GODDARD GRANT

IN-61-CR

114845-

P-186

Algorithms and Programming Tools for Image Processing on the MPP: #3

Final Report for the Period
May 1984 to July 1987

Anthony P. Reeves
School of Electrical Engineering
Cornell University
Ithaca, New York 14853

Work Supported by NASA Grant NAG 5-403

Algorithms and Programming Tools for Image Processing on the MPP: #3

Final Report for the Period
May 1984 to July 1987

Anthony P. Reeves
School of Electrical Engineering
Cornell University
Ithaca, New York 14853

Work Supported by NASA Grant NAG 5–403

Abstract

This is the third and final report on the work done for NASA Grant 5–403. All the work done for this grant is summarized in the introduction and work done since August 1986 is reported in detail. For work prior to this date, see Report #1 for the period May 1984 to November 1985 and Report #2 for the period March 1986 to August 1986. The Students who have contributed to this work are Cristina Mahon, Maria Gutierrez and Marc Willebeek-LeMair.

The work for this grant falls under the following headings: (a) fundamental algorithms for the MPP, (b) programming utilities for the MPP, (c) the Parallel Pascal Development System, and (d) performance analysis. In this report, the results of two research efforts are presented: region growing by Marc Willebeek–LeMair and performance analysis of important characteristic algorithms by Maria Gutierrez. In each case, timing results from MPP implementations are included. A paper on Marc's work is included in which parallel algorithms for region growing on the MPP are discussed. These algorithms permit different sized regions to be merged in parallel. Marias' Masters thesis, which is included, gives details on the implementation and performance of several important MPP algorithms. These include: a number of standard permutations, the FFT, convolution, arbitrary data mappings, image warping, and pyramid operations. All of these functions have been implemented on the MPP. The permutation and image warping functions have been included in the standard development system library.

Table of Contents

1. Introduction

The implementation of a number of fundamental algorithms on the Massively Parallel Processor (MPP) has been studied. The work reported here may be categorized under the following four headings: (a) fundamental algorithms for the MPP, (b) programming utilities for the MPP, (c) the Parallel Pascal Development System, and (d) performance analysis.

In fundamental algorithms we first explored basic algorithms which were known to match well to the MPP namely local filtering including convolution. Algorithms which did not match directly onto the MPP were then considered these included: the FFT, image rotation, interpolation, image warping, pyramid operations, sorting and parallel region growing. We also explored enhancements to the Parallel Pascal software environment which would allow us and other researchers to more conveniently use the MPP.

First, a set of utility programs were written for commonly used functions that were not intrinsic to the Parallel Pascal Language. These included operations such as random number generation, large array manipulation, masked reduction functions, near neighbor functions pyramid primitive operations and standard data permutations.

Second, since since Pascal (and also Parallel Pascal) does not have a library mechanism, a library preprocessor was developed. This permitted algorithms and utility functions to be installed in a *System Library* in such a way that they could be invoked to operate on different data array sizes and also on different data types. These features are not usually found with other library preprocessors but are very important for the MPP environment.

Third, the development and maintenance of a Parallel Pascal Development system, which allowed programs in Parallel Pascal to be developed and tested on any conventional computer, became vital to our work. This research was started before the MPP was operational. Furthermore, once the MPP was operational, we have had continuous problems with both the data link and the system environment when using the MPP remotely. The Development System permitted algorithm development on the local computing resources and reduced the contention for access to the MPP. The Development System documentation [1] has been distributed to the MPP users and we have distributed the Development System including the System Library to a number of different sites.

Finally, we developed methods to characterize the performance of the MPP. This was useful to both provide information on permutation costs to MPP programmers and to asses the efficiency of the programming environment. A number of the fundamental algorithms and their performance on the MPP are covered in this final report.

The Students who have contributed to this work are Cristina Mahon, Maria Gutierrez and Marc Willebeek-LeMair. Cristina and Maria have now completed their masters degrees and Marc is still working towards the PhD degree. The results of this research program include (a) a number of novel algorithms which have been implemented on the MPP, (b) a set of useful programming utilities which have been organized in a system library, (c) a (documented) program development system which can be used on conventional computers (this includes the system library) and (d) a detailed performance characterization of the MPP and its software environment.

The remainder of this section outlines the achievements of the research program. Sections two and three outline the work covered by the previous two reports and section four describes the work done during the period covered by this report.

1.1 Fundamental Algorithms

A number of important fundamental algorithms which test various aspects of the MPP architecture have been developed and analyzed. All of these algorithms were first developed on a conventional computer using the Parallel Pascal Development System; they were then moved to the MPP where their performance was tested. Making these algorithms operational on the MPP typically involved more work than their initial development. Most of the results for the MPP implementations are given in this report.

The algorithms which have been developed include: the FFT, two dimensional convolution, median filtering, sorting, image warping, image rotation (with interpolation), random number generation, pyramid filtering (Gaussian and Laplacian), general data mapping, and parallel region growing.

1.2 Programming Utilities

The Parallel Pascal environment was intended to be augmented with a system library of commonly used functions. A large number of utility functions have been developed and installed in such a library. These include: masked reduction functions, large array shift functions, regular permutations, and the fundamental algorithms mentioned above.

1.3 The Parallel Pascal Development System

A program development system has been developed for both the MPP and conventional serial computers. This system greatly simplifies the development of high level language programs for the MPP. Furthermore, it allows programs to be developed and tested on any conventional computer using the debugging aids available for Pascal on that computer. This environment consists of a set of system programs and a library of general purpose Parallel Pascal functions. The documentation for the Parallel Pascal Development system is given in report #1. Updates to this library are described in report #2 and this report. We have distributed the UNIX version of the Parallel Pascal System to number of sites; the VMS version has been distributed by NASA on the MPP software tape.

1.4 Performance Analysis

Conventional performance measures for supercomputers, such as Fortran loops, are not a good way to benchmark the MPP. Massively parallel systems are not suitable for all scientific applications; however, there is an important class of applications for which they are very effective. We are interested here in evaluating the performance of such systems for applications to which they are well matched. Typical algorithms of such applications are the Fast Fourier Transform (FFT) and matrix convolution; the performance of both of these operations on the MPP is considered in detail.

In order to further analyze the performance of a massively parallel processor system it is necessary to characterize its data permutation ability. A characterization scheme has been developed which involves measuring the ability to perform a set of regular permutations. These permutations occur in many scientific problems and knowledge of their implementation speed may also be useful in guiding a programmer to develop efficient programs.

1.5 Visits to NASA

During this report period Maria, Marc and myself have visited the NASA Goddard Space Flight Center on several occasions. Algorithms have been tested on the MPP and a presentation on the development system was made to the MPP users group.

1.6 Pioneer Remote Users

In addition to the development of the reported algorithms and software this grant provided us with the opportunity to be the pioneer remote users of the MPP. Most of the work on the MPP was done from Cornell over a telephone line which in itself absorbed a significant amount of the available funds. We also made several visits to NASA; on two occasions seminars were presented on the Parallel Pascal environment. Considerable discussions were held with NASA on the development of the I/O system for the MPP and other aspects of the high level language environment. The complete Parallel Pascal development system has been installed on the MPP host and has been distributed by us to a number of remote sites. It has been ported to a number of conventional computers including VAX systems running either VMS or UNIX.

2. Summary of Research Covered in Report #1.

A major contribution to this initial work was made by Cristina Mahon (previously Cristina Moura); her masters thesis [2] is included in report #1. In addition to developing a number of algorithms for the MPP, Cristina was one of the first remote users of the MPP. She developed a number of programs for conveniently using the MPP from a remote site and much of her time was spent identifying bugs in the initial version of the MPP code generator. She developed a number of novel algorithms for arbitrary data mappings, permutations, and image rotation including interpolation. These algorithms were all tested and evaluated using the Parallel Pascal Development system; several of the algorithms were implemented on the MPP. These algorithms have been published [3] and a copy of this paper has been included in report #1. The specification of the Parallel Pascal language has now been published [4, 5]; these papers are included in report #1.

In addition to Cristina's work, the manual for the Parallel Pascal development System and the documentation for the system library is included in report #1.

Some of the research highlights which are covered in report #1 are listed below. The following three subsections deal with fundamental algorithms, programming utilities and the development system. No work on performance analysis was done during this first period.

2.1. Fundamental Algorithms

Three types of algorithm were developed during the period of report #1. These were heuristic data mapping, matrix rotation and interpolation.

A Data Mapping Algorithm. A fast heuristic arbitrary data mapping algorithm has been developed. For most mappings this is much faster than other techniques such as sorting. This has been implemented on the MPP for both regular (128 x 128) and large ((n * 128) x (m * 128) two dimensional arrays.

Matrix Rotation Algorithms. Fast matrix rotation algorithms have been implemented based on the above data mapping function. The nearest neighbor algorithm has been tested on the MPP. Large matrix nearest neighbor rotation, and interpolation schemes have been developed and tested on the Development System only.

Interpolation Algorithms. A high speed interpolation algorithm has been implemented for bilinear and bicubic interpolation for image rotations (any angle) and small matrix warps. These have been tested on the Development System but not on the MPP.

2.2. The System Library

A set of general purpose library programs have been developed. All of these programs run correctly on the development system and nearly all of them have been tested on the MPP.

General Utilities. Programs in the general utilities group include a parallel random number generator, an index generator, simplified I/O functions and a parallel ceiling function.

Masked Reduction Functions. It is frequently necessary to apply a reduction function to a subregion of an array. The masked reduction library functions are similar to the primitive reduction functions except that a Boolean array mask parameter is required.

Large Array Utilities. The large array utilities are shift and rotate functions that operate on matrices which have dimensions that are multiples of 128. Both the crinkled and blocked data structures are supported. There are also shift and rotate functions which treat a 128 x 128 array as a vector of 16384 elements. These functions do not use the hardware spiral interconnections; the use of the regular mesh interconnections is faster for multiple element shifts.

Near Neighbor Convolution Functions. Many low level image processing applications require convolutions between images and small kernel matrices. Programs in this group simplify the entry of small matrices and the application of these matrices to image arrays.

Pyramid Operations. A pyramid convolution function has been implemented; this is a three dimensional convolution function which operates on the 13 near neighbors of a pyramid data structure. This data structure is embedded within an MPP array. Functions are also available for both the vertical and horizontal data shift operations that are associated with pyramid algorithms.

2.3. The Development System

A Single Compiler Command. A command file has been written both for the MPP and the development system which, with a simple noninteractive command, compiles a program, makes all the library links, and in the case of the MPP, loads the program onto the system.

The Library Preprocessor. Standard Pascal does not have any library facilities. A general purpose library preprocessor has been developed which works for both the MPP compiler and the development system. The preprocessor looks for library subprograms first in named files, then in the local directory, and finally in a system library directory. The MPP compiler version of the preprocessor also examines a special MPP system library before the general system library. This library contains system library programs which have been modified to overcome deficiencies in the MPP compiler. This library preprocessor can be used in conjunction with the assembly language library feature which is built into the MPP Parallel Pascal compiler. It is also able to work in conjunction with any library facilities that are available with a local Pascal compiler that is used with the development system.

The Parallel Pascal Translator. The translator is the heart of the development system. It translates a Parallel Pascal program into standard Pascal for execution on a conventional serial computer. The translator is a Pascal program with over 8000 lines of code. It has been in a very stable form for over a year now. It still has some limitations but these are now well documented. In addition to being used by this and other MPP research groups it has also been used in a Parallel Processing course which has now been offered three times with an enrollment of about 50 students each time.

3. Research Covered in Report #2

The work described in this report was conducted by Maria Gutierrez, Marc Willebeek-LeMair, and myself. The work for this period of the grant falls into two main categories: algorithms for the MPP and performance analysis of data manipulations for the MPP and related architectures. Maria has developed a number of novel algorithms for image warping and pyramid image filtering. Marc has investigated techniques for the parallel processing of a large number of independent irregular shaped regions on the MPP. In addition some new utilities for dealing with very long vectors and for sorting have been developed. Not all algorithms have been made to work on the MPP. The effort in the final period of the grant will concentrate on the MPP implementation.

The performance of the MPP for a number of basic data manipulations has been determined. From these results it is possible to predict the efficiency of the MPP for a number of algorithms and applications. Some of these results have been published [6, 7, 8] and these papers are included in report #2.

Some of the highlights of the research for this report period for fundamental algorithms, utilities and performance analysis are listed below. The Parallel Pascal development system has been improved, and better documentation including a tutorial has been written [1]. The new tutorial is included in report #2.

3.1 Fundamental Algorithms

Image Processing Algorithms. Algorithms for image warping have been developed; both nearest neighbor and bilinear interpolation schemes have been implemented. Both of these functions are guided by the same heuristic technique which is very efficient for small arbitrary warps but can also deal with large image distortions.

Building on the pyramid processing primitives, which were mentioned in the previous report, Laplacian and Gaussian pyramid image filters have been implemented. These algorithms are used to decompose an image into a number of bandpass filtered subimages. A number of interesting efficient image analysis and image filtering algorithms have been based on this pyramid of subimages.

Local Region Processing. A new approach to the parallel processing of independent regions in parallel on the MPP is being investigated. For each region in an image a tree is created which spans the region and can be used to compute features of the region. Special provisions have been made for generating trees for non–simply connected regions. Techniques for parallel region merging have been developed. After merging two regions, a new tree is generated which covers the region in a minimum amount of time. Using these tree procedures, an image segmentation algorithm, based on the split and merge paradigm, has been implemented.

3.2. General Utilities

A general purpose sorting algorithm has been implemented. Based on the bitonic sorting technique this program can sort the rows, the columns or treat the whole PE matrix as a long vector. Any basic data type can be sorted.

There are new utilities for local mean, local median and local maximum filters. Also, a general purpose binary matching algorithm has been developed. Documentation for all of these functions is given in report #2.

3.3. Performance Analysis

An analysis of different data permutations and manipulations on the MPP is presented in [6] which is also included report #2. This analysis expresses the cost of data manipulations in terms of elemental arithmetic operations; Boolean, integer and floating point data types are considered. Results are computed for arrays of size 128 x 128, 256 x 256, and 512 x 512. An extended version of this paper, which includes a general description of the MPP, is given in [7] which is also included in report #2.

There has been much recent interest in the implementation of parallel pyramid data processors. Such a processor could be maded with the MPP by augmenting the processor array with a pyramid structure of additional processing elements. A pyramid processor based on the MPP is considered in [8] which is also included in report #2. The results from an analysis of the proposed system indicate that, in general, there would be little advantage in having the additional pyramid hardware for implementing many of the pyramid algorithms.

4. Research Covered in this Report

Significant work has been done on fundamental algorithms and utilities during this last grant period while no new work has been done to the Development system with the exception of updating the library with the newly implemented algorithms. The main tasks for the final part of this research project have been to implement a number of algorithms on the MPP and do a detailed evaluation of their performance.

The work by Marc on parallel region growing is given in Appendix A. He identifies a number of primitives for parallel region growing and uses a dynamic embedded tree structure to represent these regions.

Marias work is described in her thesis [9] which is given in Appendix B. Main highlights of her achievements include: the implementation of algorithms for image warping, pyramid growing and the FFT on the MPP; the implementation and performance evaluation of a number of regular permutations on the MPP; and the derivation of optimal execution times for many algorithms on the MPP.

4.1 Fundamental Algorithms

Parallel Region Growing. A number of primitive operations for parallel region growing have been designed and implemented. The results of applying a split and merge region growing algorithm to various test images on the MPP is reported in Appendix A. The implementation is currently not very efficient since it involves a large number of Boolean array operations which have been programmed in Parallel Pascal. The current implementation is fast enough for convenient algorithm prototyping.

The Fast Fourier Transform. An FFT function has been implemented on the MPP which can transform either the rows or the columns of a 128 x 128 array in one pass or a two dimensional transform in two passes. The manual page documentation is included in Appendix C and algorithm and timing details are given in Appendix A.

Convolution and Pyramid Building. The implementation of Pyramid filtering and two dimensional convolution operations is reported in Appendix B. Convolution operations are quite efficient; however, the implementation of the pyramid functions was much less than optimal due to the large number of Boolean array operations in the pyramid algorithms.

Image Warping and Interpolation. The implementation of a heuristic image warping algorithm on the MPP is reported in Appendix B. Both nearest neighbor and bilinear interpolation are considered; the warp algorithm is tested with image rotation and fish–eye image warps. The heuristic algorithm is compared with a simple direct mapping technique. Once again, the heuristic algorithm does not work as well in practice due to the use of the inefficient Boolean operations. A further problem is the suboptimal implementation of the reduction functions. However, in many cases, the heuristic algorithm is still much faster than the simple algorithm.

4.2 Programming Utilities

Permutation functions. A number of regular permutation functions have been developed to support the FFT algorithm and for performance analysis. The permutations are exchange, shuffle, butterfly and bit reverse. All super and sub variations of these permutations have also been implemented. The manual page documentation for these functions is given in Appendix C. For more details see Appendix B.

4.3 Performance Analysis

The Transfer Ratio. A comparative measure, the *transfer ratio* [7], is used to express the cost of an algorithm. The *transfer ratio* is defined as the ratio of the time for the data transfer over the time for an elemental operation. The time for an elemental operation is defined as the average between the time of a multiplication and the time of an addition on the processor array [7].

Data Permutations. The performance of the MPP for a number of classical regular data permutations has been explored. Due to the orthogonality of these permutations, only one dimension of the MPP array needs to be considered in order to characterize its performance. On the MPP each permutation is performed concurrently on each row of the processor array; i.e., 128 sets of 128 elements (the performance for permuting the columns is identical). Since all the permutations considered are orthogonal with respect to the two dimensions of the MPP mesh connections, these results may be simply extended to the case of permuting a 16384 element vector (or 128 x 128 matrix); the transfer ratio cost will be doubled and the compiler efficiency will remain the same. See Appendix B for details of these permutations and their performance on the MPP.

Conclusion

Experience has been gained form the implementation of a number of fundamental algorithms on the MPP from which the following general conclusions can be drawn.

1. MPP Permutation Capability

   The capability of the MPP to perform a set of regular permutations has been studied in detail. The results indicate that the optimal implementation times for floating point data transfers may be reasonable for many applications but the permutation of small length data may not be very efficient. The performance of the total system for permutations is also quite good for floating point data but significant savings might be made if the shorter data type permutations were reprogrammed in PCU microcode.

2. Parallel Pascal Compiler Performance

Several characteristic algorithms have been considered: convolution which is implemented with a few primitive operations, the FFT which involves significant data permutations, and the heuristic algorithm which has a data dependent behavior. In general, the Parallel Pascal Compiler performed well for large data types and deterministic algorithms (which provided the lightest load for the MCU). It did not perform as well for complex algorithms involving Boolean data or reduction functions; however, in this case it was still quite adequate for algorithm prototyping. It is not clear that an optimizing compiler would be very much faster, for the difficult algorithms, unless it generated PCU microcode for critical sections.

3. The Development System

The development system has proved to be an indispensable tool for program development. It has been stable for a number of years and it is suitable for reliable algorithm development before implementation on the MPP.

4. Performance Analysis

A number of tools for performance analysis of highly parallel systems have been developed. A table of the performance of permutations and other data manipulations on the MPP is a useful aid for an MPP programmer.

5. Parallel Pascal

The language Parallel Pascal has proved to be adequate to express all algorithms in an efficient style. The language has also been found to be suitable for programming a class of problems for hypercube systems [10]. A problem with the MPP code generator was that it did not implement all the language features and, furthermore, some features were implemented inefficiently. As intended in the original design, the language is augmented by the functions in the system library by means of the library preprocessor which is essentially transparent to the user programmer. On the MPP implementation, a set of identical functions but written in microcode for efficiency could be developed and installed, especially for data permutation operations which use special features of the processor array. These functions would be indistinguishable to the user from other library functions.

6. MPP Processing Speed

In terms of processing speed, using Batchers figures [11] the peak performance of the MPP is 288 MFLOPS; from our primitive operation measurements the fastest rate we could expect to attain is 210 MFLOPS (due to the slower add time). For the convolution algorithm a rate of 155 MFLOPS was attained and for a 128 x 128 FFT the rate was 87 MFLOPS. These algorithms were conveniently programmed in Parallel Pascal. Furthermore, 128 x 128 is the worst case size for the FFT implementation on the MPP; for either larger or smaller matrix sizes the comparative overhead due to interprocessor communication would be less.

References

1. A. P. Reeves, *Parallel Pascal Development System Version 2.0*, Cornell University, School of Electrical Engineering Technical Report, October, 1986.
2. C. Moura, *Programming Tools and Algorithms for the Massively Parallel Processor*, MS Thesis, Cornell University, 1985.
3. A. P. Reeves and C. H. Moura, "Data Mapping and Rotation Functions for the Massively Parallel Processor," *Proceedings of Computer Architecture for Pattern Analysis and Image Database Management*, pp. 412–419 (November 1985).
4. A. P. Reeves, "Parallel Pascal: An Extended Pascal for Parallel Computers," *Journal of Parallel and Distributed Computing* **1** pp. 64–80 (1984).
5. A. P. Reeves, "Parallel Pascal and the Massively Parallel Processor," pp. 230–260 in *The Massively Parallel Processor*, ed. J. Potter, MIT Press (1985).
6. A. P. Reeves and C. H. Moura, "Data Manipulations on the Massively Parallel Processor," *Proceedings of the Nineteenth Hawaii International Conference on System Sciences*, pp. 222–229 (January, 1986).
7. A. P. Reeves, "The Massively Parallel Processor: A Highly Parallel Scientific Computer," pp. 239–252 in *Data Analysis in Astronomy II*, ed. V. Di Gesu, Plenum Press (1986).
8. A. P. Reeves, "Pyramid Algorithms on Processor Arrays," pp. 195–213 in *Pyramidal Systems for Computer Visison*, ed. V. Cantoni and S. Levialdi, Academic Press (1986).
9. M. Gutierrez, *Algorithms and Performance Analysis for the Massively Parallel Processor*, MS Thesis, Cornell University, January 1988.
10. A. P. Reeves and D. Bergmark, "Parallel Pascal and the FPS Hypercube Supercomputer," *Proceedings of the 1987 International Conference on Parallel Processing*, pp. 385–388 (August 1987).
11. K. E. Batcher, "Design of a Massively Parallel Processor," *IEEE Transactions on Computers* **C–29**(9) pp. 836–840 (September 1981).

APPENDIX A: REGION GROWING ON A HIGHLY PARALLEL MESH-CONNECTED SIMD COMPUTER

REGION GROWING ON A HIGHLY PARALLEL MESH-CONNECTED SIMD COMPUTER

Marc Willebeek-LeMair[†] and Anthony P. Reeves[*]

[†] School of Electrical Engineering
Cornell University
Ithaca, New York 14853

[*] Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

Index Terms: Image structure, Segmentation and 2-D description, Parallel processing, Region growing, and Split and merge.

Abstract

The region growing paradigm for image segmentation groups neighboring pixels into regions depending upon a predetermined homogeneity criteria. A parallel method for region growing on a highly parallel SIMD mesh computer is presented. In order to do this, techniques to independently compute functions on irregular shaped regions have been developed. These techniques make use of an embedded tree data structure to represent the regions in an image. The results of implementing a parallel split and merge region growing algorithm on the Massively Parallel Processor are discussed. The approach is shown to be efficient primarily for images involving large numbers of regions.

1. Introduction

Region growing is a general technique for image segmentation. Frequently, the basic scheme is to combine pixels with adjacent pixels to form regions; regions are then merged with other regions to "grow" larger regions. The association of neighboring pixels and neighboring regions in the region growing process is often governed by a homogeneity criterion that must be satisfied in order for the pixels and regions to combine. The homogeneity criteria are application dependent and may be dynamic within a given application.

In addition to the homogeneity criterion the order in which regions are merged can have an important effect on the final result; many sequential region growing algorithms ignore this fact. We consider parallel region growing techniques here, in which a merge decision can be based upon the best of all alternatives for all regions simultaneously.

Highly parallel SIMD processors have been shown to be very effective for regular algorithms such as image filtering and the FFT. This work extends the domain of the SIMD processor to the irregular processing characteristics of region growing algorithms. Our results indicate that, especially for the case of a very large number of small objects, powerful parallel region growing techniques can be implemented in a reasonable amount of time. Section two of this paper outlines the principles of region growing with emphasis on the split and merge algorithm. Section three discusses the characteristics of the SIMD architecture and section four considers the parallel implementation of a region growing technique on an SIMD architecture. The algorithm complexity is discussed in section five. An example of the parallel region growing technique is presented through a split and merge implementation on the Massively Parallel Processor (MPP) in section six.

2. Region Growing

Region growing is a technique for partitioning an image by linking individual pixels into groups of pixels called regions. The merging of pixels or regions to form larger regions is usually governed by a pre-defined homogeneity criterion that must be satisfied. In this section we first define a region and discuss its properties, next, we review the concept of homogeneity criteria, and finally, we present a split and merge algorithm as an example of a region growing technique.

2.1. Regions

A region might correspond to a world object or part of one. As defined by [1], a region is a four-connected, two-dimensional area that is allowed to be non-simply connected (contain holes). A single pixel cannot belong to more than one region. These properties stated more formally are as follows.

A region R is considered to be a set of points with the following properties:

(1) x_i in a region R is connected to x_j iff there is a sequence $\{x_i, \ldots, x_j\}$ such that x_k and x_{k+1} are connected and all the points are in R.

(2) R is a connected region if the set of points x in R has the property that every pair of points is connected.

(3) I, the entire image = $\bigcup_{k=1}^{m} R_k$.

(4) $R_i \cap R_j = 0$, $i \neq j$.

2.2. Homogeneity Criteria

When grouping pixels into regions it is usually necessary that the groups satisfy some sort of homogeneity criteria. Therefore, the grouping of neighboring pixels into regions is dependent on the characteristics of the individual pixels. Once pixels have been combined to form a region, the region assumes certain properties based on the combined characteristics of the pixels as a group (e.g. area or texture). A homogeneity criteria can be designed to specify such things as the maximum range or gradient allowable within a region, etc. This criteria will then be used as a test to determine whether or not a given group of pixels can be classified as a region.

As an example, the pixel range homogeneity criteria H(R) is defined as follows:

$$H(R) \equiv \begin{cases} true, & if\ for\ all\ points\ in\ R, \\ & f(x) - f(y) < T. \\ false, & otherwise. \end{cases} \tag{2-1}$$

This particular criterion requires that the range between the minimum and maximum values within a region not exceed a threshold, T.

A variety of homogeneity criteria have been investigated for region growing, ranging from statistical techniques, which involve the distribution of pixel grey levels, to state-space approaches, which represent regions by their boundaries [1,2,3].

A problem with many region growing schemes is that large regions require an excessive number of merge steps. A computationally expedient technique called split and merge [4] addresses the large region problem with a preprocessing split phase.

2.3. The Split and Merge Approach

The split and merge technique requires two types of operations; in our algorithm a fast split phase is followed by one or more merge phases. The split stage rapidly partitions an image into square regions which conform to a first homogeneity criterion; then a region growing technique is used to merge these square regions into larger regions which conform to a second homogeneity criterion. High speed is achieved by the top down approach that rapidly deals with very large regions. Such regions require many iterations to achieve with just a region growing paradigm.

(1) Splitting

The first phase of the algorithm is concerned with dividing the image up into homogeneous, square regions of varying dimensions. Beginning with the entire image as the area in question, an area is checked for homogeneity. Should the homogeneity check fail, the area is split into four quadrants. Each of the quadrants is then checked the same way and recursively split until the homogeneity requirement is satisfied. If an area passes the homogeneity test then it is considered a region and left alone until the merge stage of the algorithm. The scheme is illustrated in Figure 1. For this case the homogeneity criterion requires that the range of the pixel values in a region must not be greater than 2. Initially each pixel is labeled with a unique identifier number (id). While splitting, pixels are assigned a region id number corresponding to the pixel id of the pixel located in the upper left hand corner of the quadrant to which they are associated.

(2) Merging

In the merge phase, adjacent regions are merged to form non-square regions. Region pairs that satisfy the homogeneity criterion are allowed to merge. During a single merge iteration a given region can merge with only one other region. Figure 2 presents an example of the merging stage for the regions split in Figure 1. In this case, the same homogeneity criterion as for the split phase is used again. Now, however, merged region pairs assume the smaller id of the two regions. There is also a certain hierarchy to the merge sequence since those region pairs that best satisfy the merge criterion are allowed to merge first. Ties are broken by, arbitrarily, allowing the pair with the larger id's to merge first.

For many merging criteria, including the one used in the example, the order of merging is important and this affects both the execution time and the final result. An approach which involves increasing the threshold value, in stages, in the merge phase, has been found to improve the quality (i.e., produce less regions) in the final result for region growing using both range images [13] as well as grey level images [5]. Certain constraints imposed on the merging order, as mentioned above, help to improve on the final results, other constraints, however, are imposed in the parallel merging strategy to avoid violating the homogeneity criteria. These constraints will be addressed in section 4.

3. The Mesh-connected SIMD Architecture

Many highly parallel computer architectures designed for low level image processing applications have been proposed and implemented [6,7,8]. These designs exhibit a variety of interconnection schemes between processors and the processors themselves range from simple processing elements in the SIMD arrangements to much more sophisticated processors in the MIMD systems. The architecture of interest in this paper is the highly parallel (tens of thousands of binary processors) mesh-connected SIMD processor array. The SIMD mesh consists of an array of identical processing elements (PE's) with near-neighbor connections. We will be considering the 4-connected case in which processors are connected to their neighbors to the north, east, south, and west. The architecture we are considering is illustrated in Figure 3. The array of PE's is controlled by a host computer that issues it instructions when a parallel array computation needs to be performed and is linked to memory via a bi-directional I/O path. Each PE contains an ALU with a limited amount of local memory.

The SIMD-mesh architecture is particularly well suited for low-level image processing. The physical layout of the processor array allows for a direct mapping of pixels to PE's. Furthermore, the type of computations inherent in low-level processing, such as image filtering and edge detection, involve the analysis of a pixels local neighborhood.

3.1. Massively Parallel Processor Architecture

The Massively Parallel Processor [11] is a SIMD mesh-connected computer arranged in a two-dimensional 128x128 array of PE's. The PE's are bit-serial, allowing for a flexible data format and efficient utilization of resources. Each PE can perform high speed arithmetic and implement all sixteen possible boolean functions. In addition, each processor contains a mask register that can be set to inhibit execution of an instruction. PE's are connected to their near-neighbors to the north, east, south, and west. For data to be transmitted from one point of the array to another, it must be routed via a path linking both points through adjacent processors. The MPP is equipped with a built-in sum-OR tree which combines the output from all PE's in a tree of OR elements. This can be used to check for termination or convergence of a repeated sequence of instructions.

4. Parallel Region Growing

A parallel region growing strategy is affected by both the form of the homogeneity criteria and the constraints of the parallel processor architecture. In this section a set of representative local processing functions which can be used to realize a large number of homogeneity criteria are defined. Their implementation on the mesh-connected SIMD architecture, outlined in the previous section, is considered.

4.1. Parallel Region Growing Primitives

A general set of primitive operations, which can be used on a multi-processor system to implement parallel region growing algorithms, is described in this section. The strategy for mapping image points onto processors in a multi-processor system may vary from one architecture to another. Conceptually, a region is characterized by the values and spatial locations of its individual points, as well as the properties of all individual points combined. Therefore, a suitable representation of a region would be: a unique region id number assigned to all points in the region in conjunction with a region descriptor that contains all relevant global information needed to describe the region. Figure 4(a) illustrates the partition of an image, where each region in the partition is described using the representation suggested above. We assume that the region descriptor information is located within a given processor However, there is no guarantee or constraint that confines all image points within a region to a single processor.

The primitive operations required by a parallel region growing algorithm are as follows. First, in order to adequately characterize a region, a method to accumulate information from all region points to the region descriptor is needed. Second, in order to update region points concerning changes in status (i.e. region id due to merging), information held at the descriptor must be distributed out to all region points, particularly boundary points. Third, in order to interact with all neighboring regions in parallel, a method for exchanging information between all neighboring regions, sharing a common boundary is needed. These tasks are fundamental in a parallel region growing process and can be accomplished using the following set of primitives.

(1) **Reduction primitive:** reduces information from many points in a region to a single value and records it at the region descriptor. Typical reduction functions used are minimum, maximum, and sum.
(2) **Distribution primitive:** distributes a value from the region descriptor to all points in the region.
(3) **Exchange primitive:** exchanges information between points across a common boundary shared by neighboring regions.

Having defined a set of general parallel region growing primitives, we now consider their implementation on a mesh-connected SIMD architecture. When processing images on a highly parallel array of processors each pixel in the image can be directly mapped to a processor in the array. Initially each processor will only have information about a pixel's value and its spatial location in the image (this is straight forward with this architecture since a pixel's location in the processor array is a direct spatial mapping of its location in the image). To adhere to the region representation outlined above, as pixels are joined to form regions, a given PE within each region is designated to be the region descriptor (Fig 4(b)). What we require is a systematic way of choosing the descriptor PE and a method to efficiently implement the primitives described above. The nature of the primitives, particularly the reduction and distribution functions, suggest that a tree structure could be an efficient mechanism to incorporate into the region representation outlined thus far. This reasoning helped us arrive at the embedded tree data structure described in the following section.

4.2. The Embedded Tree Data Structure

4.2.1. Embedded Tree Description

An embedded tree structure is used to represent a region in an image. The tree structure is embedded within a region's boundaries on the PE array. Each region PE is assigned to a vertex in that region's embedded tree. A natural choice of PE to be designated as region descriptor is the PE located at the corresponding location of the tree root. What follows is a formal definition of a tree along with some additional terminology concerning trees.

Definition [9]. A **tree** is a directed acyclic graph satisfying the following properties:

(1) There is exactly one vertex, called the root, which no edges enter.

(2) Every vertex except the root has exactly one entering edge.

(3) There is a path (which is easily shown unique) from the root to each vertex.

A sample tree structure is presented in Figure 5(a). The arrows indicate the edge directions and will help to illustrate the following terminology.

Terminology [9]. If (v,w) is a directed edge of the tree, then v is called the *parent* of w and w the *child* of v. A vertex with no children is called a *leaf*. The *depth of a tree* is the length of the longest path from the root to a leaf. For example, in Figure 5, vertex a is the parent of vertices b, c, and d, and conversely vertices b, c, and d are children of vertex a. Vertices e, f, i, j, k, and l are all leaves, and the depth of the tree is 3.

Definition. An **embedded tree** is a tree confined within a specified boundary, and linked in such a way that all points within the boundary correspond to a unique vertex in the tree.

An example of an embedded tree is shown in Figure 5(b). Since each point in the image, or a region for that matter, is mapped to its own processor, we can think of the processors themselves as the tree vertices. Furthermore, edges of the tree correspond to interconnections between adjacent processors.

We chose the tree data structure because of its fan-in, fan-out nature and because it is easily extended to cover a two-dimensional arbitrary shaped region. The tree's fan-in and fan-out qualities make it very efficient for implementing the reduction and distribution primitives. By using a tree to link together the various pixels in a region, where the pixels form the vertices of the tree and the region descriptor is located at the tree root, it becomes possible to broadcast information from the region descriptors to other region pixels, or conversely, to accumulate region information at the descriptor PE's, within all regions simultaneously. What is important, however, is the way the tree is constructed. The primitives' efficiency is directly dependent on the depth of the tree. When processing regions of an image in parallel, the complexity of the functions employing the tree structure is bounded by the maximum tree depth of all regions in the image. Therefore, it is essential that the trees be constructed in such a way so as to minimize the tree depth of all regions.

4.2.2. Embedded Tree Generation

An iterative shrinking algorithm is used to create an efficient tree structure. This technique systematically removes pixels around a region's border and creates pointers linking the removed pixels to still existing neighboring pixels. The shrinking process continues until only a single pixel of each region remains. This remaining pixel is defined as the tree root or region descriptor. This strategy will centralize a tree root within a region.

The shrinking algorithm makes use of the near-neighbor connection scheme existing in the mesh. A pixel can be removed in the shrinking process only if certain conditions regarding its neighbors are true. This requires that each PE investigate its neighbors' values in order to make a decision on whether or not to "shrink". Initially all PE's in the array contain a binary one. As pixels are removed in the shrinking process these ones are changed to zeroes. Two things about a PE's neighbors need be known in order to determine whether or not the conditions required for shrinking exist, (1) their region id number and (2) their binary value. The set of conditions shown in Figure 6 pertain to the shrinking of the central pixel into its neighbor to the south. Shrinking may occur into any one of a pixels four near-neighbors. To determine whether a pixel may shrink to the west, north, or east, the conditions need to be adjusted accordingly. In one iteration of the shrinking algorithm the conditions are checked for each of the four neighbor directions in turn. Should a pixel satisfy the conditions in any one of these directions, its binary value is set to zero and a set of pointers are created. Each processor contains a pointer set which is encoded in four bits; one bit for each possible direction. The resulting embedded tree is represented by a double-linked set of pointers. A *parent-pointer* is created at the location of the removed pixel with a value corresponding to the shrinking direction. Similarly, a *child-pointer* is created at the neighboring PE, into which the pixel was shrunk, with a value corresponding to the direction of its new child. For example, if *parent-pointer[d]* is true for PE(i,j) then the pixel at (i,j) is linked to its parent in the direction corresponding to d. We can ascertain that at the parent node, *child-pointer[d]* will be true for the value of d corresponding to the opposite direction. Following the definition of a tree a PE may have up to four *child-pointers*, but never more than a single *parent-pointer*. The region PE with no *parent-pointers* is the root of the region tree, and those PE's with no *child-pointers* are designated as the tree leaves. The result of applying the tree generating algorithm is illustrated in Figure 7. The region pixels are linked together by the *parent-pointers*, shown in 7(c), created during the shrinking process. A corresponding set of *child-pointers* exist.

4.2.3. Regions with Holes

The algorithm just described cannot generate embedded trees for non-simply connected regions (i.e. regions with holes). Rather than converging to a single point, the algorithm converges to a single element wide, ring of region pixels which surround the enclosed region. This ring is similar to a medial axis of the region to which it belongs. The result of applying the shrinking algorithm to a region with a hole is shown in Figure 8(a). Since a root node must be chosen from the remaining pixels contained in the ring, it is most efficient to choose that pixel with maximum distance to the region boundary. This would help to minimize the height of the region's tree. Should two or more pixels possess the same height, the pixel with the highest id is selected, arbitrarily, to be the region root. The remaining pixels in the ring are then, systematically, linked to each other in the direction of the chosen root. The ring uncoupling is illustrated in Figure 8(b). Pixels neighboring the root are uncoupled from the ring and linked to the root first. Their neighbors in the ring are removed next, and so on, until all ring pixels have a linked path to the root. This solution will also work for regions with more than one hole.

Unfortunately, however, the presence of holes can lead to rather inefficient (deep) tree structures.

4.3. General Parallel Region Growing Algorithm

Region growing, as mentioned earlier, is achieved by merging regions to form larger regions that conform to a pre-defined homogeneity criterion. Using the embedded tree structure, two regions can be merged together by discarding the embedded trees of the individual regions and re-generating a single more efficient tree for the combined regions. Merging is performed in iterations and terminates when no more regions are able to merge without violating the homogeneity criterion. The parallel merging is accomplished by regions exchanging information with their neighbors and accumulating this information at the region descriptors for evaluation. Regions can then select a merge partner based on the homogeneity criterion being used. Although more than one neighbor may be suitable for merging, merging is performed in a hierarchical fashion giving preference to those regions that best satisfy the criterion and allowing them to merge first. The algorithm terminates when all possible merges have been performed.

The order of merging and a solution to the merge contention problem are important aspects in the parallel approach. It is not possible for a region to merge with more than one other region during a single merging iteration. The resulting region could otherwise be in violation of the homogeneity requirements. Most contentions are resolved by allowing regions that are "most homogeneous" to merge first. However, should a situation arise in which three regions want to merge with each other (all with equivalent homogeneity values) the contention is broken by forcing each region to select only one merge partner; that one could be arbitrarily chosen as the region with the highest id. Two common merge contention situations are illustrated in Figure 9. Merging is not performed unless both regions select each other as merge partners. This insures that no more than two regions from a group will select each other to merge with. A region that was unable to merge during a given iteration because its selection was not mutual, may succeed in a subsequent iteration.

5. Algorithm Complexity

The region growing algorithm complexity depends upon both the implementation of the region growing primitives and the number of iterations required to arrive at the desired result. The cost of implementing a region growing primitive, Embed, Dist, or a reduction, is $O(d)$ where d is the maximum distance across any region. The Exchange primitive is implemented in constant time.

The speed of the primitives also depends somewhat on the region shape and topology. Convex blob-like regions containing $O(d^2)$ elements are processed in $O(d)$ time. Non-simply connected regions require slightly more time than simple regions due to the additional time taken to compute the embedded tree. The complexity for these regions is still $O(d)$ and, in general, the cost increase will be less than 2. Strange concave shapes, such as spirals, require the most time. Here d is the longest path between two pixels in a region which does not go outside the region. However, these regions rarely occur in practice. The ideal cost complexity is $O(\log d)$ but, this cannot be achieved because of routing limitations on a mesh-connected architecture.

The number of iterations also depends upon the region dimension d. In an ideal situation, merging of a two-dimensional region can be achieved in $O(\log d)$ parallel merges starting with each pixel as a region; however, if only one merge occurs during each iteration then the worst case complexity is $O(d)$.

There are two heuristics which can greatly reduce the number of merge iterations. The first is the split phase which was mentioned earlier. The split algorithm only requires $O(\log d)$ computations for the region growing primitives (although routing is still $O(d)$) and $O(\log d)$ iterations to complete. Second, in many practical applications there is a "background" behind a collection of objects. This background, being a single region, could dominate the algorithm cost. However, in most cases it is possible to remove the background from consideration by a fairly simple thresholding technique. The algorithm cost then depends on the span of the largest object which remains.

6. Parallel Split and Merge on the Massively Parallel Processor

A parallel split and merge algorithm has been implemented on the Massively Parallel Processor (MPP), using the parallel region growing technique outlined in the previous sections.

6.1. The Parallel Split and Merge Algorithm

In order to implement the parallel region growing algorithm, a procedure for generating the embedded tree data structures described in section 4.2 and a set of primitives, closely resembling those outlined in section 4.1, were designed. These primitives are as follows.

EMBED(ridm, pptrs, cptrs, maxtd):

This procedure receives as input the region id matrix, **ridm**, and performs the region shrinking and embedded tree generation. It returns a matrix of parent and child pointers (**pptrs,cptrs**), and a value proportional to the maximum tree depth, **maxtd**. Region id zero is reserved to mask out regions that need not be processed.

MIN(pptrs, maxtd, pixval, rootval):

MAX(pptrs, maxtd, pixval, rootval):

These procedures receive as input the tree parent pointers, **pptrs**, the maximum tree depth, **maxtd**, and the pixel values, **pixval**, and perform reduction functions. They deliver the region minimum and maximum, respectively, to the region roots (**rootval**).

DIST(cptrs, maxtd, rootval, pixval):

This procedure receives as input the tree child pointers (**cptrs**), the maximum tree depth (**maxtd**), and the values contained at the region roots (**rootval**) and distributes them throughout the regions (**pixval**).

EXCHANGE(ridm, pixval, direction, neighbval):

This procedure receives as input the region id matrix (**ridm**) and the pixel values (**pixval**) to be exchanged across the region border. The parameter **direction** is needed to discriminate between neighboring regions bordering in different directions. The pixel values of neighboring regions are returned in **neighbval.**

Since all regions are square in the split phase of the algorithm, it is not necessary to create embedded trees to represent the regions. On the MPP the regions in the split phase will have dimensions mxm where m is a power of 2. Hence, the upper-left corner PE of each region can be designated as the region root. Then, by initially treating each pixel in the image as an independent region, an iterative merging technique can be used to create larger square regions comprised of four subregions (one in each quadrant). This has the same effect as a top-down split but is more efficient. Using the homogeneity criterion suggested earlier, eqn (2.1), all information needed to describe a region can be accumulated at the root during the region growing process. In effect, a larger region's properties (e.g. minimum and maximum values) are obtained from the properties of its four subregions. There is no need to re-evaluate the minimum or maximum values of all pixels in a region, but simply to calculate the minimum and maximum of the values held at the roots of the four subregions. A larger region is created by merging its four quadrants only if all four quadrants are homogeneous regions and the combined properties of the subregions satisfy the homogeneity criterion. Hence, reduction operations only require a constant number of steps to perform and the distribution operation can be performed in *O(log(m))* steps. However, since the MPP only has near-neighbor connections the cost of routing is still *O(m)*. This could be reduced if a more complex interconnection network between PE's were available (e.g. *O(log(m))* given a hypercube interconnection). In any case, this is much more efficient than using the embedded trees, and only requires a slight modification of the parallel primitives described above. The modified primitives for square regions are described below.

SQRMIN(iteration,pixval,rootval):
SQRMAX(iteration,pixval,rootval):

These procedures receive as input the splitting iteration number to determine the dimensions of the regions being processed and the locations of the region roots. The region minima and maxima are calculated by first shifting the minimum and maximum values of the southwest and southeast quadrants to the roots of the northwest and northeast quadrants, respectively. Then, the partial result at the northeast quadrant is shifted left to the northwest quadrant root (now also the square's root) to determine the square's minimum and maximum. The results are returned in **rootval.**

SQRDIST(dimensions,rootval,pixval):

The parameter **dimensions** is an array containing the dimensions of each region. This information is used to determine how far down, and then across, the value at the root needs to be broadcast to reach all region PE's. The result is returned in **pixval.**

Both phases of the split and merge algorithm are described below. Each phase of the algorithm is first outlined by dividing it into a series of steps. The corresponding pseudo-code for each phase is then also broken down into the same steps. The homogeneity criterion suggested earlier, eqn.(2-1), is used.

The Split phase algorithm

1. Since initially all pixels are treated as independent regions, the region minimums and maximums are set to equal the pixel values and the split iteration is set to one.

2. Each region's maximum and minimum values are computed.
3. The region ranges are calculated at the roots.
4. If any regions display a range within the allowed threshold, the new region dimensions are set at the roots.
5. The iteration value is incremented and, unless either the iteration value exceeds log(n) (n is the image dimension) or no merges occurred during the last iteration, control is returned to step 2.
6. The regions' root PE id's are distributed to all region PE's using the region dimensions contained at the region roots.

The Split phase pseudo-code

```
     Comment: Variable declarations:
     idm, pid, rootnode : PARALLEL ARRAY [1..128,1..128] OF [0..16383];
     pixval, regmin, regmax, T, range : PARALLEL ARRAY [1..128,1..128] OF [0..255];
     dimension : PARALLEL ARRAY [1..128,1..128] OF [1..128];
     iteration : [1..7];
     merge : BOOLEAN;

1.   Comment: Initialize region maximum and minimum values to the
     Comment: pixel values,pixval. Set the split iteration to one.
     regmin ← pixval;
     regmax ← pixval;
     iteration ← 1;

2.   REPEAT
       Comment: The region minima and maxima are calculated by
       Comment: propagating the minimum and maximum values of
       Comment: the subregions to the square's root.
       SQRMIN(iteration, regmin, regmin);
       SQRMAX(iteration, regmax, regmax);

3.     Comment: Each region's range is calculated at the region root.
       Comment: Note that the region rootnode id's can be calculated
       Comment: as a function, f, of the iteration and the unique
       Comment: processor id's (pid).
       rootnode ← f(pid,iteration);
       WHERE rootnode DO
         range ← regmax- regmin;

4.     Comment: If any range is within the homogeneity threshold, T,
       Comment: then set the region dimension at the region root.
       merge ← ANY(range < T);
       IF merge THEN
         WHERE ((range < T) AND rootnode) DO
           dimension ← $2^{iteration}$;

5.     Comment: The iteration value is incremented and the program
       Comment: loops back to step 2 unless either there were no
       Comment: merges during the last iteration or the iteration
       Comment: value has exceeded its limit.
       Comment: Splitting is performed iteratively with the splitting
       Comment: dimension of the sub-quadrants increasing by a factor
       Comment: of 2 at each subsequent iteration. Hence a maximum of
       Comment: log(n) iterations are needed (n is the image dimension).
```

```
        iteration ← iteration + 1;
    UNTIL ((iteration > log(n)) OR NOT merge);

6.  Comment: Once all possible quadrants have merged the regions are
    Comment: assigned an id equivalent to the region's root PE id
    Comment: This value, held in pid is distributed from the
    Comment: root. The region dimension matrix, dimension is
    Comment: used to control the broadcasting of this value to all
    Comment: region PE's in idm.
    SQRDIST(dimension, pid, idm);
```

The Merge phase algorithm

1. Embedded trees for all regions are constructed using the region id's.
2. Minimum and maximum values and region id's are exchanged between neighboring regions, to determine the range of combined regions, and possible merges.
3. Lowest merge ranges are accumulated at the region roots and distributed to all region PE's.
4. In case more than one neighboring region yields the same "lowest merge range", the region with the higher id is selected.
5. Choices of merge selection are exchanged with neighbors. If two regions select each other, both regions merge by assuming the smaller id of the two.
6. Unless no merges occurred during the last iteration, control returns to step 1.

The Merge phase pseudo-code

```
Comment: Variable declarations.
ridm, regmins, regmaxs, mergeneighbid, mergechoices, mergechoice,
neighbmergechoice : PARALLEL ARRAY [1..128,1..128] OF [0..16383];
pixval, mintoroot, maxtoroot, mergerange, neighbmin, neighbmax, newmergerange,
bestrangeval : PARALLEL ARRAY [1..128,1..128] OF [0..255];
cptr, pptr : PARALLEL ARRAY [0..5,1..128,1..128] OF BOOLEAN;
merger : PARALLEL ARRAY [1..128,1..128] OF BOOLEAN;
maxtd, i : INTEGER;

1.  Comment: create embedded trees for all regions using
    Comment: the region id matrix, ridm. Then determine each
    Comment: region's minimum and maximum from the pixel values,
    Comment: pixval, and distribute these out to all region pixels as
    Comment: regmins and regmaxs.
    EMBED(ridm, pptrs, cptrs, maxtd);
    MIN(pptrs, maxtd, pixval, mintoroot);
    MAX(pptrs, maxtd, pixval, maxtoroot);
    DIST(cptrs, maxtd, mintoroot, regmins);
    DIST(cptrs, maxtd, maxtoroot, regmaxs);

2.  Comment: By initializing the mergerange as the homogeneity
    Comment: threshold, T, only neighbors yielding an acceptable
    Comment: merging threshold are considered.
    mergerange ← T;
    Comment: All four neighboring directions are examined and
    Comment: both minimum and maximum values as well as the
    Comment: neighbors id are obtained.
    FOR i ← 1 TO 4 DO
```

```
BEGIN
  EXCHANGE(ridm, regmin, i, neigbmin);
  EXCHANGE(ridm, regmax, i, neigbmax);
  EXCHANGE(ridm, ridm, i, neighbid);
  Comment: In order to determine the range of a possible
  Comment: merge between a region and its neighbor, the
  Comment: extremes of both regions minimums and maximums
  Comment: must be evaluated.
  WHERE neighbmax < regmax DO
    neighbmax ← regmax;
  WHERE neighbmin > regmin DO
    neighbmin ← regmin;
  newmergerange ← neighbmax-neighbmin;
  Comment: As each neighboring direction is examined,
  Comment: the merge range and neighbor id of the
  Comment: neighbor yielding the best result is retained.
  WHERE (newmergerange < mergerange) AND (ridm <> neighbid) DO
  BEGIN
    mergerange ← newmergrange;
    mergeneighbid ← neighbid
  END;
END;

3.    Comment: The best (smallest) range value of all ranges retained
      Comment: by border elements in mergerange is delivered to the
      Comment: region root and re-distributed to all region elements
      Comment: in bestrangeval.
      MIN(pptrs, maxtd, mergerange, mintoroot);
      DIST(cptrs, maxtd, mintoroot, bestrangeval);

4.    Comment: In order to arbitrate ties between neighboring
      Comment: regions that yielded the same merge range, the
      Comment: region with the largest id is chosen.
      WHERE bestrangeval = mergerange DO
        mergechoices ← mergeneighbid;
      MAX(pptrs, maxtd, mergechoices, maxtoroot);
      DIST(cptrs, maxtd, maxtoroot, mergechoice);

5.    Comment: Once again all four neighbor directions are
      Comment: polled to locate the neighbor chosen for
      Comment: merging. A merger exists where both regions
      Comment: have chosen each other for merging.
      FOR i ← 1 TO 4 DO
      BEGIN
        EXCHANGE(ridm, mergechoice, i, neighbmergechoice);
        EXCHANGE(ridm, ridm, i, neighbid);
        WHERE (ridm = neighbmergechoice) and (mergechoice = neighbid) DO
          merger ← true;
        haveamerge ← ANY(merger);
        Comment: If a merger exists then the smaller id of both
        Comment: regions is selected and this new id is
        Comment: distributed to all region pixels.
        IF haveamerge THEN
        BEGIN
          WHERE merger AND (neighbid < ridm) DO
        ridm ← neighbid;
      MIN(pptrs, maxtd, ridm, mintoroot);
      DIST(cptrs, maxtd, mintoroot, ridm);
    END;
  END;
```

6.2. Results

A high-level language (Parallel Pascal [12]) implementation of the region growing primitives was run on the MPP. A series of timing tests were conducted using images of square NxN regions with the set of dimensions: N = 8, 16, 32, 64 (Table 1). These test sets were representative of blob-like regions with d=N.

Table 1. Measured Timing Results for Primitives

Dimension	Time (ms)			
N	embed	min/max	dist	exchange
8	12.5207	1.9675	1.5291	0.0660
16	22.2983	3.5137	2.7260	0.0660
32	41.8582	6.6043	5.1177	0.0660
64	80.9739	12.7876	9.9032	0.0660

The following set of timing expressions, given in milliseconds, were derived from these results.

$T_{embed}(N) = 1.222N + 2.741$

$T_{min/max}(N) = 0.1932N + 0.4210$

$T_{dist}(N) = 0.1495N + 0.3322$

$T_{exchange}(N) = 0.0660$

The measured values do not deviate from the above equations by more than 2 percent.

An estimated performance analysis was conducted assuming an optimal machine-level encoding of the primitives. The clock cycle time for the MPP is 100 nanoseconds. Furthermore, it was assumed that loop set-up in the host could be overlapped with array computations. This is possible on the MPP since a FIFO buffer exists between the host and the PE array which allows the host to perform serial calculations while the array is busy processing instructions in the buffer. The analysis yielded the following expressions.

$\hat{T}_{embed}(N) = 0.1570N + 0.0197$

$\hat{T}_{min/max}(N) = 0.0168N + 0.0034$

$\hat{T}_{dist}(N) = 0.0104N + 0.0034$

$\hat{T}_{exchange}(N) = 0.0358$

The estimated results range between 2 to 15 times faster than the measured results. This is probably due to two main reasons. First, the Parallel Pascal code generator produces inefficient code since it currently does not perform any code optimizations; second, the host may not be able to generate instructions fast enough to keep the processor array busy; especially for boolean data operations. (Conditional branches which are dependent on processor array values cause the FIFO buffer to empty, however no such branches occur in the implementation of the primitives).

A similar comparison analysis was conducted for the split and merge phases of the algorithm. The measured results for the blob-like regions are summarized in Table 2.

Table 2. Measured Timing Results for Split and Merge Phases

Dimension	Time (ms)	
N	Split	Merge
8	2.7276	38.1122
16	3.4329	67.1308
32	4.2128	125.1579
64	5.2824	241.2121

These results yielded the following expressions.

$T_{split}(N) = 0.01518N + 0.5839\log(N) + 0.8077$

$T_{merge}(N) = 3.627N + 9.104$

The first term in the split phase expression represents the cost due to the shift operations needed to route information through an NxN mesh connected region. This could be reduced to *O(logN)* given a hypercube interconnection scheme. The second term in the expression is related to the number of arithmetic operations performed to create an NxN region. The third term includes the overhead operations that are independent of N. The merge phase results are primarily a product of the primitive operations and therefore yield an expression of *O(N)*. The estimated performance using an optimal encoding is as follows.

$$\hat{T}_{split}(N) = 0.0045N + 0.100\log N + 0.052$$

$$\hat{T}_{merge}(N) = 0.3746N + 0.3737$$

For the split phase the estimated performance was three to five times faster than the measured results. Since the merge phase of the algorithm is highly dependent on the parallel primitive implementation, we could predict that the estimated performance will be approximately ten times faster than measured results. This is in fact the case as demonstrated by the expression above.

7. Conclusion

A technique for region growing on a highly parallel SIMD computer has been described. Three fundamental primitives for region growing have been defined: reduction, distribution, and neighbor exchange. Furthermore, it has been shown that the split algorithm on the target architecture is efficiently implemented by region growing using special primitives for processing square regions.

A simple homogeneity criteria, pixel range, has been used to demonstrate the basic parallel techniques. However, arbitrary complex homogeneity functions can be computed with this scheme. Additional reduction functions such as PRODUCT and MEAN may be implemented with a similar efficiency to the primitives MAX and MIN.

An embedded tree structure has been introduced for rapid primitive computation on a mesh-connected SIMD system.

REFERENCES

[1] D. H. Ballard and C. M. Brown, *Computer Vision*, Prentice-Hall, Englewood Cliffs, New Jersey, (1982).

[2] A. Rosenfeld and A. Kak, *Digital Picture Processing*, Academic Press, New York, New York, (1982).

[3] S. W. Zucker, "Survey, region growing: childhood and adolescence," *Computer Graphics and Image Processing* 5, pp. 382-399, (1976).

[4] S. L. Horowitz and T. Pavlidis, "Picture segmentation by a directed split-and-merge procedure," *Proceedings 2nd IJCPR*, pp. 424-433, (August 1974).

[5] J. P. Gambotto and O. Monga, "A parallel and hierarchical algorithm for region growing," *Proceedings of the 1985 IEEE Conference on Computer Vision and Pattern Recognition* pp. 649-652 (1985).

[6] A. P. Reeves, "Survey, parallel computer architectures for image processing," *Computer Vision Graphics and Image Processing* 25, 68-88 (1984).

[7] A. P. Reeves, "Highly parallel computer architectures for scientific applications,"

[8] J. Kittler and M. J. B. Duff, *Image Processing System Architectures* , Research Studies Press, Letchworth, England, (1985).

[9] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms* , Addison-Wesley, Reading, Mass., (1974).

[10] A. P. Reeves, "A systematically designed binary array processor," *IEEE Transactions on Computers* C-29, pp 278-287 (1980).

[11] K. E. Batcher, "Design of a massively parallel processor," *IEEE Transactions on Computers* C-29(9) pp.836-840 (September 1981).

[12] A. P. Reeves, "Parallel Pascal: an extended Pascal for parallel computers," *Journal of Parallel and Distributed Computing* 1 pp. 64-80 (1984).

[13] R. W. Taylor, M. Savini, A. P. Reeves, "A fast algorithm for range image segmentation," submitted to *1987 IEEE Systems Man and Cybernetics Annual Conference* (October 1987).

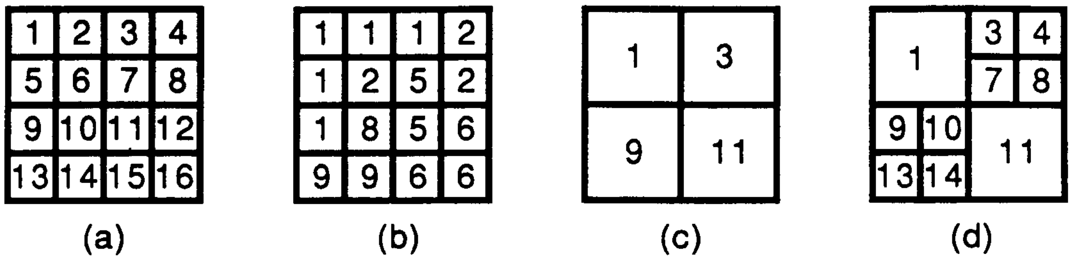


Figure 1. Split and merge splitting phase. (a) Pixel id numbers. (b) Pixel values. (c) Region id's after first split. (d) Region id's after second and final split.

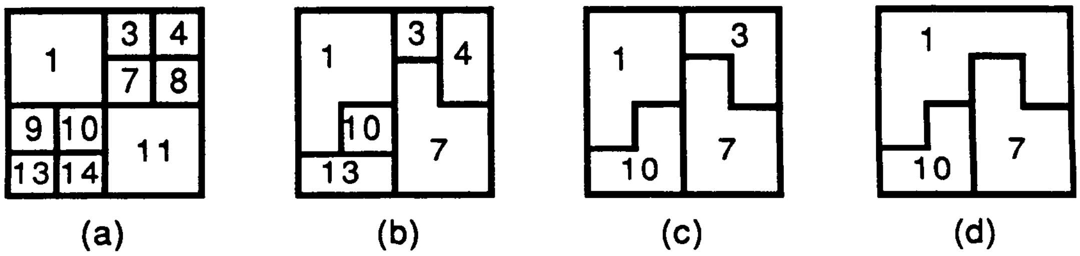


Figure 2. Split and merge merging phase. (a) Region id's after splitting phase. (b) Region id's after first merge. (c) Region id's after second merge. (d) Region id's after third and final merge.

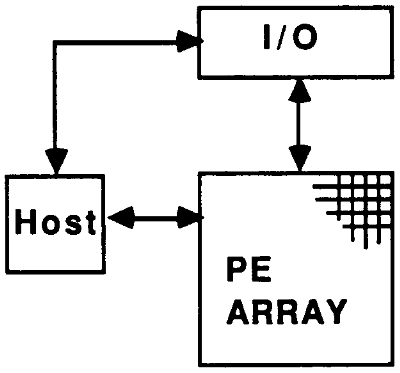


Figure 3. Mesh-connected SIMD architecture.

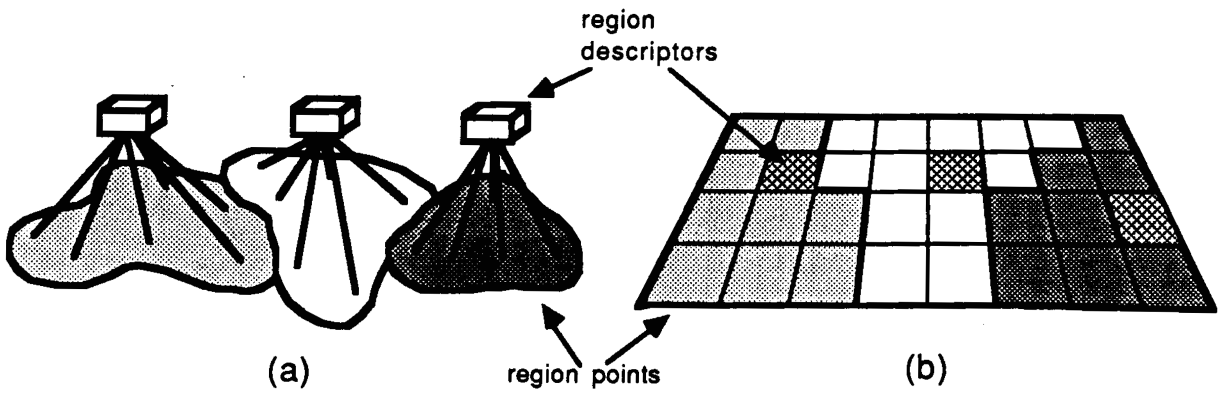


Figure 4. Region representation. (a) A generalized representation of a region consists of a spatial distribution of the region points relative to their location in the image, and a corresponding region descriptor for each region. (b) A region representation on a mesh of processing elements consists of each region pixel mapped to its own PE and a designated region PE to serve as the region's descriptor.

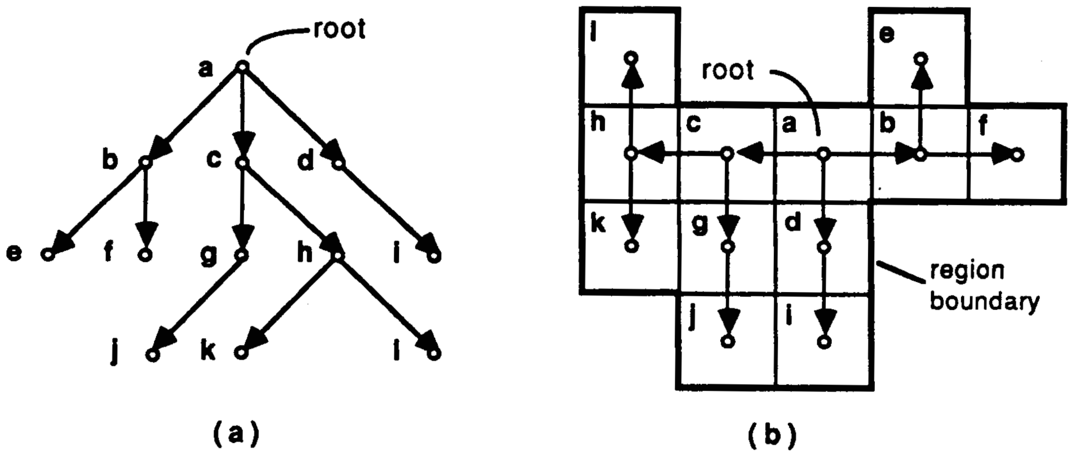
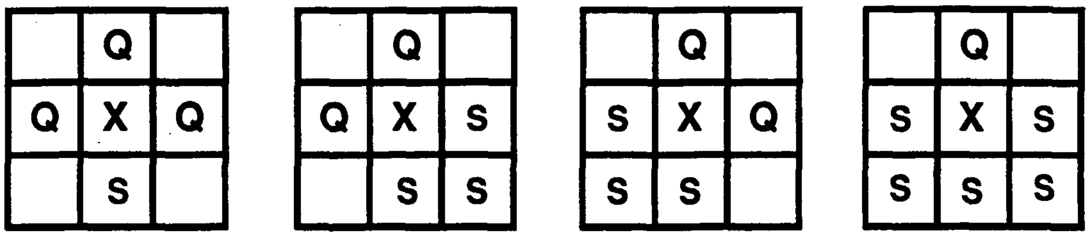


Figure 5. (a) Tree data structure. (b) Embedded tree data structure.



S- pixel of same region and of binary value one

Q- pixel of different region or pixel of binary value zero

Figure 6. In order for pixel X to shrink to the south one of the near-neighbor arrangements shown must exist.

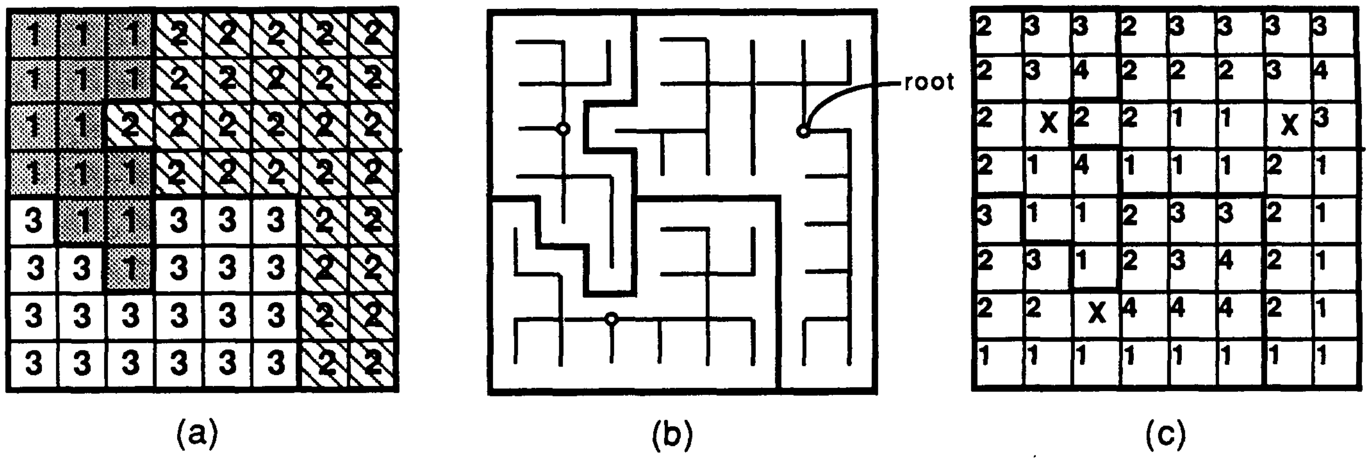


Figure 7. Embedded tree data structure. (a) Regions are represented by a unique id number. (b) Embedded trees created by shrinking algorithm. (c) Parent-pointers using the following encoding: 1=N, 2=E, 3=S, 4=W.

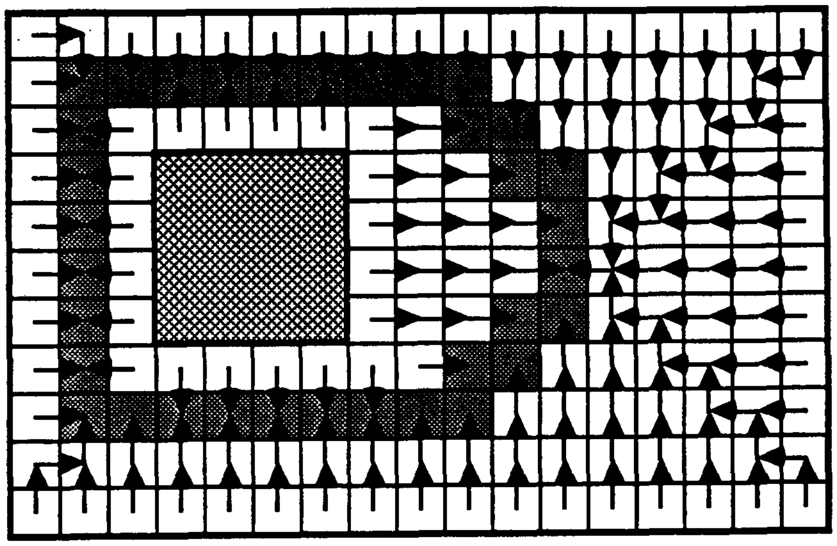


Figure 8(a). The result of the shrinking algorithm performed on a region with a hole is a ring in the shape of a medial axis.

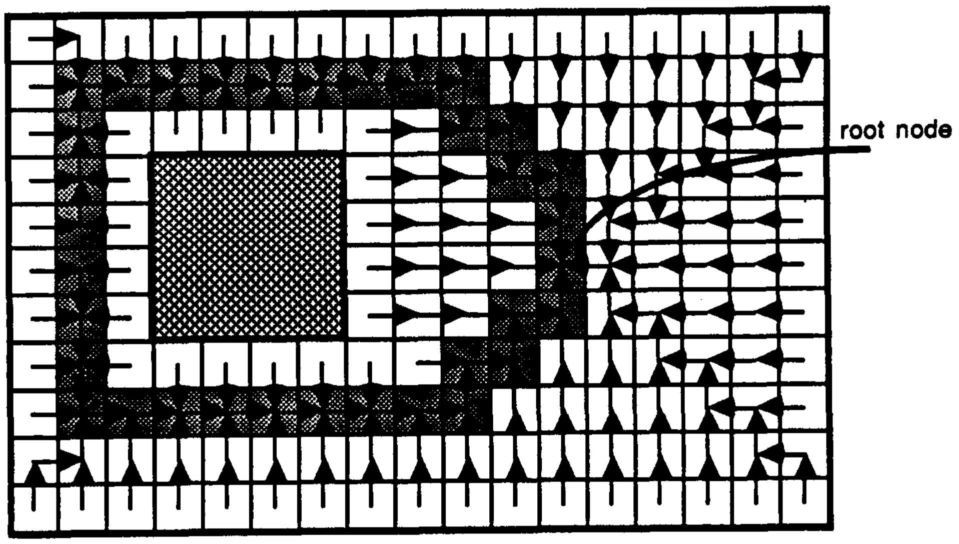


Figure 8(b). The ring is uncoupled in such a way that the region root is located at the point on the ring that is furthest from the boundary.

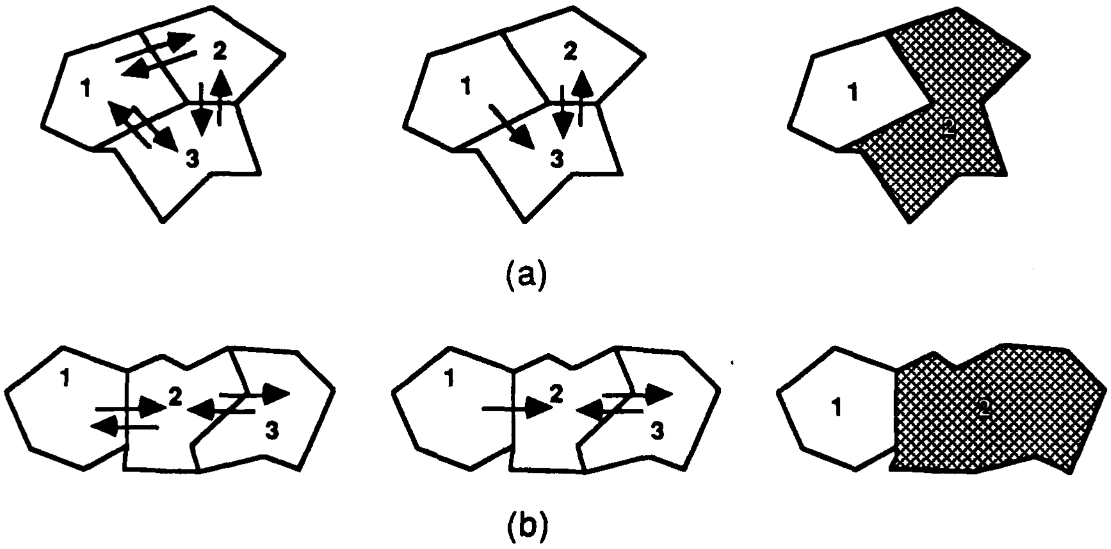


Figure 9. Merging order. Contention for merging is indicated by the arrows in the diagram. First, regions choose a neighbor according to the homogeneity criterion. Second, in case of a tie, they select the region with the higher id. Finally, regions are allowed to merge only if the selection is mutual. (a) and (b) illustrate two different cases that may arrise.

APPENDIX B: ALGORITHMS AND PERFORMANCE ANALYSIS FOR THE MASSIVELY PARALLEL PROCESSOR

ALGORITHMS AND PERFORMANCE ANALYSIS FOR THE MASSIVELY PARALLEL PROCESSOR

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

Maria Clara Gutierrez

January 1988

BIOGRAPHICAL SKETCH

Maria Clara Gutierrez was born in Bogota, Colombia in November 28, 1962 to Hugo and Bertha Gutierrez. She received a Bachelor of Science in Electrical Engineering with high honors at The University of Texas at Austin in May 1985. Subsequently, she came to Cornell University to pursue the Master of Science degree. She did her graduate work under the guidance of Professor A. P. Reeves. She received her degree in January 1988.

DEDICATION

To my family, especially to my father and mother.

ACKNOWLEDGMENTS

I would like to thank Professor Anthony P. Reeves, my advisor, for his guidance and support which made this thesis possible, and Professor Frank T. Luk for agreeing to be in my committee and for his encouragement.

I would like also to specially thank my husband Vernon Marchal and my family for their continuous love, encouragement, and support through sometimes difficult times.

I thank also everyone at Phillips 210 and 401 for their help and for the nice times that we had.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1. Introduction to Parallel Computers

Computers play an increasingly prominent role in scientific research; without them, science and technology would not have reached the levels of development of today. Still, many research problems exist which require very large computational capabilities several orders of magnitude greater than what is now available from conventional computers. See [1] [2] and [3] for discussions of such research problems. For example, in the area of computational aerodynamics, Peterson ([1] p. 68) states that future research will require computers with sustained rates of approximately 1 to 1000 billion floating point operations per second (FLOPS).

The fastest conventional super computers, such as Cray with a rate of 10^8 FLOPS (100 MFLOPS), achieve their high performance by increasing the speed of the electronic components and logic circuits, and by introducing pipelining and parallelism at the circuit level. However, if speed improvements are only obtained through progress in technology, it is not likely that the required sustained rates, in the order of billions of FLOPS, will be achieved.

The computational capabilities of computers can be increased by several orders of magnitude, through the introduction of parallelism at all levels of the computer architecture, including the processor level. The concept of parallelism in computers is not new (according to Hockney and Jesshope [4] it goes as far back as 1842 in a publication entitled: "Sketch of the Analytical Engine Invented by Charles Babbage"), but its implementation has just recently been made possible with recent advances in VLSI technology and the resulting enormous improvements in processing, storage, and communications. The result has been the development of the Parallel Computer, which may consist of hundreds or thousands of processors connected together, and operating simultaneously to solve a given problem.

Even though technology advances have made parallel computers possible, before they can be effectively used, a number of problems must be addressed. In addition to the usual design problems of conventional computers, several other problems, particular to parallel systems, include: the number of processors, fault tolerance, programability, load balancing and the architecture of the interconnection network. Of these, the last problem is crucial, for the interconnection network (IN) must match the communication needed by the algorithm being solved, otherwise the processors may spend a considerable amount of time communicating among each other rather than doing useful work.

The optimal IN is the crossbar where every processor is connected to every other one. Unfortunately, this IN is not practical for a large number of processors because its cost complexity is $O(n^2)$, where n is the number of processors, and thus, the crossbar IN is too expensive for large n. Another option for the IN is a ring topology where the processors may be considered to be arranged in a circle, such that each processor is connected to its adjacent two neighbors. Although this IN is inexpensive, $O(n)$, for most applications it would be inefficient and unreliable. Several IN organisations exist which fall between these two extremes, in cost and performance. One such IN is the mesh-connected network: it is easily implemented, $O(n)$, and it fits the communication requirements of a large number of algorithms, especially in the area of image processing.

An existing parallel computer that uses a mesh-connected network is the Massively Parallel Proccesor (MPP). This thesis is concerned with the development and performance analysis of several important algorithms which have been developed for the MPP. The effectiveness of the near-neighbor IN will be studied with respect to these algorithms. Of particular interest is the amount of time which is spent on interprocessor communication rather than data computation.

In the following chapters a number of important algorithms, which have been developed for the MPP, are described and their performance on the MPP is discussed. Chapter 2 presents several regular permutations: the *exchange*, *shuffle*, *butterfly*, *bit reversal*, and where applicable, the *sub* and *super* permutations. In addition, an FFT program, which uses some of these permutations, is presented. Chapter 3 presents a heuristic mapping algorithm for irregular permutations, using near-neighbor, bilinear, and cubic interpolation. This algorithm was run with two different irregular permutations or image warping operations: Rotation with translation and 'Fisheye' warp. Chapter 4 presents a convolution operation, and two pyramid algorithms: Gaussian and Laplacian filtering. These algorithms build the pyramid from the image located at its lowest level. In the resulting pyramid, each level is the filtered copy of the adjacent lower level; the filtering is obtained with convolution operations. Chapter 5 presents conclusions. The remainder of this chapter will present an overview of the MPP's hardware and software, and the optimal and measured execution times of several primitive operations of Parallel Pascal, the high level language of the MPP. These primitive operations are used in the above mentioned algorithms.

1.1. MPP Hardware

The MPP was designed to be a high speed computer for image processing, but it is currently also used for several other applications. For a listing of current research projects on the MPP see [5]. According to Flynn's taxonomy [4], the MPP is an SIMD computer. This taxonomy classifies computers in four classes, of which two are important for parallel computers:

1. SIMD (Single Instruction, Multiple Data): In this class all the processors execute the same instruction stream but on different data.

2. MIMD (Multiple Instruction, Multiple Data): the processors execute different instruction streams on different data.

The MPP involves an array of 128×128 (16,384) single bit processing elements (PE's). The MPP is an SIMD computer because these PE's operate synchroneously; the same instructions are applied to different data with a cycle time of 100 nsec. Figure 1.1 is a block diagram of the MPP. It consists of four main components: The array unit (ARU), which processes a two dimensional (2d) 128×128 array of data, is controlled by the array control unit (ACU). The data is sent to the ARU through the Staging Memory. Finally the Host computer is the front end from which the user controls the MPP. The following will be an overview of these components. Refer to [6] and [7] for more detailed information.

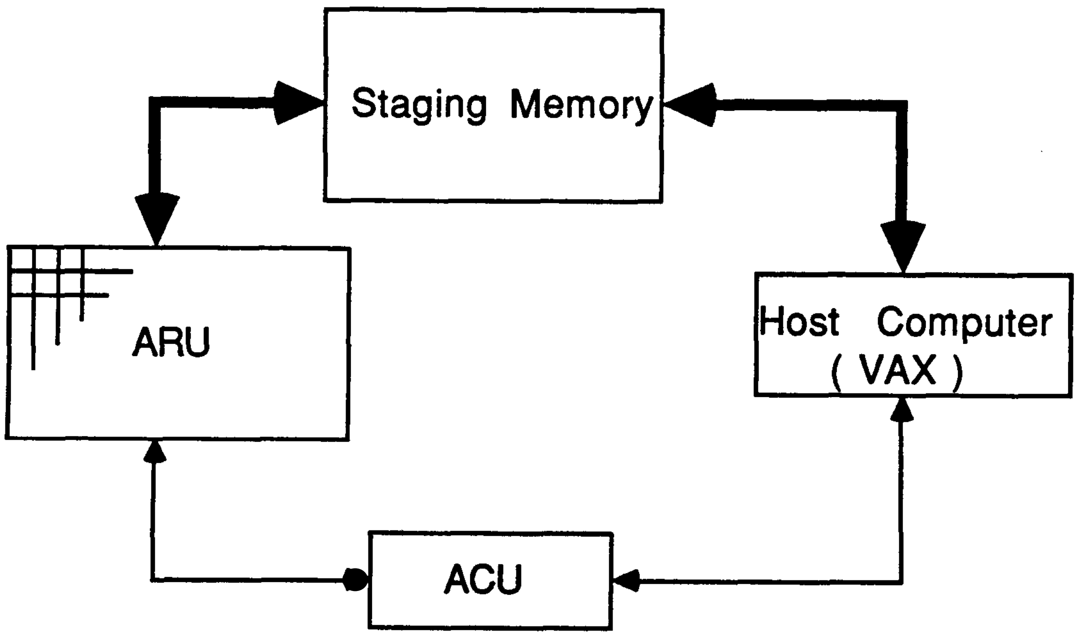


Figure 1.1. Block Diagram of the MPP.

The ARU is organized as a number of 2d planes. Each plane consists of 16,384 bits, i.e. 128 rows by 128 columns. The memory consists of 1,024 planes and an array of data is stored vertically across several planes. For example an 128x128 array of 8 bit elements would be stored in eight consecutive memory planes. The 36 other planes form the PE's; the block diagram of a PE is shown in Figure 1.2 ([6] p. 209).

A PE is composed of four subunits:

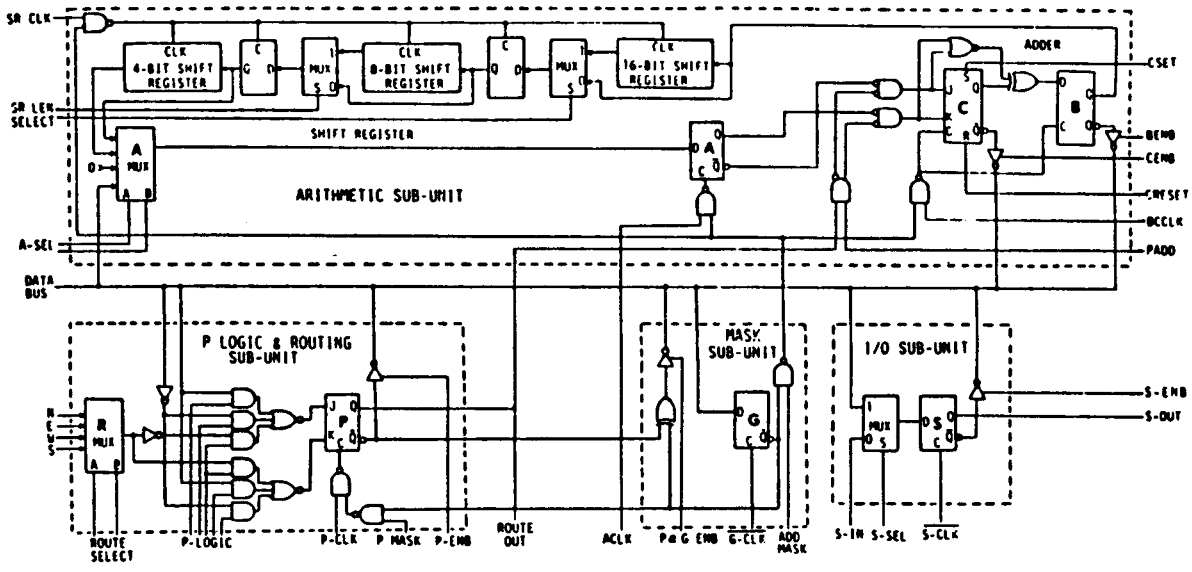


Figure 1.2. MPP Processing Element.

- The Arithmetic unit with a variable length planar shift register of up to 30 processing planes, and the A, B, and C planes used for the bit-serial arithmetic operations;
- The Logic and Routing unit with the P plane;
- The Mask unit masks the activity in the other processing planes. If a bit in the G plane is set, the operation in the corresponding bit of the processing plane is executed, otherwise it won't be;
- The I/O subunit handles the I/O of the ARU with the S plane.

The **ACU** is divided into three independent control units (Refer to Figure 1.3): First, the **Main Control Unit** (MCU) runs the main MPP application program which is stored in the Main control memory. The MCU performs all the scalar operations and initates array operations by entering calls on the Call queue. It also initiates I/O operations. Second, the **Processing Element Control Unit** (PECU) controls the processing planes of the ARU. The PE Control Memory stores the instructions and the Call queue stores calls from the MCU. These calls tell the PECU which instructions the ARU executes next. Third, the **I/O Control Unit** controls the operations of the S plane of the ARU. All three control units operate simultaneously to allow overlapping of their operations.

The **Staging Memory** is in the data path between the ARU and the front end computer. It is needed for reformating the data since the front end computer operates on an array of data an item at a time and the ARU operates a plane at a

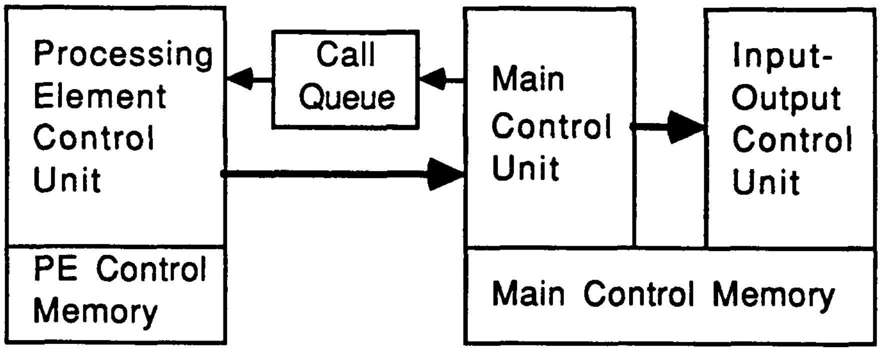


Figure 1.3. Array Control Unit.

time. The Stager also serves as temporary storage since for some applications the ARU memory of 1,024 planes is not enough.

The **Front End Computer** is the interface between a user and the MPP. This Host computer is a VAX-11/780 with a VMS operating system. When a program is run, the program sections preceded with a {$h+} directive are executed by the host, and the sections preceded with a {$h-} are executed by the MCU. Normally there is only one {$h+} section and it is used to start the MPP. It should be noted that the host is shared with other users which can affect its speed.

1.2. MPP Software

The MPP's high level language is Parallel Pascal [8]. All the algorithms presented in this thesis were developed in this language. Parallel Pascal is a superset of the Pascal programming language. Briefly, the extensions of parallel pascal involve the following:

- A new data-type array called 'parallel array'. This array is to be used in the ARU.
- The where-do-otherwise statement which is a parallel extension of the conditional statement if-then-else. The condition is a parallel array of Boolean.
- The extension of all conventional expressions to array data types. For example, suppose that A, B, and C are 2d arrays of the same size. To add A to B and to store the result in C, all that is needed is the expression:

  C := A + B
- Built in data manipulations functions which include: data reduction (*or, and, min, max, sum, product*), data permutation (*shift, rotate, transpose*), and data broadcast (*expand*) primitives.

The development of a MPP Parallel Pascal program involves several operations(see Figure 1.4). First, a given algorithm is encoded in a Parallel Pascal procedure or function; this program is stored in a library file. Then, a test program using this procedure or function, is developed. The test program declares the library subprogram as an *external* procedure or function as follows:

{$ *Library_name.pl* }

procedure procedure_name(list of formal parameters); **extern**;

The Extern program, a library preprocessor, replaces the external call with the source code of the procedure stored in the library file. The program is now ready to be compiled and run. It is very helpful at this point, especially for a remote MPP user, to test and debug the program with the aid of the Parallel Pascal Translator (PPT). PPT translates a Parallel Pascal program into conventional Pascal program. Thus, with PPT, a Parallel Pascal program can be run and tested in any computer that supports the Pascal language.

Once the Parallel Pascal program has been debugged, it is ready to be compiled and run in the MPP. Several steps are involved in the compilation of a program in the MPP (see Figure 1.4). Refer to [9] for more detailed information.

First, the Parallel Pascal compiler reads the Parallel Pascal program and produces a P-code file (PCD) and a listing file (LIS). The code generator program then reads the P-code file and produces the VAX macro assembly code (MAR) and the MCU assembly code (MCL). The code generator program also outputs a code generator debug information file (CGL).

In the second and third steps, these assembly language codes are compiled. The *MCL* command invokes the compiler for MCU code and it produces an object file (MOB). The *MAC* command invokes the compiler for VAX code and it also produces an object file (OBJ).

In the fourth and fifth steps, the above referenced object files are linked, and executable files are produced. Specifically, in the fourth step, *Mpplink* is invoked

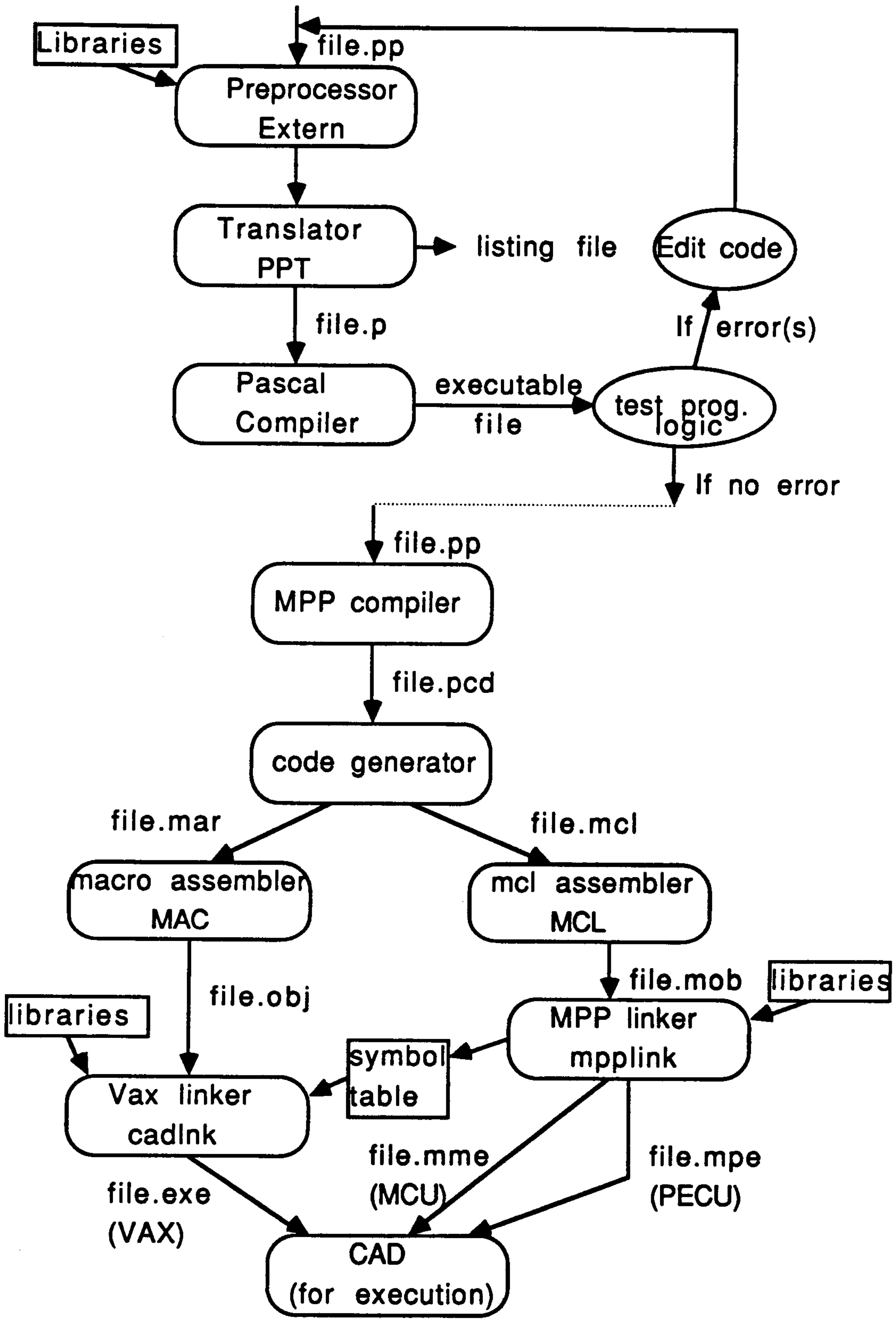


Figure 1.4. Flowchart of Compilation Steps.

and outputs the following files: MCU executable (MME), PECU array executable (MPE), MCU symbol table (MSY), PECU array symbol table (PSY), and a symbol table (STB) file. The fifth step (*Cadlnk*) uses the symbol table file to link the MPP side programs (MCU and PECU) to the VAX side programs. Cadlnk outputs a VAX executable file (EXE).

If no compilation errors are found, the program is ready to be executed. To start execution in the MPP the command *cad filename filename filename* is typed. Cad (control and debug) is an interactive program used to control the MPP. If *PPT* was used to debug the program, then the debug capabilities of *cad* will likely not be required.

When a user needs to measure the execution times of various sections of the code, four timing routines are available. These are Pfm_init, Pfm_start, Pfm_stop, and Pfm_close. To facilitate their use, these procedures are externally declared in the program and later loaded in the link step of compilation. Pfm_init initiallizes the timer and pfm_close prints the times measured. To time a given section of code, a pfm_start command is placed at the beginning of the section to start the timer. A pfm_stop command is placed at the end of the section to stop the timer.

The execution times of several primitive operations were measured using the timing routines. For example, the multiplication of two 128×128 arrays of reals takes 81.1 μsec. To measure this multiplication the following program segments were timed:

```
Program segment #1:
For i := 1 to 1000 do
begin
a := b  (* a and b are parallel arrays of reals *)
end;
```

```
Program segment #2:
For i := 1 to 1000 do
begin
a := a * b
end;
```

The time required to run the first program segment is equal to 6.60 msec, this is the time the MPP takes to execute the loop and an assignment per iteration. The time required for the second program segment is equal to 87.7 msec; this is the time the MPP takes to execute the loop and a multiplication with an assignment per iteration. Therefore the time required for a multiplication is the difference of the two divided by 1000.

$$t_{rm} = \frac{t_{ps2} - t_{psr1}}{1000}$$

where the t_{rm} is the time for a real multiplication, and t_{ps1} and t_{ps2} are the measured times for program segment #1 and program segment #2 respectively.

The cost of the basic primitive parallel operations of the MPP, when programed in Parallel Pascal, were measured in this way. When calculating the execution times of operations on 8-bit integers or Boolean data types, the time to execute program segment #1 is equal to 6.48 msec. The measured operation costs are presented in Table 1.1. Optimal times for these operations were estimated for the processor array by itself; these are also presented in Table 1.1. Optimal arithmetic times were obtained from [10], and the remaining optimal times were derived by counting the clock cycles for optimal microcode instruction sequences applied to the PE array hardware. The difference between the measured and optimal times is due to the MCU overhead, and to the overhead introduced by the Parallel Pascal compiler which does not yet have a code optimization stage.

According to the execution times presented in Table 1.1, the Boolean operations are the least efficient. The MCU adds an overhead of 3 or more μsec. per operation, which in the case of Boolean operations, dominates the execution times and causes an order of magnitude in loss of performance. On average, the Boolean measured times are about 20 times slower than the corresponding optimal times. For floating point (f.p.) operations, the 3 or more μsec. overhead is negligeble since the execution times of the operations are on the order of tens or hundreds of μsecs. On average, the f.p. optimal times are 2 times faster than the measured times. The f.p. optimal times obtained in [10] use the IBM format (i.e. hexadeximal base exponent) which is faster than the VAX format (i.e. binary base exponent) used for the measured times. The factor of 2 between measured and optimal times was thought to be caused by the difference in format, however, it has been stated [11] that, in the VAX format, the rate for f.p. multiply is 420 MFLOPS (76.2 μsec) and for f.p. add is 215 MFLOPS (39.0 μsec). These rates are only slightly slower than those of the IBM format, and therefore do not explain the obtained difference factor of 2. Finally, it was noted that for the shift operation, the fewer the number of bits of the elements of the array being shifted and the smaller the distance of the shift, the less efficient becomes the operation.

In the following chapters, when an algorithm is presented, the corresponding measured and estimated execution times will be given. The measured times were obtained using the timing functions of the MPP; and the estimated times were calculated by tracing the Parallel Pascal code of the algorithm and adding the optimal execution times, given in table 1.1, of the encountered array instructions. Any scalar operations, executed by the MCU, are assumed to be concurrent with the execution of the array operations, and thus, are not taken into account. As an example, the following is a program segment of the *shuffle* permutation, discussed in Chapter 2, for which the calculation of the estimated execution time is shown.

```
x := 1; y := 0                                             -1-
tmx1 := mx;                                                -2-
tmx2 := mx;                                                -3-

num := 2;                                                  -4-
while num < tn do                                          -5-
 begin
   tmx1 := shift(tmx1, -x, -y); (* shift down *)           -6-
   where id = num do                                       -7-
      mx := tmx1;                                          -8-
   num := num + 2;                                         -9-
 end;
```

where *x* and *y* are integers, and *num* and *tn* are 8-bit integers (*tn* = 128); *id* is a parallel array of 8-bit integer; *mx*, *tmx1*, and *tmx2* are parallel arrays of either 32-bit

Table 1.1. Optimal and Measured execution times of some typical operations.

Operation	Optimal time	Measured time
assignment r	6.4	6.6
assignment i8	1.6	1.8
assignment b	0.2	1.6
add r	38.1	75.8
mult r	75.8	81.1
mult r × s	43.9	87.7
sin r	-	334.
add i8	2.5	4.0
mult i8	8.8	9.0
mult i8 × s	7.0	7.5
div i8	17.6	24.1
mod i8	17.6	24.0
trunc	38.1	145.
round	38.1	145.
and	0.3	13.7
or	0.3	13.8
not	0.3	3.4
odd(i8)	0.5	2.9
any	0.5	2.4
min(r)	32.0	71.5
max(r)	32.0	71.2
min(i8)	8.0	33.1
max(i8)	8.0	33.1
compare i8	2.5	3.9
where b	0.1	3.1
shift(r,0,1)	3.2	9.9
shift(r,0,64)	205.	212.
shift(r,64,64)	410.	505.
shift(i8,0,1)	0.8	4.3
shift(i8,0,64)	51.2	62.9
shift(i8,64,64)	102.	126.
shift(b,0,1)	0.1	4.3
shift(b,0,64)	6.4	7.6
shift(b,64,64)	12.8	14.5
procedure call	-	150.(*)

time in μsec.
s = scalar, r = array of real, i8 = array of 8-bit integer, b = array of Boolean.
(*) The measured time for a procedure call varies according to the variables passed on the call.

reals, 8-bit integers, or Boolean. Assume the number of bits of the elements is equal to N_b .

The optimal execution time is equal to the sum of:

- 2 assign array of $N_b = 2 \times (0.2 \times N_b)$ (-1- and -2-)

the following are executed $\frac{tn}{2} - 1$ (i.e. 63) times:

- shift array of N_b by $1 = 0.3 \times N_b$ (-6-)
- where b + compare i8 = 0.1 + 2.5 (-7-)
- 1 assign array of $N_b = 0.2 \times N_b$ (-8-)

Note that an assignment operation takes two cycles per bit, and a shift operation takes per bit two cycles (for load and store) plus the shift distance. The total optimal time is equal to $31.9 \times N_b + 163.8$ µsec. For example, for arrays of 8-bit integer where $N_b = 8$, the optimal time is equal to 419.1 µsec.

CHAPTER 2

DATA PERMUTATIONS AND THE FAST FOURIER TRANSFORM

2.1. Introduction

An interconnection network can be defined by the set of permutation functions that it can implement. Using the notation of [4], a permutation function is performed on an ordered set of N elements, and it is defined by an one-to-one function $\pi(x)$. Both x and $\pi(x)$ are integers between 0 and N-1; x and $\pi(x)$ represent the addresses of the elements before and after the permutation, respectively.

The near-neighbor interconnection network of the MPP can only directly implement the *shift* permutation. This permutation is possible because the PE array has, in addition to the near neighbor connections, toroidal end-around edge connections. Any other permutation can only be achieved through the *shift* permutation.

In Parallel Pascal, the *shift* permutation is specified with the built in function 'rotate'. Its arguments consist of the array to be shifted, and the amount and direction of the shift. In the MPP, the cost of a *shift* permutation (t_s) depends on the amount of the shift (d) and on the number of bits of the array elements (N_b). In general the cost is equal to:

$$t_s = (2 + d) * 0.1 * N_b \ \mu\text{sec}$$

In this chapter, the algorithms developed to implement four other fundamental permutations in the MPP will be presented. In addition, a Fast Fourier Transform algorithm that uses two of these permutations will be presented.

2.2. Data Permutations

The permutations which have been implemented for the MPP are *exchange* (ε), *shuffle* (σ), *butterfly* (β), and *bit reversal* (ρ). Where applicable, the respective *Super* and *Sub* permutations were also implemented. In the MPP, a permutation is performed concurrently to N sets of N elements (N = 128). A set of elements is either a row or a column, depending on what coordinate is chosen. In general, the cost of a 2d permutation will be twice the cost of the 1d permutation results which are presented here.

The following four sections will present the definitions of the four above referenced permutations, along with their respective algorithms and performance results. A permutation will be defined by considering the binary representation of **x**:

$$x = (b_n, b_{n-1}, \ldots, b_1)$$

$$x = b_n 2^{n-1} + b_{n-1} 2^{n-2} + \ldots + b_1$$

The above expressions represent the binary address of an element in $N = 2^n$, and a permutation is defined by the permutation on the bits of this address [4]. As a performance measure, the *transfer ratio* is used. The *transfer ratio* is defined as the ratio of the time for the data transfer over the time for an elemental operation [5]. In this case, the data transfer is equal to the execution time of a permutation. The time for an elemental operation is defined as the average between the time of a multiplication and the time of an addition on the ARU. Based on the measured execution times given in Table 1.1. in Chapter 1, the time for an elemental operation for 32-bit floating point elements is equal to 78.5 μsec, and for 8-bit integer elements it is equal to 6.5 μsec. For Boolean elements, the time for an elemental operation is on average equal to 0.2 μsec [5].

Each permutation was coded in Parallel Pascal, and was run on the MPP using three types of data: 32-bit floating point, 8-bit integers, and Boolean. The measured execution times (tm_{π}), and the respective transfer ratios (r_{π}) will be given. For a π *sub*-permutation at bit **k** the measured execution time is called $tm_{\pi,k}$; and for a *super*-permutation, it is called $tm_{\pi,\bar{k}}$ where **k** is an integer in the range of 1 to n.

In addition to measured execution times, estimated execution times (to_{π}) have been calculated using the method described in section 1.3 of Chapter 1. The optimal execution times of the operations, given in table 1.1 of Chapter 1, were used in the calculations. In all the time equations presented in this chapter, N_b represents the number of bits of the data elements: N_b is equal to 32 for floating point elements, to 8 for integer elements, and to 1 for Boolean elements. Also, the time unit of these equations is μseconds.

2.2.1. Exchange Permutation

$\varepsilon_{(k)}(x) = (b_n, \ldots, \overline{b_k}, \ldots, b_1)$ *where* $1 \leq k \leq n$

The *exchange* permutation consists of complementing bit k of the input address. Thus, this permutation consists of exchanging every pair of elements, where two elements form a pair if their addresses are the same except for the kth bit. As an example, consider a set of sixteen ordered elements (N = 16, n = 4):

(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)

The binary representation of *exchange* bit 2 is:

$\varepsilon_{(2)}(x) = (b_4, b_3, \overline{b_2}, b_1)$

and the result is:

(2, 3, 0, 1, 6, 7, 4, 5, 10, 11, 8, 9, 14, 15, 12, 13)

Algorithm

In the *exchange* permutation all the elements move an equal distance, except that half of the elements move in one direction and the other half move in the opposite direction. This procedure is accomplished in two steps:

1 - Calculate the amount of the shift which is equal to 2^{k-1}

2 - Perform the exchange of pairs: where $b_k = 1$ then shift up (or left), and where $b_k = 0$ then shift down (or right).

Results

See tables 2.1 to 2.3 for the execution times and the respective transfer ratios. The estimated execution times were calculated using the following equation:

syntax error
file -, between lines 1326 and 1326

2.2.2. Shuffle Permutations

$\sigma(x) = (b_{n-1}, b_{n-2}, \ldots, b_1, b_n)$

The *shuffle* permutation consists of a circular left shift of the bits of the input address. The resulting permutation consists of splitting in half the set of N elements, and then interleaving them like in a perfect card shuffle. As an example, consider an ordered set of sixteen elements. The binary representation of the *shuffle*

Table 2.1. Exchange time results for 32-bit floating point elements.

bit k	to	tm	r
1	53.8	130.	1.65
2	60.2	151.	1.93
3	73.0	180.	2.30
4	98.6	221.	2.81
5	150.0	288.	3.66
6	252.0	406.	5.17
7	464.0	627.	7.99

time in µsec.

Table 2.2. Exchange time results for 8-bit integer elements.

bit k	to	tm	r
1	34.6	112.	17.2
2	36.2	128.	19.7
3	39.4	145.	22.3
4	45.8	164.	25.2
5	58.6	188.	28.9
6	84.2	230.	35.4
7	135.0	297.	45.7

time in µsec.

Table 2.3. Exchange time results for Boolean elements.

bit **k**	to	tm	r
1	29.0	112.	560
2	29.2	128.	639
3	29.6	144.	718
4	30.4	160.	798
5	32.0	175.	876
6	35.2	191.	955
7	41.6	208.	1040

time in μsec.

is:

$$\sigma(x) = (b_3, b_2, \ldots, b_1, b_n)$$

and the result is:

(0, 8, 1, 9, 2, 10, 3, 11, 4, 12, 5, 13, 6, 14, 7, 15)

Algorithm

1 - Map the upper half of the input matrix by shifting the elements down a total of $\frac{N}{2} - 1$ steps. For each shift down, one element will be located in the correct position, and therefore stored in the result array.

2 - Map the lower half of the input matrix. The same procedure of step **1** is followed except that the elements are shifted up.

Results

Refer to table 2.4. for the estimated and measured execution times, as well as the transfer ratios. The estimated execution times were calculated using the following equation:

$$to_\sigma = 63.4 * N_b + 819.0$$

In addition to the *perfect shuffle*, the *sub-shuffle* and the *super-shuffle* permutations were implemented:

Sub-Shuffle

$$\sigma_{(k)}\,(x) = (b_n, \ldots, b_{k+1}, b_{k-1}, \ldots, b_1, b_k)$$

The *sub-shuffle* permutation is defined by the circular left shift of the **k** least significant bits of the input address. In the resulting permutation, the set of elements is divided into 2^{n-k} groups, each one of size 2^k, and a perfect shuffle is performed to each of the subgroups. As an example, consider an ordered set of sixteen elements. The binary representation of the *sub-shuffle* at bit 3 is:

$$\sigma_{(3)}\,(x) = (b_4, b_2, b_1, b_3)$$

and the result is:

(0, 4, 1, 5, 2, 6, 3, 7, 8, 12, 9, 13, 10, 14, 11, 15)

The *sub-shuffle* algorithm is the same as for the *shuffle*, except that there are 2^{n-k}

Table 2.4. Perfect Shuffle time results.

	to	tm	r
32-bit f.p.	2.34	3.93	50.1
8-bit int.	0.84	3.70	569
Boolean	0.39	3.66	18300

time in msec.

halves, and total amount of the shift is $2^{k-1} - 1$. Please refer to table 2.6 for the timing results. The estimated execution times are given in table 2.5 and were calculated using the following equation:

$$to_{\sigma,\,k} = 17.6 + 0.4 * N_b + (0.5 * N_b + 2.6)\,(2^k - 2)$$

Super-Shuffle

$$\sigma^{(k)}\,(x) = (b_{n-1}, \ldots, b_{n-k+1}, b_n, b_{n-k}, \ldots, b_1)$$

The *super-shuffle* permutation is defined by the circular left shift of the **k** most

Table 2.5. Sub-Shuffle estimated time results.

bit **k**	to 32-bit	to 8-bit	to Boolean
1	30.4	20.8	18.0
2	67.6	34.0	24.2
3	142.0	60.4	36.6
4	291.0	113.0	61.4
5	588.0	219.0	111.0
6	1180.	450.0	210.0
7	2370.	852.0	409.0

time in μsec.

Table 2.6. Sub-Shuffle measured time results.

bit **k**	tm 32-bit	r	tm 8-bit	r	tm Bool.	r
1	0.075	0.96	0.072	11.1	0.072	361
2	0.149	1.90	0.137	21.1	0.136	681
3	0.280	3.57	0.257	39.5	0.256	1280
4	0.536	6.82	0.490	75.4	0.489	2440
5	1.040	13.2	0.948	146.0	0.947	4740
6	2.040	26.0	1.860	286.0	1.860	9280
7	3.930	50.1	3.580	551.0	3.570	17960

time in msec.

significant bits of the input address. The resulting permutation still performs a perfect shuffle on the whole set, except that now an 'element' consists of a group of 2^{n-k} elements. As an example, consider an ordered set of sixteen elements. The binary representation of the *super-shuffle* at bit 3 is:

$$\sigma^{(3)}(x) = (b_3, b_2, b_4, b_1)$$

and the result is:

(0, 1, 8, 9, 2, 3, 10, 11, 4, 5, 12, 13, 6, 7, 14, 15)

The *super-shuffle* algorithm is the same as for the *shuffle*, except that instead of shifting by one, a shift by 2^{n-k} is performed at each step. Also, after each shift, 2^{n-k} elements are located in the correct location and therefore stored in the result array. Please refer to table 2.8 for the timing results. The estimated execution times are given in table 2.7 and were calculated with the following equation:

$$to_{\sigma,\,\bar{k}} = 17.6 + 0.4 * N_b + [(12.8 * 2^{-k} + 0.4) * N_b + 2.6]\,(2^k - 2)$$

Table 2.7. Super-Shuffle estimated time results.

bit **k**	to 32-bit	to 8-bit	to Boolean
1	30.4	20.8	18.0
2	266.0	83.6	30.4
3	430.0	132.0	45.6
4	604.0	192.0	71.2
5	876.0	291.0	120.0
6	1380.	480.0	216.0
7	2370.	852.0	409.0

time in µsec.

Table 2.8. Super-Shuffle measured time results.

bit **k**	tm 32-bit	r	tm 8-bit	r	tm Bool.	r
1	0.124	1.58	0.123	18.9	0.123	614
2	0.393	5.01	0.206	31.7	0.183	917
3	0.611	7.78	0.386	59.4	0.300	1500
4	0.889	11.3	0.530	81.5	0.530	2650
5	1.363	17.4	0.984	151.0	0.984	4920
6	2.270	28.9	1.890	291.0	1.890	9440
7	4.060	51.7	3.690	568.0	3.690	18460

time in msec.

2.2.3. Butterfly Permutations

$\beta\,(x) = (b_1, b_{n-1}, \ldots, b_2, b_n)$

The *butterfly* permutation consists of exchanging the most significant bit (MSB) and the least significant bit (LSB). Three cases arise from this exchange: First, if the bits are equal (i.e. both are equal to 1 or to 0), the permuted addresses are unchanged, and therefore the corresponding elements remain in their initial positions. Second, if MSB = 1 and LSB =0, the corresponding elements have to move up $2^{n-1} - 1$ locations away. Third, if the MSB = 0 and LSB =1, the corresponding elements have to move down $2^{n-1} - 1$ locations away.

Similarly to the *butterfly*, the *sub-butterfly* permutation consists of exchanging bit **k** (MSB) and bit **1** (LSB); and the *super-butterfly* permutation consists of exchanging bit **n** (MSB) and bit **n-k+1** (LSB); The binary representations of these two permutations are:

$\beta_{(k)}\,(x) = (b_n, \ldots, b_{k+1}, b_{k-1}, b_1, \ldots, b_k)$

$\beta^{(k)}(x) = (b_{n-k+1}, \ldots, b_{n-k+2}, b_n, b_{n-k}, \ldots, b_1)$

Consider an ordered set of sixteen elements. The following are examples of the *butterflies* permutations. The binary representation of *butterfly* is:

$\beta(x) = (b_1, b_3, b_2, b_4)$

and the result is:

(0, 8, 2, 10, 4, 12, 6, 14, 1, 9, 3, 11, 5, 13, 7, 15)

The binary representation of *sub-butterfly* at bit 3 is:

$\beta_{(3)}(x) = (b_4, b_1, b_2, b_3)$

and the result is:

(0, 4, 2, 6, 1, 5, 3, 7, 8, 12, 10, 14, 9, 13, 11, 15)

The binary representation of *super-butterfly* at bit 3 is:

$\beta^{(3)}(x) = (b_2, b_3, b_4, b_1)$

and the result is:

(0, 1, 8, 9, 4, 5, 12, 13, 2, 3, 10, 11, 6, 7, 14, 15)

Algorithm

The algorithm consists of two steps:

1 - Generate two Boolean masks: 'shiftup' indicates the positions where the MSB of the address is equal to 0 and the LSB is equal to 1, and 'shiftdown' indicates the positions where the MSB is equal to 1 and the LSB is equal to 0. Nothing needs to be done where MSB = LSB.

2 - Perform the shifts: where 'shiftup' is true, the elements are obtained with a shift up by a distance of $2^{n-1} - 1$; where 'shiftdown' is true, the elements are

obtained with a shift down by a distance of $2^{n-1} - 1$.

The same algorithm applies for the *sub* and *super butterfly* permutations, except that in the *sub-butterfly* a $2^{k-1} - 1$ shift is performed, and in the *super-butterfly* a $(2^{k-1} - 1)(2^{n-k})$ shift is performed.

Results

The *butterfly* permutation is obtained either from $\beta_{(n)}(x)$ or from $\beta^{(n)}(x)$. Please refer to table 2.10 for the timing results of the *sub-butterfly*, and to table 2.12 for the timing results of the *super-butterfly*. The respective estimated execution times are given in tables 2.9 and 2.11 and were calculated with the following equations:

$$to_{\beta,k} = 0.4 * N_b + 24.2 + 0.1 * N_b 2^k$$

$$to_{\beta,\bar{k}} = 13.4 * N_b + 24.2 - 25.6 * N_b * 2^{-k}$$

Table 2.9. Sub-Butterfly estimated time results.

bit k	to 32-bit	to 8-bit	to Boolean
1	43.4	29.0	24.8
2	49.8	30.6	25.8
3	62.6	33.8	25.4
4	88.2	40.2	26.2
5	139.0	53.0	27.8
6	242.0	78.6	31.0
7	447.0	130.0	37.4

time in μsec.

Table 2.10. Sub-Butterfly measured time results.

bit k	tm 32-bit	r	tm 8-bit	r	tm Bool.	r
1	147	1.87	136	20.9	134	672
2	161	2.05	148	22.8	148	738
3	179	2.29	155	23.8	154	771
4	212	2.70	165	25.4	161	805
5	270	3.44	178	27.4	168	840
6	380	4.83	207	31.8	175	874
7	591	7.53	265	40.8	182	912

time in µsec.

Table 2.11. Super-Butterfly estimated time results.

bit k	to 32-bit	to 8-bit	to Boolean
1	43.4	29.0	24.8
2	248.0	80.2	31.2
3	351.0	106.0	34.4
4	402.0	119.0	36.0
5	427.0	125.0	36.8
6	440.0	128.0	37.2
7	447.0	130.0	37.4

time in µsec.

Table 2.12. Super-Butterfly measured time results.

bit k	tm 32-bit	r	tm 8-bit	r	tm Bool.	r
1	195	2.49	185	28.5	184	919
2	401	5.10	223	34.3	189	945
3	503	6.41	249	38.3	189	945
4	554	7.06	261	40.2	189	945
5	580	7.39	268	41.2	190	950
6	593	7.55	271	41.7	190	650
7	599	7.63	272	41.8	190	650

time in µsec.

2.2.4. Bit Reversal Permutations

$\rho\ (x) = (b_1, b_2, \ldots, b_{n-1}, b_n)$

The *bit reversal* permutation consists of reversing the order of the bits of the input address; it is like taking the mirror image of the address. Similarly, the *sub-bit reversal* at bit k reverses the k least significant bits: bit **k** to bit **1**, and the *super-bit reversal* reverses the k most significant bits: bit **n** to bit **n-k+1**. The binary representations of these two permutations are:

$\rho_{(k)}\ (x) = (b_n, \ldots, b_{k+1}, b_1, b_2, \ldots, b_k)$

$\rho^{(k)}\ (x) = (b_{n-k+1}, b_{n-k+2}, \ldots, b_n, b_{n-k}, \ldots, b_1)$

Consider an ordered set of sixteen elements. The following are examples of the *bit reversal* permutations. The binary representation of *bit reversal* is:

$\rho\ (x) = (b_1, b_2, b_3, b_4)$

and the result is:

(0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15)

The binary representation of *sub-bit reversal* at bit 3 is:

$\rho_{(3)}\ (x) = (b_4, b_1, b_2, b_3)$

and the result is:

(0, 4, 2, 6, 1, 5, 3, 7, 8, 12, 10, 14, 9, 13, 11, 15)

The binary representation of *super-bit reversal* at bit 3 is:

$\rho^{(3)}\ (x) = (b_2, b_3, b_4, b_1)$

and the result is:

(0, 1, 8, 9, 4, 5, 12, 13, 2, 3, 10, 11, 6, 7, 14, 15)

Algorithm

The *bit reversal* permutation can be achieved with a series of bit exchanges between the pairs of corresponding MSB and LSB, i.e. between b_n and b_1, b_{n-1} and b_2, . . . , b_{n-m} and b_{m+1} where $m < \left\lfloor \frac{n}{2} \right\rfloor$. Therefore a bit reversal consists of a series of *butterflies* where the distance of each shift is calculated from the bits position numbers. If exchanging MSB bit **j** with LSB bit **i**, the amount of the shift is equal to $2^{j-1} - 2^{i-1}$. The algorithm consists of $\left\lfloor \frac{n}{2} \right\rfloor$ iterations, where each iteration consists of:

1 - Determine which pair of bits is to be exchanged, and calculate the amount of the shift.

2 - Similarly to the *butterfly*, create the shifts masks, and then perform the shifts.

The same algorithm applies for the *sub* and *super bit reversal* permutations, except that the bit exchanges are performed only to the **k** least significant bits or the **k** most significant bits, respectively. In both cases, the number of iterations is equal to $\left\lfloor \frac{k}{2} \right\rfloor$.

Results

Please refer to table 2.13 for the timing results of the *bit reversal.* The estimated execution times were calculated using the following equation:

$$to_{\mathrm{p}} = 337.8 + 22.8 * N_b$$

Please refer to table 2.15 for the timing results of the *sub-bit reversal*, and to table 2.17 for the timing results of the *sub-bit reversal*. The respective estimated execution times are given in tables 2.14 and 2.16. The estimated execution times were calculated using the following equations:

$$to_{\mathrm{p},\,k} = (212.0 + 0.6 * N_b) \left\lfloor \frac{k}{2} \right\rfloor + (0.323 * 2^k - 0.683)\, N_b$$

$$to_{\mathrm{p},\,\bar{k}} = (212.0 + 0.6 * N_b) \left\lfloor \frac{k}{2} \right\rfloor + (3.36 * k - 0.08)\, N_b$$

Table 2.13. Bit Reversal time results.

	to	tm	r
32-bit f.p.	1.07	1.57	20
8-bit int.	0.520	1.51	232
Boolean	0.361	1.50	7520

time in msec.

Table 2.14. Sub-Bit Reversal estimated time results.

bit k	to 32-bit	to 8-bit	to Boolean
1	0	0	0
2	138	119	113
3	151	122	114
4	321	249	228
5	398	268	231
6	709	431	349
7	1070	520	361

time in μsec.

Table 2.15. Sub-Bit Reversal measured time results.

bit k	tm 32-bit	r	tm 8-bit	r	tm Bool.	r
1	19.	0.24	19.	2.9	19.	94
2	442.	5.63	442.	68.0	441.	2206
3	464.	5.91	463.	71.2	463.	2320
4	917.	11.7	917.	141.0	917.	4590
5	948.	12.1	948.	146.0	947.	4740
6	1431.	18.2	1431.	220.0	1430.	7150
7	1580.	20.1	1510.	232.0	1510.	7540

time in μsec.

Table 2.16. Super-Bit Reversal estimated time results.

bit k	to 32-bit	to 8-bit	to Boolean
1	0	0	0
2	337	169	120
3	439	194	123
4	724	350	241
5	801	369	243
6	1020	509	359
7	1070	520	361

time in μsec.

Table 2.17. Super-Bit Reversal measured time results.

bit k	tm 32-bit	r	tm 8-bit	r	tm Bool.	r
1	0.184	2.34	0.184	28.3	0.184	922
2	0.688	8.76	0.687	106.0	0.687	3440
3	0.738	9.39	0.667	103.0	0.667	3340
4	1.170	14.9	1.140	175.0	1.139	5700
5	1.160	14.7	1.110	171.0	1.110	5550
6	1.610	20.5	1.550	238.0	1.550	7750
7	1.580	20.1	1.510	232.0	1.510	7550

time in msec.

2.2.5. Conclusions

An instruction used by all the permutations is the *shift* instruction. As expected, the costs of the permutations are proportional to the amount of the shift(s) and to the number of bits of the elements being shifted. The cost of a shift is in turn proportional to the value of **k**. The transfer ratio is proportional to the amount of the shift(s) and inversely proportional to the number of bits of the elements. As a result, with respect to the transfer ratio, all the permutations are most efficient when using 32-bit data and least efficient when using Boolean data. In addition, the lower the value of **k**, the lower is the transfer ratio. For example, the *exchange* permutation with the lowest transfer ratio and therefore the most efficient, is $\varepsilon_{(1)}$ for 32-bit data elements. In general, the transfer ratio is in the order of 1 to 10's for 32-bit data, 10's to 100's for 8-bit data, and 100's to 1000's for Boolean data.

Two kinds of execution times, estimated and measured, were presented in the results of each permutation. When comparing these times, it is found that the estimated times are faster than the measured times by an average factor of 1.94 for 32-bit data, 3.51 for 8-bit data, and 5.94 for Boolean data. There are several reasons for the differences between the estimated and the measured times. These reasons are presented in the remainder of this section.

First, the runtime support for the MPP Parallel Pascal compiler does not optimally implement the Parallel Pascal primitive operations as is shown in table 1.1. The optimal times were used in the calculation of the estimated execution times; therefore, it is anticipated that the estimated times will be lower than the measured times.

Second, the overhead of the MCU is not taken into account in the estimation of the execution times of the permutations. In theory, the MCU operations are executed at the same time as the ARU operations. This is possible because an instruction queue exists, between the MCU and the PECU, which allows the MCU to perform its operations while the array is busy processing instructions in the buffer. (The MCU issues the array instructions and puts them in the queue; it also executes all the scalar operarions). If during the execution of a program, the depth of the queue is always at least one, then the ARU is never idle, and therefore, the operations executed by the MCU are 'free' of cost. But if at a given time the queue is empty, the ARU could stay idle while it waits for the next instruction. This idle time constitutes the MCU overhead.

The MCU overhead is greater for arrays of Boolean because, in order to keep the ARU constantly busy, the MCU needs to issue instructions at a faster rate than for example for arrays of 32-bit elements. This is why the factor of difference between the estimated and the measured times, which reflects the MCU overhead, is greater for Boolean data than for 32-bit data. For example, the factor of difference of the *butterfly* permutations is approximately equal to 6 for Boolean and to 2 for 32-bit floating point elements. A clear case when the measured time is due to the MCU is when **k** = 1 for the *sub* and the *super-bit reversal* permutations. No ARU operations are executed when **k** = 1, and therefore the estimated times are equal to 0. The measured times are equal to 19 µsec for the *sub-bit reversal* and equal to 184 µsec for the *super-bit reversal.* These times correspond to how long it takes the MCU to determine which bit planes are to be exchanged. For **k** = 1, the least significant plane is to be exchanged with itself, so obviously nothing needs to be done.

Finally, for a *shift* instruction, the difference factor between the optimal and measured execution times (given in table 1.1 of Chapter 1) increases as the distance of the *shift* is decreased. The results obtained with the permutation functions are consistant with this statement. For example, the *shuffle* permutation, which consists of a series of shifts by a distance of 1, is the permutation with the largest average difference factor equal to 5.2 .

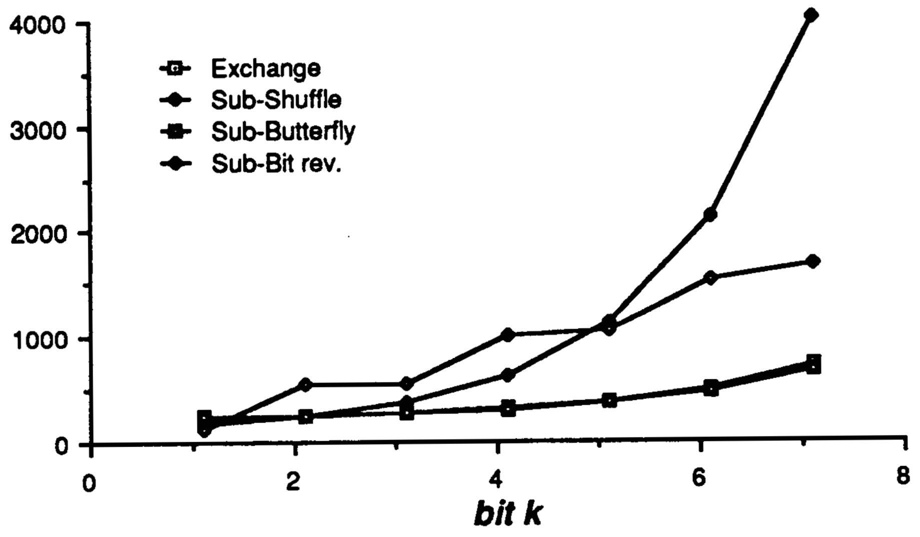


Figure 2.1. Measured execution times of the Exchange and sub-permutations.

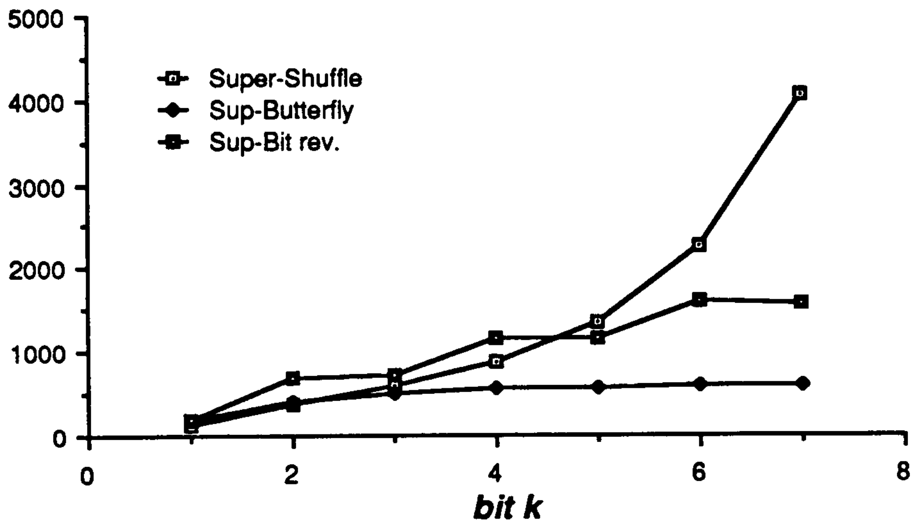


Figure 2.2. Measured execution times of the super-permutations.

In general, the execution times of the permutations are dominated by *shift* operations and by the MCU overhead; no reduction operations are used and the arithmetic operations are minimally used. Refer to Figures 2.1 and 2.2 for a plot of the measured execution times of the permutations with 32-bit data elements. Also, it should be noted, that if the Parallel Pascal 'bit-plane indexing' instruction is implemented on the MPP, the execution times of the *butterfly*, *bit reversal*, and *exchange* permutations would significantly be reduced. The execution time for 'bit-plane indexing' should be comparable to the 'odd' instruction which returns the least significant bit plane of its argument. The 'odd' instruction takes approximately 3.4 μsec. Since 'bit-plane indexing' is not currently implemented on the MPP, a function *bitplane* had to be created which takes an average of 75.0 μsec (including the function call).

2.3. Fast Fourier Transform

A Fast Fourier Transform (FFT) program, that uses the β, σ, and ρ permutations, was developed. The Fourier Transform is the frequency domain representation of a function and it is frequently used in several different scientific applications. The FFT, as its name implies, is a fast method to compute the Discrete Fourier Transform (DFT), since it reduces the calculation of the FT from $O(n^2)$ for DFT to $O(n \log_2 n)$. An N-point DFT is equal to:

$$X(n) = \sum_{k=0}^{N-1} x_o(k) * w^{nk}, \quad n = 1, 2, \ldots, N-1$$

where $w = e^{-j2\frac{\pi}{N}}$ and N is a power of 2. Please refer to [12] for a detailed discussion of the FFT.

Because of the widespread use of the FFT, it is often used as a measure of the performance of a given computer system, and thus, it is of interest to determine the execution time of an FFT on the MPP. In the following sections, the FFT algorithm that was implemented and the corresponding obtained time results will be presented.

2.3.1. Algorithm

The input to the FFT program is an N×N array, and the FFT is performed concurrently to either the rows or the columns of the array, depending on what coordinate is chosen. Consequently, a two dimensional FFT is achieved by performing an FFT to the columns and then to the rows, or vice versa.

In general, the algorithm used to perform a FFT to a given row of length N (or column) is defined as follows:

For a bit reversed result:

$$FFT_n = \beta_{(n)} W \beta_{(n-1)} W \ldots \beta_{(2)} W \sigma W \quad \textit{where } N = 2^n$$

For a normal-ordered result:

$$FFT_n = \beta_{(n)} W \beta_{(n-1)} W \ldots \beta_{(2)} W \sigma W \rho \quad \textit{where } N = 2^n$$

β, σ, and ρ refer to the permutations presented in section 2.2; **W** represents the following operation:

$$x_l = x_{l-1} + w^{p} * y_{l-1}$$

$$y_l = x_{l-1} - w^{p} * y_{l-1}$$

where $w = e^{-j2\frac{\pi}{N}}$, p represents the power of w, l represents the iteration number, and x and y are dual nodes ([12] p. 154). A diagram of **W** is shown in Figure 2.1.

A perhaps better known FFT definition is obtained if every β is replaced by a σ; but as seen in section 2.2., σ takes an order of magnitude longer to execute than β, and thus, for the MPP, the FFT with β's is executed more efficiently. Please refer to Figure 2.2 for an example of the FFT calculation for N = 16.

As mentioned previously, the FFT algorithm has complexity $O(n \log_2 n)$, but on the MPP, it takes $\log_2 n$ steps because there are N PE's for N elements, as opposed to one PE for N elements (N = 128). In the MPP, a step or iteration l consists of

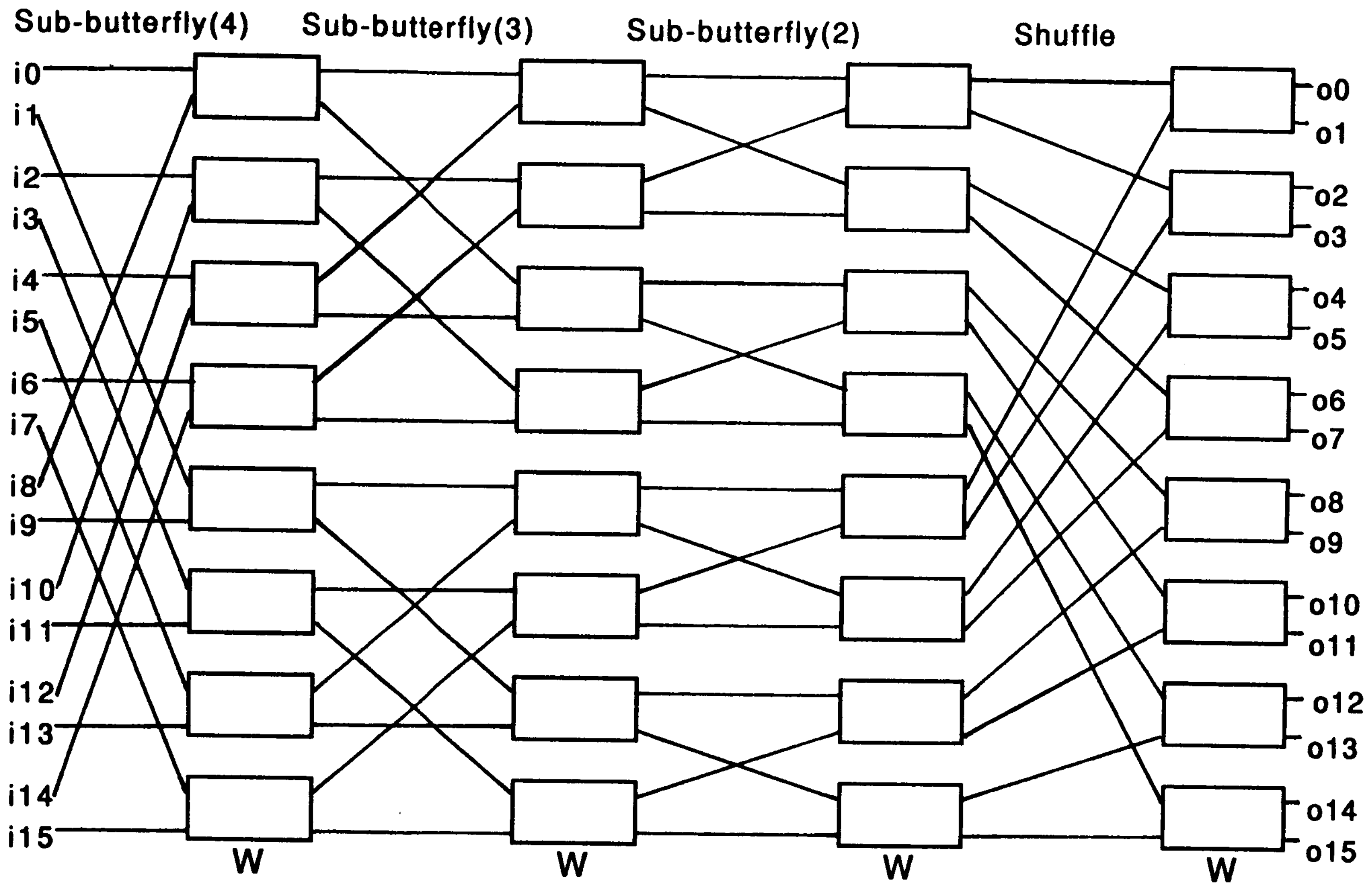


Figure 2.4. Flow of Data for FFT, N = 16.

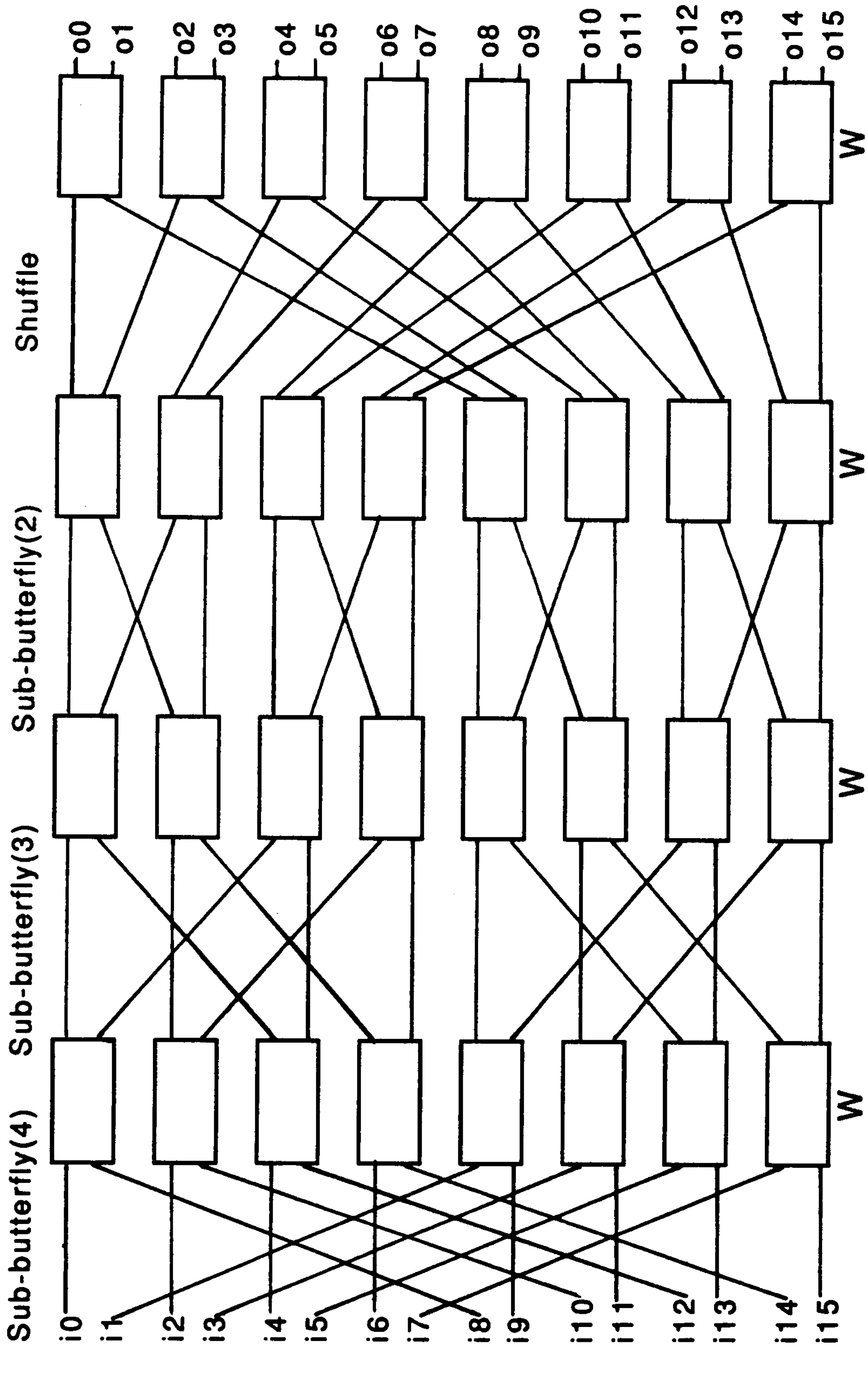


Figure 2.4. Flow of Data for FFT, N = 16.

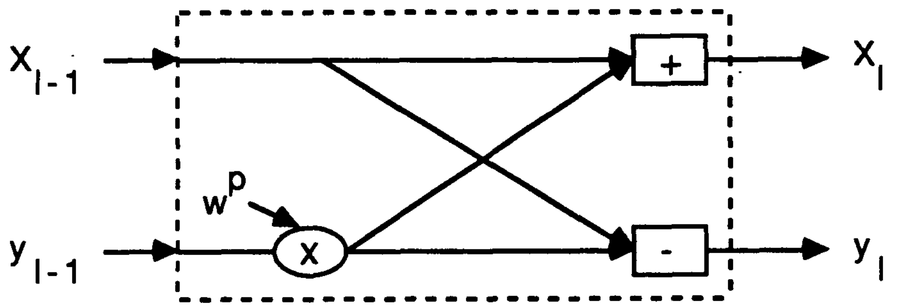


Figure 2.3. *W* operation.

the following:

1 - Perform the $\beta_{(n-l+1)}$ permutation except when $l = n$, in this case a σ permutation is performed. At this point all the dual nodes pairs are located in adjacent PE's.

2 - Calculate the weights, w^p's. The method used is simple but yet efficient: First the values of the *p's* are generated, these are dependant on l and on the corresponding array position. Then the real part of w is equal to $cos(\frac{2\pi}{N}\ p)$ and the imaginary part is equal to $-\ sin(\frac{2\pi}{N}\ p)$.

3 - Calculate x_l's and y_l's, i.e. perform the W operation presented above. Both the x_l's and the y_l's are calculated concurrently in the procedure *complexmult.* Initially, the x_{l-1}'s are in the even numbered PE's and a copy is send to the corresponding dual odd numbered PE; the opposite is done to the y_{l-1}'s. Then the complex multiplication and addition are performed.

At the beginning of the execution of the FFT program, three parameters need to be defined. The first parameter indicates whether the FT is performed on the rows or on the columns. The second parameter indicates whether an FFT or an Inverse FFT (IFFT) is performed. If an IFFT is chosen, the algorithm is the same as for the FFT (given above) with two exceptions: first, before starting the iterations, the input elements are divided by the number of elements i.e. by N; and second, the definition of the imaginary part of the weight w is equal to $+\,sin(\frac{2\pi}{N}\,p)$. The third parameter indicates whether modulation is to be performed. Modulation is achieved by multiplying all the odd numbered input elements by -1.

2.3.2. Results

On the MPP, an FFT_7 is performed. FFT_7 is defined as:

$$FFT_7 \;=\; \beta_{(7)}\, W\, \beta_{(6)}\, W\, \beta_{(5)}\, W\, \beta_{(4)}\, W\, \beta_{(3)}\, W\, \beta_{(2)}\, W\, \sigma\, W$$

Please refer to table 2.18 for the execution times of the β and σ permutations as well as for the respective transfer ratios. The execution times of these permutation functions are more than the execution times of the same permutations presented in section 2.2, because, in the FFT, the data elements used are complex numbers i.e.

two 32-bit floating point elements. In the calculation of the transfer ratio, the time for an elemental operation is equal to the average between a complex add and a complex multiplication (i.e. 314 μsec).

The execution times of the functions that the FFT procedure uses are given in table 2.19. The 'complexmult' performs the *W* operation and its execution time is independant of *l*, the iteration number. The 'powergen' function generates the *p* values and its execution time depends on the value of *l*. Finally, table 2.20 presents the execution times of the FFT procedure.

The estimated times of table 2.20 are faster than the measured times by a factor of 1.51. The execution times of the FFT's are dominated by arithmetic operations (*W* calculation) and by shifting operations (permutations); no reduction functions are used. The MCU overhead is likely to be minimun since most of the operations are array operations of 32-bit floating point elements; therefore, the

Table 2.18. FFT Permutations time results.

Permutation	to	tm	r
$\beta_{(7)}$	851	993	3.16
$\beta_{(6)}$	451	617	1.96
$\beta_{(5)}$	252	445	1.42
$\beta_{(4)}$	154	379	1.21
$\beta_{(3)}$	107	771	2.46
$\beta_{(2)}$	85	362	1.15
σ	4390	7290	23.2

time in μsec.

Table 2.19. FFT functions time results.

Function	to	tm
complexmult	437	765
powergen $l = 2$	41.9	294
powergen $l = 3$	64.6	393
powergen $l = 4$	87.3	493
powergen $l = 5$	110.0	593
powergen $l = 6$	133.0	692
powergen $l = 7$	4.2	23

time in μsec.

Table 2.20. FFT time results.

	to	tm
FFT	14.82	22.42
FFT modulated	14.89	22.43
IFFT	15.01	22.71
IFFT modulated	15.08	22.76

time in msec.

differences between estimated and measured times are mainly due to the inefficient implementation of Parallel Pascal primitive operations.

As a performance measure of the FFT, the number of MFLOPS was calculated. An operation is defined as either a floating point multiplication or addition. It was determined that the *sine* and *cosine* operations, used in the calculation of the weight factors, have a measured execution time equal to 334 μsec, and thus, a sine or cosine operation on the MPP has a cost equivalent to four floating point multiplication operations.

In an FFT, two main calculations are performed at each iteration: first, the weights factors are generated, and second, the *W* operation is performed. In general, at each iteration 9 floating point operations per processor are necessary for the weights calculation and 8 are necessary for the *W* operation. There are seven iterations and 128^2 processors, and the FFT takes 0.0224 seconds. In conclusion, the FFT calculation achieves an approximate rate of 87 MFLOPS.

In addition to the MFLOPS calculation, the percentage of time spent on shift operations and the percentage of time spend on arithmetic calculations were found. The shift operations consist of the *butterflies* and the *shuffle* permutations, and the arithmetic operations consists of the weight factors calculation (*powergen* and the sine and cosine operations), and of the *W* operation (*complexmult*). The FFT program spends approximately 46% of the execution time on shift operations, and approximately 54% on arithmetic operations.

CHAPTER 3

IMAGE WARPING

3.1. Introduction to the Warp Operation

Many image processing applications require a warp or "rubber sheet" operation in order to remove irregular distortions, such as those caused by perspective,view angle, scanner motion during image acquisition, etc. Examples of warp operations include translation, rotation, and scale changes. The work done by Moura [13] will form the basis of the following discussion of the warp operation.

A parallel warp operation can be implemented in two steps. First, two coordinate-index matrices are generated. These matrices encode the mapping function between the input image (matrix to be warped) and the output image (result matrix), and are called, respectively, the **r** matrix (for row index) and the **c** matrix (for column index). For images of size $n \times n$ and given i,j where $i \leq n$ and $j \leq n$, the output image, R, at the position [i,j] corresponds to the input image, I, at the position designated by row coordinate r[i,j] and column coordinate c[i,j]. This mapping function can be represented by the following formula:

$$R\,[i,j] = I\left[r[i,j],\, c[i,j]\right]$$

The second step of a warp operation consists of performing the mapping indicated by the **r** and **c** matrices. Two algorithms have been explored to perform this mapping function: a simple direct algorithm and a heuristic algorithm.

The simple algorithm requires every element of the input image to be passed by every position of the output image. When an element is located at the appropriate output position, that is, when it has moved the correct distance according to the **r** and **c** matrices, its value is then stored at this position. The following is a Parallel Pascal program segment which implements this algorithm. With respect to the MPP, the sizes of the matrices are 128x128.

Note: **id1** is an index matrix where the values of the elements are equal to their corresponding row position number, i.e. id1[i,] := i for all rows.
Note: **id2** is an index matrix where the values of the elements are equal to their corresponding column position number, i.e. id2[,j] := j for all columns.

```
Begin
  (* calculate relative distance to corresponding output position *)
  r := (r - id1 + numrow) mod numrow;
  c := (c - id2 + numcol) mod numcol;

  (* perform the mapping *)
  For i := 1 to numrow do begin
    For j := 1 to numcol do begin
      where r = i and c = j do
        outimage := inimage;
      (* pass by every column position *)
      inimage := rotate(inimage, 0, 1);
    end;
    (* pass by every row position *)
    inimage := rotate(inimage, 1, 0);
  end;
end.
```

The cost complexity of this algorithm is $O(n^2)$ where n is equal to the number of rows (or columns) in the image. This algorithm always requires n^2 iterations and n^2 rotations, and, on the MPP, its optimal and measured execution times are respectively 153 msec and 600.8 msec for images with 8-bit elements, and 350 msec and 685.3 msec for images with 32-bit elements. The cost is dependent only on the size of the image and the number of bits per element, and not on the warping function. As a result of this fact, this algorithm fails to take advantage of any uniformity or locality which may exist in the movement of the elements.

3.2. A Heuristic Algorithm

A heuristic algorithm [13] has been developed for the MPP which takes advantage of uniformity or locality existing in the movement of the data elements. Its cost depends on the size of the image as well as on the specific warp operation. The greater the similarity of the movement of neighboring image elements, the lower the cost of the algorithm will be.

The following is a Parallel Pascal program segment which implements the heuristic algorithm:

Variables:

msk : array of boolean, true values indicate elements of *outimage* that are out of range.

mask : array of boolean, true values indicate elements of *outimage* which have not yet received the correct element of *inimage*.

ri, *ci* : row and column distances for the rotation.

lastrit : the last value of *rit*.

Variables used to process one column:

maskt : a version of *mask*.

rit : a version of *ri*.

inimaget : a version of *inimage*.

rt : a version of *r*.

lastrit : the last value of *rit*.

```
(* Calculate relative distances to the corresponding outputs *) {1}
r := (r - id1 + numrow) mod numrow;
c := (c - id2 + numcol) mod numcol;
(* initiallize *)
lastrit :=0;
outimage := inimage;
mask := not msk and (r <> 0) or (c <> 0);

(* iterate until the mapping is completed *) {2}
while any(mask, 1 ,2) do
    begin (* start an iteration *)
        ri := min(r, 1, 2); (* calculate the rotation distance *) {3}
        ci := min(c, 1, 2);
        inimage := rotate(inimage, ri, ci); (* perform the rotation *)
        r := r - ri; {4}
        c := c - ci;
        maskt := (r = 0) and (c = 0);
        if any(maskt, 1, 2) then
            inimaget := inimage (* store matched element(s) *) {5}
        else (* start of the else branch *) {6}
            (* of no matched element(s), process the column(s) with c = 0 *)
            begin
                where c = 0 do
                    rt := r
                otherwise
                    rt := numrow;
                rit := min(rt, 1, 2); {7}
                maskt := rt = rit;
                { the next seven statements implement }
                { the statement inimaget = rotate (inimage, rit, 0) }
                { but also take advantage of the previous rotations }
                if ci <> 0 then
```

```
                    begin
                        inimaget := inimage;
                        lastrit := 0;
                    end;
                inimaget := rotate( inimaget, rit - lastrit, 0 );
                lastrit := rit;
            end; (* end of the else branch *)
        (* store the matched elements into outimage *)
        (* update the corresponding r, c, and mask values *)
        where maskt do
            begin
                outimage := inimaget;
                r := numrow;
                c := numcol;
                mask := false;
            end;
    end; (* end of the iteration *)
```

This heuristic algorithm works in the following way: First, the relative distances to the corresponding output positions are calculated from the **r** and **c** matrices **{1}**. Then, a series of iterations are performed until the mapping is completed. Before any iteration is begun, however, a test is performed to check whether or not the mapping of all the output elements has been completed (no superfluous iterations are performed) **{2}**. Specifically, a parallel Boolean array, *mask*, is used such that if at a position in the array the value is true, this true value serves as an indication that the corresponding position in the output image has not received the correct element value. The parallel pascal reduction function 'any' is used to perform the test. This function performs a logical 'and' over all the elements of the Boolean array, and, if the result is false, it indicates that the mapping has been completed.

At each iteration, the image is rotated by the maximum distance possible so that no backtracking will be required later. This distance is determined by computing the minimums of the **r** and **c** matrices {3}. After the rotation, the **r** and **c** matrices are updated by substracting the distance of the rotation {4}. At this point, at least one or more of the elements will be matched or none will be matched. If there is a matched element(s), i.e. a position where **r** and **c** are equal to zero, the value is stored in the corresponding output element, and the corresponding **r** and **c** values are updated to the maximum possible distance *numrow* or *numcol* {5}. If there is no matched element, the element with the minimum **r** value and with **c** = 0 is then matched {6}. The matching is accomplished by rotating a temporary array, *inimaget*, by the minimum **r** value {7}. If elements remain with **c** = 0, they will be mapped in the subsequent iterations, but in such an event the accumulated rotations of the temporary array are taken into account (using the *lastrit* variable). Once all of these remaining elements with c = 0 are mapped, the above-referenced process is repeated.

The worst case cost of the heuristic algorithm is O(n^2), as this algorithm may conceivably require up to n^2 iterations. The two warp functions that we used, however, always took less. The heuristic algorithm requires less iterations because it only performs the moves that are needed by the given warp operation. But the penalty for this efficiency, is that the heuristic algorithm incurs an additional cost overhead for each iteration because of the Boolean operations, the updates of the **r** and **c** matrices, and the reduction operations it uses to check for the completition of the mapping ('any') and to determine the rotation distance ('min'). The cost of these reduction operations depend on the data size and on the system. For the MPP, this cost could become a major part of the cost of the algorithm.

3.3. Interpolation

So far we have assumed that the generated **r** and **c** matrices indicate an exact mapping, i.e. that for a given output element the **r** and **c** matrices select a specific input element. But in most cases these matrices specify a position that falls between four input elements, such as those represented in Figure 3.1.

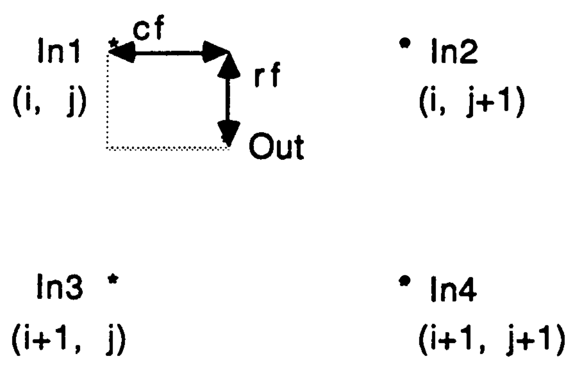


Figure 3.1. Four Element Interpolation.

Three methods exist to determine the value of a given output element for which **r** and **c** do not select a specific input element:

1 - *Near Neighbor Interpolation*: This interpolation is the simplest, but the least accurate. As its name indicates, the closest input element to the r,c position is chosen. For example, in Figure 3.1 the element i,j would be mapped to the corresponding output element.

2 - *Bilinear Interpolation*: In this method all four surrounding input elements are considered. The weight of each element with respect to the output element depends on its distance to the r,c position. Once the four elements are mapped, the output element is equal to:

$$Out = (1-cf)*(1-rf)*In1 + (1-rf)*cf*In2 + rf*(1-cf)*In3 + rf*cf*In4$$

3 - *Bicubic Interpolation*: This method is similar to the bilinear interpolation, except that the surrounding sixteen elements are considered (sixteen mappings are needed). Again, the weight of each input element depends on its distance to the r,c position. See Figure 3.2.

For methods 2 and 3, where several mappings are required, the algorithm first performs a mapping, then multiplies the mapped array by its weight and adds it to the partial result array. When this process has been performed with respect to all of the neighbors, (four for bilinear and sixteen for bicubic), the partial result is equal to the output image.

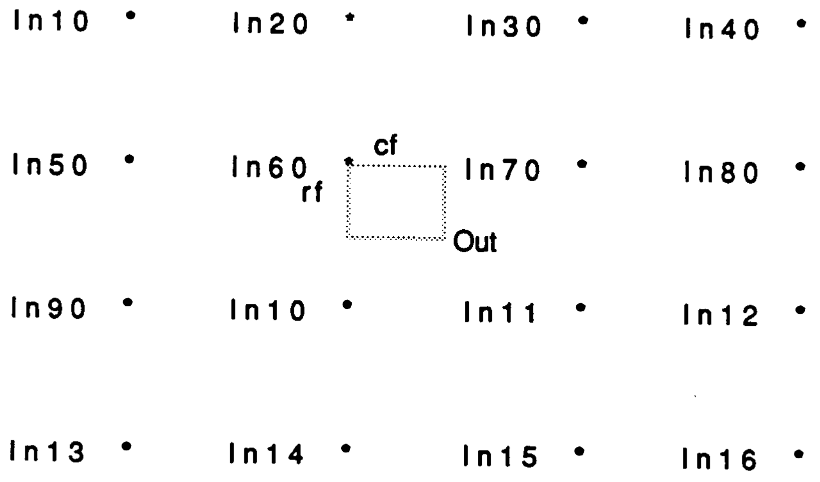


Figure 3.2. Sixteen Element Interpolation.

3.4. Results

The heuristic algorithm presented in section 3.2 was run on the MPP. Two different warping operations were used: Rotation with translation and 'Fisheye' distortion. For each of these operations, this section will both describe how the **r** and **c** matrices are generated, and present the estimated and measured execution times of the heuristic algorithm using near neighbor, bilinear, and cubic interpolation respectively. The estimated execution times were calculated using the method described in section 1.3 of Chapter 1. As usual, the optimal execution times of the instructions, given in table 1.1 of Chapter 1, are used in the calculations of the estimated times. Similarly to Chapter 2, *to* refers to the estimated time and *tm* refers to the measured time. In the *to* equations, N_b represents the number of bits of the elements of the images, and the time units of the equations are μseconds.

In addition to execution times, the transfer ratio of the near neighbor algorithm was calculated since this figure represents the transfer ratio of one mapping. As defined in Chapter 2, the transfer ratio is the ratio of the time for data transfer over the time for an elemental operation. In this case, the time for data transfer is the time to execute the heuristic algorithm i.e. to perform the mapping. The time for an elemental operation is equal to the average between the time for a multiplication and the time for an addition in the ARU. The near neighbor transfer ratio figure is used as a performance measure for data mapping; the smaller the ratio the more efficient the data mapping in the MPP will be.

For the bilinear interpolation, the computation cost is approximately four times that of the near neighbor algorithm, while for the cubic interpolation the cost is approximately sixteen times.

3.4.1. Rotation with translation warp

This warp operation consists of rotating the image by an angle θ with the center of rotation at ro,co. The mapping between two corresponding points of the input and output images is defined as follows:

$$\begin{bmatrix} xo \\ yo \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & ro \\ \sin\theta & \cos\theta & co \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} xi \\ yi \\ 1 \end{bmatrix}$$

From this, the Parallel Pascal statements that generate the **r** and **c** matrices are:

```
r := cos θ (id1-ro) - sin θ (id2-co) + ro;

c := sin θ (id1-ro) + cos θ (id2-co) + ro;
```

(id1-ro) and *(id2-co)* are the matrices with the *xi's* and the *yi's*. r and c are the matrices with the *xo's* and *yo's*.

Refer to Figure 3.3 for an example of a near neighbor rotation warp of an 8×8 image for *ro*, *co* = 1, 1 and θ = 45°.

```
11 12 13 14 15 16 17 18      11  0  0  0  0  0  0  0
19 20 21 22 23 24 25 26      20 12  0  0  0  0  0  0
27 28 29 30 31 32 33 34      20 21 14  0  0  0  0  0
35 36 37 38 39 40 41 42      29 22 23 15  0  0  0  0
43 44 45 46 47 48 49 50      38 31 23 24 17  0  0  0
51 52 53 54 55 56 57 58      47 39 32 25 25 18  0  0
59 60 61 62 63 64 65 66      47 48 41 33 26  0  0  0
67 68 69 70 71 72 73 74      56 49 49 42  0  0  0  0
      Input Matrix              45 degree rotation at 1,1
```

Figure 3.3. Example of a near-neighbor rotation warp.

Rotation Warp Mapping Time Results

The rotation with translation warping using near-neighbor, bilinear, and bicubic interpolation was run on the MPP. For each kind of interpolation the program was run and timed for different θ, *ro*, and *co*. The size of the images was 128×128.

In order to calculate the estimated execution times corresponding to each measured time, it was necessary to obtain the values for the number of iterations (ni), the number of column processings i.e. the number of times that the **else** branch in the heuristic algorithm is taken (ne), and the number of rotations (rot) required to perform one mapping for a given θ, *ro*, and *co* rotation warp. For each iteration, two 'any' and two 'min' reductions are performed, and for each column processing, one 'min' reduction is performed. Please refer to tables 3.1 and 3.2 for the cost of one mapping with the rotation warp respectively centered at 1,1 and at 64,64. The number of column processings is equal to the number of 'min' minus the number of 'any' reductions. From these tables, and with respect to the number of iterations and the number of rotations, the heuristic algorithm is found more efficient than the simple algorithm. In all cases, the heuristic algorithm requires less than the 16,384 rotations required by the simple algorithm.

The estimated times were calculated using the following equations:

- Near neighbor interpolation mapping:

$$to = 0.4*N_b + 51.2 + ni*(31.2 + 0.6*N_b) + ne*(16.3 + 0.4*N_b) + 0.1*rot*N_b$$

$$For\ \ N_b = 8 \quad to = 54.4 + ni * 36.0 + ne * 19.5 + rot * 0.8$$

$$For\ \ N_b = 32 \quad to = 64.0 + ni * 50.4 + ne * 29.1 + rot * 3.2$$

- Bilinear interpolation mapping: (approximately four times that of the near neighbor case)

$to = 930.2 + 4 * to_{nn}$

- Bicubic interpolation mapping: (approximately sixteen times that of the near neighbor case)

$to = 16618.6 + 16 * to_{nn}$

The timing results of the near neighbor rotation warp are given in table 3.3 for a rotation centered at 1,1 and in table 3.4 for a rotation centered at 64,64. In addition to the estimated (to) and the measured (tm) execution times, the transfer ratio (*r*) is also given. As stated above, the transfer ratio reflects how efficient an image mapping, using the heuristic algorithm, is on the MPP. The timing results, *to* and *tm*, for the bilinear rotation warp are given in tables 3.5 and 3.6, and for the bicubic warp, the timing results are given in tables 3.7 and 3.8.

The transfer ratios of tables 3.3 and 3.4 are very large. Their size indicates that the mapping operation for the rotation warp is very costly. It should be noted though, that the mapping is more efficient as the number of bits of the data

Table 3.1. Cost of one mapping for the Rotation with Translation Warp centered at 1, 1.

angle θ	iterations	rotations	'any' reductions	'min' reductions
0	1	0	1	0
15	910	1451	1820	2666
30	2619	3359	5238	7759
45	4069	5838	8138	12098
60	3882	7975	7764	11553
75	2232	9029	4464	6659
90	128	8129	256	382

Table 3.2. Cost of one mapping for the Rotation with Translation Warp centered at 64, 64.

angle θ	iterations	rotations	'any' reductions	'min' reductions
0	1	0	1	0
15	1053	4446	2106	3091
30	3778	8890	7556	11197
45	7994	13589	15988	23771
60	11463	16369	22926	34174
75	11781	16358	23562	35148
90	8192	16318	16384	24448

Table 3.3. Near-Neighbor timing results where *ro*, *co* = 1, 1.

Angle θ	8-bit element image			32-bit element image		
	to	tm	r	to	tm	r
0	0.054	0.34	5.2E+1	0.054	0.33	5.7E+0
15	50.47	577.4	8.9E+3	75.19	579.0	9.5E+3
30	146.2	1684.0	2.6E+4	216.2	1687.0	2.8E+4
45	228.4	2628.0	4.0E+4	339.1	2634.0	4.3E+4
60	220.1	2511.0	3.9E+4	331.5	2523.0	4.1E+4
75	130.4	1451.0	2.2E+4	205.3	1468.0	2.4E+4
90	13.62	87.99	1.4E+3	36.19	109.1	1.8E+3

time in msec.

Table 3.4. Near-Neighbor timing results where *ro*, *co* = 64, 64.

Angle θ	8-bit element image			32-bit element image		
	to	tm	r	to	tm	r
0	0.054	0.33	5.1E+1	0.064	0.35	5.0E+0
15	60.73	671.3	1.0E+5	96.03	680.6	1.1E+4
30	214.2	2432.0	3.7E+5	324.9	2447.0	4.0E+4
45	450.5	5164.0	7.9E+5	672.9	5183.0	8.5E+4
60	645.2	7626.0	1.2E+6	957.5	7441.0	1.2E+5
75	663.2	7638.0	1.2E+6	983.3	7648.0	1.3E+5
90	465.3	5320.0	8.2E+5	699.8	5323.0	8.7E+4

time in msec.

Table 3.5. Bilinear timing results where *ro*, *co* = 1, 1.

Angle θ	8-bit element image		32-bit element image	
	to	tm	to	tm
0	0.0011	0.004	0.0012	0.004
15	0.2028	2.308	0.3017	2.314
30	0.5857	6.729	0.8656	6.738
45	0.9146	10.53	1.3570	10.55
60	0.8812	9.927	1.3270	9.970
75	0.5227	5.653	0.8222	5.722
90	0.0011	0.011	0.0011	0.011

time in sec.

Table 3.6. Bilinear timing results where *ro*, *co* = 64, 64.

Angle θ	8-bit element image		32-bit element image	
	to	tm	to	tm
0	0.0011	0.004	0.0012	0.004
15	0.2438	2.695	0.3850	2.731
30	0.8576	9.744	1.300	9.800
45	1.803	20.69	2.693	20.75
60	2.282	29.74	3.831	29.79
75	2.654	30.65	3.934	30.69
90	1.862	27.74	2.800	27.78

time in sec.

Table 3.7. Bicubic timing results where *ro*, *co* = 1, 1.

Angle θ	8-bit element image		32-bit element image	
	to	tm	to	tm
0	0.0175	0.041	0.0176	0.044
15	0.8242	9.327	1.220	9.390
30	2.356	27.14	3.475	27.25
45	3.671	42.50	5.442	42.67
60	3.538	40.50	5.321	40.73
75	2.104	23.60	3.302	23.91
90	0.2346	1.724	0.5957	1.897

time in sec.

Table 3.8. Bicubic timing results where *ro*, *co* = 64, 64.

Angle θ	8-bit element image		32-bit element image	
	to	tm	to	tm
0	0.0175	0.041	0.0176	0.044
15	0.9882	10.78	1.553	10.93
30	3.443	38.98	5.215	39.23
45	7.224	82.84	10.78	83.16
60	10.34	119.0	15.34	119.3
75	10.63	122.5	15.75	122.8
90	7.461	110.5	11.21	110.8

time in sec.

increases.

In general, the estimated results were faster than the measured results by an average factor of 11.4 for rotation warps with 8-bit element images, and by an average factor of 7.6 for rotation warps with 32-bit element images. The difference between the estimated and the measured results is caused by implementation inefficiencies and overheads of the operations used in the heuristic algorithm, and by the MCU scalar operations which are not taken into account in the calculation of the estimated optimal results. As noted in Chapter 2, part of the overhead cost is due to the lack of an optimizing stage for the Parallel Pascal Compiler.

The MCU issues the instructions to be executed by the ARU, and it performs all scalar operations. Implementation overhead arises when the MCU is not able to issue instructions into the queue fast enough to keep the ARU continuously busy. When the instruction queue is empty, the ARU stays idle if waiting for the next instruction. In theory the MCU executes scalar operations while the ARU executes array operations. However, some scalar operations, in the heuristic algorithm, control the flow of program execution and cannot be executed concurrently; these are the operations associated with the *while* and *if* statements. There is one *while* statement which determines if another iteration is to be started; the evaluation of this while operation depends on the result of the reduction operation 'any' which causes the ARU instruction queue to empty. There are two conditional branches (*if* statements), one of which also evaluates with the result of an 'any' reduction, and thus, it also causes the queue to empty.

The ARU operations used in the heuristic algorithm may be divided into four groups. The non-optimal execution of these operations, along with the MCU scalar operations, contribute to the factor of difference between the optimal and the measured time results.

- The *reduction operations* used to check for the completion of the mapping ('any') and to determine the rotation distance ('min'). These account for about 44% of the estimated execution time of the heuristic algorithm. The operation 'any' takes a measured time of approximately 2.4 μsec and the operation 'min', with an array of 8-bit elements, takes 33.1 μsec. For these operations, the optimal execution time is faster than the measured time by a factor of 4.8 for the 'any' reduction, and by a factor of 4.1 for the 'min' reduction. The reduction operations contribute to the difference between the estimated and measured times of the heuristic algorithm because, in addition to their non-optimal implementation, the reduction operations introduce a MCU overhead that is not considered in the calculation of the estimated times. The result of a reduction operation is a scalar value for which the MCU must wait. Since the MCU has to wait until the reduction instruction is complete, it cannot issue any more array instructions into the queue, and therefore the queue becomes empty. It is at this time, when the queue is empty and the ARU is ready for another instruction, that the MCU overhead is created.

- The *Boolean operations* used to determine the completition of the mapping and to store the mapped elements into the appropiate output positions. These operations account for about 20% of the estimated execution time of the heuristic algorithm. As seen in table 1.1. of Chapter 1, on the MPP, Boolean operations are the least efficiently implemented. On average, the Boolean optimal times are about 20 times faster than the measured times.

- The *'update' operations* on the **r** and **c** matrices. These account for about 20% of the estimated execution time of the heuristic algorithm. On average, the optimal times of these operations (arithmetic and assignment) are about 2 times faster than the measured times.

- The *rotate operations* used to move the input image elements to the corresponding output positions. These account for about 8% for 8-bit element images and 20% for 32-bit element images. In both cases, the shift optimal times are approximately 2 times faster than the measured times.

3.4.2. 'Fisheye' warp

In the Fisheye warp operation, the sampling rate of the image is changed. The new sampling rate at a given position is determined according to its distance with respect to the *ro,co* coordinates. Specifically, the sampling rate is increased for the nearby area around the *ro,co* coordinates, and accordingly this area is mapped onto a larger one. For the remaining area of the image, the sampling rate is decreased. Thus, this remaining area is mapped into a smaller one.

The **r** and **c** matrices for this 'fisheye' warp are calculated using the (1 - cos) function. For example the **r** matrix is equal to:

- For the positions above ro:

$$r = ro + ro * \left\{ -1 + \cos\left[\frac{\pi * (\, id1 - ro\,)}{2 * ro} \right] \right\}$$

- For the positions below ro:

$$r = ro + (numrow - ro) * \left\{ -1 + \cos\left[\frac{\pi * (\, id1 - ro\,)}{2 * (numrow - ro)} \right] \right\}$$

Refer to Figure 3.4 for an example of a near neighbor In this example the warp operation is applied to both the x and the y coordinates, i.e. $rowc = colc = 1$, and the warp is centered at $ro,\ co = 4,\ 4$.

```
11 12 13 14 15 16 17 18     20 21 22 22 22 23 24 26
19 20 21 22 23 24 25 26     28 29 30 30 30 31 32 34
27 28 29 30 31 32 33 34     36 37 38 38 38 39 40 42
35 36 37 38 39 40 41 42     36 37 38 38 38 39 40 42
43 44 45 46 47 48 49 50     36 37 38 38 38 39 40 42
51 52 53 54 55 56 57 58     44 45 46 46 46 47 48 50
59 60 61 62 63 64 65 66     52 53 54 54 54 55 56 58
67 68 69 70 71 72 73 74     68 69 70 70 70 71 72 74
     Input  Matrix               Expand at 4,4
```

Figure 3.4. Example of a near-neighbor 'fisheye' warp.

Fisheye Warp Mapping Time Results

The 'fisheye' warping using near-neighbor, bilinear, and bicubic interpolation was run on the MPP. For each kind of interpolation the program was run and timed for different input parameters, namely *rowc, colc, ro*,and *co*. The parameters *rowc* and *colc* indicate whether or not to change the sampling rate in their respective coordinate. If *rowc* is 0 the sampling rate is not changed in the *rows*, while if it is a 1 the sampling rate is changed. The same applies for *colc*, but with respect to the columns. The size of the images was 128×128.

The estimated times were calculated using the equations given in section 3.4.1. Refer to table 3.9 for the cost of one mapping of the 'fisheye' warp. The number of column processings is equal to the number of 'min' minus the number of 'any' reductions. Similar to the rotation warp, in all cases of the 'fisheye' warp the heuristic algorithm required less than the 16,384 iterations and the 16,384 rotations required by the simple algorithm. Therefore, with respect to the number of iterations and the number of rotations, the heuristic algorithm is found to be more efficient than the simple algorithm.

The timing results of the near neighbor 'fisheye' warp are given in table 3.10. In addition to the estimated (to) and the measured (tm) execution times, the transfer ratio (*r*) is also given. The timing results, *to* and *tm*, for the bilinear and the bicubic 'fisheye' warps are given in tables 3.11 and 3.12, respectively.

The same measured execution time results, given in tables 3.10 to 3.11, apply if the input parameters rowc and colc were 0 1, respectively, as opposed to 1 0, and if the value 32 is replaced with any other value from 16 to 112 except for 64. On the average, the estimated results were 11.1 times faster than the measured results for 8-bit element images, and 6.5 times faster for 32-bit element images. The same

Table 3.9. Cost of one mapping for the 'Fisheye' Warp.

input	iterations	rotations	'any' reductions	'min' reductions
1 0 32 32	28	127	56	56
1 0 64 64	27	127	54	54
1 1 32 32	784	3683	1568	2297
1 1 64 64	729	3556	1458	2134
1 1 64 32	756	3683	1512	2214
1 1 32 32	756	3556	1512	2214

Table 3.10. Near-Neighbor timing results.

input	8-bit element image			32-bit element image		
	to	tm	r	to	tm	r
1 0 32 32	1.164	11.13	1.7E+3	1.882	11.40	1.9E+2
1 0 64 64	1.128	10.75	1.7E+3	1.831	11.02	1.8E+2
1 1 32 32	45.44	503.1	7.7E+4	72.58	510.7	8.4E+3
1 1 64 64	42.33	467.4	7.2E+4	67.86	474.8	7.8E+3
1 1 64 32	43.91	484.9	7.5E+4	70.38	492.6	8.1E+3
1 1 32 64	43.80	484.7	7.5E+4	69.97	492.1	8.1E+3

time in msec., input parameters are rowc, colc, ro, and co.

Table 3.11. Bilinear timing results.

input	8-bit element image		32-bit element image	
	to	tm	to	tm
1 0 32 32	5.586	45.34	8.457	46.54
1 0 64 64	5.442	45.34	8.255	46.54
1 1 32 32	182.7	1998.0	291.2	2031.0
1 1 64 64	170.0	1998.0	272.4	2031.0
1 1 64 32	176.6	1998.0	282.5	2031.0
1 1 32 64	176.1	1998.0	280.8	2031.0

time in msec., input parameters are rowc, colc, ro, and co.

Table 3.12. Bicubic timing results.

input	8-bit element image		32-bit element image	
	to	tm	to	tm
1 0 32 32	35.24	203.8	46.72	210.1
1 0 64 64	34.67	203.8	45.92	210.1
1 1 32 32	743.7	8011.0	1178.0	8141.0
1 1 32 64	693.8	8011.0	1102.0	8141.0
1 1 64 32	719.1	8011.0	1143.0	8141.0
1 1 32 64	717.5	8011.0	1136.0	8141.0

time in msec., input parameters are rowc, colc, ro, and co.

analysis for the rotation warp, given in section 3.4.1, applies for the 'fisheye' warp.

3.4.3. Conclusion

The heuristic algorithm is most efficient when few image elements are to be moved or when there is a lot of uniformity in the movement of the elements. The heuristic algorithm always takes fewer than the n^2 iterations and n^2 rotations required by the simple algorithm. However, the cost of the heuristic algorithm may still be greater if the cost of the overhead operations, (the operations other than the rotations) becomes greater than the savings from the reduced number of iterations and rotations. In addition, on the MPP, the heuristic algorithm is at a disadvantage with respect to the simple algorithm because the measured execution times are slower by a factor of 7 to 11 than the optimal execution times, as opposed to a factor of 2 to 3 for the simple algorithm.

Comparing the execution times of the simple algorithm and the heuristic algorithm for the 'Fisheye' warp, the heuristic algorithm is always faster since it takes advantage of the considerable amount of uniformity in the movement of these warp. The 'Fisheye' mapping requires as few as 27 iterations and a total rotation distance of 127 when the warp is applied in one coordinate in the middle of the image, and at most 784 iterations with a rotation distance of 3683 when the warp is applied in both coordinates at any position except the middle of the image.

Comparing the estimated execution times of the simple algorithm and the heuristic algorithm for the Rotation warp, the heuristic algorithm is less efficient for

Rotations of large angles. The number of iterations range from 128 in the case of a 90 degree Rotation centered at 1,1 to 11,781 iterations in the case of a 75 degree Rotation centered at 64,64. The rotation distances range from 1451 in the case of a 15 degree Rotation warp centered at 1,1 to 16,369 iterations in the case of a 60 degree Rotation warp centered at 64,64. The number of iterations and rotation distances are less than the 16,384 required by the simple algorithm, but not always not enough to offset the cost of the overhead operations and the greater implementation cost of the heuristic algorithm. This is especially true for the Rotation warp centered at 64,64.

CHAPTER 4

CONVOLUTION AND PYRAMID-FILTERING OPERATIONS

4.1. Convolution and Pyramid Operations

An important image processing operation is Convolution; it involves *convolving* an image with a given small matrix: the *convolution kernel.* This chapter presents the convolution operation and two pyramid filtering operations where the filtering is obtained with convolution operations. The corresponding results, obtained on the MPP, are given.

4.2. Convolution

A centered convolution operation is defined as:

$$I\,[i,j] = 4 * \sum_{x=-m}^{m} \sum_{y=-m}^{m} W[x,y] * I\left[i+x,\, j+y\right]$$

where I is the image, W is the convolution kernel of size $(2m + 1) \times (2m + 1)$. In general, the dimensions of the kernel are much smaller than the dimensions of the image.

Conceptually, the convolution result for a given image element I[i,j] is obtained by superposing the W kernel onto the image (with the center of W at the i,j position), and multiplying each kernel element with the corresponding image element. The convolution result for i,j is then equal to the summation of these products.

On the MPP, the implementation of a convolution operation involves a series of shift-multiply-add operations. The centered convolution operation using 5×5 kernels was run on the MPP. The measured execution time for images with 8-bit integer elements is 987 μsec, and for images with 32-bit floating point elements is 5.28 msec. As a performance measure of the convolution on the MPP, the number of MFLOPS was calculated. A convolution with a 5×5 kernel involves 50 operations (25 additions and 25 multiplications) per processor; and therefore the convolution operation achieves approximately a rate of 830 MOPS for 8-bit integer data and 155 MFLOPS for 32-bit floating point data. If no interprocessor communications were needed, the measured execution times would be equal to 325 μsec (33% of the total time) for 8-bit integer data and to 3.92 msec (74% of the total time) for 32-bit floating point data. The convolution with 32-bit floating point data spends a lower percentage of the execution time in interprocessor communication, and therefore, it is more efficiently implemented on the MPP than the convolution with 8-bit integer data.

4.3. Pyramid Operations

Many image processing algorithms, especially early processing and segmentation algorithms, may be efficiently implemented with pyramid data structures. A pyramid data structure consists of several copies of an image at different resolutions. As one can see in Figure 4.1 ([14] p. 106), the highest level of a pyramid consists of one element, the next level down consists of four elements in a 2×2 matrix, and so on. In general, the dimensions of a level are twice as large as the previous level. Thus, the lower the level in the pyramid, the greater the resolution of the image. For a given level **k**, there are $2^k \times 2^k$ elements, with each one being connected to one element in the level above and to four in the level below.

An important operation in many pyramid algorithms is to build the pyramid from the image located at its lowest level. In this chapter two such operations are

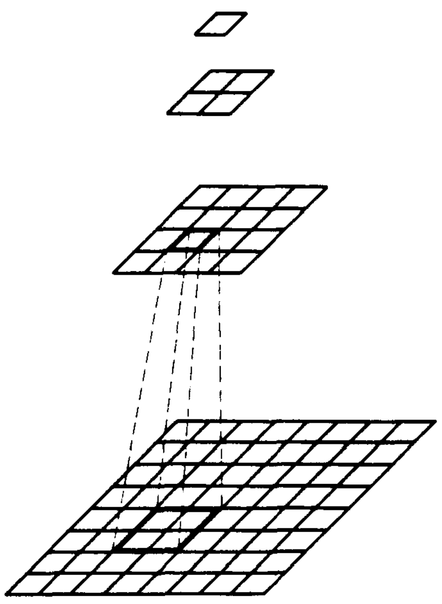


Figure 4.1. The Pyramid Structure.

considered: The Gaussian and Laplacian operations, these are filtering operations (Gaussian for low-pass filtering and Laplacian for band-pass filtering) in which each level is the filtered copy of the adjacent lower level. Even though the mesh interconnection network of the MPP does not match the connection of a pyramid structure, thereby preventing optimal results, it is of interest to emulate a fully parallel pyramid architecture, and to investigate what results can be obtained on the MPP.

4.4. Pyramid operations on the MPP

A simple way to store a pyramid structure in a two dimensional array is to store the successive levels of the pyramid in successive rows of the array. The highest level (one element) is located at the [0,0] position in the array. See Figure 4.2 for an example of a three level pyramid stored in an 8×8 array.

Three different movements must be implemented in order to perform pyramid operations on the MPP: a shift within each level, a shift to move up, and a shift to move down the pyramid. The function *pyrmskg* generates *pyrmsk*, a Boolean pyramid constrain mask, that is used by the function *xshift* to perform shifts within all the levels of the pyramid. These functions are used as follows, where x and y are the amounts of the xshift in the x and y directions respectively:

pyrmsk := *pyrmskg*(*id*1, *id*2)

pyramid := *xshift*(*pyramid*, *x*, *y*, *pyrmsk*)

The function *pyrgen* generates all the shift matrices used for moving up and down

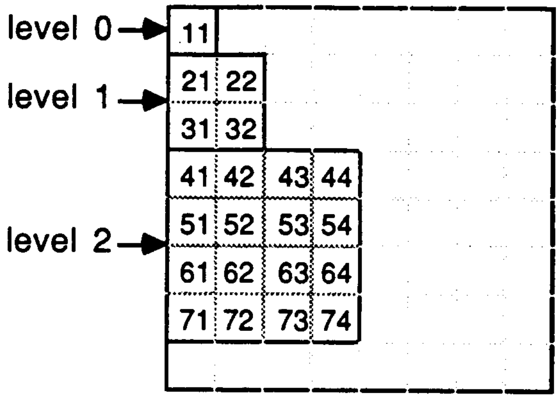


Figure 4.2. A three level Pyramid embedded in an 8×8 array.

the pyramid. These shift matrices are 'up1' and 'up2' to move up in the x (row) and in the y (column) direction respectively, and similarly 'dn1' and 'dn2' to move down. For example 'up1' is generated as follows:

$$up1 := pyrgen(id1, id2, true, 1)$$

The *gather* function performs the up and down movements. An upward shift of all levels of the pyramid is generated by:

$$pyramid := gather\ (gather\ (pyramid, up1, 1), up2, 2)$$

A downward shift is generated by:

$$pyramid := gather\ (gather\ (pyramid, dn1, 1), dn2, 2)$$

For an example of an up and down movement, achieved with the *gather* function, in a three level pyramid refer to Figure 4.3 (the input pyramid is given in Figure

4.2.). Also refer to Figure 4.3 for the four possible shift movements within the levels achieved with the *xshift* function. Finally, table 4.1 gives the execution times of these functions in the MPP. These functions were developed by Gary Ross and Anthony Reeves.

4.5. Gaussian Filtering

As stated in section 4.1, the Gaussian pyramid is a sequence of images in which each one is a low-pass filtered copy of the image below. A filtered image is obtained by convolving the image of the level below with a 5×5 kernel and then shifting the result one level up. The convolution operation is the same as the one

Table 4.1. Execution times of the pyramid primitives.

Pyramid primitive	T. execution	Transfer ratio
Pyrmskg (Boolean)	4.90 msec	4.5E+3
8-bit integers		
Pyrgen of 'up1'	40.5 μsec	6.23
Pyrgen of 'up2'	39.5 μsec	6.08
Pyrgen of 'dn1'	73.3 μsec	11.30
Pyrgen of 'dn2'	51.7 μsec	7.95
8-bit integers		
Xshift up or left by 1	190.0 μsec	29.20
Xshift down or right by 1	275.3 μsec	42.40
32-bit reals		
Xshift up or left by 1	199.2 μsec	2.54
Xshift down or right by 1	285.5 μsec	3.64
Gather (8-bit integers)	97.31 msec	1.49E+4
Gather (32-bit reals)	98.43 msec	1.25E+3

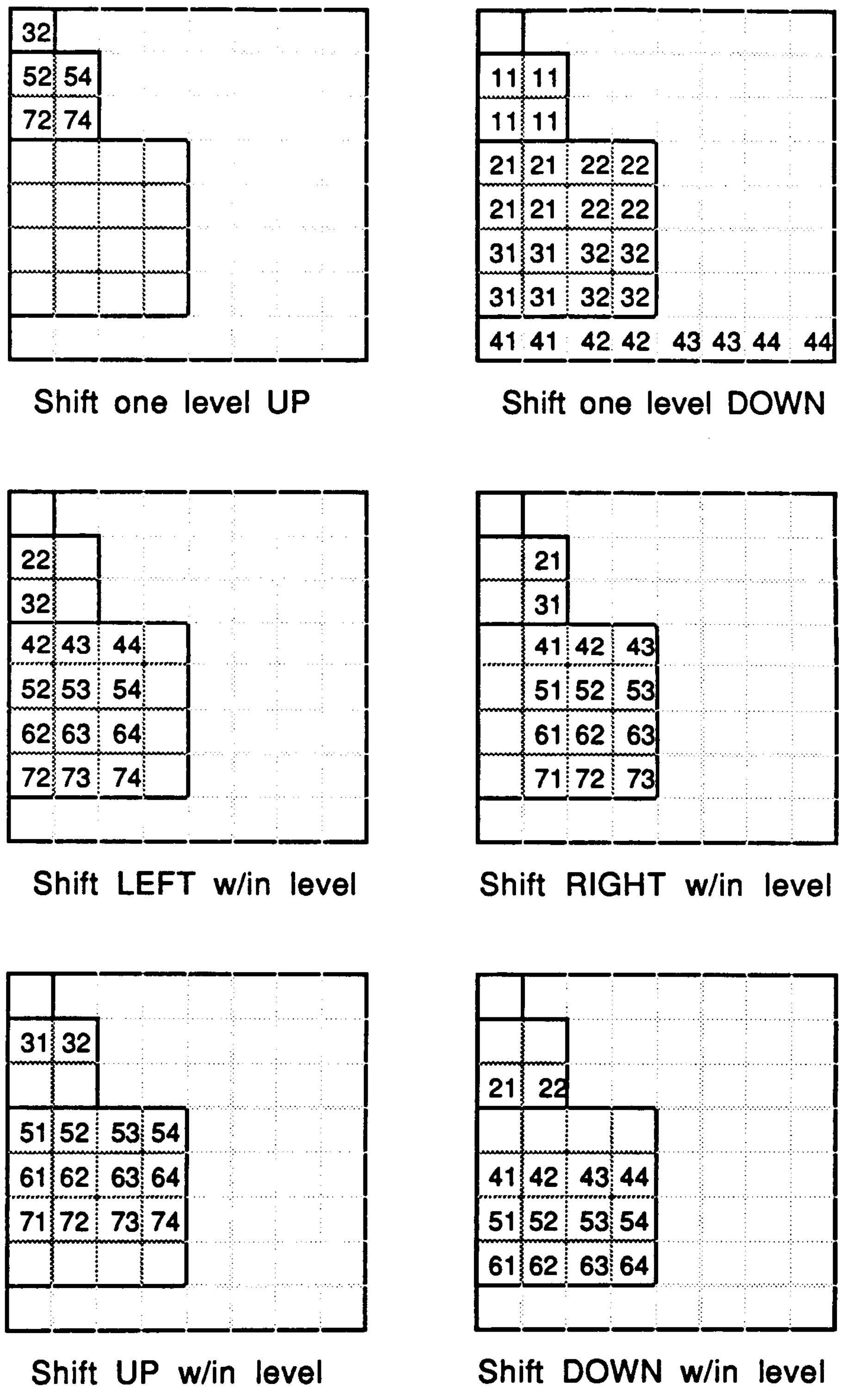


Figure 4.3. Pyramid shift movements.
(the input pyramid is given in Figure 4.2.)

presented in section 4.2, except that in order to keep the computation within the pyramid boundaries, the function *xshift* is used for interprocessor communication instead of the regular Parallel Pascal *shift*. This convolution which uses the *xshift* function will be refered to as a *constrained* convolution.

4.5.1. Algorithm

The algorithm involves two main steps:

1 - Based on the weight factor, the convolution kernel is generated (see Figure 4.4.a). The following equation is used:

$$W[x, y] := \hat{w}[x] * \hat{w}[y] \quad \text{for } -2 \le x,y \le 2$$

$$\textit{where} \;\; \hat{w}[0] = a, \;\; \hat{w}[\pm 1] = b, \;\; \text{and} \;\; \hat{w}[\pm 2] = c$$

$$a = \textit{weight factor}, \;\; b = 0.25, \;\; \text{and} \;\; c = 0.25 - 0.5 * a$$

If the value of 'a' is equal to 0.4, the convolution kernel will resemble the gaussian probability density function (thus the name of Gaussian pyramid).

2 - $(n - 1)$ iterations are performed for a $2^n \times 2^n$ array. At each iteration, the low-pass filtering is propagated one level up; thus for a n-level pyramid, n-1 iterations are needed to propagate the filtering from the lowest level to the highest. An iteration consists of the following:

(i) - performing a *constrained* convolution at each level with the function *xconv5*

(ii) - shifting the results one level up with the function *gather*

(iii) - restoring the input image at the base of the pyramid, i.e. at the lowest level.

4.5.2. Results

The Gaussian algorithm was coded in Parallel Pascal using the high level pyramid functions xshift and gather, and was executed in the MPP. Two types of images were used: images with 8-bit integer elements and images with 32-bit floating point elements. See table 4.2 for the obtained execution times.

0.0025	0.0125	0.0200	0.0125	0.0025
0.0125	0.0625	0.1000	0.0625	0.0125
0.0200	0.1000	0.1600	0.1000	0.0200
0.0125	0.0625	0.1000	0.0625	0.0125
0.0025	0.0125	0.0200	0.0125	0.0025

(4.4.a.) Gaussian convolution kernel for a = 0.4.

0.0000	0.0000	0.0000	0.0000	0.0000
0.0000	0.0125	0.0200	0.0125	0.0000
0.0000	0.0200	0.1600	0.0200	0.0000
0.0000	0.0125	0.0200	0.0125	0.0000
0.0000	0.0000	0.0000	0.0000	0.0000

(4.4.b.) Modified convolution kernel.

Figure 4.4. Pyramid convolution kernels.

Table 4.2. Gaussian Filtering execution times.

	average time measured
8-bit integers	1.178 sec
32-bit reals	1.225 sec

4.6. Laplacian Filtering

In this operation a Laplacian pyramid is generated, in which each image is a band-pass filtered copy of the image below. A band-pass filtered image at level **k** is equal to the difference between the low-pass filtered images at level **k** and at level **k-1** (expanded to the size of level **k**). In other words, it is equal to the difference between the corresponding two successive levels of the Gaussian pyramid.

4.6.1. Algorithm

The algorithm consists of three steps:

1 - Generate the Gaussian pyramid (see section 4.3.1)

2 - Generate another pyramid by expanding every level of the Gaussian pyramid. The *expand* operation is the opposite of the *reduction* operation of the Gaussian algorithm (steps 2i and 2ii of section 4.3.1). From [15], this operation is defined as follows:

Level **k** = expand (Level **k**–1)

$$\mathit{Level}\ k[i,j] = 4 * \sum_{x=-2}^{2} \sum_{y=-2}^{2} W[x,y] * \text{Level } k-1 \left[\frac{i+x}{2}, \frac{j+y}{2} \right]$$

This equation is implemented as follows:

(i) - perform a *constrained* convolution at every level with the modified kernel of Figure 4.4.b.

(ii) - shift the result one level down with the function *gather.*

Clearly, the highest level of the pyramid generated from the expand operation is equal to zero.

3 - Substract the two pyramids to obtain the Laplacian pyramid.

4.6.2. Results

The Laplacian algorithm was coded in Parallel Pascal and was executed in the MPP. See table 4.3 for the obtained execution times.

Table 4.3. Laplacian Filtering execution times.

	average time measured
8-bit integers	1.374 sec
32-bit reals	1.432 sec

Summary

Convolution operations can be implemented onthe MPP in Parallel Pascal with reasonable efficiency. A full parallel pyramid architecture has been emulated on the MPP using the general purpose data manipulation programming tools xshift and gather. While the efficiency of these tools is relatively low for any given data manipulation, they permit the simulation to be programmed at a very high level. The ability of this emulation to process real size images in a resonable amount of time has been demonstrated. A much faster implementation of the pyramid building operations could be achieved by programming them more directly.

It was found that the execution times of the pyramid operations on the MPP were dominated by the shifts within and between levels of the pyramid. The *constrained* convolution, which uses the shifts within levels operation, has an execution time of 6.22 msec for 8-bit integer data, and 8.15 msec for 32-bit floating point data. The *xshift* function seriously impacts the execution time of the convolution operation for 8-bit integer data; this *constrained* convolution is slower than the regular convolution of section 4.2 by a factor of 6. However, the *constrained* convolution for 32-bit floating point is slower than the regular convolution by only a factor of 1.5.

For the shifting between levels, a general primitive function, *gather*, is used. This function performs a shift of all levels, and it is not as efficient as only performing the shift at the required level. For example, in the first iteration of the Gaussian algorithm, a shift, from the lowest level to the next level up, is required with a maximum x-distance of 64 and y-distance of 32. This maximum distances decrease by half for each level up in the pyramid, and as little as a maximum x-distance of 2 and y-distance of 1 are required for a shift to the highest pyramid level (in the last iteration). This shift to the highest pyramid level can be much more efficiently performed if, rather than using a general operation for shifts between levels, a special operation that performs this particular shift is used.

CHAPTER 5

CONCLUSION

This thesis presented several important algorithms developed for a SIMD mesh-connected parallel computer, the Massively Parallel Processor (MPP). For each algorithm the obtained timing and performance results were discussed. From these results, it was found that for Boolean operations, up to an order of magnitude is lost in performance with the current implementation of the high level language Parallel Pascal and the MCU control overhead. Therefore, assembly coding is very desirable for efficient programming of Boolean operations on the MPP. The implementation of 8-bit integer operations is also not very efficient, but it is significantly better than the Boolean implementation. On the other hand, the implementation of floating point operations in Parallel Pascal is in general quite adequate for most applications.

In the current MPP environment, efficient results can be obtained for algorithms using floating point operations, unless bit plane or serial control (by the MCU) operations are also substantially used. For example, the performance results of the FFT and the Convolution algorithms are quite reasonable. These algorithms are presented in Chapter 2 and 4 respectively. The FFT achieves a rate of 87 MFLOPS with 54% of the time performing arithmetic operations and 46% shift operations (communication). The Convolution algorithm achieves a rate of 155

MFLOPS with 74% of the time performing arithmetic operations. It was estimated that if these algorithms were microcoded then a 20% decrease of execution time could be obtained.

For the permutation algorithms presented in Chapter 2, the results using 32-bit floating point data are the most efficient. Comparatively, the *exchange* permutation with a transfer ratio always less than 10, is the fastest permutation. It is followed by the *butterfly* permutation again with a transfer ratio always less than 10. Finally, the *bit reversal* is faster than the *shuffle* for permutations involving 5 to 7 bits of the elements' binary address. The *bit reversal* and the *shuffle* have a maximum transfer ratio equal to 20, and 50 respectively.

The heuristic algorithm for warp mapping always takes fewer than the n^2 iterations and n^2 rotations required by a simple exhaustive algorithm. The heuristic algorithm programmed in Parallel Pascal is not very efficiently implemented on the MPP because it uses a significant number of Boolean operations and it involves the MCU for dynamic control of the flow of execution. As a result, the measured execution times differ from the estimated optimal times by approximately a factor of 10. However, the heuristic algorithm is still faster than the simple algorithm (and more so if optimaly coded), in favorable cases such as when few elements are to be moved or when there is a lot of uniformity in the movement of the elements.

In conclusion, a number of important algorithms have been effectively programmed for the MPP in Parallel Pascal. In all cases the algorithms were first rapidly developed using a program development system on a conventional computer and were then ported to the MPP. For high efficiency, the next and final stage of program development would be to program the critical sections of these algorithms in *Pearl*, the PE array assembly language. While some improvement in performance could be achieved with a compiler optimization stage, more significant improvements could be made by improving the implementation of some of the less efficient primitive operations highlighted in Table 1.1. For floating-point number intensive algorithms the Parallel Pascal implementation is reasonably efficient; in all cases the Parallel Pascal environment was efficient enough for algorithm prototyping.

APPENDIX A

PERMUTATIONS AND FFT PARALLEL PASCAL CODE

Parallel Pascal code of the **exchange** permutation:

```
(* This procedure exchanges the bit which number is equal to bitnum
procedure exchange(var mx:pla; var tn, gamma, bitnum:integer;
                   var id:pli8, var dircol:boolean);
                   extern rh, ch;                               *)

var
  tmx :$1;
  bitn, weight, x, y: integer;

(*******************)
function bitplane(matrix:$5; bitnum:integer):plb;
var x, z : integer;
begin
  z := 1;
  for x := 1 to bitnum do
    z := z * 2;

  matrix := matrix div z;
  bitplane := odd(matrix);
end;
(*******************)

begin

  bitn := bitnum - 1;

  (* calculate amount of shift *)
  weight := 1;
  for x := 1 to bitn do
     weight := weight * 2;

  if dircol then begin
         x := weight; y := 0
         end
```

```
          else begin
          x := 0; y := weight
          end;

 tmx := mx;
 where bitplane(id,bitn) do
    mx := shift(mx, -x, -y) (* shift down *)
   otherwise
    mx := shift(tmx, x, y);  (* shift up *)

end;
```

Parallel Pascal code of the **shuffle** permutation:

```
(* Shuffle permutation; msb goes to the lsb position
procedure shuffle(var mx:pla; var tn, gamma:integer;
                  var id:pli8, var dircol:boolean);
                  extern rh, ch;                            *)

var tmx1, tmx2: $1;
   num, x, y: integer;

begin

 if dircol then begin
          x := 1; y := 0
          end
          else begin
          x := 0; y := 1
          end;

 tmx1 := mx;
 tmx2 := mx;

 (* map the upper half of the input matrix *)
 num := 2;
 while num < tn do
   begin
    tmx1 := shift(tmx1, -x, -y); (* shift down *)
    where id = num do
```

```
      mx := tmx1;
    num := num + 2;
  end;

 (* map the lower half of the input matrix *)
 num := tn - 3;
 while num > 0 do
  begin
    tmx2 := shift(tmx2, x, y);   (* shift up *)
    where id = num do
      mx := tmx2;
    num := num - 2;
  end;

end;
```

Parallel Pascal code of the **sub-shuffle** permutation:

```
(* Sub-shuffle permutation; bit number=(numbits-1) goes to lsb position
This procedure changes the variable id.
procedure subshuffle(var mx:pla; var tn, gamma, numbits:integer;
                     var id:pli8, var dircol:boolean);
                     extern rh, ch;                          *)

var tmx1, tmx2: $1;
   ttn, num, x, y: integer;

begin

 (* divide the input matrix into groups of size = 2**numbits *)
 if numbits = gamma then ttn := tn
    else begin
      ttn := 1;
      for i := 1 to numbits do
         ttn := ttn * 2;
      id := id mod ttn;
    end; (* of else *)

 if dircol then begin
         x := 1; y := 0
```

```
          end
          else begin
          x := 0; y := 1
          end;

 tmx1 := mx;
 tmx2 := mx;

 (* map the upper halves *)
 num := 2;
 while num < ttn do
   begin
     tmx1 := shift(tmx1, -x, -y); (* shift down *)
     where id = num do
         mx := tmx1;
     num := num + 2;
   end;

 (* map the lower halves *)
 num := ttn - 3;
 while num > 0 do
   begin
     tmx2 := shift(tmx2, x, y);   (* shift up *)
     where id = num do
         mx := tmx2;
     num := num - 2;
   end;

end;
```

Parallel Pascal code of the **super-shuffle** permutation:

```
(* SUPer-shuffle permutation;msb bit goes to position number=(gamma-numbits+1)
This procedure changes the variable id.
procedure supshuffle(var mx:pla; var tn, gamma, numbits:integer;
                  var id:pli8, var dircol:boolean);
                  extern rh, ch;                              *)

var tmx1, tmx2: $1;
   num, x, y, grpsize, ttn: integer;
```

```
begin

 grpsize := 1;
 for x := 1 to (gamma - numbits) do
    grpsize := grpsize * 2;

 ttn := 1;
 for x := 1 to numbits do
    ttn:= ttn * 2;

 id := id div grpsize;

 if dircol then begin
          x := grpsize; y := 0
          end
          else begin
          x := 0; y := grpsize
          end;

 tmx1 := mx;
 tmx2 := mx;

 (* map the upper half of the input matrix *)
 num := 2;
 while num < ttn do
   begin
    tmx1 := shift(tmx1, -x, -y); (* shift down *)
    where id = num do
       mx := tmx1;
    num := num + 2;
   end;

 (* map the lower half of the input matrix *)
 num := ttn - 3;
 while num > 0 do
   begin
    tmx2 := shift(tmx2, x, y);   (* shift up *)
    where id = num do
       mx := tmx2;
    num := num - 2;
   end;

end;
```

Parallel Pascal code of the **sub-butterfly** permutation:

```
(* Butterfly and sub-Butterflies mapping
   This procedure changes the variable id
procedure butterfly(var mx:pla; var tn,gamma,numbits:integer;
                    var id:pli8; var dircol:boolean);
                    extern rh, ch;       *)
type
  plb = parallel array[0..$7,0..$8] of boolean;

var
  i, x, y, width, dist: integer;
  shiftup, shiftdwn, mskodd: plb;
  tmx: $1;

begin

  tmx := mx;

  width := 1;
  for i := 1 to (numbits - 1) do
    width := width * 2;    (* 2**(numbits-1) *)

  dist := width - 1;

  id := id mod (width * 2);
  mskodd := odd(id);

  (* create the shift masks *)
  shiftup := (id < width) and mskodd;
  shiftdwn := (id >= width) and not mskodd;

  (* perform the butterfly *)
  if dircol then begin
          x := dist; y := 0;
          end
          else begin
          x := 0; y := dist;
          end;
  where shiftup do
    mx:= shift(tmx, x, y);

  where shiftdwn do
    mx:= shift(tmx, -x, -y);

end;
```

Parallel Pascal code of the **super-butterfly** permutation:

```
(* SUPer Butterfly mapping
procedure supbutfly(var mx:pla; var tn,gamma,numbits:integer;
                    var id:pli8; var dircol:boolean);
                    extern rh, ch;        *)
(* this procedure changes variable id *)

type
  plb = parallel array [0..$7,0..$8] of boolean;

var
  x, y, grpsize, diff, width, dist: integer;
  shiftup, shiftdwn: plb;
  tmx: $1;

begin

  tmx := mx;

  diff := gamma - numbits;

  grpsize := 1;
  for x := 1 to diff do
    grpsize := 2 * grpsize;

  id := id div grpsize;
  mskodd := odd(id);

  width := 1;
  for x := 1 to (numbits - 1) do
    width := 2 * width;         (* 2**numbits / 2 *)
  dist := (width - 1) * grpsize;

  (* create the shift masks *)
  shiftup := (id < width) and mskodd;
  shiftdwn := (id >= width) and not mskodd;

  (* perform the butterfly *)
  if dircol then begin
          x := dist; y := 0;
          end
          else begin
          x := 0; y := dist;
          end;

  where shiftup do
    mx:= shift(tmx, x, y);

  where shiftdwn do
    mx:= shift(tmx, -x, -y);

end;
```

Parallel Pascal code of the **bit reversal** permutation:

```
(*  BIT REVersal using the Bit swapping method
procedure bitrevb(var mx:pla; var tn, gamma: integer; var id:pli8;
         var dircol:boolean); extern rh, ch;
*)
type  plb = parallel array[0..$6,0..$7] of boolean;

var lo, hi, wlo, whi : integer;
    numshift : integer;
    x, y : integer;
    tmx : $1;

(******************)
function bitplane(matrix:$4; bitnum:integer):plb;
var x, z : integer;
begin
   z := 1;
```

```
  for x := 1 to bitnum do
    z := z * 2;

  matrix := matrix div z;
  bitplane := odd(matrix);
end;
(*******************)

begin

(* initiallization *)
  (* least significant bit *)
  lo := 0;
  wlo := 1;

  (* most significant bit *)
  hi := gamma -1;
  whi := tn div 2;

(* Perform the bit reversal *)
  while hi > lo do begin

    tmx := mx;
    numshift := whi - wlo;

    if dircol then begin
       x := numshift; y := 0;
      end
      else begin
       x := 0; y := numshift;
      end;

    (* case when more significant bit is 1 and less sign. bit is 0 *)
    (* shift down *)
    where bitplane(id,lo) < bitplane(id,hi) do
       mx := shift(tmx, -x, -y);

    (* case when more significant bit is 0 and less sign. bit is 1 *)
    (* shift up *)
    where bitplane(id,lo) > bitplane(id,hi) do
       mx := shift(tmx, x, y);

    lo := lo + 1;
    hi := hi - 1;
    wlo := wlo * 2;
    whi := whi div 2;

  end;
end;
```

Parallel Pascal code of the **sub-bit reversal** permutation:

```
(*  SUB Bit Reversal using the Bit swapping method
procedure subbrb(var mx:pla; var tn, gamma, numbits: integer; var id:pli8;
        var dircol:boolean); extern rh, ch;
*)
type  plb = parallel array[0..$7,0..$8] of boolean;

var lo, hi, wlo, whi : integer;
    numshift : integer;
    x, y : integer;
    tmx : $1;

(*******************)
function bitplane(matrix:$5; bitnum:integer):plb;
var x, z : integer;
begin
   z := 1;
   for x := 1 to bitnum do
     z := z * 2;

   matrix := matrix div z;
   bitplane := odd(matrix);
end;
(*******************)

begin

(* initiallization *)
  (* least significant bit *)
  lo := 0;
  wlo := 1;
  (* more significant bit *)
  hi := numbits - 1;
  wlo := 1;
  if numbits = gamma
     then whi := tn div 2
     else begin
          whi := 1;
          for x := 1 to hi do
              whi := whi * 2;
          end;

(* Perform the bit reversal *)
  while hi > lo do begin

    tmx := mx;
    numshift := whi - wlo;

    if dircol then begin
        x := numshift; y := 0;
       end
       else begin
        x := 0; y := numshift;
       end;

    (* case when more significant bit is 1 and less sign. bit is 0 *)
    (* shift down *)
    where bitplane(id,lo) < bitplane(id,hi) do
        mx := shift(tmx, -x, -y);

    (* case when more significant bit is 0 and less sign. bit is 1 *)
    (* shift up *)
    where bitplane(id,lo) > bitplane(id,hi) do
        mx := shift(tmx, x, y);

    lo := lo + 1;
    hi := hi - 1;
    wlo := wlo * 2;
    whi := whi div 2;

  end;
end;
```

Parallel Pascal code of the **super-bit reversal** permutation:

```
(*  SUPer Bit Reversal using the Bit swapping method
procedure supbrb(var mx:pla; var tn, gamma, numbits: integer; var id:pli8;
        var dircol:boolean); extern rh, ch;
*)
type  plb = parallel array[0..$7,0..$8] of boolean;

var lo, hi, wlo, whi : integer;
    numshift : integer;
    x, y : integer;
    tmx : $1;

(*******************)
function bitplane(matrix:$5; bitnum:integer):plb;
var x, z : integer;
begin
   z := 1;
   for x := 1 to bitnum do
     z := z * 2;

   matrix := matrix div z;
   bitplane := odd(matrix);
end;
(*******************)

begin

(* initiallization *)
  (* most significant bit *)
  hi := gamma - 1;
  whi := tn div 2;

  (* less significant bit *)
  lo := gamma - numbits;
  wlo := 1;
  for x := 1 to lo do
       wlo := wlo * 2;

(* Perform the bit reversal *)
  while hi > lo do begin

    tmx := mx;
    numshift := whi - wlo;

    if dircol then begin
       x := numshift; y := 0;
       end
       else begin
       x := 0; y := numshift;
       end;

    (* case when more significant bit is 1 and less sign. bit is 0 *)
    (* shift down *)
    where bitplane(id,lo) < bitplane(id,hi) do
       mx := shift(tmx, -x, -y);

    (* case when more significant bit is 0 and less sign. bit is 1 *)
    (* shift up *)
    where bitplane(id,lo) > bitplane(id,hi) do
       mx := shift(tmx, x, y);

    lo := lo + 1;
    hi := hi - 1;
    wlo := wlo * 2;
    whi := whi div 2;

  end;
end;
```

Parallel Pascal code of the **Fast Fourier Transform**:

```
(*
procedure fft(var matrixr:plr; var matrixi:plr; var tn, gamma:integer;
             var id:pli8; var dircol, modul, ifft:boolean);
             extern rh, ch;
*)

type plb = parallel array[0..$9,0..$10] of boolean;

var tn2, i, ti, tl : integer;
    x, y : integer;
    konst, pi : real;
    mskodd : plb;
    power : $5;
    tpower : $1;
    weightr, weighti : $1;


(*--------------------------------------------------------------*)
procedure powergen(var power:$5; var tn2, gamma, stage:integer;
                   var id:$5);

var k, tk : $5;
    twotoi, twotox : integer;
    i, x : integer;

begin
   if stage = 1 then power := 0
    else begin
     if stage = gamma then
        where id >= tn2 do power := power + 1
      else

      begin
       x := gamma - stage;
       twotox := 1;
       for i := 1 to x do
          twotox := twotox * 2;
       k := id div twotox;
       power := 0;
       twotoi := 2;

       x := tn2 div twotoi;

       i := 1;
       while i < stage do begin

          tk := k mod twotoi;
         where (tk >= (twotoi div 2)) do
            power := power + x;
         twotoi := twotoi * 2;
          i := i + 1;
          x := x div 2;

       end;
      end;
   end;

end;

(*-----------------------------------------------------------*)
procedure complexmult(var matrixr, matrixi, weightr, weighti:$1;
                    var mskodd:plb; var dircol:boolean);

var tmatrixr, tmatrixi : $1;
    matr : $1;
    x, y: integer;

begin

   if dircol then begin
            x := 1; y := 0
            end
            else begin
            x := 0; y := 1
            end;

   tmatrixr := matrixr;
   tmatrixi := matrixi;
   where mskodd do begin
     tmatrixr := shift(tmatrixr, -x, -y);
     tmatrixi := shift(tmatrixi, -x, -y);
    end;
   where not mskodd do begin
     matrixr := shift(matrixr, x, y);
     matrixi := shift(matrixi, x, y);
    end;
   matr := matrixr;
   matrixr := (weightr * matrixr)-(weighti * matrixi);
   matrixi := (weightr * matrixi)+(weighti * matr);
   matrixr := tmatrixr + matrixr;
   matrixi := tmatrixi + matrixi;

end;

(*-----------------------------------------------------------*)
begin (* fft.pl *)
```

```
pi := 3.1415926535;
tn2 := tn div 2;
konst := pi / tn2;
mskodd := odd(id);

(* check if want IFFT, if so divide by the number of elements*)
  if ifft then begin
       matrixr := matrixr / tn;
       matrixi := matrixi / tn;
     end;

(* to achieve modulation multiply odd input elements by -1 *)
(* ie. change sign *)

if modul then begin
  where mskodd do begin
    matrixr := - matrixr;
    matrixi := - matrixi;
  end;
 end;

for i := 1 to gamma  do
 begin
  ti := i;
  if ti= 1 then begin
    butterfly(matrixr, matrixi, tn, gamma, gamma, mskodd, id, dircol);
    power := 0;
    weightr := 1;
    weighti := 0;
   end
  else begin
    if ti = gamma then
          shuffle(matrixr, matrixi, tn, gamma, id, dircol)
      else begin
          tl := gamma - ti + 1;
          butterfly(matrixr, matrixi, tn, gamma, tl, mskodd, id, dircol);
      end;
    powergen(power, tn2, gamma, ti, id);
    tpower := power * konst;
    weightr := cos(tpower);
    weighti := - sin(tpower);
   end;
```

```
    where mskodd do begin
       weightr := - weightr;
       weighti := - weighti;
      end;

    (* check if want IFFT, if so change the exp to positive power *)
    if ifft then weighti := - weighti;

    complexmult(matrixr, matrixi, weightr, weighti, mskodd, dircol);

   end;

end;
```

Parallel Pascal code of the **butterfly** permutation for
the **Fast Fourier Transform**:

```
(*
procedure butterfly(var matrixr, matrixi:plr; var tn,gamma,numbits:integer;
                    var mskodd:plb; var id:pli8; var dircol:boolean);
                    extern rh, ch;        *)


var
  i, x, y, z, width, dist: integer;
  shiftup, shiftdwn: $6;
  tmxr, tmxi: $1;

begin

  tmxr := matrixr;
  tmxi := matrixi;

  width := 1;
  for i := 1 to (numbits - 1) do
    width := width * 2;

  dist := width - 1;

  (* create the shift masks *)
```

```
shiftup := (id < width) and mskodd;
shiftdwn := (id >= width) and not mskodd;

width := width * 2;

z := 0;
if dircol then begin

while width < tn do begin
   where id >= width do begin
     shiftup := shift(shiftup, -width, z);
     shiftdwn := shift(shiftdwn, -width, z)
    end;
   width := width * 2;
  end;

end
else begin

while width < tn do begin
   where id >= width do begin
     shiftup := shift(shiftup, z, -width);
     shiftdwn := shift(shiftdwn, z, -width)
    end;
   width := width * 2;
  end;

end;

(* perform the butterfly *)
if dircol then begin
        x := dist; y := 0;
        end
        else begin
        x := 0; y := dist;
        end;

where shiftup do begin
  matrixr:= shift(tmxr, x, y);
  matrixi:= shift(tmxi, x, y);
 end;

where shiftdwn do begin
```

```
    matrixr:= shift(tmxr, -x, -y);
    matrixi:= shift(tmxi, -x, -y);
   end;

end;
```

Parallel Pascal code of the **shuffle** permutation for
the **Fast Fourier Transform**:

```
(*
procedure shuffle(var matrixr, matrixi:plr; var tn, gamma:integer;
                  var id:pli8, var dircol:boolean);
                  extern rh, ch;                                  *)

var tmx1r, tmx1i, tmx2r, tmx2i : $1;
    num, x, y: integer;

begin

  if dircol then begin
          x := 1; y := 0
          end
          else begin
          x := 0; y := 1
          end;

  num := 2;

  tmx1r := matrixr;
  tmx1i := matrixi;
  tmx2r := matrixr;
  tmx2i := matrixi;

  while num < tn do
    begin
      tmx1r := shift(tmx1r, -x, -y);
      tmx1i := shift(tmx1i, -x, -y);
      where id = num do begin
          matrixr := tmx1r;
          matrixi := tmx1i;
```

```
        end;
      num := num + 2;
    end;

  num := tn - 3;
  while num > 0 do
    begin
      tmx2r := shift(tmx2r, x, y);
      tmx2i := shift(tmx2i, x, y);
      where id = num do begin
          matrixr := tmx2r;
          matrixi := tmx2i;
        end;
      num := num - 2;
    end;

end;
```

Parallel Pascal code of the **bit reversal** permutation for
the **Fast Fourier Transform**:

```
(*  BIT REVersal using the Bit swapping method
procedure bitrevb(var matrixr, matrixi:plr; var tn, gamma: integer; var id:pli8;
        var dircol:boolean); extern rh, ch;
*)
type  plb = parallel array[0..$7,0..$8] of boolean;

var lo, hi, wlo, whi : integer;
    numshift : integer;
    x, y : integer;
    tmxr, tmxi : $1;

(******************)
function bitplane(matrix:$5; bitnum:integer):plb;
var x, z : integer;
begin
  z := 1;
  for x := 1 to bitnum do
    z := z * 2;
  matrix := matrix div z;
  bitplane := odd(matrix);
end;
(******************)

begin

(* initiallization *)
  (* least significant bit *)
  lo := 0;
  wlo := 1;

  (* most significant bit *)
  hi := gamma -1;
  whi := tn div 2;

(* Perform the bit reversal *)
  while hi > lo do begin

    tmxr := matrixr;
    tmxi := matrixi;
    numshift := whi - wlo;

    if dircol then begin
        x := numshift; y := 0;
       end
       else begin
        x := 0; y := numshift;
       end;

    (* case when more significant bit is 1 and less sign. bit is 0 *)
    (* shift down *)
    where bitplane(id,lo) < bitplane(id,hi) do
     begin
        matrixr := shift(tmxr, -x, -y);
        matrixi := shift(tmxi, -x, -y);
     end;

    (* case when more significant bit is 0 and less sign. bit is 1 *)
    (* shift up *)
    where bitplane(id,lo) > bitplane(id,hi) do
     begin
        matrixr := shift(tmxr, x, y);
```

```
            matrixi := shift(tmxi, x, y);
          end;

        lo := lo + 1;
        hi := hi - 1;
        wlo := wlo * 2;
        whi := whi div 2;

      end;
    end;
```

REFERENCES

[1] Special issue on "Supercomputers - their impact on science and technology", *Proceedings of the IEEE*, vol. 72 #1, January 1984.

[2] "Computing at the Frontiers of Science and Engineering", *Computer*, vol. 18 #11, November 1985.

[3] Special Issue, *Communications of the ACM*, vol. 28 #4, April 1985.

[4] R. W. Hockney and C. R. Jesshope, *Parallel Computers*, Arrowsmith Ltd., Bristol, 1983.

[5] A. P. Reeves, "The Massively Parallel Processor: A Highly Parallel Scientific Computer", *International Worshop on Data Analysis in Astronomy*, Erice, Italy, April 1986.

[6] J. L. Potter ed., *The Massively Parallel Processor*, MIT Press, 1985.

[7] *General Description of the MPP*, NASA, Goddard Space Flight Center, April 1983.

[8] A. P. Reeves, "Parallel Pascal: An Extended Pascal for Parallel Computers", *Journal of Parallel and Distributed Computing 1*, pp.64-80, 1984.

[9] J. Fisher ed., *MPP User's Guide*, NASA, Goddard Space Flight Center, January 1986.

[10] K. E. Batcher, "Design of a Massively Parallel Processor", *IEEE Transactions on Computers*, C-29(9) pp.836-840, September 1981.

[11] J. Devaney, private communication.

[12] E. O. Brigham, *The Fast Fourier Transform*, Prentice-Hall, Englewood Cliffs, New Jersey, 1974.

[13] A. P. Reeves and C. H. Moura, "Data Mapping and Rotations Functions for The Massively Parallel Processor", *Proceedings of IEEE Computer Society Workshop on Computer Architectures for Pattern Analysis and Image Data Base Management*, 1985.

[14] D. H. Ballard and C. M. Brown, *Computer Vision*, Prentice Hall, Englewood Cliffs, New Jersey, 1982.

[15] P. J. Burt, "The Pyramid as a Stucture for Efficient Computation".

[16] P. J. Burt, "Laplacian Pyramid as a Compact Image Code", *IEEE Transactions on Communications*, Vol. 31 #4, pp.532-540, 1983.

APPENDIX C: PERMUTATION FUNCTION DOCUMENTATION

NAME

FFT - fast fourier transform

SYNOPSIS

```
{$library fft.pl}

procedure fft(var mxr, mxi:plr; var n,
              var dircol, modul, ifft:boolean);
              extern rh, ch;
```

TYPES

plr = parallel array [rl..rh,cl..ch] of real;
pli8 = parallel array [rl..rh,cl..ch] of 0..255;

EXTERN CONSTANTS

rh = the largest row number of the input matrix
ch = the largest column number of the input matrix

VARS

Id is the index matrix. It is equal to id1 when dircol is true, and equal to id2 when dircol is false (see mat(2) for information on id1 and id2).

DESCRIPTION

The *fft* is performed concurrently to n sets of n elements. A set of elements is either a row (if dircol is false) or a column (if dircol is true) of the matrices *mxr,* the real part, and *mxi,* the imaginary part. The *fft* procedure uses the butterfly, shuffle, and bit reversal permutations from the library **fftpermut.pl**. Modulation is performed when 'modul' is true, and an IFFT is performed instead of the FFT if 'ifft' is true.

AUTHOR

Maria C. Gutierrez

NAME

exchange – data permutation

SYNOPSIS

{$library exchanges.pl}

```
procedure exchange(var mx:pla; var n, gamma,
              var dircol:boolean); extern rh, ch;
```

TYPES

pla = parallel array [rl..rh,cl..ch] of btype;
pli8 = parallel array [rl..rh,cl..ch] of 0..255;
Where btype is any data type

EXTERN CONSTANTS

rh = the largest row number of the input matrix
ch = the largest column number of the input matrix

VARS

Id is the index matrix. It is equal to id1 when dircol is true, and equal to id2 when dircol is false (see mat(2) for information on id1 and id2).

DESCRIPTION

The permutation is performed concurrently to n sets of n elements. A set of elements is either a row (if dircol is false) or a column (if dircol is true) of the matrix *mx*. For each set, the *exchange* permutation consists of complementing bit 'bitnum' of the binary input address, where 'bitnum' is in the range of 1 to gamma. Thus, this permutation consists of exchanging every pair of elements, where two elements form a pair if their addresses are the same except for the kth bit.

AUTHOR

Maria C. Gutierrez

NAME

shuffle, sub–shuffle, super–shuffle – data permutations

SYNOPSIS

```
{$library shuffles.pl}

procedure shuffle(var mx:pla; var n, gamma:integer;
                var dircol:boolean); extern rh, ch;

procedure subshuffle(var mx:pla; var n, gamma,
                var dircol:boolean); extern rh, ch;

procedure supshuffle(var mx:pla; var n, gamma,
                var dircol:boolean); extern rh, ch;
```

TYPES

pla = parallel array [rl..rh,cl..ch] of btype;
pli8 = parallel array [rl..rh,cl..ch] of 0..255;
Where btype is any data type

EXTERN CONSTANTS

rh = the largest row number of the input matrix
ch = the largest column number of the input matrix

VARS

Id is the index matrix. It is equal to id1 when dircol is true, and equal to id2 when dircol is false (see mat(2) for information on id1 and id2).

DESCRIPTION

The permutations are performed concurrently to n sets of n elements. A set of elements is either a row (if dircol is false) or a column (if dircol is true) of the matrix *mx*. For each set, the *shuffle* permutation consists of a circular left shift of the bits of the input address. The resulting permutation consists of splitting in half the set of n elements and then interleaving them like in a perfect card shuffle. Similarly, the *sub–shuffle* and the *super–shuffle* consist of a circular left shift of the 'numbits' least significant bits (sub–shuffle) and most significant bits (super–shuffle);

AUTHOR

Maria C. Gutierrez

NAME

sub-butterfly, super-butterfly - data permutations

SYNOPSIS

```
{$library .pl}

procedure butterfly(var mx:pla; var n, gamma,
                var dircol:boolean); extern rh, ch;

procedure supbutfly(var mx:pla; var n, gamma,
                var dircol:boolean); extern rh, ch;
```

TYPES

pla = parallel array [rl..rh,cl..ch] of btype;
pli8 = parallel array [rl..rh,cl..ch] of 0..255;
Where btype is any data type

EXTERN CONSTANTS

rh = the largest row number of the input matrix
ch = the largest column number of the input matrix

VARS

Id is the index matrix. It is equal to id1 when dircol is true, and equal to id2 when dircol is false (see mat(2) for information on id1 and id2).

DESCRIPTION

The permutations are performed concurrently to n sets of n elements. A set of elements is either a row (if dircol is false) or a column (if dircol is true) of the matrix *mx*. For each set, the *butterfly* permutation consists of exchanging bit 'numbits' (msb) with bit '1' (lsb). Similarly, the *super-butterfly* consist of exchanging bit 'gamma' (msb) with bit

AUTHOR

Maria C. Gutierrez

NAME

bit reversal, sub-bit reversal, super-bit reversal – data permutations

SYNOPSIS

```
{$library bitrevbs.pl}

procedure bitrevb(var mx:pla; var n, gamma:integer;
                  var dircol:boolean); extern rh, ch;

procedure subbrb(var mx:pla; var n, gamma,
                  var dircol:boolean); extern rh, ch;

procedure supbrb(var mx:pla; var n, gamma,
                  var dircol:boolean); extern rh, ch;
```

TYPES

pla = parallel array [rl..rh,cl..ch] of btype;
pli8 = parallel array [rl..rh,cl..ch] of 0..255;
Where btype is any data type

EXTERN CONSTANTS

rh = the largest row number of the input matrix
ch = the largest column number of the input matrix

VARS

Id is the index matrix. It is equal to id1 when dircol is true, and equal to id2 when dircol is false (see mat(2) for information on id1 and id2).

DESCRIPTION

The permutations are performed concurrently to n sets of n elements. A set of elements is either a row (if dircol is false) or a column (if dircol is true) of the matrix *mx*. For each set, the *bit reversal* permutation consists of reversing the order of the bits of the input address; it is like taking the mirror image of the address. Similarly, the *sub-bit reversal* and the *super-bit reversal* permutations reverse the 'numbits' least significant bits (sub-bit reversal) and most significant bits (super-bit reversal);

AUTHOR

Maria C. Gutierrez