1-1-1988

Experimental Benchmarks and Initial Evaluation of the Performance of the PASM System Prototype

T. L. Casavant
Purdue University

H. S. Siegel
Purdue University

T. Schwederski
Purdue University

Leah H. Jamieson
Purdue University

A. Fineberg
Purdue University

See next page for additional authors

Authors
T. L. Casavant, H. S. Siegel, T. Schwederski, Leah H. Jamieson, A. Fineberg, and M. J. McPheters

Experimental Benchmarks and Initial Evaluation of the Performance of the PASM System Prototype

T. L. Casavant, H. J. Siegel,
T. Schwederski, L. H. Jamieson,
S. A. Fineberg, M. J. McPheters,
E. C. Bronson, W. Disch,
K. Schurecht, E. H. Loh, C. Ringer,
B. Cox and C. A. Toomey

TR-EE 88-2
January 1988

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

Experimental Benchmarks and Initial Evaluation of the Performance of the PASM System Prototype

T.L. Casavant, H.J. Siegel, T. Schwederski, L.H. Jamieson, S.A. Fineberg, M.J. McPheters, E.C. Bronson, W. Disch, K. Schurecht, E.H. Loh, C. Ringer, B. Cox, C.A. Toomey

Purdue University School of Electrical Engineering

Technical Report #TR-EE 88-2

January, 1988

TABLE OF CONTENTS

Introduction

The work reported here represents experiences with the PASM parallel processing system prototype during its first operational year. Most of the experiments were performed by students in the Fall semester of 1987. The first programming, and the first timing measurements, were made during the summer of 1987 by Sam Fineberg.

The goal of the collection of experiments presented here was to undertake an *Application-driven Architecture Study* of the PASM system as a paradigm for parallel architecture evaluation in general. PASM was an excellent vehicle for experimenting with this evaluation technique due to its unique architectural features. Among these are:

1. A reconfigurable, partitionable multistage circuit-switched network.
2. Support for both SIMD and MIMD programs.
3. Ability to execute hybrid SIMD/MIMD programs.
4. An instruction queue which allows overlap of control-flow and data manipulation between micro-control (MC) units and processing elements (PE). It had been hypothesized that superlinear speed-up over the number of PEs could be attained with this feature, and experimental results verified this.
5. Support for barrier synchronization of MIMD tasks. This feature was exploited in some non-standard ways to show the ability to decouple variant length SIMD instructions into multiple MIMD streams for an overall performance benefit.

This type of study is expected to continue in the future on PASM and other parallel machines at Purdue. This report should serve as a guide for this future work as well.

T.L. Casavant
School of EE
Purdue University
Spring 1988

This work supported by NSF Grant # CCR-8809600, the NSF Software Engineering Research Center (SERC), and SRC Grant # 6925.

PART I

Image Processing

Experiences with Parallel Image Smoothing

Wolfram Disch and Thomas L. Casavant

Abstract

This paper reports results of some of the first programming experiences with the PASM parallel processing system prototype at Purdue. PASM is a **PA**rtitionable SIMD/MIMD system designed for conducting research in parallel computing and for developing software for several applications. The image processing task of *image smoothing* is used to evaluate several features of the PASM architecture. The results include an observation of super-linear speedup over the number of processing elements (PEs) when operating in SIMD mode. This advantage comes from the ability of the microcontrollers (which act as control units in SIMD mode) to execute control flow operations in parallel with the PEs in SIMD programs. Also included is a comparison of computation versus communication overheads. Other experiments and analysis show some of the potential advantages of mixed-mode SIMD/MIMD programs and alude to the problems of structuring programs in this way.

1. Introduction

This paper reports on experimental measurements derived from early programming experiences with simple image processing applications on the PASM parallel processing system prototype at Purdue [SiS81, SiS87]. The intention of the research project surrounding this work is *Application-Driven Architecture Study*, in which easily understood algorithms are implemented as programs, and controlled experimentation is done with respect to variable program characteristics. The reported results are derived from the first segment of a study on Image Processing which includes the following algorithms: *Image Smoothing, Sobel Image Generation and Threshold Determination, Contour Tracing and Block Truncation Coding.* In particular, we describe a parallel implementation of the *image smoothing* phase, programmed and executed on the PASM prototype. The experiments are based on SIMD (single instruction stream – multiple data streams), MIMD (multiple instruction streams – multiple data streams) and Hybrid S/MIMD programs [Fly66]. The results reported are focused on showing the differences with respect to the execution times among the parallel versions and between an efficient serial version in order to determine the usefulness of multi-mode

machines and to shed light on the necessary HLL constructs and semantics useful for this type of computer.

Section 2 presents the problem and overviews PASM and its prototype, while section 3 explains the basic algorithm that was used. Section 4 describes the experiments and the program variations used. Section 5 shows the performance of each program with respect to its execution time and section 6 provides a discussion of the results with respect to the architecture of the PASM system prototype.

2. Background

2.1. Overview of Image Smoothing

Image Smoothing represents one of the first algorithms used when performing Image Processing. The use of Image Smoothing is to filter noise from corrupted signals. One way to smooth an image is the average method [SiS87b]. A window, including a certain number of pixels, is defined. The smoothed value is obtained by averaging the gray levels of the pixels in the window. There are different methods for calculating the average. One is to divide the sum of the gray levels by the number of pixels in the window; i.e. each pixel receives the same weight. Another method distributes 50% to the center pixel and 50% for all the others. Our approach uses the first method.

2.2. Overview of PASM and the PASM Prototype

PASM is a partitionable SIMD/MIMD parallel processing system being designed to include over a thousand processors [SiS81]. It is a dynamically reconfigurable architecture, where the processors can be partitioned to form independent virtual SIMD and/or MIMD machines of various sizes. A 30-processor prototype has been completed [SiS87a] and was used in the algorithms described in this paper.

The *Parallel Computation Unit* contains $N=2^m$ PEs (numbered from 0 to N−1), and an interconnection network. Each *PE* (processing element) is a processor/memory pair. The *PE processors* are sophisticated microprocessors that perform the actual SIMD and MIMD operations. The *PE memory modules* are used by the processors for data storage in SIMD mode and both data and instruction storage in MIMD mode. The *Micro Controllers (MCs)* are a set of $Q=2^q$ processors, numbered from 0 to Q−1, which act as the control units for the PEs in SIMD mode and orchestrate the activities of the

PEs in MIMD mode. Each MC controls N/Q PEs. In SIMD mode, each MC fetches instructions and common data from its associated memory module, executes the control flow instructions (e.g., branches), and broadcasts the data processing instructions to its PEs. In MIMD mode, each MC gets instructions and common data for coordinating its PEs from its memory.

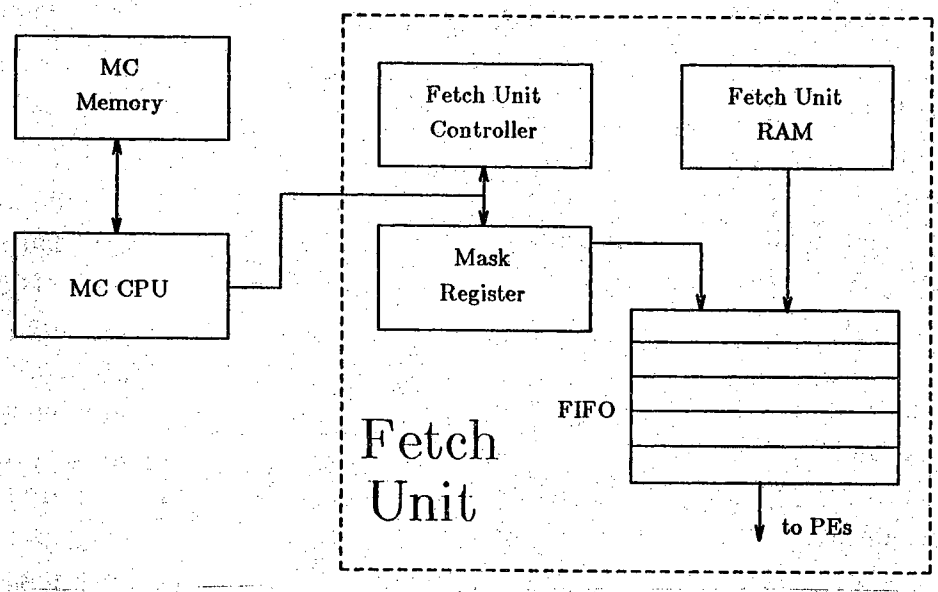


Figure 1: Simplified MC structure.

A 30 processor prototype of the PASM system was completed in December 1986, with N=16 and Q=4. This system employs Motorola MC68000 processors as PE and MC CPUs, with a clock speed of 8 MHz. The interconnection network is a circuit-switched Extra-Stage Cube network, which is a fault-tolerant variation of the multistage cube network. Since knowledge about the MC and the way in which SIMD instructions are implemented with standard MC68000 microprocessors is essential to the understanding of the behavior that was observed in the experiments, the SIMD instruction broadcast mechanism is overviewed below. Consider the simplified MC structure shown in Figure 1. The MC contains a memory module from which the MC CPU reads instructions and data. Whenever the MC needs to broadcast SIMD instructions to its associated PEs, it first sets the Mask Register in the Fetch Unit, thereby determining which PEs will participate in the following instructions. It then writes a control word to the

Fetch Unit Controller which specifies the location and size of a block of SIMD instructions in the Fetch Unit RAM. The Fetch Unit Controller automatically moves this block word by word into the Fetch Unit Queue. Whenever a instruction word is enqueued, the current value of the mask register is enqueued as well. Since the Fetch Unit enqueues blocks of SIMD instructions automatically, the MC CPU can proceed with other operations without waiting for all instructions to be enqueued.

PEs execute SIMD instructions by performing an instruction fetch from a reserved memory area called the SIMD instruction space. Whenever logic in the PEs detects an access to this area, a request for an SIMD instruction is sent to the Fetch Unit. Only after all PEs that are enabled for the current instruction have issued a request is the instruction released by the Fetch Unit FIFO, and the enabled PEs receive and execute the instruction. Disabled PEs do not participate in the instruction and wait until an instruction is broadcast for which they are enabled. This way, switching from SIMD and MIMD mode is reduced to executing a jump instruction to the reserved memory space, and a switch from MIMD to SIMD mode is performed by sending a jump to the appropriate PE MIMD instruction address located in the PE main memory space.

In order to make comparisons of the speed of the PASM prototype relative to other machines and to compare the relative speeds of SIMD and MIMD instruction fetches, the raw performance of PASM in SIMD and MIMD mode was measured and is illustrated in Table 1 in MIPS (millions of integer instructions per second) for two different types of instructions.

Mode	Operation	Processing Rate
SIMD	16-bit Reg.-to-Reg. add	22 MIPS
MIMD	16-bit Reg.-to-Reg. add	18 MIPS
SIMD	16-bit Reg.-to-Mem. add	6.4 MIPS
MIMD	16-bit Reg.-to-Mem. add	6.0 MIPS

Table 1: Prototype Raw Performance

3. Image Smoothing

3.1. Serial Image Smoothing Algorithm

The SISD algorithm offers itself as a good candidate for describing the fundamental structure of the parallel image smoothing algorithm. Assuming an input image, X, and an output image, Y of size $n \times n$ in which each pixel is an 8-bit unsigned integer representing one of 256 gray levels, each pixel in the smoothed image is the average of the gray level of itself and its 8 nearest neighbors. In other words, the average of the gray levels in a 3×3 window is determined for each pixel in X. The top, bottom, left and right edge pixels of the smoothed image are not calculated since their corresponding pixels in the input image do not have 8 adjacent neighbors; they are set to zero.

$$Y_{ij} = \begin{cases} \frac{1}{9} \sum_{k=i-1}^{k=i+1} \sum_{l=j-1}^{l=j+1} X_{kl} & i,j \neq 0 \; ; \; i,j \neq n \\ 0 & \text{otherwise} \end{cases}$$

3.2. Parallel Image Smoothing Algorithm

In the parallel Image Smoothing programs implemented here, the data was equally distributed among 4 and 16 *PEs*, respectively. Thus, each *PE* holds its own square subimage. Since each PE holds only one subimage, data from the borders will have to be transferred from the surrounding PEs in order to calculate the smoothed value of the edge pixels. The *data transfers* occur simultaneously across PEs by using the interconnection network. This represented the first segment of the parallel Image Smoothing algorithm. The second segment includes the *calculation* of the smoothed image which was realized in SIMD, MIMD and Hybrid S/MIMD modes.

Now let us consider the parallel Image Smoothing algorithm in detail. Assuming an $n \times n$ image, the data transfer requires n transfer operations to transmit data from each *PE* border to a specific *PE*. Furthermore, 4 transfers are needed to get the 4 corner pixels. This is shown in figure 2. Therefore, a total of 4n+4 network transfer operations are required to provide each PE with the data needed in order to calculate the smoothed image. Since the architecture of PASM does not support DMA block transfer, each byte must

be transmitted separately. Nevertheless, the data transfer occurs simultaneously across PEs. Each PE sends data through the interconnection network to the corresponding PEs. Then, all PEs must wait for the data to be received. The 4n+4 network transfer operations require 8 network circuit setups to connect the PEs as shown in figure 2.

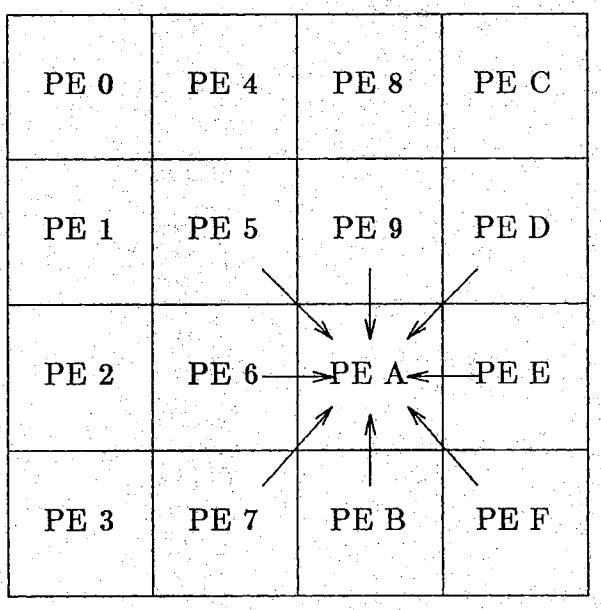


Figure 2: Image Partitioning and PE Interconnection.

Before presenting the calculation portion of the program, the image storage will be discussed. The data for both the input image and the output image are stored in rows as shown in figure 3. The data of the input subimage and the top and bottom data transferred from the other PEs are stored contiguously in a linear data structure. The data coming from the left and the right bordered PEs are stored after the bottom data. This structure permits fast storage of data from an incoming image at a high data rate (e.g. from a video camera) into the PEs, and after processing to another peripheral.

For the calculation of the smoothed value, a window of 9 pixels has been defined. Obviously, the smoothed value is the average of the gray levels of all pixels in the window. In this algorithm, a more efficient programming technique using pointers is used. Referring to Figure 3, five pointers are used. Three pointers are indexing the subimage, the others are the starting address

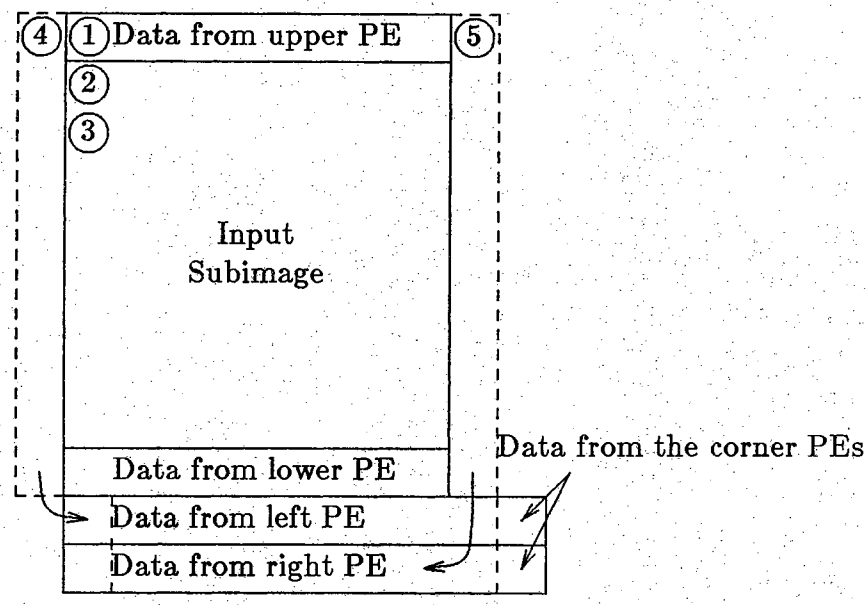


Figure 3: Image Storage Model

of the left and the right border data, respectively. Also, sums of the gray levels of three pixels standing in columns are defined (in the following, *tps* means three pixel sum value). Intermediate tps values are stored in a FIFO queue and added to a variable S. At the beginning of the loop for calculating a row, S is set up with the first three tps. For the next step, the new tps is added, and the oldest tps is subtracted from S. This tps is obtained from the top of the queue. The next tps becomes the top of the queue. Thus, S contains in each step through the loop, the sum of the gray levels of the pixels in the window. The smoothed value is obtained by dividing S by the number of pixels in the window. These steps are executed for each row of the image.

Basically, the parallel algorithm has the same control-flow structure as the SISD algorithm. However, network transfer operations were added and several variations for the different parallel modes were performed.

4. Variations of the different versions and experiments

4.1. SIMD Program

In SIMD mode, the program consists of a pure SIMD code. The network transfer operations in SIMD are executed synchronously in a straightforward fashion. That implies a faster execution time than in MIMD mode, in which the program must be blocked in a loop, polling the network buffer for incoming data. The second segment of the algorithm is divided into the calculation of the smoothed image and the clearing of the edges. An advantage of this part is the potential overlapping of instructions sent to the FIFO and those executed in the PEs while the control flow operations are executed in the MCs. Another advantage of SIMD mode is due to the actual memory boards implemented in the PASM system prototype. As can be seen from table 1, executing instructions from the fetch unit queue is faster than from the PE memory.

4.2. MIMD Program

This version is programmed in pure MIMD mode; the MC is only used to calculate the execution time of the program by waiting until the PEs finish execution. Since the network operations are carried out in MIMD mode, the data transmission is executed asynchronously. Hence, there is more overhead than in SIMD mode to check whether the network buffer is ready to accept data during send operations and when data is received from other PEs.

The calculation part of the program determines first, the inner pixels of the smoothed image. Then, all the edge pixels having eight neighbors are calculated and those pixels which do not have eight adjacent neighbors are cleared. Since each PE corresponds to a specific part of the total image it needs to be calculated and cleared on different sides. Thus, various MIMD sub-programs are required to execute the MIMD version.

One advantage of the MIMD version is due to the MC68000. The MC68000 has instructions with data dependent execution times (e.g. the divide instruction execution time differs up to 70% for 16-bit operations). In SIMD mode, all PEs have to wait on each divide instruction until the last PE has completed the instruction before fetching the next instruction. In MIMD mode, each PE works independently. The execution time of the program depends on the PE which finishes last.

4.3. Hybrid S/MIMD Program

In the Hybrid S/MIMD mode, data transfer and the calculation of the inner pixels of each subimage is written in SIMD mode. The third part is identically with the MIMD program. It calculates the edge pixels. This program should out-perform the other versions since the data-independent segment of the algorithm is written in SIMD mode and the data-dependent segment is written in MIMD mode.

4.4. Experiments

One goal of this research was to identify the most efficient mode of parallelism for *Image Smoothing* on PASM. The programs were written in MC68000 assembly language. Pure SIMD, pure MIMD and Hybrid S/MIMD implementations were tested for 4 PEs. The 16 PE versions have been implemented, but at the time of this writing had not been tested due to hardware difficulties. An SISD (serial) version was programmed in order to make comparisons with the parallel versions.

In detail, execution times were measured with respect to the image sizes 64^2, 256^2 and 1024^2. To illustrate the data dependent execution times, artificial input images with constant 00, alternating 00 and FF and constant FF data bytes were created, respectively.

5. Data Measurements

Figure 4 illustrates the execution times of the SIMD, MIMD, Hybrid S/MIMD and the SISD version. For comparison purposes the computation time of the SISD version is quarter scaled. Therefore, whenever any curve passes below the SISD curve, super-linear speedup with respect to the number of PEs is being exhibited. This discussed further in section 6. There are 2 intersection points in Figure 4; one between the SISD and the SIMD version shown on Figure 5a; the other between the SISD and the Hybrid S/MIMD version shown on Figure 5b. Figures 5a and 5b represent smaller windows of graph 1.

6. Discussion

As expected, the SIMD version outperformed the MIMD version. Table 2 points out that the SIMD version is 8-20% faster. First of all, this is based on the overlapping of the control flow instructions in the MC and the data processing instructions in the PEs in SIMD mode. In addition, there are

Table 2: Time of Computation in sec.

image size	image data	serial	parallel(4 PEs)		
			SIMD	MIMD	Hybrid
64^2	0000	0.1435	0.0718	0.0896	0.0687
	00FF	0.1399	0.0709	0.0887	0.0679
	FFFF	0.1364	0.0699	0.0878	0.0670
256^2	0000	2.3655	0.6126	0.7038	0.5751
	00FF	2.3052	0.5983	0.6888	0.5615
	FFFF	2.2465	0.5844	0.6742	0.5471
1024^2	0000	38.134	9.1379	9.8796	8.5877
	00FF	37.167	8.9103	9.6369	8.3697
	FFFF	32.216	8.6889	9.3987	8.1356

differences in fetch time as mentioned in section 2. The graphs illustrate that the execution times of the SIMD and the MIMD versions diverge for larger sizes of images. That means that the part which is responsible for the speed up of the SIMD version has a greater impact for larger loop iteration counts. Finally, the overhead to synchronize the network in MIMD mode should be mentioned as the major cause of the slowness of that version.

The *Hybrid S/MIMD version* was found to be the fastest version. It outperforms all parallel and the serial versions. Table 2 shows it to be 4-6% faster than the SIMD version. Since the Hybrid S/MIMD and the SIMD version contains nearly the same code, the difference is not as great as that between the SIMD and MIMD versions. This is based on the fact that the Hybrid version uses the calculation and the clear routine for the edges in MIMD mode. So there is no overhead as in the SIMD program which calculates at first the whole image, and then clears the proper edges after that. In other words, the Hybrid routine does not calculate the edges twice as it has to be done in the SIMD version. However, also in this case the graphs point out the diverging of the execution times.

The tests of the serial versus the parallel versions produce surprising results. The execution times of the Hybrid S/MIMD version disprove the supposition that the execution time of the parallel version would have a speed

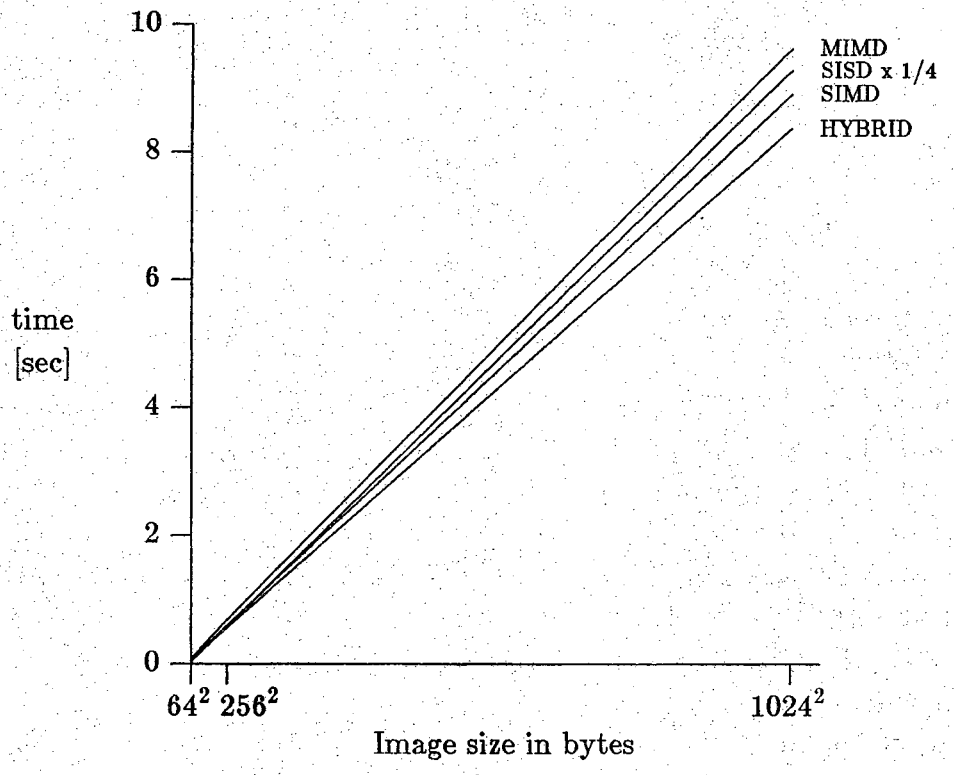


Figure 4: Speed comparison between Parallel Computation Modes

up of less than the number of active PEs. This assumption is based on the fact that the parallel versions need to perform network transfer operations; unlike the serial version [SiS87b]. The graph demonstrates obviously that in this case the Hybrid S/MIMD and the SIMD version is more than 4 times faster than the SISD version referring to the 1024^2 pixel image. This comes from the fact that the control flow instructions executed in the MC save more time than the transfer operations used. In addition, two intersection points can be seen. From that size of an image, up to larger sizes, begins the most efficient advantage of the parallel version over the serial version. In other words, for smaller image sizes the network transfer operations in the Hybrid S/MIMD and the SIMD version use more time than the overlapping of the control flow instructions can save, respectively.

7. Conclusion

This study of different parallel versions of the *Image Smoothing* algorithm has shown that the Hybrid S/MIMD version outperforms the SIMD and the

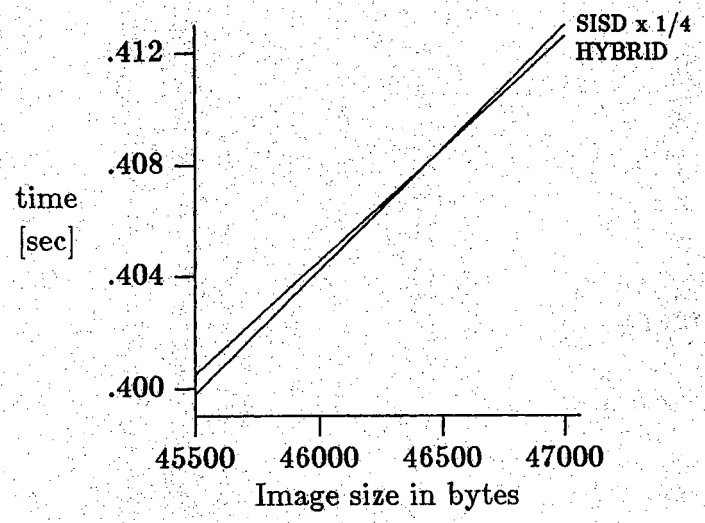


Figure 5a: Speed comparison between Hybrid and SISD version

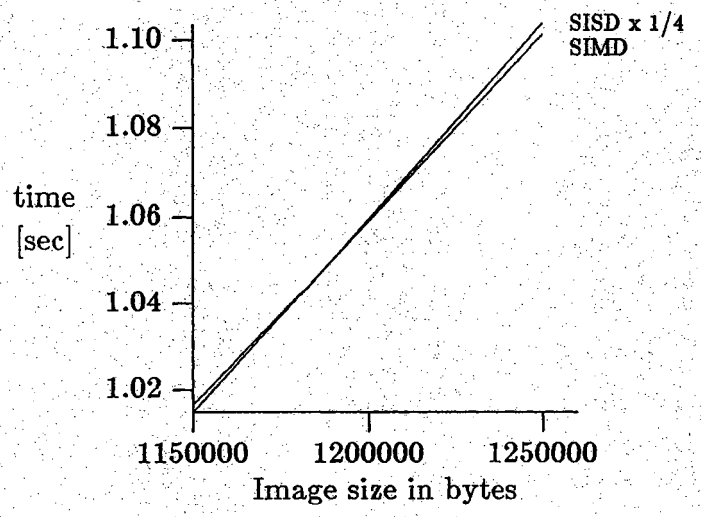


Figure 5b: Speed comparison between SIMD and SISD version

MIMD version clearly. Therefore, it serves as a useful paradigm for illustrating some of the potential advantages of a partitionable SIMD/MIMD parallel processing system such as PASM. Furthermore it turned out that the Hybrid S/MIMD and the SIMD version are more than 4 times faster than the SISD version when 4 PEs are used and the images are larger than 200^2 bytes.

While this fact seems counter-intuitive at first, in SIMD mode, a speedup of up to 2p (assuming p PEs) should be attainable since each PE has a logical MC to which control-flow activities may be off-loaded. The MIMD version represents the most inefficient algorithm. It is also less than 4 times faster than the SISD version. That results from the fact that the MIMD version has the same mode as the SISD version, but added network transfers.

Probably the most important implication of this work is related to the level at which programming was done — assembly language. Many of the performance differences observed rely on the fact that mode-switching, and network access times were on the order of a few instruction cycles. The important issue to be addressed is related to the problem of how to develop *efficient* HLL constructs which preserve these performance benefits while providing adequate expressive power to the programmer. It is the intention of this work to identify which semantics are most useful and to guide the development of efficient HLL and OS interfaces for them.

References

[Fly66] M. J. Flynn, "Very high-speed computing systems," *Proceedings of the IEEE,* Vol. 54, December 1966, pp. 1901-1909.

[SiS81] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller, Jr., H. E. Smalley, Jr., and S. D. Smith, "PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Transactions on Computers,* Vol. C-30, December 1981, pp. 934-947.

[SiS87a] H. J. Siegel, T. Schwederski, J. T. Kuehn, and N. J. Davis IV, "An overview of the PASM parallel processing system,"in *Computer Architecture,* D. D. Gajski, V. M. Milutinovic, H. J. Siegel, and B. P. Furht, eds., IEEE Computer Society Press, Washington, D.C., 1987, pp. 387-407.

[SiS87b] H. J. Siegel, T.Schwederski, D. G. Meyer, and W. Tsun-Yuk Hsu, "Large-scale parallel processing systems," *Microprocessors and Microsystems,* Vol. 11, No.1, Jan/Feb 1987.

[Sto80] H. S. Stone, "Parallel computers,"in *Introduction to Computer Architecture (second edition),* H. S. Stone, ed., Science Research Associates, Inc., Chicago, IL, 1980, pp. 363-425.

Threshold Generation for Image Processing on PASM

Kurt Schurecht

Abstract

Research is being done on how to best use the PASM parallel processing system the most efficiently with respect to many applications. Four image processing applications have been chosen, smoothing, thresholding, edge tracing, and block truncation coding. Different configurations of the PASM machine are being analyzed to obtain the optimal algorithm times with respect to the Sobel Threshold generation procedure. Different size images as well as different data sets have been examined. Results have shown that different configurations of the PASM system can cause vastly different running times.

1. Introduction

For many years theoretical work has been done relating to parallel machines and parallel machine algorithms, but most has not had concrete data to back it up. Two types of parallel machine taxonomies described by Flynn [Fly66] are SIMD (single instruction stream - multiple data stream) and MIMD (multiple instruction stream - multiple data stream). Work has recently begun that compares these two modes within the scope of many different applications. This paper describes an application from the image processing area using the PASM prototype system. The PASM system allows the user to switch quickly from SIMD to MIMD mode. A major result of this paper is to show the benefits and drawbacks of both SIMD and MIMD modes in the use of the PASM system and with respect to the chosen algorithm. This paper describes algorithms to find threshold levels for a given image, with emphasis on where SIMD mode or MIMD mode is more appropriate. Timing measurements have been made to compare the two modes, and with these times, some qualified conclusions about the two modes are made. The tests have been run using best, average, and worst case data sets, with three different image sizes. All programs have been written to run on four

processing elements controlled by one micro-controller.

Section 2 explains the PASM prototype and the background of the threshold problem. Section 3 details the problem in more depth, showing the algorithm used. The data taken and results of that data are presented in sections 4 through 6. Some conclusions and further work are detailed in sections 7 and 8.

2. Background

2.1. PASM Architecture

PASM is a partitionable SIMD/MIMD parallel processing system being designed to include over a thousand processors [SiS81]. It is a dynamically reconfigurable architecture, where the processors can be partitioned to form independent virtual SIMD and/or MIMD machines of various sizes. A 30-processor prototype has been completed [SiS87] and was used in the experiments described in this paper.

The prototype includes four micro-controllers that each control four processing elements. Each MC and PE has its own memory associated with it. Each MC also has a Fetch Unit associated with it to send SIMD commands to its PEs. The programs described in this paper use one MC, its Fetch Unit and the four PEs associated with it. When processing in SIMD mode, the MC sends a command to its associated Fetch Unit to send a set of instructions from the Fetch Unit's memory to the PEs under it. A mask register in the Fetch Unit determines which if any of the PEs associated with it will execute those instructions. The Fetch Unit places the instructions in a FIFO queue and each enabled PE takes the instruction off the head of the queue. When ALL PEs have finished that instruction, they all get the next one off the head of the queue. If the queue becomes empty, the PEs wait until another batch of instructions are placed in the queue. During the time while the PEs are executing instructions already in the queue, the MC can be doing other instructions. It may be taking care of looping overhead, or requesting the next set of instructions to be sent.

To switch from SIMD to MIMD mode a machine instruction is sent through the FIFO that executes a jump in the PEs into the MIMD memory area. Once this jump is executed, the PEs are no longer synchronized and may act alone. At this point the PEs start executing instructions that are stored at that location in their own memory. The PEs continue executing alone until they reach an instruction to return to SIMD memory. When this happens, each PE must wait until all others have also returned to SIMD space before they can get the next instruction off the FIFO queue.

The last relevant part of the architecture is the extra stage cube interconnection network. This network connects every PE to every other PE for use in sending data that needs to be shared between them. A routing tag is sent to set up the connection and then the information can be sent along the line. The receiving PE must be ready to receive data though. Finally a routine is run to drop the path. Once the path is dropped, the PE can connect to another PE.

2.2. Threshold Determination

Threshold levels are used to find the most defined edge in an image. In a picture, there are generally many different shades of the object, and also many small objects. The goal of thresholding is to find the most defined shape in the picture. The threshold level when found, is sent to a program that will trace the edge of the contour found by the threshold level. For example, if the image were a bird in the sky, the edge trace program would want to trace only the outline of the bird, not that of the clouds or just part of the bird. If the threshold level is chosen correctly, the bird is reduced to a block of black and the surroundings are reduced to a white background. With this image the edge detection program can trace the outline of the bird. The threshold generation algorithm used in this paper [TuA83] uses a Sobel edge operator and figure of merit, both defined later, to find the best threshold level.

The images used in this algorithm are stored as pixels of gray levels ranging from $00 to $FF. These gray levels are stored in consecutive locations of the PE memory. Each PE memory contains a subset of the entire image. All PEs then find a threshold gray level for their subset of the image.

3. Project Description

3.1. Problem

Threshold generation consists of choosing a gray level that will best find the edge of an image when used in conjunction with an edge tracing program. The threshold level is used by setting all pixel values equal to or greater than the threshold level to $FF and all values less to $00. If the threshold is chosen correctly, the crossovers between $00 and $FF will be the best edge to trace. To find the best edge, a Sobel operator is used on a window around each pixel value. A window is formed around each pixel, consisting of the pixel and its eight nearest neighbors. The following operation is performed on the window.

$$\Delta X = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * I \qquad \Delta Y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * I$$

$$\text{Gradient} = \sqrt{\Delta X^2 + \Delta Y^2}$$

ΔX and ΔY are found by multiplying the corresponding value in the matrix by the pixel in that window position, and then adding each of the nine results. The Sobel value is then found using ΔX and ΔY. The Sobel value is the gradient of the window. It is a measure of how well the window can act as an edge. Windows with greatly different gray levels have higher gradients than windows with very similar gray levels. The minimum pixel value and the maximum pixel value are found for each window also, to be used later. Next, each threshold level is checked to find the best value for edge detection. A figure of merit is found for each threshold level, with a high value meaning it is a good edge detector. The figure of merit is found by comparing the threshold level to each window figured above. If the threshold level is between the minimum and the maximum of the window then that window would contain an edge if the threshold level were used. In this case, the gradient found earlier for this window is added into this threshold's total. When all pixel windows have been checked, all gradient values are added together and averaged to find the figure of merit for that threshold. The threshold level chosen to send to the edge detection program is the one with the highest valued figure of merit.

On PASM, this same procedure is done with a few minor modifications. The data is assumed to be evenly distributed in each PE, with each PE containing a square subsection of the whole image. When the windows are formed for the pixels on a PE edge, the PE needs the pixels contained on the edge of the neighboring PE. When handling corners, likewise, the corner pixel in the diagonal PE is also needed. In total there are eight PEs that need to be connected to to get all necessary data for the computations, as shown if Figure 1. The threshold generation procedure on PASM involves four steps. The first involves transferring all needed outside edge pixels to PEs that will need them for their calculations. The second involves figuring the minimum, maximum, and gradient of each pixel window not on a PE edge. The third step is to take the edge pixel windows and find their minimum, maximum, and gradient. Finally, the figure of merit is determined for each threshold level and the highest threshold level is stored for use by the edge detection

program. The threshold level found in one PE has no effect on the threshold level found in any other PE. Each PE finds its own threshold level to send to the next program.

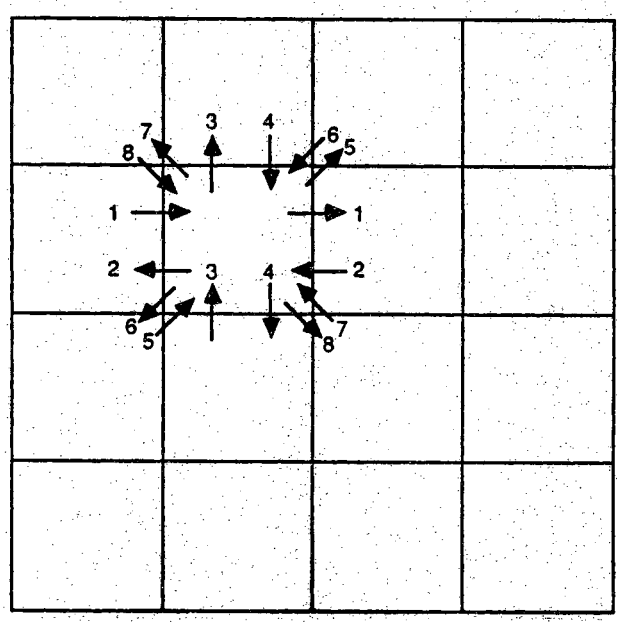


Figure 1

3.2. Algorithms

The following algorithm applies to SIMD and MIMD parallelism. The basic algorithm structure remains the same, but the actual code differs.

Threshold generation algorithm:

1. Connect to and transfer edges to each of 8 neighboring PEs
2. For each interior pixel
   Find gradient, min, max of window
3. For each edge pixel
   Find gradient, min, max of window
4. For each candidate threshold value
   For each pixel figured above
      If threshold > min and < max
         Add gradient to total
         Add 1 to matched
      FOM = total / matched
5. Threshold = Candidate Threshold with highest FOM

3.3. Programs

Completed to this point are programs that handle 4 PEs with 1 MC. There are three programs in ~/pasm/appl/histks that find threshold levels and save timing information. The programs are:

thr4s - a four pe program with steps 1,2 in SIMD mode
steps 3,4,5 in MIMD mode
only three transfers are conducted

thr4m - a four PE program with step 1 in SIMD mode
steps 2,3,4,5 in MIMD mode
only three transfers are conducted

thr4ma- a four PE program with step 1 in SIMD mode
steps 2,3,4,5 in MIMD mode
all eight transfers are conducted

Since each of these programs use 4 PEs, each data set is in a corner of the complete image, and only three transfers are actually necessary. This makes the 4 PE programs easier to write, but the code is very difficult to change for increasing to 8 and 16 PEs. Therefore code with both types of transfers has been included.

4. Experiments Performed

All experiments performed have been done using 4 PEs. Each of the three programs have been tested using data blocks in each PE of 16x16, 32x32, and 64x64. This implies complete images of size 32x32, 64x64, and 128x128 respectively. The data for the 32x32 case was also varied to get best case, average case, and worst case times. The data for all other sizes was only taken with average case data. Examples of each data set for a size of 8x8 follows.

Best Case	Average Case	Worst Case
00 00 00 00	00 01 02 03	00 FF 00 FF
00 00 00 00	04 05 06 07	00 FF 00 FF
00 00 00 00	08 09 0A 0B	00 FF 00 FF
00 00 00 00	0C 0D 0E 0F	00 FF 00 FF

5. Measured Data

Timings were taken that measured steps 1, 2, and 3 separately, and 4 and 5 together. All timings are shown in the table below. The size refers to the size of the data in each PE and the letters following refer to each of the sections and whether that portion of the program is written in SIMD or MIMD mode. The last letter corresponds to the type of transfers that were used in that program. "a" designates three transfers performed, "b" designates eight transfers. Best case is data set "A", average case is data set "B", and worst case is data set "C".

Timings for Threshold Programs					
Data	size/type	transfers	inside	edges	thresh
B	16ssmma	1.02	49	7.4	681
B	32ssmma	1.44	225	15.5	3460
B	64ssmma	2.29	965	31.7	16900
A	32ssmma	1.44	197	13.6	2980
C	32smmma	1.44	207	14.4	5090
B	16smmma	1.02	49	7.4	681
B	32smmma	1.44	227	15.5	3460
B	64smmma	2.29	971	31.7	16900
A	32smmma	1.44	203	13.7	2980
C	32smmma	1.44	211	14.4	5090
B	16smmmb	1.64	49	7.4	681
B	32smmmb	2.60	227	15.5	3460
B	64smmmb	4.57	971	31.7	16900

All times in milliseconds.

6. Interpretation of Data

6.1. Problem Related

The most obvious point from the data in the previous section is that the time for the last set of operations is very data dependent. The time varies from almost three to over five seconds. This data dependency will play an important part in the justifications in the next section. The data dependency also holds for the inside and edge times, but to a lesser extent. The problem related results of the data above give justifications as to why some portions of the program need to be in SIMD or MIMD mode.

6.2. Architecture Related

The main test of this program on PASM was to see if programs run fastest in SIMD, MIMD, or a hybrid of S/MIMD. Some of the routines lend themselves to using SIMD mode, as in step 1. Other steps lend themselves to MIMD mode, as in step 3. Steps 2, 4, and 5 though are not necessarily faster one way from just looking at the algorithm. There is one main advantage to each SIMD mode and MIMD mode. The data above has been taken to try and show the effects of each of these advantages. The advantage of SIMD mode comes from the fact that the code is straight inline code. The looping structure inherent in the program can be eliminated when using SIMD code, by having the MC do the looping and sending only one stream of inline code to the PEs. This takes away all of the overhead of looping out of the processor that has to do the raw number crunching, leaving it to concentrate on its task. There is a drawback of SIMD, which is the advantage of MIMD. When the SIMD program executes a variable length instruction, or sequence of instructions, each of the PEs must finish that instruction before any of them can go on to the next instruction. Because of this, all variable length instructions will take the time of the worst case time of all PEs. This can be very costly when an instruction, or group of instructions has a large time variation.

Looking at step 1, there are no variable length instructions, but there are transfers. Transfers are must faster to execute in SIMD mode. The subroutine calls to send data over the network can be replaced by one "move" instruction if the sending PE knows that the receiving PE is ready to receive. Implementing this section in SIMD mode, each PE knows exactly when the other PEs are ready and the transfer time can be minimized by replacing the subroutine calls. The main SIMD advantage also comes into play. Using SIMD mode, all the looping can be done in the MC. This function is obviously best written in SIMD mode.

Step 2 is not as easy to discern. The compares for the minimum and maximum are done as a small MIMD subroutine. For these purposes, this subroutine can be thought of as a variable length instruction. Two factors are weighting the timing. In SIMD mode, the looping overhead is eliminated, but if a new minimum is found in any PE, all PEs must wait until that new value is saved before they can go on to the next instruction. The two different programs that were timed show that elimination of the looping saves at most 6ms. The difference in best and worst case data is 24ms for the MIMD version. Therefore if the worst case is distributed properly, either all in one PE, or distributed over all PEs, the time for all PEs, even ones with best case data will be 24ms longer than necessary. This is a much greater time than the 6ms saved from the looping. A similar argument can be made for steps 4 and 5.

Step 3 has been implemented in MIMD mode, because the PEs that contain the outside edges of the image do not figure those edges. Therefore, a check is made to determine which PE is figuring its edge and then finds only the necessary edge window values. In the four PE case, each PE figures only two edges.

In general, it can be said that this program is a very good improvement over a comparable serial version. The only overhead incurred is the time for transfers and the time for finding edge values. A serial program would have those two times subtracted, but the inside and threshold times multiplied by N, with N being the number of PEs used in the parallel version. This time is much greater than the overhead for measured for transfers.

7. Future Work

Future possibilities in this area include expanding the number of PEs to eight and sixteen. The program thr4mb is a good starting place because it already makes the eight transfers, whether they are used or not. Sections 4 and 5 could also be tested comparing SIMD and MIMD modes, but the conclusions should be the same as section 2. Section 3 can be modified to work in SIMD mode by disabling PEs that do not take part in figuring a certain edge. Other possibilities are to examine the actual code for finding the figure of merit and best threshold level, to see if there is a way to optimize it and reduce the overall time of execution. A program also needs to be written that will take a file, split it up for placement in PEs and create downloadable files to place the data.

8. Conclusions

Examples of algorithms that show a definite speedup over their serial counterpart have been shown. The different parts of the chosen algorithm have given examples of where SIMD and where MIMD modes of operation work best. The SIMD / MIMD tradeoff has been discussed in detail, showing the attributes of each. Each of the programs written has been detailed and a future trail of work has been left for someone to pick up.

References

[Fly66] M. J. Flynn, "Very high-speed computing systems," *Proceedings of the IEEE,* Vol. 54, December 1966, pp. 1901-1909.

[SiS81] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller, Jr., H. E. Smalley, Jr., and S. D. Smith, "PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Transactions on Computers,* Vol. C-30, December 1981, pp. 934-947.

[SiS87] H. J. Siegel, T. Schwederski, J. T. Kuehn, and N. J. Davis IV, "An overview of the PASM parallel processing system," in *Computer Architecture*, D. D. Gajski, V. M. Milutinovic, H. J. Siegel, and B. P. Furht, eds., IEEE Computer Society Press, Washington, D.C., 1987, pp. 387-407.

[TuA83] D. L. Tuomenoksa, G. B. Adams III, H. J. Siegel, and O. R. Mitchell, "A parallel algorithm for contour extraction: advantages and architectural implications," *1983 IEEE Comp. Soc. Symp. Computer Vision and Pattern Recognition,* June 1983, pp. 336-344.

MIMD Contour Tracing for Image Processing

Brian Cox

Abstract

As part of a coordinated study of novel parallel architectures, contour tracing of images is being performed on the PASM system prototype. The group of applications was concerned with image smoothing, histogramming, image compaction and contour tracing. The contour tracing algorithm was designed as an MIMD program. The project was divided into two phases. In the first phase, a graphic image was divided into separate subimages and all local contours were traced and recorded. In phase two, the partial contours previously traced are connected. Several experiments have been run on the first phase of the algorithm with different data image sets. Analysis is focused on determining communication overheads, speedup over serial program versions, and overall efficiency. Continuing work is extending these programs to examine SIMD/MIMD trade-offs.

1. Introduction

In the area of parallel processing, there are several types of applications that benefit from parallelism. One of these applications is image processing. Many parallel image processing algorithms have been studied such as image coding [9], image correlation [7,12], image segmentation [14], two-dimensional FFT [10], histogramming [11], and line segment generation [13]. The algorithm of interest to this project is contour extraction. The parallel implementation of contour extraction can be beneficial from applications such as quality control inspection of printed circuit boards to military projects in which both algorithm speed and accuracy are crucial [8]. Contour extraction can be divided into two major algorithms: edge-guided thresholding and contour tracing. The edge-guided thresholding algorithm is used to determine a set of optimal thresholds which are used in the contour tracing algorithm to segment an image and trace the contours. This project is concerned with the parallel implementation of the contour tracing algorithm and its relation to the PASM architecture.

2. PASM

The PASM architecture is capable of dynamically reconfiguring to operate in SIMD and/or MIMD mode. The partitionable SIMD/MIMD machine consists of a control unit, an interconnection network, N processing elements , and Q micro controllers. The control unit is responsible for the overall coordination of the PASM system. The interconnection network is a circuit-switched Extra-Stage Cube network, which is a fault-tolerant variation of the multistage cube network. The processing elements (PEs) are sophisticated microprocessors that perform the actual SIMD and MIMD operations. The processing elements are controlled by the micro controllers (MCs). The PASM architecture when completed will consist of 1024 PEs and 32 MCs [11]. At this time, a PASM prototype has been completed which consists of 16 PEs and 4 MCs [2].

Several applications have been designed and implemented to test the performance of the PASM prototype. These applications range from matrix multiplications to AI algorithms [3,6]. The application group, of which this project is a part, is concerned with image processing. The image processing applications consist of four parts: image smoothing, histogramming, contour extraction and image compaction [4,5]. This paper is concerned with the application of contour extraction. The contour extraction application is benificial in testing of the PASM prototype [1].

3. Problem Structure

The implementation of contour extraction on a partitionable SIMD/MIMD machine such as PASM is advantageous in that it demonstrates several features of this type of machine. An SIMD/MIMD machine consists of a control unit, an interconnection network, and N processing elements. The PASM prototype has 16 processing elements(PEs), four of which are used in segmenting the graphic image in this project. When operating in SIMD mode, the control unit broadcasts instructions to all processors and each processor executes the instruction on data in its own memory. This mode of operation is well suited to edge-guided thresholding. The edge-guided thresholding algorithm uses a Sobel edge operator, meaning each pixel is processed identically. Therefore, SIMD parallelism would be the most efficient mode of operation for this algorithm. When operating in MIMD mode, each processor fetches instructions from its own memory and executes them on data in its own memory. Contour tracing, which is divided into two phases, is more suitable to this mode. The interconnection network is used in both algorithms. In the contour tracing algorithm (PHASE II) the interconnection network would be used to transfer partial contour information between PEs.

This project, however, is mainly concerned with PHASE I which requires only local data and no PE-to-PE communication.

4. Problem Solution

4.1. High-Level Description

As previously stated, the contour tracing project is divided into two phases. In PHASE I, the subimage within each PE is segmented and all local contours are traced and recorded. In PHASE II, the partial contours traced during PHASE I are connected. Initially, the graphic image is divided into subimages within each PE. For this project, four PEs were used with each subimage having a resolution of 8-by-8 pixels. Each PE also contains a list of threshold values for its individual subimage which were generated using edge-guided thresholding.

PHASE I begins by segmenting the image accordingly to each threshold value. The threshold values are considered seperately. Each pixel in the source image is compared to the current threshold value. If the pixel value is less than the threshold value, a zero is stored for that pixel. If the pixel value is greater than or equal to the threshold, the pixel value is set to one. Therefore, only valid contours are left for tracing. Tracing begins by scanning rows of the image from left to right, starting with the top row. The scanning will stop when a pixel with a value of one is found with a zero-valued pixel to either side. This pixel is the beginning of a new contour and is marked as the starting point. To determine the direction of this new contour, the surrounding pixels are scanned for a pixel of value one. The surrounding pixels are initially scanned counterclockwise as seen in Fig. 1. For easy reference to each pixel, a standard Cartesian coordinate system is used with the addition of each PE number (i). The i-x-y coordinates of this contour are stored and the next pixel is treated as the center point of the 3-by-3 window in Fig. 1. Counterclockwise scanning will begin again with this new point. The tracing will continue until a point of indecision is reached. Initially, if all the surrounding pixels of the startpoint are zero the pixel is not the start of a contour and is ignored. However, if the pixel is an edge pixel with an adjacent PE, the pixel could be the starting point of a contour which extends into the subimage of the adjacent PE. This pixel is marked as a startpoint and extension is verified in PHASE II. When a point of indecision is reached, data from the adjacent subimage will be needed to determine the direction of the contour. This point is marked as an endpoint and the PE(s) which contains the subimage with the possible extension is recorded. The tracing will then return to the startpoint and begin tracing in a clockwise direction. This

continues until another point of indecision is reached. When tracing clockwise, the i-x-y sequence of the pixels should be stored in front of the previous i-x-y contour sequence. This partial contour sequence is pointed to in the contour table located in each PE.

The contour table, which was not implemented in this project, should contain an entry for each individual contour. The contour table consists of the following fields:

1) a contour identification number
2) the threshold value which generated the contour
3) the number of pixels in the contour
4) a flag indicating if the contour is closed or partial
5) a pointer to the array containing the i-x-y sequence of the contour
6) a flag indicating whether the partial contour has been connected
7) the physical address of the PE which linked the contour
8) the physical PE address and identification number which is blocking extension of the contour
9) a locked/unlocked semaphore

A contour table should be constructed for each contour in each subimage for each threshold value.

3.	2.	1.
4.	.	0.
5.	6.	7.

Figure 1

An example 10-by-20 contour image is shown in Fig. 2. The images have been segmented in each PE with similar thresholds. PHASE I begins, simultaneously, in each PE by scanning each pixel. In PE 0 the startpoint is found to be (0,3,3). Counterclockwise tracing traces the contour to a point of indecision at (0,7,9). This point is recorded as an endpoint, the blocking PE is stored, and tracing returns to the startpoint for clockwise tracing. After the clockwise tracing has reached the point of indecision at (0,3,9), the first pixel

in the i-x-y sequence should then be (0,3,9). The scanning will then resume and find no more pixels since each previous pixel has been marked as already part of a contour. PHASE I will then repeat the process for a new threshold value. PHASE I is then complete and PHASE II begins. Since this project is mainly concerned with PHASE I, a brief description of PHASE II is given for clarity.

PHASE II will connect the partial contours traced in PHASE I. PHASE II can either begin after all the PEs in PHASE I have completed processing or after each individual PE has completed its processing. Since a complete contour may be contained in several PEs, a priority is established for which PE will link the contours. The priority is that each PE will only attempt to close a contour which is bordering a subimage to the left or above. To prevent several PEs from accessing a contour table at the same time a semaphore is introduced. The semaphore is a variable which locks a contour table while it is being modified to prevent other PEs from simultaneously modifing the same table. When a PE attempts to close a contour it must check the bordering pixels in the adjacent subimage. It compares the bordering pixels to the pixels in the

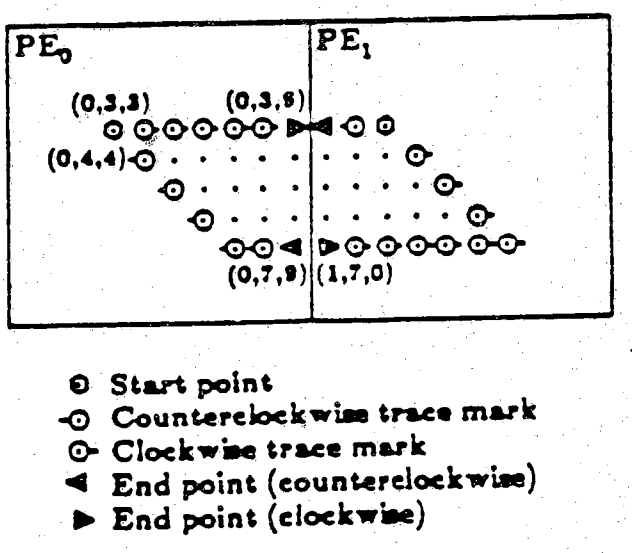


Figure 2

partial contour list for endpoints which match the border pixels. If the contour is found, the partial contour is transferred to the PE attempting closure. The i-x-y sequence is added to the contour to form an extended trace. If the contour is not found in the adjacent contour table, the PE attempting closure will probe into the adjacent subimage with the current threshold in an attempt to find a contour which may not have been detected with the adjacent PE thresholds. A limit is placed on the length of a contour to assure termination of the algorithm.

4.2. Algorithm

The actual MIMD algorithm was implemented in 68000 assembly code. Following is an outline of this algorithm:

I. Initialization
   A. Constants
   B. Data

II. Segmentation

III. Scanning Routine
   A. Initialize x-y coordinates
   B. Scan Image
      1. check for edgepixels
         a) valid startpoint?
      2. check for internal startpoint

IV. Startpoint Routine
   A. Save x-y coordinate of startpoint
   B. Check for edgepoint
   C. Trace counterclockwise
   D. Check for edgepoint
   E. Trace clockwise
   F. Reset for newcontour

V. Edgepoint Routine
   A. Check for corner blocking pixels
      1. Determine blocking PEs
      2. Store string for table
      3. Store pixel
   B. Check for normal edges
      1. Determine blocking PE
      2. Store string for table
      3. Store pixel

VI. Tracecounterclock Routine
   A. Compute x-y locations
   B. Valid startpoint?

VII. Traceclock Routine
   A. Compute x-y locations
   B. Valid startpoint?

VIII. Store Routine
   A. Modify traced pixels
   B. Reset image pointer

5. Program Variants

Two versions of the contour tracing algorithm were developed: a parallel version and a serial version. To effectively test the efficiency of parallelism, the two versions of the algorithm should be optimized. The parallel algorithm should be the best possible parallel version while the serial algorithm should be the best possible serial version. While these programs are far from the best possible implementations, they are, however, the best possible versions relative to each other, since the serial version is based upon the same programming techniques as the parallel version. Therefore, the speedup analysis due to parallelism should be accurate.

5.1. Experiments

The experiments were performed on 8-by-8 pixel images with four processing elements. This gives a complete graphic image of 16-by-16 pixels. Since the contour tracing algorithm is best suited for MIMD, the only mode of parallelism used was MIMD. The contour tracing algorithm is highly data

dependent. Therefore, several data sets were used and designed to test the performance of the parallel algorithm. The data sets range from a minimal set of elements to a maximal set. The data images consist of a zero element data set, a horizontal and vertical contour data set, a spiral image data set, a square image data set, and a random contour data set. The zero element data set has no contours, thus having the fastest execution time. The horizontal and vertical contour images represent the maximal number of individual contours that an image can hold. The execution times for these data sets should have the longest execution times due to the large number of contours and pixels that need to be traced. The spiral image has a maximal number of pixels in a single contour and no edge pixels. Thus, the execution delay is due mainly to the tracing routines within the algorithms. The square image is representative of a closed contour and the random data is used just to introduce some variability. The data sets were run five times on both the parallel version and serial version of the algorithm. The execution times were then averaged to eliminate any unnecessary delays such as that incurred by dynamic memory.

6. Measured Data

	Parallel Execution Times (m-sec)	Serial Execution Times (m-sec)	Speedup
no_data	2.900	11.880	4.10
horiz_data	5.696	20.404	3.58
vert_data	6.256	24.456	3.91
spiral_data	4.592	18.720	4.08
square_data	5.004	20.416	4.08
random_data	5.396	21.688	4.02

7. Interpretation of Data

7.1. Problem Related

From comparing the serial execution times to the parallel, it is obvious that there is a considerable speedup of the parallel algorithm. For all experiments, the speedup is close to O(n). The speedup is less than O(n) for a few data sets due to the extra analysis of the parallel algorithm. Since the graphic image is divided into four subimages, the parallel algorithm must check each PE for edges which block the extension of the contour. Within the

edgeblock subroutine, the algorithm must determine which PE(s) is blocking the contour and store the string. Therefore, the subroutine must check each coordinate to determine if it is an edge pixel or a corner pixel. The number of blocking PEs can range from zero to three. The edgeblock routine is the most time consuming.

In the serial version, the image is not subdivided and intensive edge detection is not required, therefore, eliminating the overhead. PHASE II of the parallel algorithm is also not required. The extension of the contours would increase the overhead and decrease the overall speedup. This is the main advantage to the serial approach. One of the disadvantages to the serial approach is the edge-guided thresholding required for this algorithm. The EGT performance decreases with the increase of image size. Thus, for reasons of speed and accuracy, the parallel approach is more favorable.

From the data, it can be seen that the majority of the speedup figures are greater than O(n). Since it is impossible to achieve a speedup greater than O(n), it can be assumed that the speedup figures are inaccurate. This is due to the fact that the parallel and serial implementations are not ideal. To achieve an accurate speedup time due to parallelism, the execution times of a best possible serial algorithm should be divided by the execution times of the best possible parallel algorithm.

7.2. Architecture Related

The PASM architecture is very well suited for the needs of the entire contour extraction algorithm. The SIMD capability of PASM allows the execution time of the Sobel operator to be a minimum. Both the EGT and contour tracing require PE-to-PE communication. Therefore, the performance of the interconnection network must be efficient. Since, PHASE I of the contour tracing requires only local data, there were no PE-to-PE communications to decrease execution times. Also, because of the implementation of the algorithm in MIMD, the Fetch Unit delays were a minimum. One minor architecture related delay affecting PHASE I was the dynamic memory refreshing. This had a very minor effect on the entire parallel contour tracing algorithm and was very similiar in the serial approach. This project did not fully utilize the PASM architecture. The complete contour extraction algorithm would be much more appropriate for testing PASM.

8. Future Work

To fully test the PASM architecture, the complete contour extraction algorithm should be implemented. PHASE II of this project would allow a more complete study on the interconnection network and a more accurate comparison of the serial and parallel approaches. This would require the combined execution of the EGT and the contour tracing. Once completed, more realistic results could be obtained by using actual image data sets. A complete study could be done by combining these algorithms with several image processing tasks such as image smoothing, two-dimensional FFT, and histogramming. Since the field of image processing is so vast, the problems for future study are limitless. The major task in relation to this project, however, should be the completion of PHASE II.

9. Conclusions

Overall, the contour tracing algorithm is very well suited for parallel implementation on PASM. The complete design utilizes the architecture very well. From the testing of PHASE I, it can be seen that the parallel programming approach has several advantages that include a considerable speedup of execution and an improved accuracy due to the Sobel operator. Thus, the parallel implementation of the contour extraction routines could be very useful for many applications.

References

[1] D. L. Tuomenoksa, G. B. Adams III, H. J. Siegel, O. R. Mitchell, "A Parallel Algorithm for Contour Extraction: Advantages and Architectural Implications," *Proceedings of IEEE Computer Society Conference of Computer Vision and Pattern Recognition* , June 1983.

[2] H. J. Siegel, T. Schwederski, J. T. Kuehn, N. J. Davis IV, "An Overview of the PASM Parallel Processing System," *Computer Architecture* , D. D. Gajski, V. M. Milutinovic, H. J. Siegel, B. P. Furht, eds., IEEE Computer Society Press, Washington, D.C., 1987, pp. 387-407.

[3] S. Fineberg, T. Casavant, T. Schwederski, H. J. Siegel, "Non-Deterministic Instruction Time Experiments on the PASM System Prototype," *1988 IEEE International Conference on Parallel Processing* , Chicago, August, 1988.

[4] W. Disch, T. Casavant,"Experiences with Parallel Image Smoothing on the PASM System Prototype," *ACM SIGPLAN Symposium on Parallel Programming: Experiences with Applications, Languages and Systems,* June 1988.

[5] K. Schurecht, "Threshold Generation on PASM," Final Report, EE 495, School of EE, Purdue University, Fall 1987.

[6] C. Ringer, "AI Graph Searching and Parallel N-Min-Finding on PASM," Final Report, EE 696, School of EE, Purdue University, Fall 1987.

[7] D. L. Ackerman, "Algorithm design for digital image correlation on a parallel processing system," in *High Speed Computer and Algorithm Organization* , edited by D. J. Kuck, D. H. Lawrie, and A. H. Sameh, Academic Press, Inc., New York, 1977, pp. 307-308.

[8] J. A. Cornell, "Parallel processing of ballistic missile defense radar data with PEPE," *COMPCON '72,* Sept. 1972, pp. 69-72.

[9] T. N. Mudge, E. J. Delp, L. J. Siegel, and H. J. Siegel, "Image coding using the multimicroprocessor system PASM," *1982 IEEE Comput. Soc. Conf. Patern Recognition Image Processing,* June 1982, pp. 200-205.

[10] P. T. Mueller, Jr., L. J. Siegel, and H. J. Siegel, "Parallel algorithms for the two-dimensional FFT," *5th Int'l Conf. Pattern Recognition,* Dec. 1980, pp. 497-502.

[11] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller, Jr., H. E. Smalley, Jr., and S. D. Smith, "PASM: a partionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Trans. Comput.,* vol. C-30, pp. 934-947, Dec. 1981.

[12] L. J. Siegel, H. J. Siegel, and A. E. Feather, "Parallel processing approaches to image correlation," *IEEE Trans. Comput.,* vol. C-31, pp. 208-218, Mar. 1982.

[13] C. D. Stamopoulous, "Parallel algorithms for joining two points by a straight line segment," *IEEE Trans. Comput.,* vol C-23, pp. 642-646, June 1974.

[14] R. J. Douglass, "A pipeline architecture for image segmentation," *15th Hawaii Int'l Conf. on System Sciences,* Jan. 1982, pp. 360-367.

Parallel Block Truncation Coding of Images

Chris A. Toomey

Abstract

This work is a practical look at the question of computational speedup, from the point of view of a specific algorithm (Block Truncation Coding) implemented on a specific parallel processor (the PASM prototype). By examining actual timing data from a real application program, much can be learned about the proper design of parallel computers and algorithms.

The paper discusses specifics of the PASM prototype (a partionable SIMD/MIMD non-shared memory machine) and Block Truncation Coding (an image compression algorithm). Theoretical and actual execution times were compared for different image sizes and numbers of processors used. The paper ends with general conclusions and ideas for future research.

1. Introduction

The fundamental purpose of parallel processing is increased computation speed. This paper is a practical look at the question of computational speedup, from the point of view of a specific algorithm (Block Truncation Coding) implemented on a specific parallel processor (the PASM prototype). By examining actual timing data from a real application program, much can be learned about the proper design of parallel computers and algorithms.

There two main reasons for using parallel processing to increase execution speed. One reason is to get maximum speed for minimum cost. This requires using efficient parallel algortihms, to maximum speed for the number of processors being used. The other reason for parallel processing is to increase maximum execution speed. The goal in this case is maximum speedup, even if the processors are not efficiently used. Both viewpoints will be examined in this paper.

The paper begins with a description of the PASM prototype (the parallel computer used) and Block Truncation Coding (the algorithm investigated). Next, implementation questions are discussed. Then the experiments performed and data gathered are presented. The paper ends with general conclusions derived from this project and suggestions for future research.

2. Overview of the PASM Prototype

The PASM prototype is a partionable SIMD/MIMD computer being used for parallel processing research at Purdue University. The PASM prototype (henceforth referred to as PASM) is collection of 30 microprocessors set-up for executing SIMD, MIMD, and hybrid SIMD/MIMD programs. These microprocessors are divided into 4 groups of 5 processors (called MC-groups) and assorted additional processors for system control and memory management. Each of the four MC-groups consists of one Micro Controller (MC) and four Processing Elements (PEs). The MCs and PEs use Motorola MC68000 microprocessors (running at 8MHz) as CPUs. Each processor has two dual-ported memory boards (1 Mbyte/board) so that program execution and data loading/storing can be overlapped. Since only MIMD programs were implemented in this project, the SIMD instruction broadcast hardware (Fetch Unit) can be ignored. Because the algorithms studied require no interprocessor communication, the programs can be run without modification on either MCs, PEs, or a combination of both.

For additional information on specifics of the PASM prototype, see references 1 and 2.

3. Overview of Block Truncation Coding

Block Truncation Coding (BTC) is an image compression algorithm that works by storing an approximation of the original image on a block by block basis. The original image consists of a height by width array of pixels, each pixel stored as one byte of grey level information. First the image to be compressed is split into 4 pixel by 4 pixel blocks. Then each block is compressed totally independently of all other blocks. The 16 bytes of grey level data is converted to 16 bits of pixel data (the bit-plane), an 8 bit mean value, and an 8 bit standard deviation value. The complete encoding alogrithm appears on the following page.

The BTC decoding algorithm works by computing two numberes to replace the one bit values used for each pixel (the bit-plane). These two values (which shall be called low and high) are computed in such a way that when the replacement is done, the mean value and standard deviation of the decoded block is the same as the original. The complete decoding algorithm appears below.

BTC Encoding Algorithm

For each 4×4 block do
- Compute the mean value of the 16 pixels in that block.
- Compute the mean square value of the 16 pixels in that block.
- For each pixel in the current block do
  - If pixel brightness $\geqq$ mean then pixel bit value = 1
  - If pixel brightness $<$ mean then pixel bit value = 0
- Compute standard deviation of the 16 pixels in that block.
- Store the mean, std, and bit-plane for that block.

BTC Decoding Algorithm

For each 4×4 block do
- Get the mean, std, and bit-plane for that block.
- Compute q = the number of bits in the bit-plane that are ones.
- Compute the low fill value: low = mean - $(\text{std}[q/(16-q)])^{.5}$
- Compute the high fill value: high = mean + $(\text{std}[(16-q)/q])^{.5}$
- For each pixel in the current block do
  - If pixel bit value = 1 then pixel value = high
  - If pixel bit value = 0 then pixel value = low
  - Store the pixel value.

To help clarify the encoding/decoding procedure, consider the following example (note: the sample used was selected to make the example clear and easy to understand, not because it is a typical image block).

Sample Block

10	20	30	40
20	30	40	50
30	40	50	60
40	50	60	70

mean = (10+20+30+40+20+30+40+50+30+40+50+60+40+50+60+70)/16 = 40
variance = (100+400+900+1600+400+900+1600+ ... +3600+4900)/16 = 1850
std = [variance - mean^2]$^{.5}$ = 16

The resulting bit-plane is:

0 0 0 1
0 0 1 1
0 1 1 1
1 1 1 1

The encoding algorithm would store mean, std, and the bit-plane (requiring a total of 32 bits).

To decode the block, first q would be computed.

q = 10

Then high and low would be computed.

low = 19
high = 52

The resulting decoded block is:

19 19 19 52
19 19 52 52
19 52 52 52
52 52 52 52

For additional information on the Block Truncation Coding algorithms, see references 3, 4, and 5.

4. Theoretical Execution Times for BTC

The terms speedup and efficiency were mentioned above as being two of the prime criteria for judging parallel algorithms. Speedup shows how much faster a parallel algorithm works than a serial algorithm for the same problem. It is computed by the following equation:

$$\text{Speedup} = \text{Serial Execution Time} / \text{Parallel Execution Time}$$

Obviously, the higher the speedup is, the faster the algorithm will run. In general, execution time decreases as more processors are used, increasing speedup. The increase in speedup due to using more processors depends on the efficiency of the parallel algorithm. Efficiency is computed as follows:

$$\text{Efficiency} = \text{Speedup} / \text{Number of Processors Used}$$

In general, efficiency is less than or equal to 1. An efficiency of 1 is called linear speedup, because an increase in number of processors used causes a linearly proportional increase in execution speed. The timing relationship for such algorithms is of the form:

$$\text{Execution Time} = F(n)/N$$

F(n) is some function of the input size n, and N is the number of processors used. Few algorithms exactly fit this form, but many approximate it. In the case of Block Truncation Coding, the theoretical timing relationships are:

$$\text{Encode Time} = k_1 + k_2 n/N$$
$$\text{Decode Time} = k_3 + k_4 n/N$$

N is the number of processors used, n is the input size (in pixels), and the k's are constants dependant or the actual implementation. The k_1 constant represents the initialization time required before any blocks are actually encoded. The k_2 constant represents the time required per pixel for one processor (not including the initialization time). The constants k_3 and k_4 work the same for the decoding algorithm. Obviously if k_1 and k_3 are very small, or the number of pixels per processor (n/N) is very large, the equations approximate linear speedup very closely. Therefore, there is a need to actually implement the algorithms to determine how close they come to the ideal of linear speedup, for different values of n/N.

5. Implementation

The BTC encoding and decoding algorithms were implemented in MC68000 assembly language. Assembly language was used because it provides maximum execution speed and because high-level parallel optimizing compilers are not available for PASM. The two algorithms were written in a single program, for simplicity of experimentation (they are totally separate routines, but putting them in one file simplifies downloading, etc.)

There were two possible ways to optimize the BTC programs for fast execution speed. The first would be to maximize speedup for the resulting program. Maximum speedup would require encoding/decoding only one block per processor. The resulting execution times would then be:

$$\text{Minimum Encode Time} = k_1 + k_2$$
$$\text{Minimum Decode Time} = k_3 + k_4$$

By minimizing the constants, the resulting programs would run tremendously fast (less than 100 μs each). Unfortunately, for images of realistic size, this would require far too many processors. Therefore the programs were optimized for the other case.

The other way to optimize the programs was in terms of high efficiency. This approach has each processor encoding/decoding several image blocks. As a result the k_2 and k_4 constants become most important, because they are multiplied by n/N, which for realistic images would be at least 16 or more. Therefore the primary goal was to optimize k_2 and k_4.

Three important methods of optimization were used. The first was to use assembly language to write the programs, instead of a higher level language. Even if a high-level optimizing compiler was available for PASM, it could not compete with the speeds possible with quality assembly code. Second, Some time was saved by expanding loops. Specifically, this meant repeating sections of code the 16 times needed to process each pixel in a block. This saved loop overhead time, which would have been significant. And third, extensive use of lookup tables was made. These lookup tables store pre-computed values for different calculations (such as the square root of an 8-bit number), saved significant amounts of execution time.

6. Experiments performed

Three different experiments were conducted with the BTC encoding/decoding program. The first experiment was to determine the effects of image variations (other than image size) on execution speed. The second experiment was to see the effect of image size on execution speed, given a constant number of processors. The third experiment tested execution speed for a constant image size using a variable number of processors.

In the first experiment, 20 randomly generated images were used to determine approximate best case, worst case, and average case execution times. These were then compared to produce an approximate percentage variation in execution speed due to image variations. The random images consisted of 4096 pixels (64×64) each, and were processed using four PEs.

In the second experiment, random images of three different sizes were tested to determine the effects of image size on processing rate. Images sizes of 1024 pixels (32×32), 4094 pixels (64×64), and 16384 pixels (128×128) were used. The images were processed using four PEs, with the four PE execution times being averaged to produce the actual timing data.

In the third experiment, a random image was processed using different numbers of PEs, in order to determine execution speedup and efficiency due to increasing parallelism. The same image was processed using 1, 2, and 4 PEs. The random image consisted of 4096 pixels (64×64).

7. Data

The tables below list the actual timing data collected for each of the three experiments. Image size is in pixels and execution times are in milliseconds. Except for experiment 1, all execution times are the average values for all images used and all PEs used.

Experiment 1

Data Type	Encode Time	Decode Time
Best case	20.008	9.452
Average case	20.0348	9.4864
Worst case	20.064	9.532

Experiment 2

Image Size	Encode Time	Decode Time
1024	5.058	2.436
4096	20.037	9.488
16384	79.864	37.564

Experiment 3

Processors	Encode	Speedup	Decode	Speedup
1	79.868	1.0000	37.576	1.0000
2	39.944	1.9995	18.810	1.9977
4	20.037	3.9860	9.488	3.9604

8. Interpretation of Data

Using the data gathered in experiment one, it is possible to compute an approximate percentage varation in execution speed due to image variations. The actual results are:

$$\text{Actual Encoding Time} = \text{Average Encoding Time } \pm .14\%$$
$$\text{Actual Decoding Time} = \text{Average Decoding Time } \pm .42\%$$

Clearly the effect of image differences (other than size) on execution time is practically insignificant.

In order to determine the affect of image size on execution speed, the results of experiment two were graphed as shown on page 11. The graph clearly shows that execution time increases linearly with image size, as expected. From the data it is possible to compute k_1 and k_3, the time taken when the image size is zero. The values are:

$$k_1 = 70.9\ \mu s$$
$$k_3 = 94.1\ \mu s$$

The constants k_2 and k_4 are the slopes of the encoding and decoding lines, respectively. Calculating the slopes results in:

$$k_2 = 19.48\ \mu s$$

$$k_4 = 9.148\ \mu s$$

In order to determine the affect of increasing parallelism on execution speed, the results of experiment three were graphed as shown on pages 12-15. The two speedup graphs show that both the encoding and decoding algorithms exhibit (approximately) linear speedup. The two efficiency graphs show that efficiency drops only slightly as parallelism increases.

Using the results of the three experiments, it is possible to produce equations for the actual execution times based on image size and number of processors. The resulting equations are:

$$\text{Encoding Time} = 70.9 + 19.48 * (n/N)\ \mu s \ \pm .14\%$$
$$\text{Decoding Time} = 94.1 + 9.148 * (n/N)\ \mu s \ \pm .42\%$$

Clearly these equations match the theroretical equations of section 4.

9. Future Research

There are several possible lines of research continuing where this project left off. The most obvious possibility would be to just continue directly. Additional optimization may be possible. More data could be gathered (time and machine problems prevented tests using larger images or more processors). A hybrid SIMD/MIMD version could be written (the current version is entirely MIMD). Note: It is not recommended that a pure SIMD version be written using the current condition code hardware. There is little chance that a pure SIMD version could run faster than an MIMD version on this machine (due to the nature of the problem and the nature of the computer being used). To write such a version may be an example of using the wrong hardware for the wrong problem.

Another research idea than beating this relatively dead horse, would be to convert the BTC programs to do real disk I/O to get data and store results. Because there is native operating system kernel, all data was downloaded with the programs, totally ignoring the usefulness of having two dual-ported memory boards with each processor. It would be very good to find out how well the I/O processors and dual-ported memory boards work for a real program using large amounts of data.

References

[1] T. Schwederski, W. G. Nation, H. J. Siegel, and D. G. Meyer, "The Implementation of the PASM Prototype Control Hierarchy"

[2] H. J. Siegel, T. Schwederski, J. T. Kuehn, and N. J. Davis IV, "An Overview of the PASM Parallel Processing System," in Computer Architecture, D. D. Gajski, V. M. Milutinovic, H. J. Siegel, and B. P. Furht, eds., IEEE Computer Society Press, Washington, D.C., 1987, pp. 387-407.

[3] L. J. Siegel, E. J. Delp, T. N. Mudge, and H. J. Siegel, "Block Truncation Coding on PASM," 19th Annual Allerton Conference on Communication, Control, and Computing, Allerton House, Monticello, IL, Sept. 1981.

[4] E. J. Delp and O. R. Mitchell, "Image Compression Using Block Truncation Coding," IEEE Trans. Commun., vol. COM-27, pp. 1335-1342, Sept. 1979.

[5] O. R. Mitchell and E. J. Delp, "Multilevel Graphics Representation Using Block Truncation Coding," Proceedings of the IEEE, vol. 68, no. 7, July 1980.

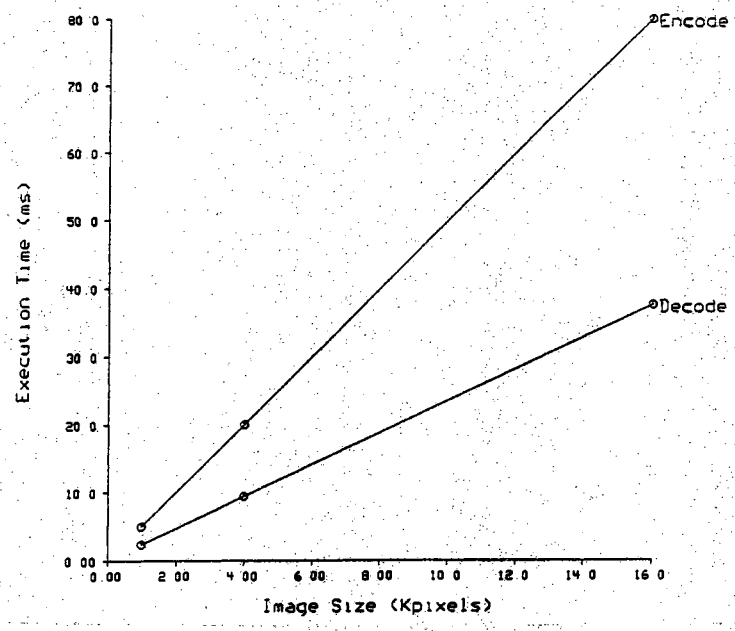


Figure 1.
Execution Time vs Image Size for 4 PEs.

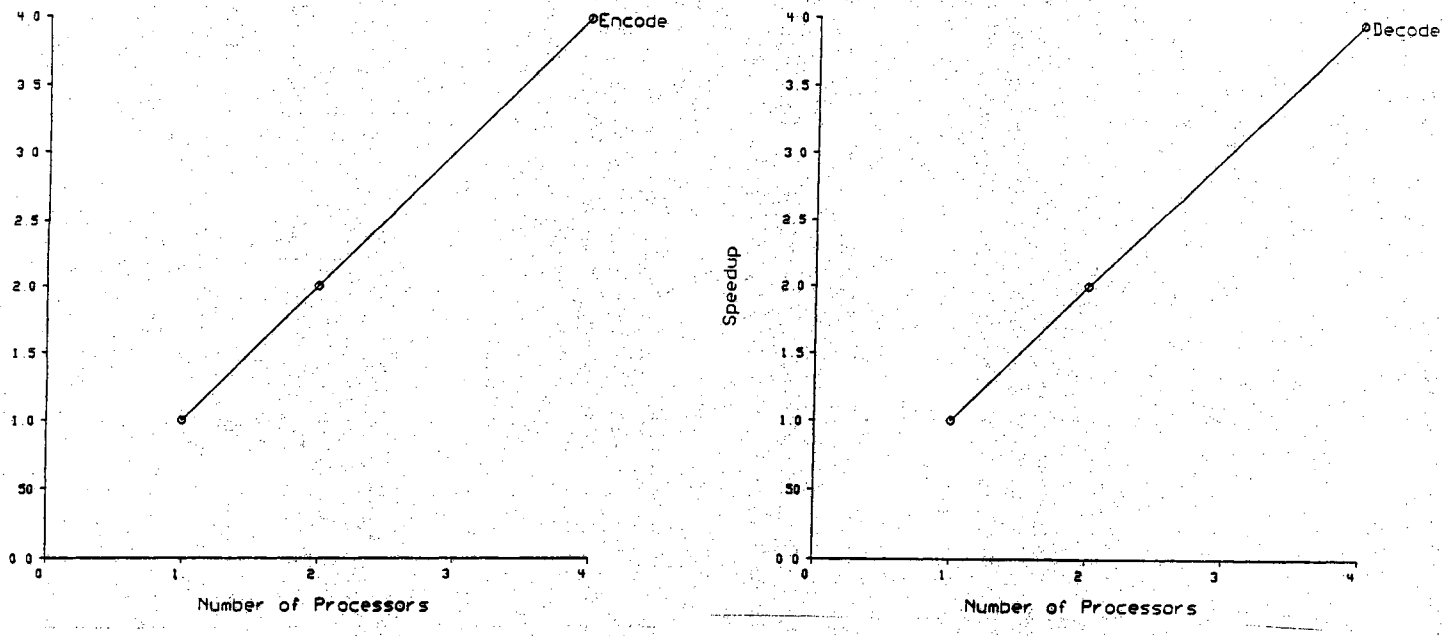


Figure 2.
Speed-up for Encode and Decode vs Number of PEs.

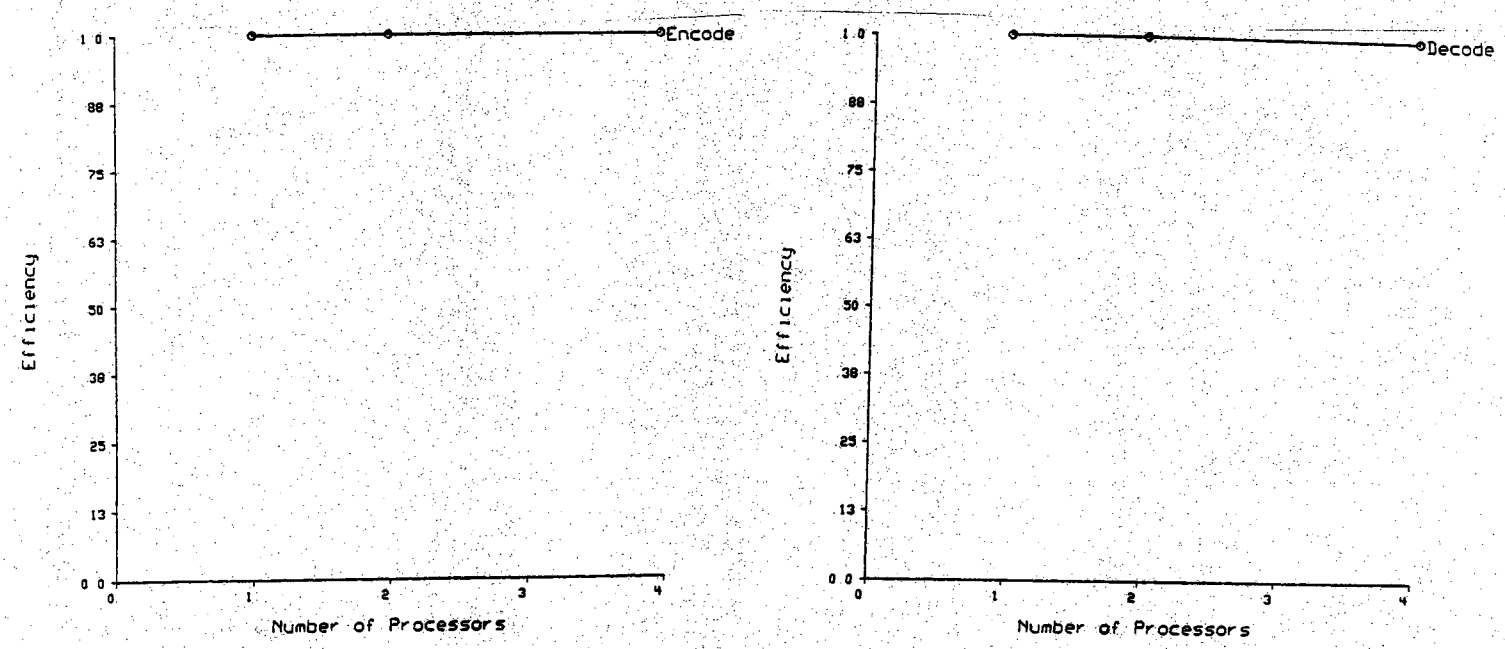


Figure 3.
Efficiency for Encode and Decode vs Number of PEs.

PART II

Mathematical Operators

Non-Deterministic Instruction Time Experiments

Samuel A. Fineberg, Thomas L. Casavant*, Thomas Schwederski, H.J. Siegel

Abstract

Experimentation aimed at determining the minimum-granularity at which variable-length SIMD operations may be decoupled into identical asynchronous MIMD streams for a performance benefit is reported. The experimentation is based on timing measurements made on the PASM system prototype at Purdue. The application used to measure and evaluate this phenomenon was matrix multiplication, which has feasible solutions in both SIMD and MIMD modes of computation, as well as in a hybrid of SIMD and MIMD modes. Matrix multiplication was coded in these three ways and experiments were performed which examine the tradeoffs among all of these modes.

1. Introduction

While extensive past efforts have dealt with analytical and simulated performance analysis of SIMD and MIMD algorithms, computations, and machines, this work describes empirically-based research generated from experiments on a parallel machine. This research was performed in an attempt to gain insight into the effect of certain aspects of novel architectures on applications programs. Specifically, the performance of the PASM prototype, a machine capable of both SIMD and MIMD modes of computation, is evaluated from the perspective of matrix multiplication. This application was chosen because it has obvious optimal solutions and a simple enough structure to permit analysis of architecture features through controlled measurements of program execution time. The experiments described are

* Supported by the Supercomputing Research Center under contract number 6925.

** Currently on leave from Purdue.

based on SIMD, MIMD, and hybrid S/MIMD algorithms for multiplying $n \times n$ matrices for values of n ranging from 4 to 256. Operations were performed on 16-bit integers utilizing 16 processors in several 4, 8, and 16 processor configurations.

The primary architecture feature being evaluated in this work is the ability to decouple small grains of variable execution-time operations from SIMD sections of code into multiple asynchronous MIMD threads of control. This unique feature derives from the ability to dynamically reconfigure the parallelism mode of PASM.

Results indicate that when mode-changing operations induce a minimal overhead, benefits of such decoupling may be found even for relatively small amounts of variation in the execution-time of individual operations. This same low-overhead mode-changing feature was also used to greatly improve the performance of the inter-process communication components of parallel programs by using the implicit hardware synchronization of SIMD mode to reduce the complexity of message passing protocols through the PASM interconnection network. Finally, experiments indicate that due to the existence of finite queues for issuing instructions from the control units to the processing elements in SIMD mode, superlinear speed-up [1] is achievable.

Section 2 briefly describes generally related work, and Section 3 overviews PASM and its prototype. Section 4 describes the basic algorithm that was used while Section 5 describes the programmed variations of this algorithm as implemented on PASM for use in the experiments presented in Section 6. In Sections 7 through 11, the empirical results are discussed under special consideration of the PASM architecture as well as the central issue of decoupling variable-length SIMD operations into multiple asynchronous MIMD streams.

2. Background and Related Work

Related experimental research has been carried out on several machines through the use of both simulation and experimental techniques. Simulation-based analysis was performed by Su and Thakore for the SM3 system and a hypercube architecture [SuT87]. Experimental work involving measurements on working machines has also been performed. Examples include work

[1] We define superlinear speed-up as the condition in which the speed-up to number of PEs (processing elements) ratio is greater than 1.

involving several machines: the BBN Butterfly [CrG85], Cm* [GeS87], the Encore Multimax [Hud88], the Intel Hypercube [Hud88], PASM [FiC87], and the Warp system [AnA87]. In these efforts, matrix multiplication was normally employed as an example algorithm. Other reported work involving efficiency measurements and algorithm optimization on parallel machines includes work done on an Alliant FX/8 [JaM86, Han88], a CRAY XMP [Cal84], and a combination of Apollo work-stations and an Alliant FX/8 [KuN88].

3. Overview of PASM and the PASM Prototype

The PASM (partitionable SIMD/MIMD) system is a dynamically reconfigurable architecture in which the processors may be partitioned to form independent virtual SIMD and/or MIMD machines of various sizes [SiS81]. A 30-processor prototype has been constructed and was used in the experiments described in Section 6. This section discusses the PASM architecture characteristics which are most relevant to the reported experimentation. For a more general description of the architecture, see [SiS87].

The *Parallel Computation Unit* of PASM contains N PEs where N is a power of 2 (numbered from 0 to N−1), and an interconnection network. Each *PE* (processing element) is a processor/memory pair. The *PE processors* are sophisticated microprocessors that perform the actual SIMD and MIMD operations. The *PE memory modules* are used by the processors for data storage in SIMD mode and both data and instruction storage in MIMD mode. The *Micro Controllers (MCs)* are a set of $Q=2^q$ processors, numbered from 0 to Q−1, which act as the control units for the PEs in SIMD mode and orchestrate the activities of the PEs in MIMD mode. Each MC controls N/Q PEs. PASM has been designed for N=1024 and Q=32 (N=16 and Q=4 in the prototype). A set of MCs and their associated PEs form a virtual machine. In SIMD mode, each MC fetches instructions and common data from its associated memory module, executes the control flow instructions (e.g., branches), and broadcasts the data processing instructions to its PEs. In MIMD mode, each MC gets instructions and common data for coordinating its PEs from its memory.

The PASM prototype system was built for N=16 and Q=4. This system employs Motorola MC68000 processors as PE and MC CPUs, with a clock speed of 8 MHz. The interconnection network is a circuit-switched Extra-Stage Cube network, which is a fault-tolerant variation of the multistage cube network. Because knowledge about the MC and the way in which SIMD

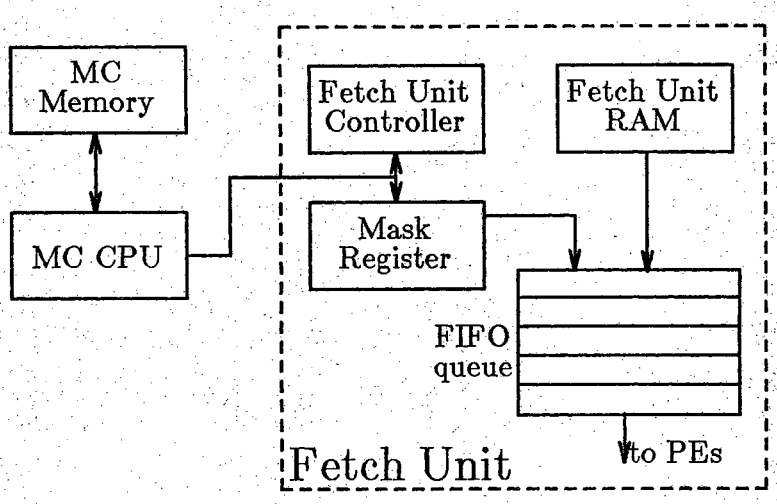


Figure 1: Simplified MC structure.

instructions are implemented with standard MC68000 microprocessors is essential to the understanding of the behavior that was observed in the experiments, the SIMD instruction broadcast mechanism is overviewed below.

Consider the simplified MC structure shown in Figure 1. The MC contains a memory module from which the MC CPU reads instructions and data. Whenever the MC needs to broadcast SIMD instructions to its associated PEs, it first sets the Mask Register in the Fetch Unit, thereby determining which PEs will participate in the following instructions. It then writes a control word to the Fetch Unit Controller which specifies the location and size of a block of SIMD instructions in the Fetch Unit RAM. The Fetch Unit Controller automatically moves this block word by word into the Fetch Unit Queue. Whenever an instruction word is enqueued, the current value of the Mask Register is enqueued as well. Because the Fetch Unit enqueues blocks of SIMD instructions automatically, the MC CPU can proceed with other operations without waiting for all instructions to be enqueued.

PEs execute SIMD instructions by performing an instruction fetch from a reserved memory area called the *SIMD instruction space.* Whenever logic in the PEs detects an access to this area, a request for an SIMD instruction is sent to the Fetch Unit. Only after all PEs that are enabled for the current instruction have issued a request is the instruction released by the Fetch Unit queue, and the enabled PEs receive and execute the instruction. Disabled PEs do not participate in the instruction and wait until an instruction is broadcast

for which they are enabled. This way, switching from MIMD to SIMD mode is reduced to executing a jump instruction to the reserved memory space, and a switch from SIMD to MIMD mode is performed by sending a jump to the appropriate PE MIMD instruction address located in the PE main memory space.

The SIMD instruction broadcast mechanism can also be utilized for *barrier synchronization* [LuB80] of MIMD programs. Assume a program uses a single MC group, and requires the PEs to synchronize R times. First, the MC enables all its PEs by writing an appropriate mask to the Fetch Unit Mask Register. Then it instructs the Fetch Unit Controller to enqueue R arbitrary data words, and starts its PEs which begin to execute their MIMD program. If the PEs need to synchronize (e.g., before a network transfer), they issue a read instruction to access a location in the SIMD instruction space. Because the hardware in the PEs treats SIMD instruction fetches and data reads the same way, the PEs will be allowed to proceed only after all PEs have read from SIMD space. Thus, the PEs are synchronized. The R synchronizations require R data fetches from the SIMD space. Thus, the Fetch Unit Queue is empty when the MIMD program completes, and subsequent SIMD programs are not affected by this use of the SIMD instruction broadcast mechanism.

In order to make comparisons of the speed of the PASM prototype relative to other machines and to compare the relative speeds of SIMD and MIMD instruction fetches, the actual raw performance of PASM in SIMD and MIMD mode was measured on the prototype and is illustrated in Table 1 in MIPS (millions of integer instructions per second) for two different types of instructions. The difference in speed between SIMD and MIMD modes can be attributed to two factors. SIMD instructions are fetched from the Fetch Unit Queue in the MC, and the queue can deliver data with one less wait state than can the PEs' main memories. In addition, PEs' main memories are implemented with dynamic memories. While care was taken in the hardware design that all refresh operations occur simultaneously in all PEs, and are performed invisible to the PE CPU, some delay is still possible. No such delay occurs during SIMD instruction fetches because the Fetch Unit queue is implemented with static RAM components. Measurements were made with repeated blocks of straight line code which were large enough to make the loop control overlap insignificant.

Table 1: Prototype raw performance.

Mode	Operation	Processing Rate
SIMD	16-bit Reg.-to-Reg. add	22 MIPS
MIMD	16-bit Reg.-to-Reg. add	18 MIPS
SIMD	16-bit Reg.-to-Mem. add	6.4 MIPS
MIMD	16-bit Reg.-to-Mem. add	6.0 MIPS

4. Matrix Multiplication Algorithms Used

The parallel matrix multiplication algorithm used here had $O(n^3/p)$ time and space complexity for multiplying two $n \times n$ matrices employing p PEs. Figure 2 shows an $O(n^3)$ time and space complexity serial algorithm. This particular algorithm is provided to illustrate the ordering of multiplications as they are done in the parallel version of Figure 3. Figure 4 demonstrates the progress of the serial algorithm for n=4. The two data-flow graphs illustrate what occurs during the first two iterations of the second *j* loop of Figure 3. The *i* loop of the serial algorithm simulates the PE number in the parallel algorithm. The calculation of ((i+j) mod n) in the serial version allows the rows of the B matrix to be stepped through as the *j* loop proceeds with the initial B matrix row number being *i*. The serial algorithm used in the measurements on PASM, however, was optimized in order to permit accurate evaluation of speed-up, and therefore did not perform multiplies in this columnar manner. Rather, it followed a more straightforward row-column order.

In the parallel algorithm, the outer *for all* loop represents iteration across space rather than time. Each PE contains n/p adjacent columns of each matrix as shown in Figure 5. Within each PE these columns are numbered from 0 to (n/p)−1 as shown in the algorithm of Figure 3. This layout is similar to that used by Su and Thakore in their experiments for the SM3 system [SuT87]. Using the *for v* loop, each of these adjacent columns is stepped through by each PE in sequence, and each PE appears as if it has n/p

* This effectively rotates all internal columns of the A matrix to the left without destroying the data in column 0 of the PE, or actually moving the data.

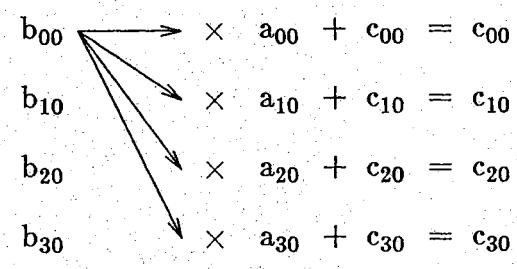
```
for i=0 to n−1 do
   for j=0 to n−1 do
      c_{i,j}=0;
for i=0 to n−1 do
   for j=0 to n−1 do
      for k=0 to n−1 do
         c_{k,i} = c_{k,i} + a_{k,((i+j) mod n)} b_{((i+j) mod n),i};
```

Figure 2: Serial matrix multiplication algorithm.

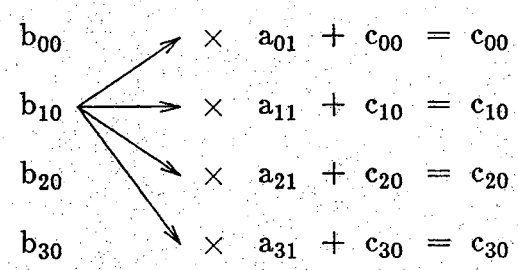
```
for all i, 0≦i≦n−1, do
   for v=0 to (n/p)−1 do
      for j=0 to n−1 do
         c_{j,v} = 0;
   for j=0 to n−1 do
      for v=0 to (n/p)−1 do
         for k=0 to n−1 do begin
            c_{k,v} = c_{k,v} + a_{k,v} b_{((i(n/p)+v+j) mod n),v};
      for v=1 to (n/p)−1 do
         change the pointer to column v−1 of the A matrix to point to
         column v;*
      for k=0 to n−1 do
         send a_{k,0} to PE (i−1) mod p;
         receive a value and move it into a_{k,((n/p)−1)};
```

Figure 3: Parallel matrix multiplication algorithm.

virtual PEs within it. The virtual PE number is then defined as (n/p)i+v. Thus, the row subscript of B is calculated by replacing *i* in Figure 3 with this virtual PE number. Data movement internal to each PE involves only a pointer adjustment. Only on the boundaries (i.e. the highest and lowest numbered columns of each PE) is the inter-PE network employed.



(a)



(b)

Figure 4: Two iterations of the serial algorithm for n=4.
(a) i=0, j=0, $0 \leqq k \leqq 3$. (b) i=0, j=1, $0 \leqq k \leqq 3$.

PE 0 ••• PE 3

a_{00}	a_{01}	b_{00}	b_{01}	c_{00}	c_{01}		a_{06}	a_{07}	b_{06}	b_{07}	c_{06}	c_{07}
a_{10}	a_{11}	b_{10}	b_{11}	c_{10}	c_{11}		a_{16}	a_{17}	b_{16}	b_{17}	c_{16}	c_{17}
a_{20}	a_{21}	b_{20}	b_{21}	c_{20}	c_{21}		a_{26}	a_{27}	b_{26}	b_{27}	c_{26}	c_{27}
a_{30}	a_{31}	b_{30}	b_{31}	c_{30}	c_{31}	•••	a_{36}	a_{37}	b_{36}	b_{37}	c_{36}	c_{37}
a_{40}	a_{41}	b_{40}	b_{41}	c_{40}	c_{41}		a_{46}	a_{47}	b_{46}	b_{47}	c_{46}	c_{47}
a_{50}	a_{51}	b_{50}	b_{51}	c_{50}	c_{51}		a_{56}	a_{57}	b_{56}	b_{57}	c_{56}	c_{57}
a_{60}	a_{61}	b_{60}	b_{61}	c_{60}	c_{61}		a_{66}	a_{67}	b_{66}	b_{67}	c_{66}	c_{67}
a_{70}	a_{71}	b_{70}	b_{71}	c_{70}	c_{71}		a_{76}	a_{77}	b_{76}	b_{77}	c_{76}	c_{77}

Figure 5: Data Layout for n=8, p=4.

This particular algorithm was chosen over a more standard parallel matrix multiplication algorithm (e.g., see Stone [Sto80]) for several reasons. First, if a broadcast approach is used to distribute the "a" coefficients to the

PEs, n network set-up cycles are incurred in addition to n network transfer cycles. In the chosen algorithm, the network remains in one configuration (i.e., PE i connected to PE (i-1) mod p), thus eliminating any recurring network set-up costs, while not incurring any additional network transfer costs. Also, this algorithm facilitates a columnar data format which was preferable for several reasons. First, because all matrices are stored in columnar format, B×A may be calculated as well as A×B without rearrangement of the data. Second, each matrix may be used in subsequent multiplications without reformatting. Data uniformity is also desirable to facilitate parallel I/O transfers of large data sets from secondary memory.

What follows is a semantic description of the progress of the algorithm. During each of the n^2/p iterations of the innermost loop of the algorithm shown in Figure 3, each of the elements of the columns of the A matrix is multiplied by an element of the B matrix. Note that due to the columnar storage, the column of the B matrix matches the internal column number of the A matrix. However, the absolute row of B must match as the absolute column number of the A matrix (i.e. the column number when j=0 and k=0). This value is then added to an element of the C matrix. Therefore, there is a total of n multiplications and additions per inner loop with this second loop being executed n/p times. In the second innermost loop, the columns of the A matrix are shifted one column to the left. Within each PE, this transfer involves a single memory move, because a pointer to the entire column is changed rather than moving its elements. However, for the lowest numbered column of each PE, the transfer employs the interconnection network. This column is transferred through the network and stored in the highest numbered column of PE ((i−1) mod p). The data received through the network is placed in the PEs memory as its highest numbered column. This transfer requires n network operations (one for each element of the column). This procedure is repeated until all of the columns of the A matrix have been through each of the (n/p) positions of each PE for a total of n^2 network transfer operation times incurred. During each of these elemental time periods, p values are exchanged.

Consider the time required for index calculation. The constant i×(n/p) was pre-calculated and placed in the programs data segment since it was constant in each PE for a given value of n and p. Also, the j+k operation involved in the B matrix row calculation was done outside the k loop and therefore only contributes O(n) time complexity per PE. The calculation of the A and C matrix row indices was done with the MC68000's auto-increment

mode. Due to the pipelined structure of the MC68000 this does not add any extra execution time of the non-autoincrement mode. Therefore, the index calculation, as a separate component of the execution, time is not significant.

The current implementation of the network in PASM supports 8-bit data transfers. Because these experiments involved 16-bit data, each element transfer required two shift operations (one for transmitting and one for receiving), an OR operation, and two network operations. Because no DMA block transfers were possible given the current implementation of PASM, each column transfer required n single-element transfers for a total of 2n network operations per column.

Being circuit switched, setting up a path in the PASM prototype network is a time consuming operation. However, in this algorithm only a single path set-up is required, (i.e. PE i always sends to PE (i−1) mod p). Thus the measurements made do not reflect any significant influence from network reconfiguration overhead. Hence, there were $2n^2$ network accesses, n^3/p multiplications, and n^3/p additions required. This resulted in a $O(n^3/p)$ growth in execution time.

5. Implementations of the Algorithm

Three variations of the parallel algorithm, as well as an efficient serial version, were programmed in MC68000 assembly language for execution on the PASM prototype. The parallel versions included a pure SIMD, a pure MIMD, and a hybrid S/MIMD version. These three programs may be executed on 4, 8, or 16 processors simply by changing variables embedded in their data sections.

5.1. SIMD

The SIMD version executes all looping and control flow instructions in the MCs. Arithmetic, data movement, and index calculation instructions are executed on the PEs in SIMD mode. The PE instruction stream is obtained through the MC's Fetch Unit Queue and is executed synchronously on all PEs.

In PASM, the network appears to the PEs as two memory locations (transmit and receive registers). Network transfers are made directly to the transfer registers using memory-to-memory move instructions.

For several reasons, the SIMD version appeared to be the most natural choice for implementation. First, in the matrix multiplication algorithm used all PEs are always enabled, thus eliminating the need for enabling and

disabling the PEs. Second, the implicit synchronization inherent in SIMD mode allowed the network transfer operations to be carried out in a straightforward fashion requiring only two memory-to-memory move instructions. Third, the only data-dependent portion of the algorithm is the actual multiplication instruction, which has a variable execution length due to its microcoded implementation in the MC68000. A final advantage of the SIMD version is due to the use of a FIFO queue in the Fetch Unit of the MCs. Because this queue buffers instructions being sent to the PEs, the execution of SIMD instructions by the PEs can be overlapped with the execution of control flow instructions by the MCs.

In addition to these conceptual factors involved in the SIMD version, there are some factors that were present due to the implementation of the PASM prototype. First, instructions may be accessed more quickly from the Fetch Unit Queue than from the PEs main memory. This is due to the use of faster memory technology in the queue. Also, the overlap of the control flow instructions with PE instructions is only present if the queue remains non-empty. In other words, the PEs can only proceed if the MCs supply instructions faster than the PEs can remove them from the queue.

5.2. MIMD

The second version was a pure MIMD program in which the MCs were only used for initiating the PE programs. The PEs executed all instructions asynchronously including all network, control flow, and arithmetic operations. Although the network hardware prevents overwriting of old data in the transfer register, the asynchronous network operations necessitated polling of the network buffer in order to determine whether it was ready to accept new data. After transmission, the network buffer must be polled to assure that the data is valid before a receive operation can be completed.

The major advantage of the MIMD version was rooted in the variation of the execution time of the MC68000 multiply instruction. Multiply or divide instructions require an amount of time which is related to the number of 1's in the binary representation of one operand. Assume an algorithm is executed on K PEs, each PE executes J instructions, and instruction j on PE k takes time τ_{jk}. Then the total execution time in SIMD mode (τ_{SIMD}) is the sum of the worst case times for each instruction as given by:

$$\tau_{SIMD} = \sum_{j=1}^{J} \max_{k=0}^{K} \tau_{jk}$$

In MIMD mode each PE proceeds independently, and therefore the execution time (τ_{MIMD}) is the worst case sum of instruction execution times as given by:

$$\tau_{MIMD} = \max_{k=0}^{K} \sum_{j=1}^{J} \tau_{jk}$$

In general, $\tau_{MIMD} \leqq \tau_{SIMD}$.

5.3. S/MIMD

The hybrid S/MIMD algorithm was developed to take advantage of the fast barrier synchronization mechanism described in Section 3 and to exploit the execution time advantage of the MIMD program (i.e. decoupling at low cost). In this version, the main program was the same as in the MIMD case. The difference was in the method of determining whether the network was ready to accept a transfer operation. Rather than polling the network buffer, barrier synchronization was used to allow network operations to be carried out as simple memory-to-memory move operations as in the SIMD version. This lowered the amount of network overhead to a level comparable but slightly greater than the SIMD version due to the mode switching time. The other advantages of SIMD mode (i.e., faster instruction fetch and control flow instruction overlap) could not be realized in this version.

6. Experiments Performed

Experiments were performed on n×n matrices and measurements were made of the execution times for n= 4, 8, 16, 64, 128, and 256. The algorithm was implemented for SIMD, MIMD, and S/MIMD mode and was run on p = 4, 8 and 16 PEs. All operations were 16-bit unsigned integer operations and overflow was ignored. To allow for varying machine and problem size, loops were utilized wherever possible.

To measure the amount of asynchronous execution necessary to yield better performance by the hybrid version over the SIMD version, the number of multiplies in each innermost loop of the algorithm was made to be a dependent variable. These multiplies were added as straight line code in order to prevent skewing of execution time data due to control flow overlap. The multiplies were added to study the effect on the total execution time and did not affect the values in the C matrix. Let T_{SIMD} and $T_{S/MIMD}$ be the total

execution time for the SIMD and S/MIMD programs respectively. The performance of each of the components of the execution time was measured at points corresponding to quantities of inner loop multiplications where:

$T_{SIMD} < T_{S/MIMD}$,
$T_{SIMD} = T_{S/MIMD}$, and
$T_{SIMD} > T_{S/MIMD}$.

Measurements were made with the internal system timers (MC68230). Experiments were performed for each version with the identity matrix in A and random data in B. While the value of the multiplier used in the MC68000 multiplication instruction affects the execution time, the data value of the multiplicand has no effect. Therefore, the elements of the A matrix, which were always used as the multiplicand could be chosen as the identity matrix without affecting program performance. By using the identity matrix, matrix multiplication results could be easily verified, thereby simplifying the debugging process. Random data, produced from a uniformly distributed random number generator, was chosen for these experiments in order to represent the average case, and the same data sets were used on all versions of the algorithm with the same value of n and p.

7. Speed-up & Overall Comparison

Figure 6 illustrates execution time of matrix multiplication vs. problem size observed in the parallel versions of the algorithm for p=8. The difference between the SISD time and that of the parallel versions represents an improvement by a factor of approximately p.

Although not readily apparent in the graph, it should be noted that $T_{MIMD}/T_{S/MIMD}$ decreases as n increases. The only difference between these two versions is attributed to the contribution to the execution time of communication. Note that for p fixed, and small n (e.g. n=8), the time complexity of the multiplications is $\frac{n^3}{p}$ or $\frac{n^2\,(8)}{8} = n^2$. This is the same order of contribution as communication. Hence, for small n, the $O(n^2)$ communication contribution dominates the $O(n^3)$ arithmetic. However, for larger n, the $O(n^3)$ component ultimately dominates and all three curves converge.

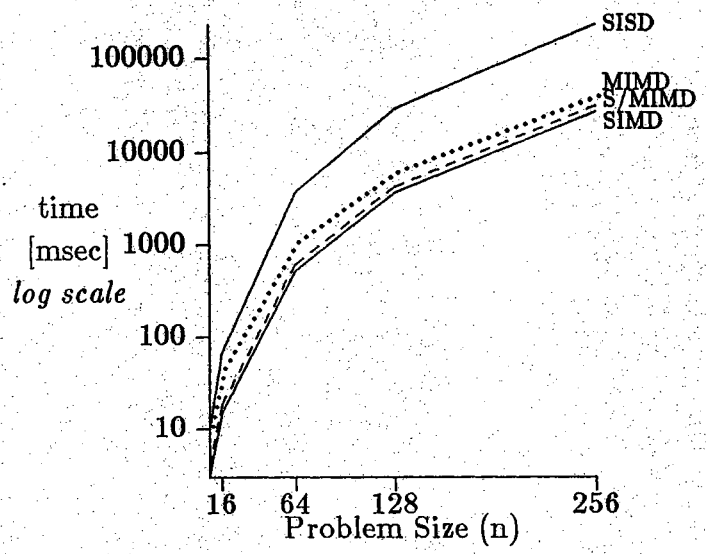


Figure 6: Execution time vs. problem size for p=8
and one multiply per inner loop.

The third aspect of this graph is the apparent advantage of the SIMD version over the S/MIMD version. The difference is caused by the ability of the MCs to execute control flow in parallel with arithmetic. However, the S/MIMD version has the potential for better performance due to the decoupling effect associated with MIMD execution of data-dependent execution time operations. In order to determine the point where these graphs cross, however, experiments were conducted which added more data-dependent instructions in a controlled way.

8. Execution Time vs. Number of Variable Length Operations

To determine the amount of asynchronous execution needed to achieve a benefit when executing a portion of a computation asynchronously in MIMD mode, additional multiplication operations were added to the innermost loop of the algorithm. Figure 7 plots total execution time for SIMD and S/MIMD programs with added multiplications vs. the number of added multiply instructions for n=64 and p=4 with random data. The lines plotted include 3 different points with the number of multiplications ranging from 13 to 15. These lines are disjoint at the endpoints with the SIMD version being faster for small numbers of added multiplies and S/MIMD being faster as the

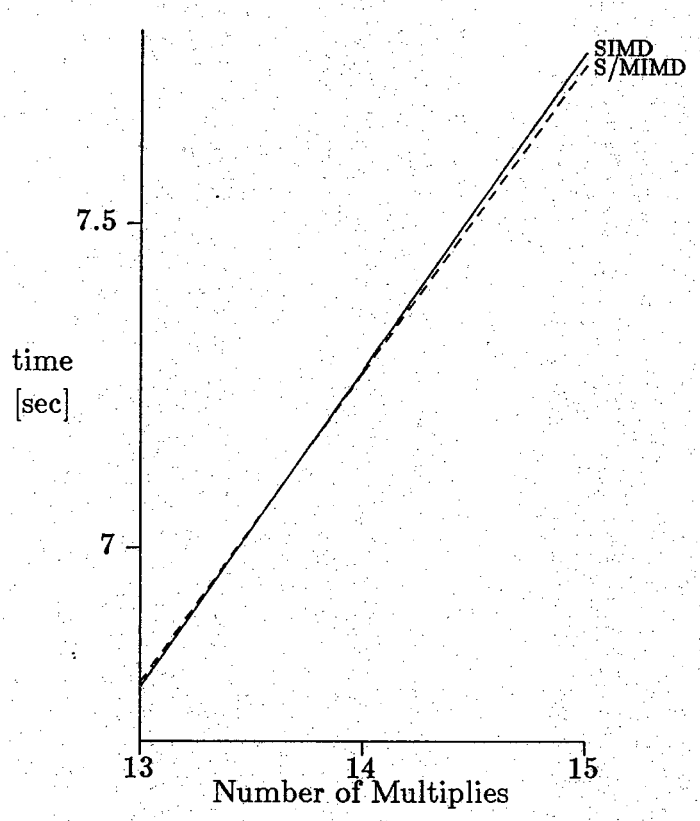


Figure 7: Execution time vs. number of inner loop multiplications for n=64 and p=4.

number of added multiplies is increased. The point at which $T_{SIMD} = T_{S/MIMD}$ was with approximately fourteen added multiplications. This was due to the increase in execution efficiency when the multiplications were executed asynchronously. i.e., fewer processors were idle while waiting for all multiplications to complete.

9. Contributions to Execution Time

To further demonstrate that the execution time advantage was manifested in the multiplication instruction execution time, the contributions of the total execution time of the hybrid and SIMD programs were broken down and plotted. Figures 8, 9, and 10 contain plots of execution time vs. problem size at each of the endpoints and at the crossover point of Figure 7.

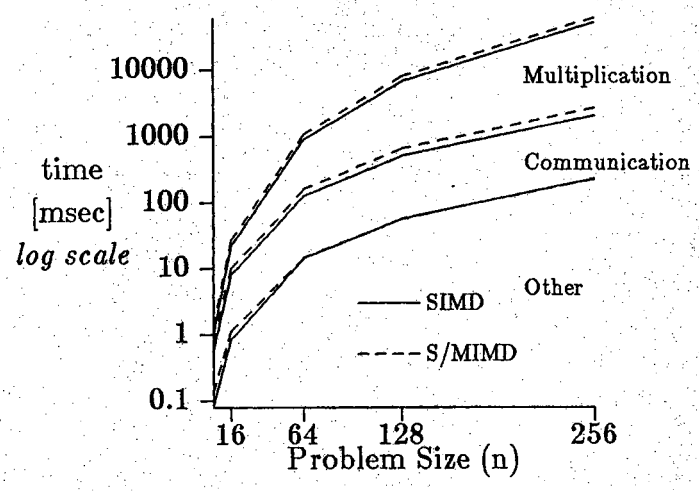


Figure 8: Contributions to execution time for matrix multiplication with one multiply per inner loop and p=4.

The times shown are broken down into: (i) multiplication time, (ii) communication time, and (iii) other contributions such as time for clearing the C matrix and shifting pointers for internal data movement. Multiplication and communication times include related address calculation operations. The multiplication time also includes the addition operation required to add the calculated value to the proper C matrix element. Figure 8 shows clearly that as problem size increases the time required for the multiplications increases faster than the communication time. This was mainly due to to the difference in the order of the communication time and the multiplication time (i.e. $O(n^2)$ vs. $O(n^3/p)$). Due to this difference in time complexity, the time required for the multiplication instructions becomes the largest component of execution time, even without the added multiplication instructions. The S/MIMD program, however, does not execute faster than the SIMD version due to both the control unit instruction overlap and the faster memory access time of the Fetch Unit Queue unless extra data-dependent instructions are added.

In Figure 9, the execution times are equal at n=64. With the total time broken down, it is apparent that the matrix multiplication times are close for all values of n, and when n=64 the matrix multiplication time is less in the S/MIMD program than in the SIMD program. However, the matrix multiplication time was the same because the communication time in the

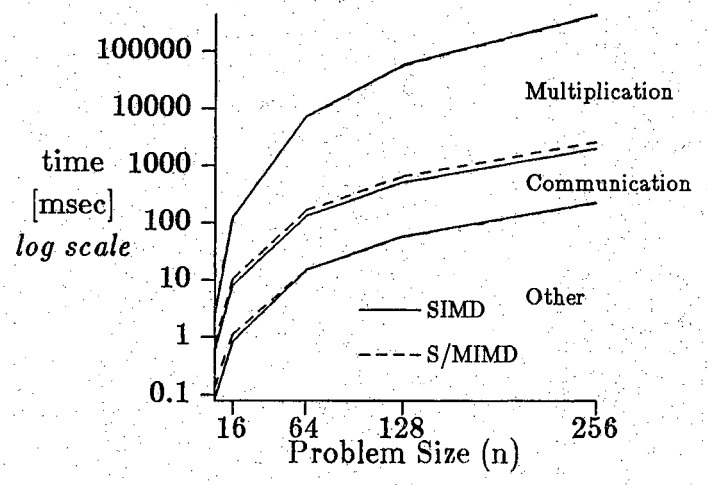


Figure 9: Contributions to execution time for matrix multiplication with 14 multiplies per inner loop and p=4.

S/MIMD version was slightly more than in the SIMD version. Also, it should be noted that this effect would be greater if the constant value representing the instruction fetch time advantage were removed.

Figure 10 demonstrates the advantage provided by the asynchronous multiplication instructions when enough were added to make the other effects diminish in importance. In this version with 30 added multiplications per inner loop the S/MIMD version is faster for the larger values of n and this difference increases with n.

10. Efficiency vs. Problem Size

Figure 11 plots efficiency vs. problem size for the three modes of computation possible on PASM with p=4 as well as the serial case where efficiency is defined as:

$$E = \frac{T_{serial}}{T_{parallel} \times p}$$

The efficiency of the S/MIMD and MIMD versions increased with the problem size, and never reaches or exceeds unity. The reason for the increasing efficiency can be accounted for by the fact that the quantity of communication overhead increases as $O(n^2)$, and the computation increases as $O(n^3/p)$. The

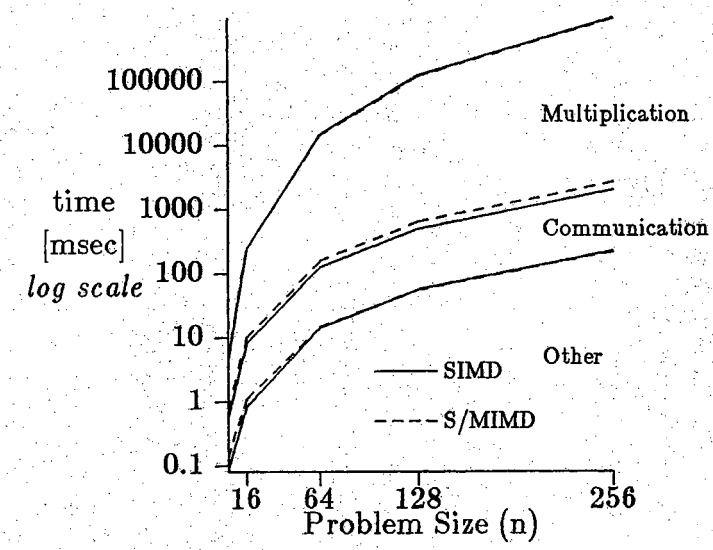


Figure 10: Contributions to execution time for matrix multiplication with 30 multiplies per inner loop and p=4.

best efficiency was 96% for the S/MIMD version and 87% for MIMD version (for n=256 and no added multiplies). The MIMD efficiency was lower due to the extra overhead required for the MIMD communication.

The SIMD version, however, was not only more efficient than the MIMD and S/MIMD versions, but was able to achieve an efficiency greater than unity when compared only to the number of PEs employed. This difference can be attributed to the ability of the PEs to do computation while the MCs are doing looping and other control operations. If the queue can remain non-empty and non-full at all times, it should be possible to eliminate all of the time required for the control operations. Because this amount increases with n, it is not surprising that the benefit also increases with n. This amount of benefit is related to the the ratio of control operations versus computation and communication operations. This does, however, demonstrate that the overlap of control flow and computation is possible and does provide some efficiency benefits — especially for applications that strongly exhibit a large quantity of control flow operations that can be performed on the MCs. This effect was predicted earlier by Kuehn et al in [KuS86].

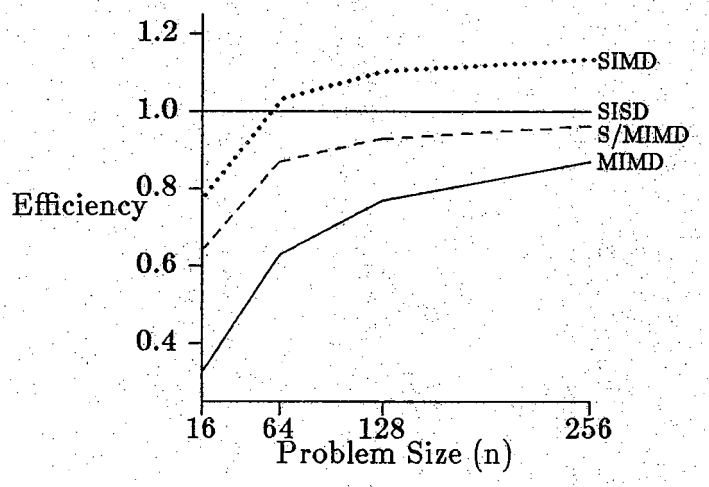


Figure 11: Efficiency vs. problem size for p=4 and one multiply per inner loop.

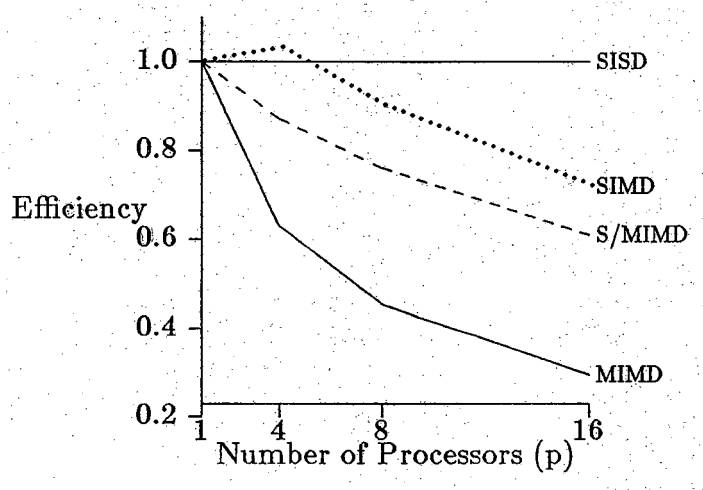


Figure 12: Efficiency vs. number of processors for n=64 and one multiply per inner loop.

11. Efficiency vs. Number of PEs

Figure 12 shows how efficiency drops as the number of processors utilized increases. This drop in efficiency is due to several factors. First, the value of n/p drops as p increases representing a decrease in the amount of computation done by each processor. While this does allow better parallelization of the algorithm, it makes the time consumed by inter-processor communication and other factors not present in the serial version become more significant compared to the time required by the computation portion of the algorithm.

12. Summary

Experiments designed to examine the tradeoffs among the SIMD, SISD, MIMD, and MIMD with barrier synchronized modes on the PASM parallel processing system prototype were described. In particular, the effects of instructions with data dependent execution times were considered. Tests were coded and executed on the prototype. Runtimes for different numbers of multiplies, numbers of processors, array sizes, and modes of parallelism were collected. This data was evaluated and discussed, analyzing the effects of the various parameters in the tests.

The experiments presented used an actual parallel system and pointed out some of the trade-offs among these modes of parallelism. Experiments such as these on working prototypes are important in order to begin to learn how to effectively harness the power of parallel processing.

Acknowledgements: The authors of this paper acknowledge many useful discussions with Pierre Pero, Tom Pusateri, Ed Bronson, Henry Dietz, Wayne Nation, and the other members of the PASM working group.

References

[AnA87] M. Annaratone, E. Arnould, T. Gross, H. T. Kung, M. Lam, O. Menzilcioglu, and J. A. Webb, "The Warp computer: architecture, implementation, and performance," *IEEE Transactions on Computers,* Vol. C-36, December 1987, pp. 1523-1538.

[Cal84] D. A. Calahan, "Influence of task granularity on vector multiprocessor performance," *1984 International Conference on Parallel Processing,* August 1984, pp. 278-284.

[CrG85] W. Crowther, J. Goodhue, R. Thomas, W. Milliken, and T. Blackadar, "Performance measurements on a 128-node butterfly parallel processor," *1985 International Conference on Parallel Processing,* August 1985, pp. 531-540.

[FiC87] S. A. Fineberg, T. L. Casavant, and T. Schwederski, "Mixed-mode computing with the PASM prototype," *25th Allerton Conference on Control, Communications and Computing,* September 1987, pp. 258-267.

[GeS87] E. F. Gehringer, D. P. Siewiorek, and Z. Segall, *Parallel processing: the Cm* experience,* Digital Press, Bedford, MA, 1987.

[Han88] F.B. Hanson, "Vector multiprocessor implementation for computational stochastic dynamic programming," *IEEE Technical Committee on Distributed Processing Newsletter,* Vol. 10, 1988, (to appear).

[Hud88] P. Hudak, "Exploring parafunctional programming: separating the what from the how," *IEEE Software,* Vol. 5, January 1988, pp. 54-61.

[JaM86] W. Jalby and U. Meier, "Optimizing matrix operations on a parallel multiprocessor with a hierarchical memory system," *1986 International Conference on Parallel Processing,* August 1986, pp. 429-432.

[KuN88] J. G. Kuhl, J. J. Norton, and S. R. Sataluri, "A large-scale application of coarse-grained parallel and distributed processing," *IEEE Technical Committee on Distributed Processing Newsletter,* Vol. 10, 1988, (to appear).

[KuS86] J. T. Kuehn and H. J. Siegel, "Simulation based performance measures for SIMD/MIMD processing," in *Evaluation of Multicomputers for Image Processing,* L. Uhr, K. Preston, Jr., S. Levialdi, and M. J. B. Duff, eds., Academic Press, Orlando, FL, 1986, pp. 139-158.

[LuB80] S. F. Lundstrom and G. H. Barnes, "A controllable MIMD architecture," *1980 International Conference on Parallel Processing,* August 1980, pp. 165-173.

[SiS81] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller, Jr., H. E. Smalley, Jr., and S. D. Smith, "PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Transactions on Computers,* Vol. C-30, December 1981, pp. 934-947.

[SiS87] H. J. Siegel, T. Schwederski, J. T. Kuehn, and N. J. Davis IV, "An overview of the PASM parallel processing system," in *Computer Architecture,* D. D. Gajski, V. M. Milutinovic, H. J. Siegel, and B. P. Furht, eds., IEEE Computer Society Press, Washington, D.C., 1987, pp. 387-407.

[Sto80] H. S. Stone, "Parallel computers," in *Introduction to Computer Architecture (second edition),* H. S. Stone, ed., Science Research Associates, Inc., Chicago, IL, 1980, pp. 363-425.

[SuT87] S.Y W. Su and A. K. Thakore, "Matrix operations on a multicomputer system with switchable main memory modules and dynamic control," *IEEE Transactions on Computers,* Vol. C-36, December 1987, pp. 1467-1484.

Experimental Analysis of Multi-Mode Fast Fourier Transforms

Edward C. Bronson, Thomas L. Casavant, Leah H. Jamieson

Abstract

This paper describes a detailed study of parallel fast Fourier transform programs executing on the 30-processor prototype of the PASM parallel processing system. Detailed execution time measurements using specialized timing hardware were made for the complete FFT and for components of MIMD, SIMD, and hybrid SIMD/MIMD (mixed mode) implementations. Compared to a baseline serial FFT, the parallel MIMD, SIMD, and hybrid implementations achieved efficiencies of 0.47, 0.70, and 0.76 respectively. The component measurements isolated the effects of floating point arithmetic operations, interconnection network transfer operations, and program control overhead. Using these detailed component measurements, an expression to project the execution time for an M-point FFT executing on M/2 PASM processing elements (PEs) is derived. The measured execution times for 4-PE and 8-PE programs verify this expression to within 1%. This expression is then used to obtain an accurate extrapolation of the execution time and speedup of the MIMD, SIMD, and hybrid programs to a full 1024-processor PASM system. Overall, the experimental results demonstrate the value of the multi-mode capability of PASM and the suitability of PASM computationally intensive algorithms such as the FFT.

1. Introduction

This paper describes a detailed study of parallel fast Fourier transform programs executing on the 30-processor prototype of the PASM parallel processing system. PASM is a dynamically reconfigurable architecture designed to allow both SIMD and MIMD operation, and to provide the flexible computation and communications capability needed for the wide range of algorithms used in image and speech processing applications [SiS81, SiS87]. In

this paper we use the FFT algorithm as a vehicle for comparing the SIMD, MIMD, and hybrid SIMD/MIMD modes of operation. The FFT programs exercise PASM's floating point hardware for arithmetic operations, the multistage Cube interconnection network, and the specialized timing hardware. Detailed experimental results are obtained for small FFTs on the prototype hardware, then are extrapolated to obtain execution time and speedup figures for a full 1024-*processing element* (*PE*) PASM system. The extrapolation technique is verified analytically and the measurements used for the components of the extrapolation are verified experimentally to within 1% using 4-PE and 8-PE programs.

In the experiments reported, three implementations of a 4-PE single precision floating-point 8-point FFT are studied. These programs were written to examine the trade-offs between the different modes of parallel computation on PASM. An SIMD version performs all FFT operations in SIMD mode. An MIMD version performs arithmetic operations in MIMD mode and polls the interconnection network to determine the network status during data transfer operations. A third program uses barrier synchronization to align the operations of the PEs during interconnection network transfer operation in place of polling and testing the status of the network. This program is a hybrid of the SIMD and MIMD modes of computation: the arithmetic calculations and the network transfers are performed in MIMD mode, while the barrier synchronization operation is performed by using hardware designed for SIMD operation. The hybrid mode gave the best execution time, 9% faster than the SIMD implementation and 39% faster than the MIMD version. Measurements of the components of the implementations isolate the effects of the floating point arithmetic operations, interconnection network transfer operations, and program control overhead, and allow interpretation of the differences in the three overall execution times. Effects due to the number of memory wait states, movement to and from the floating point coprocessor, masking to enable and disable PEs, synchronization, network setup and data transfer, and mode switching are analyzed. Finally the detailed component measurements are used to project a speedup of 814 for a 1024-PE 2048-point hybrid algorithm.

The programs and execution times presented in this paper are among the first applications of the PASM system and are the first floating-point program results obtained on the system. The results demonstrate the value of the multi-mode capability of PASM and its suitability for computationally intensive algorithms such as the FFT. The ability to obtain very detailed

measurements has proven invaluable in understanding and interpreting results from the different implementations of the algorithm and in projecting the results from the prototype to a larger system.

The following section presents an overview of the PASM system and details of the PASM prototype. The fast Fourier transform algorithm is described in Section III. In Section IV, details of the various FFT program implementations are described. Section V presents the measurements techniques used. The experimental results are presented in Section VI and discussed in Section VII.

2. Overveiw of PASM and the PASM Prototype

PASM is a dynamically reconfigurable architecture in which the processors may be partitioned to form independent virtual SIMD and/or MIMD machines of various sizes [SiS81, SiS87]. A 30-processor prototype has been completed and was used in the experiments described in Section VI. This section discusses the PASM architecture characteristics which are most relevant to the reported experimentation. For a more general description of the architecture, see [SiS87].

The *Parallel Computation Unit* of PASM contains N *processing elements* (*PEs*) (numbered from 0 to N−1, where N is a power of 2) and an interconnection network. Each PE is a processor/memory pair. The *PE processors* are sophisticated microprocessors that perform the actual SIMD and MIMD operations. The *PE memory modules* are used by the processors for data storage in SIMD mode and both data and instruction storage in MIMD mode. The *Micro Controllers* (*MCs*) are a set of $Q=2^q$ processors, numbered from 0 to Q−1, which act as the control units for the PEs in SIMD mode and orchestrate the activities of the PEs in MIMD mode. Each MC controls N/Q PEs. PASM has been designed for N=1024 and Q=32. A set of MCs and their associated PEs form a virtual machine. In SIMD mode, each MC fetches instructions and common data from its associated memory module, executes the control flow instructions (e.g., branches), and broadcasts the data processing instructions to its PEs. In MIMD mode, each MC gets instructions and common data for coordinating its PEs from its memory.

The PASM prototype system, completed in December 1986, was built for N=16 PEs and Q=4 microcontrollers. This system employs Motorola MC68000 processors as PE and MC CPUs, with a clock speed of 8 MHz. The interconnection network is a circuit-switched Extra-Stage Cube network

[AdS82], which is a fault-tolerant variation of the multistage cube network. In the following paragraphs, aspects of the prototype system that are essential to the understanding of the algorithm implementations are described.

Consider the simplified MC structure shown in Figure 1. The MC contains a memory module from which the MC CPU reads instructions and data. Whenever the MC needs to broadcast SIMD instructions to its associated PEs, it first sets the Mask Register in the Fetch Unit, thereby determining which PEs will participate in the following instructions. It then writes a control word to the Fetch Unit Controller which specifies the location and size of a block of SIMD instructions in the Fetch Unit RAM. The Fetch Unit Controller automatically moves this block word by word into the Fetch Unit Queue. Whenever an instruction word is enqueued, the current value of the Mask Register is enqueued as well.

PEs execute SIMD instructions by performing an instruction fetch from a reserved memory area called the *SIMD instruction space.* Whenever logic in the PEs detects an access to this area, a request for an SIMD instruction is sent to the Fetch Unit. Only after all PEs that are enabled for the current instruction have issued a request is the instruction released by the Fetch Unit FIFO, and the enabled PEs receive and execute the instruction. Disabled PEs do not participate in the instruction and wait until an instruction is broadcast for which they are enabled. This way, switching from MIMD to SIMD mode is reduced to executing a jump instruction to the reserved memory space, and a switch from SIMD to MIMD mode is performed by sending a jump to the appropriate PE MIMD instruction address located in the PE main memory space.

The SIMD instruction broadcast mechanism can also be utilized for *barrier synchronization* [LuB80] of MIMD programs. Assume a program uses a single MC group, and requires the PEs to synchronize R times. First, the MC enables all its PEs by writing an appropriate mask to the Fetch Unit Mask Register. Then it instructs the Fetch Unit Controller to enqueue R arbitrary data words, and starts its PEs which begin to execute their MIMD program. If the PEs need to synchronize (e.g., before a network transfer), they issue a read instruction to access a location in the SIMD instruction space. Because the hardware in the PEs treats SIMD instruction fetches and data reads the same way, the PEs will be allowed to proceed only after all PEs have read from SIMD space. Thus, the PEs are synchronized. The R synchronizations require R data fetches from the SIMD space. Thus, the Fetch Unit Queue is empty when the MIMD program completes, and subsequent SIMD programs

are not affected by this use of the SIMD instruction broadcast mechanism.

Along with the main MC68000 processor, each PE has a Motorola MC68881 floating-point coprocessor [Mot85]. The MC68881 is a full implementation of the *IEEE floating-point standard.* An IEEE format single precision floating-point value is 32-bits in length. Intended primarily for use as a coprocessor to the MC68020 microprocessor, communication during floating-point operations proceeds as with any peripheral. The PASM prototype hardware permits the MC68000 processor to be run with a 8 MHz clock and the MC68881 coprocessor to operate with a 16 MHz clock.

Each PE contains special purpose hardware timing circuitry. Two independent timers, each consisting of a Motorola MC68230 Timer enhanced with additional TTL counting logic to improve resolution, can be used to count processor clock cycles. After initialization, each timer can be started and stopped by writing a 16-bit word to a timer control register. Each PE contains independent timer circuitry and the timers can be started and stopped when processing in SIMD or MIMD mode. Since timer initialization is performed independently of starting a timer, each timer can be started and stopped multiple times during the execution of a program to measure the elapsed time of non-contiguous portions of code.

The circuit-switched PASM interconnection network is capable of operating in both point-to-point and broadcast modes. In order to communicate with another PE using the network, the initiating PE must set up a path through the network. A path is established by first writing a PE routing tag to the network *Data Transfer Register* (*DTR*). The PE must then set a bit in a control register to instruct the network interface to interpret the value in the DTR as a routing tag for setting up the network. The routing tag will be the first data item received from the network at the beginning of an network transfer. Byte data values may now be written to the DTR and automatically sent through the network. The receiving PE reads the transferred byte from its DTR. At the end of a network transfer, the sending PE must write a "drop path request" to the network control register. This will close the established network path and free the network for further transfers.

The execution time of an MC68000 instruction is dependent upon the speed of the program memory that is used. A memory read or write cycle time requires a minimum of four clock periods. Accessing slower memory will cause the generation of one or more *wait states* which will increase an

instruction cycle time. Each memory wait state requires an additional clock cycle to perform a 16-bit read or write. The memory of each PASM PE contains static RAM which can be switched to operate at 0 or 1 wait state.

In order to make comparisons of the speed of the 16-PE PASM prototype relative to other machines and to compare the relative speeds of SIMD and MIMD instructions, the actual raw performance of PASM in SIMD and MIMD mode was measured on the prototype. Raw processing rates for 16-bit integer addition and floating-point addition operations are given in Table 1. The difference in speed between SIMD and MIMD modes can be attributed to the operation of the Fetch Unit hardware. SIMD instructions are fetched from the Fetch Unit Queue in the MC; MIMD instructions are fetched from the PE memories. The queue delivers data to the PEs with a delay of 2 wait states whereas the PE static RAM memory operates at 0 or 1 wait states. The speed of executing a program in MIMD mode will depend upon the number of wait states of the program memory.

3. Fast Fourier Transform

The *discrete Fourier transform* (*DFT*) of a complex M-point sequence, $s_m, 0 \leqq m < M$, is defined as

$$\mathcal{F}_k = \sum_{m=0}^{M-1} s_m e^{-j(2\pi/M)mk}, \qquad 0 \leqq k < M \tag{1}$$

where $j^2 = -1$ [OpS75]. The direct calculation of the DFT using Equation 1 requires $O(M^2)$ operations. The *fast Fourier transform* (*FFT*) computes the DFT of a sequence in $O(M\log_2 M)$ serial operations. One FFT formulation is the radix two *decimation-in-time* (*DIT*) algorithm. In this algorithm, the M-point input sequence, s, is divided into two M/2-point subsequences

$$s^a_m = s_{2m}, \qquad m = 0, 1, \cdots, M/2-1, \tag{2}$$

and

$$s^b_m = s_{2m+1}, \qquad m = 0, 1, \cdots, M/2-1. \tag{3}$$

The DFT of the sequence s can now be written using the two subsequences as

$$\mathscr{F}_k = \sum_{m=0}^{\frac{M}{2}-1} s_m^a W^{2mk} + \sum_{m=0}^{\frac{M}{2}-1} s_m^b W^{(2m+1)k} \tag{4}$$

$$k = 0, 1, 2, \cdots, M-1 ,$$

where $W = e^{-j(2\pi/M)}$ and is called a *twiddle factor*. By factoring out W^k, Equation 4 can be rewritten as

$$\mathscr{F}_k = S_k^a + W^k S_k^b \tag{5}$$

where S_k^a and S_k^b are the M/2-point DFTS of s_m^a and s_m^b, respectively. Equation 5 shows that an M-point DFT can be computed from two M/2-point DFTs. By halving the number of points in the transform at each stage of the FFT, the DFT of an M-point sequence can be computed in $O(M\log_2 M)$ operations.

Figure 2 is a signal flow graph of an 8-point radix two DIT FFT algorithm mapped to 4-PEs. The algorithm consists of $\log_2 M$ stages. At each stage, M/2 butterfly operations are executed. A butterfly operation is shown in Figure 3. For each butterfly operation the input consists of two complex items, A and B. One complex multiplication and two complex additions are performed, and two complex outputs, X and Y, are generated. The twiddle factor, W^k, used in the calculation of each butterfly is marked in Figure 2 and the value differs from stage to stage and among the stages. It should be noted that $W^0 = 1$ and therefore the first stage of the algorithm requires no multiplications. Similarly, the twiddle factor $W^{M/4} = j$ and therefore, the second stage of the algorithm also requires no multiplications. The M/2 butterfly operations performed within each stage are independent and can be executed in parallel. Parallel algorithms to perform an M-point FFT in M/2 PEs are presented in [Be69], [Pea77], [Sto71], and [JaM86].

Between stages, the PEs must exchange data items before performing the next set of butterfly operations. This exchange can be performed by the cube interconnection function. The *Cube* interconnection function, $cube_c$, $0 \leqq c < \log_2 M/2$, is defined as

$$\mathrm{cube}_c(p_{n-1} \cdots p_{c+1}\, p_c\, p_{c-1} \cdots p_0) = p_{n-1} \cdots p_{c+1}\, \overline{p_c}\, p_{c-1} \cdots p_0, \quad (6)$$

where $p_{n-1} \cdots p_0$ is the binary representation of an arbitrary logical PE address, and $\overline{p_c}$ is the complement of p_c [Sie85].

The complexity of the M-point radix two DIT FFT serial algorithm and the parallel algorithm in M/2 PEs, where M is a power of 2, are given in Table 2.[‡] The multiplication step entries of Table 2 reflect that there are no multiplication steps necessary in the first stage of the DIT FFT since $W^0 = 1$ nor are there any in the second stage of the FFT since $W^{M/4} = j$.

4. Implementation Issues

Three 4-PE parallel FFT programs, one 8-PE parallel FFT program, and one serial FFT program were executed on PASM. Execution time measurements were made for each of these programs. The experimental results are presented in section VI and discussed in Section VII. In this section, we outline the programs that were implemented and discuss relevant details of the implementations.

Three implementations of a 4-PE single precision floating-point 8-point FFT were studied. These programs were written to examine the trade-offs between the different modes of parallel computation in PASM. An SIMD version performs all FFT operations in SIMD mode. A second program is an MIMD version that calculates the butterfly operations in MIMD mode and polls the interconnection network to determine the network status during interconnection network transfer operations. A third 4-PE program uses barrier synchronization to align the operations of the PEs during an interconnection network transfer operation in place of polling and testing the status of the network. This program is a ***hybrid*** of both the SIMD and MIMD modes of computation. Although the butterfly calculations and the network transfers are all performed in MIMD mode, the barrier synchronization operation is performed using hardware designed for SIMD operation. The execution times for each of the program component parts of these three FFT programs were measured. A discussion comparing each of the component

[‡] Parallel FFT algorithms using fewer than M/2 PEs are presented in [JaM86]. In the experiments described here, we consider only the M-point, M/2 PE case. Because of the similarity of the algorithms using fewer PEs to those examined here, similar execution characteristics and speedups can be projected.

parts of these programs is presented in Section VII. Values obtained for the component parts for each of these programs were used to predict the execution time of larger FFTs executing on a greater number of PEs and allows accurate extrapolation of the results to a full 1024-PE PASM system.

An 8-PE SIMD 16-point FFT program was also implemented. The execution time measured for this program was used to verify the projected execution time expression presented in Section VII.

A single PE 8-point FFT program was implemented in order to obtain serial execution times. Execution time measurements from this program were used to calculate the values for speedup presented in Section VII.

All of the programs were written in MC68000 assembly language [Mot84] as straight in-line code with no loops. This generated the fastest possible code and eliminated issues of programming style from the execution time studies. All MC control code such as instructions to perform PE masking and operations to direct the operation of the Fetch Unit were explicitly written into each program.

The programs also used the MC68881 floating-point coprocessor to perform all arithmetic. A floating-point operation is initiated when the MC68000 processor writes an instruction word to the command register of the MC68881. This is followed by successive reads and writes by the MC68000 to the response and operand registers of the coprocessor. The amount of communication with the coprocessor will depend upon the floating-point operation being performed. An operation between two floating-point registers within the MC68881 requires a single write to the command register and a single read of the response register. Moving a single precision value to or from the coprocessor requires two additional 16-bit writes or reads to the operand register. In a typical application, the main processor would test the response register of the coprocessor to determine whether any additional processing is required, whether there is an error condition, or to determine whether the floating-point operation has completed. Detailed analysis of the number of accesses to the coprocessor registers by the MC68000 during floating-point operations has provided exact execution times for these operations on a PASM PE. The execution times for each coprocessor operation are independent of the value of the floating-point data. Efficient procedures were written to interface with the MC68881 and perform the floating-point operations by waiting a constant delay during operation without testing the coprocessor response register. These procedures were originally written to operate the

PE's coprocessor in SIMD mode. This was necessary since branching and testing in SIMD mode may be very inefficient. The resulting procedures were always more efficient than code that polled the coprocessor response register. These *constant delay* procedures were used for both SIMD and MIMD modes of computation. Six different coprocessor operations are used in the various butterfly calculations: floating-point add, subtract, and multiply, single precision moves to and from the coprocessor, and moves between the floating-point registers. Testing the coprocessor for error states during program execution was not performed.

Each cube interconnection network transfer required the transfer of two 32-bit single precision floating-point numbers (the real and imaginary parts of the complex X or Y). Since the network is 8-bits wide, transferring each floating-point number requires 4 writes and 4 reads to the network DTR. A cube interconnection transfer operation proceeds in the following way. First, the sending PE writes the routing tag to the DTR and requests a network path. A cube interconnection network function is non-blocking and the entire network is configured within a few clock cycles after the last PE requests a network path. The transmitted routing tag is read from the receiving PE's DTR. Each floating-point number is then transferred a byte at a time. The receiving PE must reassemble the floating-point number. The partitioning and recombination of floating-point operands are performed in MC68000 registers. After the transfer and reassembly of the second floating point number, the sending PE drops the interconnection network path.

Each program has an *initialization phase*, an *FFT algorithm phase*, and an *output phase*. Execution time measurements are made on the FFT algorithm phase of each program. The processing of the initialization and output phases is performed using both SIMD and MIMD modes of computation. In the initialization phase, the MC and each PE pre-compute and store all necessary data in preparation for the timing of the FFT algorithm phase of the program. During the output phase, the execution time and the transformed data are printed.

The FFT programs are written as efficiently as possible by performing all computations that are not dependent the operations of the FFT in the initialization phase. This optimization includes: ordering and initializing the input data in PE memory, pre-calculating the PE masks used by the MC, and pre-calculating the logical PE number, cube function network routing tags, and FFT twiddle factors in each PE. Each PE has internal access to its own physical PE number. For a fixed PASM partition size, the logical PE number

can be determined from the physical PE number. The order of the input data is dependent upon the logical PE number. The value of the PE masks vary with the number of PEs. The routing tags for each interconnection network operation can be computed from the physical PE number and the physical MC number. The values for the twiddle factors are dependent upon the size of the FFT, the stage number of the FFT, and the logical PE number. These values are calculated by each PE and stored in memory. No pre-loaded register values are assumed. The interconnection network hardware is also initialized in the initialization phase. This only involves clearing control registers so that any existing network connection is dropped.

In the FFT algorithm phase, each PE obtains the two complex floating-point input data items from PE memory, computes the FFT, and stores the transformed data back to PE memory. The storage location of the intermediate data during program execution is dependent upon the number of data points within each PE. Since the M-point parallel FFT programs discussed in this paper are computed using M/2 PEs, each PE contains only two complex data items (four 32-bit single precision floating-point numbers). During the butterfly calculations, all of the data resides in registers of the floating-point coprocessor. Each inter-stage interconnection network operation transfers only one of the two complex values. The data that is not transferred remains in each PE's coprocessor registers during the network transfer. Therefore, it is necessary for each PE to move only two floating-point numbers to and from the coprocessor registers before and after the network transfers. The data that is transferred is stored in the data registers of the MC68000 during the interconnection network operation. In the serial FFT program, all of the data must reside within a single PE. Although MC68000 data registers are not used for network transfer operations, there are not enough data registers or coprocessor floating-point registers to store all of the intermediate data. Therefore, all of the intermediate data is stored in memory. It is necessary to move both of the complex data items to coprocessor registers before each butterfly operation and and return the data values to memory afterwards.

5. Measurement Techniques

This section describes the techniques used to obtain the execution times for the programs and program components presented in Section VI. Execution times were obtained using the special purpose PE hardware timing circuitry described in Section II. The timers were configured to count 8 MHz

clock periods resulting in timing accuracy of ± 125 nanoseconds.

As shown in Table 1, the mode of processing and the memory cycle time will greatly influence the execution time of a program executing on PASM. A block of PE instructions will execute faster in MIMD mode from 0 wait state static RAM memory than the same set of instructions executed in MIMD mode from 1 wait state RAM or in SIMD mode from the Fetch Unit Queue. In order to compare the execution of programs operating in SIMD or MIMD mode and from memory with varying wait states, it is necessary to normalize the memory access time of all instructions. The memory access time for all instructions was normalized to 2 wait states (six clock cycles). This is the PE SIMD instruction bus access time and no normalization is necessary for SIMD instruction fetches. All other memory access cycles must be normalized.

During program execution, only static RAM memory was used within each PE. For SIMD mode, the static RAM was only used for variable storage. For each experiment, the execution time was measured once using 0 wait state static RAM and again using 1 wait state static RAM. The difference between these two execution times is the time required for a single wait state per memory cycle. Adding the difference between these two execution times to the 1 wait state execution time is equivalent to the program executing using 2 wait state memory. By using this 2 wait state normalized execution time, the time of an instruction fetch in SIMD mode from the Fetch Unit Queue is equivalent to an instruction access in MIMD mode from memory. Direct comparison of program times is then possible.

The time required to start and stop the timers will vary according to the mode of computation, the number of PEs enabled, and the access time of the memory in which the instruction are stored. This *timer overhead* was removed from the measured program execution times before the 2 wait state memory access time normalization was calculated.

Execution time measurements were made by inserting instructions to start and stop the timers in the code before program assembly. The execution time for a parallel program is the greatest amount of time required by the MC and any one PE to complete execution. When measuring the execution time of a complete FFT program the timers were started and stopped simultaneously in SIMD mode. The measured times for SIMD mode operations agreed within 1 clock cycle across all PEs. In MIMD mode, since each PE operates independently, the measured execution times across the PEs varied.

All execution times were measured for a single pass through the program. If repeated executions of the program resulted in varying execution times, the measurement was repeated until a clear median was established. This variance in execution times was observed only when executing SIMD mode programs and was less than 3% of the total program execution time. In MIMD mode, repeated execution time measurements were always within 1 clock cycle. The variation in SIMD mode execution times is due to synchronization of instructions in PEs with processor clocks that are not always in phase. Each PE has its own internal independent 16 MHz clock. The 8 MHz clock signal used to operate the MC68000 processor is obtained by dividing the output of a 16 MHz clock. There is no circuitry to synchronize PE clocks. Therefore, the phase of any two 8 MHz PE clocks will differ by as much as one clock cycle of the 16 MHz clock (0.063 ns). The relative phase of any two PE clocks will change during the algorithm as different sets of PEs are enabled and disabled by the MC.

Upon completion of the initialization phase of the program, each PE waits for an instruction from the Fetch Unit Queue. Measurements were made for execution of the entire FFT and for the components of each program. If a measurement of the execution time for the entire FFT is being made, the first instruction executed by all of the PEs (in SIMD mode) is to enable the timing hardware. For the SIMD program, each PE continues to execute only instructions read from the SIMD instruction space. The last instruction of the FFT algorithm phase will disable the timing hardware. For the MIMD and hybrid programs, the next instruction will be a jump from the SIMD instruction space to PE memory. From this point, the PE will execute instructions from its own memory until jumping back to SIMD instruction space at the completion of the FFT algorithm phase to disable the timing hardware. When components of the program are being timed, the timing hardware is enabled and disabled at intermediate points during execution of the FFT.

6. Experimental Results

Execution time measurements of the complete FFT algorithm were made for the three 4-PE 8-point parallel implementations, for the 8-PE 16-point SIMD program, and the single PE 8-point serial program. The execution times for the 4-PE 8-point FFT programs are shown in Figure 4. In addition to these complete FFT execution time measurements, the components of the three 4-PE parallel programs were studied (see Figure 2). These

measurements included the execution time of the FFT stage 1, $cube_1$ interconnection function, FFT stage 2, $cube_0$ function, FFT stage 3, register initialization, and MIMD mode program control overhead.

The execution times for the components of each of the 4-PE programs are shown in Figure 5. The length of each bar in Figure 5 indicates the maximum execution time for each program component. The *FFT stage execution time* includes the time required to compute the floating-point butterfly operation plus the time required to move floating-point data to, from, and within the coprocessor. The *network execution time* is the time to transfer a complex floating-point value from the MC68000 data registers of the sending PE to the data registers of the receiving PE. This includes the time to write the routing tag to the network, request a network path, transfer the data one byte at a time through the network, reconstruct the transferred data, and drop the interconnection network path. A solid line across a bar indicates that while some of the PEs executed the program component at the maximum time indicated by the length of the bar, other PEs only required the time indicated by the solid line. This is due to the specific implementation of the FFT algorithm and will be described later. A dotted line across a bar indicates the minimum execution time for the program component. The measured execution times across all PEs for this component lie between the time indicated by the length of the bar and the time indicated by the dotted line. The times presented in Figure 5 are quite accurate: the summation of the component execution times for each of the 4-PE program sum to within 1% of the execution time for the complete program. The execution times presented here are discussed in the next section.

7. Discussion

7.1. Comparison of Execution Times

Figure 4 shows that the MIMD program has the longest execution time for any of the 4-PE parallel programs. This parallel implementation of the FFT algorithm has a speedup of 1.87 over the serial FFT program. The SIMD program requires 33% less time than the MIMD program with a speedup over the serial FFT of 2.78. The execution time of the hybrid SIMD/MIMD mixed mode program is 9% less than the execution time of the SIMD program. The speedup for this program with respect to the serial program is 3.05. The reasons for the variation in execution times can be determined by examining the individual program components (Figure 5).

In stage 1, each program executes a W^0 butterfly. This butterfly requires 2 additions and two subtractions, and these floating-point operations require the same time when executing in either SIMD, MIMD, or hybrid modes. The additional execution time for the SIMD stage 1 is a result of the data movement from the coprocessor after the butterfly calculation. In the $cube_1$ network transfer that follows stage 1, PEs 0 and 1 transfer Y while PEs 2 and 3 transfer X. In the MIMD and hybrid modes, each PE moves the appropriate complex X or Y value from the coprocessor registers to the data registers of the MC68000 processor. This requires two floating-point move operations. In SIMD mode, PEs 0 and 1 must first be enabled while PEs 2 and 3 are disabled. The complex Y value is then moved from the coprocessor. PEs 0 and 1 are then disabled while PEs 2 and 3 are enabled. The complex X value is then moved from the coprocessor requiring two more floating-point move operations. Compared to the MIMD and hybrid implementations the SIMD mode program requires two additional move operations.

In stage 2, the difference between the execution time for the SIMD mode programs and the MIMD mode programs is even greater than for stage 1. One half of the PEs perform a W^0 butterfly while the other half compute an W^2 butterfly. Both of these butterfly operations require two floating-point additions and two floating-point subtractions. In the MIMD and hybrid program versions, calculation of this stage 2 is straightforward. Each PE moves the recently transferred data item to the coprocessor registers, computes the butterfly, and moves a single complex data item from the coprocessor in preparation for the $cube_0$ network transfer. The SIMD stage 2 operation is much more complex. Although W^0 and W^2 butterflies require the same number of arithmetic operations, the butterfly computations combine the A and B data values in a different order. By using a judicious sequence of masking operations, it is possible to move the data correctly into coprocessor registers so that the addition and subtraction operations can be performed simultaneously in all PEs. Additional masking and data movement is then necessary to prepare for the interconnection network transfer.

Another reason for the longer stage 2 SIMD execution time is the necessity for an SIMD stage computation to leave the data that is not transferred in the correct floating-point registers across all PEs. For example, after stage 2, PEs 0 and 2 transfer the Y butterfly output value over the network. The X value remains in the coprocessor registers and becomes the A input value for the stage 3 butterfly operation. Since the stage 3 butterfly is performed in SIMD mode, the A input value must be in the same coprocessor

registers across all PEs. These registers will be referred to as the *A storage registers.* Another set of coprocessor registers is used as the *B storage registers.* The sequence of arithmetic operations performed during the calculation of the stage 2 butterfly leaves different butterfly output values in the coprocessor registers for the PEs executing the W^0 butterfly than for the PEs executing the W^2 butterfly. It is necessary for PE 0 to obtain the X butterfly outputs from one set of registers and move the values to the A storage registers while PE 2 must obtain the X butterfly outputs from a different set of registers. A similar sequence of operations must be performed by PEs 1 and 3 in order to move the Y butterfly output values to the B storage registers. Since the PEs executing the same butterfly operation must transfer different output values, the data movement required by each PE is different. As part of the SIMD stage 2 calculation, the MC must enable and disable all of the PEs in various combinations in order to move the X and Y output values to the correct A and B storage registers. This data movement is not necessary for the MIMD or hybrid programs since each PE computes the stages independently and knows the storage locations of the data from the previous stage.

In stage 3, PEs 2 and 3 compute butterflies complete with twiddle factors and multiply operations. PEs 0 and 1 compute the less complex W^0 and W^2 butterflies. For MIMD and hybrid mode, the execution time for the butterflies computed by PEs 2 and 3 are indicated by the length of the bar. The execution time for PEs 0 and 1 is indicated by the solid line across the bar. In SIMD mode, all of the PEs execute butterflies with twiddle factors and multiply operations. Since one half of the PEs transferred the A value in the preceding $cube_0$ function and the other half transferred the B data item, extra processing is required by the SIMD stage 3 to enable and disable the two sets of PEs and move the data values to different coprocessor registers.

The execution time required for the interconnection network transfers varies widely among the three program implementations. The SIMD network operation requires the least amount of processing time. Since all PEs execute the network operations in lock-step fashion, the data transfers are synchronized. There is no need to test the network for a pending network transfer or to determine if there is transferred data to read from the DTR. In MIMD mode, each PE executes each butterfly stage independently and no synchrony can be assumed when reaching the interconnection network component of the program. Therefore, it is necessary for each PE to test the network before transferring a data item and to wait on the network for a data

item to become available. This testing and waiting on the network results in a high network transfer time. Like the MIMD program, each PE executes each butterfly stage independently in the hybrid version. The hybrid version performs a barrier synchronization during the interconnection network transfer. Once all of the PEs are synchronized, the data is sent and received without testing the status of the network. The execution time for the hybrid version is slightly greater than for the SIMD version. The difference is the time required to synchronize the PEs.

Since the execution time for the SIMD interconnection network transfer is less than the time for barrier synchronization network transfer used in the hybrid program, it would appear that a faster program could be constructed by using the SIMD mode network transfer. This is not the case. The overhead incurred by jumping to SIMD instruction space before the network transfer and back to MIMD program space for the next butterfly stage exceeds the expected time savings. In addition, each time MIMD mode operation is resumed, it would be necessary to test and branch in order for each PE to determine which butterfly operation it is to perform. The execution time overhead for these test and branch operations will exceed the time for testing and branching of an MIMD program that remains in MIMD mode and uses barrier synchronization.

7.2. Projecting Program Execution Times

In this section, an expression for the execution time of an M-point FFT program running on M/2 PASM PEs, $M \geqq 4$, is presented. The expression is a linear sum of the execution times of the components of an FFT program. For an M-point FFT, the number of each component to sum is either fixed for all size FFTs or can be expressed as a function of M. The expression presented is used to predict the execution times for larger size FFTs using a greater number of PEs and allows us to extrapolate our results to a full 1024-PE PASM system. The values to be used for the terms of the expression were obtained from execution time measurements made on the 4-PE 8-point FFT program. For the 4-PE case, the expression yields execution times for the MIMD, SIMD, and hybrid programs that are within 1% of the measured execution times. The expression was also validated for the 8-PE 16-point SIMD FFT program. The execution time measured for this program is also within 1% of the execution time predicted by the expression.

For M/2 PEs and M data items the execution time of an FFT program can be expressed as

$$\mathbf{T_{FFT}(M/2) = J + R + (A + C)\log_2(M/2) + S_T} \tag{7}$$

where the component execution times are defined as

J	jump to and from SIMD instruction space
R	data and address register initialization
A	a single MIMD test and branch operation
C	complex floating-point cube interconnection network transfer
$\mathbf{S_T}$	execution of all FFT butterfly stages

The total time to execute all of the butterfly stages can be expressed as

$$\mathbf{S_T = S_1 + S_2 + (\log_2 M/2 - 2)S_m + S_f} \tag{8}$$

where the component execution times are defined as

$\mathbf{S_1}$	stage 1 (W^0)
$\mathbf{S_2}$	stage 2 (W^0 and $W^{M/4}$)
$\mathbf{S_m}$	intermediate stage (with a multiplication butterfly)
$\mathbf{S_f}$	final stage

The execution time, $\mathbf{T_{FFT}(M/2)}$, for an M-point FFT program executing on M/2 PASM PEs, $M/2 \geqq 4$, can be projected using the data in Table 3.

The graph shown in Figure 6 illustrates the projected execution times for an M-point FFT program on M/2 PEs for M = 4 to M = 1024. The times are extrapolated with high confidence since the expression used to generate the aggregate times was verified by comparing the predicted 8-PE aggregate time to the actual measured time for an SIMD version of the algorithm. The predicted time was derived by using component times from the 4-PE version. These same component times were then used to determine the projected times shown in Figure 6.

As the number of points in the FFT (and number of PEs) increases, the effect of MIMD network operations (the main difference between the Hybrid and MIMD versions) causes the gap between the performance of the MIMD version and both the SIMD and Hybrid versions to widen. Note the general

logarithmic growth in execution time (the horizontal axis is a log scale) as the number of points in the FFT increases. This was predicted by equations 7 and 8.

7.3. Measurement and Projection of Speedup

In order to analyze further the performance of the MIMD, SIMD, and hybrid implementations, a serial FFT executing in a single PASM PE was implemented and used to obtain estimates of speedup for the parallel programs. The serial version is comparable to the parallel versions except that intermediate values are stored in memory instead of in registers. This is more realistic for the serial implementation since a single processor will not have enough registers to hold these values, whereas the M/2-PE algorithm uses 2 floating-point registers for storage in each PE, independent of M. The resulting speedups for the 4-PE algorithms are 1.87, 2.78, and 3.05 for the MIMD, SIMD, and hybrid implementations respectively, corresponding to efficiencies of 0.47 (MIMD), 0.70 (SIMD), and 0.76 (hybrid).

Figure 7 illustrates the projected speedups for system sizes up to 1024 PEs. These figures are based on the extrapolated execution times of the parallel algorithms obtained with equations 7 and 8, and comparable analytic prediction of the larger serial versions. Note that for 1024-PEs in SIMD and hybrid modes, the predicted efficiency is almost 80%. Hence, the overhead of communication remains low. This is significant and has even greater positive implications for a similar system with a message or packet-switched network: the FFT is nearly worst case with respect to network configuration overheads in a circuit-switched system.

8. Conclusions

This work focused on obtaining performance measurements for various implementations of fast Fourier transform algorithm running on the prototype PASM parallel processing system. Detailed measurements allowed evaluation of the effects of a number of aspects of the architecture on the performance of the FFT programs. Most notable is the significant performance advantage of the SIMD implementation over the MIMD implementation, and the even further improvement attained with a hybrid implementation. The difference between the SIMD and MIMD implementations can be attributed primarily to interconnection network time; the improvement gained with the hybrid version is principally due to MIMD execution of arithmetic operations

combined with barrier synchronization at the points at which data transfers occur. This constitutes one of the first results of this kind, in which controlled experiments on fixed hardware were used to make comparisons of these fundamental modes of computing. The results demonstrate the value of the multi-mode capability of PASM, and the viability of mode switching to obtain "the best of both worlds."

Also notable are the projections in which the information obtained by executing the FFT programs on a small number of processors is used to extrapolate performance for larger FFTs on a larger system. Although 8-point FFTs are used as the basis for these projections, these algorithms exhibit all of the basic parts of larger FFT algorithms. The detailed measurements of the components of the implementations allow us to do a very accurate construction of the execution times and speedups for larger size problems. The projections accurately model interconnection network access, data transfers, floating point arithmetic, coprocessor access, use of registers and memory, and program control overhead. The extrapolation is verified to within 1% by comparing the predicted 16-point 8-PE time to the actual measured time for a 16-point 8-PE implementation. The projections indicate a widening of the gap between the performance of the MIMD version and the SIMD and hybrid implementations due to network operation costs.

All of the programs were written in MC68000 assembly language. Many of the interesting comparisons between the various implementations of the FFT would not have been observed if the programs had not been written at this fine level of detail. The detailed experiments reported here provided significant insight into many aspects of the PASM architecture and prototype implementation. This knowledge will be useful for optimizing high level parallel language compilers designed to produce code for executing on PASM.

9. Acknowledgements

The authors would like to thank Sam Fineberg, Wayne Nation, Pierre Pero, Tom Schwederski, and H. J. Siegel for their many helpful discussions.

References

[AdS82] G. B. Adams and H. J. Siegel, "The extra stage cube: a fault-tolerant interconnection network for supersystems," *IEEE Trans. Computers,* Vol. C-31, May 1982, pp. 443-454.

[Be69] G. D. Bergland, "Fast Fourier transform hardware implementations -- an overview," *IEEE Trans. Audio Electroacoustics,* Vol. AU-17, June 1969, pp. 104-108.

[JaM86] L. H. Jamieson, P. T. Mueller, Jr., and H. J. Siegel, "FFT algorithms for SIMD parallel processing systems," *J. Parallel and Distributed Computing,* Vol. 3, Mar. 1986, pp. 47-71.

[LuB80] S. F. Lundstrom and G. H. Barnes, "A Controllable MIMD Architecture," *1980 International Conference on Parallel Processing,* August 1980, pp. 165-173.

[Mot84] Motorola, *MC68000 16/32-Bit Microprocessor Programmer's Reference Manual,* fourth edition, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1984.

[Mot85] Motorola, *MC68881 Floating-Point Coprocessor User's Manual,* first edition, MC68881UM/AD, Motorola MOS Integrated Circuits Division, Austin, Texas, 1985.

[OpS75] A. V. Oppenheim and R. W. Schafer, *Digital Signal Processing,* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1975.

[Pea77] M. C. Pease, "The indirect binary n-cube microprocessor array," *IEEE Trans. Computers,* Vol. C-26, May 1977, pp. 458-473.

[SiS81] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller, Jr., H. E. Smalley, Jr., and S. D. Smith, "PASM: a partionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Trans. Computers,* Vol. C-30, Dec. 1981, pp. 934-947.

[Sie85] H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies,* Lexington Books, D. C. Heath, Lexington, MA, 1985.

[SiS87] H. J. Siegel, T. Schwederski, J. T. Kuehn, and N. J. Davis IV, "An overview of the PASM parallel processing system," in *Computer Architecture,* D. D. Gajski, V. M. Milutinovic, H. J. Siegel, and B. P. Furht, eds., IEEE Computer Society Press, Washington, D.C., 1987, pp. 387-407.

[Sto71] H. S. Stone, "Parallel processing with the perfect shuffle," *IEEE Trans. Computers,* Vol. C-20, Feb. 1971, pp. 153-161.

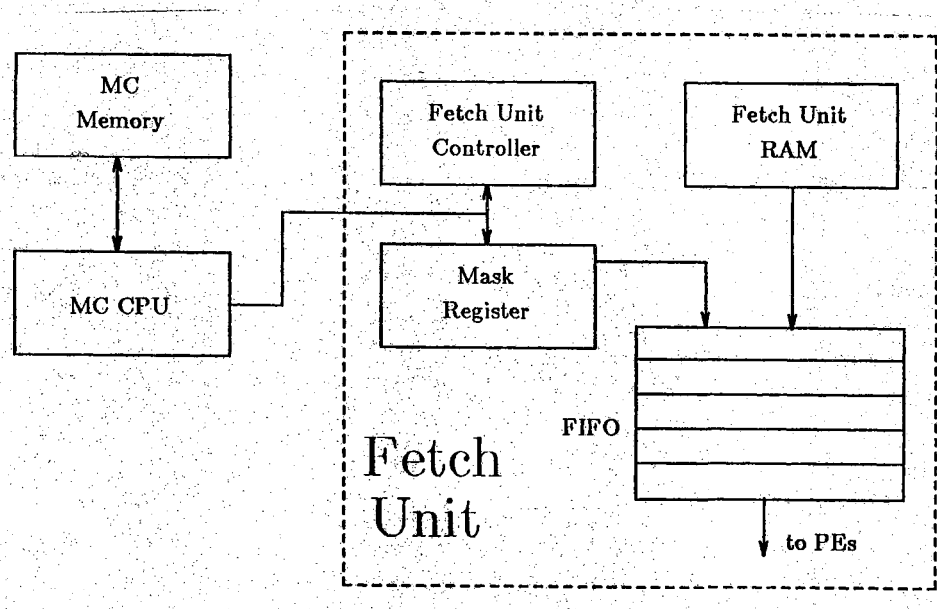


Figure 1. Simplified MC structure.

Table 1. Raw performance of the PASM prototype.

Register-to-Register Operation	Mode	Instruction Memory Wait States	Processing Rate
16-bit integer addition	MIMD	0	25.6 MIPS
		1	21.2 MIPS
	SIMD	-	18.3 MIPS
single precision floating-point addition	MIMD	0	4.7 MFLOPS
		1	4.3 MFLOPS
	SIMD	-	3.9 MFLOPS

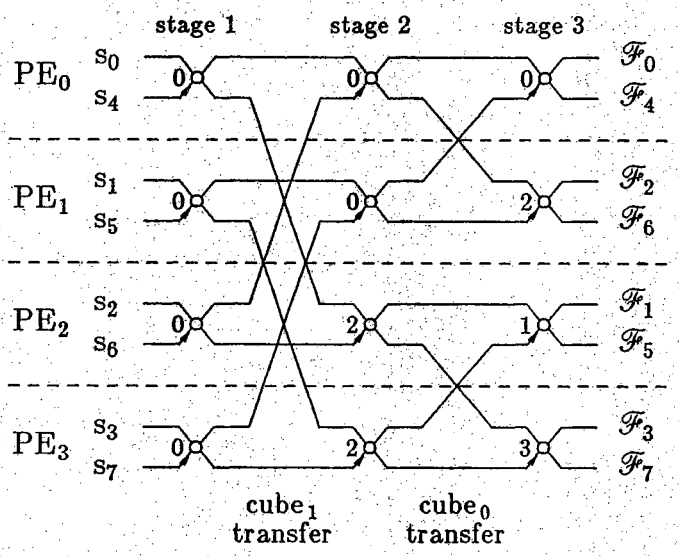


Figure 2. Signal flow diagram of an 8-point FFT on 4 PEs. The value of the twiddle factor for each butterfly operation is W^k, where k is the number adjacent to each butterfly arrow.

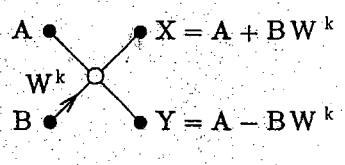


Figure 3. FFT butterfly operation.

Table 2. Complexity of the M-point radix two DIT FFT algorithms.

	Multiplication Steps	**Addition Steps**	**Cube Transfer Steps**
serial	$(M/2)(\log_2 M - 2)$	$M\log_2 M$	-
M/2 PEs	$\log_2 M - 2$	$2\log_2 M$	$\log_2 M/2$

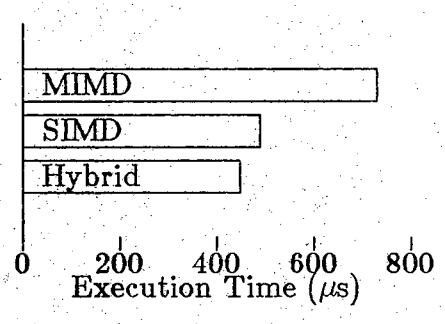


Figure 4. Execution time for 8-point FFT programs on 4 PASM PEs.

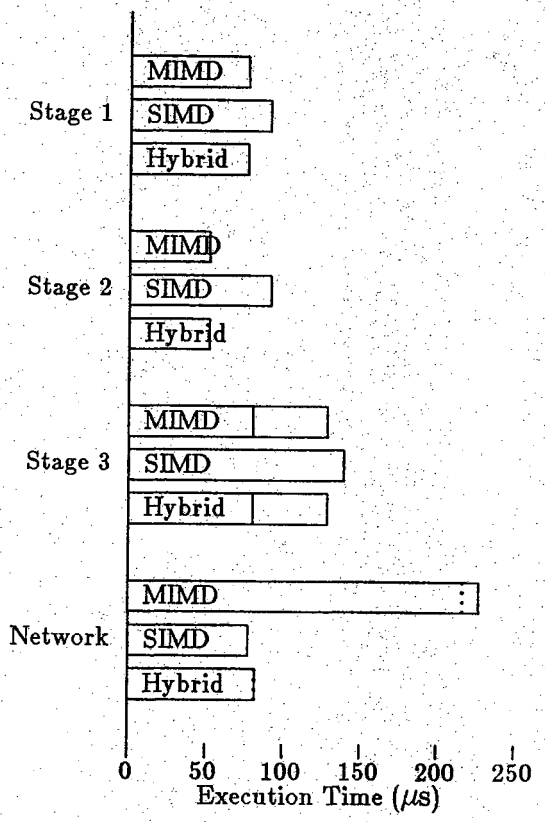


Figure 5. Execution time for components of 8-point FFT programs on 4 PASM PEs.

Table 3. Execution time (μs) for components of FFT programs.

	J	R	B	C	S_1	S_2	S_m	S_f
MIMD	4.000	17.000	5.000	222.000	76.500	52.000	99.500	128.000
SIMD	0.000	13.500	0.000	77.625	90.625	91.500	111.000	139.500
Hybrid	4.000	17.000	5.000	80.750	76.500	52.000	99.500	128.000

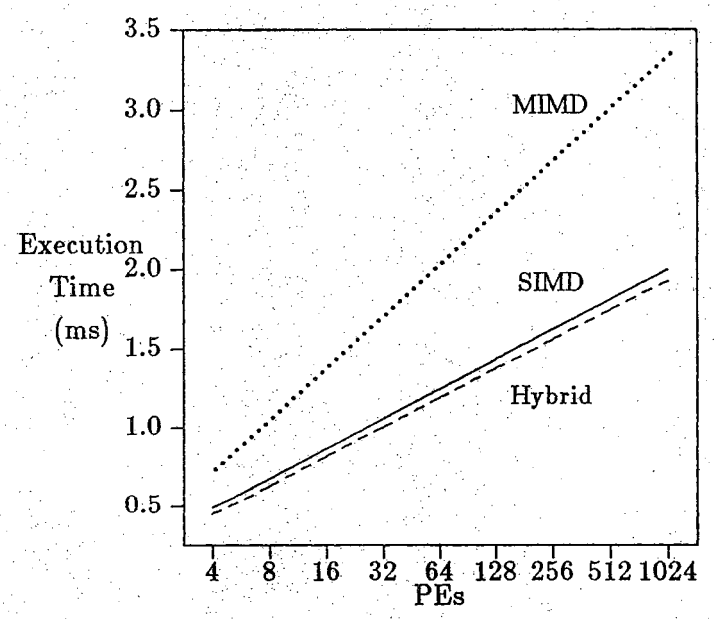


Figure 6. Projected execution times for an M-point FFT program on M/2 PASM PEs, $8 \leqq M \leqq 2048$, where M is a power of 2.

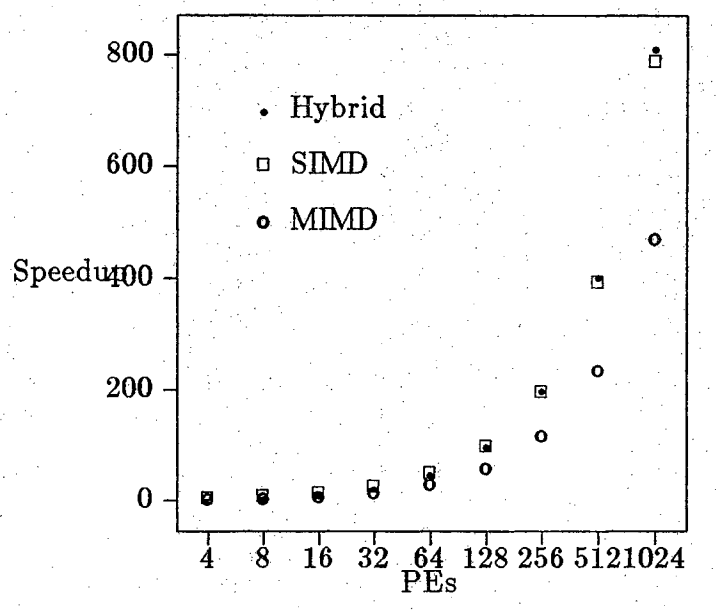


Figure 7. Projected speedup for an M-point FFT program on M/2 PASM PEs, $8 \leqq M \leqq 2048$, where M is a power of 2.

Parallel 2DFFT Implementation

Eng Hwie Loh

Abstract

As part of a coordinated architecture study of novel machines, implementation of 2 Dimension Fast Fourier Transform (2DFFT) on PASM has been conducted. FFT is used in many areas such as image processing, speech analysis, optics, antennas, and random process. The goal of the project is to compare the performance in different modes: SIMD, SIMD/MIMD and MIMD for the implementation of 2DFFT on PASM. The implementation of decimation in time of serial FFT is used as a baseline algorithm for comparison to the parallel version of 2DFFT.

1. Introduction

The PASM prototype was completed in December 1986. There has been an effort to develop application programs to utilize the features of PASM. One of these applications is the Two Dimensional Fast Fourier Transform (2DFFT) which is the main topic of this project report. Discrete Fourier Transform is used in wide areas such as optics, antennas, random process, probability, image processing, and speech analysis. 2DFFT is used in image processing to extract features and improve image quality. The main objective of this project is to implement 2DFFT on PASM, and to compare the performance in different modes: SIMD, SIMD/MIMD and MIMD.

Section 2 gives the background and references to start the project, Section 3 describes the algorithm implemented in this project, Section 4 describes the specifications of the experiment performed, and results are presented in Section 5. In Section 6, discussion and interpretation of results are given, Section 7 provides a trail for someone who wants to continue this project.

2. Background

2.1. Problem-area related references and background

Background on Discrete Fourier Transforms can be found in [Ziemer]. It gives a basic understanding about Fourier Transform, and then it discusses Discrete Fourier Transform and presents an introduction to Fast Fourier Transform. It shows the flow graph of the computation for the FFT. For the complete discussion on Cooley-Tukey algorithm used in this project for the 1DFFT, [Blahut] gives a very detail description of the algorithm.

The reference that explains how to parallelize 2DFFT can be found in [Mueller]. It describes how to parallelize both 1DFFT and 2DFFt, and how to do the transfer using the PASM interconnection network efficiently.

2.2. Relevant Part of PASM

A 30 processor prototype of the PASM system was completed in December 1986, with 16 PEs (PE processors are microprocessors that perform the actual SIMD and MIMD operations) and 4 MCs (Micro Controllers are processors which act as the control units for PEs in SIMD mode and orchestrate the activities of the PEs in MIMD mode. Each MC controls 4 PEs. This system employs Motorola MC68000 processors as PE and MC CPUs, with a clock speed of 8 MHz. The interconnection network is a circuit-switched Extra-Stage Cube network, which is a fault-tolerant variation of the multistage cube network. Since knowledge about the MC and the way in which SIMD instructions are implemented with standard MC68000 microprocessors is essential to the understanding of the behavior that was observed in the experiments, the SIMD instruction broadcast mechanism is overviewed below. Consider the simplified MC structure shown in Figure 1. The MC contains a memory module from which the MC CPU reads instructions and data. Whenever the MC needs to broadcast SIMD instructions to its associated PEs, it first sets the Mask Register in the Fetch Unit, thereby determining which PEs will participate in the following instructions. It then writes a control word to the Fetch Unit Controller which specifies the location and size of a block of SIMD instructions in the Fetch Unit RAM. The Fetch Unit Controller automatically moves this block word by word into the Fetch Unit Queue. Whenever a instruction word is enqueued, the current value of the mask register is enqueued as well. Since the Fetch Unit enqueues blocks of SIMD instructions automatically, the MC

CPU can proceed with other operations without waiting for all instructions to be enqueued.

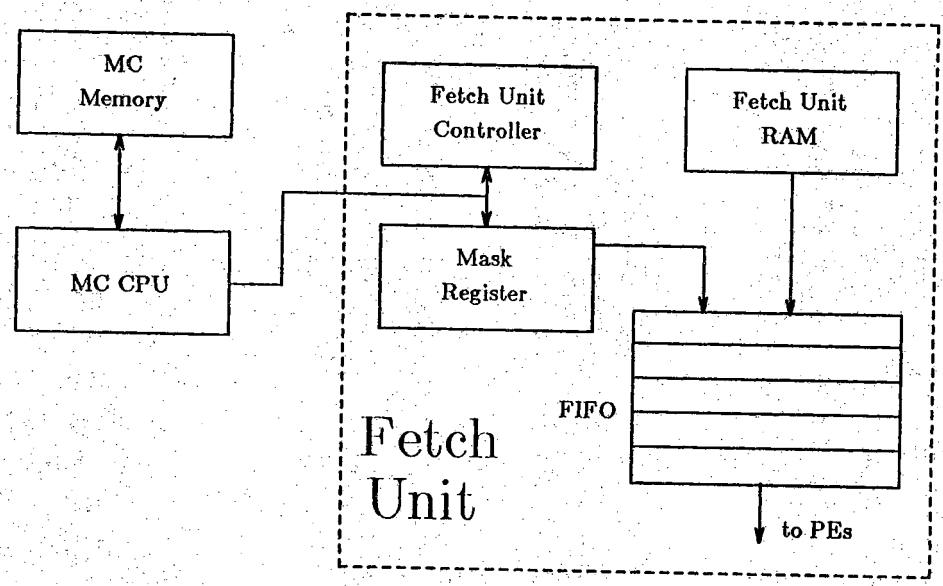


Figure 1: Simplified MC structure.

PEs execute SIMD instructions by performing an instruction fetch from a reserved memory area called the SIMD instruction space. Whenever logic in the PEs detects an access to this area, a request for an SIMD instruction is sent to the Fetch Unit. Only after all PEs that are enabled for the current instruction have issued a request is the instruction released by the Fetch Unit FIFO, and the enabled PEs receive and execute the instruction. Disabled PEs do not participate in the instruction and wait until an instruction is broadcast for which they are enabled. This way, switching from SIMD and MIMD mode is reduced to executing a jump instruction to the reserved memory space, and a switch from MIMD to SIMD mode is performed by sending a jump to the appropriate PE MIMD instruction address located in the PE main memory space.

The SIMD instruction broadcast mechanism can also be utilized for barrier synchronization [Schwed] of MIMD programs. Assume a program uses a single MC group, and requires the PEs to synchronize R times. First, the MC enables all its PEs by writing an appropriate mask to the Fetch Unit Mask register. Then it instructs the Fetch Unit Controller to enqueue R

arbitrary data words, and starts its PEs which begin to execute their MIMD program. If the PEs need to synchronize (e.g., before a network transfer), they issue a read instruction to access a location in the SIMD instruction space. Since the hardware in the PEs treats SIMD instruction fetches and data reads the same way, the PEs will be allowed to proceed only after all PEs have read from SIMD space. Thus, the PEs are synchronized. The R synchronizations require R data fetches from the SIMD space. Thus, the Fetch Unit Queue is empty when the MIMD program completes, and subsequent SIMD programs are not affected by this use of the SIMD broadcast mechanism.

3. Problem Description

To implement 2DFFT algorithm for this project, it is necessary to understand how serial 1DFFT works since the parallel version of 2DFFT algorithm is based on the serial 1DFFT.

3.1. Cooley-Tukey algorithm for 1DFFT

The definition of the one dimensional Discrete Fourier Transform is

$$X(n) = \sum_{k=0}^{N-1} x(k) W_N^{kn}$$

where

N = the number of input samples x
$k <= \{0,1,2, \ldots ,N-1\}$
$n <= \{0,1,2, \ldots ,N-1\}$
$W_N = \exp(-j2\pi/N)$

Multiplication step complexity is N^2.

Note that for N a power of 2, the equation may be written as:

$$X(n) = \sum_{k=0}^{N/2-1} x(2k)\, W_N^{2kn} + W_N^n \sum_{k=0}^{N/2-1} x(2k+1)\, W_N^{2kn}$$

$$X(n + N/2) = \sum_{k=0}^{N/2-1} x(2k)\, W_N^{2kn} - W_N^n \sum_{k=0}^{N/2-1} x(2k+1)\, W_N^{2kn}$$

where n = 0,1,2,3,. . ., N/2 - 1
Multiplication step complexity is now N log N

For the details of the derivation refer to [Blahut]. The decimation in time FFT or Cooley-Tukey algorithm breaks the input data vector into the set of components with odd index and the set with even index. The output vector is broken into the set containing the first N/2 components and the set containing the second N/2 components.

An example showing the flowgraph for an 8 point (N=8) 1DFFT Cooley-Tukey algorithm and can be found in Figure 2. Figure 2 shows that the x(k) inputs are arranged such to perform a butterfly computation for two adjacent inputs. Note that the results of tansformation is in the right order.

Figure 3 shows the implementation of this algorithm in a high-level language.

3.2. 2DFFT

The definition of 2DFFT is

$$X(u,v) = \sum_{l=0}^{N-1} \sum_{m=0}^{N-1} x(l,m)\, W_N^{vm}$$

We can see that to do 2DFFT computation, we can perform 1DFFT on the rows of the N x N matrix and then perform another 1DFFT on the columns of the matrix of the intermediate results. for two different types of instructions.

```
for  k = 1 to m          /* N=2^m */
begin
     le = 2^k
     le1 = le/2
     u = (1.0 , 0.0)
     W = cmplex (cos(π/le1), sin((−π)/le1))
     for j = 1 to le1
     begin
          for i = j to N step le
          begin
               ip = 1 + le1
               t = fa(l,ip) x u
               fa(l,ip) = fa(l,i) - t
               fa(l,i) = fa(l,i) + t
          end
     u = u x W
     end
end
```

Figure 3: Cooley-Tukey algorithm in a high-level language.

3.3 Implementation of 2DFFT on PASM

3.3.1. Procedures

For an N x N input matrix and p PEs, each PE is assigned N/p rows.

N X N inputs matrix
u,v = 0,1,2,..., N-1
l,m = 0,1,2,..., N-1
$W_N = \exp(\ -j2\pi/N\ )$

In the first stage, each PE performs 1DFFT calculations for the rows assigned to that PE to produce an intermediate matrix result. Then, the matrix is transposed, and assigned N/p rows of the new matrix to each PE. 1DFFT is then performed on the rows assigned to that PE. The result of the second stage is a transposed matrix of the 2DFFT transformation.

For a more clear explanation, Figure 4 shows how to parallelize a 2DFFT for N x N inputs and N PEs. First, each PE is assigned a row of the matrix. Then each PE is to do a serial 1DFFT on each row simultaneously. After that, the new matrix is transposed and then each PE is assigned a row of that matrix. Each PE then performs, for the second time, a serial 1DFFT on the row, (i.e. column of the original matrix) it is assigned.

3.3.2. Transpose alternatives

There are two possible methods to do the transpose using the network transfers.

The first is by connecting each PE to its adjacent PE and then to do simultaneous transfers N x N times. For each N transfers, each PE takes and saves the value needed, after N x N transfers each PE will have all the value needed from the transposed matrix (Figure 5).

The second Method is to connect each PE to the PE that has value needed and then do the transfers simultaneously, then drop the path, then again connect to other PE to get the next needed value. After N transfer, the matrix is already transposed (Figure 6).

The second method should take less time to do the transpose since the transfers is in O(N) and the first method is in O(N X N), but the second method will have to do the open and drop path N times.

4. Experiments Performed

The experiments that can be performed at this point are 4 x 4 2DFFT using 4 PEs in SIMD, S/MIMD, and MIMD modes. All operations were 8 bit complex integer operations, and overflow was ignored. The two method to do the transpose are performed to see which one is faster to do the transpose. Method one requires $O(N^2)$ transfers and method 2 requires O(N) transfers but with additional open and drop path.

The SIMD version of the method 1 is not a pure SIMD version. The reason is not possible to write the pure SIMD version is because of the need to save the data needed after each N transfers. Each PE has to save the needed data from different location, so this part has to be executed in MIMD mode. The rest of the program are executed in SIMD mode.

SIMD/MIMD version of method 1 executes the FFT calculations in SIMD mode and does the transfers to transpose the intermediate result matrix in

MIMD mode. The barrier synchronization is used to perform the transfers data.

The MIMD version of method 1 executes the FFT calculations and performs the transpose in MIMD modes. The barrier synchronization is used each time the FFT calculation and the transpose is performed. Clearly the MIMD version requires more jump action from SIMD mode to MIMD mode.

For the method one, the SIMD, S/MIMD, MIMD methods are all performed , but for the method 2 only the SIMD is performed since the correct results could not be obtained (the fault might be in the network transfer though the time will be the same).

The data sets for the experiments in this case will not make any difference, so only one data set was used.

5. Data Measurements Taken

Method 1

SIMD	0.78 milliseconds
S/MIMD	0.842 milliseconds
MIMD	0.888 milliseconds

Method 2

SIMD	1.04 milliseconds

6. Discussion and Interpretation of Results

SIMD version of the program is the fastest among other modes in method 1. The main reason is that this version requires the least overhead to jump from SIMD mode to MIMD mode since the frequency to jump to MIMD mode is also the least. Note that the programs for all three modes are basically the same, only the frequency of jumping from SIMD to MIMD mode makes one mode faster than the others. With that reason in mind, it could be understood that SIMD/MIMD mode is the second fastest since it has more frequency than SIMD but less frequency than MIMD version to jump from SIMD mode to MIMD mode. And the the slower is the MIMD mode with the same reason above.

The result of method 2 is very surprising since it takes longer to the finish executing the program compare to the SIMD version of the method 1. Once again, method 2 only has transfers in O(N) compare to method 2 which is $O(N^2)$, but method 2 has has to open and drop path O(N) compare to 1 time for method 1. From the result, we can conclude that the overhead to do the open and drop path is very high.

7. Future Work

Much further effort is needed in order to utilize this program. One possibility is to expand the program to be able to execute larger input matrix (i.e 1024 x 1024 matrix). To do that, one must write a general program which is able to perform transformations for any input matrix and for any number of PEs. The 2DFFT calculations need to use floating point computations since the results are generally noninteger complex numbers.

8. Conclusions

For the 4 x 4 matrix inputs, method 1 gives a faster speed than the method 2, and for the method 1 the SIMD is the faster, followed by the S/MIMD than MIMD.

The fact that the method 2 is slower though it requires less transfers means that the overhead to do the open and drop path is very significant.

References

[Blahut] Richard E. Blahut, "Fast Algorithms For Digital Processing", Addison-Wesley publishing Company, Reading, Massachussetts, 1983, pp. 115-127.

[Ziemer] Rodger E. Ziemer, *et al.,* "Signals and Systems", Macmillan Publishing Co., Inc., New York, pp. 388-412.

[Mueller] Philip T. Mueller, et., " FFT Algorithms for SIMD Parallel Processing Systems", Journal of Parallel and Distributed Computing 3, 1986, pp. 48-71.

[Siegel] H. J. Siegel, T. Schwederski, J. T. Kuehn, and N. J. Davis IV, "An Overview of the PASM Parallel Processing System," in *Computer Architecture,* D. D. Gajski, V. M. Milutinovic, H. J. Siegel, and B. P. Furht, eds., IEEE Computer Society Press, Washington, D. C.,

1987, pp. 387-407.

[Schwed] T. Schwederski, H. G. Dietz, "Barrier Synchronization in the PASM Parallel Processing System," 3rd SIAM Conference on Parallel Processing for Scientific Computing, Los Angeles, CA, Dec. 1987.

PART III

Speech Processing and AI-Related

Experimental Analysis of SIMD Recursive Digital Filtering on the PASM System Prototype

Michael J. McPheters Jr. and Thomas L. Casavant

Abstract

Experimental analyses of an implementation of an SIMD algorithm for recursive digital filtering using the PASM parallel processing system prototype at Purdue are presented. The algorithm used easily generalizes to use N PEs (processing elements). Timing-based analyses are made based on a four PE version by examining the following constituent execution times: micro-controller execution time, PE execution time, broadcast communication time, and the execution time of five additional phases in the recursive digital filtering summation calculation. Broadcast execution time was found to account for roughly 44% of the total execution time and the implication of this is discussed for larger problem sizes and machine sizes. The total measured execution time is verified through summation of execution times for the various components of the algorithm.

1. Introduction

This paper reports on experimental measurements of an SIMD recursive digital filtering algorithm implemented on the PASM (PArtitionable SIMD/MIMD) parallel processing system prototype at Purdue [SiS81, SiS87]. An SIMD algorithm was chosen to perform recursive digital filtering because SIMD structures most naturally allow for exploitation of the parallelism found in this application [YoS81]. The main purpose of this research project was to provide information for an *Application-Driven Architecture Study,* in which easily understood algorithms are implemented as programs, and controlled experimentation is done with respect to execution time of the algorithm in order to evaluate particular architecture features. This phase of architecture evaluation research represents some of the first experimentation with the broadcasting feature of the PASM Extra-Stage Cube interconnection network [AdS82]. Finally, this work is useful in gaining insight into the potential use of PASM for this type of application. The results reported are focused on showing the difference in execution times among the various phases of the algorithm implementation. For example, when the algorithm is mapped to the machine such that one input sample is assigned to each PE, and one

output is generated per PE, the broadcast execution time was found to be 44.59% of the total execution time, while the partial summation calculation was 12.33%.

Section 2 provides background information and related work while Section 3 gives an overview of PASM and its prototype. Section 4 reviews the basic operations of digital filtering and describes the algorithm that was used. The experiments performed, the results, and a general discussion of results and their implication are presented in Sections 5 through 8.

2. Background and Related Work

Related experimental research has been carried out on several machines through the use of both simulation and experimental techniques. Simulation-based analysis was performed by Yoder and Siegel [YoS81] for the PASM system, and by Su and Thakore for the SM3 system and a hypercube architecture [SuT87]. Experimental work involving measurements on working machines has also been performed. Examples include work involving several machines: the BBN Butterfly [CrG85], Cm* [GeS87], the Encore Multimax [Hud88], the Intel Hypercube [Hud88], PASM [FiC88], and the Warp system [AnA87]. In these efforts, matrix multiplication was normally employed as an example algorithm. Other reported work involving efficiency measurements and algorithm optimization on parallel machines includes work done on an Alliant FX/8 [JaM86, Han88], a CRAY XMP [Cal84], and a combination of Apollo work-stations and an Alliant FX/8 [KuN88].

Research involving digital filtering algorithm implementation has been examined by Hodges et al [HoB80]. This work examined the use of skewed SIMD-mode parallel digital filtering. Using this approach, all PEs execute the same instruction stream, but the execution of given instructions are skewed in time. Work done by Yoder and Siegel examined different strategies for recursive digital filtering computations. Their work compares the use of a systolic array and SIMD algorithms and performs simulation-based analyses of the SIMD versions for PASM [YoS81].

3. Overview of PASM and the PASM Prototype

PASM is a dynamically reconfigurable architecture in which the processors may be partitioned to form independent virtual SIMD and/or MIMD machines of various sizes [SiS81]. A 30-processor (16 in the computation unit) prototype has been completed and was used in the

experiments described in Section 5. This section discusses the PASM architecture characteristics which are most relevant to the reported experimentation. For a more general description of the architecture and prototype, see [SiS87].

The *Parallel Computation Unit* of PASM contains N PEs where N is a power of 2 (numbered from 0 to N−1), and an interconnection network. Each *PE* (processing element) is a processor/memory pair. The *PE processors* are sophisticated microprocessors that perform the actual SIMD and MIMD operations. The *PE memory modules* are used by the processors for data storage in SIMD mode and both data and instruction storage in MIMD mode. The *Micro Controllers (MCs)* are a set of $Q=2^q$ processors, numbered from 0 to Q−1, which act as the control units for the PEs in SIMD mode and orchestrate the activities of the PEs in MIMD mode. Each MC controls N/Q PEs. PASM has been designed for N=1024 and Q=32 (N=16 and Q=4 in the prototype). A set of MCs and their associated PEs form a virtual machine. In SIMD mode, each MC fetches instructions and common data from its associated memory module, executes the control flow instructions (i.e., branches), and broadcasts the data processing instructions to its PEs. In MIMD mode, each MC gets instructions and common data for coordinating its PEs from its memory.

The PASM prototype system, completed in December 1986, was built for N=16 and Q=4. This system employs Motorola MC68000 processors as PE and MC CPUs, with a clock speed of 8 MHz. The interconnection network is a circuit-switched Extra-Stage Cube network, which is a fault-tolerant variation of the multistage cube network.

The PASM network is capable of operating in both point-to-point and broadcast modes. The recursive filtering application described in this work makes use of the broadcast facilities. In order to establish a broadcast communication session, the sending node first must set up a path through the network. This path is established by the execution of PE code which writes a routing tag (for broadcasting, this value is $F0FC) into the DTR (Data Transfer Register). The PE then sets a bit in a second control register to instruct the network interface to interpret the value in the DTR as a routing tag for setting up the network. Once the routing tag has been written to the DTR each PE (sending and receiving) must poll a third control register to verify that the path was actually set up. This polling will place the used routing tag in a PE data register for each PE to verify. Byte data values are then written to the DTR and automatically sent through the network. The

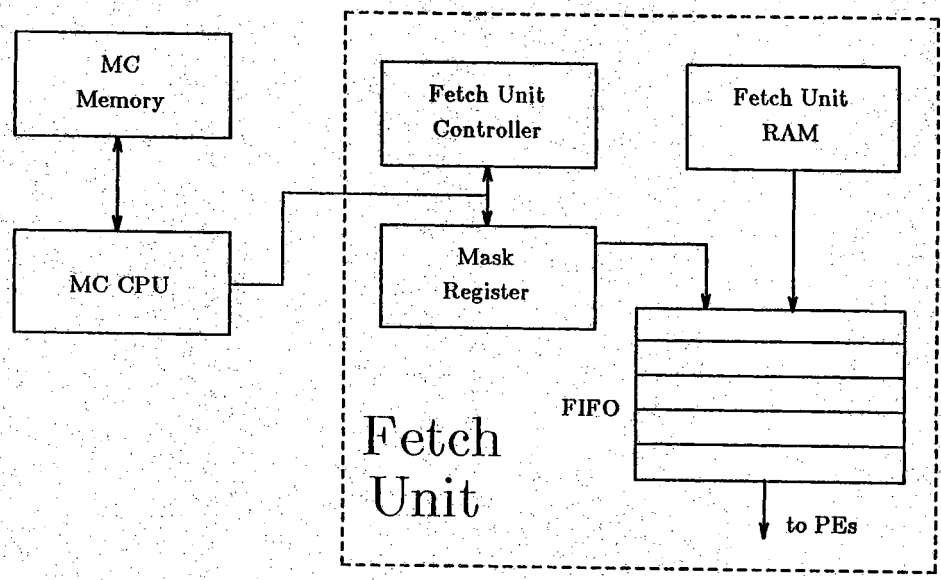


Figure 1: Simplified MC structure.

receiving PEs then initiate a read from the DTR to obtain the sent data value which again is put in a PE data register. To conclude the broadcast session, the sending PE code then initiates a drop path request that will free the network by closing the established network path.

Consider the simplified MC structure shown in Figure 1. The MC contains a memory module from which the MC CPU reads instructions and data. Whenever the MC needs to broadcast SIMD instructions to its associated PEs, it first sets the Mask Register in the Fetch Unit, thereby determining which PEs will participate in the following instructions. It then writes a control word to the Fetch Unit Controller which specifies the location and size of a block of SIMD instructions in the Fetch Unit RAM. The Fetch Unit Controller automatically moves this block word by word into the Fetch Unit Queue. Whenever an instruction word is enqueued, the current value of the Mask Register is enqueued as well. Because the Fetch Unit enqueues blocks of SIMD instructions automatically, the MC CPU can proceed with other operations without waiting for all instructions to be enqueued.

PEs execute SIMD instructions by performing an instruction fetch from a reserved memory area called the *SIMD instruction space*. Whenever logic in the PEs detects an access to this area, a request for an SIMD instruction is sent to the Fetch Unit. Only after all PEs that are enabled for the current

instruction have issued a request is the instruction released by the Fetch Unit FIFO. The enabled PEs then receive and execute the instruction. Disabled PEs do not participate in the instruction and wait until an instruction is broadcast for which they are enabled. This way, switching from MIMD to SIMD mode is reduced to executing a jump instruction to the reserved memory space, and a switch from SIMD to MIMD mode is performed by sending a jump to the appropriate PE MIMD instruction address located in the PE main memory space.

In order to make comparisons of the speed of the PASM prototype relative to other machines and to compare the relative speeds of SIMD and MIMD instruction fetches, the actual raw performance of PASM in SIMD and MIMD mode was measured on the prototype and is illustrated in Table 1 in MIPS (millions of integer instructions per second) for two different types of instructions.

Mode	Operation	Processing Rate
SIMD	16-bit Reg.-to-Reg. add	22 MIPS
MIMD	16-bit Reg.-to-Reg. add	18 MIPS
SIMD	16-bit Reg.-to-Mem. add	6.4 MIPS
MIMD	16-bit Reg.-to-Mem. add	6.0 MIPS

Table 1: Prototype raw performance.

4. The Recursive Digital Filtering Algorithm

4.1. Algorithm Used

The basic operations in digital filtering are the computation of sum of products terms, with output y_m given by

$$y_m = \sum_{k=1}^{p} a_k \, y_{(m-k)} + \sum_{k=0}^{q} b_k \, x_{(m-k)} \quad p,q \leqq m \leqq M$$

where x_m is a sample input to the filter. The a_k's and b_k's are the filter coefficients which define the characteristics of the filter operation to be performed, and M is the number of samples in the signal to be filtered

[YoS81]. The parallel recursive digital filtering algorithm used to compute y_m is shown in Figure 2.

ADDR = Address of PE (e.g., ADDR = 0 in PE 0)
DTRin = Data Transfer Register input to the PE
DTRout = Data Transfer Register output from the PE
COEF[] = Vector of coefficients
FLAG[i] = Equals 1 if COEF[i] is an "a" coefficient
SUM = Contains partially computed y_m

```
for j := 0 to M+p+q-1 do
  where ADDR = j mod (p+q+1) do        /* Select PE containing
      DTRout := SUM;                    * new y value y_{j-(q+1)}
                                        */

      SUM := 0;                         /* Broadcast that y and
      broadcast;                         * start new sum there
                                        */

  where FLAG[j mod (p+q+1)] = 1 do      /* In each PE, select
      TMP := DTRin;                      * either broadcast y
                                        */
  elsewhere do
      TMP := x_j;                       /* or the new x value
                                        */

  SUM := SUM + TMP *
    COEF[j mod (p+q+1)];
```

Figure 2: General SIMD Digital Filtering Algorithm.

The first step in the high level algorithm is to select which PE contains the newly computed y_m value based on the loop index, j, and number of summation terms, p+q+1. The selected PE will then be enabled and the SUM (i.e., y_m) will be broadcast to all PEs, including itself, and then reset to zero to begin calculating a new y_m. In the second step, using the FLAG matrix, each PE must determine which data value to use in the running sum calculation and store it in TMP. This value stored in TMP may be either the broadcast y_m output value or the new x_m filter input value. In the third step, because each PE holds a copy of the filter coefficients, the algorithm makes use of the loop index and number of summation terms, to determine which coefficient to use in a given step of the algorithm. The third step also involves computing the partial running sum

within each PE using one multiplication and one addition.

The broadcast flow for the y_m calculation is illustrated in Figure 3 where the double boxes indicate the start of a new y_m computation.

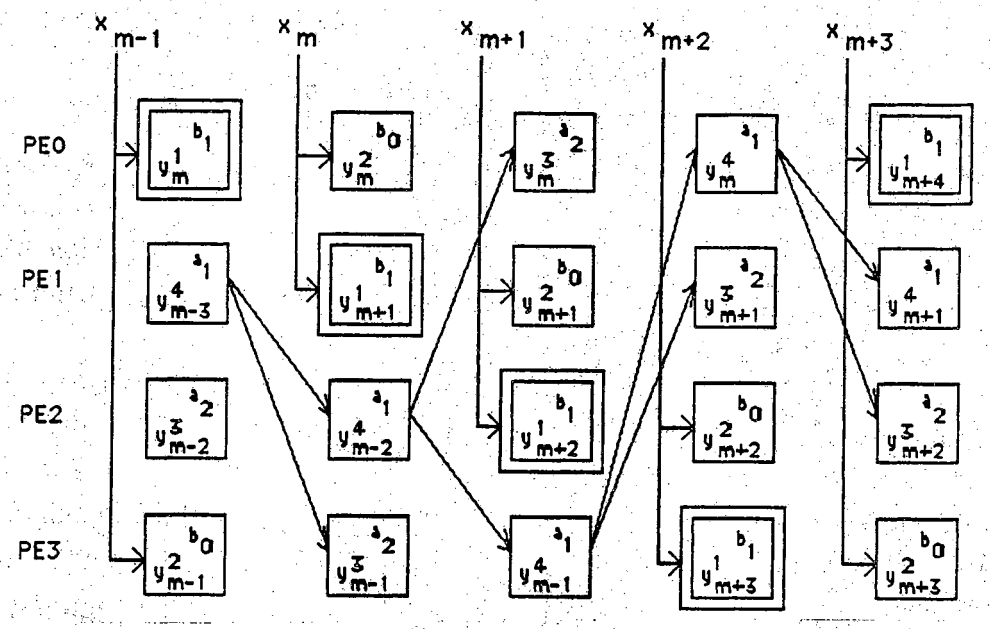


Figure 3: Data Flow Diagram for p = 2 and q = 1.

The operations executed during one stage (i.e., one loop iteration) include one selection of a filter coefficient, two broadcasts, one addition, one multiplication, and one scalar assignment. Therefore, because all PEs are effectively active computing one part of y_m, one output (y_m) is completed at each stage.

4.2. Implementation Discussion

The algorithm of Figure 2 was implemented in 68000 assembly language on the PASM system prototype. The "where...do" and "where...elsewhere" statements are used to indicate the setting of the Mask Register in the Fetch Unit. This conditional evaluation for the "where" is done in the MC in order to determine which PE or sets of PEs are to be activated for the set of SIMD instruction(s) following the "do". In the "where...elsewhere" statement, the conditional "where" is also evaluated in the MC. The PEs for which the condition holds are enabled and execute the SIMD instruction(s) following the "do", while the remaining PEs execute the SIMD instruction(s) following the "elsewhere do". The FLAG matrix is also evaluated in the MC. When FLAG[i]

is 1, then the associated filter coefficient in COEF is an "a" coefficient which would indicate that a PE should use the broadcasted y_m value, otherwise (i.e., when FLAG[i] = 0) a new x_m filter input value should be used. Finally, the coefficient matrix is set up such that each PE only needs a single row of the matrix to compute y_m as shown in Figure 4.

PE	COEF[0]	COEF[1]	COEF[2]	....COEF[p+q−1]	COEF[p+q]
0	b_q	b_{q-1}	b_{q-2}	....a_2	a_1
1	a_1	b_q	b_{q-1}	a_3	a_2
2	a_2	a_1	b_q	a_4	a_3
.	.	.	.	.	.
.	.	.	.	.	.
p	a_p	a_{p-1}	a_{p-2}	b_1	b_0
p+1	b_0	a_p	a_{p-1}	b_2	b_1
.	.	.	.	.	.
.	.	.	.	.	.
p+q-1	b_{q-2}	b_{q-3}	b_{q-4}	....b_q	b_{q-1}
p+q	b_{q-1}	b_{q-2}	b_{q-3}	....a_1	b_q

Figure 4: Skewed Coefficient Storage.

4.3. SIMD vs. Systolic

Both systolic array and SIMD structures are well suited for exploiting the parallelism inherent in certain tasks performed on vectors and arrays [YoS81]. There are two potential advantages of choosing an SIMD implementation. First, an SIMD machine supporting dynamic broadcast reconfiguration allows for a value of y_m to be computed every time unit, whereas a systolic implementation would require two time units. The systolic implementation allows flow between cells in a pipelined fashion such that communication with the outside world can only occur at the "boundary cells" [Kun82]. For a systolic implementation the x_m values flowing "up" the pipeline must be synchronized with the y_m values flowing "down" the pipeline so that they meet in the correct PE with the correct coefficient [YoS81]. Therefore, valid data only exists in the even numbered PEs during odd numbered cycles and in odd numbered PEs during even numbered cycles, hence effectively requiring two time units for each y_m computation. The broadcast for an SIMD implementation can be handled by using the

interconnection network to transfer the data item to the set of desired PEs (as done on PASM), hence effectively requiring only one time unit for each y_m computation. Second, an SIMD implementation easily allows for strip-mine mapping of larger problem sizes to smaller machines when the problem size increases beyond the number of PEs available.

5. Experiments Performed

Timing measurements were made for eight separate phases of this SIMD implementation of recursive digital filtering. All operations were performed with integer byte-operands (8-bit), and overflow was ignored. The experimental results were obtained from the calculation of 52 y_m values where the 52 x_m inputs were unity and the filter coefficients a_k and b_k were also set to unity. This choice of input data simplified debugging, but also permitted gathering of the needed timing measurements.

The following eight time measurements were made using the system's internal timers (Motorola 68230) and were repeated five times. The five trials were used to reduce anomalous data resulting from asynchronous hardware behavior[1]. The timer clock period was four microseconds, therefore, five decimal places to the right of the decimal point were significant.

1. *MC execution time for computing* y_m. This is the aggregate algorithm time to compute a single y_m value as measured in the MC. It is the total measured MC time divided by 52 to obtain the average MC execution time for computing y_m. The timer chip on the MC board was used.
2. *PE execution time for computing* y_m. This is the aggregate algorithm time to compute a single y_m value as measured in the PE. It is the total measured PE time divided by 52 to obtain the average PE execution time for computing y_m. The PE timers were used and the largest single PE time of the four PEs was reported.
3. *Broadcast communication time.* This is the time needed to establish the communication path from one PE to the other PEs and itself, send and receive data, and then to drop the communication path so that a new path may be established in the next stage.

---

[1] The PEs in SIMD mode operate on separate clocks, however hardware synchronization is done on word fetches. Thus, some PEs may take more time to fetch complete instructions than others.

4. *Path acknowledge time.* This is the time needed for each PE to acknowledge receipt of a routing tag. This is a direct measure of the amount of serialization caused by SIMD execution of the broadcast.

5. *Summation calculation execution time.* This is the time needed to compute one stage of the summation. This is dominated by the coefficient multiplication time and the time required to add this product to the partial running sum in each PE.

6. *Execution time for choosing* y_m *after broadcast.* This is the time needed to determine if y_m should be used for the y_m calculation in the current stage based on the FLAG matrix. Choosing y_m involves requesting a network read and storing the read y_m value in TMP.

7. *Execution time for choosing* x_m. This is the time needed to determine if x_m should be used for the y_m calculation in this given stage based on the FLAG matrix. Choosing x_m involves indexing into the data array of x values using the loop index and then storing it in TMP.

8. *Execution time to increment mod 4 stage counter.* This is the time needed to increment the stage counter by one or reset it to zero when the fourth stage has been reached. There are four stages because there are four terms in the y_m calculation.

6. Data Measurements

The data measurements appear in Table 2.

Measurement	Time	Percent of Total Time
MC execution time	0.4281	100%
PE execution time	0.4311 *	100%
Broadcast execution time (B)	0.1880	44.59%
Path acknowledge time (A)	0.0520	12.33%
Summation calculation (S) execution time	0.0520	12.33%
Execution time for y_m (CY)	0.0632	7.50%
Execution time for x_m (CX)	0.0680	8.06%
Execution time for (X) counter increment	0.0640	15.18%

*greater than MC execution time due to MC overhead to start and stop timer in PEs.

Table 2: Experimental Measurements
(All times in milliseconds.)

Note, one can verify the total time to calculate four y_m values based on Equation 1.

$$4\ y_m = 4\ B + 4\ A + 4\ S + 2\ CY + 2\ CX + 4\ I \qquad (1)$$

To verify the data values obtained, note that the left side of this equation is equal to four times the PE execution time. The measured PE time was 1.7244ms, and the right hand side is equal to 1.6864ms. This represents approximately a 2.20% error which is insignificant relative to other differences noted. This 2.20% error occurred for primarily two reasons. First, the PE execution time includes the overhead time for manipulating the timers in each PE which would account for the higher left-hand side of the equation. Second, the measured time for each phase is an average of five trials which would reduce the effects of extreme data measurements on the right side.

The percentages in Table 2 are expressed relative to the right hand side of Equation 1. For example, the broadcast execution time percentage of 44.59% is calculated by dividing four times the broadcast execution time (0.7420ms) by the total execution time (1.6864ms).

7. Discussion and Interpretation

As seen from Table 2, the execution time for the calculation of y_m consists of 44.59% broadcast time. This means that over 40% of the total calculation time is spent sending previous values of y_m to the other PEs. The PE that has completed its y_m calculation must set up the network for the broadcast, broadcast y_m, and drop the network broadcast path, thus causing a long serialization which has great negative potential impact for larger numbers of processors due to effects of Amdahl's Law.

The actual summation time (multiplication and addition) for this mapping is only 12.33% of the y_m calculation time. Each PE needs to compute the partial running sum by multiplying either y_m or x_m by a filter coefficient and then adding it to the previous partial sum. The coefficient and previous partial sum stored in each PE memory allows them to be accessed locally. This fact accounts for this relatively small time contribution.

From these two measurements, it may be concluded that the present mapping of one x_m and y_m value per PE results in very inefficient use of processing resources. If the number of samples per PE increased, the broadcast time would remain constant while the percentage of time required for carrying out the summation will increase. At the point where the parallelism benefit derived from dividing the problem across PEs increases past the overhead from broadcasting, a point of marginal efficiency may be defined. Beyond this point the parallel version is justified with respect to its SISD (serial) counterpart. Below this point, the communication overhead causes the execution time to be worse then it would be for a serial implementation. The major implication is that for realistic problem sizes (e.g., $N \geqq 64$) that a relatively modest number of PEs (e.g., 4-16) would provide tangible improvement over serial execution and for large numbers of PEs, such resources could likely not be well-utilized.

The percentage of time spent choosing y_m and x_m are 7.50% and 8.06%, respectively. The time for choosing y_m is relatively low because it is in a data register after a network read is initiated. The time for choosing x_m is also relatively low, compared to other contributions, because it is accessed locally from each PE memory, yet slightly larger than y_m because the access is to memory versus a data register.

Finally, the y_m calculation includes some pure overhead. This overhead includes the counter increment time used to index into the coefficient array stored in each PE. Also, the path acknowledge time is overhead in that a network read is required to flush the routing tag from the network.

8. Future Implications and Extensions

The reported work represents early findings from the past 6-8 months. While the results offered here are quite useful, one of the greatest uses is toward guiding future work. First, this implementation was for byte data which quickly becomes insufficient to store the running sum. Therefore, versions are being written for word and floating point data. Second, the problem size can be increased such that the implementation would require more PEs (i.e., 8 or 16). The problem size in this case is increased by increasing the number of terms in the y_m calculation to fit the number of PEs available. Finally, the problem size can be increased again, but this time the number of PEs would be less than the number of terms in the y_m calculation. Therefore the efficiency-related hypothesis of the previous section may be verified.

This future work would focus on the broadcast feature of PASM. In particular, the point where the negative effects of the broadcast communication time are overcome by the positive effects of parallelism (i.e., the marginal efficiency point) can be achieved when the problem size is greater than the number of PEs. From the current results, we anticipate that this point would occur when the ratio of problem size to the number of PEs is equal to four. However, when increasing the problem size beyond the number of available PEs, a new approach to the algorithm implementation also needs to be examined.

Another area of experimentation would be to analyze possible data dependencies by examining the impact that word data has on the execution times. In this case, the broadcast time would most likely increase while the other times would remain relatively constant. These results will indicate the importance of large bandwidth systems and also allow prediction of execution times for floating point data.

Aside from the possibilities for future work, having coded the given algorithm in assembly language pointed out an additional implication of this work. When compared to programming in a HLL (High Level Language), programming in assembly language is arguably more difficult. This relative difficulty of assembly language programming would make large application programs rather burdensome to code, thus there is a need for more abstract programming support on PASM. However, our purpose has been to examine the characteristics of an experimental architecture. In doing so, assembly language was used by necessity. As a result two unexpected observations resulted.

First, because there is some overhead associated with using a HLL, it is not possible to fully achieve the performance benefits as seen with low level assembly

language programming. Assembly language allows for more efficient coding of loops, array access, and access to the network for interprocessor communication which can not be provided with a HLL because of its higher level of abstraction. Second, having an operating system on PASM would again not typically allow for the writing of efficient code as seen here because of the overhead associated with an operating system. An operating system, in general, would decrease the overall performance of most programs because of its underlying purpose to control and manage the security and integrity of the whole system environment. Program performance would be inhibited because the operating system controls parts of the system in a way more general than that needed depending on the specific application. Thus, these services would be considered overhead. These observations will help in determining which characteristics would be needed in a HLL in order to maintain comparable speed-up gains as seen with assembly language programming with no resident OS.

9. Summary

Experiments designed to measure execution times of the various phases in SIMD recursive digital filter calculation of y_m on PASM were described. An equation was given that verified the total y_m calculation time with respect to the various phases in the summation calculation. The results reported show the difference in execution times among the various phases of the implementation. In particular, the broadcast execution time was found to be 44.59% of the total execution time while the partial summation calculation was only 12.33% of the total execution time.

The experiments presented used an actual parallel machine (the PASM system prototype) and showed that broadcast communication time is a significant part of SIMD recursive digital filtering algorithms.

10. Acknowledgements

The authors would like to thank Dr. Leah Jamieson, Ed Bronson, Dr. Thomas Schwederski, Sam Fineberg, Wolfram Disch, and the PASM applications group for their many useful discussions.

References

[AdS82] G. B. Adams III and H. J. Siegel, "The Extra Stage Cube: A Fault-Tolerant Interconnection Network for Supersystems," *IEEE Transactions on Computers,* Vol. C-31, May 1982, pp. 443-454.

[AnA87] M. Annaratone, E. Arnould, T. Gross, H. T. Kung, M. Lam, O. Menzilcioglu, and J. A. Webb, "The Warp Computer: Architecture, Implementation, and Performance," *IEEE Transactions on Computers,* Vol. C-36, December 1987, pp. 1523-1538.

[Cal84] D. A. Calahan, "Influence of task granularity on vector multiprocessor performance," *1984 International Conference on Parallel Processing,* August 1984, pp. 278-284.

[CrG85] W. Crowther, J. Goodhue, R. Thomas, W. Milliken, and T. Blackadar, "Performance measurements on a 128-node butterfly parallel processor," August 1985, pp. 531-540. *1985 International Conference on Parallel Processing,*

[FiC88] S. Fineberg, T. Casavant, T. Schwederski, H.J. Siegel, "Non-Deterministic Instruction Time Experiments on the PASM System Prototype," *1988 International Conference on Parallel Processing,* Chicago, August, 1988.

[GeS87] E. F. Gehringer, D. P. Siewiorek, and Z. Segall, *Parallel Processing: The Cm* Experience,* Digital Press, Bedford, MA, 1987.

[Han88] F. B. Hanson, "Vector Multiprocessor Implementation for Computational Stochastic Dynamic Programming," *IEEE Technical Committee on Distributed Processing Newsletter,* Vol. 10, 1988.

[HoB80] C. J. M. Hodges, T. P. Barnwell, III, and D. McWhorter, "The Implementation of an All Digital Speech Synthesizer Using a Multiprocessor Architecture," *1980 IEEE International Conference on Acoustics, Speech, and Signal Processing Proceedings,* April 1980, pp. 855-858.

[Hud88] P. Hudak, "Exploring Parafunctional Programming: Separating the What from the How," *IEEE Software,* Vol. 5, January 1988, pp. 54-61.

[JaM86] W. Jalby and U. Meier, "Optimizing Matrix Operations on a Parallel Multiprocessor with a Hierarchical Memory System," *1986*

International Conference on Parallel Processing, August 1986, pp. 429-432.

[KuN88] J. G. Kuhl, J. J. Norton, and S. R. Sataluri, "A Large-Scale Application of Coarse-Grained Parallel and Distributed Processing," *IEEE Technical Committee on Distributed Processing Newsletter,* Vol. 10, 1988.

[Kun82] H. T. Kung, "Why Systolic Architectures?," *Computer,* January 1982, pp. 37-46.

[SiS81] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller, Jr., H. E. Smalley, Jr., and S. D. Smith, "PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Transactions on Computers,* Vol. C-30, December 1981, pp. 934-947.

[SiS87] H. J. Siegel, T. Schwederski, J. T. Kuehn, and N. J. Davis IV, "An overview of the PASM parallel processing system," in *Computer Architecture,* D. D. Gajski, V. M. Milutinovic, H. J. Siegel, and B. P. Furht, eds., IEEE Computer Society Press, Washington, D.C., 1987, pp. 387-407.

[Sto80] H. S. Stone, "Parallel computers," in *Introduction to Computer Architecture (second edition),* H. S. Stone, ed., Science Research Associates, Inc., Chicago, IL, 1980, pp. 363-425.

[SuT87] S. Y. W. Su and A. K. Thakore, "Matrix Operations on a Multicomputer System with Switchable Main Memory Modules and Dynamic Control," *IEEE Transactions on Computers,* Vol. C-36, December 1987, pp. 1467-1484.

[YoS81] M. A. Yoder, L. J. Siegel, "Systolic and SIMD Algorithms for Digital Filtering," *Proceedings of the Nineteenth Annual Allerton Conference on Communication, Control, and Computing,* University of Illinois at Urbana-Champaign, October 1981, pp. 880-889.

AI Graph Searching and Parallel N-Min-Finding

Carol Ringer

Abstract

This work focuses on ways of parallelizing a searching procedure that could be implemented on PASM. In particular it focuses on the problem of finding N minimum values in N processing elements (PE's), which is a subproblem that evolves from a larger searching problem. Given a sorted list of values in each PE memory find the N minimum values of the combined lists. A four PE SIMD version of the N-MIN-FINDING algorithm has been implemented.

The choice of an all SIMD version of the program is based on the importance of having the PE's easily synchronized to facilitate the network transfers as fast as possible. Forcing the smallest value array to have identical data in all PE's allows the sorting to be very efficient in SIMD mode. In this mode, branching overheads are incurred in the MC's and can be overlapped with the actual comparison and movement of data in the PE's.

1. Introduction

Current work being done in the area of artificial intelligence often deals with problems that have a combinatorially large problem space. Searching a large problem space exhaustively is inefficient and usually impractical. Finding ways to speed-up and optimize a searching procedure is of major importance.[LiW84]

Research has been done on ways to optimize the solution search procedure by making intelligent guesses about the best path to take [Nil80]. Another approach for speeding up the search is parallelizing it so that more than one path is explored at a time.

This paper focuses on ways of parallelizing a searching procedure that could be implemented on PASM.[Sis87] In particular it focuses on the problem of finding N minimum values in N processing elements (PE's), which is a

subproblem that evolved from the larger searching problem.

2. Background

2.1. Problem-area related references and background

Search problems can usually be represented as an acyclic graph or tree. [WaL85] One general technique for searching a graph or tree is a branch-and-bound algorithm. A branch-and-bound algorithm decomposes a problem into smaller subproblems and keeps decomposing it until a solution is found or the problem is determined to be unsolvable. The decomposition of the problem is achieved by using branching and selection rules. Elimination rules can be used to reduce the search space, and a termination rule is used to check for the goal state or solution.[LiW84]

GRAPHSEARCH is a branch-and-bound algorithm described in [Nil80] that keeps a record of the rule applications to a problem space which preserves the shortest path to the goal or subgoals. The general graph search algorithm as taken from [Nil80] is given below.

Procedure GRAPHSEARCH

1 Create a *search graph*, G, consisting solely of the start node, s. Put s on a list called OPEN.

2 Create a list called CLOSED that is initially empty.

3 LOOP: if OPEN is empty, exit with failure.

4 Select the first node on OPEN, remove it from OPEN, and put it on CLOSED. Call this node n.

5 If n is a goal node, exit successfully with the solution obtained by tracing a path along the pointers from n to s in G. (Pointers established in step 7)

6 Expand node n, generating the set, M, of its successors and install them as successors of n in G.

7 Establish a pointer to n from those members of M that were not already in G (i.e., not already on either OPEN or CLOSED). Add these members of M to OPEN. For each member of M that was already on OPEN or CLOSED, decide whether or not to redirect its pointer to n. For each member of M already on CLOSED, decide for each of its descendants in G whether or not to redirect its pointer.

8 Reorder the list OPEN, either according to some arbitrary scheme or according to heuristic merit.

9 Go to LOOP

Step 8 refers to ordering the list OPEN according to heuristic merit. The heuristic merit is some estimation of the *promise* of a node in the graph with respect to achieving the goal. A node has more *promise* if it is on the shortest path to the goal. The expansion in step 6 includes calculating this *promise* value. Nilsson claims that if a heuristic function calculated on a node n, is a lower bound on the actual cost of going from node n to the goal then GRAPHSEARCH is admissible. An admissible search algorithm always terminates in an optimal path from the start node to the goal node.

2.2. PASM Overview

Heuristic estimation is one way to optimize the GRAPHSEARCH algorithm. A way to speed-up the process is by parallelizing the node expansion process. The PASM architecture can be used to parallelize the GRAPHSEARCH algorithm by using one micro controller (MC) to keep track of the graph search path and use the PE's for expanding and sorting the nodes. A "parallelized for PASM" graphsearch algorithm is given below. This algorithm also incorporates the idea of heuristic estimation.

Procedure PARALLEL GRAPHSEARCH

1 In 1 PE create a search graph G consisting solely of the start node, s, put s on a list called OPEN

2 In 1 MC create a list called CLOSED that is initially empty

3 LOOP: Check the size of OPEN if it is 0, exit with failure.

4 IF the size of OPEN is less than the number of PE's (N)
THEN distribute those nodes to the PE's
ELSE pick the N minimum values from OPEN and distribute them among the PE's
Send and identifier for each node back to MC that will be put on CLOSED. MC then must send parent address back to the nodes that will be expanded in PE's.

5 In MC check the node identifiers sent back and see if the goal state has been reached if so exit successfully with a path to the goal.

6 Expand the nodes in the PE's and insert sort them according to heuristic merit into the OPEN lists of the PE's.

7 For simplicity assume the problem space can be represented as a tree, which implies that a node can only be generated once when its unique parent is expanded. In this case no redirecting of pointers is involved. If this is not the case each new node must be checked against all other nodes on OPEN and CLOSED to see if a shorter path is found, this could be a very cumbersome task. In this simplier case this step is inherent because each node will have as part of its record the address of the parent node in MC* sent in step 4.

8 This step is taken care of in step 6 when the new nodes are merged into the OPEN lists.

9 Go to LOOP

One major overhead in the parallel graphsearch algorithm that is not found in the serial algorithm is the cost of finding N nodes with the smallest *promise* values such that the most promising nodes will be expanded next. The distributed architecture of PASM is good for node expansion but the non-shared memory of the PE's makes finding the N-minimum *promise* values a non-trivial task. Section 3 describes how this task of "N min finding" is accomplished on PASM.

3. Project Description

3.1. Problem

The problem as described in section 2 can be summarized as follows:

> Given a sorted list of values in each PE memory find the N minimum values of the combined lists.

Since the records associated with each node could be rather large it would be impractical to try to combine the lists in some way to discover the smallest nodes. A smarter approach is to utilize the network of PASM to sort according the *promise* value such that in the end each PE knows where the smallest nodes are located and passing records is done only if necessary.

3.2. Algorithms

A high level language algorithm for N-MIN-FINDING on PASM is given below:

Assumption: Each PE has a list of values sorted from smallest to largest.

```
Procedure N-MIN-FINDING

array[PEi] <-- list_value
passreg(PEi) <-- list_value
list <-- list_next

while (count != 0)

    for j = 1 to N-1

        passreg(PEi) <-- passreg(PEi+1)
        array[(PEi+j) mod N] <-- passreg(PEi)

    bubblesort (array)

    if (list_value < array[N-1])

        then passreg(PEi) <-- list_value
        else passreg(PEi) <-- MAX

    count <-- #PE's that have smaller values

    for k = 1 to Q-1

        passreg(PEi) <-- count
```

passreg(PEi) <-- *passreg*(PEi+1)
count <-- *passreg*(PEi) + *count*

where,

N = number of PE's being used

Q = number of MC's being used

list - is the sorted list (OPEN in GRAPHSEARCH) of the node records.

list_value - is the *promise* value field of each node record.

array - is the space for the *promise* values appended with their PE number to be stored. At the end of the process it will contain the N-minimum values.

passreg(PEi) - is the register in PEi used for passing the values through the network.

count - is the number of PE's that have a smaller value on their *list* than the last value in *array*.

Notice that during the network transfer the passed values are put into the array according to their original PE number. This results in all PE's having identical arrays to sort and thus the bubblesort can be carried out in SIMD mode with the MC's controlling the looping and allowing all PE's to be enabled throughout the sort.

The count transfer loop at the end of the program is not executed in the four PE version because only one MC is used. When more PE's are used (and therefore more MC's) it only takes Q - 1 transfer and add steps to obtain the total count in all PE's. This is possible because all PE's in an MC group have the same count value and the network is configured such that PE's in one MC group are connected to PE's in another MC group.[SiS87]

3.3. Programs

The four PE version of the N-MIN-FINDING program written for PASM is appended to the end of this report. Notice that the modifications for eight and sixteen PE versions are basically changing some program constants, noted in the comments, and revising output routines.

4. Experiments Performed

4.1. Number of PE's and modes of parallelism

A four PE version of the N-MIN-FINDING algorithm has been written, debugged, and tested. The program is entirely SIMD except for the routine that sends the data back to MC for printing. This routine could be SIMD also with some modifications.

The choice of an all SIMD version of the program is based on the importance of having the PE's easily synchronized to facilitate the network transfers as fast as possible. Forcing the smallest value array to have identical data in all PE's allows the sorting to be very efficient in SIMD mode. In this mode the branching overhead is taken care of in the MC's and can be overlapped with the actual comparison and movement of data in the PE's. The fact that the data is the same in this operation implies maximum processor utilization throughout the sort.

4.2. Data set characteristics

The data sets used for testing and timing the N-MIN-FINDING program were different distributions of the N smallest values throughout the PE's. The inherent best case data for the N-MIN-FINDING algorithm, precluding the bubblesort effects, is data that is distributed with one minimum value in each PE. Data arranged in this way will cause only one iteration of the while (count $<>$ 0) loop. The inherent worst case data set is data that has all N minimum values in one PE. This data arrangement causes N iterations of the while loop.

5. Data measurements taken

Sample data and timing measurements are given in the table below. Distribution of the N minimum values is shown for each data set. All times are in milliseconds.

Data Set #	Arrangement PE0	PE1	PE2	PE3	Total Time	Transfer Time	Sort Time	Overhead Time
1	10	11	12	13	2.5	0.152	1.3	0
2	40	31	22	13	4.25	0.152	3.02	0
3	30 40	21		33	6.36	0.204	4.31	0.736
4			32	13 23 33	17.1	0.256	14.2	1.47
5	10 10 10 10				14.28	0.308	10.6	2.20
6				03 03 03 03	20.92	0.308	17.2	2.20

Table 1
Overhead is checking the PE's for smaller values and making other adjustments that must happen before the next iteration. Times in milliseconds.

Notice that there are two best case data sets (1 & 2) and two worst case data sets (5 & 6). A characteristic of the bubblesort algorithm used is that preordered data (as in set #1) takes much less time to sort than reversed data (as in set #2). To negate the effects of bubblesort the best and worst case times for N-min-finding are calculated by taking the average of the "bubblesort best and worst" for the different data distributions. A summary of the timing measurements are given in the table below.

Best Case	Ave. Case	Worst Case
3.38	10.85	17.6

Table 2
The best and worst case times are averages of data sets 1 & 2 and 5 & 6 respectively.

6. Discussion of interpretation of data

6.1. Problem related

As can be seen from the tables the execution times are very dependent on the data distribution in the PE's. The fact that the bubblesort algorithm is dependent on the ordering of the data in the array is one factor in the time discrepancy. The sorting time is between 60% and 80% of the total execution time so implementing a faster sorting algorithm might make the total time faster and not as data dependent. The fact remains that the cases where the N smallest values are in one PE will be about N times slower than the cases where the minimum values are evenly distributed. The best and worst case times in the summary table back up this theory.

6.2. PASM Architecture related

The network transfers only accounted for between 2% and 6% of the total execution time. Network overhead is data dependent only in the sense that more iterations of the loop and thus more transfers are needed when the minimum values are concentrated in one or two PE's.

The timings for eight and sixteen PE's should be proportional to the times of four PE's except for the overhead times. The overhead times in the eight and sixteen PE versions will include a recursive doubling procedure that sums the total number of PE's that have a value smaller than the last item in the sorted array. This procedure adds Q - 1 (where Q = the number of MC's being used[Sis86]) transfers per iteration of the loop. The four PE version does not need to do these transfers because the MC can examine the condition code register to see if any PE's still need to transfer a value. Another approach to determining the termination condition that may not add as much transfer overhead when more PE's are used is to look at the number of MAX values that are passed through the PE's so that each PE will know when there are no more

PE's that have a smaller value. The trade off between extra comparison steps and more network transfers would have to be examined more closely for larger groups of PE's.

7. Conclusions

The conclusions that can be drawn from the work done on N-MIN-FINDING and PARALLEL GRAPHSEARCH so far are best expressed in terms of complexity comparisons.

7.1. Comparing parallel min-finding with serial min-finding

The complexity of straight serial min-finding for n values, which is really just sorting, that assumes a faster sorting algorithm could be used (Quicksort)[Wir76] is given by:

$$\text{serial complexity} = c_1(n\log n)$$

The complexity of the N-MIN-FINDING algorithm described in section 3 for n values and N PE's, and also assuming a Quicksort algorithm can be used for all the sorting, is given by:

$$\text{parallel complexity} = c_1[(n/N)\log(n/N) + i\Theta(N\log N)] + ic_2(N + Q - 1)$$

where,

i = expected number of iterations of algorithm
c_1 = sorting complexity constant
c_2 = communications complexity constant
Q = number of MC's being used

According to the measurements shown in section 5 c_2 is small compared to c_1. If the last term is ignored in parallel expression, then the comparison really lies in sorting times of two cases. From these general expressions it seems that the parallel algorithm will be most efficient when N is small and n is large.

7.2. Comparing parallel graph search with serial graph search

In the context of graph search the overhead of N-min-finding can be counter balanced with speed gained by expanding more than one node at a time.

The serial graph search complexity is estimated by:

$$m[t + c_1\Theta(n^2)]$$

The parallel graph search complexity is estimated by:

$$(m/N)[t + c_1\Theta(n^2) + ic_2\Theta(N\log N) + ic_3(N + Q - 1)]$$

where,

m = number of nodes expanded to get to the goal
t = time to expand the average node
i = expected number of iterations of algorithm
c_1 = insert sorting complexity constant
c_2 = quick sorting complexity constant
c_3 = communications complexity constant

If the communications constant can be considered small then the trade-off between serial and parallel is based on the size of n and N and also m and t. From the estimates it appears that a parallel graph search would be most efficient if the problem space is large and the average time to expand a node is large.

8. Future work

There are many ways the work on the general searching problem using PASM can continue.

The N-MIN-finding algorithm still has other combinations of sorting algorithms and transfer procedures that could be combined. For example looking at the the trade-offs of the different ways to determine the termination condition mentioned in the last section. Also, finding a way to avoid putting the MAX values into the array so the number of items to be sorted is less.

The PARALLEL GRAPHSEARCH algorithm described in section 2 has other subproblems related to the PASM architecture that need to be solved before a parallel graph search can be tested and compared to serial versions. A method must be developed to distribute the nodes to be expanded among the PE's given that N minimum values are found and each PE knows where they are located. Different applications of graph search could be studied to discover when a parallel version will perform better than a serial version. Other searching, sorting, and min/max-finding problems could also be explored and hopefully this report will give guidance to someone interested in pursuing them.

References

[LiW84] G.Li, B.Wah, "Computational Efficiency of Parallel Approximate Branch-And-Bound Algorithms," *Proc. Int'l Conf. Parallel Processing,* 1984, pp.473-480

[Nil80] N.J.Nilsson, *Principles of Artificial Intelligence,* Morgan Kaufmann, Los Altos, CA, 1980

[SiS87] H.J.Siegel, T.Schwederski, J.T.Kuehn, N.J.Davis IV, "An Overview of the PASM Parallel Processing System," *Tutorial: Computer Architecture,* D.D.Gajski, V.M.Milutinovic, H.J.Siegel, and B.P.Furht,eds., IEEE Computer Society Press, Washington, D.C., 1987, pp.387-407.

[WaL85] B.W.Wah, G.Li, C.Yu, "Multiprocessing of Combinatorial Search Problems," *Computer,* Vol. 10, No. 6, 1985, pp93-108

[Wir76] N.Wirth, *Algorithms + Data Structures = Programs,* Prentice-Hall, Englewood Cliffs, NJ, 1976