JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 11, 239–251 (1991)

Experimental Analysis of a Mixed-Mode Parallel Architecture Using Bitonic Sequence Sorting*

SAMUEL A. FINEBERG AND THOMAS L. CASAVANT

Parallel Processing Laboratory, Department of Electrical and Computer Engineering, University of Iowa, Iowa City, Iowa 52242

AND

HOWARD JAY SIEGEL

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

Experimentation aimed at determining the potential benefit of mixed-mode SIMD/MIMD parallel architectures is reported. The experimentation is based on timing measurements made on the PASM system prototype at Purdue utilizing carefully coded synthetic variations of a well-known algorithm. The synthetic algorithms used to measure and evaluate this system were based on the bitonic sorting of sequences stored in the processing elements. This computation was mapped to both the SIMD and MIMD modes of parallelism, as well as to two hybrids of the SIMD and MIMD modes. The computations were coded in these four ways and experiments that explore the trade-offs among them were performed. Consideration is given to the overhead caused by enabling or disabling PEs in SIMD mode and a lower bound on this overhead is shown. Finally, a more efficient PE mask generation scheme for multiple "off-the-shelf" microprocessor-based SIMD systems is emulated and its performance is analyzed. The results of these experiments are presented and are discussed with special consideration of the effects of the system's architecture. The goal is to (as much as possible) obtain implementation independent analyses of the attributes of mixed-mode parallel processing with respect to the computational characteristics of the application being examined. The results are used to gain insight into the impact of computation mode on synchronization and data-conditional aspects of system performance.

1. INTRODUCTION

While extensive past efforts have dealt with the design and analysis of SIMD and MIMD algorithms, computations, and machines, this work describes empirically based architecture research generated from experiments on a *mixed-mode* machine capable of operating in both modes of parallelism. Furthermore, it is assumed that a mixed-mode machine can switch between modes at instruction level granularity with generally negligible overhead. Examples of mixed-mode machines include PASM [40] and OPSILA [3]. This research was performed in an attempt to gain insight into the characteristics of the general class of SIMD/MIMD mixed-mode parallel architectures. Specifically, the performance of synchronization and data-conditional aspects of the PASM prototype mixed-mode machine are evaluated by exercising the relevant architecture features via synthetic computations. These computations happen to be based on the bitonic sorting [4] of sequences of numbers. This computation was chosen because it has a structure simple enough to permit analysis of architectural features through controlled measurements of program execution time. However, in some cases, to conduct the architecture measurements desired, the synthetic algorithm did not, in fact, perform sequence sorting. That is, the object of the experiments was to evaluate the architecture—*not* study sequence sorting.

The experiments described are based on pure SIMD, pure MIMD, and two distinct mixed-mode SIMD/MIMD versions of an algorithm based on 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 two mixed-mode versions are *S/MIMD*, which uses MIMD mode to evaluate all conditionals and execute conditional instructions and SIMD mode otherwise, and *BMIMD*, which operates exclusively in MIMD mode but uses PASM's SIMD hardware to perform a constant time barrier synchronization just prior to each network transfer. The results show that the S/MIMD version has the best performance, the BMIMD is next, followed by the pure MIMD and pure SIMD, in that order, demonstrating the potential advantage of a mixed-mode architecture for tasks whose

* This research was supported by the National Science Foundation under Grants CCR-8704826 and CCR-8809600, by the Air Force Office of Scientific Research under Grant F49620-86-K-0006, by the Naval Ocean Systems Center under the High Performance Computing Block, ONT, by the Office of Naval Research under Grant N00014-90-J-1937, and by the NSF Software Engineering Research Center (SERC).

0743-7315/91 $3.00

computational characteristics are similar to those of this application.

In addition to examining the trade-offs among the above modes of parallelism, another architectural design feature is evaluated here. This is the dynamic evaluation of processing element (PE) condition codes (derived from the processor status register) and subsequent enabling and disabling of PEs in multimicroprocessor-based SIMD designs. (A multimicroprocessor-based system is defined here as a system utilizing multiple "off-the-shelf" microprocessors for its PEs.) In such systems, a nonzero amount of overhead is required for the calculation of condition code values and to subsequently enable or disable PEs. This overhead cost further emphasizes the advantage of a mixed-mode architecture for algorithms involving conditional statements based on PE data. The actual impact of the condition code design is measured for the PASM prototype. It is shown that the current PE mask generation method can be enhanced so that system performance is significantly improved. Evaluation of the performance of an enhanced design is performed via an extrapolation of actual system measurements based on synthetic computations.

Analyses are made to obtain results that are as independent of actual architecture implementation as possible. This is necessary to gain insight into the general characteristics of mixed-mode parallel computation. Concerns such as processor technology always present a problem, and in our study these aspects are dealt with by focusing on a comparison *among* algorithm versions and how these reflect on the basic modes of parallelism. Issues of system design choices are also filtered out by experimentation with synthetic versions that provide emulated timings for various modified system implementations. For example, it is shown that the effects of PASM's current "Condition Code Logic" can be altered to allow a more generic comparison of the SIMD mode with other modes of computation.

It is *not* the goal of this study to find a fast parallel sorting algorithm; the fact that the application used is sorting is irrelevant. The goal of the study presented here is to examine the trade-offs among the different modes of parallelism and their hybrids. For this particular study, a realistic application that was inter-PE communication intensive and local data conditional intensive was sought. Furthermore, the ability to vary this intensity was desired. The bitonic sequence sorting algorithm had the desired computational characteristics for this study and so was chosen.

Section 2 briefly reviews related work, and Section 3 overviews PASM and its prototype. Section 4 describes the basic algorithm that was used, while Section 5 provides the programmed variations of this algorithm as implemented on PASM for use in the experiments. The raw data are presented in Section 6. In Section 7, the empirical results are discussed with special consideration of the PASM architecture, the effects of inter-PE communication overhead, and condition code evaluation. A modification to the current design of PASM's condition code logic is proposed and its performance is evaluated in Section 8.

2. BACKGROUND AND RELATED WORK

Work relating to this research includes measurements of algorithm performance on actual parallel systems, analytical studies of parallel systems, and simulations of parallel systems [20]. These measurements fall into three major categories: *benchmarks, initial experiments*, and *highly efficient parallel programs*.

Benchmarks are intended to measure the peak performance of a system and to compare system speeds utilizing a common task and the system's own compilers. Many packages are available for this method, including LINPACK [12], EISPACK [42], the Los Alamos benchmark [18], and the Livermore benchmark [30]. Dongarra has compiled a list of the relative efficiencies of over 100 computers using the LINPACK package [11] and much work has been done in developing new benchmarks. Benchmarks are very good for determining the relative performance of machines running standard software and with parallelizing compilers. However, they are not a true *architecture* evaluation metric because they do not take suboptimal compilers and the effects of program restructuring into account. Work involving the performance benefit of program restructuring utilizing various system models includes [27, 28, 22]. Other work on the effects of program restructuring has been done for some specific commercial machines, including the CRAY X-MP [6], the Alliant FX/8 [23], and the CRAY-1 [43].

The second category, initial experiments, includes programs designed to test either a prototype or a simulator to better determine how it will perform with existing applications. These are usually coded with either an explicitly parallel language (e.g., C with tasking and communications extensions) or in assembly language and tend to be well-understood algorithms. Here, the algorithms need not necessarily be coded efficiently. These programs include matrix multiplication [14], FFT [5], partial differential equation solvers [15], and other relatively simple parallel algorithms [7]. Work in this area has been performed on the BBN Butterfly [9], Cm* [15, 16], the CM-1 and CM-2 [46], OPSILA [3, 13], PASM [5, 14], and Warp [2]. Work of this type has also been performed on simulation models of proposed architectures, such as an optical architecture [29], dataflow architectures [17, 32], VMP [8], and SM3 [45].

The final category, highly efficient parallel programs, includes work performed to achieve very high efficiency and to determine the limits of parallelism in an application. The programs used in this type of work may be coded in part in assembly language and utilize highly efficient techniques to gain the maximum performance from a machine. An early example of this work was performed by Rosenfeld [34] on

a simulated architecture. Other work includes [15], which explored the speedup possible on the Cm* testbed. An effort in this area has been performed by Gustafson *et al.* [19], utilizing a 1024-processor NCube. This work illustrated the *scaled speedup* approach and showed how it was possible to achieve a speedup of 1020 on a 1024-processor hypercube performing a nontrivial application. Other work has been performed on PASM showing how super-unitary speedup [21] could be achieved relative to the number of PEs [14].

The research being presented here utilizes elements of all of these techniques to gain insight into parallel architectures. This work is distinguished from the above related efforts in that it focuses on mixed-mode parallel computing and on the specific effects of a system's architecture and mode of computation.

3. OVERVIEW OF PASM AND THE PASM PROTOTYPE

The PASM (*par*titionable *S*IMD/*M*IMD) system is a dynamically reconfigurable architectural design where (1) the processors may be partitioned to form cooperating or independent submachines of various sizes, (2) each submachine can operate in SIMD or MIMD mode and switch between the two with instruction level granularity and generally negligible overhead (mixed-mode parallelism), and (3) the processors communicate through a multistage cube network that can be software configured to provide different connection topologies. A 30-processor prototype has been constructed at Purdue and was used in the experiments described in Section 6. This section discusses the PASM architecture characteristics that are most relevant to the reported experimentation. For a more general description of the architecture and its use, see [38, 39, 40].

The *Parallel Computation Unit* of PASM contains $P = 2^p$ PEs (numbered from 0 to $P - 1$) 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 $Q - 1$, that act as the control units for the PEs in SIMD mode and may orchestrate the activities of the PEs in MIMD mode. Each MC controls P/Q PEs, and together an MC and its PEs are referred to as an *MC group*. PASM has been designed for $P = 1024$ and $Q = 32$ ($P = 16$ and $Q = 4$ in the prototype). A set of MC groups form a *submachine*. 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 may get instructions and common data for coordinating its PEs from its memory. Special hardware coordinates the multiple MCs that are concurrently operating together as a submachine [35].

The PASM prototype system was built for $P = 16$ and $Q = 4$. This system employs Motorola MC68000 processors with a clock speed of 8 MHz as PE and MC CPUs. The interconnection network is a circuit-switched Extra Stage Cube network, which is a fault-tolerant variation of the multistage cube network [1, 41]. 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 Fig. 1 [35]. The MC contains a memory module from which the MC CPU reads instructions and data. Whenever the MC CPU 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. A mask consists of a simple bit vector where if $v(i) = 1$, PE i of the MC group is enabled. Because the Fetch Unit Controller 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 the *SIMD instruction space*, a reserved area within the PE logical address space. This is handled by the PE's Instruction Broadcast Unit (IBU), shown in Fig. 2 [35]. Whenever the IBU detects an access to this area (which does not correspond to part of the PE's physical memory), a request for an SIMD instruction is sent to the Fetch Unit. Only after all PEs in the submachine that are enabled for the current instruction have issued a request is the instruction

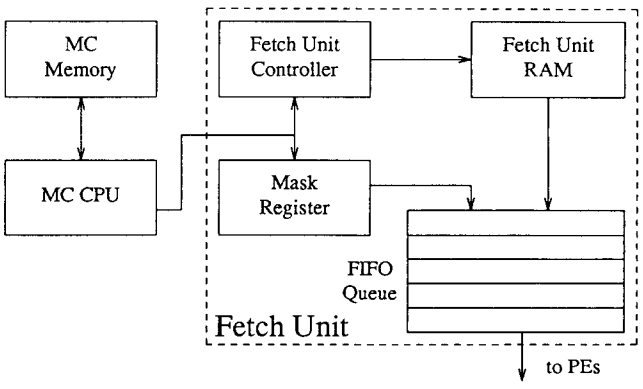

FIG. 1. Simplified MC structure.

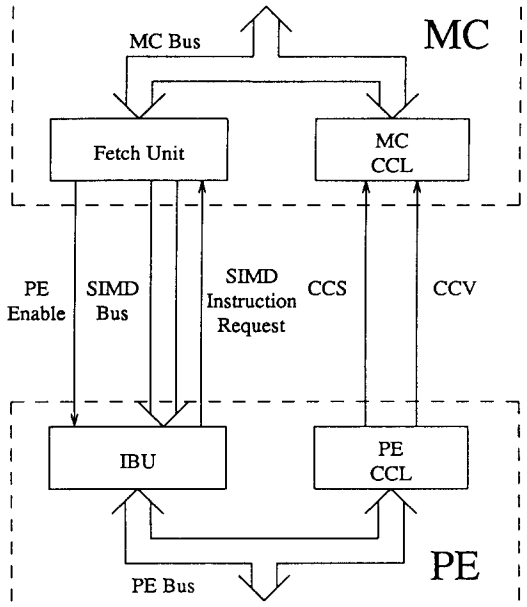

FIG. 2. MC–PE connections.

released by the Fetch Unit Queue. Then the PEs enabled for this instruction receive and execute it. Disabled PEs do not participate in the instruction and wait until an instruction for which they are enabled is broadcast. Thus, a switch from MIMD to SIMD mode is reduced to executing a JMP instruction to the reserved SIMD instruction space, and a switch from SIMD to MIMD mode is performed by executing a JMP to the appropriate PE MIMD instruction address located in the PE's actual main memory space. PEs remain in a given mode unless explicitly switched to the other mode. In both modes, local PE data are stored in the PE's actual main memory space.

The SIMD instruction broadcast mechanism can also be utilized for *barrier synchronization* [10, 25] of MIMD programs. Assume that 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.

Consider the time required for synchronization after all PEs in the submachine have requested synchronization, i.e., not including the time each PE needs to wait for the other PEs in the submachine to complete all operations and generate synchronization requests. Because all of these requested data fetches are satisfied simultaneously, the amount of time required for this synchronization is constant. Further, it is equal to the memory access time of a location within the PE's SIMD address space. Because this time does not increase significantly with the number of MC groups in the submachine, this is considered to be of $O(1)$ time complexity. For a more complete description of the implementation of the SIMD instruction broadcast mechanism, see [35].

Another component of PASM that is critical for this study is its circuit-switched multistage cube interconnection network. The PASM 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 Input (DTRin) associated with that PE. The PE must then set a bit in a control register to instruct the network interface to interpret the value in the DTRin as a routing tag for setting up a path through the network. The routing tag will be the first data item received from the network at the beginning of a network transfer. Word (16-bit) data values may now be written to the DTRin and automatically sent through the network. The receiving PE reads the transferred word from its network Data Transfer Register Output (DTRout). 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.

In SIMD mode, the MCs can perform data-conditional operations that are dependent on PE results. To support this, the Condition Code Logic (CCL) (see Fig. 2) is provided [35]. This permits the MCs to request condition code information from the PEs. This conditional information may be used to generate SIMD masks on the basis of the results of operations performed in the PEs. It may also be used to alter program control flow at the MC level (e.g., "if any" or "if all" statements). In addition, it may be used to generate masks that are not purely a function of local conditional information from within each PE (e.g., the condition code bit from one PE may be needed to compute the mask bit for another PE). The CCL consists of two parts, the PE Condition Code Logic (PE-CCL) and the MC Condition Code Logic (MC-CCL). The PE-CCL consists of two 1-bit registers, the PE Condition Code Synchronization (PE-CCS) register and the PE Condition Code Value (PE-CCV) register. These registers are mapped into the MC-CCL, which consists of two (P/Q)-bit registers. These are the MC Condition Code Synchronization (MC-CCS) register and the MC Condition Code Value (MC-CCV) register. Each of these contains P/Q bits corresponding to the bits of the PE-CCVs and PE-CCSs in the MC's submachine. The operation of the CCL 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 [37]. There are $P = 2^p$ PEs numbered (addressed) from 0 to $P - 1$. Each PE includes a processor, a memory module, a single data transfer register (DTR), and a register containing the PE's address (ADDR). "ADDR(i)" denotes the ith 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 is defined using $b_{p-1} \cdots b_1 b_0$ as the binary representation of an arbitrary PE address and $\bar{b}_i$ is the complement of b_i. The *Cube* network consists of p functions,

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

for $0 \leq i < p$.

Batcher's bitonic sorting method is described in [4, 26, 33, 44]. The SIMD version presented here is from [36]. A *bitonic sequence* is (a) a sequence of numbers $x_0, x_1, \ldots, x_{r-1}$, such that

$$x_0 \leq \cdots \leq x_{k-1} \leq x_k \geq x_{k+1} \geq \cdots \geq x_{r-1},$$

for some k, $0 \leq k < r$, or (b) a cyclic rotation of such a sequence. Consider a list of P items, one in the DTR of each PE. Assume the data in the DTRs of the P PEs form a bitonic sequence when listed in order from PE 0 to PE $P - 1$, where $r = P$ and $k = P/2$. Let A denote an internal CPU register in each PE. Then to sort the DTR data so that the data item in PE i is less than or equal to the data item in PE $i + 1$, $0 \leq i \leq P - 2$, all PEs execute the algorithm shown in Fig. 3.

In the algorithm, each PE saves the value of its DTR in A. Then, pairs of PEs whose addresses differ only in the ith bit position exchange data. In each PE, the DTR data received is then compared with the old DTR data saved in A. The minimum of each data pair is saved in the PE with a 0 in the ith position and the maximum in the PE with a 1 in the ith position. All PEs whose ith address bit is 0 compute the $P/2$ minimums in parallel, and then all PEs whose ith 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 pairs of PEs.

```
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)
```

FIG. 3. Bitonic sequence sorter.

To sort arbitrary sequences, as opposed to bitonic sequences, p stages of bitonic sorters are needed, numbered from 1 to p. The ith stage acts as 2^{p-i} 2^i-element bitonic sequence sorters, $1 \leq i \leq p$. Half of these sorters produce ascending sequences, as described in Fig. 3. The other half produce descending sequences by switching the "min" and "max" functions in the 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 $i + 1$. Therefore, 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 [26]). Thus, the algorithm shown in Fig. 4, executed by all enabled PEs in the SIMD machine, sorts an unordered list of P data items, one in the DTR of each PE.

In the arbitrary sequence sorter, the conditional "if $j = p$ or ADDR(j) = 0" determines, for each value of j, which PEs will act as ascending sorters and which will act as descending sorters. The variable *type* specifies whether a sort 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 is studied. 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. Figure 4 is modified in the following way to handle lists. Consider an instance of "$X \leftarrow \min(X, Y)$" or "$X \leftarrow \max(X, Y)$." 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 Fig. 4 is shown in Fig. 5. 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, 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'. This is a simple $O(N/P)$ merge routine for merging two sorted lists. It requires at most $3(N/P)$ *if-then-else* conditional operations. These are to check which of the elements being examined (using data local to the PEs) is lower, to check if the end of list X has been reached, and to check if the end of list Y has been reached. In MIMD mode, when the end of a list is reached, a PE simply copies the remainder of the other list onto the end of the new sorted sequence (reducing

```
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)
```

FIG. 4. Arbitrary sequence sorter.

the number of needed conditional statements). In SIMD mode, PEs may reach the end of a list at different times. To handle this, the completed list is represented by an element with a value larger than all allowable values. This causes the remainder of the other list to be copied onto the end of the new sorted sequence.

After completing the merge, the PEs use "choose(X, X', Y')" to choose between the two lists (pointers) depending on each PE's value of *type* and "ADDR(i)." This is analogous to establishing ascending and descending sequences in Fig. 4 (with the elements within each list X' and Y' always in ascending order):

```
choose (X,X',Y') {
  if(type = ADDR(i))
    then X ← X'
    else X ← Y'
  }
```

"Choose" contains several statements that are dependent on PE numbers and therefore requires masking in SIMD mode. However, this situation differs from the merge routine in that the masks are independent of local PE data and are precomputed. Thus, the CCL is unnecessary in the implementation of this routine. Each PE now has a new X list and deletes its Y list.

5. IMPLEMENTATIONS OF THE ALGORITHM

Programs implementing the sorting algorithm described above 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 S/MIMD version. In each 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. The elements within a subset are sorted in advance and placed in a PE in ascending order. Upon 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 $i \leqslant j$). All calculations are made as early as possible so that no invariants are recomputed. This includes the calculation of *type* in each iteration of the j loop and the masks for the choose routine in SIMD mode (which are stored in tables) for each iteration of the i loop. Because the masking required for the choose routine depended only on the PE number and i, these masks could be precomputed and therefore did not add any overhead in SIMD mode other than the serialization inherent in masking (i.e., the "then" must precede the "else," see Section 5.1).

5.1. *MIMD*

In the MIMD version, all instructions are executed in the PEs and are fetched from the PEs' own memory modules. All data are contained in the PEs' memory modules. MIMD mode is a natural choice over SIMD for this algorithm because, while programs in each PE are the same, the *if-then-else* constructs based on PE data conditions in the merge routine are executed more efficiently in MIMD mode. When these are executed in SIMD mode, the *then* must finish on the PEs for which the *if* condition is true while the PEs for which it is 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-conditional statements independently and therefore are never forced to be idle because of such a statement. In addition, the generation of conditional masks in SIMD mode adds considerable overhead, as is discussed in Section 7.2.

However, a disadvantage of MIMD mode is that after the choose routine, the PEs cannot proceed completely independently. 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 network reads and writes. These require software polling by the PEs, which adds substantial overhead to 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-reg-

```
for i = j-1 step -1 until 0 do
    for q = 1 step +1 until N/P do
        DTRin ← X[q]
        cube_i
        Y[q] ← DTRout
    merge (X,Y,X',Y')
    choose (X,X',Y')
```

FIG. 5. Inner loop for sorting lists.

ister 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. To receive data, a similar procedure in which the network output buffer is polled to determine if it contains any data must be carried out. Then, when there is data to be read, a memory-to-memory move instruction is required.

This overhead could be reduced if DMA block transfers were possible. In this case only a single transfer would be necessary in each iteration of the *i* loop in Figure 5. However, it was assumed that the basic model under consideration was that of a single-word DTR-based network. In addition, interrupt-driven network routines were not utilized because their use would not be of any benefit. This is because of the data dependency that exists between the completion of the network transfer and the merge. Furthermore, they would add overhead related to interrupt handling. It is possible that a combination of code reordering, interrupt-drive I/O, and DMA control could improve the performance of the pure MIMD version. However, the results presented here indicate that a mixed-mode version will still perform better than an improved MIMD version.

5.2. *SIMD*

In the SIMD version, a PE receives all instructions from its MC's Fetch Unit through a FIFO queue, as described in Section 3. All data stored in the PEs are held in their memory modules. Looping and control flow instructions are executed on the MCs, while data manipulation and effective address calculation instructions are executed on the PEs. Also, PE masks are needed to enable and disable PEs. The masks are generated on the basis of either the PE number or PE local data contained in their condition code status register. The masks based on PE number that are used in the choose routine are precalculated because they are static for a given problem size and number of PEs being used. The data-conditional masks (*if-then-else* statements) are contained in the merge portion of the algorithm and utilize the CCL to determine the proper masks.

SIMD mode is the slowest version of the algorithm for several reasons. First, as stated previously there is added serialization due to the inability of the PEs to overlap the *then* and *else* statements of the *if-then-else* constructs; i.e., when some PEs are executing the *then* statements, none can be executing the *else* statements. This occurs for both the merge and the choose routines. Because these are present in the merge routine, this serialization increases as N increases (for fixed P). In addition, the current prototype implementation of the CCL design can be enhanced, as is discussed in Section 7.2.

An advantage, however, of SIMD mode is the ability to do very quick network transfers. In PASM, the network appears as two memory-mapped I/O registers, one for sending data (DTRin) and one for receiving (DTRout) data. Because in SIMD mode all PEs operate in lock-step, there is no need to poll the network input buffer before sending data to determine if the previous transfer has completed. This eliminates a substantial overhead that is incurred in the MIMD version. For SIMD mode, only the final memory-to-memory move instruction used in MIMD mode is needed for sending or receiving.

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 overlap has been shown to yield a significant performance benefit [14]. However, this is dependent on the FIFO remaining nonempty. Unfortunately, each time the CCL is used it requires that the FIFO empty at least once. This is because the new mask value must be determined using the CCL before the Mask Register can be loaded for the new instructions to be enqueued. When this occurs, the overlap effect is eliminated. This makes the overlap benefit less evident in the portions of the algorithm where data dependent masks are generated (i.e., the merge routine). However, when masks based on PE addresses (not PE data) can be precomputed they do not prevent MC/PE overlap (i.e., the mask controlling the choose routine).

5.3. *BMIMD*

To reduce the network overhead in the MIMD version, a BMIMD version of the algorithm was developed. This version is identical to the MIMD version except that before each network transfer occurs, the PEs are barrier synchronized utilizing the SIMD instruction space, as described in Section 3. This synchronization consists of a single memory move instruction and allows the network registers to be written to and read from without polling (as in SIMD mode). This provides a substantial reduction in execution time over the MIMD version by greatly reducing network overhead. As in the MIMD case, if block transfers were possible, only a single such transfer would be required for each iteration of the *i* loop in Fig. 5. Thus, the synchronization overhead would be reduced to a single barrier per *i*-loop iteration.

5.4. *S/MIMD*

While the BMIMD version reduced the overhead required for network transfers, it did not eliminate it as the SIMD mode did (i.e., it had to do the SIMD memory space read to synchronize). Also, the benefit of the control flow overlap is not possible in the BMIMD version. To examine this effect, a fourth version of the algorithm that transfers 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 was created. The merge and choose routines are executed in MIMD mode. All other operations are done in SIMD mode. This permits the i, j, and q loops to be controlled by the MC, thus allowing this control flow to be overlapped. It also allows all network transfers to be done in SIMD mode. The only added overhead is in the movement between SIMD and MIMD modes. These mode switches are performed with JMP instructions and are therefore of minimal cost.

6. EXPERIMENTS PERFORMED

The algorithm versions were coded in MC68000 assembly language and executed on the PASM prototype. Timings were made using PASM's internal timers (MC68230 chips). The execution times of the four versions were measured for $4 \leqslant n \leqslant 9$ with $p = 2$, 3 and $5 \leqslant n \leqslant 9$ with $p = 4$ ($N = 2^n$, $P = 2^p$). These measurements are shown in Figs. 6 through 10. These figures utilize a log scale (vertical axis) for execution time and a linear scale (horizontal axis) for problem size, N, and are each for a fixed size submachine of PEs, P. Also, an SIMD version that did not test condition codes (SIMD/no-CC) was timed 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 on this version are provided in Section 7.2. In addition, an SIMD modified condition code (SIMD/m-CC) version that emulated a more efficient mask generation method was timed. This version also did not produce valid output. It is described in Section 8.

7. DISCUSSION

In this section, the different versions of the algorithm are discussed and compared. Also, the effects of the architectural features on these programs are evaluated and the differences in these effects across the versions are examined. Absolute execution times are unimportant here; it is the relative execution times as a function of mode of parallelism, N, and P that are of interest. In each case, the inherent attributes of the modes of parallelism and of the system's architecture are considered.

7.1. *Overall Comparison*

As can be seen from Figs. 6, 7, and 8, the fastest version was clearly the S/MIMD version, with the execution time of the BMIMD version slightly greater at all points. After these two mixed-mode versions, the MIMD version was next, with the SIMD version being the slowest. Note the similarity between Figs. 6 and 7. This is due to the time complexity of the algorithm, which is proportional to $(N/P)p(p + 1)/2$. This complexity is the same for $P = 4$ and $P = 8$ ($(N/P)p(p + 1)/2 = (N/4)(2 \times 3)/2 = (N/8)(3 \times 4)/2$). However, slight differences still occur due to variations in overheads and control flow overlap.

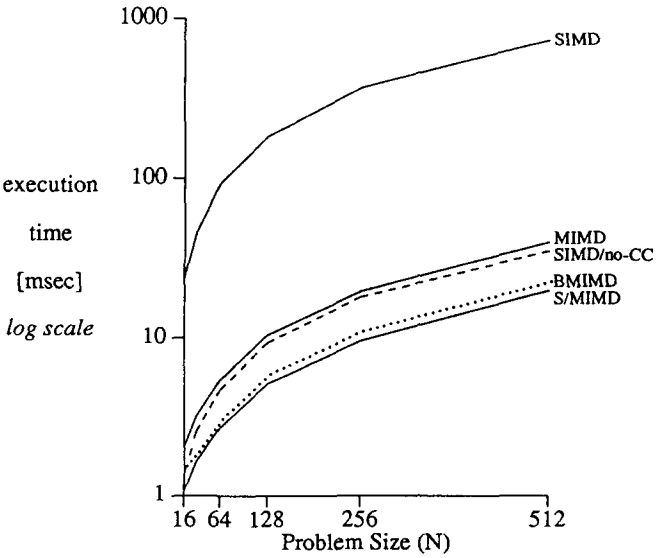

FIG. 6. Execution time vs problem size for $P = 4$.

MIMD vs BMIMD. First, consider the MIMD and BMIMD versions. Conceptually, these were the most similar 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. (Note that a constant vertical distance between lines of positive slope represents an increasing difference on this log scale.) The use of barrier synchronization is the only difference between the MIMD and BMIMD versions. The number of network transfers is $(N/P)p(p + 1)/2$, so as N increases the advantage gained by the more efficient network transfer in BMIMD mode also grows.

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. Its speed can be attributed to its having almost no overhead 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 transferring control between SIMD and MIMD modes. Note that as N increases the advantage of S/MIMD over MIMD and BMIMD improves due to the $(N/P)p(p + 1)/2$ complexity of the q loop (see Fig. 5) operations. These operations benefit from SIMD control flow overlap and SIMD synchronized network transfers. Also, the mode changing overhead of S/MIMD mode is outside of the q loop and occurs only $p(p + 1)/2$ times. In BMIMD mode, the barrier synchronization overhead is within the q loop and therefore occurs $(N/P)p(p + 1)/2$ times. However, for this particular

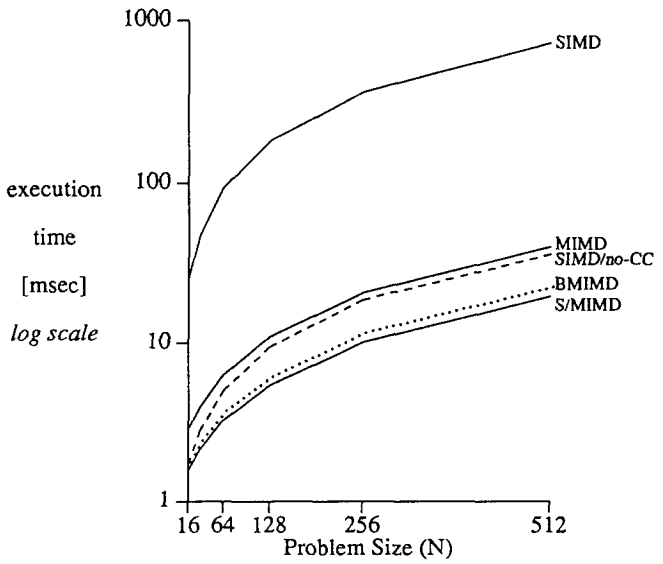

FIG. 7. Execution time vs problem size for $P = 8$.

study, the difference between BMIMD and S/MIMD is not that great.

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. The use of the CCL occurs within the $O((N/P)p(p+1)/2)$ time complexity merge routine that was the source of all of the PE local data conditional instruction overhead and most of the *if-then-else* serialization due to masking (as was mentioned in Section 5.2).

7.2. *Effects of the Condition Code Logic*

Consider the execution times of the SIMD version both with and without conditional mask generation (Figs. 6 to 8). While the shapes of these two curves are similar, there was a large difference between the performances of these versions. This difference is due to the overhead caused by PE mask generation. The SIMD version represents the current design of the PASM prototype's CCL, while the SIMD with no conditional mask generation (SIMD/no-CC) version represents a lower bound on execution time if conditional masks could be generated with zero cost. These data were collected by removing the relevant statements from the program, e.g., doing the comparison and transferring the results from the PEs to the MC. This is to avoid any time penalty caused by any inefficiencies in the specific implementation of data-conditional masking in the prototype.

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 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 accessible directly by the MC (for "if any" type instructions) and set directly by the PEs whenever an operation that affects the processor status is performed. This situation is approximated by the SIMD/no-CC version 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 multimicroprocessor PASM architecture, so it became necessary to design external hardware capable of disabling PEs, 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. To better explain the effects of the current implementation of the PASM CCL, the operation of the current design is now described. Referring to Figs. 1 and 2, consider how an MC receives conditional information from its PEs.

To simplify the implementation of the prototype, data-conditional masking is done by having an MC read its PEs' conditional values and set the Fetch Unit Mask Register appropriately. When an MC wants to test a status flag on its PEs it must:

(1) send an instruction (via the Fetch Unit Queue) to its PEs to clear their PE-CCS registers;

(2) wait for all of its PEs to finish this operation by waiting for all bits of its MC-CCS register to clear (this requires that the Fetch Unit Queue empty while the MC waits for its MC-CCS to clear);

(3) send instructions to its PEs to move their status register contents to a memory-mapped register and indicate what condition (e.g., $\leq$) should be tested by special hardware provided to calculate this value and place it in the PE-CCV;

(4) send an instruction to the PEs to set their PE-CCS registers to indicate that the value in their PE-CCV register is valid;

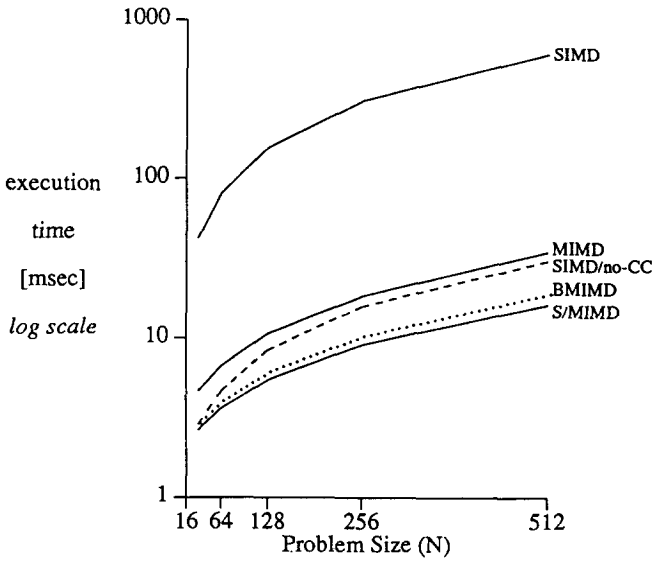

FIG. 8. Execution time vs problem size for $P = 16$.

(5) poll its MC-CCS register until all PEs have set their PE-CCS registers (this requires that the Fetch Unit Queue empty while the MC waits for its MC-CCS to become set).

The MC can then read the status information from its MC-CCV register and generate the appropriate mask. Hardware for combining condition codes from different MCs when necessary is also provided. For the results presented here, this whole process (with 8 MHz MC68,000 technology) takes about 448 CPU cycles (56 μs), thus adding a considerable overhead to the mask generation.

Another problem with generating masks on PASM is that each time a mask is generated, the Fetch Unit Queue is emptied (as mentioned in step 5 above). This, unfortunately, is necessary because masks cannot be changed once an instruction has been enqueued by the MC. This is analogous to the problem of a pipeline's emptying due to conditional branches in a typical RISC processor. 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 to send this instruction until it knows the status of the PEs, 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 degrades the performance of the SIMD version; however, its effect is hard to quantify because it is dependent on the application and programming style. This effect is further exacerbated by the current CCL design because of the length of time required for this mask generation.

7.3. *SIMD/no-CC vs S/MIMD and BMIMD*

In this section, the SIMD/no-CC version that has no conditional mask generation is compared with the two fastest versions to get a better sense of the differences between SIMD/no-CC and the mixed-mode versions. This case still assumes a Mask Register in each MC; however, it simulates zero-cost condition code determination. It does not attempt to emulate a masking scheme where PEs can enable and disable themselves as in the CM-2[46] and other SIMD machines. As is evident from Figs. 6 through 8, the SIMD/no-CC version is relatively close to the others for small N, but its execution time increases sharply as N increases (recall that execution time is on a log scale). In the S/MIMD and BMIMD versions, the execution time increases smoothly and with a slope much smaller than that in the SIMD/no-CC version. This is mainly caused by the added serialization in the merge routine due to the three *if-then-else* statements it contains. One is to check which list has the smallest element, one is to check if the X list has reached the end, and the third is to check if the Y list has reached the end (see Section 4). The merge is executed $p(p+1)/2$ times, and for each merge execution, each of the three *if-then-else* statements is executed up to N/P times. Therefore, the number of iterations of this loop will increase $O(N)$ for P held fixed. Hence, the serialization added due to the *if-then-else* statements will also increase $O(N)$ for P held fixed. 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, it did not add additional overhead because the *then* and *else* segments could be overlapped across PEs (i.e., the *then* and *else* are not serialized; each PE independently executes the *then* or the *else*).

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-memory move instruction to load the memory-mapped DTR. Because there must be $(N/P)p(p+1)/2$ network transfers, the network synchronization overhead for BMIMD mode will increase $O(N)$ for P held fixed while mode switching overhead for the S/MIMD version remains constant for P held fixed. The SIMD/no-CC version has no extra barrier or mode switching overhead, but due to the other factors mentioned, it is not faster than the BMIMD and S/MIMD versions. This effect becomes evident however only for large N, and this, along with the control flow overlap, makes S/MIMD the fastest version.

While the performance gain of S/MIMD mode over BMIMD mode in this particular study may not in itself justify the added cost of support for SIMD mode, SIMD mode uses less memory (i.e., programs are not replicated), is easier to program, and can provide more significant speedups in other applications [14, 38]. This issue is the subject of ongoing research.

8. A POSSIBLE MODIFICATION TO THE PASM CCL

The current design of PASM's CCL has been shown to be somewhat inefficient. Given this fact, and the fact that a lower bound on execution time has been measured, consider a different masking scheme appropriate for the design of multimicroprocessor-based SIMD/MIMD computers.

The SIMD/m-CC version, which was intended to emulate a more efficient mask generation method, was programmed and executed on PASM. This version assumes that the worst case time for the MC to read the PE status information and combine the PE-CCV registers into a mask vector is no longer than PASM's memory module access time ($\sim$600 ns). This combining would involve the generation of a bit mask representing the results of a logical operation from raw status information and would be performed by new logic added to the MC. This new logic is called the MC's Combining Logic. The emulation of this new mechanism was accomplished by simply adding a register-to-memory move instruction to the PE's program and a memory-to-register move to the MC's program wherever a condition test result should be sent from a PE to its MC (the PE status bits are still set and a vector

of all ones is enqueued as the mask value in the Fetch Unit Queue). It is assumed that the access time of the PE-CCV and MC-CCV registers will be as least as fast as the dynamic RAM used in the memory modules. Because the register-to-memory and memory-to-register moves will require ~1200 ns, which is assumed to be longer than would be needed in an actual implementation of this modified CCL, the time required by the Combining Logic to calculate the mask vector is included in this delay. This synthetic version did not, however, produce valid sorted output because the masks were never actually set properly. The SIMD/m-CC version is plotted along with the SIMD/no-CC and SIMD execution times for $P = 8$ in Fig. 9.

This graph demonstrates two things. First, it is possible to implement the condition code logic in a way that substantially reduces overhead relative to that in 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 Fig. 10. However, there is still overhead required for conditional mask generation, and this overhead increases with N, thus keeping the relationship among all curves the same with respect to the mixed-mode versions for $N \geq 32$. Unfortunately, this relatively minimal overhead is virtually impossible to avoid in a multimicroprocessor system made from off-the-shelf components and is one of the major drawbacks of using off-the-shelf components in an SIMD machine. However, this can likely be justified simply by the cost considerations of designing a custom processor. Also, the use of standard components simplifies the design of the PEs and enables them to run efficiently in MIMD mode. For a more detailed discussion of masking techniques, and possible hardware modifications to the PASM prototype to support them, see [31].

It is important at this time to point out that this algorithm was intended to be a worst case situation. Many algorithms

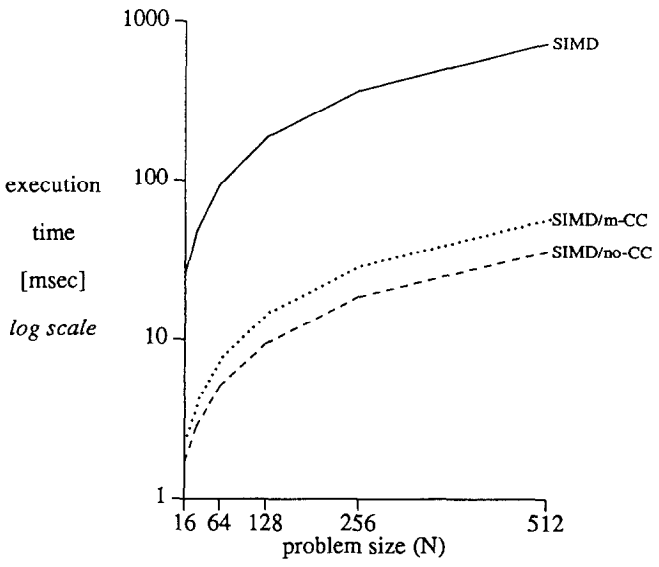

FIG. 9. Execution time vs problem size for $P = 8$.

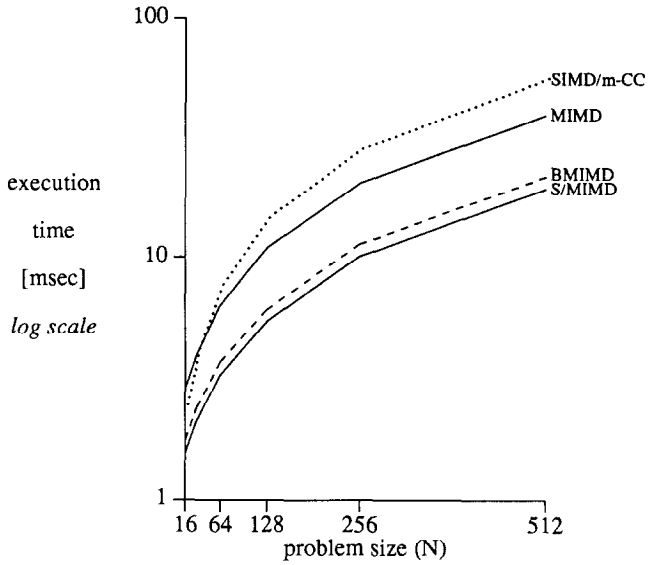

FIG. 10. Execution time vs problem size for $P = 8$.

do not exhibit as much use of conditional masking, and for those algorithms SIMD mode may be the best, as shown elsewhere [14]. Therefore, a more efficient masking scheme may allow some algorithms to perform better in SIMD mode than in S/MIMD mode by eliminating the mode switching overhead.

9. CONCLUSION

Experiments on the PASM prototype designed to examine the trade-offs among the SIMD, MIMD, BMIMD, and S/MIMD modes of parallelism were described. In addition, the effects of SIMD PE mask generation were considered. Experiments consisted of measurements of the execution time of synthetic variations of a bitonic sorting algorithm. The algorithm has feasible implementations in each of the SIMD, MIMD, BMIMD, and S/MIMD modes and, further, exercises PE mask generation and network transfer aspects of the architecture. Execution times for different size lists, numbers of processors, and modes of parallelism were collected. These data were evaluated and discussed, while the effects of the various parameters in the test were examined.

Had the initial list within each PE been unsorted rather than sorted, added time cost would occur before the start of the current routine. This would represent an extra time added to the total sort time but would not affect the relative performance of the different modes for the algorithm as studied. It would only increase the number of data-conditional operations, thus making the pure SIMD version perform worse (all of the other modes would do the initial sort in MIMD fashion).

A lower bound on the execution time of PASM's SIMD PE mask generation was measured. On the basis of this experience, a naive change in the design of PASM's mask generation logic that brought PASM's performance closer to this lower bound was proposed and experiments to quantitatively measure the effect of this change were performed.

This lower bound helps to indicate some deficiencies inherent in SIMD mode (i.e., *if-then-else* serialization) and demonstrates the need for mixed-mode architectures. This part of the study also demonstrates how particular details of the PASM prototype implementation can be stripped away in order to make the results of this research more generally applicable.

It was *not* the goal of this paper to present a method for parallel sorting on PASM. The goal was to investigate the impacts of inherent attributes of SIMD, MIMD, and mixed-mode parallelism for a given set of computational characteristics. This was done by a combination of experimentation on actual hardware and algorithm complexity analyses.

The results of these mixed-mode studies can be used in the development of a taxonomy of algorithm characteristics with respect to parallel implementation [24]. To map algorithms to reconfigurable parallel architectures efficiently, one must understand the ways in which the computational structures that comprise the algorithm will be executed using different modes of parallelism. The research presented here examines the implications of SIMD, MIMD, and mixed-mode approaches to computation structures such as PE data transfers, PE local data conditional masking, and data independent PE-address-based masking.

In summary, it was also shown that a mixed-mode machine can utilize its mode switching capability to support SIMD/MIMD parallelism and hardware barrier synchronization and to improve its performance over both pure SIMD and pure MIMD modes. These features were exploited in the mixed-mode S/MIMD and BMIMD versions. Through the use of these mixed-mode techniques, the advantages of both the SIMD and MIMD modes were simultaneously exploited, and the advantage of a machine with this mode switching capability for the class of tasks with the computational characteristics examined was demonstrated.

ACKNOWLEDGMENTS

The authors of this paper acknowledge many useful discussions with Mark Allemang, Ed Bronson, Wayne Nation, Pierre Pero, Tom Schwederski, and the other members of the Purdue Parallel Processing Laboratory Users Group (PPLUG). Preliminary versions of portions of this material were presented at the 1990 International Conference on Parallel Processing and at Supercomputing '90.

REFERENCES

1. Adams, G. B., III, and Siegel, H. J. The extra stage cube: A fault-tolerant interconnection network for supersystems. *IEEE Trans. Comput.* **C-31** (May 1982), 443–454.
2. Annaratone, M., Arnould, E., Gross, T., Kung, H. T., Lam, M., Menzilcioglu, O., and Webb, J. A. The Warp computer: Architecture, implementation, and performance. *IEEE Trans. Comput.* **C-36** (Dec. 1987), 1523–1538.
3. Auguin, M., and Boeri, F. The OPSILA computer. *In* Consard, M. (Ed.). *Parallel Languages and Architectures.* Elsevier Science, Amsterdam, 1986, pp. 143–153.
4. Batcher, K. E. Sorting networks and their applications. *Proc. AFIPS 1968 Spring Joint Computer Conference,* AFIPS Press, Montvale, NJ, 1968, pp. 307–314.
5. Bronson, E. C., Casavant, T. L., and Jamieson, L. H. Experimental application-driven architecture analysis of an SIMD/MIMD parallel processing system. *Trans. Parallel Distributed Systems* **1** (Apr. 1990), 195–205.
6. Calahan, D. A. Influence of task granularity on vector multiprocessor performance. *Proc. 1984 International Conference on Parallel Processing.* IEEE Computer Society, Los Alamitos, CA, Aug. 1984, pp. 278–284.
7. Casavant, T. L., Siegel, H. J., Schwederski, T., Jamieson, L. H., Fineberg, S. A., McPheters, M. J., Bronson, E. C., Disch, W., Schurecht, K., Loh, E. H., Ringer, C., Cox, B., and Toomey, C. A. Experimental benchmarks and initial evaluation of the performance of the PASM system prototype. Tech. Rep. TR-EE 88-2, Purdue University School of Electrical Engineering, 1988.
8. Cheriton, D. R., Gupta, A., Boyle, P. D., and Goosen, H. A. The VMP multiprocessor: Initial experience, refinements, and performance evaluation. *Proc. 15th Annual International Symposium on Computer Architecture,* IEEE Computer Society, Los Alamitos, CA, May 1988, pp. 410–421.
9. Crowther, W., Goodhue, J., Thomas, R., Miliken, W., and Blackadar, T. Performance measurements on a 128-node butterfly parallel processor. *Proc. 1985 International Conference on Parallel Processing.* IEEE Computer Society, Los Alamitos, CA, Aug. 1985, pp. 531–540.
10. Dietz, H. G., Schwederski, T., O'Keefe, M. O., and Zaafrani, A. Static synchronization beyond VLIW. *Proc. Supercomputing '89,* IEEE Computer Society, Los Alamitos, CA, Nov. 1989, pp. 416–425.
11. Dongarra, J. J. Performance of various computers using standard linear equations in a Fortran environment. *Comput. Architecture News* **18** (Mar. 1990), 17–31.
12. Dongarra, J. J., Bunch, J. R., Moler, C. B., and Stewart, G. W. *LINPACK User's Guide.* SIAM, Philadelphia, 1979.
13. Duclos, P., Boeri, F., Auguin, M., and Giraudon, G. Image processing on a SIMD/SPMD architecture: OPSILA. *Proc. International Conference on Pattern Recognition,* IEEE Computer Society, Los Alamitos, CA, Nov. 1988, pp. 430–433.
14. Fineberg, S. A., Casavant, T. L., Schwederski, T., and Siegel, H. J. Nondeterministic instruction time experiments on the PASM system prototype. *Proc. 1988 International Conference on Parallel Processing.* Penn State Press, University Park, PA, Aug. 1988, pp. 444–451.
15. Gehringer, E. F., Jones, A. K., and Segall, Z. The Cm* testbed. *Computer* **15** (Oct. 1982), 40–53.
16. Gehringer, E. F., Siewiorek, D. P., and Segall, Z. *Parallel Processing: The Cm* Experience.* Digital Press, Bedford, MA, 1987.
17. Gostelow, K. P., and Thomas, R. E. Performance of a simulated dataflow computer. *IEEE Trans. Comput.* **C-29** (Oct. 1980), 905–919.
18. Griffin, J. H., and Simmons, M. L. Los Alamos National Laboratory computer benchmarking 1983. Tech. Rep. LA 10151-MS, Los Alamos National Laboratory, 1984.
19. Gustafson, J. L., Montry, G. R., and Benner, R. E. Development of parallel methods for a 1024-processor hypercube. *SIAM J. Sci. Statist. Comput.* **9** (July 1988), 609–638.
20. Heidelberger, P., and Lavenberg, S. S. Computer performance evaluation methodology. *IEEE Trans. Comput.* **C-33** (Dec. 1984), 1195–1220.
21. Hembold, D. P., and McDowell, C. E. Modeling speedup(n) greater than n. *Proc. International Conference on Parallel Processing.* Penn State Press, University Park, PA, Aug. 1989, pp. III-219–225.
22. Hwu, W. W., and Cheng, P. P. Exploiting parallel microprocessor microarchitectures with a compiler code generator. *Proc. 15th Annual International Symposium on Computer Architecture,* IEEE Computer Society, Los Alamitos, CA, May 1988, pp. 45–53.
23. Jalby, W., and Meier, U. Optimizing matrix operations on a parallel multiprocessor with a hierarchical memory system. *Proc. 1986 Inter-*

national Conference on Parallel Processing. IEEE Computer Society, Los Alamitos, CA, Aug. 1986, pp. 429–432.

24. Jamieson, L. H. Characterizing parallel algorithms. *In* Jamieson, L. H., Gannon, D. B., and Douglass, R. J. (Eds.). *The Characteristics of Parallel Aglorithms.* MIT Press, Cambridge, MA, 1987, pp. 65–100.
25. Jordan, H. F. A special purpose architecture for finite element analysis. *Proc. 1978 International Conference on Parallel Processing.* IEEE Computer Society, Los Alamitos, CA, Aug. 1978, pp. 263–266.
26. Knuth, D. E. *The Art of Computer Programming. Vol. 3. Sorting and Searching.* Addison-Wesley, Reading, MA, 1973.
27. Kuck, D. J., Muraoka, Y., and Chen, S. On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speedup. *IEEE Trans. Comput.* **C-21** (Dec. 1972), 1293–1310.
28. Kuck, D. J., Sameh, A. H., Cytron, R., Veidenbaum, A. V., Polychronopoulos, C. D., Lee, G., McDaniel, T., Leasure, B. R., Beckman, C., Davies, J. R., and Kruskal, C. P. The effects of program restructuring, algorithm change, and architecture choice on program performance. *Proc. 1984 International Conference on Parallel Processing,* IEEE Computer Society, Los Alamitos, CA, Aug. 1984, pp. 129–138.
29. Louri, A., and Hwang, K. A bit-plane architecture for optical computing with two-dimensional symbolic substitution. *Proc. 15th Annual International Symposium on Computer Architecture,* IEEE Computer Society, Los Alamitos, CA, May 1988, pp. 18–27.
30. McMahon, F. H. The Livermore fortran kernels: A computer test of numerical performance range. Tech. Rep. UCRL-53745, Lawrence Livermore National Laboratory, 1986.
31. Nation, W. G., Fineberg, S. A., Allemang, M. D., Schwederski, T., Casavant, T. L., and Siegel, H. J. Efficient masking techniques for large-scale SIMD architectures. *Proc. Frontiers '90: The Third Symposium on the Frontiers of Massively Parallel Computation,* IEEE Computer Society, Los Alamitos, CA, Oct. 1990, pp. 259–264.
32. Patnaik, L. M., Govindarjan, R., and Ramdoss, N. S. Design and performance evaluation of EXMAN: An extended manchester data flow computer. *IEEE Trans. Comput.* **C-35** (March 1986), 229–243.
33. Quinn, M. J. *Designing Efficient Algorithms for Parallel Computers.* McGraw-Hill, New York, 1987.
34. Rosenfield, J. L. A case study in programming for parallel-processors. *Comm. ACM* **12** (Dec. 1969), 645–655.
35. Schwederski, T., Nation, W. G., Siegel, H. J., and Meyer, D. G. Design and implementation of the PASM prototype control hierarchy. *Proc. Second International Supercomputing Conference,* International Supercomputing Institute, St. Petersburg, FL, May 1987, pp. 418–427.
36. Siegel, H. J. Partitionable SIMD computer system interconnection network universality. *Proc. Sixteenth Allerton Conference on Communication, Control, and Computing,* University of Illinois, Urbana-Champaign, IL, Oct. 1978, pp. 586–595.
37. Siegel, H. J. A model of SIMD machines and a comparison of various interconnection networks. *IEEE Trans. Comput.* **C-28** (Dec. 1979), 907–917.
38. Siegel, H. J., Armstrong, J. B., and Watson, D. W. Mapping tasks onto the PASM reconfigurable parallel processing system. *Proc. 1990 Parallel Computing Workshop.* Sponsored by the CIS Department of The Ohio State University, Columbus, OH, March 1990, 13–24.
39. Siegel, H. J., Nation, W. G., and Allemang, M. D. The organization of the PASM reconfigurable parallel processing system. *Proc. 1990 Parallel Computing Workshop.* Sponsored by the CIS Department of The Ohio State University, Columbus, OH, March 1990, 1–12.
40. Siegel, H. J., Siegel, L. J., Kemmerer, F. C., Mueller, P. T., Jr., Smalley, H. E., Jr., and Smith, S. D. PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition. *IEEE Trans. Comput.* **C-30** (Dec. 1981), 934–947.
41. Siegel, H. J. *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies.* McGraw-Hill, New York, 1990, 2nd ed.
42. Smith, B. T., Boyle, J. M., Dongarra, J. J., Garbow, B. S., Ikebe, Y., Klema, V. C., and Molder, C. B. *Matrix Eigensystem Routines—Eispack Guide.* Springer-Verlag, Heidelberg, 1976.
43. Srini, V. P., and Asenjo, J. F. Analysis of CRAY-1S architecture. *Proc. 10th Annual International Symposium on Computer Architecture,* IEEE Computer Society, Los Alamitos, CA, May 1983, pp. 194–206.
44. Stone, H. S. Parallel processing with the perfect shuffle. *IEEE Trans. Comput.* **C-20** (Feb. 1971), 153–161.
45. Su, S. Y. W., and Thakore, A. K. Matrix operations on a multicomputer system with switchable main memory modules and dynamic control. *IEEE Trans. Comput.* **C-36** (Dec. 1987), 1467–1484.
46. Zenios, S. A., and Lasken, R. A. The connection machines CM-1 and CM-2: Solving nonlinear network problems. *Proc. 1988 International Conference on Supercomputing,* Association for Computing Machinery, New York, NY, July 1988, pp. 648–658.

Received August 10, 1989; revised October 31, 1990; accepted November 7, 1990

SAMUEL A. FINEBERG received the B.S.C.E.E. degree in 1988 and the M.S.E.E. degree in 1989 from the School of Electrical Engineering of Purdue University in West Lafayette, Indiana. He is currently enrolled in the Ph.D. program in the Electrical and Computer Engineering Department of the University of Iowa. His research interests include parallel computer architecture and performance evaluation. He is a student member of the Computer Society of the Institute of Electrical and Electronics Engineers (IEEE) and of the Association for Computing Machinery (ACM).

THOMAS L. CASAVANT received the B.S. degree in computer science in 1982, the M.S. degree in electrical and computer engineering in 1983, and the Ph.D. degree in electrical and computer engineering from the University of Iowa in 1986. From 1986 to 1989, Dr. Casavant was on the faculty of the School of Electrical Engineering at Purdue University, where he also served as Director of the Parallel Processing Laboratory. He is currently an assistant professor on the faculty of the Department of Electrical and Computer Engineering at the University of Iowa. His research interests include parallel processing, computer architecture, programming environments for parallel computers, and performance analysis. Dr. Casavant has served as Program Committee Vice-Chair for the International Conference on Distributed Computing Systems (1990), Track Co-Chair for COMPSAC (1991), and as guest editor for an issue of *IEEE Computer* on the subject of distributed computing systems (1991). He is a member of the IEEE Computer Society and the ACM.

H. J. SIEGEL received two B.S. degrees from the Massachusetts Institute of Technology (MIT) and M.A., M.S.E., and Ph.D. degrees from Princeton University. He is a professor and the Coordinator of the Parallel Processing Laboratory in the School of Electrical Engineering at Purdue University. He has coauthored over 130 technical papers, coedited four volumes, authored one book (*Interconnection Networks for Large-Scale Parallel Processing*), consulted, given tutorials, and prepared video tape courses, all on parallel processing. He was General Chairman of the *Third International Conference on Distributed Computing Systems* (1982), Program Co-Chairman of the *1983 International Conference on Parallel Processing,* and General Chairman of the *15th Annual International Symposium on Computer Architecture* (1988). He has been a guest-editor of two issues of the *IEEE Transactions on Computers.* He is a fellow of the IEEE and a member of the ACM.