transform of tractional brownian motion. *IEEE Trans. Information Theory*, 37:1156–1158, 1991.

[164] O. Rioul and M. Vetterli. Wavelets and signal processing. *IEEE Signal Process. Mag.*, 8:14–38, 1991.

[165] Yves Robert. *The Impact of Vector and Parallel Architectures on the Gaussian Elimination Algorithm.* Manchester University Press, 1990.

[166] D. Saupe. Breaking the time complexity of fractal image compression. Technical report, Universittat Freiburg, Institute für Informatik, ftp site: ftp.informatik.uni-freiburg.de: /papers/fractal/Saup*, 1994.

[167] D. Saupe and R. Hamzaoui. A guided tour of the fractal image compression literature. Technical report, Universität Freiburg, Institute für Informatik, ftp site: ftp.informatik.uni-freiburg.de: /papers/fractal/Guide*, 1994.

[168] R. Scheifler, J. Gettys, J. Flowers, R. Newman, and D. Rosenthal. *X Window System: The Complete Guide to Xlib, Xprotocol, ICCCM, XLFD.* Digital Press, second edition, 1990.

[169] D. T. Schmidt and V. S. Sunderam. Empirical-analysis of overheads in cluster environments. *Concurrency-Practice and Experience*, 6(1):1–32, 1994.

[170] R. Shonkwiler. An image algorithm for computing the hausdorff distance efficiently in linear time. *Information Processing Letters*, 30:87–89, 1989.

[171] R. Stallman. *GNU Emacs Manual*, fourth edition, version 7 edition, February 1986.

[172] M. Stone. Cross-validatory choice and assessment of statistical predictions. *J. Roy. Statist. Soc B*, 36:111–147, 1974.

[173] V. S. Sunderam. Pvm: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[174] V. S. Sunderam. Methodologies and systems for heterogeneous concurrent computing. Technical report, Emory University, Department of Mathematics and Computer Science, ftp Site: ftp.mathcs.emory.edu /pub/vss, 1993.

[175] V. S. Sunderam, G. A. Geist, J. Dongarra, and R. Manchek. The pvm concurrent computing system: Evolution, experiences and trends. Technical report, Emory University, Department of Mathematics and Computer Science, ftp Site: ftp.mathcs.emory.edu /pub/vss, 1993.

[176] Daniel Tabak. *Multiprocessors.* Prentice-Hall Int., Inc., 1990.

[177] J. W. Tukey. Nonlinear (non-superposable) methods for smoothing data. In *Cong. Rec. EASCON'74*, 1974.

[178] L. H. Turcotte. A survey of software environments for exploiting networked computing resources. Technical report, Engineering Research Center for Computational Field Simulation, P.O.Box 6176, Mississippi State, MS 39762, ftp site: bulldog.wes.army.mil, 1993.

[179] S. G. Tyan. *Median Filtering: Deterministic Properties.* Two-Dimensional Digital Signal Processing II. Springer Verlag, T.S.Huang, ed. edition, 1981.

[180] E.R. Vrscay. *Moment and Collage Methods for the Inverse Problem of Fractal Construction with Iterated Function Systems*, pages 443–459. Fractals in the Fundamental and Applies Sciences. Elsevier Science Publisher B.V. (North-Holland), peitgen, h.-o. and henriques, j.m. and penedo, l.f. edition, 1991.

[181] E.R. Vrscay and C.J. Roehrig. *Iterated function systems and the inverse problem of fractal construction using moments*, pages 250–259. Computers and Mathematics. springer Verlag, kaltofen, e. and watt, s.m. edition, 1989.

[182] E. Walach and E. Karnin. A fractal based approach to image compression. In *Proc. ICASSP*, pages 529–532, 1986.

[183] P.D. Wendt, E.J. Coyle, and N.C. Gallagher. Stack filters. *IEEE Trans. Acoust., Speech, Signal Process.*, 34:898–911, 1986.

[184] P.D. Wendt, E.J. Coyle, and N.C. Gallagher. Some convergence properties of median filters. *IEEE Trans. Circuits Syst.*, 34:276–286, 1987.

[185] S. White, A. Alund, and V. S. Sunderam. Performance of the nas parallel benchmarks on pvm based networks. Technical report, Emory University Department of Mathematics and Computer Science, ftp site: ftp.mathcs.emory.edu /pub/vss, 1993.

[186] R. Wichman, J. Astola, P. Heinonnen, and Y. Neuvo. Fir-median hybrid filters with excellent transient response in noisy conditions. *IEEE Trans. Acoust., Speech, Signal Process.*, 38:2108–2117, 1990.

[187] Wm. D. Withers. Newton's method for fractal approximation. *Const. approximation*, 5:151–170, 1989.

[188] G.W. Wornell. A karhunen-loève-like expansion for $1/f$ processes via wavelets. *IEEE Trans. Information Theory*, 36:859–861, 1990.

[189] G.W. Wornell. Wavelet-based representations for 1/f family of fractal processes. *Proc. of the IEEE*, 81:1428–1450, 1993.

[190] G.W. Wornell and A.V. Openheim. Estimation of fractal signals from noisy measurements using wavelets. *IEEE Trans. Signal Process.*, 40:611–623, 1992.

[191] G.W. Wornell and A.V. Openheim. Wavelet-based representations for a class of self-similar signals with application to fractal modulation. *IEEE Trans. Information Theory*, 38:785–800, 1992.

[192] O. Yli-Harja, J. Astola, and Y. Neuvo. Analysis of the properties of median and weighted median filters using threshold logic and stack filter representation. *IEEE Trans. Signal Process.*, 39:395–410, 1991.

[193] G. J. Yong and T. S. Huang. The effect of median filtering in edge location estimation. *Computer Vision, Graphics and Image Processing*, 15:224–245, 1981.

[194] B. Zeng, M. Gabbouj, and Y. Neuvo. A unified design method for rank order, stack, and generalized stack filters based on classical bayes decision. *IEEE Trans. Circuits and Syst.*, 38:1003–1020, 1991.