PARALLEL ALGORITHMS FOR MULTI-DIMENSIONAL WAVELET TRANSFORMS ON SHARED AND DISTRIBUTED MEMORY MACHINES

by

Lihua Yang

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

A thesis submitted to the Faculty and the Board of Trustees of the Colorado School of Mines in partial fulfillment of the requirements for the degree of Master of Science (Mathematical and Computer Sciences).

Golden, Colorado
Date June 9, 1997

Signed: Lihua Yang
Lihua Yang

Approved: Manavendra Misra
Dr. Manavendra Misra
Assistant Professor, MCS
Thesis Advisor

Golden, Colorado
Date June 9, 1997

Graeme Fairweather
Dr. Graeme Fairweather
Professor and Head
Department of Mathematical and
Computer Sciences

ABSTRACT

This thesis presents parallel algorithms for computing multi-dimensional wavelet transforms. The algorithms are implemented on both shared memory machines and distributed memory machines. Two models of parallel computing, QSM (Queuing Shared Memory), and C^3 (Computation, Communication and Congestion) are used to estimate the performance of the algorithms.

Traditional data partitioning methods call for data redistribution once a one dimensional wavelet transform is computed along each dimension. Therefore the traditional data partitioning method is not efficient for computing multi-dimensional wavelet transforms due to the high communication cost between processors in a distributed memory environment. In order to minimize communication on distributed memory machines, a block data partitioning method is presented. This requires data communication only among neighboring processors, and data redistribution among all of the processors is avoided. When a NOW (Network Of Workstations) is used, especially on a network connected by Ethernet, the block partitioning method still leads to high cost for data communication. To solve this problem, two new methods called CRBP (*Communication Reduced Block Partitioning*) and CRLP (*Communication Reduced Layer Partitioning*) are proposed.

In the CRBP and CRLP methods, once the data are distributed to each processor, n-dimensional discrete wavelet transforms are applied to the local data blocks in each processor simultaneously. Once the transforms are carried out, the results are gathered. Communication among the processors is only necessary during data distribution and data gathering.

The efficiency of the algorithms is compared through several examples implemented on a cluster of SGI workstations. The SGI cluster is a local network used as a distributed memory machine. The parallel algorithms reduce the cost both in terms of core memory used and runtime.

On shared memory machines, two kinds of parallel approaches are used to compute multi-dimensional wavelet transforms in parallel: homogeneous parallelism, and heterogeneous parallelism. In the homogeneous parallelism approach, the traditional data partitioning scheme is used. The traditional data partitioning method takes advantage of the fact that in shared memory machines each processor can access the data from the global shared address space at the same speed. The CRBP method is used in the heterogeneous parallelism approach.

The effectiveness of these approaches is demonstrated through several examples implemented on an SGI Power Challenge. The results show that near linear speedups can be achieved on the problems with mid-sized data sets.

Keywords: Parallel computing, parallel model, data distribution, C^3, QSM, wavelet transform, MPI.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Thanks to Dr. Manavendra Misra, Dr. John A. DeSanto and Dr. Willy Hereman for critiquing and proofreading this thesis and serving on my committee.

The author is very pleased to appreciate the advice, enthusiasm, and encouragement from my advisor Dr. Manavendra Misra. His help and encouragement made me complete this research.

Many thanks to Dr. Jean Bell for valuable discussions, proofreading parts of the draft of this thesis and her support in difficult moments of this research.

Many thanks to my fellow classmates for making my time at the Colorado School of Mines memorable. Special thanks to Jeff Boleng, Mohammad Abdulrahim for fruitful discussions and helpful suggestions. Thanks to Zhaobo Meng and John Wasinger for proofreading parts of the draft of this thesis.

I would like to thank Dr. David Larue, who helped me in getting MPI started and in obtaining the parameters of the machines.

Many thanks to the Department of Mathematical and Computer Sciences, the Computing Center and the Graduate School of the Colorado School of Mines for financial support during my study at CSM.

Special thanks to my parents, my husband Zhaobo Meng, and my lovely daughter Connie Y. Meng, without their support, this research would not have been possible.

Chapter 1

INTRODUCTION

Wavelet transforms provide new ways to represent and analyze signals. Due to their unique time-frequency localization properties, wavelet transforms have been successfully used in many different areas, such as image compression, pattern recognition, signal processing, and numerical analysis. Although the 1-D wavelet transform has a sequential time complexity of $O(N)$ (Holmström, 1995), where N is the length of the signal, multi-dimensional wavelet transforms are prohibitively costly from a computational perspective. In the n-dimensional case, the sequential time complexity is $O(\Pi_{i=1}^{n} N_i)$, where N_i, $i = 1, 2, \cdots, n$, is the length of the signal in the ith dimension respectively. Using parallel computers, the execution time for a multi-dimensional wavelet transform can be reduced, using an efficient parallel algorithm. As is well-known, the performance of most parallel algorithms depends on the parallel model and machine architecture chosen, so developing parallel algorithms that suit different models and architectures is important for wavelet transforms.

A number of models for parallel algorithm design and analysis have been developed and studied in the last 20 years (Gibbons, 1996). Many parallel algorithms have been designed both on Parallel Random Access Machine (PRAM) models and network based models. Examples of such parallel algorithms include list ranking and parallel prefix Euler tour techniques (Cormen *et al.*, 1990; Reif, 1993; Sabot, 1995).

In a PRAM, p ordinary processors share a global memory, and the processors can perform various arithmetic and logical operations in parallel. All processors can read

from or write to the global memory in parallel. The PRAM is simple, and it has been criticized for failing to model essential realities of parallel machines. Gibbons (1996) proposed the QSM (Queuing Shared Memory) model, a shared memory model that is suitable for bulk-synchronous algorithms running on MIMD machines. The QSM is a high-level shared memory model, yet it allows efficient emulation on lower-level more realistic models (Gibbons, 1996). The QSM is a reasonable choice for modeling shared memory computing and developing multi-dimensional wavelet transform algorithms.

The level of network-based models are considered by many to be too low. Therefore network-based models are not applied broadly, and often do not reflect the current generation of parallel machines (Gibbons, 1996). Recently, some models, such as the BSP (Bulk-Synchronous Parallel) model (Valiant, 1990; Kalantery *et al.*, 1995), and the LogP model (Culler *et al.*, 1993; Culler *et al.*, 1996), have gained popularity for developing parallel algorithms. However, these models do not model link or processor congestion due to communication. Hambrush and Khokhar (1996) developed the C^3 (Computation, Communication and Congestion) model, a parallel model for developing and analyzing algorithms on coarse-grained distributed memory machines. The C^3 model evaluates the complexity of computation, the pattern of communication, and the potential congestion arising during communication. It can serve as a platform for the development of coarse-grained algorithms sensitive to the system parameters. Therefore, the C^3 model has been selected as the model for distributed memory machines in this research.

1.1 Motivation

A strong motivation for studying multi-dimensional wavelet transforms is that we often have to deal with multi-dimensional signals that represent reality. For example,

the measurement of physical signals are most generally 3-D in space and 1-D in time, which is already 4-D; and if the measurement also depends on other parameters (for example, the coordinate system also moves), then additional dimensions are involved. We will therefore discuss the general n-D case, where n, the number of dimensions, can be a variable itself. Similar to the n-D Fourier transforms, multi-dimensional wavelet transforms have been successfully applied to signal analysis and data processing. For example, multi-dimensional data compression can reach a compression ratio higher than the one dimensional wavelet transform (Meng & Yang, 1995). Besides, multi-dimensional wavelet transforms have outstanding direction sensitive features (very similar to multi-dimensional Fourier transforms), which could be used in filter design (e.g. to design a fan filter to reduce the noise in geophysical data) and seismic data processing (Yang & Meng, 1995).

Although there has been a substantial amount of work performed in the area of parallel wavelet transforms, the parallel implementation of n-D wavelet transforms is still an open problem. There are two classes of work done on parallel wavelet transforms. The first class implements parallel wavelet transforms in hardware (Kita, 1993; Hoyt & Weschsler, 1992; Parhi & Nishitani, 1993; Sahinoglou & Cabrera, 1991). The advantage of these methods is high efficiency, however, the disadvantage is the extra and special hardware needed. The second class implements parallel wavelet transforms in software (Misra & Prasanna, 1992; Misra & Nichols, 1994; Holmström, 1995; Lu, 1993; Lega *et al.*, 1995; Caulfield, 1992; Pic & Essafi, 1993). The advantage here is that no special hardware is needed. The disadvantage is, of course, that it is less efficient than the hardware implementations. These articles use fine-grained Single Instruction Multiple Data (SIMD) stream machines such as the CM-200, CM-2, SYMPATI-2. The work presented here differs from the above and implements

parallel wavelet transforms on coarse-grained distributed memory and shared memory architectures.

Are there any fundamentally harder issues related to exploiting parallelism in n-D transforms as compared to 1-D wavelet transforms? The answer is yes. The difficulties in n-dimensional parallel wavelet transforms are mainly due to the heavy data dependency in the computation. There are four steps in designing parallel algorithms, i.e., partitioning, communication, agglomeration and mapping (Foster, 1994), so the most difficult part for n-D wavelet transforms is how to efficiently map the data to different PE's and carry out the required communication of data efficiently.

1.2 Research Performed

The research performed in this thesis is the implementation of parallel n-D wavelet transforms on shared memory machines and distributed memory machines. Attractive speedups over sequential implementations have been achieved on shared memory machines. Parallel implementations on distributed memory machines are shown to gain from the reduction of core memory and runtime costs. The main steps taken in this research work are as follows:

- Select a sequential algorithm for computing n-dimensional wavelet transforms (Press *et al.*, 1992).

- Design the parallel algorithms for distributed memory machines. In order to minimize the data communication among processors, several data partitioning and data mapping methods are presented and discussed. Estimate the performance, then redesign the algorithm to reduce the communication, implement on the local network SGI cluster.

- Assess the performance of the algorithms by applying the wavelet transforms to image compression. Comparing the reconstructed image with the original, the results show that the parallel wavelet transform algorithms are reliable.

- Design the parallel algorithm for shared memory machines, estimate the performance, implement the design on the SGI Power Challenge.

- Test the code using data sets of different sizes, record each runtime and speedup.

1.3 Main Contributions of This Research

The main contributions of this research include:

- The general case, n-dimensional, parallel wavelet transforms are carried out in this research.

- Most previous work on parallel wavelet transforms was carried out on fine grained SIMD machines. This work developed parallel algorithms for coarse grained machines.

- For distributed memory machines, two new algorithms, CRBP and CRLP, are designed in this research. In the CRBP and CRLP, once the data are distributed to each processor, n-dimensional discrete wavelet transforms are applied to the local data blocks in each processor simultaneously. Once the transforms are carried out, the results are gathered. Communication between processors is only necessary during data distribution and data gathering. Therefore the CRBP and CRLP algorithms are especially efficient when a NOW (Network Of Workstations) is used. Both CRBP and CRLP algorithms are tested on the SGI cluster, with various data set sizes, and various number of workstations being

used. The results show that attractive speedup can be achieved. Examples of image compression using wavelet transforms show that high quality reconstructions can be achieved with high compression ratios.

- For shared memory machines, two kinds of parallel approaches, homogeneous parallelism and heterogeneous parallelism, are used to compute n-dimensional wavelet transforms in parallel. Homogeneous and heterogeneous parallel algorithms on n-dimensional wavelet transforms are designed in this research. The effectiveness of these approaches is demonstrated through several examples implemented on an SGI Power Challenge.

Therefore, this research provides a new way to compute the n-dimensional wavelet transforms in parallel.

1.4 Thesis Arrangement

The remaining chapters in the thesis are organized as follows:

Chapter 2 investigates the shared memory model QSM and the distributed memory model C^3. The definitions, performance metrics, and the advantages of these two models are described.

Chapter 3 introduces 1-D wavelet transforms, n-dimensional wavelet transforms, and the related sequential algorithms to implement these transforms. Related research by others on the parallel implementation of wavelet transforms is described in the last section of this chapter.

Chapter 4 presents the parallel algorithms for n-dimensional wavelet transforms on distributed memory machines. Issues such as data partitioning, data communication and data mapping are discussed in detail. Several kinds of data partitioning

methods are described and compared with each other. The performance of the algorithms on C^3 is also estimated. The implementation of these parallel algorithms on a network of workstations is addressed. The Message-Passing Interface (MPI) (Pacheco, 1997) is used in this implementation. The assessment of the algorithms and the numerical results are presented in this chapter.

Chapter 5 presents the parallel algorithms for n-dimensional wavelet transforms on shared memory machines. The exploitation of homogeneous and heterogeneous parallelism are used to design the algorithms. We also estimate the algorithm performance on QSM and implement the algorithms on an SGI Power Challenge. The C Multi-Processing (MP) directives are inserted in the code, directing the compiler to generate calls to the multiprocessing library. The runtime performance is also presented.

Chapter 6 presents the conclusions and future work arising from this research.

Chapter 2

PARALLEL MODELS OF COMPUTATION: C^3 AND QSM

2.1 Introduction

An important research area in parallel processing is to develop efficient models of parallel computation. Efficient and widely used models provide a standard that can be relied upon by programmers, algorithm designers, software and hardware vendors. These models can make parallel machines cheaper to build and easier to use. In this thesis, two recently proposed models have been chosen to serve as models of shared memory and distributed memory computation. These two models are QSM (Queuing Shared Memory) and C^3 (Computation, Communication and Congestion). In the following two sections, these models will be introduced briefly.

2.2 QSM: Queuing Shared Memory Model

Gibbons (1996) has developed a new model, the Queuing Shared Memory (QSM) model, that accounts for limited communication bandwidth while keeping a shared memory abstraction. The QSM model consists of a number of processors, each with its own private memory, communicating by reading and writing locations in a shared memory. Processors execute a sequence of synchronized phases, each consisting of three sub-phases: Read, Compute and Write. Each processor can operate asynchronously during the sub-phases, so the QSM model is suitable for bulk-synchronous algorithms running on MIMD (Multiple Instruction, Multiple Data Stream) shared

memory machines. The term bulk-synchronous means that the processors operate asynchronously between a sequence of barriers. The QSM model is simple, and has only two parameters: the number of processors p and the bandwidth gap g. The bandwidth gap g is the rate of operations divided by the rate of communication among the processors. Another desirable feature of the QSM model is that it is easy to emulate it on lower-level machine models. This emulation is *work efficient*, where *work* of an algorithm is defined as the number of processors multiplied by the time complexity of the algorithm; therefore, *work efficiency* means that the two models perform the same amount of work. These features of QSM imply that the model can be applied widely. QSM therefore provides a nice balance between simplicity, accuracy and broad applicability (Gibbons, 1996). From this point of view, in many respects, the QSM model is more desirable than other shared memory models such as PRAM (Parallel Random Access Machine).

2.2.1 Model definition

The QSM model consists of a number of processors, each with its own private memory, communicating by reading and writing locations in shared memory. Processors execute a sequence of synchronized phases, each consisting of the following three sub-phases:

- Read: each processor i copies the contents of r_i shared memory locations into its local memory.
- Compute: each processor i performs C_i RAM operations, involving only its private state and private memory.
- Write: each processor i writes to w_i shared memory locations.

Concurrent reads or concurrent writes to the same shared memory location are permitted in a phase (both reads and writes occurring simultaneously are forbidden). In the case of multiple simultaneous writes to a location x, an arbitrary write to x succeeds in writing at the end of the phase. But there is a cost for such *contention*, and the cost for a phase depends on the maximum *contention* to a location in the phase. The maximum *contention* of a QSM phase is the maximum over all locations x, of the number of processors reading x (or the number of processors writing x) during the phase. The shared memory of the QSM model can be viewed as a collection of queues, one per shared memory location; requests to read or write at a location queue up and are serviced one at a time. The maximum *contention* is the maximum delay encountered in a queue. The cost for a phase is defined as: $\max\{m_{op}, k, g \cdot m_{rw}\}$, where m_{op} is the maximum number of local operations by a processor, k is the maximum contention, g (≥ 1) is the local instruction rate divided by the communication rate, and m_{rw} is the maximum number of shared memory reads or writes by a processor. The time complexity of a QSM algorithm is therefore the sum of the time costs for all phases.

Note that the parameter g reflects the gap between the local instruction rate and the communication rate.

2.3 C^3: A Model of Distributed Memory Parallel Computation

C^3, a parallel model for coarse-grained distributed memory machines, was developed by Hambrusch *et al.* (1996). The advantage of the C^3 model is that for a given parallel algorithm and target architecture, the C^3 model evaluates the complexity of computation, the pattern of communication, and the potential congestion arising during communication (Hambrusch & Khokhar, 1996). The parameters of the model

include:

- p, the number of processors;
- h, the latency of the communication network, i.e., the average distance:

$$\sum_{0 \leq i,j \leq p-1} d_{i,j}/p^2,$$

  where $d_{i,j}$ is the minimum number of communication hops between two processors i and j;

- b, the bisection width of the communication network. The bisection width is defined as the minimum number of links that have to be cut in order to disconnect the network into two halves with identical number of processors;
- s, the set-up cost for a message;
- and l, the length of a packet (in bytes).

The C^3 model can serve as a platform for the development of coarse-grained algorithms sensitive to the parameters of a parallel machine (Hambrusch & Khokhar, 1996).

2.3.1 The definition of the C^3 model

In the C^3 model, it is assumed that computation is synchronized by a barrier-style synchronization mechanism. An algorithm can be partitioned into a sequence of super-steps, each corresponding to local computation followed by sending and receiving messages. The processors are synchronized at the end of each super-step. The cost of a super-step depends on the cost of computation units and the cost of

communication units in the super-step. The performance of an algorithm is the sum of the time costs for its super-steps.

The metric used by the C^3 model to compute the communication and computation units of a super-step includes the parameters defined above: the number of processors p, the latency of the communication network h, the bisection width of the communication network b, the set-up cost for a message s and the length of a packet l (in bytes). The following two subsections will discuss the metric in detail.

2.3.2 Cost of computation and communication units

The computation units in a super-step are charged $\max_{0 \leq i \leq p-1} \lceil t_i/l \rceil$, where t_i bytes are accessed by processor P_i in one step. If $t_i < l$, one computation unit is charged. This encourages at least one unit of computation in a super-step.

The communication units charged to one super-step reflect the time spent in sending and receiving messages, the time messages are enroute under ideal conditions, the amount of congestion that could occur, and an estimate on the resulting delay.

There are two communication protocols considered in the model: blocking and non-blocking. The blocking protocol includes blocking send and blocking receive, and the non-blocking protocol includes non-blocking send and non-blocking receive. The source processor can not perform other operations during a blocking send until that destination processor has completed the reception of the message. During a non-blocking send, the source processor can perform other operations once the send buffer has been filled with a message. Non-blocking sends therefore allow the overlapping of communication and computation operations. Non-blocking and blocking receive operations can be defined in a similar way.

While sending a single message consisting of $L_{i,j}$ bytes from processor p_i to

processor p_j, for non-blocking send and receive protocol, the send time for a message is

$$S_{i,j} = s + \lceil \frac{L_{i,j}}{l} \rceil \cdot h,$$

where s is the set up cost for a message, and $\lceil L_{i,j}/l \rceil$ is the number of the packets, and h is the network latency. The corresponding receive time is

$$R_{i,j} = \lceil \frac{L_{i,j}}{l} \rceil.$$

For blocking send and non-blocking receive protocol, processor p_i is charged $s+h$ to initiate communication with processor p_j. After that both processors are engaged in the sending of the message, both send and receive time accumulate another $s + h$ when p_j sends a confirmation back to p_i. Therefore the send time for a message is

$$S_{i,j} = 2(s + h) + \lceil \frac{L_{i,j}}{l} \rceil \cdot h,$$

where processor p_i is charged $s + h$ to initiate communication with processor p_j, both send and receive time add another $s + h$ when p_j sends a confirmation back to p_i. The corresponding receive time is

$$R_{i,j} = s + h + \lceil \frac{L_{i,j}}{l} \rceil \cdot h.$$

If sending multiple messages from p_i, the metric is more complicated, for details refer to Hambrusch and Khokhar (1996).

A processor spends $S_i + R_i$ time in sending and receiving a message in a super-step. The overall communication cost of a super-step of all processors is therefore $\max_{0 \leq i \leq p-1}\{S_i + R_i\}$, not including the delay caused by congestion on the links of

the network.

Congestion plays an important role in the time required to finish the communication. Congestion depends on the architecture, the routing path, and the amount of data sent between processor pairs. It is therefore difficult to evaluate. The C^3 model measures two kinds of congestion: C_l, the congestion over links and C_p the congestion at the processors. Two variables are defined: *cong*, the total number of processor pairs communicating and L_a, the average number of packets routed between processors.

The C^3 model defines the congestion over links:

$$C_l = L_a \cdot \lceil cong/b \rceil,$$

where b is the bisection width of the machine; and the congestion at processors:

$$C_p = L_a \cdot \lceil cong/p \rceil \cdot h,$$

where p is the number of processors and h is the latency of the network.

Overall, the total number of the communication units charged to a super-step is: $\max_{0 \le i \le p-1}\{S_i + R_i\} + C_l + C_p$.

The two parallel models QSM and C^3 have been discussed as above. In Chapters 4 and 5, these models will be used to evaluate the parallel wavelet transform algorithms on real parallel architectures, although these computational models do not accurately capture all the salient features of the machines used in this work.

Chapter 3

WAVELET TRANSFORMS

This chapter describes the mathematical theory underlying wavelet transforms, presents the sequential wavelet transform algorithm, and introduces the state of art in parallel implementation of wavelet transforms.

3.1 Introduction

Wavelets can be defined as localized waves (Strang & Nguyen, 1996). The wavelet transform is a tool to represent data or functions using different frequency components. These components can then be used to study various characteristics of the data. Such a process is called multiresolution analysis (Daubechies, 1992; Strang & Nguyen, 1996).

For simplicity, we will use signal processing examples to explain the wavelet transforms. A signal is a series evolved in time (for example, the displacements of a particle in motion; the amplitude of the pressure on an eardrum), which can be viewed either in frequency or the time domain. The most attractive feature of a wavelet transform is that it separates different frequency components at different times.

There are many kinds of wavelets, such as Haar, Gabor, Mallat and Daubechies (Daubechies, 1992; Chui, 1992; Strang & Nguyen, 1996; Mallat, 1989). Among these different wavelets, perhaps the most popular are the Daubechies wavelets (Daubechies, 1992). There are two main advantages of the Daubechies wavelets: the transform uses

a finite number of filter coefficients; and both the mother wavelet and the scaling function are compactly supported (finitely localized). In addition, the Fourier transform of these functions are compactly supported too. The last property leads to many important applications in physics. So far, no other wavelets found have the last property, though they may have other nice properties that Daubechies wavelets lack (especially smoothness). In parallel computing, it is desirable to use finite length filters. This thesis will therefore focus on Daubechies wavelets.

In section 3.2, one dimensional wavelet transforms will be discussed; in section 3.3, the multi-dimensional wavelet transforms will be discussed; in section 3.4, the state of art in parallel implementation of wavelet transforms will be introduced.

3.2 One Dimensional Wavelet Transform

Section 3.2.1 describes the mathematical theory underlying wavelet transforms. Section 3.2.2 describes the computation of the Discrete Wavelet Transforms (DWT).

3.2.1 Wavelet transform

The one dimensional wavelet transform is related to two functions: the scaling function ϕ and the mother wavelet ψ. The scaling function ϕ is the unique function characterized by

$$\phi(x) = \sqrt{2}\sum_i h_i \phi(2x - i), \tag{3.1}$$

and the mother wavelet ψ is characterized by

$$\psi(x) = \sqrt{2}\sum_i g_i \phi(2x - i), \tag{3.2}$$

where h_i is called the *summing* filter, and g_i is called the *differencing* filter. Each kind of wavelet transform defines its own filter coefficients.

To construct the Daubechies compactly supported wavelets, we need to find the finite *summing* filter (Daubechies, 1992) $\{h_i\}$ satisfying

$$\sum_i h_i h_{i+2k} = \delta_{k,0}, \tag{3.3}$$

$$\sum_i h_{2i} = \sum_i h_{2i+1} = 2^{-1/2}. \tag{3.4}$$

The length of the filter $\{h_i\}$ is called the *order* of the Daubechies' wavelets. The order could be $4, 6, \cdots, 20$ (orders higher than 20 are rarely seen). The higher the order, the smoother the mother wavelet and scaling function, and the longer the length of the filters. This in turn implies that higher order wavelet transforms are more costly to compute. For more details see Daubechies (1992). Figure 3.1 shows the Daubechies No. 6 scaling function and the mother wavelet.

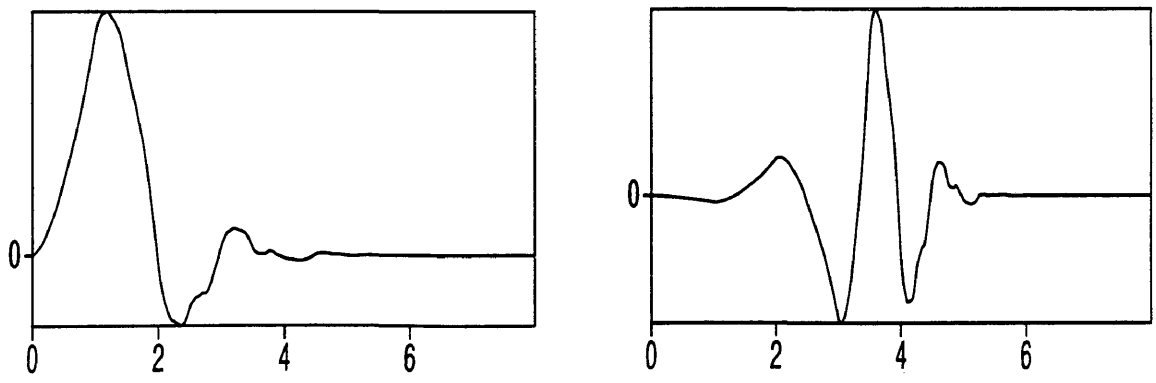

FIG. 3.1. The DAUB6 scaling function ϕ (left) and mother wavelet ψ (right).

These two sets of functions will form the *basis* for signal representation. The

scaling function and the mother wavelet are specially chosen to have very important properties such as *localization* and high *order of zero-crossing*. Here, *localization* means that both the scaling function and the mother wavelet are window-like functions both in *time* and *frequency* domain; the mother wavelet with many *order of zero-crossing* means that such a function can represent a signal with less coefficients (Wickerhauser, 1992) than one with less order of *zero-crossing*.

A scaling function or a mother wavelet can be *dilated* to the jth level, and *translated* to the ith position, as shown by:

$$\psi_{j,i}(x) = 2^{-j/2}\psi(2^j x + i). \tag{3.5}$$

We can calculate the wavelet coefficients of a signal f by the following process

$$c_i^0 =< f, \phi_{0,i} > \tag{3.6}$$

and for every level $j = 1, 2, ...$ to decompose by the Mallat fast decomposition algorithm (Mallat, 1989)

$$c_i^j = \sum_n h_{n-2i} c_n^{j-1}, \tag{3.7}$$

$$d_i^j = \sum_n g_{n-2i} c_n^{j-1}, \tag{3.8}$$

where $g_i = (-1)^i h_{-i+1}$. Figure 3.2 shows the pyramid structure of the wavelet transform.

To reconstruct the signal f, by the reconstruction algorithm (Mallat, 1989)

$$c_i^{j-1} = \sum_k [h_{i-2k} c_k^j + g_{i-2k} d_k^j], \tag{3.9}$$

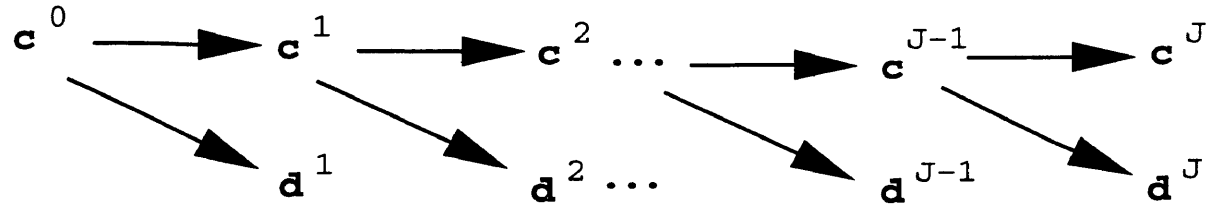

FIG. 3.2. The pyramid structure of the wavelet transform.

we can obtain the coefficients $\{c_i^0, i \in \mathbf{Z}\}$, and then obtain f^0 by

$$f^0(x) = \sum_n c_i^0 \phi_{0,i}(x). \tag{3.10}$$

Figure 3.3 shows the pyramid structure of the inverse wavelet transform.

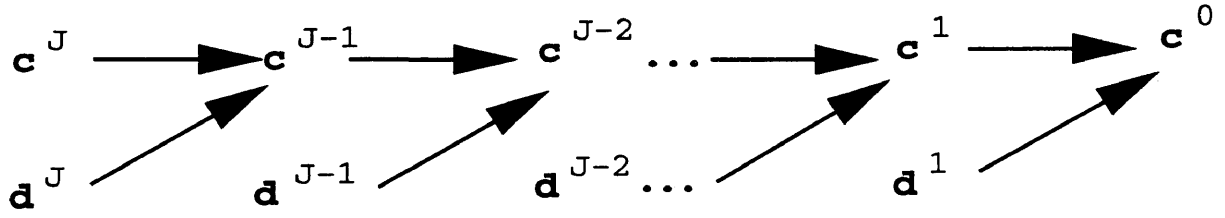

FIG. 3.3. The pyramid structure of the inverse wavelet transform.

3.2.2 Computation of the 1-D discrete wavelet transform

The discrete wavelet transform (DWT) is a fast, linear operation that operates on a data vector whose length is usually an integer power of two, transforming it into a numerically different vector of the same length. The wavelet transform is invertible and orthogonal, the inverse transform is simply the transpose of the transform when it is viewed as a matrix (Press *et al.*, 1992).

The one dimensional wavelet transform can be viewed as the pre-multiplication of the column data vector by the transform matrix W. The rows of the transform

matrix are composed of the wavelet basis functions. The DWT gains efficiency by breaking W into a series of sparse filtering matrices interspersed with permutations of the data.

Equation 3.11 illustrates one level of the DWT as the product of the a filter matrix and the discretized signal ($C's$). The odd rows of the filter matrix are *summing* filters and the even rows are the *differencing* filters. This equation shows the filtering operation for a wavelet transform with four filter taps (such as the Daubechies No.4 transform). The wrap-around of the last two rows is due to the periodic boundary conditions.

$$\begin{bmatrix} h_0 & h_1 & h_2 & h_3 & & & & \\ h_3 & -h_2 & h_1 & -h_0 & & & & \\ & & h_0 & h_1 & h_2 & h_3 & & \\ & & h_3 & -h_2 & h_1 & -h_0 & & \\ & & & & h_0 & h_1 & h_2 & h_3 \\ & & & & h_3 & -h_2 & h_1 & -h_0 \\ h_2 & h_3 & & & & & h_0 & h_1 \\ h_1 & -h_0 & & & & & h_3 & -h_2 \end{bmatrix} \begin{bmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \\ C_7 \end{bmatrix} = \begin{bmatrix} C'_0 \\ D'_0 \\ C'_1 \\ D'_1 \\ C'_2 \\ D'_2 \\ C'_3 \\ D'_3 \end{bmatrix}, \tag{3.11}$$

where

$$h_0 = (1 + \sqrt{3})/4\sqrt{2},$$

$$h_1 = (3 + \sqrt{3})/4\sqrt{2},$$

$$h_2 = (3 - \sqrt{3})/4\sqrt{2},$$

$$h_3 = (1 - \sqrt{3})/4\sqrt{2}$$

for the DAUB4 transform.

Figure 3.4 illustrates that the results are permuted to separate the *summing* data (C's) and the *differencing* data (D's). The *differencing* data are stored while a filter

matrix of half the original size is applied to the *summing* data (C's). The operations at each level divide the data into two halves (half C's and half D's). This process can be terminated at any level. The length of the output is the same as the input and could be used to reconstruct the original data by a inverse wavelet transform. If all transform values are kept, the transform is loss-less. However, if the appropriate wavelet transform is used, the D values are close to zero and can be discarded. This results in a compression of the original data. Since the transform is orthogonal, the inverse wavelet transform matrix is simply the transpose of the wavelet transform matrix.

Assume that the original data are represented by `c[n]`, where `n` is the vector length, `nt` is the length of $C's$ at the highest level, `order` is the length of the filter, `h` is the summing filter and `g` is the differencing filter. The 1-D wavelet transform algorithm can be described as follows:

```
Procedure wt1d(c[], n, nt) {
   initialize temporary variables;
   temp[]=0;
   nhalf=n/2;
   for (j=n; j>nt; j/=2) {
     for (i=0; i<j; i+=2) {
        ic=0;
        for (k=0;k<order;k++) {
           temp[ic]=temp[ic]+h[k]*c[(i+k) mod n];
           temp[ic+nhalf]=temp[ic+half]+g[k]*c[i+k];
        }
        ic=ic+1;
     }
   }
   for (i=0; i<n; i+=1)
     c[i]=temp[i];

}
```

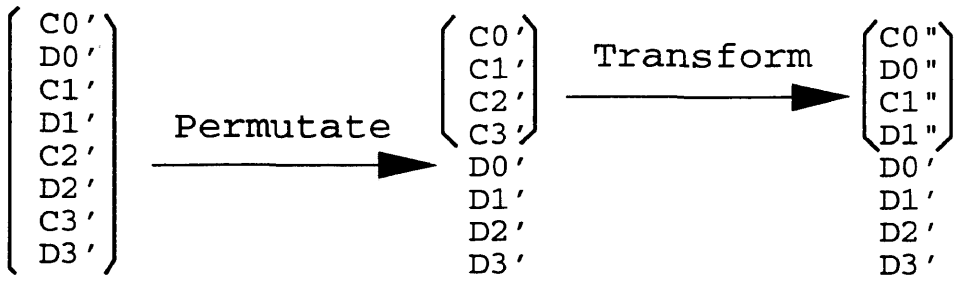

FIG. 3.4. The pyramid structure of the wavelet transform.

3.3 The Multi-Dimensional Wavelet Transform

Section 3.3.1 discusses the construction of n-dimensional scaling functions and mother wavelets. Section 3.32 describes the computation of the n-dimensional Discrete Wavelet Transforms (DWT).

3.3.1 Construction of multi-dimensional wavelets

There are two different ways to design an n-dimensional wavelet transform. One is to generate an n-Dimensional mother wavelet and scaling functions directly by satisfying those equations that were used to create a 1-D mother wavelet and scaling function (e.g., Daubechies, 1992). The other (more desirable) method uses a single 1-D mother wavelet and scaling function to generate an n-dimensional wavelet and scaling functions. In this thesis, we will use the latter approach.

The n-dimensional mother wavelet can be defined as following,

$$\psi_{j;i_1,i_2,\cdots,i_N}(x_1,x_2,\cdots,x_N) = \psi_{j;i_1}(x_1)\psi_{j;i_2}(x_2)\cdots\psi_{j,i_N}(x_N), \tag{3.12}$$

and the n-dimensional scaling functions can be defined similarly,

$$\phi_{j;i_1,i_2,\cdots,i_N}(x_1,x_2,\cdots,x_N) = \phi_{j;i_1}(x_1)\phi_{j;i_2}(x_2)\cdots\phi_{j,i_N}(x_N). \tag{3.13}$$

To represent an n-dimensional space, one still needs scaling functions which span the dimensionally emphasized subspaces:

$$\psi_{j;i_1,i_2,\cdots,i_N}^{l_1,l_2,\cdots,l_N}(x_1,x_2,\cdots,x_N) = \psi_{j;i_1}^{l_1}(x_1)\psi_{j;i_2}^{l_2}(x_2)\cdots\psi_{j,i_N}^{l_N}(x_n), \tag{3.14}$$

where

$$\psi_{j;i_k}^{l_k}(x_k) = \begin{cases} \psi_{j;i_k}(x_k), & \text{if } i_k = 1, \\ \phi_{j;i_k}(x_k), & \text{if } i_k = 0. \end{cases} \tag{3.15}$$

Figure 3.5 illustrates the 2-D DAUB6 scaling functions and mother wavelet.

3.3.2 Computation of multi-dimensional wavelet transforms

A wavelet transform of an n-dimensional array is most easily obtained by transforming the array sequentially on its first index (for all values of its other indices), and then on its second index, third index and so on. Each transformation can be viewed as a multiplication by an orthogonal matrix. By matrix associativity, the result is independent of the order in which the indices were transformed. The situation is similar to the computation of n-dimensional Fast Fourier Transforms.

Figure 3.6 illustrates the n-dimensional wavelet transform procedure. First, apply 1-D wavelet transforms to the rows in the first dimension, then, apply 1-D wavelet transforms along the second dimension of the result, so on, and finally apply 1-D wavelet transforms along the nth dimension to the result of $(n-1)$th dimension transforms.

Assume the data set is `c[]`, the number of dimensions is `ndim`, the lengths along each dimension are stored in `Npoint[ndim]`, the length of data points left in each dimension are stored in `lenlc[ndim]`. The pseudocode to compute the n-D DWT sequentially is as follows:

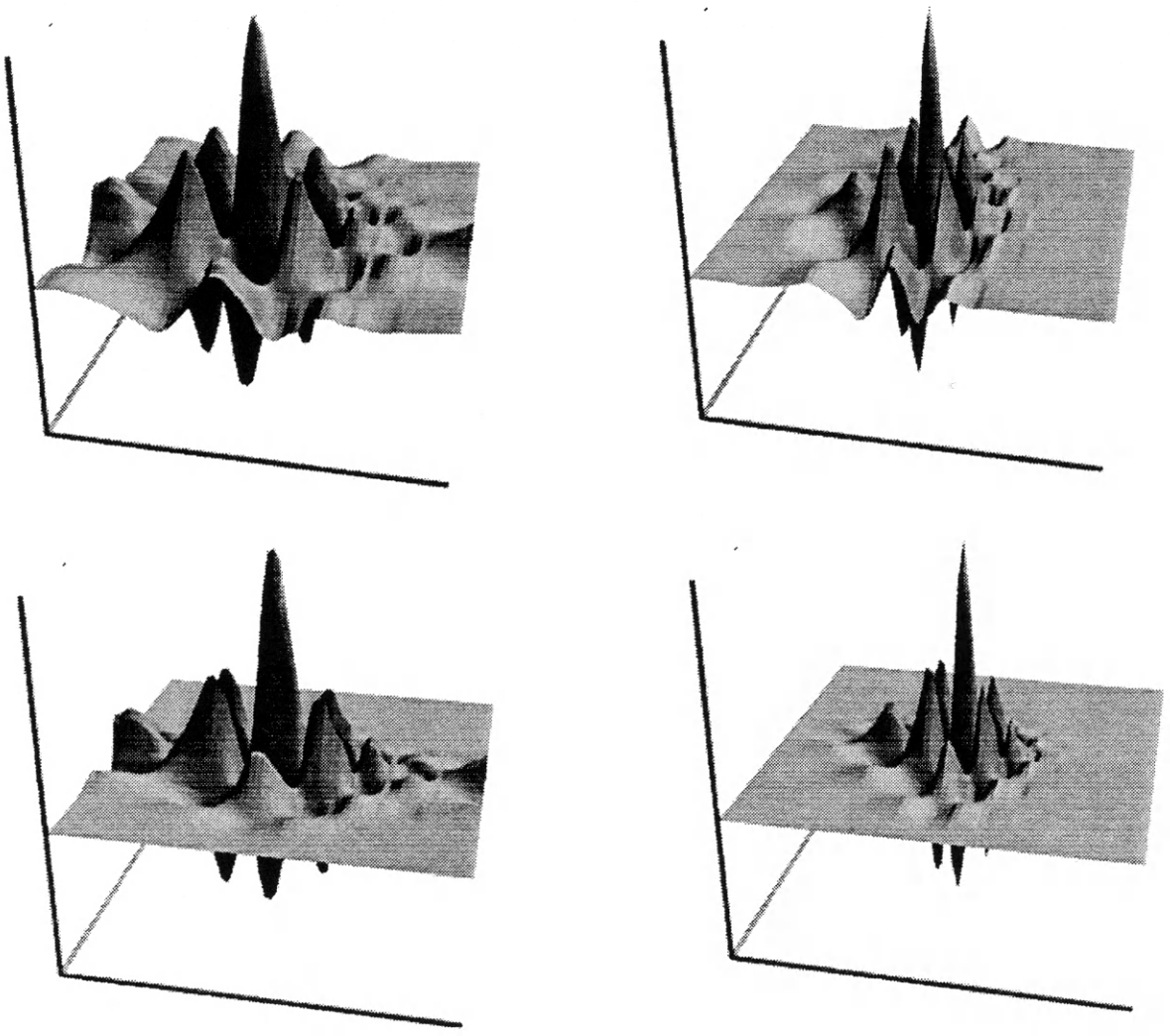

FIG. 3.5. DAUB6 scaling functions and mother wavelet. Upper left: the 2-D scaling function $\phi(x)\phi(y)$, lower left: the 2-D scaling function $\phi(x)\psi(y)$, upper right: the 2-D scaling function $\psi(x)\phi(y)$ and lower right: the 2-D mother wavelet $\psi(x)\psi(y)$.

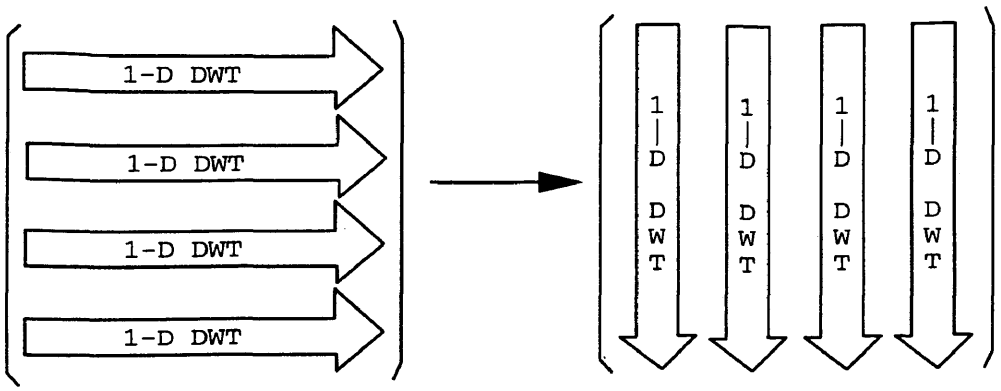

FIG. 3.6. The computation of the n-dimensional wavelet transform.

```
Procedure wtn(c[ntot], ndim, Npoint[ndim], lenlc[ndim]) {
   nprev=1;
   nnew=1;
   for (i=1;i<=ndim;i++) {
      nnew=nprev*Npoint[i];
      for (i1=0;i1<ntot;i1+=nnew) {
         for (i2=1;i2<=nprev;i2+=nprev) {
            for (i3=i1+i2,k=1;k<=n;k+=1,i3+=nprev)
               temp[k]=c[i3];
            wt1d(temp,Npoint[i],lenlc[i]);
            for (i3=i1+i2,k=1;k<=n;i3+=nprev)
               c[i3]=temp[k]; {
         }
      }
      nprev=nnew;
   }
}
```

3.4 State of the Art in Parallel Implementation of Wavelet Transforms

The research on parallel wavelet transforms can be classified into two categories. The first class of methods implement parallel wavelet transforms on special purpose hardware (Kita, 1993; Hoyt & Weschsler, 1992; Parhi & Nishitani, 1993; Sahinoglou & Cabrera, 1991). The parallel algorithm is designed and embedded into the application

specific integrated circuit (ASIC) via very large scale integration (VLSI), which is very costly. The second class of methods implement parallel wavelet transforms in software. I will focus on the software implementation method.

Misra & Prasanna (1992), Misra & Nichols (1994), Misra (1994) discuss the parallel computation of the 2-D Gabor wavelet transform and its implementation on the Connection Machine-2, and on Meshes and Hypercubes separately; Lu (1993) discusses parallel computation of the 2-D Mallat wavelet transform; Pic & Essafi (1993) use a method of convolution before down-sampling for parallelizing wavelet transforms; Holmstrom (1995) presents two new algorithms for fast wavelet transforms implemented on the CM-200 and CM-5; Caulfield (1992) discusses parallel discrete and continuous wavelet transforms; Lega & School (1995) present a parallel algorithm which helps to recognize structure rapidly in a 3-D set of discrete data resulting from numerical experiments, and to study their morphological properties. In the following subsections, some of this work will be described briefly.

3.4.1 Parallelizing 2-D Gabor function based wavelet transform

Misra & Prasanna (1992), Misra & Nichols (1994), Misra (1994) discuss the parallel computation of 2-D Gabor wavelet transforms. The application of the wavelet transform discussed in this work is image processing. The wavelet transform of an $N \times N$ image, based on a family of wavelets with $\nu\mu$ members (ν frequency levels, and μ orientations), can be performed in $O(\nu\mu N)$ time on an $N \times N$ mesh. The papers show that an architecture that provides hypercube-based communication along rows and columns of an array has a much higher bandwidth of communication than the mesh, and that this interconnection scheme optimally exploits the regular pattern of communication required by the task. The transform can be computed optimally in

$O(\nu\mu \log N)$ time on this architecture. Results of implementations on the CM-2 are presented.

3.4.2 Parallelizing the Daubechies wavelet transform

Pic & Essafi (1993) present an algorithm to compute the Daubechies wavelet transform on the Connection Machine CM-2 and the Sympati 2. Both of these machines are SIMD architectures. The application described in their paper is image processing. The CM-2 possessed a large number of processors (8K were used in this paper), so the algorithm used one processor for each pixel of the image. The SYMPATI 2 is an SIMD computer with between 32 and 256 processors. The architecture is a linear array of processors with an internal memory address computation system. Direct access is allowed from one processor to 3 neighboring registers in each direction and 2 neighboring memory units in each direction. The data-mapping method allows the computation of 2-D data transforms as easily as 1-D transforms without conflicts. The article addresses the absolute time performance but does not mention the sequential time, so one can not evaluate speedup values.

Holmstrom (1995) presents an algorithm to compute wavelet transforms on the Connection Machine CM-200 and CM-5. The original data are presented in an array. In order to reduce global communication requirements, the algorithm splits the array into two parts: one with even indices and the other with odd indices, and the filter coefficients are split into even and odd parts too. The data communication is done by a shift of arrays. A nice feature of the algorithm is that it takes advantage of the FORTRAN 90 command of *cshift* to implement the data communication. The article presents the time performance but does not give the sequential algorithm performance, so one can not estimate speedups.

Lega *et al.* (1995) present a parallel algorithm which allows rapid recognization of structures in a 3-D discrete dataset acquired in numerical experiments, and to study their morphological properties. The fast parallel implementation on a Connection Machine CM-200 made the algorithm interesting for other areas in computational physics for morphological comparisons.

The related work on wavelet transforms on massively parallel SIMD machines has been discussed above. The question being addressed here is whether wavelet transforms are suitable for coarse-grained implementation, especially, distributed memory systems such as networks of workstations. This issue will be discussed in the following chapters.

Chapter 4

PARALLEL MULTI-DIMENSIONAL WAVELET TRANSFORMS ON DISTRIBUTED MEMORY MACHINES

In this chapter, parallel algorithms for multi-dimensional wavelet transforms are presented, and several data partitioning and data communication patterns are discussed and evaluated. The performance of the algorithms are estimated on the distributed memory machine model C^3. The efficient algorithms among them, CRBP (Communication Reduced Block Partitioning) algorithm and CRLP (Communication Reduced Layer Partitioning) algorithm are implemented. The results show that these algorithms achieve good speedups. We describe how the algorithms take advantage of system resources such as shared Network File System (NFS), and therefore can efficiently solve larger problems that are too large to fit on a single machine.

4.1 Processing Paradigm

Fine-grained machines have been used to parallelize 1-D and 2-D wavelet transforms (Misra & Prasanna, 1992; Misra & Nichols, 1994; Holmström, 1995; Lu, 1993; Caulfield, 1992; Pic & Essafi, 1993). However, fine-grained machines are sometimes not available due to their high costs. Also, the trend in the high performance machine market seems to be moving away from fine-grained machines and towards using existing workstations as loosely coupled parallel computers. In such a case, a cluster of workstations are connected via a Local Area Network (LAN). Message passing software allows the LAN to function as a loosely-coupled parallel computer with each

workstation operating as a processor in a distributed memory machine.

Distributed processing is of particular interest to companies with exiting LANs who wish to extend the useful life of their current systems. Often, the workstations in the LAN run as separate computers during the day. During evenings and weekends, the LAN is used as a large virtual computer for large computational problems. This distributed processing paradigm is very attractive for economic reasons. Addition of more processors is as simple as adding more workstations to the network. These workstations are usable for other tasks in addition to their use as processors in the virtual machine. This greatly increases the utilization of the machines in the network.

Distributed processing can be applied in two different ways. First, distributed processing can be used to run large problems through the sharing of system resources, such as core memory and disk space. This approach need not provide significant gains in time performance. However, this type of application does allow for the solution of problems that are too large to fit on a single machine. Second, distributed processing can be used to exploit parallelism within the algorithm, thus achieving a speedup over sequential implementations. Due to the high communication cost associated with distributed processing, the latter might be less desirable as compared to the first if the application program requires a large amount of communication. In this research on the parallel implementation of wavelet transforms, the results show that both reduction of core memory and runtime costs can be obtained if appropriate algorithms are used.

4.2 The Parallel Computation of n-Dimensional DWT

Section 3.3 described the sequential algorithm to compute the n-dimensional wavelet transform. Based on wavelet theory, we know that we can not parallelize

the n-dimensional wavelet transforms such that one processor works on the first dimension and another works on another dimension. Another possibility is to assign all the processors to work on the same dimension in parallel. This possibility can be manifested in many options. These options will be investigated in the following sections.

4.2.1 The traditional data partitioning approach

The traditional data partitioning approach can be used to partition the data and computations of the n-dimensional wavelet transform. Assume p is the number of processors, in this scheme, the data are divided into p blocks along the first dimension. Once 1-D wavelet transforms along the first dimension are computed, the data are redivided into p blocks along the second dimension; once 1-D DWTs along the second dimension are carried out, the data are redivided into p blocks along the third dimension, and so on. Figure 4.1 illustrates the traditional data partitioning in the 2-D case. Figure 4.2 illustrates the traditional data partitioning in the 3-D case.

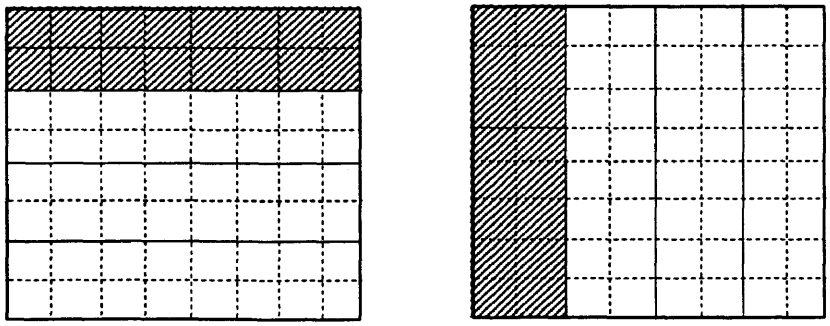

FIG. 4.1. The traditional data partitioning distributes a two dimensional array of size 8×8 onto four processors. (a) the data partitioning along the first dimension; (b) the data partitioning along the second dimension.

The algorithm can be described as follows:

```
for (dim=1; dim<ndim; dim++) {
   in parallel do
   {
     distribute the data along the current dimension;
     for (every line in this dimension in current processor) {
       copy the data to temporary variable;
       1-D wavelet transform on this data;
     }
     gather the data in the root processor;
   }
}
```

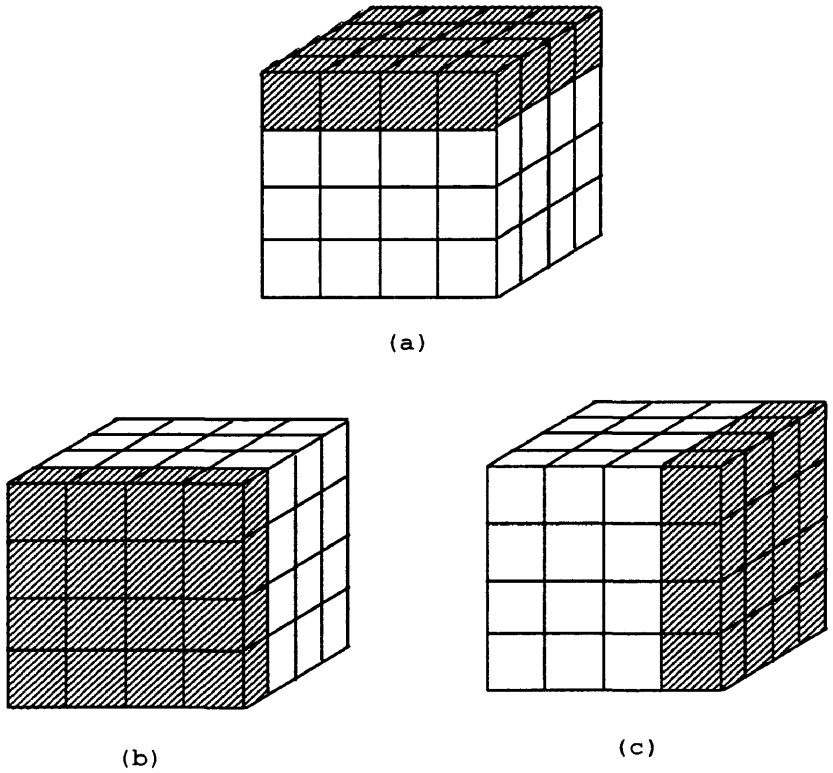

FIG. 4.2. The traditional partitioning distributes a three dimensional array of size $4 \times 4 \times 4$ onto four processors. (a) the data partitioning along the first dimension; (b) the data partitioning along the second dimension; (c) the data partitioning along the third dimension.

Notice that the redistribution of data requires data communication that is costly among the processors, especially when an Ethernet connected network of workstation is used. To reduce the data communication cost, the block partitioning is proposed as follows.

4.2.2 The block data partitioning approach

The block partitioning divides the data set along each dimension, to get p blocks of the same size, where p is the number of processors. Figure 4.3 illustrates the block partitioning in the 2-D case with four processors. Figure 4.4 illustrates block partitioning in the 3-D case, with eight processors.

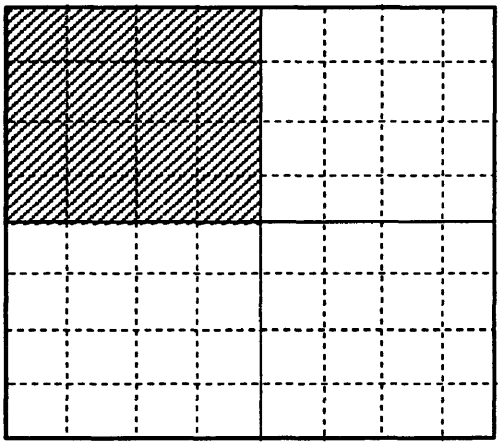

FIG. 4.3. 2-D block partitioning.

If block partitioning is used, the parallel algorithm to compute the n-dimensional transform can be described as follows:

```
Distribute the blocks of data to the processors
for (dim=1; dim< ndim; dim++) {
   in parallel do
   {
   if (level < nlevel) {
```

```
        communicate boundary data between neighboring processors;
        for (every line in this dimension in this block) {
          put the data to temporary variable;
          wavelet transform for one level ;
          level increases by 1;
        }
}
write the result to the data file.
```

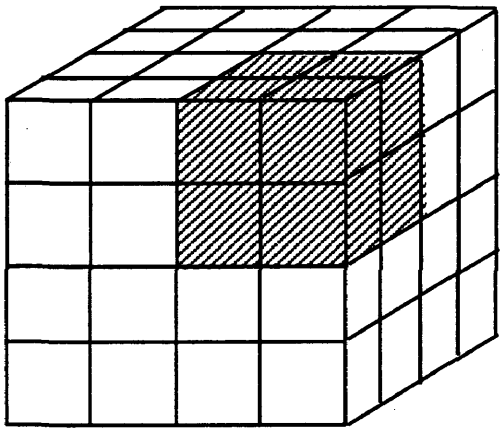

FIG. 4.4. 3-D block partitioning.

Assume the data set is 16×16, and the number of processors is 4, the levels of the transform along each dimension $nlevel = 2$, the algorithm to compute the DAUB4 transform can be described as follows.

- Distribution of the data is illustrated in Figure 4.5.
- To compute the first level of the DWT along the first dimension, the communication of boundary data proceeds as follows: PE1 gets PE2's last two columns and PE2 gets the first two columns of PE1; PE3 and PE4 do the same thing. Figure 4.6 illustrates the result of this process.
- Apply the first level of the wavelet transform to every line of the first dimension,

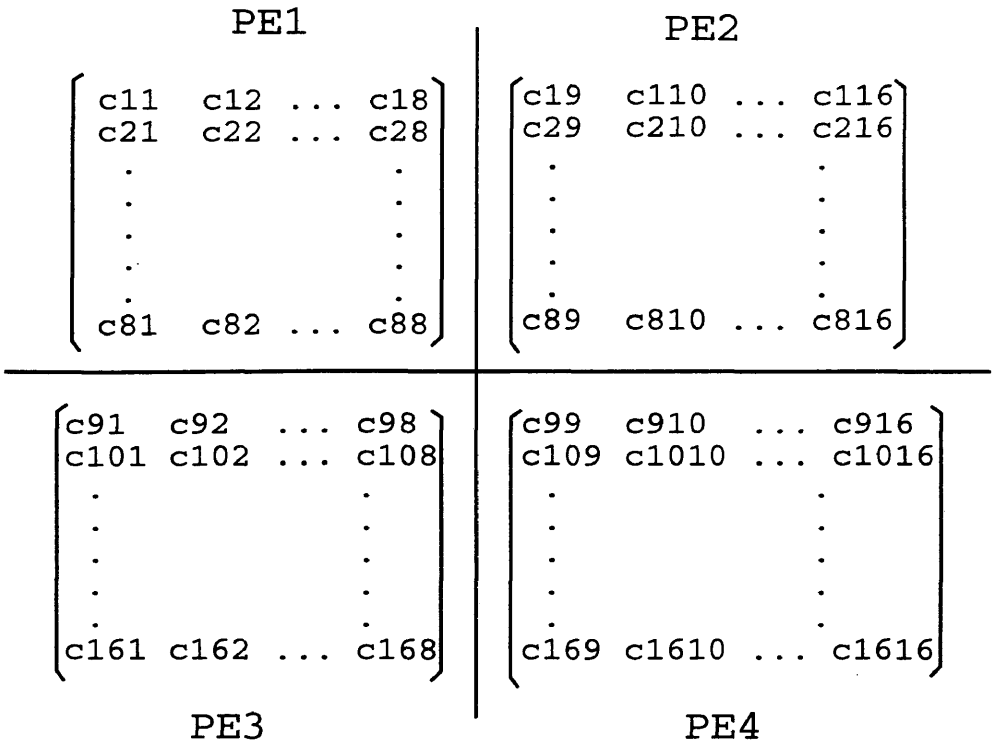

FIG. 4.5. Block partitioning with 4 processors, and 16 × 16 data set.

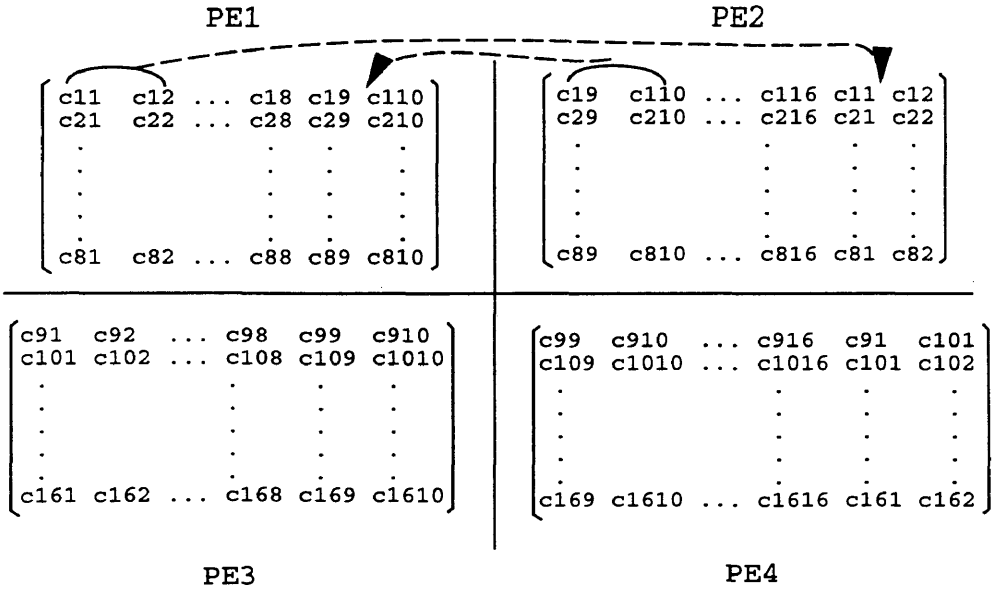

FIG. 4.6. Block partitioning of the first level along the first dimension and data communication with 4 processors, and 16 × 16 data set.

no wrap-around of boundary values is needed, the results are illustrated in figure 4.7.

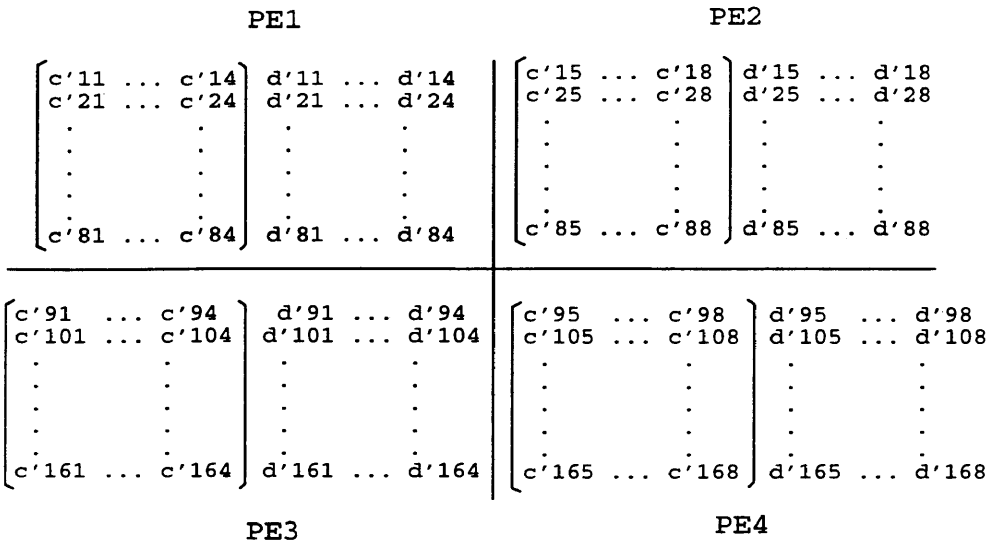

FIG. 4.7. The data communication for the result of the first level of the first dimension DWT with 4 processors, and 16 × 16 data set.

- To compute the second level of the computation along the first dimension, new boundary data need to be communicated as illustrated in figure 4.8.
- Apply the second level of the wavelet transform to each line of the first dimension, no wrap-around of boundary data is needed and the results are illustrated in figure 4.9.
- Once the computation along the first dimension is carried out, apply the parallel wavelet transform along the second dimension to the result of the first dimension. The data communication of the first level along the second dimension required for the computation is illustrated in Figure 4.10.
- Apply the first level of the wavelet transform along the second dimension, and

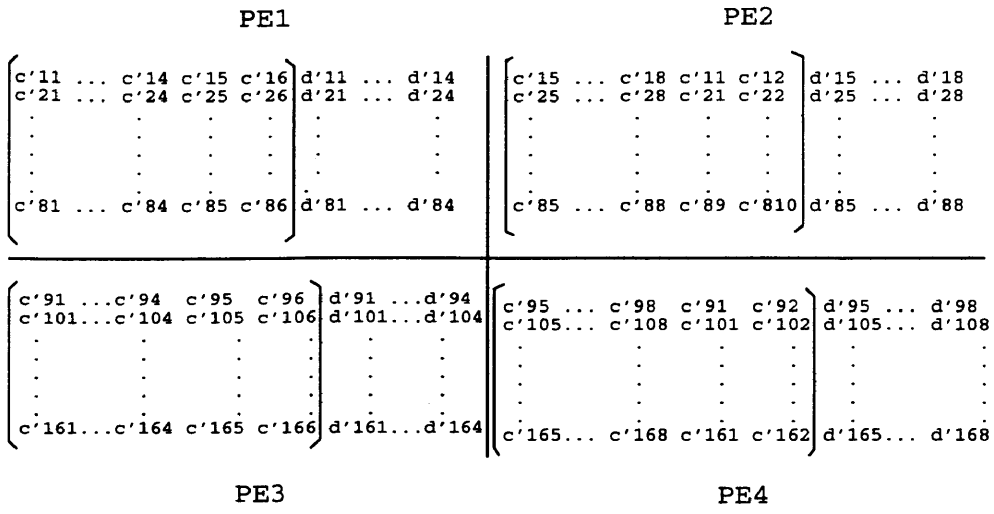

FIG. 4.8. The data communication for the second level of the first dimension with 4 processors, and 16 × 16 data set.

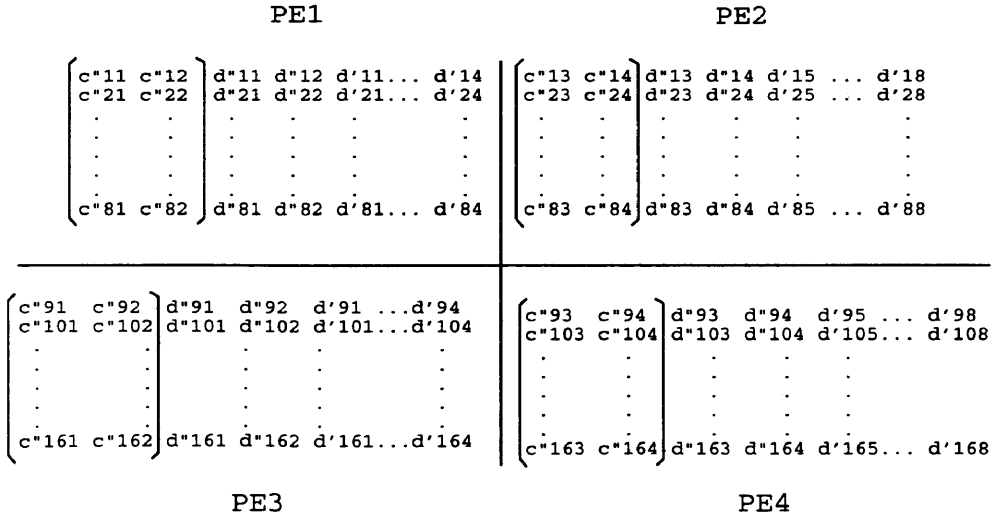

FIG. 4.9. Block partitioning, the result of the second level of first dimension DWT with 4 processors, and 16 by 16 data set.

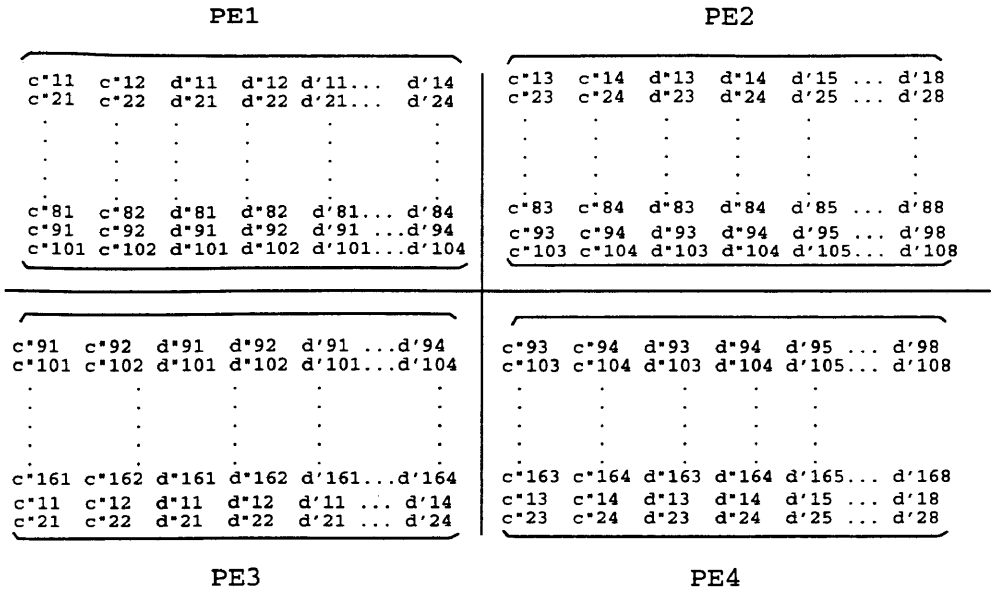

FIG. 4.10. Block partitioning of the first level of the second dimension data communication with 4 processors, and 16 × 16 data set.

then do the data communication followed by the second level of the wavelet transform. Repeat this procedure; once the $(n-1)th$ dimension is done, apply the parallel wavelet transform to the result of the $(n-1)th$ dimension along the nth dimension.

The block partitioning method reduces the data communication among the processors. But data communication still remains costly, especially in environments of clusters of workstations connected by Ethernet. Also, if one is using a transform with a higher number of filter taps, more columns of data need to be exchanged between PEs. How can one minimize the data communication between neighboring processors? An efficient alternative is to view each data block as an independent dataset and apply the wavelet transform on each processor separately, then put the result together. When applying the inverse transform, partition the data in the same way as the forward transform. The reader must be cautioned that this approach does

alter the DWT algorithm. However, in cases where we are considering coarse grained parallelism and the data contains limited high frequency information near the partition edges, the results of the transform are accurate enough for most purposes. The experiments in Section 4.3 show that this method is efficient and it greatly reduces the runtime cost. The next section presents this Communication Reduced Block Partitioning (CRBP) method and the related algorithm.

4.2.3 CRBP and the related DWT algorithm

The CRBP partitioning is the same as the block partitioning except that the CRBP needs no communication among neighboring processors. Figure 4.4 illustrates the 2-D block partitioning for 8 processors. Figure 4.11 illustrates the 3-D block partitioning for 2, 4, and 6 processors.

The n-D DWT algorithm using CRBP can be described as following:

Step 1: `Distribute the data to the` n `processors (via NFS in a NOW);`
Step 2: `Computation;`

```
for (dim=0; dim< ndim; dim++) {
  for (each line in this dimension) {
    copy the data in this line to a temporary variable;
    apply 1-D wavelet transform on this data;
  }
}
```

Step3: `write the results to data files (via NFS).`

When the data are distributed to the n processors via NFS in a NOW, all the processors are reading portions of the datafile from one server workstation. All the other workstations are opening this file through NFS, and reading appropriate sections. The data are then transported over the network to these machines.

Block partitioning divides the data along two or more dimensions, if the data are divided only along the last dimension, the data distribution will be more effi-

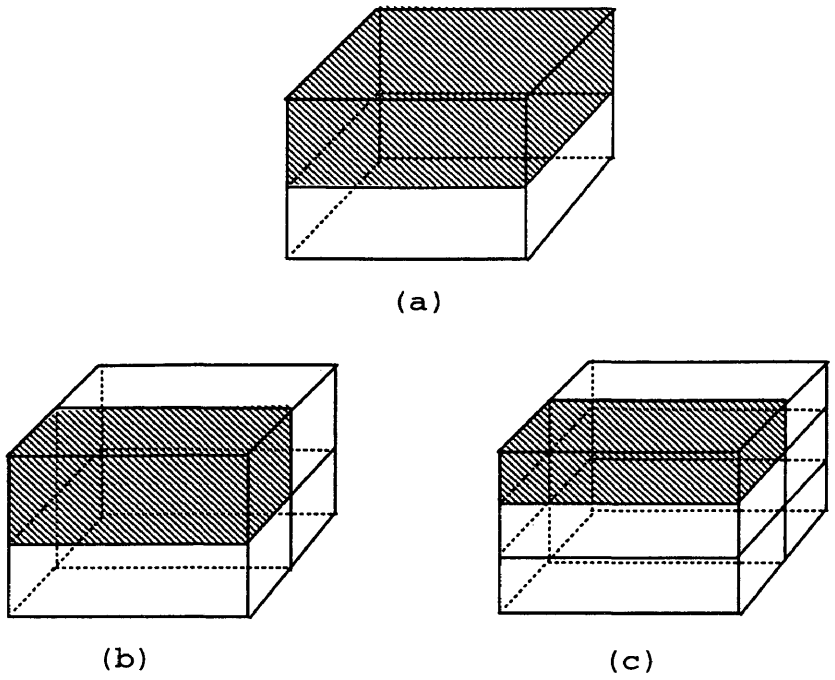

FIG. 4.11. The block partitioning in the 3-D case for (a) two processors; (b) four processors; (c) six processors.

cient, because the data in every layer is consecutive. In the next subsection, the CRLP (Communication Reduced Layer Partitioning) and the related parallel DWT algorithm will be presented.

4.2.4 CRLP and the related DWT algorithm

Assume p processors are used, the layer partitioning divides the data along the last dimension. Figure 4.12 illustrates the layer partitioning in 2-D and 3-D case. The number of processors is four. The figure also shows how a 3-D data set is partitioned among 6 PEs.

The parallel DWT algorithm with CRLP is the same as the parallel algorithm with CRBP except the part related to data distribution. Assume $ntot$, the total length of the data equals $N_1 \times N_2 \times ... \times N_{ndim}$. Details of the CRLP data distribution step

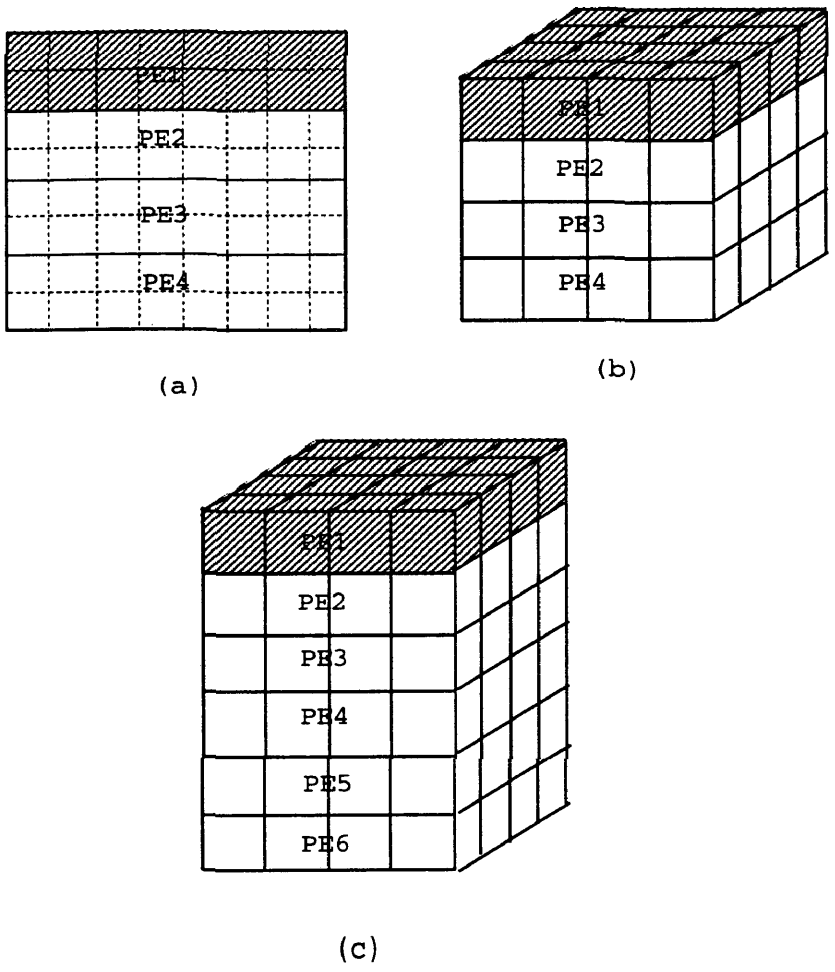

FIG. 4.12. The layer partitioning distributes the data onto processors. (a) two dimensional four processors case; (b) three dimensional four processors case; (c) three dimensional six processors case.

can be described as follows:

```
    To distribute the data to the n processors via NFS:
Sub-step 1: each processor i opens the data file;
Sub-step 2: move each file pointer to position:  i*ntot/p;
Sub-step 3: read data block (from the current position, the length of
            each data block is:  ntot/p) to memory;
```

4.3 Predicted Performance

As specified in Chapter 2, the C^3 model is designed for tightly-coupled coarse-grained machines. The Ethernet connected network of workstations can be looked upon as a loosely-coupled coarse-grained machine. In this work, we use a cluster of SGI workstations to model a virtual parallel machine. The C^3 model is not very suitable for the SGI cluster, but there are no other models that are more appropriate. We therefore use it to analyze our algorithm.

4.3.1 The machine parameters

The machine parameters used in the C^3 model are listed as follows. The metrics are used to compute the communication and computation cost of an algorithm.

In the SGI cluster, the number of the processors is p, where $1 <= p <= 8$.

The latency of the communication network is represented by h. The Ethernet is a 10Mb/s local area network standard (Johnson, 1996). In the SGI cluster, each computer was attached in the same way to a single piece of cable. The cable snaked through the building, with all computers hooked onto it. When one computer sends out a message, all the computers in the network can receive the message. This network architecture is called a bus topology (figure 4.13).

A big disadvantage of bus topologies is that only one message packet can be successfully transmitted at a time. Because all transmissions share the same medium,

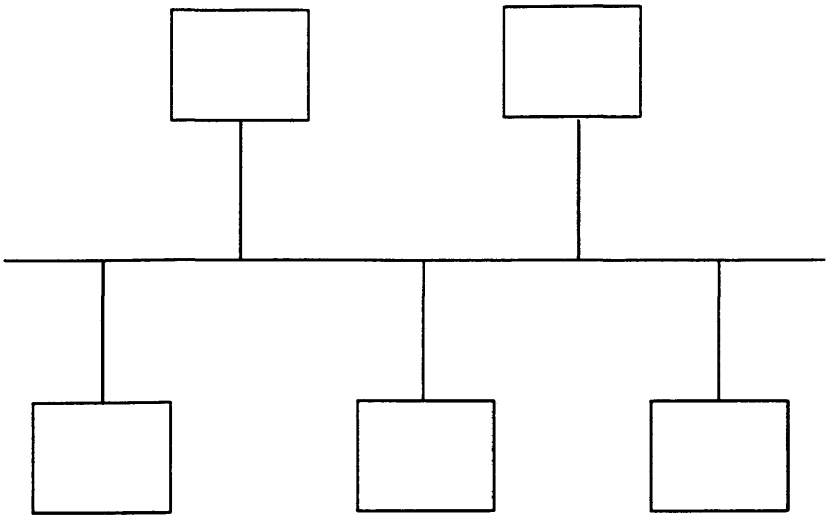

FIG. 4.13. Architecture of the bus topology.

simultaneous transmissions cause collisions. Once a collision occurs, the packets must be resent. So, while there is no other traffic, the latency of the communication network h can be defined as 1, otherwise it is hard to determine. No intensive statistics were performed to determine the value of h under various traffic conditions.

The bisection width of this communication network b is 1. The bisection width is defined as the minimum number of links that have to be cut in order to disconnect the network into two halves with identical number of processors. The set-up cost for a message in the Ethernet connected network is $s = 1500ms$ (micro-seconds) (Foster, 1994). The maximum length of a packet l in the Ethernet connected network is 1500bits. If one discounts the header in the packet, the length of data in one packet is about 1372bits. In the following subsection, the algorithm performance will be presented.

4.3.2 Algorithm analysis

In order to analyze the time performance of the algorithm, we can divide the CRLP algorithm to three super-steps: data distribution, computation and data congregation.

In the data distribution step, there is no computation, so the overall cost of this step is the overall communication cost, $\max_{0\leq i\leq p-1}\{S_i + R_i\} + C_l + C_p$, where S_i is the sending cost for processor i, R_i is the receiving cost for processor i, C_l is the congestion cost over links, and C_p is the congestion cost at processors. During the data distribution, the root processor ($i = 0$) has the maximum cost, because the root processor sends the data to the other $p-1$ processors, and the $p-1$ processors only receive the data from the root processor. In the case when the root processor sends to one processor at a time (blocking send and non-blocking receive protocol), $S = s+\lceil L/l\rceil\cdot h$, where $s = 1500ms$, $L = ntot/p$, $l = 1372$ bits $= 184$ bytes, and $h = 1$. Converting the set-up cost s to units, the number of units corresponding to one set-up cost is approximately 80. Therefore, for the root processor to send to $p-1$ processors, the cost is $S_0 = (p-1)\cdot S = (p-1)\cdot(80+\lceil 4\cdot ntot/(184p)\rceil)$. For the root processor, the receiving cost $R_0 = 0$. The congestion cost over the link is: $C_l = L_a\cdot\lceil cong/b\rceil$, where L_a, the average number of packets routed between processors is $L/l = 4\cdot ntot/(pl)$, and $cong$, the total number of processor pairs communicating, is $p-1$, and the bisection width of the network is $b = 1$. Therefore $C_l = L_a\cdot(p-1) = 4\cdot ntot\cdot(p-1)/(184p)$, this is the worst case. To adjust the congestion time, the C_l can be multiplied by a coefficient a, where $a \geq 0$ and $a \leq 1$. There is no congestion on the processors, so $C_p = 0$. The cost for the data distribution step is $S_0+C_l = 80(p-1)+4(1+a)(p-1)\cdot ntot/(184p)$. In the computation step, the total cost is: $\max_{0\leq i\leq p-1}\lceil t_i/l\rceil$, where t_i bytes are accessed by processor P_i in one step, and $t_i = 8\cdot order\cdot ntot/p$, where $order$ is the length of

the filter, $order$=4 for DAUB4, the transform used in the research. The data block on each processor is of the same size, so the computation for each processor is the same. The computation cost should be $32 \cdot ntot/(184p)$. Note that this assumes that all machines run at the same speed.

In the data gather step, $(p-1)$ processors send the data back to the root processor, the cost is $4(p-1) \cdot ntot/(184p)$. Therefore the total cost for this algorithm is $80(p-1) + (4(2+a)(p-1) \cdot ntot + 32 \cdot ntot)/(184p)$.

Figure 4.14 shows the predicted performance of the CRLP algorithm when the data set size is $64 \times 64 \times 64$.

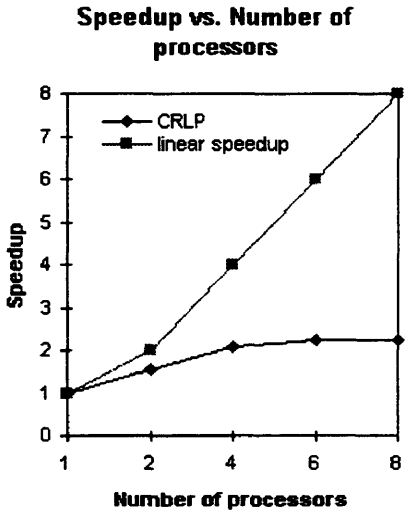

FIG. 4.14. The predicted speedup for CRLP, with data size of $64 \times 64 \times 64$.

4.4 Numerical Results

The numerical results obtained for implementations of the CRBP and CRLP algorithms on a network of SGI workstations are given below. The workstations

were networked using 10Mbits/s Ethernet. The message passing software used was the MPICH (Gropp *et al.*, 1996) implementation of Message Passing Interface (MPI) (Pacheco, 1997). Mapping the data to the workstations is done by reading the data files via NFS from a common server that stores the data file. This method has been found to be faster than transmission of the data via `MPI_send` and `MPI_recv` (receive) commands.

4.4.1 The assessment of the partitioning with reduced communication

As was pointed out in Section 4.2.2, the CRBP and CRLP approaches are altering the original n-dimensional DWT algorithm. A natural question to ask is whether the new partitioning results in the correct computation of DWTs. Theoretically speaking, one can definitely construct a counter-example to show where such an approach fails. However, this approach turns out to be an efficient alternative for algorithms on an architecture with costly communications, if the original data do not contain very high frequencies and the DWT is computed on a few PEs in a coarse grained way. Fortunately most physical signals, including seismic data and image data, satisfy this condition very well. To numerically test this idea, the DWT was applied to the task of the image compression. The example uses a 256×256 gray-scale image. It was distributed to four workstations by block partitioning, and the algorithm used is CRBP. Figure 4.15 (a) shows the original image, (b) shows the reconstructed image where the DAUB4 wavelet was used with the compression threshold chosen as 0.005, and the compression ratio is 12.5 : 1, and (c) shows the reconstructed image where the DAUB16 wavelet was used, the same compression threshold as (b), and the compression ratio is 10 : 1.

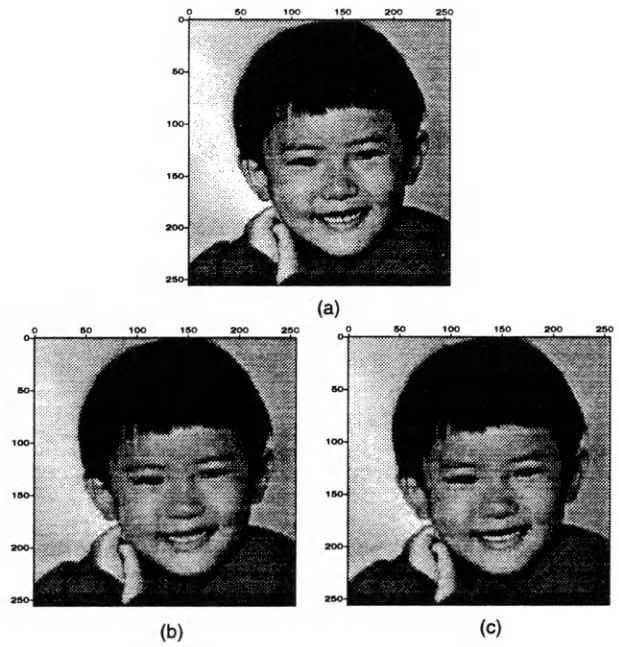

FIG. 4.15. Connie Y. Meng's picture. (a) Original, (b) reconstructed by DAUB4, (c) reconstructed by DAUB16 (using the CRBP algorithm).

These three pictures show that for the compression of this kind of image, the CRBP is reliable. But if there are high frequencies around the boundary where the data are split between processors, the quality of the reconstruction will not be as good as these.

In the next subsection, the speedups obtained by the parallel implementation will be presented.

4.4.2 Timing results

The Daubechies wavelet transforms were implemented in this research on the SGI cluster. The DAUB4 transform was used in all the computations reported here. The MPI (Message Passing Interface) is used to compute the n-dimensional DWT. MPI is not a new programming language, it is simply a library of definitions and

functions that can be used in C (and Fortran) programs (Pacheco, 1997). Every MPI program must contain the preprocessor directive `#include "mpi.h"`. MPI uses a consistent scheme for MPI-defined identifiers. All the identifiers begin with the string `MPI_`. Before any other functions can be called, `MPI_Init` must be called, and it should only be called once. After a program has finished using the MPI library, it must call `MPI_Finalize`. This cleans up any unfinished business left by MPI, e.g., it frees memory allocated by MPI. The MPI function `MPI_Barrier(MPI_COMM_WORLD)` is used to synchronize all the processes and `MPI_Wtime()` is used for timing.

A skeleton of the MPI code structure used in this program is illustrated as follows:

```
Main(int agrc, char* agrv[]) {
    variables declaration;
    MPI_Init(&argc, &argv);
    /* Find out how many processes are involved
    in the execution of a program */
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    /* Find out the rank of the current process */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_io_rank);
    allocate memory;
    /*Block the calling process until all processes
    have entered the function */
    MPI_Barrier(MPI_COMM_WORLD);
    /*Begin timing*/
    start=MPI_Wtime();
    read the data from data files;
    computation;
    save the results to data files;
    MPI_Barrier(MPI_COMM_WORLD);
    /* End timing */
    finish=MPI_Wtime();
    /* Compute the time used */
    cputime=finish-start;
    MPI_Finalize();
}
```

The transform computations were run on different numbers of workstations in

Table 4.1. The machine parameters of the machines in the SGI cluster.

	Machine Types					
Parameters	A	B	C	D	E	F
OS(IRIX)	5.3	5.3	5.3	5.3	5.3	5.2
Clock speed(Mhz)	200	175	200	100	132	100
Processor type(MIPS)	R4400	R10100	R4600	R4600	R4600	R4600
Data cache(KB)	16	32	16	16	16	16
Instruction cache(KB)	16	32	16	16	16	16
Memory(MB)	64	64	32	32	32	16
Disk space(GB)	2	2	1	1	1	0.5

order to compare speedups obtained in each case. Results are reported for runs on 1, 2, 4 and 8 workstations. The parameters of the machines are listed in Table 4.1, where A, B, C, D, E and F stand for different machine types available on the network.

When one or two workstations are used, the machines used are of type A. When four workstations are used, three of them are of type A, and one is of type B. When six workstations are used, the machines used are of type A(3), B(1), C(1) and E(1). When eight workstations are used, three are of type A, and the rest of them are of type B, C, D, E and F.

The CRBP and CRLP algorithms are tested separately. Figure 4.16 compares the speedups of CRBP and CRLP against linear speedup, and it shows:

- When 2, 4 and 6 processor are used, the speedup of CRLP is close to linear; when 8 processors are used, the speedup is even worse than the 4 processors case. This is because, firstly, the addition of the two weak machines of type D and F hurts the overall performance. Remember that at the beginning and end of timing, the MPI function *MPI_Barrier()* is used to synchronize all of the processors. If even one machine is slow, all of the others must wait for the slow one. Secondly, the increasing data traffic (during the data distribution

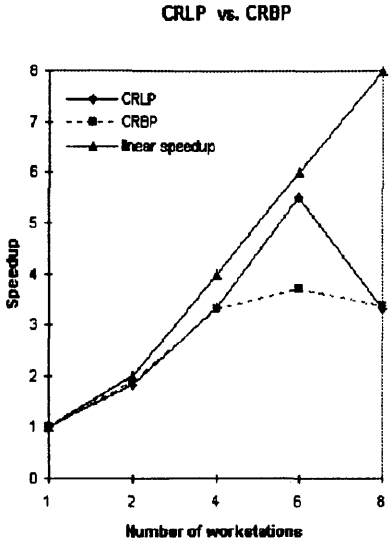

FIG. 4.16. The speedups in CRBP and CRLP, with data of size of 128 × 128 × 384.

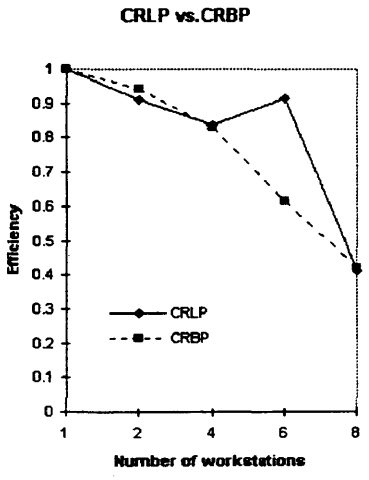

FIG. 4.17. The efficiencies in CRBP and CRLP, with data of size 128 × 128 × 384.

and data congregation super-steps) increases the possibility of message collisions when more processors are used.

- The speedup of the CRLP approach is better than that of CRBP. This is because in the CRLP scheme, each data layer consists of consecutive positions in the original data set, the file pointer only needs to jump to the appropriate position at the beginning of data communication. In the CRBP scheme, each data block consists of non-consecutive positions, the file pointer needs to jump many times to read the data. The jumping of the pointers via NFS consumes time. Figure 4.17 compares the machine efficiencies of CRLP with those of CRBP, and it shows that CRLP is better than CRBP too.

Figure 4.18 and Figure 4.19 compare the processing time and speedups of CRBP and CRLP for different numbers of workstations. The size of the data set is 256 × 256 × 256. From these two figures, we can see that when 8 processors are used, the runtime is shorter than when 4 processors are used, and the speedup is better than that when 4 processors are used. This is because when the problem size goes up, the parallel algorithm is more efficient, and the communication costs constitute a smaller fraction of the overall time.

Figure 4.20 and Figure 4.21 compare the processing time and speedups of CRBP against CRLP for different numbers of workstations. The size of the dataset is 32 × 32 × 32. These two figures show that when the problem size is small, the parallel algorithm efficiency is low as the system communication cost constitutes a higher fraction of the overall time.

Table 4.2 shows the runtime of the CRLP algorithm on different data sizes and different numbers of processors. From this table, we can see that:

- In the fifth column (the number of processors is 6), a runtime is reported only

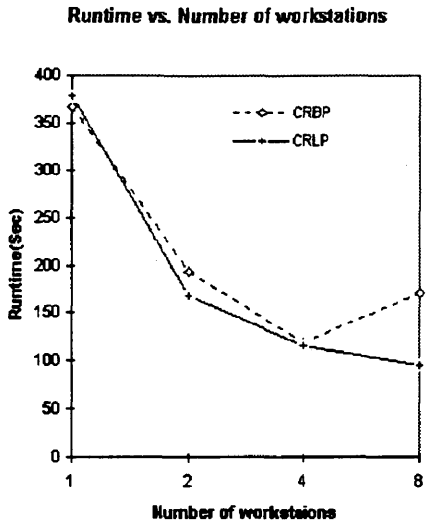

FIG. 4.18. The run time in seconds on data of size $256 \times 256 \times 256$.

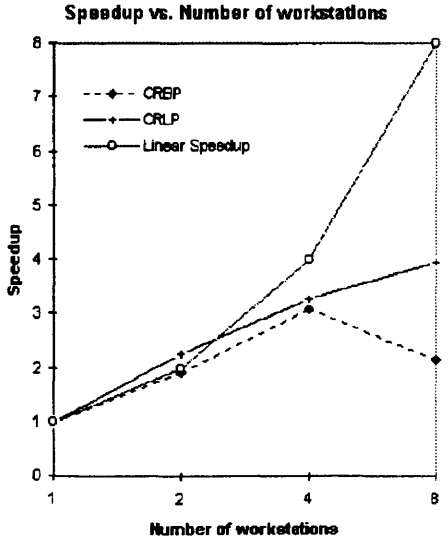

FIG. 4.19. The speedups for the CRBP and CRLP cases, with data of size $256 \times 256 \times 256$.

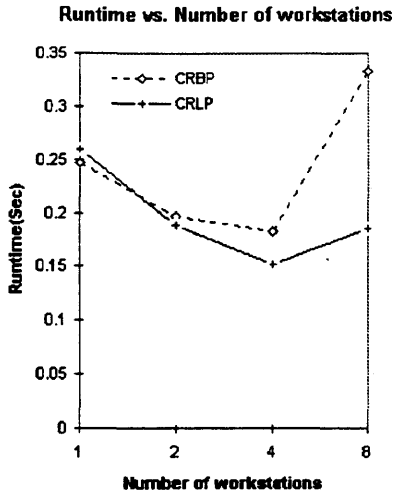

FIG. 4.20. The processing times in seconds, with data of size $32 \times 32 \times 32$.

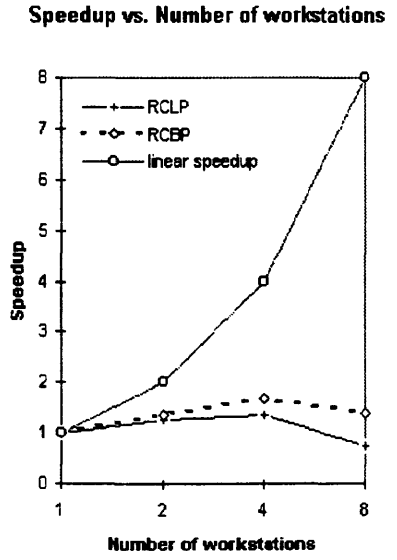

FIG. 4.21. The speedups, with data of size $32 \times 32 \times 32$.

for one data size (row 6), and the others are empty. Refer back to figure 4.12, for the 6 processors case, division is done along the second and the third dimension, and the length of third dimension is divided by 3, so the data length along this dimension needs to be a multiple of 6.

- When the problem size is $256 \times 256 \times 512$ (the data type is float), the data requires 128MB. However, the largest core memory is only 64MB, so if static allocation is used, the data size is too large to fit in memory on a single machine. When dynamic allocation is used, the data would fit in virtual memory, but this can cause excessive swapping, resulting in degraded performance. Hence, this problem size (and larger problems) cannot be run on a single workstation. Similarly, a problem size of $256 \times 256 \times 1024$ cannot be run on two workstations either.

- Notice that when the problem sizes are less than $128 \times 128 \times 384$ (24MB), the time performance when 8 processors are used is even worse than when 4 processors are used. The explanation is the same as that provided for Figure 4.16. The time performances get better when the problem sizes are larger than $256 \times 256 \times 256$, the explanation is the same as the explanation provided for Figure 4.19.

Table 4.3 shows the runtimes of the CRBP algorithm on different numbers of processors and different data sizes. Notice that the time performances for the 8 processors case are worse than for the 4 processors case for most problem sizes. Besides the two reasons given for Figure 4.16, there is an additional reason. The data blocks assigned to each processor in the CRBP scheme do not come from consecutive locations in the original data. Therefore, the larger the number of processors used, the

Table 4.2. The runtimes of the CRLP algorithm run on different number of processors and different data sizes.

	Number of Processors				
Problem Size	1	2	4	6	8
32×32×32	0.259	0.188	0.152		0.185
64×64×64	2.033	1.097	0.582		0.547
128×128×128	21.621	18.591	10.667		12.448
128×128×384	67.406	37.020	20.162	12.267	20.338
256×256×256	378.700	167.943	115.858		95.847
256×256×512		510.106	239.635		188.280
256×256×1024			732.652		566.500

Table 4.3. The runtimes of the CRBP implementation on different numbers of processors and different data sizes.

	Number of Processors				
Problem Size	1	2	4	6	8
32×32×32	0.247	0.197	0.182		0.334
64×64×64	1.701	1.044	0.678		1.547
128×128×128	20.958	10.552	4.546		8.045
128×128×384	62.324	33.140	18.776	16.793	18.470
256×256×256	366.712	193.223	118.706		171.469
256×256×512		509.315	247.391		326.919
256×256×1024			733.121		857.336

smaller the consecutive data block. This implies that the file pointers need to be moved more often and this results in worse performance.

4.5 Conclusions

This chapter presented parallel algorithms to compute wavelet transforms. Data partitioning and data communication patterns are described in detail. The CRLP and CRBP parallel algorithms are implemented. The results show near linear speedups for the mid-sized dataset on distributed memory machines (represented by a network of workstations). In addition to this, distributed memory machines can run larger

problems (524288KB) that are too large to fit on a single machine.

Chapter 5

PARALLEL MULTI-DIMENSIONAL WAVELET TRANSFORMS ON SHARED MEMORY MACHINES

In this chapter, we will describe appropriate parallel algorithms for n-dimensional wavelet transforms on shared memory machines, then estimate the algorithm performance on the QSM model and present the runtime performance.

5.1 Machine Architecture

This research used the SGI Power Challenge to implement parallel n-dimensional DWT algorithms. An SGI Power Challenge is a shared memory multiprocessor architecture based on the MIPS superscalar RISC R8000 chip. All processors share a global memory address space. The processors communicate via a fast shared-bus interconnect, and the bus has a bandwidth of 1.2GB per second. Each processor has a local 4MB cache, and any processor can address any memory location at the same speed. Figure 5.1 illustrates the SGI Power Challenge configuration.

5.2 Parallel Computation of Multi-dimensional Wavelet Transforms

This research compares two approaches to compute multi-dimensional wavelet transforms on shared memory machines: one is called homogeneous parallelism, and the other is called heterogeneous parallelism. Here homogeneous parallelism means that all the software threads execute the same code on different data. Running a loop

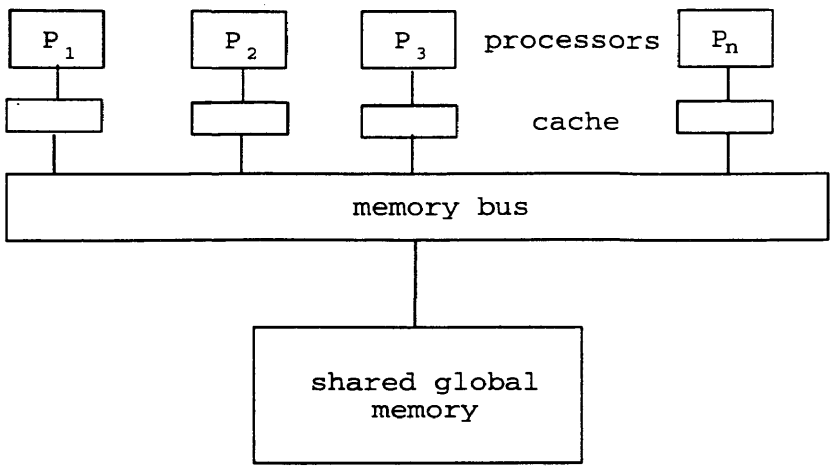

FIG. 5.1. The SGI Power Challenge configuration.

in parallel belongs to the homogeneous parallelism class. Heterogeneous parallelism means the code in each thread of execution can be different.

5.2.1 Homogeneous parallelism approach to computing DWTs

In the homogeneous parallelism approach, the traditional data partitioning scheme is used. In this scheme, the data are divided into p blocks along the first dimension, where p is the number of processors. Once a 1-D DWT is computed along this dimension, the data are divided into p blocks along the second dimension, and so on. Figure 4.1 and 4.2 illustrate the the traditional partitioning scheme.

The homogeneous parallelism algorithm to compute n-dimensional wavelet transforms can be described as follows:

Assume the data set is `cd[]`, the number of dimensions is `ndim`, the lengths along the different dimensions are stored in `Npoint[ndim]`, the lengths of data left in each dimension are stored in `lenlc[ndim]`. `NP` is the number of processors, and the current processor identifier is `my_id`, `temp[NP][lenlc[ndim]]` is a temporary array to store a vector of data.

```
Step 1:
   Read the data from the data file to the shared memory;
Step 2:
   for (i=1; i<=ndim; i++) {
     #pragma parallel shared(cd, temp) local(i) byvalue(lenlc)
     {
     if (i==ndim) then outloop=ndim-1 else outloop=ndim;
     #pragma pfor iterate (j=0; Npoint[outloop]; 1)
     for (j=0; j<Npoint[outloop]; j++) {
       my_id=mpc_my_threadnum();
       read one row of data along current dimension
       and save it to a temporary array in the current processor
       compute a 1-D DWT on the temporary array;
     }
     }
   }
Step3:

   Write the result to a file.
```

5.2.2 Heterogeneous parallelism approach to computing DWTs

In the heterogeneous parallelism approach to compute DWTs, the CRBP scheme is used (note that the CRLP method could also have been used). We define a function to implement the sequential algorithm to compute the n-dimensional wavelet transform, and pass the beginning and end positions of each block as function parameters. The `#pragma independent` directive is used, which allows each thread to work on its own data block.

Figure 4.3 illustrates the block partitioning scheme in the 2-D case. Figure 4.4 illustrates the block partitioning scheme in the 3-D case.

The algorithm that exploits heterogeneous parallelism to compute of multi-dimensional wavelet transforms can be described as follows:

Step1: `Read the data from the data file into shared memory;`
Step1:

```
p = number of threads;
#pragma parallel local(Npoints[ndim]) shared(cd)
{
  for each thread p {
    #pragma independent
    n-D DWT computed on the block in each thread;
  }
}
```

Step3: `Write the result to a file.`

5.3 Predicted Performance

In Chapter 2, the QSM model was presented as a model of shared memory computing. The QSM model consists of a number of processors, each with its own private memory, communicating by reading and writing locations in the shared memory. Processors execute a sequence of synchronized phases, each consisting of the following three sub-phases: `read`, `compute`, and `write`. The cost of a phase is defined as: $\max\{m_{op}, k, g \cdot m_{rw}\}$, where m_{op} is the maximum number of local operations by a processor, k is the maximum contention, g (≥ 1) is the local instruction rate divided by the communication rate, and m_{rw} is the maximum number of shared memory reads or writes by a processor. The time complexity of a QSM algorithm is therefore the sum of the time costs for all phases in the algorithm.

5.3.1 The machine parameters

The machine parameters that are used as performance metrics in the QSM model are listed below:

- In the SGI Power Challenge used for this research, the number of the processors is p, where $1 <= p <= 6$.

- The bandwidth gap g (≥ 1) is the local instruction rate divided by the communication rate. Assume $g = 10$.

The above are the parameters of the model for the Power Challenge. In the following subsection, the algorithm's predicted performance will be presented.

5.3.2 Algorithm analysis

When the data set is of size $N_1 \times N_2 \times N_3 ... \times N_n$, the homogeneous parallel algorithm to compute the n-dimensional DWT can be divided into n super steps, where n is the number of dimensions. Each superstep computes a single dimension DWT. Each of the supersteps consist of three sub-phases: `read`, `compute` and `write`. The cost of a phase is defined as: $\max\{m_{op}, k, g \cdot m_{rw}\}$, where $m_{op} = order \times N_1 \times N_2 \times N_3 ... \times N_n/p$ ($order$ is the length of the filter, $order = 4$ for DAUB4, the transform used for this work); $k = 1$ (no concurrent read or write); $g = 10$, and $m_{rw} = 2 \times N_1 \times N_2 \times N_3 ... \times N_n/p$. Therefore the cost of a superstep is $20 \times N_1 \times N_2 \times N_3 ... \times N_n/p$. The total cost of the algorithm is $20 \times n \times N_1 \times N_2 \times N_3 ... \times N_n/p$. Notice that this results in a linear speedup with respect to the number of processors.

5.4 Numerical Results

The Daubechies wavelet transforms have been implemented on an SGI Power Challenge. The C Multi-Processing (MP) directives such as `#pragma parallel`, `#pragma pfor` and `#pragma independent` are inserted in the C code to generate calls to the multiprocessing library. The C timing function `clock()` is used for timing. The usage of `clock()` is as follows:

```
Main(int agrc, char* agrv[]) {
        variables declaration;
```

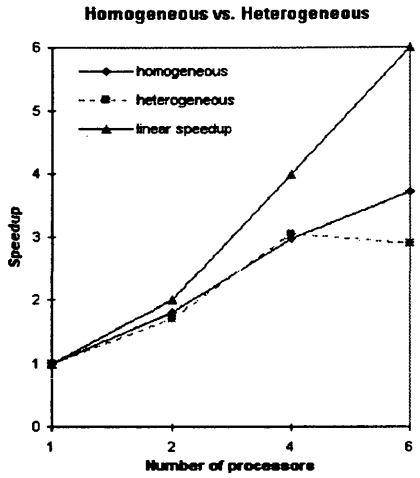

FIG. 5.2. The speedups obtained when the two kinds of algorithms are used on data of size $128 \times 128 \times 144$.

```
    start=clock();
    read the data from the file to memory;
    computation;
    save the results to the data files;
    finish=clock();
    cputime=finish - start;
}
```

Figure 5.2 compares the speedups when exploiting homogeneous parallelism with those when exploiting heterogeneous parallelism. The result shows that when the number of processors increases, homogeneous parallelism performs better than the heterogeneous parallelism. This is because `#pragma pfor` is more efficient than `#pragma independent`. Since `#pragma independent` can run different code blocks in parallel while `#pragma pfor` can only run the same code on different data (different iterations of loops), therefore the overhead for preparing `#pragma independent` threads is more than that for preparing `#pragma pfor` threads.

Table 5.1 shows the runtimes of the traditional data partitioning algorithm (ho-

Table 5.1. The runtime of the homogeneous parallelism implementation on different number of processors with different data sizes.

	Number of Processors			
Problem Size	1	2	4	6
32×32×32	0.320	0.290	0.270	0.240
64×64×64	2.670	1.880	1.410	1.360
128×128×128	25.770	15.150	9.850	8.320
128×256×256	107.590	59.900	36.220	28.880

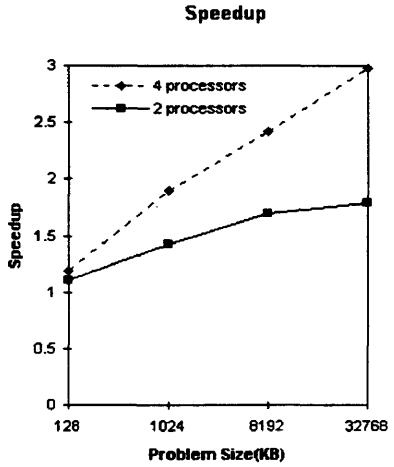

FIG. 5.3. The speedups obtained when homogenous parallelism is implemented on different numbers of processors with different data sizes.

mogeneous parallelism) on different numbers of processors with different data sizes. From this table, we can see that when the problem size increases, the speedup becomes better if the same number of processors are used. Figure 5.3 shows this clearly. This is because when the problem size increases, the length of threads in each processor becomes longer, so the algorithm efficiency becomes better. If the length of the threads is short, the overhead for setting up and clearing the threads in a substantial fraction of the overall runtime, so the processor efficiency is low.

Table 5.2 shows the runtimes of the heterogeneous parallelism algorithm on dif-

ferent numbers of processors with different data sizes. The data partitioning scheme used is CRBP. This table shows:

- In the fifth column (the number of processors is 6), runtimes are reported only for two data sizes (row 6 and 8). Refer back to Figure 4.11, division is done along the third dimension, so the data length along this dimension needs to be a multiple of 6.

- In row 5 and row 7, the runtime with 6 processors being used is even greater than the run time with 4 processors being used. There are two reasons for this: first, the data blocks assigned to the processors in the CRBP scheme are not from consecutive locations in the original data. Therefore, larger the number of processors used, the smaller the consecutive data block. This implies that the file pointers need to be moved more often when reading/writing the blocks and this results in worse performance. Second, because the system has only 6 processors, one processor must do some system management work. If all of the p processors are used, the time performance is even worse than for $p-1$ or $p-2$ processors as the processor doing the system management work has additional overhead and causes the bottleneck in the performance of the overall program.

- When the problem size increase, the speedup becomes better if the same number of processors are used. Figure 5.4 shows this clearly. The explanation is as the same as figure 5.3

5.5 Conclusions

This chapter presented two parallel algorithms to implement n-dimensional DWTs on shared memory machines and showed how to implement them on the SGI Power

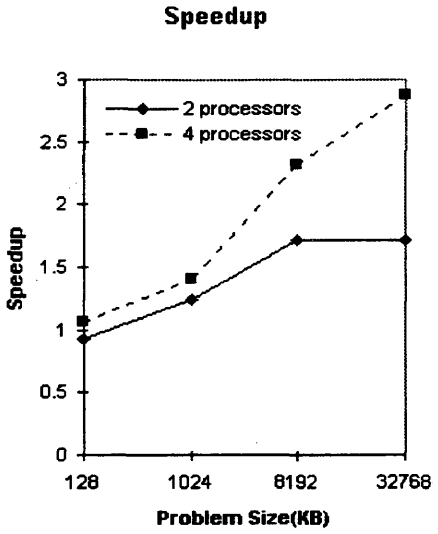

FIG. 5.4. The speedups obtained when heterogeneous parallelism is implemented on different numbers of processors with different data sizes.

Table 5.2. The runtime of the heterogeneous parallel implementation on different number of processors with different data sizes.

	Number of Processors			
Problem Size	1	2	4	6
32×32×32	0.35	0.38	0.33	
64×64×64	2.78	2.250	1.98	
128×128×128	25.980	15.130	11.21	
128×128×144	29.360	16.840	10.660	12.82
128×256×256	107.420	62.670	37.270	
256×256×144	120.380	70.450	39.440	41.700

Challenge.

There are two data partitioning schemes and two parallelism approaches used in the algorithms. Traditional data partitioning is used in the homogeneous parallelism algorithm, and block partitioning is used in the heterogeneous parallelism algorithm. The results show both parallel algorithms to compute n-dimensional DWTs are efficient. Comparing the homogeneous parallelism algorithm with the heterogeneous parallelism algorithm, the homogeneous parallelism algorithm to compute DWTs is more accurate and efficient. Given the drawback of block partitioning (discussed in Chapter 4), if there are very high frequencies in the data, the quality of the reconstructed data can be affected. The homogeneous algorithm to compute the DWT is as accurate as the sequential algorithm. The `#pragma pfor` is more efficient than `#pragma independent`, so the homogeneous algorithm of DWT is more efficient than heterogeneous algorithm of DWT. Note that CRLP could be used for the heterogeneous case too.

Chapter 6

CONCLUSIONS

The overall goal of this research was to design efficient parallel algorithms to compute n-dimensional wavelet transforms on both coarse-grained shared memory as well as distributed memory machines. Wavelet transforms have been successfully used in many different areas, however multi-dimensional wavelet transforms can be prohibitively costly from the computational perspective. Thus, the parallelization of the multi-dimensional wavelet transform is of significance.

There has been some research on the parallel implementation of wavelet transforms, especially 1-D and 2-D transforms. This thesis considers the more general case, the parallel n-dimensional wavelet transforms, where n is a variable. This should cover all issues in parallelization of wavelet transforms of any dimensionality.

One approach to implementing n-dimensional discrete wavelet transforms in parallel involves the traditional way to partition data among PEs. This partitioning approach and the associated implementation algorithm was discussed. On distributed memory machines, data communication is very costly among processors. To reduce the data communication cost, the block partitioning scheme was proposed in this thesis. The block partitioning method dramatically reduces the data communication among the processors. However, the data communication cost still remains high, especially in an environment of a cluster of workstations connected by Ethernet. To further optimize the parallelization, the CRBP and CRLP were proposed along with appropriate implementation algorithms. CRBP and CRLP minimize the data com-

munication among the neighboring processors, and show attractive speedups on distributed memory machines. Another advantage of these methods is that distributed memory machines can run large problems that may be too large for a single machine.

Two data partitioning schemes and two parallelism approaches to implement n-dimensional DWTs on shared memory machines are discussed in this thesis. Traditional data partitioning is used in the homogeneous parallelism algorithm, and CRBP is used in the heterogeneous parallelism algorithm. The results show both parallel algorithms to compute the n-dimensional DWT are efficient. Comparing the homogeneous parallelism algorithm with the heterogeneous parallelism algorithm, the former is more accurate and efficient. Because of the drawback of CRBP (discussed in Chapter 4), if there are very high frequencies in the data, the quality of the reconstructed data can be affected. The homogeneous parallelism algorithm to compute the DWTs is as accurate as the sequential algorithm. The results show that the presented parallel algorithms to compute DWTs are efficient.

When we compare the algorithms for shared memory machines with those for distributed memory machines it is apparent that the homogenous parallelism approach is the method of choice for shared memory machines because it is accurate and quick. The CRLP approach turns out to be the fastest algorithm on distributed memory machines.

There are several feasible directions for future research in this area. The first is to add load balancing into the algorithms developed for coarse-grained distributed memory machines. As mentioned in Chapter 4, if different machine models are connected within the same LAN, the weak machines hurt the overall performance. If the algorithms can assign loads to machines according to the machine's capabilities, the efficiency of algorithms will be further enhanced.

Another direction of future research is to investigate the application of shared distributed memory machines such as the SGI/Cray Origin 2000. One can port the code to the Origin 2000 straight from the shared memory SGI Power Challenge, but the memory architecture is actually distributed. Thus code written for the SGI Power Challenge can be easily ported on Origin 2000, but the speedup is not as high as on the Power Challenge. In the future, we need to investigate the new memory architecture, and design an efficient algorithm for the SGI Cray Origin 2000. The work in this thesis has provided a strong base and a variety of possible research directions.

The third possible direction is to apply the n-dimensional wavelet transforms to real applications such as seismic data. Due to the vast amount of data available in seismic applications, parallelism is of great significance.

In summary, this research has achieved progress in the following areas:

- It developed and implemented the most general case n-dimensional wavelet transform parallel algorithms.

- The parallel algorithms to compute DWTs developed in the research are for coarse grained machines, while most previous efforts were for fine grained SIMD machines.

- For distributed memory machines, two new algorithms, CRBP and CRLP, are designed in this research. In the CRBP and CRLP, once the data are distributed to each processor, n-dimensional discrete wavelet transforms are applied to the local data blocks in each processor simultaneously. Once the transforms are carried out, the results are gathered. Communication between processors is only necessary during data distribution and data gathering. Therefore the CRBP and CRLP algorithms are especially efficient when a NOW (Network Of Workstations) is used. Both CRBP and CRLP algorithms are tested on the SGI

cluster, with various data set sizes, and various number of workstations being used. The results show that attractive speedup can be achieved. Examples of image compression using wavelet transforms show that high quality reconstructions can be achieved with high compression ratios.

- For shared memory machines, two kinds of parallel approaches, homogeneous parallelism and heterogeneous parallelism, are used to compute multi-dimensional wavelet transforms in parallel. Homogeneous and heterogeneous parallel algorithms on n-dimensional wavelet transforms are designed in this research. The effectiveness of these approaches is demonstrated through several examples implemented on an SGI Power Challenge.

In summary, this research provides a new way to compute the n-dimensional wavelet transforms in parallel.

REFERENCES

Caulfield, H. J. 1992. Parallel discrete and continuous wavelet transforms. *Optical Engineering*, **31**, 1835–1839.

Chui, Carles K. 1992. *An Introduction to Wavelets.* New York: Academic Press, Inc.

Cormen, Thomas H., Leiserson, Charles E., & Rivest, Ronald L. 1990. *Introduction to algorithms.* Cambridge: MIT Press.

Culler, D., Karp, R., & Pratterson, D. 1993. LogP: Towards a Realistic Model of Parallel Computation. *Pages 1–12 of: Proceedings of 4th ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming.*

Culler, D., Karp, R., & Pratterson, D. 1996. LogP: A Practical Model of Parallel Computation. *Communication of the ACM*, **39**, 78–85.

Daubechies, I. 1992. *Ten Lectures on Wavelets.* Philadelphia: SIAM.

Foster, I. T. 1994. *Designing and building parallel programs.* New York: Addison-Wesley Publishing Company.

Gibbons, Phillip B. 1996. What good are shared-memory models. *In: International Confrence in Parallel Processing.*

Gropp, W., Lusk, E., & Skjellum, A. 1996. *A high-performance, portable implementation of the MPI message passing interface standard.*

Hambrusch, S. E., & Khokhar, A. A. 1996. C^3: A Parallel model for coarse-grained machines. *Journal of Parallel and Distributed Computing*, **32**, 139–154.

Holmström, Mats. 1995. Parallelizing the fast wavelet transform. *Parallel Computing*, **21**, 1837–1848.

Hoyt, J. D., & Weschsler, H. 1992. The Wavelet Transform-a CMOS VLSI ASIC implementation. *Pages 19–22 of: 11th IAPR International Conference on Pattern Recongnition*, vol. IV.

Johnson, Howard W. 1996. *Fast Ethernet-Dawn of a new network.* Upper Saddle River, NJ: Prentice Hall PTR.

Kalantery, N., Winter, S.C., & Wilson, D.R. 1995. From BSP to a virtual von Neumann machine. *Computing and Control Engineering Journal*, **6**, 131–136.

Kita, Takuro. 1993. The optimum approximation of multi-dimensional signals using parallel wavelet filter banks. *IEICE Trans. Fundamentals*, **E76-A**, 1830–1848.

Lega, E., Scholl, H., Alimi, J. M., Bijaoui, A., & Bury, P. 1995. A Parallel algorithm for structure detection based on Wavelet and segmentation analysis. *Parallel Computing*, **21**, 265–285.

Lu, J. 1993. Parallelizing Mallat Algorithm for 2-d Wavelet Transform. *Information Processing Letters*, **45**, 255–259.

Mallat, S. 1989. Multiresolution approximations and wavelet orthonormal bases of $L^2(R)$. *Trans. Amer. Math. Soc.*, **315**, 69–87.

Meng, Z., & Yang, L. 1995. Seismic data compression with wavelet. *Oil Geophysical Prospecting*, **30**.

Misra, M., & Nichols, T. 1994. Computation of 2-D wavelet transforms on the Connection Machines-2. *Pages 1–10 of: Proceedings of the International Conference on Applications in Parallel and Distributed Computing.*

Misra, M., & Prasanna, V. K. 1992. Parallel computation of wavelet transforms. *In: Proceeding of the 11th International Conference on Pattern Recognition.*

Pacheco, Peter S. 1997. *Parallel Programming with MPI.* San Francisco, CA: Morgan Kaufmann Publishers.

Parhi, K. K., & Nishitani, T. 1993. VLSI arichitectures for discrete wavelet transforms. *IEEE Transactions on Very Large Scale Integration Systems*, **1**, 191–202.

Pic, M. M., & Essafi, H. 1993. Wavelet Transform on Connection Machine and Sympatia. *Int. J. Modern Physics*, **4**, 97–103.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. 1992. *Numerical Recipes in C.* New York: Cambridge University Press.

Reif, John H. 1993. *Synthesis of parallel algorithms.* San Mateo, CA: Morgan Kaufmann Publishers.

Sabot, Gary W. 1995. *High Performance computing– Problem solving with parallel and vector architectures.* New York: Addison-Wesley Publishing Company.

Sahinoglou, H., & Cabrera, S. D. 1991. A high-speed pyramid image coding algorithm for a VLSI implementation. *IEEE Transactions on Circuits and Systems*, **1**, 369–374.

Strang, G., & Nguyen, T. 1996. *Wavelets and Filter Banks.* Wellesley: Cambridge Press.

Valiant, L. G. 1990. A Bridging Model for Parallel Computation. *Communications of ACM*, **33**, 103–111.

Wickerhauser, M. V. 1992. Acoustic signal compression with wavelet packets. *Wavelet Analysis and Its Applications, Chui, C. K. (ed.)*, 679–700.

Yang, L., & Meng, Z. 1995. One dimensional ladder inversion of wave equation by wavelet representation. *Chinese Journal of Geophysics*, **38**, 815–822.

er: lyang
st: blackarrow
ass: blackarrow
o: stdin