Purdue University
Purdue e-Pubs

Department of Electrical and Computer Engineering Technical Reports

Department of Electrical and Computer Engineering

11-1-1988

Experimental Evaluation of SIMD PE-Mask Generation and Hybrid Mode Parallel Computing on Multi- Microprocessor Systems

Samuel A. Fineberg
Purdue University, fineberg@ecn.purdue.edu

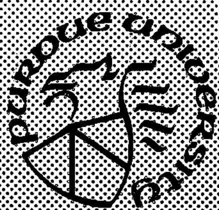
Thomas L. Casavant
Purdue University, tome@ecn.purdue.edu

Howard Jay Siegel
Purdue University, hj@ecn.purdue.edu

Follow this and additional works at: https://docs.lib.purdue.edu/ecetr

Fineberg, Samuel A.; Casavant, Thomas L.; and Siegel, Howard Jay, "Experimental Evaluation of SIMD PE-Mask Generation and Hybrid Mode Parallel Computing on Multi- Microprocessor Systems" (1988). *Department of Electrical and Computer Engineering Technical Reports.* Paper 632.
https://docs.lib.purdue.edu/ecetr/632

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.



Experimental Evaluation of SIMD PE-Mask Generation and Hybrid Mode Parallel Computing on Multi-Microprocessor Systems

Samuel A. Fineberg
Thomas L. Casavant
Howard Jay Siegel

TR-EE 88-55
November 1988

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

Experimental Evaluation of SIMD PE-Mask Generation and Hybrid Mode Parallel Computing on Multi-Microprocessor Systems

Samuel A. Fineberg
(317) 494-0705
fineberg @ ecn.purdue.edu

Thomas L. Casavant
(317) 494-9143
tomc @ ecn.purdue.edu

Howard Jay Siegel
(317) 494-3444
hj @ ecn.purdue.edu

Parallel Processing Laboratory
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907

November 1988

Abstract

Experimentation aimed at determining the potential efficiency of multi-microprocessor designs of SIMD machines 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 bitonic sorting, which has feasible solutions in both SIMD and MIMD modes of computation, as well as in at least two hybrids of SIMD and MIMD modes. Bitonic sorting was coded in these four ways and experiments were performed that examine the tradeoffs among all of these modes. Also, a new PE mask generation scheme for multiple "of-the-shelf" microprocessor based SIMD systems is proposed, and its performance was measured.

Keywords: Parallel Processing, Performance Evaluation, PASM, Computer Architecture, Hybrid SIMD/MIMD Computing

This work supported by the National Science Foundation under Grant Numbers CCR-8704826, CCR-8809600, by the NSF Software Engineering Research Center (SERC), and by the Air Force Office of Scientific Research under Grant Number F49620-86-K-0006.

1. Introduction

While extensive past efforts have dealt with design and analysis of SIMD and MIMD algorithms, computations, and machines, this work describes empirically-based architecture research generated from experiments on a parallel machine. This research was performed in an attempt to gain insight into the dynamic characteristics of novel parallel architectures. Specifically, the performance of the PASM prototype, a machine capable of both SIMD and MIMD modes of computation, is evaluated from the perspective of bitonic sorting [Bat68]. This application was chosen because it has a simple enough structure to permit analysis of architecture features through controlled measurements of program execution time. The experiments described are based on pure SIMD, pure MIMD, and two distinct hybrid SIMD/MIMD algorithms for bitonic merging (sorting) of N distinct elements using P processors, $N > P$, when each processor is initially associated with N/P-elements. This is done for values of N ranging from 16 to 512, and for P equal to 4, 8, and 16.

The primary architectural design feature being evaluated in this work is the dynamic evaluation of processing element (PE) condition codes (held in the processor status register) and subsequent generation of PE masks for the enabling and disabling of PEs in multi-microprocessor based SIMD designs. (We define a multi-microprocessor based system as a system utilizing multiple "off-the-shelf" microprocessors for its PEs). In such systems, a minimum and non-zero amount of overhead is required for the control unit (CU) to determine condition codes of PEs to subsequently generate PE masks.

This work measures the actual impact of this feature for the 16-processor PASM prototype constructed at Purdue. We show that the current PE mask generation design can be enhanced, using a more complex design, so that system performance is significantly improved. Evaluation of the performance of the enhanced design is an extrapolation of actual system measurements.

We also show the performance of bitonic merging for both basic modes of PASM and two hybrid versions — one *(S/MIMD)* that uses MIMD mode to evaluate all conditionals and execute conditional code and SIMD mode otherwise, and a second *(BMIMD)* that operates in MIMD mode but uses PASM's SIMD hardware to perform a barrier synchronization just prior to a network transfer. The results show that the S/MIMD version has the best performance, the BMIMD is next, followed by the pure MIMD and SIMD, in that order, thus demonstrating the advantage of a hybrid mode architecture for this application.

Section 2 briefly describes 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 and to generate the raw data presented in Section 6. In Sections 7 and 8, the empirical results are discussed under special consideration of the PASM architecture and the effects of condition code evaluation.

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 performance analysis studies include those for the SM3 system and a hypercube architecture [SuT87], a CRAY-1s [SrA83], an optical architecture [LoH88], and the VMP multiprocessor [ChG88]. Experimental work involving measurements on working machines includes the BBN Butterfly [CrG85], Cm* [GeS87], the Encore Multimax [Hud88], the Intel Hypercube [Hud88], PASM [CaS88, FiC88], the CM-1 and CM-2 [ZeL88], a 1024 processor NCUBE [GuM88], the Warp system [AnA87], an Alliant FX/8 [Han88, JaM86], a CRAY XMP [Cal84], and a combination of Apollo workstations and an Alliant FX/8 [KuN88]. The work presented here adds to this body of knowledge by experimentally evaluating the architecture of a multi-microprocessor based SIMD/MIMD system.

3. Overview of PASM and the PASM Prototype

The PASM (partitionable SIMD/MIMD) system is a dynamically reconfigurable architecture in that 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 $P = 2^p$ PEs (numbered from 0 to $P-1$), 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 P/Q PEs. PASM has been designed for P=1024 and Q=32 (P=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.

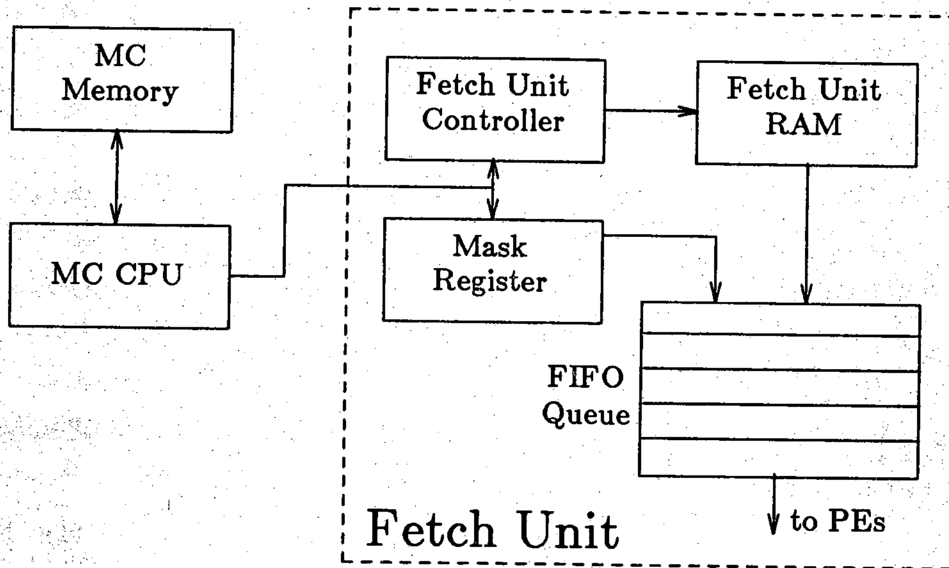


Figure 1: Simplified MC structure.

The PASM prototype system was built for P=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 [Sie85]. Because knowledge about the MC and the way in which SIMD mode is 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 that 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.

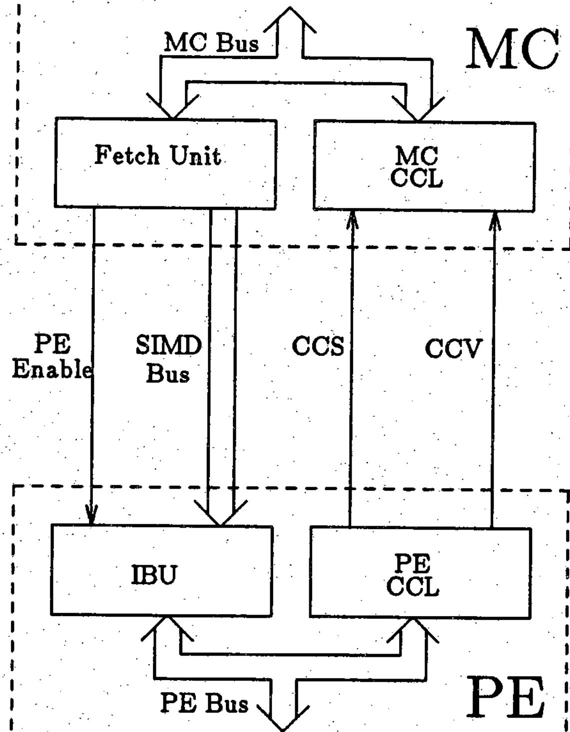


Figure 2: MC-PE connections.

PEs execute SIMD instructions by performing an instruction fetch from a reserved memory area called the *SIMD instruction space*. This is handled by the PEs' *Instruction Broadcast Unit (IBU)*, shown in Figure 2. Whenever the IBU 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 JMP instruction to

the reserved memory space, and a switch from SIMD to MIMD mode is performed by sending a JMP 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, DiS88] 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 that 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 data from SIMD space (the PEs are always following their MIMD program counters, i.e., the computation mode remains MIMD). 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.

The circuit-switched PASM interconnection network operates as follows, the sending PE establishes a path through the network 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. Word (16-bit) data values may now be written to the DTR and automatically sent through the network. The receiving PE reads the transferred word 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.

In SIMD mode, the MCs can perform data conditional operations that are dependent on PE results. To support this, *Condition Code Logic (CCL)* [ScN87] is provided. This permits the MCs to request *condition code (CC)* information from the PEs in order to generate SIMD masks based on the results of operations performed in the PEs. This is described in Section 7.2.

4. Bitonic Sorting Algorithm for SIMD Machines

The model of SIMD machines used as a framework for the description of the bitonic sorting algorithm in this section was introduced in [Sie79]. There are $P = 2^p$ *processing elements (PEs)*, numbered (addressed) from 0 to P−1. Each PE includes a processor, memory, a data transfer register (*DTR*), and a register containing the PE's address (*ADDR*). ADDR(i) denotes the i-th bit of ADDR. An *interconnection network* is a set of *interconnection functions*, each a bijection on the set of PE addresses. When an interconnection function f is executed the contents of the DTR of PE i are copied into the DTR of PE f(i), for all i, $0 \leq i < P$, simultaneously. A Cube network will be defined using $b_{p-1} \ldots b_1 b_0$ as the binary representation of an arbitrary PE address and $\overline{b_i}$ as the complement of b_i. The *Cube* network consists of p functions:

$$\text{cube}_i(b_{b-1} \ldots b_1 b_0) = b_{p-1} \ldots b_{i+1} \overline{b_i} b_{i-1} \ldots b_0,$$

for $0 \leq i < p$.

Batcher's bitonic sorting method is described in [Bat68, Knu73, Qui87, Sto71]. The SIMD version presented here is from [Sie78]. A *bitonic sequence* is (a) a sequence of numbers $x_0, x_1, \ldots, x_{r-1}$, such that $x_0 \leq \cdots x_{k-1} \leq x_k \geq x_{k+1} \geq \cdots x_{r-1}$, for some k, $0 \leq k < r$, or (b) a cyclic rotation of such a sequence. Assume the data in the DTRs of the P PEs form a bitonic sequence when listed in order (from PE 0 to PE P−1). Let A denote an internal CPU register in each PE. Then to sort the DTR data all PEs execute Algorithm 1.

```
for i = p−1 step −1 until 0 do
    A ← DTR
    cube_i
    if ADDR(i) = 0
        then DTR ← min(DTR,A)
        else DTR ← max(DTR,A)
```

Algorithm 1: Bitonic sequence sorter.

Each PE saves the value of its DTR in A. Then, PEs whose addresses differ only in the i-th bit position exchange data. The DTR data received is then compared with the old DTR data saved in A. The minimum of each data pair is moved to the PE with a 0 in the i-th position and the maximum to the PE with a 1 in the i-th position. All PEs whose i-th address bit is 0 compute the P/2 minimum in parallel, and then all PEs whose i-th address bit is 1 compute the P/2 maximums in parallel. The

interconnection network is used to implement the $cube_i$ function to transfer the DTR data between different PEs.

To sort arbitrary sequences, as opposed to bitonic sequences, p stages of bitonic sorters are needed, numbered from 1 to p. The i-th stage acts as 2^{p-i} 2^i-element bitonic sequence sorters. Half of these are ascending sorters, producing ascending sequences, as described above. The other half are descending sorters, producing descending sequences by switching the "min" and "max" functions in the above algorithm. By alternating the ascending and descending bitonic sequence sorters at stage i, $2^{p-(i+1)}$ bitonic sequences of length 2^{i+1} are formed. These bitonic sequences are sorted by stage i+1. An arbitrary sequence sorter is constructed by starting with P/2 alternating ascending and descending 2-element bitonic sorters, then P/4 alternating ascending and descending 4-element bitonic sorters, etc., ending with a single P-element bitonic sorter (see [Knu73]). Thus, the following instructions, executed by all enabled PEs in the SIMD machine, sort the data in the DTRs of the PEs:

```
for j = 1 step +1 until p do
    if j=p or ADDR(j)=0 then type ← 0 else type ← 1
    for i = j−1 step −1 until 0 do
        A ← DTR
        cube_i
        if type=0
            then if ADDR(i)=0 then DTR ← min(DTR,A)
                              else DTR ← max (DTR,A)
            else if ADDR(i)=0 then DTR ← max(DTR,A)
                              else DTR ← min(DTR,A)
```

Algorithm 2: Arbitrary sequence sorter.

The conditional "if j=p or ADDR(j) = 0" determines which PEs will act as ascending comparators and which will act as descending comparators. The variable "type" specifies whether a comparison should produce an ascending or descending sequence. The number of interprocessor data transfers used is p(p+1)/2. The asymptotic time complexity of the entire algorithm is $O(p^2)$.

In the following sections, a variation on the bitonic sorting algorithm that sorts lists is implemented. It is assumed that the number of items to be sorted is $N = 2^n$, and each PE initially contains a sorted list of N/P data items. Algorithm 2 is modified in the following way to handle lists. Consider an instance of "X ← min(X,Y)" or "X ← max(X,Y)." X and Y are now N/P element lists. An ordered

merge of lists X and Y is performed. The N/P smallest elements are the value of the "min," while the N/P largest elements are the value of the "max."

The modified "i" loop from Algorithm 2 is shown in Algorithm 3. In each PE, the portion of the list to be sorted is held in X, a one-dimensional array of N/P elements. For the "q" loop, each PE α receives a copy of the X array of PE $cube_i$(alpha), and stores it in Y. Then, in each PE, merge(X,Y) merges the sorted lists X and Y into the sorted list $X \cup Y$ and places the lesser half in X and the greater half in Y. Depending on each PE's value of "type" and "ADDR," swap(X,Y) swaps the two lists (pointers). This is analogous to establishing ascending and descending sequences (with the elements within each list X and Y always in ascending order). Each PE now has a new X list and deletes its Y list.

```
for i = j-1 step -1 until 0 do
    for q = 1 step +1 until N/P do
        DTR = X[q]
        cube_i
        Y[q] = DTR
    merge (X,Y)
    if (type⊕ ADDR(i)) = 1 then swap(X,Y)
```

Algorithm 3: Inner loop for sorting lists.

5. Algorithm Versions

Programs implementing the above described sorting algorithm were coded in each of PASM's basic modes, SIMD and MIMD. The algorithm was also coded in a barrier synchronized version (BMIMD) and an SIMD/MIMD hybrid (S/MIMD) version. In all of these versions, each PE contains a list of N/P data elements, each of which is an (N/P)-element subset of N unsigned uniformly distributed random 8-bit integers. These subsets were sorted in advance and placed in each PE in ascending order. At completion of the algorithm, all lists are merged such that the data will be in ascending order within and across PEs, (i.e. if i is in PE x and j is in PE x+1 then $i \leq j$). All calculations were made as early as possible so that no invariants are recomputed. This included the calculation of the "type" at each point in the algorithm and the masks for the "swap" routine in SIMD mode for each iteration.

5.1. MIMD

In the MIMD version, all instructions were executed in the PEs and were fetched from the PEs' own memory module. All data were contained in the PEs' memory module. MIMD mode is a natural choice over SIMD for this algorithm because, while programs in each PE are the same, there are many *if-then-else* constructs in this code, especially in the merge routine. When these are executed in SIMD mode, the *then* must finish on the PEs for which the *if* condition was true while the PE's for which it was false remain idle. Then, this latter set of PEs will execute the *else* statement while the former group of PEs remains idle. This adds more serialization than is necessary in MIMD mode. In MIMD mode, PEs execute the data conditionals independently and therefore are never forced to be idle because of such a statement. PP This is because the network in PASM appears as an I/O device and the PEs' CPUs must move data into and out of the network explicitly. Therefore, a PE must wait to send its data until the PE it is sending to is ready to receive, and vice versa. The PE cannot proceed to the merge portion of the algorithm while waiting to send or receive data because the merge is dependent on data from this transfer. Thus, it must wait until the network operation can take place, hence requiring the use of blocking reads and writes. These, required software polling by the PEs and added substantial overhead to the MIMD version.

5.2. SIMD

In the SIMD version, a PE received all instructions from its MC's Fetch Unit through a FIFO queue, as described in Section 3. All data stored in the PEs were held in their memory module. Looping and control flow instructions were executed on the MCs while data manipulation and effective address calculation instructions were executed on the PEs. Also, PE masks were needed to enable and disable PEs. The masks were generated based on either the PE number or on data contained in their condition code status register. The masks based on PE number were precalculated because they were static for a given problem size and number of PEs being used. The data conditional masks were contained in the merge portion of the algorithm and utilized the CCL to determine the proper masks.

SIMD mode was the slowest version of the algorithm for several reasons. First, as stated previously there was added serialization due to the inability of the PEs to overlap the *then* and *else* statements of the *if-then-else* constructs. Because most of these were contained in the merge routine, this serialization increased as N increased (for fixed P). Another problem with SIMD mode was the current design of the CCL.

This design is inefficient due to the handshake required between the MC and its PEs.

An advantage, however, of SIMD mode was the ability to do very quick network transfers. In PASM, the network appears as two memory-mapped I/O registers, one for sending and one for receiving data. Because all PEs are in lock-step, there is no need to poll the network input buffer before a transfer. This eliminates a substantial overhead that was incurred in the MIMD version. In the MIMD version, even if the network is ready to accept data, the PE must first check the network input buffer, which requires a memory-to-register move, a logical AND, and a compare instruction. If the previous transfer has not yet completed, the PEs must repeat this test until it has. Finally, when the buffer is ready a memory-to-memory move instruction is needed to perform the transfer. For SIMD mode, only the final memory-to-memory move instruction is needed.

In addition, SIMD mode has the ability to overlap MC control flow instructions with PE computations. This is because of the FIFO queue in each MC's Fetch Unit. The MCs send blocks of instructions to the PEs and these blocks are enqueued by the Fetch Unit for execution on the PEs. Because of this enqueuing, the MC is free to execute control flow instructions while the PEs execute the enqueued instructions. This, however, depends on the FIFO remaining non-empty. Unfortunately, when the CCL is used it requires the FIFO to empty at least once, because the new mask value must be determined using the CCL before new instructions can be enqueued. When this occurs, the overlap effect is eliminated. This made the overlap benefit less evident in the portions of the algorithm where data dependent masks are generated, i.e., the merge routine. However, because masks are also enqueued, when the masks based on PE address (not PE data) can be pre-computed they do not prevent overlap, i.e. the masks controlling the "swap" routine.

5.3. BMIMD

In order to reduce the network overhead in the MIMD version, a BMIMD version of the algorithm was developed. This version was identical to the MIMD version except that before each network transfer occurred the PEs were barrier synchronized utilizing the SIMD instruction space, as described in section 3. The Fetch Unit is mapped into a region of PE memory allowing the PEs to synchronize by reading data from this region. This synchronization consists of a single memory move instruction and allows the network registers to written to be written to and read without polling (as in SIMD mode). This provided a substantial reduction in execution time over the MIMD version by greatly reducing network overhead.

5.4. S/MIMD

While the BMIMD version did reduce the overhead required for network transfers, it did not eliminate it as SIMD mode did (i.e., it had to do the SIMD memory space read to synchronize). Also, the benefit of the control flow overlap was not possible in the BMIMD version. In order to examine this effect, a fourth version was created which transferred complete control back and forth between SIMD and MIMD modes to optimize the algorithm with respect to the best possible mode for each segment of the algorithm. The "merge" and "swap" routines were executed in MIMD mode. All other operations were done in SIMD mode. This permitted the "i," "j," and "q" loops to be controlled by the MC, thus allowing this control flow to be overlapped. It also let all network transfers be done in SIMD mode. The only added overhead was in the movement between SIMD and MIMD modes. These mode switches were performed with JMP instructions.

6. Data Measurements

The algorithm versions were coded in MC68000 assembly language, executed on the PASM prototype, and timings were made using PASM's internal timers (MC68230). The execution time of the four versions were measured for $N = 32, 64, 128, 256$, and 512 for $P = 4, 8$, and 16 as well as $N = 16$ for $P = 4$ and 8. These measurements are shown in Figures 3 through 7. These Figures utilize a log scale (vertical axis) for time and a linear scale (horizontal axis) for problem size, N, and are each for a fixed size group of PEs, P. Also, an SIMD version that did not test condition codes was timed in order to determine a lower bound for SIMD execution time, i.e., it was intended to simulate zero-cost mask generation. This version did not produce valid output and ignored condition codes by always enabling all PEs. More details about this last version will be provided later.

7. Discussion

7.1. Overall Comparison

As can be seen from Figures 3, 4, and 5, the fastest version was clearly the S/MIMD version, with the execution time of the BMIMD version sligtly greater at all points. After these two "hybrid" versions the MIMD version was next, with the SIMD version being the slowest. The reasons for these rankings can be explained by the factors mentioned earlier.

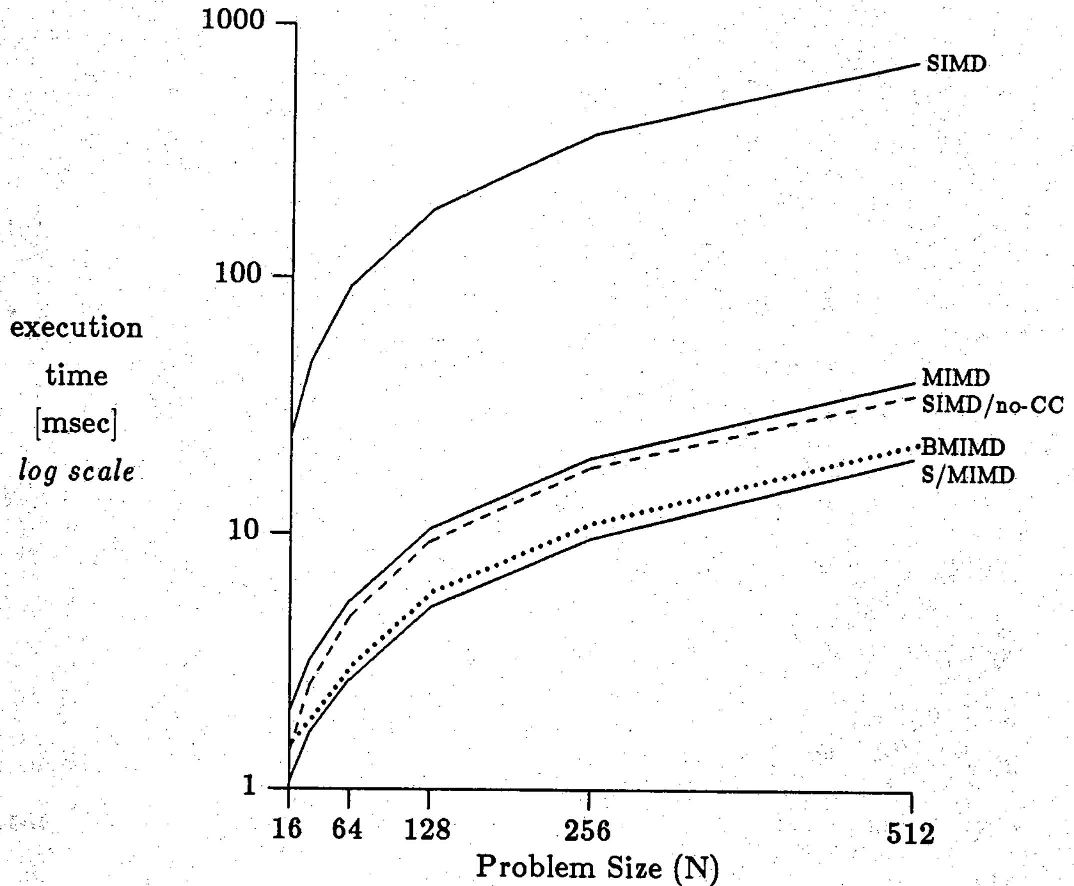


Figure 3: Execution time vs. problem size for P=4.

MIMD vs. BMIMD

First, consider the MIMD and BMIMD versions. These were the most similarly implemented of the four versions. As can be seen from the graph, their execution times start out close and as N increases, the difference between the execution times of MIMD and the BMIMD versions increases. Recall that the use of barrier synchronization is the only difference between the MIMD and BMIMD versions. The number of network transfers is $\left(\frac{N}{P}\right)p(p+1)/2$, so as N increases the advantage gained by the more efficient network transfer in BMIMD mode also grows.

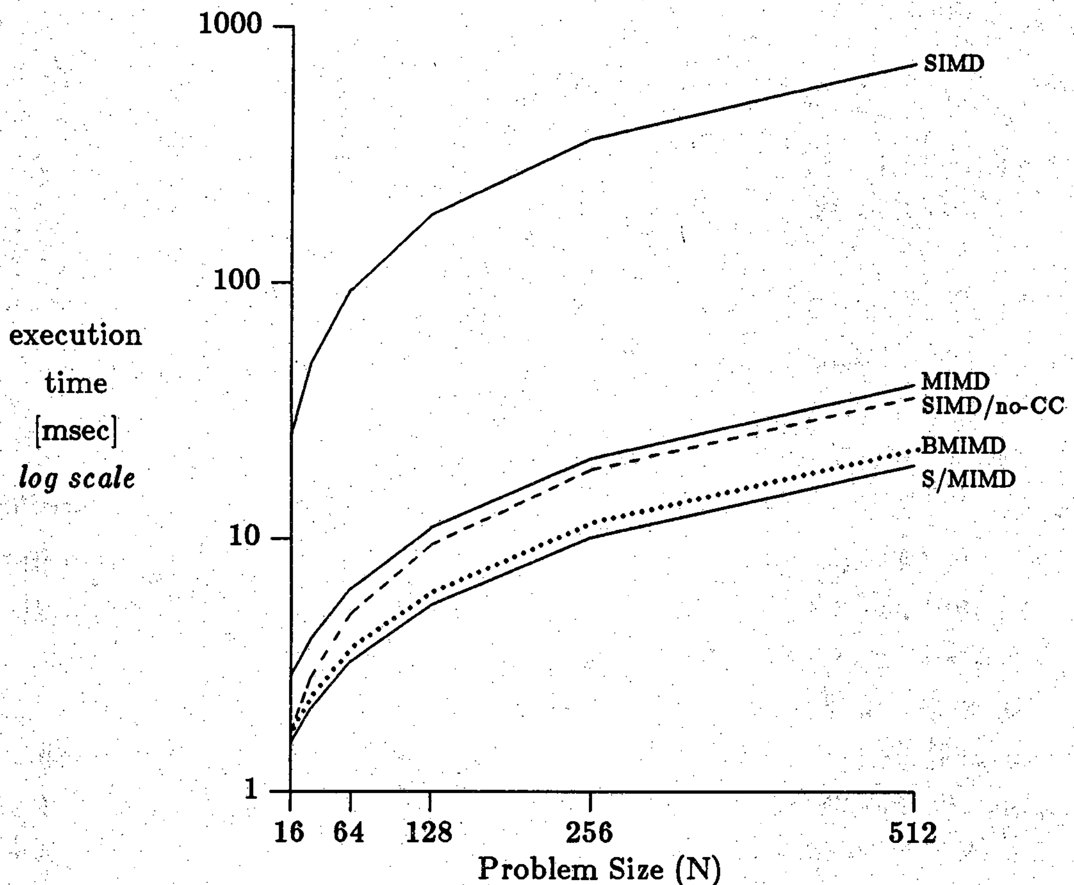


Figure 4: Execution time vs. problem size for P=8.

S/MIMD Comparison

The S/MIMD version was faster than the other versions and improved in relation to the MIMD and BMIMD versions as N increased. (Note that a constant distance between lines represents an increasing difference on this log scale). Its speed can be attributed to having almost no overhead necessary for network transfers, and the added advantage of the control flow overlap when in SIMD mode. In fact, the only overhead present in this version but not present in the others was that of explicitly transfering control between SIMD and MIMD modes. Notice that as N increases the advantage of S/MIMD over MIMD and BMIMD improves due to the $O(\frac{N}{P}\log_2{}^2P)$ complexity of the "q" loop which benefits from SIMD control flow overlap and SIMD network transfers. Also, the mode changing overhead of S/MIMD mode is outside of the "q" loop and occurs p(p+1)/2 times. In BMIMD mode, the

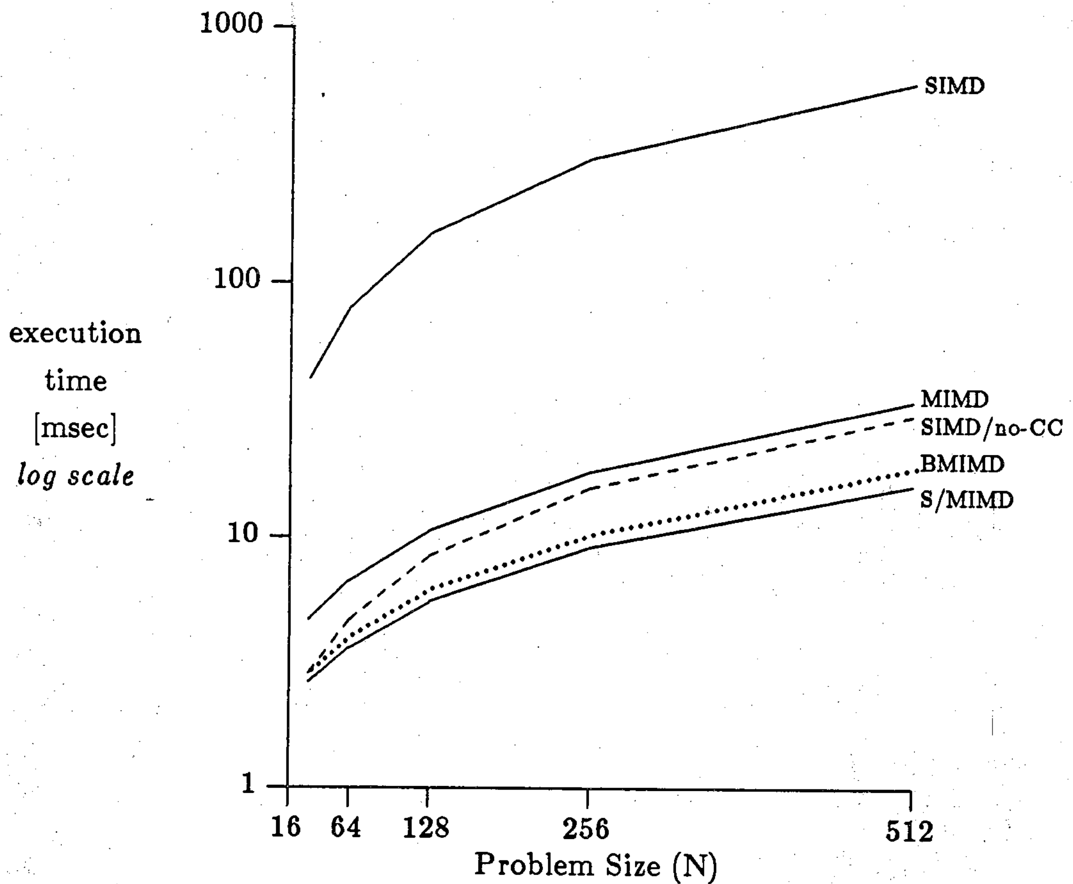


Figure 5: Execution time vs. problem size for P=16.

barrier synchronization is within the "q" loop, and therefore occurs $\left(\frac{N}{P}\right)p(p+1)/2$ times.

SIMD Comparison

There is a large difference evident between the SIMD version and the other three versions. This can most readily be attributed to the design of the CCL. Note that the use of the CCL occurs within the $O(\frac{N}{P}\log_2{}^2P)$ complexity merge routine that was the source of both the bulk of the conditional instructions and the *if-then-else* serialization.

7.2. Effects of the Condition Code Logic

Consider the execution times of the SIMD version both with and without conditional mask generation (Figures 3 to 6). While the shapes of these two curves are similar, there was a large difference between the performance of these versions. As stated earlier, this difference is due to the overhead caused by PE mask generation. The SIMD version represents the current design of PASM's CCL, while the SIMD version with no conditional mask generation represents a lower bound on execution time if conditional masks could be generated with zero cost.

One of the main goals of the PASM architecture was to create a system capable of operating as an SIMD machine from "off the shelf" processors. This approach had several obstacles, one of which was the problem of determining the status of the PEs in order to generate conditional PE masks. Typical microprocessors do not have the contents of their internal status register available on external pins. In an SIMD machine utilizing *custom processor* design, there could obviously be a global status register that can be accessed directly by the CU and set directly by the PEs whenever an operation is performed that affects the processor status. This situation is approximated by the SIMD version without conditional mask generation *(SIMD/no-CC)* because the setting of conditional bits in a custom processor design would be passive and would not require any added overhead. However, this was not possible in the multi-microprocessor PASM architecture, so it became necessary to design external hardware capable of combining status information given to it by the PEs and making the result available to the MCs. This operation is performed on PASM by the CCL briefly described in Section 3. Consider how an MC receives conditional information from its PEs (see Figure 2).

When an MC wants to test a status flag on its PEs it must:

(1) send an instruction to the PEs to clear their *Condition Code Synchronization (CCS)* register;

(2) wait for all of its PEs to finish this operation by waiting for its CCS register to clear;

(3) send instructions to its PEs to set their *Condition Code Value (CCV)* registers to indicate the value of the status flag in question;

(4) instruct PEs to set their CCS lines to indicate that the value in their CCV is valid;

(5) poll its CCS register until all PEs have set their CCS registers.

The MC can then read the status information from its CCV register and generate the appropriate mask. Hardware is also provided for combining condition codes from different MCs when necessary. For the results presented here, this process itself (with 8 MHz 68000 technology) takes about 56μsec, thus adding a considerable overhead to the mask generation.

Another problem with generating masks on PASM was that each time a mask was generated, the Fetch Unit Queue was emptied. This, unfortunately, is necessary on any architecture that has an SIMD instruction pipeline (queue) with masks associated with instructions in such a way that the masks cannot be changed once an instruction has been enqueued by the CU. This is analogous to the problem of a pipeline emptying due to conditional branches in a typical pipeline processor. This problem was independent of the design of the CCL and requires a major change in the Fetch Unit design to be overcome. This pipeline flushing is necessary because the MC is unable to send instructions into the Fetch Unit Queue until their masks are known. Furthermore, the MC must wait until it knows the status of the PEs to send this instruction, thus making the PEs stand idle while this mask is being generated. Normally, instructions executed by the MC may be overlapped with the execution of PE instructions. This added MC/PE serialization also hurts the performance of the SIMD version; however, its effect is hard to quantify because it is dependent on programming style, e.g. pre-testing of condition codes. This effect is further exacerbated by the current CCL design because the time required for this mask generation is longer than necessary.

7.3. SIMD (without conditional masks) vs. S/MIMD and BMIMD

In this section, the SIMD/no-CC version that has no conditional mask generation will be compared with the two fastest versions in order to get a better sense of the differences between SIMD/no-CC and MIMD modes. As is evident from Figures 3 through 5, the SIMD/no-CC version is relatively close to the others for small N, but its execution time increases sharply as N increases (recall execution time is on a log scale). In the S/MIMD and BMIMD versions, their execution increases smoothly and with a much smaller slope than the SIMD/no-CC version. This was mainly caused by the added serialization in the merge routine due to the three *if-then-else* else statements it contains. One is to check which list has the smallest element, one is to check if the "A" list has reached the end, and the third is to check if the "B" list has reached the end. Because the merge is executed p(p+1)/2 times, and for each merge execution each of the *if-then-else* statements are executed up to N/P times,

the number of iterations of this loop will increase as O(N) for P held fixed. Hence, the serialization added due to the *if-then-else* statements will also increase as O(N). While the number of these constructs increases similarly in the MIMD design of the merge routine that was implemented in the BMIMD and S/MIMD versions, they did not add additional overhead because the *then* and *else* segments could be overlapped across PEs.

Another factor to consider was the efficiency of the network transfers in the SIMD/no-CC and S/MIMD versions. While the BMIMD version has very low overhead transfers, there is still a finite overhead required for each network operation over the necessary memory to DTR move instruction. Because there must be (N/P)p(p+1)/2 network transfers, the network synchronization overhead for BMIMD mode will increase O(N) while overhead for the S/MIMD version remains constant. The SIMD/no-CC version has no extra network overhead, but due to the other factors mentioned it is not faster than the BMIMD and S/MIMD versions. This effect becomes evident though only for large N, and this, along with the control flow overlap, makes S/MIMD the fastest version.

8. A Possible Modification to the PASM CCL

The current design of PASM's CCL has been shown to be somewhat inefficient. Given that fact, and a lower bound on execution time has been measured, we can determine how one might improve this inherent problem in the design of multi-microprocessor based SIMD/MIMD computers. First, one must consider what is absolutely necessary in order to send status information from the PEs to their CU. In any multi-microprocessor SIMD machine, the PEs must explicitly write their status information to a device that can combine this information for their CUs and then the CU must read this and generate a mask. Disregarding the time required to generate the mask from the conditional information, this operation requires a minimum of one register-to-memory write performed by the PEs and a memory-to-register read by the MC (using memory-mapped I/O). One possible method of implementing this would be to utilize the CU's Bus (see Figure 2) handshake to replace the CCS register. This would eliminate the need to do software polling to determine when the CCV is valid. After the MC reads this register the data would automatically become invalid, thus eliminating the need for PEs to invalidate their CCV's. This could be done by having the MC send an instruction to the PEs to send their status registers to a memory mapped register which would determine whether the condition to be tested was true or not, and then read from this new CCV

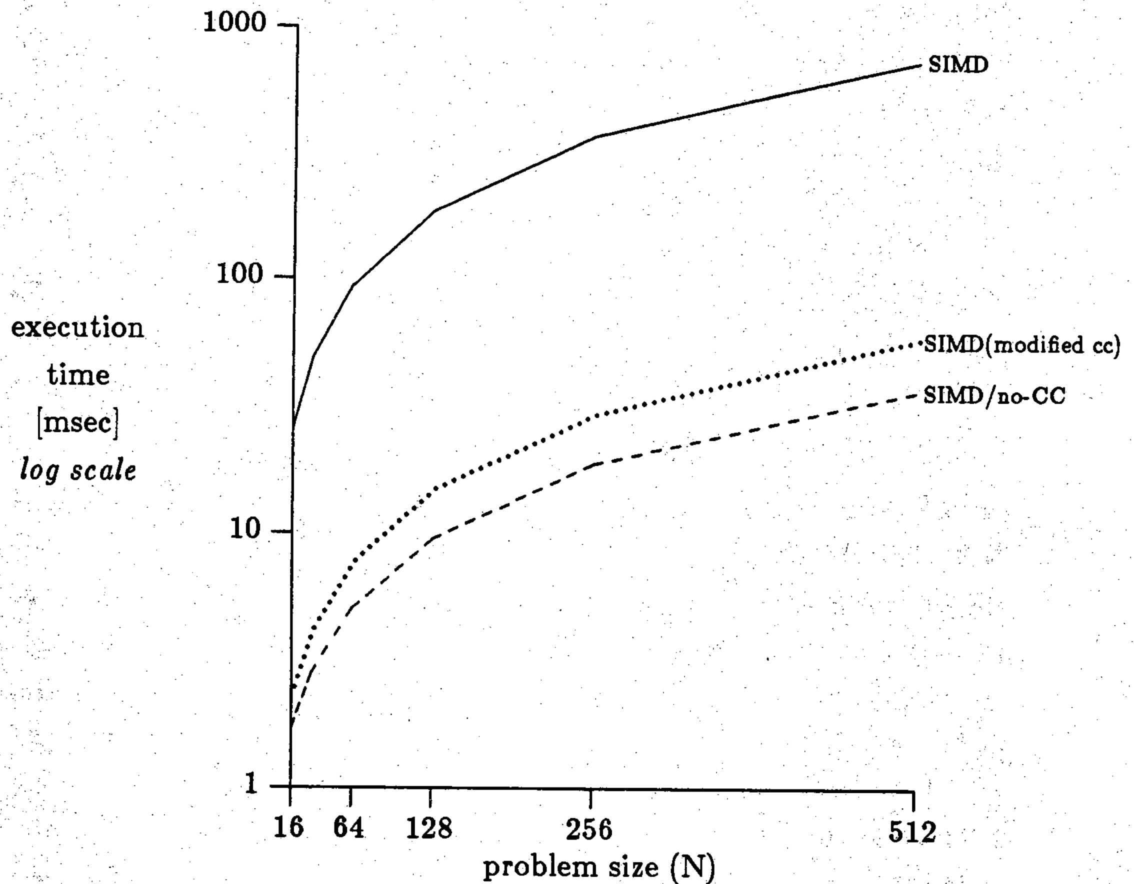


Figure 6: Execution time vs. problem size for P=16.

register. This new hardware will then combine the PEs' CCVs and generate a CC vector, place the vector in the MC's CCV, and set the Bus handshake line to allow the memory read initiated earlier by the MC to complete. While this would require a Bus timeout to prevent deadlock, it would greatly reduce the overhead in reading status information by allowing hardware polling of the CCS and eliminating the need for the PEs to explicitly poll their CCS's. A version which was intended to simulate this was programmed and executed on PASM. This version assumed that the worst case time to combine the CCV registers was no longer than PASM's memory module access time ($\sim$ 600 ns). This was accomplished by simply adding a register-to-memory move on the PEs and a memory-to-register move on the MCs wherever a condition should be tested. (It did not, however, produce valid sorted output because the conditions were never actually tested.) This version is plotted along with the SIMD/no-CC and actual SIMD time for P=8 in Figure 6.

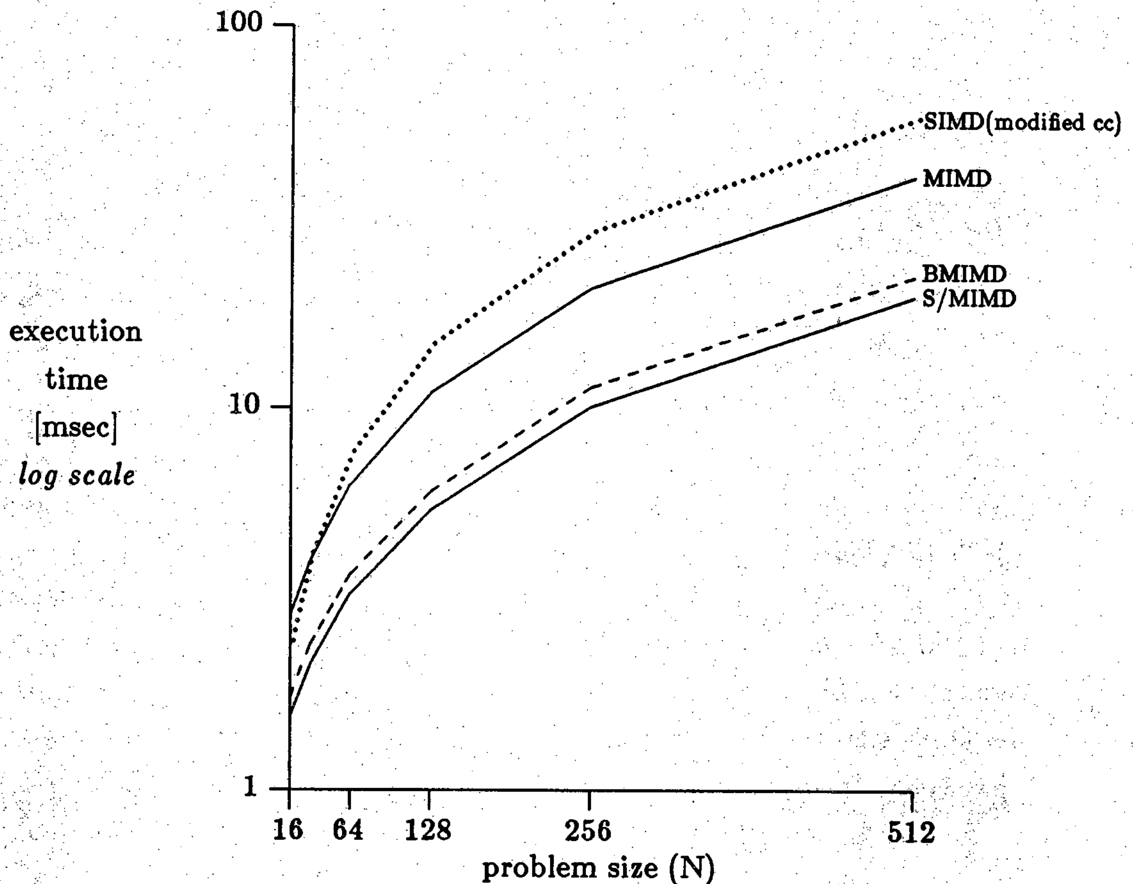


Figure 7: Execution time vs. problem size for P=8.

This graph demonstrates two things. First, it is possible to implement the condition code logic in a way that substantially reduces overhead over the current design. This new version was substantially faster than the SIMD version with the current design and was faster than the MIMD version for small values of N as seen in Figure 7. However, there is still considerable overhead required for conditional mask generation and this overhead increases with N, thus keeping the relationship the same with respect to the hybrid versions. Unfortunately, this relatively minimal overhead is virtually impossible to avoid in a multi-microprocessor system made from "off-the-shelf" components and is one of the major drawbacks of using "of-the-shelf" components in an SIMD machine. However, this can be justified by the cost considerations of designing a custom processor alone. Also, the use of standard components simplifies the design of the PEs and enables them to run efficiently in MIMD mode. Consider the hardware necessary to combine condition codes with custom processors.

Lines would have to be run from each processor to its MC to indicate its status and several levels of logic would be necessary to combine this status information. These would hinder PE performance because the conditional information would be sent to the MC for every operation involving status flags. Therefore, in MIMD mode there would be added delay for evaluating conditionals even though the MC will not need this state information. Also, in SIMD mode there would be overhead every time the PEs set their status register. The status register is not only set when testing conditionals, but also on most arithmetic and logical operations. Therefore, a certain amount of added overhead would be added for a custom processor design, and this overhead would exist for all instructions that set processor status bits whether needed or not and this delay would limit processor cycle time.

9. Conclusion

Experiments designed to examine the tradeoffs among the SIMD, MIMD, BMIMD, and S/MIMD modes on the PASM parallel processing system prototype were described. In particular, the effects of SIMD PE-mask generation and hybrid mode computing were considered. Experiments consisted of measurements of the execution time of a bitonic sorting algorithm. This algorithm has feasible implementations in each of these modes, and further, exercises PE mask generation, as well as reconfigurable network, aspects of PASM. Execution times for different size lists, numbers of processors, and modes of parallelism were collected. This data was evaluated and discussed, examining the effects of the various parameters in the tests.

An upper bound on the performance of PASM's PE mask generation was measured. Based on this experience, a change in the design of PASM's mask generation logic was proposed that brought PASM's performance closer to this upper bound and experiments were performed to quantitatively measure the effect of this change.

It was also shown that PASM can utilize its mode switching and hardware barrier synchronization abilities to improve its performance over both pure SIMD and MIMD modes. These abilities were exploited in the S/MIMD and BMIMD versions. Through the use of these techniques, the advantages of both the SIMD and MIMD modes on PASM were exploited, and demonstrate the advantage of a machine with this mode-switching capability.

Acknowledgements: The authors of this paper acknowledge many useful discussions with Ed Bronson, Wayne Nation, Pierre Pero, Tom Schwederski, and the other members of the Parallel Processing Laboratory Users Group (PPLUG).

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.

[Bat68] K. E. Batcher, "Sorting networks and their applications," *AFIPS 1968 Spring Joint Computer Conference,* 1968, pp. 307-314.

[CaS88] 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, *Experimental Benchmarks and Initial Evaluation of the Performance of the PASM System Prototype,* TR-EE 88-2, Purdue University, 1988.

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

[ChG88] D. R. Cheriton, A. Gupta, P. D. Boyle, and H. A. Goosen, "The VMP multiprocessor: initial experience, refinements, and performance evaluation," *International Symposium on Computer Architecture,* May 1988, pp. 410-421.

[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.

[DiS88] H. G. Dietz and T. Schwederski, *Extending Static Synchronization Beyond SIMD and VLIW,* TR-EE 88-25, Purdue University, 1988.

[FiC88] S. A. Fineberg, T. L. Casavant, T. Schwederski, and H. J. Siegel, "Nondeterministic instruction time experiments on the PASM system prototype," *1988 International Conference on Parallel Processing,* August 1988, pp. 444-451.

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

[GuM88] J. L. Gustafson, G. R. Montry, and R. E. Benner, "Development of parallel methods for a 1024-processor hypercube," *SIAM Journal on Scientific and Statistical Computing,* Vol. 9, July 1988, pp. xx

[Han88] F. B. Hanson, "Vector multiprocessor implementation for computational stochastic dynamic programming," *IEEE Technical Committee on Distributed Processing Newsletter,* Vol. 10, 1988, pp. 44-49.

[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.

[Knu73] D. E. Knuth, *The Art of Computer Programming: Vol. 3 Sorting and Searching,* Addison-Wesley, Reading, MA, 1973.

[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, pp. 35-43.

[LoH88] A. Louri and K. Hwang, "A bit-plane architecture for optical computing with two-dimensional symbolic substitution," *International Symposium on Computer Architecture,* May 1988, pp. 18-27.

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

[Qui87] M. J. Quinn, *Designing Efficient Algorithms for Parallel Computers,* McGraw-Hill, New York, 1987.

[ScN87] T. Schwederski, W. G. Nation, H. J. Siegel, and D. G. Meyer, "Design and implementation of the PASM prototype control hierarchy," *Second International Supercomputing Conference,* May 1987, pp. 418-427.

[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.

[Sie78] H. J. Siegel, "Partitionable SIMD computer system interconnection network universality," *Sixteenth Allerton Conference on Communication, Control, and Computing,* October 1978, pp. 586-595.

[Sie79] H. J. Siegel, "A model of SIMD machines and a comparison of various interconnection networks," *IEEE Transactions on Computers,* Vol. C-28, December 1979, pp. 907-917.

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

[SrA83] V. P. Srini and J. F. Asenjo, "Analysis of CRAY-1s architecture," *International Symposium on Computer Architecture,* May 1983, pp. 194-206.

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

[SuT87] S.Y. 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.

[ZeL88] S. A. Zenios and R. A. Lasken, "The connection machines CM-1 and CM-2: solving nonlinear network problems," *1988 International Conference on Supercomputing,* July 1988, pp. 648-658.