Processor Autonomy on SIMD Architectures

P. J. Narayanan
Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213-3890
U. S. A.
E-mail: *pjn@cs.cmu.edu*

Abstract

Flynn classified high speed (parallel) computers into four categories. Of these, the single instruction stream, multiple data stream (SIMD) processor array machines have become very popular in practical parallel processing. The commercially available processor array machines display important architectural variety, while belonging to SIMD category of machines. In this paper, we further categorize the SIMD class of machines on the basis of processor autonomy *of the machines, which is the capability of the individual processing elements (PEs) to act autonomously in some significant way. For each autonomy class, we provide examples and illustrate some of its important algorithmic features. We also discuss how each type of autonomy can be simulated on machines without it. We study the addressing autonomous class of machines in greater detail by discussing three algorithms on machines with and without that type of autonomy. A discussion on how processor autonomy appears in algorithms in the literature and what impact they can have in the future machines also is provided.*

1 Introduction

Flynn classified high speed (parallel) computers into the following four categories based on how the individual processors receive instructions and data: Single instruction stream single data stream (SISD), single instruction stream multiple data stream (SIMD), multiple instruction stream single data stream (MISD), and multiple instruction stream multiple data stream (MIMD) [11]. Of these, the SISD category consists of the conventional serial processors. Pipelined processors, where each processor performs a part of the computation on the same data stream, belong to the MISD category. Parallel processors belong to the SIMD or the MIMD category based on the absence/presence of private instruction streams for each processor.

ICS-7/93 Tokyo, Japan
© 1993 ACM 0-89791-600-X/93/0007/0127...$1.50

We restrict the focus to SIMD machines in this paper. The architecture of a typical SIMD machine is shown in Figure 1. The common instruction stream is issued by the controller to all processing elements (PEs) through an instruction bus. This bus is also used for broadcasting scalar values to the PEs. The PEs have private memories and internal registere. Operands and results of all arithmetic and logical operations reside in the private memory or an internal register of each PE. Communications are performed through the interconnection network using the communication instructions. The address of memory operands is typically the same on all PEs and is usually supplied by the controller.

Processor array machines belonging to the SIMD category are very popular in practical parallel processing today. Many such machines are available commercially today, such as the Connection Machine CM-2 [1], and the MasPar MP-1 [4]. These machines exhibit a number of variations of the SIMD paradigm described above, differing from one another in important aspects of the model. In this paper, we subcategorize the SIMD model on the basis of *processor autonomy*. Processor autonomy of a machine is the capability of its individual processing elements to act independently in a significant way. The level of autonomy has its impact on the algorithms that can be implemented efficiently on the machine.

We describe the aspects in which the autonomy of individual PEs differ and define the autonomy categories in Section 2. A detailed study of the individual autonomy classes and their impact on the architectures of the machine are given in the next section. A detailed discussion on addressing autonomous class appears in Section 4. Section 5 documents how processor autonomy had been used in different forms in the literature.

2 Processor autonomy

Processor autonomy in SIMD architectures has been studied a little in the past. Maresca and Li discuss processor

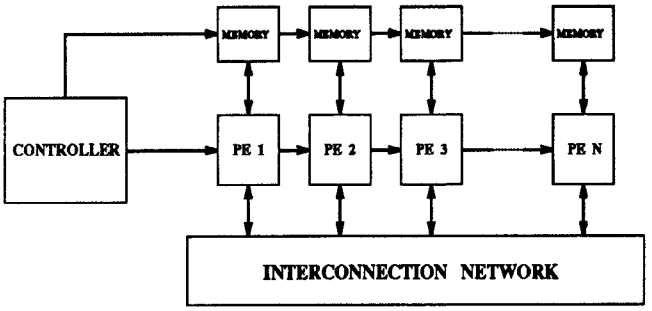

Figure 1: Architecture of a generic SIMD processor array machine

autonomies and the implementation of one type of autonomy in their polymorphic-torus architecture [18]. Fountain discusses the role of local autonomy in processor arrays in his book [12] and in a later paper [14]. In another paper, Duff and Fountain identified autonomies ranging from pure SIMD machines to full MIMD machine [10].

There are three architectural aspects that form the basis of our study on processor autonomies on SIMD machines: the selection of the instruction to execute in every cycle, the selection of the operands for an instruction, and the selection of the partner (the source or destination PE) in a communication operation. These architectural aspects are not exclusive of one another; PEs of different machines can have different levels of freedom of selection in one or more of these aspects. We categorize the SIMD model into six subcategories based on the level and kind of autonomy the individual PEs possess in these aspects. Our categorization is motivated by Fountain's scheme, but differ from it importantly.

We define the following six subcategories of SIMD processor arrays:

1. *Pure SIMD*: Machines with no local control.
2. *Activity control*: Machines that can participate in a computation step or abstain from it based on a local condition. This is a type of control on instruction selection.
3. *Connection autonomy*: Machines that can locally select one among the immediate neighbors as the source or destination of a one-step communication operation. This is a type of control on partner selection in direct communication steps.
4. *Communication autonomy*: Machines that can use a local variable as the source or destination address in a general (multi-step) communication operation. This is a type of control on partner selection in general communication steps.
5. *Addressing autonomy*: Machines that can use a local variable as the address of an operand in a computation step. This is a type of control on operand selection.
6. *Operation autonomy*: Machines that can locally select one of a few operations for execution. This is a type of control on instruction selection.

The pure SIMD category contains machines with no autonomy. Machines belonging to it do not possess any other kind of autonomy. The latter five autonomous categories are technically independent of one another. However, activity control is a part of every category in practice, except for the first. Connection autonomy and communication autonomy are related, the latter being a generalization of the former in some sense. However, many practical architectures without connection autonomy do provide (virtual) communication autonomy, but rarely the other way around. Addressing autonomy and operation autonomy are independent of other types of autonomy and can appear together with them.

In the next section, we describe each category in detail. We illustrate each category using the following conventions: The processing elements are represented by small circles. The shading of each PE denotes the operation it executes; inactive PEs are represented by dotted outlines. The operand selection is illustrated by shading a particular cell of an iconified memory array. The partner selection in a communication operation is illustrated by arrows from the source to the destination.

3 The autonomy classes in detail

We describe the salient features of each autonomy class in this section. Wherever appropriate, we give a set of operations that distinguish the autonomy class from others. These are operation that can be performed efficiently with that type of autonomy while being inefficient without it. We also discuss, for each autonomy class, how its capabilities can be simulated on machines without that type of autonomy.

3.1 Pure SIMD class

The category of pure SIMD architectures is the most restricted category of SIMD machines. The individual processing elements have no local control over any parameter of the execution on pure SIMD machines. Computations are uniform and regular over the processor array. Figure 2 shows the operation selection, the operand selection, and the partner selection in pure SIMD architectures. Typical examples of this class are systolic processor arrays designed for specific functions where each cell performs the same computation and communication during every cycle. Examples include the systolic convolver and the systolic matrix multiplier [30]. Even simple conditional constructs such as *if-then-else* cannot be implemented efficiently on this category of machines. For instance, consider the following simple statement to be executed independently on each PE:

Program *Conditional*
  if (A)
    then $a = b$
    else $c = d$
End *Conditional*

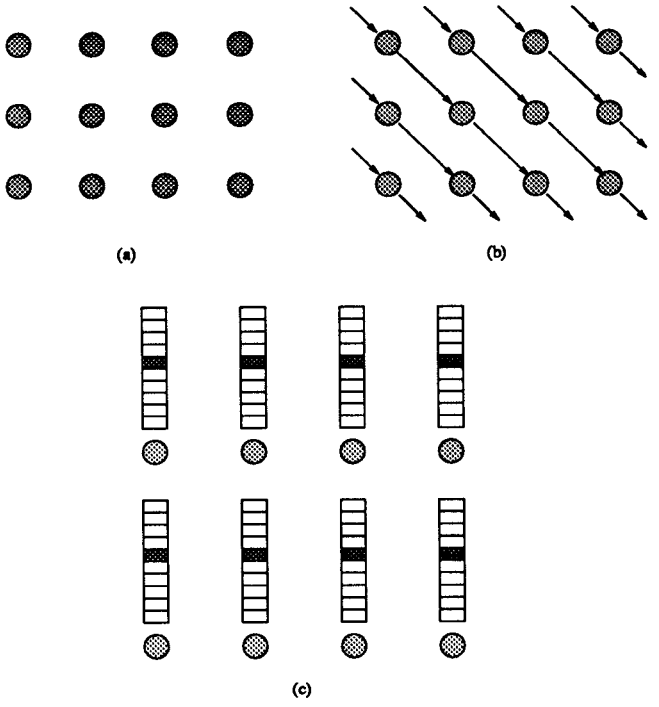

Figure 2: Pure SIMD architectures. (a) Operations are uniform (b) Communications are regular (c) Addresses are the same in all PEs

On pure SIMD architectures, all PEs have to be active at all times. Thus, a will have to be assigned to an expression that evaluates to b if the condition A is true and to a if it is false. To do this, we need to generate an A-mask, equal in length to a and b, that contains all 1's if A is true and all 0's otherwise. The same applies to the assignment involving c and d. Thus, the statement above can be coded as

Program *SIMD-Conditional*

(* Extend condition to a mask of all 1's or 0's *)

A-mask = A

A-bar = ~A-mask (* The inverted mask *)

a = (A-mask AND b) OR (A-bar AND a)

c = (A-mask AND c) OR (A-bar AND d)

End *SIMD-Conditional*

This requires at least eight expression evaluations and four assignments.

3.2 Activity control

In this category of machines, each processing element can choose to take part in a computation step or abstain from it, based on the value of a bit stored locally. Figure 3 illustrates how architectures with activity control differ from pure SIMD machines in the selection of instructions. This

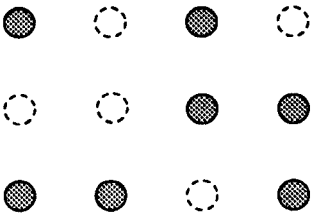

Figure 3: Activity control lets individual PEs drop out of each instruction

gives the architecture the power to implement *if-then-else* efficiently. For example, the *if-then-else* statement given above can be implemented by deactivating PEs where A is false to execute the *then* part and deactivating PEs where A is true to execute the *else* part, requiring only one expression evaluation and two assignments. Activity control is usually implemented using an activity or context register which is used to gate all operations that modify the memory or internal registers. It is interesting to note that a large number of common data parallel algorithms exploit only this level of autonomy even on more powerful machines. Examples include matrix multiplication [30], image convolution [16], and connected component labeling [22]. Most general purpose processor arrays possess activity control. Examples of machines with activity control as the highest form of autonomy include ICL DAP [25] and MPP [2].

Algorithms that require activity control can be implemented on architectures without it by transforming each conditional operation into a number of complex expressions involving logical ANDs and ORs, as in the *if-then-else* example above. Implementing loops that may reach their terminating conditions at different instances on different processing elements is more tricky. This will require conditioning the results of every statement inside the loop using masks derived from its termination condition like in the *if-then-else* example above, to ensure that the values do not change on PEs that have reached the termination condition. Nested loops will require nested conditioning, resulting in inefficient implementations.

3.3 Connection autonomy

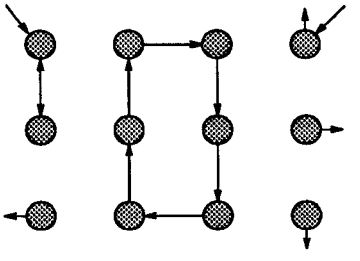

Figure 4: Connection autonomy permits different neighbors

In a connection autonomous machine, each PE can select a neighbor among the immediate neighbors as the partner (source or destination) of a communication step. This is important for algorithms that need to establish variable connections with neighbors on the basis of a local condition. Connection autonomy can be implemented by routing the inputs from the neighbors to each processor through a multiplexer that is controlled locally. Figure 4 illustrates the freedom to select a communication partner provided by connection autonomy. Examples of machines in this category include the Polymorphic-Torus architecture [17], the Reconfigurable Processor Array (RPA) [26], the CAAPP layer of the Image Understanding Architecture (IUA) [31], and CLIP7 [12, 13]. Operations on the connected components of an image or a graph can take advantage of this level of autonomy efficiently by linking each node to its component-neighbor, which could differ from node to node. Maresca and Li have compared the complexity of tree-embedding on meshes, hypercubes, tree architectures, and the polymorphic-torus [18]. The mesh connected polymorphic-torus was found to have lower propagation delay between two nodes of the embedding than even hypercubes due to its connection autonomy.

Algorithms that require connection autonomy can be implemented on architectures without it by cycling through the distinct neighbors sequentially for each communication step, incurring a worst case slowdown by a factor equal to the maximum number of neighbors of any PE, i.e., the maximum degree of the interconnection graph.

3.4 Communication autonomy

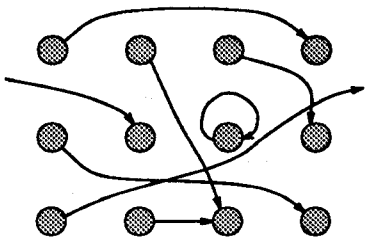

Figure 5: Communication autonomy allows general communications

In a communication autonomous machine, the address of the source/destination in a general communication operation is specified in each PE by a local variable. Connection autonomy and communication autonomy differ only in the scale and the range of the communications they refer to. Connection autonomy is the freedom to choose locally one of the few immediate neighbors; communication autonomy is the independence to select locally one of the many PEs that are reachable from the given one in a general communication operation. Figure 5 shows a general communication pattern possible on communication autonomous architectures. Without communication autonomy, each PE has to use the same rule to select the partner of a long-distance communication operation, such as: Send to the PE 10 locations to the left or send to the PE that differs in the 7^{th} bit. A general purpose routing mechanism is necessary to implement communication autonomy on all interconnection topologies except complete graphs (where communication autonomy is synonymous with connection autonomy). Thus, the autonomy is usually a virtual one and the time for a general communication operation may depend on the particular pattern of communication. Communication autonomy is an important algorithmic concept nevertheless. Hence, we find even machines with no connection autonomy possessing communication autonomy, though the latter appears to be a generalization of the former. Richer physical interconnectivity, however, results in an efficient implementation of communication autonomy. Examples of machines with communication autonomy include the Connection Machine CM-2 [1] and the MasPar MP-1 [4]. Communication autonomy enables the machines to implement general graph algorithms efficiently. Graphs with irregular structures can be processed efficiently only on machines with communication autonomy.

Algorithms that require communication autonomy can be implemented on architectures that lack it by serializing the communication for each combination of distinct source-destination pairs. The slowdown will depend on the specific pattern of communication.

3.5 Addressing autonomy

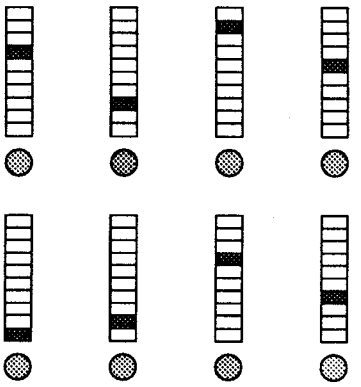

Figure 6: Addressing autonomy permits operands to be locally selected

In this category of SIMD machines, local memory operands can be addressed using variables residing in the local memory, i.e., the PEs possess local indirect addressing capability. Addressing autonomy is independent of connection or communication autonomies. Figure 6 illustrates how each PE could address a different location in its local memory at the same time. Addressing autonomous machines require a hardware mechanism to modify the memory address locally in every processor, thus making Figure 1 somewhat

incorrect[1]. Examples of addressing autonomous machines include MasPar MP-1 [4, 23] and CLIP7 [12].

Algorithms that require addressing autonomy can be implemented on other architectures by cycling through the distinct memory addresses in each instruction cycle. The slowdown will depend on the specific addressing pattern, with a worst case slowdown of the order of *min*(Size of local memory, Number of PEs). Machines that cannot cycle through a dynamic list of addresses suffer this slowdown in each instruction. We provide detailed examples demonstrating the algorithmic advantage of addressing autonomous machines over machines without addressing autonomy in Section 4.

3.6 Operation autonomy

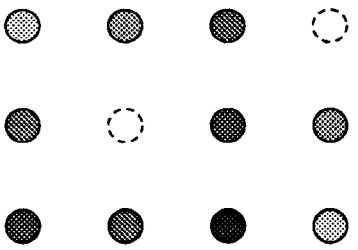

Figure 7: Operation autonomy permits different operations on each PE

Operation autonomy refers to the (limited) independence of individual PEs in chosing the operation to be performed. The level of autonomy does not make the machines MIMD in nature. Thus, the individual PEs are not equipped with full program memories and instruction sequencers. Figure 7 illustrates how the operation is locally selected in this class of architectures. The controller issues the opcode for a generic class of instructions. Each PE selects an instance of the class locally based on its internal state by, say, plugging the last few bits of the instruction broadcast by the controller. Thus, for example, the controller may issue a general logical instruction; each PE can choose for itself from logical AND, OR, EXOR, NAND, or NOR. In an alternate implementation, the controller could broadcast multiple parallel instructions on multiple instruction buses simultaneously in each cycle. Each PE selects one of the instructions for execution based on a local condition. These schemes can be considered to be a generalization of activity control to a multiway choice of (similar) operations based on each PE's internal state. The iWarp processor array is almost full MIMD capable and can simulate operation autonomy efficiently [5]. The Multi-SIMD (MSIMD) architecture of PASM [27] and the Single Program Multiple Data (SPMD) mode of CM-5 [28] represent various levels of operation autonomy. The processing elements of the BLITZEN massively parallel processor array have a locally loadable register whose contents select one of two complementary operations locally [7]. The normalization of results in floating point operations can benefit from operation autonomy as different processors can shift the mantissa left or right depending on its magnitude and can either increment the exponent or decrement it. Operation autonomy is helpful in implementing multistage pipelined algorithms with different regions of the array acting as different stages. The additional power operation autonomy needs to be explored further.

Operation autonomous algorithms can be implemented on other architectures by cycling through all instructions executed by at least one PE in each cycle. This slows down the execution by a worst-case factor of the number of different instructions the PEs can execute in an instruction cycle.

[1]Ken Batcher, however, questioned the applicability of the name SIMD to architectures without local independent addressing as early as in 1986. He argued that architectures without indirect addressing were SISD [3].

4 Addressing autonomy on SIMD machines

In this section, we explore the category of addressing autonomous SIMD machines. It is feasible to implement addressing autonomy on processor arrays under today's technology as is demonstrated by MasPar machines. We provide examples in this section that demonstrates the added power of addressing autonomy, both asymptotically and practically. We compare three operations on processor arrays with and without addressing autonomy. In all three cases, the machine is assumed to have activity control. Section 4.1 discusses independent merging, Section 4.2 discusses independent sorting, and Section 4.3 discusses independent searching. A short discussion on how addressing autonomy can help virtual processing on SIMD processor array appears in the next section.

4.1 Independent merging

In this section, we address the following problem for an SIMD processor array: How can we merge two sorted lists in the local memory of a processing element of the array, independent of the other PEs? Each PE operates on its local memory and no communication takes place. Independent merging is useful in the independent sorting operation discussed later and in other operations that need large lists to be stored and manipulated in each processing element. Many parallel programs have portions in which independent sequential processing is performed by each PE. Independent merging is as important to this phase as merging is to sequential computing.

The sequential merging algorithm has a time complexity linear in the length of the merged list. It serves as the basis of the independent merging algorithm and is sketched here. The algorithm given below merges the lists of numbers A and B of lengths l and m respectively, sorted in ascending order, and stores the result in the list C.

Program *SequentialMerge*

$j := 1,\ k := 1$

for $i := 1$ to $l + m$ do

if $(j \leq l)$ AND $(A[j] \leq B[k])$

then $C[i] := A[j],\ j := j + 1$

else $C[i] := B[k],\ k := k + 1$

End *SequentialMerge*

The above algorithm will perform independent merging on addressing autonomous SIMD machines if j and k are independent local pointers. The *then* part will be executed on PEs that satisfy the condition and the *else* part on others sequentially. Addressing autonomy enables the pointers to advance independently in each PE. Thus, independent merging can be performed in $O(l + m)$, or linear, time on SIMD machines with addressing autonomy.

On machines with no addressing autonomy, pointers must advance in synchrony in all PEs. In the worst case, all elements of list A lie between elements i and $i + 1$ of list B in some PE, for a different i. However, there are only $m + 1$ possible i values. Thus, independent merging is an $O(l\,m)$, or quadratic, time operation in the worst case on SIMD machines with no addressing autonomy, whatever be the algorithm. This asymptotic performance can be achieved by repeatedly inserting the elements of the shorter list into the longer list, which is an easy algorithm to implement.

Independent Merge Timing (in seconds) for 8-bit random integers			
List Length	Without autonomy	With autonomy	Speedup
32	0.02	0.01	2.0
64	0.09	0.01	9.0
128	0.34	0.02	17.0
256	1.11	0.05	22.2
512	4.21	0.12	35.1
1024	57.18	0.28	204.2
1536	117.40	0.43	273.0
2048	184.96	0.53	349.0
3072	425.43	0.72	590.9
4096	757.42	0.72	1052.0
6144	1133.59	1.02	1111.4

Table 1: Timing of independent merging with and without addressing autonomy on a MasPar MP-1

The running times of independent merging with and without addressing autonomy is shown in Table 1, on an 8K processor MasPar MP-1. Merging without addressing autonomy was implemented by repeatedly inserting the elements of one list into the other. Figure 8 plots the speedup of the addressing autonomous algorithm over the non-autonomous algorithm for various list lengths. As can be observed from the table, addressing autonomy improves the performance of independent merging by orders of magnitude.

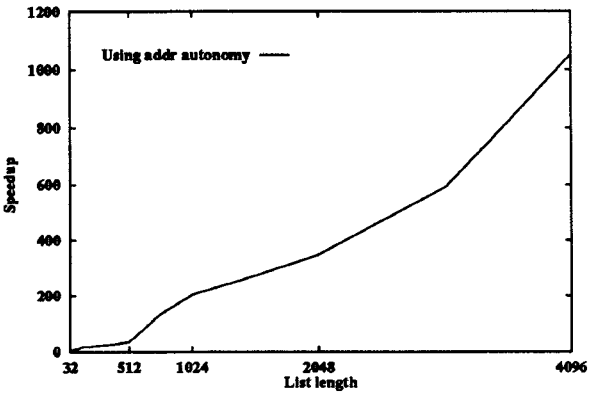

Figure 8: Speedup gained by addressing autonomy for independent merging on a MasPar MP-1

4.2 Independent sorting

In this section, we study the problem of independent sorting on an SIMD machine. The problem is to sort lists of numbers stored in the local memory of each PE independently in each processor. No communication takes place in this operation. Independent sorting is an important operation in the computation of rank based local parameters in image processing such as median and rank order filters. While sorting a large list using a processor array, the data needs to be partitioned among the PEs. The sublists are first sorted within each PE. These sorted partitions are later merged across all PEs to yield the large sorted list. The first step of sorting within each PE is an independent sorting operation.

On an addressing autonomous SIMD machine, a list of length n can be sorted in $O(n \log n)$ time using either the heapsort algorithm or the mergesort algorithm. A bottom up mergesort algorithm is presented below to sort the list A containing n elements.

Program *MergeSort*

RunSize := 1

while (RunSize $\leq n/2$) do

for i := 1 to n/RunSize step 2 do

MergeLists(A[$(i - 1)$ * RunSize],

A[i * RunSize], RunSize)

RunSize = 2 * RunSize

End *MergeSort*

Without addressing autonomy, each merge is quadratic in the length of the lists. The running time of the sorting algorithm then becomes $\frac{n}{2}1 + \frac{n}{4}2^2 + \frac{n}{8}4^2 + \cdots + 2(\frac{n}{2})^2 = \frac{n}{2}(1 + 2 + 4 + 8 + \cdots + n) = \frac{1}{2}n(2n - 1) = O(n^2)$. Simple sorting algorithms such as bubble sort and insertion sort also have $O(n^2)$ running time on machines without addressing autonomy. Bubble sort is easy to implement and is faster in practice.

Independent Sort Timing (in seconds) for 8-bit random integers			
List Length	Bubble Sort	Merge Sort	Speedup
32	0.01	0.02	0.5
64	0.03	0.04	0.8
128	0.13	0.12	1.1
256	0.43	0.22	2.0
512	1.52	0.52	2.9
1024	5.51	1.11	5.0
1536	12.01	1.81	6.6
2048	20.81	2.31	9.0
3072	45.51	3.81	11.9
4096	79.61	5.01	15.9
6144	176.31	8.11	21.7
8192	310.73	8.61	36.1

Table 2: Running times of different independent sorting algorithms on a MasPar MP-1

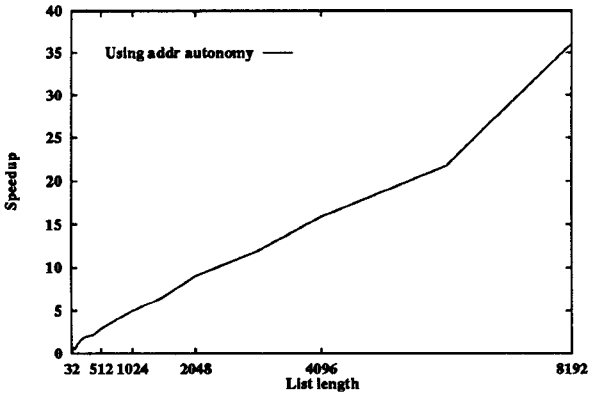

Figure 9: Speedup gained by addressing autonomy for independent sorting on an MP-1

The running times of independent sorting with and without addressing autonomy is shown in Table 2, on an 8K processor MasPar MP-1. Figure 9 plots the speedup of the addressing autonomous algorithm over the non-autonomous algorithm for different list lengths. It can be observed that addressing autonomy greatly improves the performance of independent sorting. Even though merging is a step in the algorithm, the performance of sorting is less spectacular. The simple and tight code of bubble sort makes up for the algorithmic advantage provided by the merge sort, especially on shorter lists. This brings out another important aspect of SIMD programming: Simplicity of the controller code is critical to a fast implementation.

4.3 Independent searching

In this problem, each PE searches for a given element in a list stored in its local memory, independently of other PEs. This problem can arise in a number of situations. For instance, assume that a graph is mapped to the processor array such that each PE represents a vertex and stores the adjacency information (as an adjacency matrix or an adjacency list) and other relevant information for that vertex. In graph algorithms, it is common for a node to select a neighbor that satisfies a particular condition for a parameter value for communication/processing. For example, every vertex might want to identify a neighbor with a label given by a local variable. This can be done by searching a sorted list containing the parameter values (the vertex labels sorted for the neighborhood in the example) of all neighbors.

On address autonomous SIMD machines, we can use binary search to search for a specific element in a sorted list, as the pointers can advance independently in each PE. Without addressing autonomy, the pointers have to move in unison and we cannot use any searching algorithm more intelligent than sequential searching. Thus independent sorting is an $O(\log n)$ operation on addressing autonomous machines and an $O(n)$ operation without addressing autonomy, where n is the length of the list.

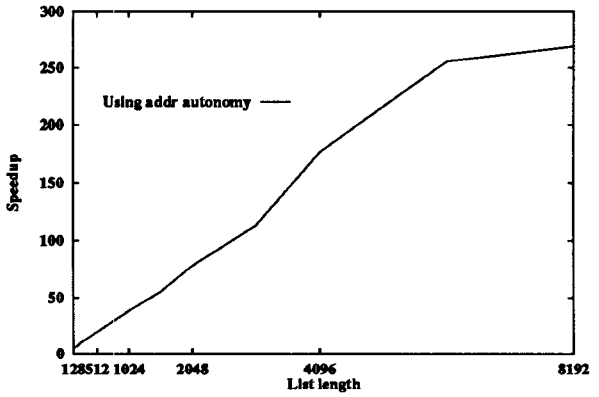

Figure 10: Speedup gained by addressing autonomy for independent searching on an MP-1

Table 3 compares the time for independent searching with and without addressing autonomy on a MasPar MP-1. The

Independent Search Times (in milliseconds) for 8-bit random integers			
List Length	Without autonomy	With autonomy	Speedup
128	2.66	0.55	4.85
192	4.32	0.59	7.37
256	6.13	0.59	10.46
384	9.13	0.61	14.97
512	11.72	0.60	19.60
768	17.38	0.60	29.07
1024	24.12	0.62	38.72
1536	33.69	0.61	55.23
2048	45.80	0.59	78.16
3072	67.58	0.60	113.01
4096	103.22	0.59	176.15
6144	149.81	0.59	255.64
8192	160.84	0.60	268.96

Table 3: Running times of different independent searching algorithms on a MasPar MP-1

speedup of the addressing autonomous algorithm over the non-autonomous one is plotted in Figure 10 for different list lengths. Again, we observe orders of magnitude improvement in performance using addressing autonomy.

5 Processor autonomy and SIMD machines

In this section, we point out the use of processor autonomy in a few algorithms in the literature. We also discuss how processor autonomy can have an impact on some of the architectural studies being carried out on SIMD machines.

5.1 Algorithms from the literature

Tomboulian and Pappas discuss how local indirect addressing can be used to achieve near mean convergence rates for the computation of Mandelbrot sets on a MasPar MP-1 [29]. Computation of Mandelbrot sets belongs to the class of problems where identical computation is performed on a number of data points, but the convergence rate is data dependent. With indirect addressing, the computation can proceed to the next data point upon convergence. Thus, mean convergence rates can be achieved on sufficiently large problems. Another application of indirect addressing on SIMD processor arrays to improve the processing of recursive neighborhood operations can be found in [15].

The data replication technique to speed up the processing of small data structures on large processor arrays combines operation parallelism with data parallelism. This is done by dividing the computation among multiple copies of the data structure. For the different copies to perform different useful computations, processor autonomy is essential. Replicated data algorithms utilizing communication autonomy and addressing autonomy are present in the literature [19, 20, 21].

It is common to map a vertex of the graph to a PE while processing graphs in parallel. Many problems require accessing one of the neighbors of the graph based on the present state of the algorithm and a few features of the local vertex. Thus, different vertices access different neighbors. For instance, in a parallel implementation of the traveling salesman problem, the graph representing the cities and the cost of traveling between them is stored in a distributed fashion in a processor array machine. The cost of traveling up to a number of vertices is available at every stage of the algorithm. The algorithm proceeds by adding another city to each of the paths found so far. Each vertex will need to access a different neighbor, no matter how we store the neighborhood information of the graph. To process the graphs efficiently (without serializing the access at any stage), a PE should be able to address its local memory independently. Thus address autonomy is essential to the efficient implementation of such algorithms.

Addressing autonomy facilitates the use of local memory allocation and management in each PE. This could be important to many applications in which each PE handles a non-uniform amound of data. All PEs must allocate memory equal to the largest chunk on machines without addressing autonomy, resulting in inefficient use of the memory.

5.2 A note on virtual processing

Although large processor arrays are available today, problems involving large data sets also are common. While solving a problem involving data structures larger than the machine, the data can be processed in chunks that fit into the machine (cut and stack) or the machine can be "enlarged" to fit the data. The latter, called *virtual processing*, is implemented by making each physical processing element simulate multiple virtual ones. The number of virtual PEs simulated by each physical PE is known as the *virtual processing ratio (VP ratio)*.

Virtual processing is typically implemented by partitioning the memory of each PE among the virtual PEs it simulates. Each instruction cycle is split into subcycles that execute the instruction on one memory partition. This fits the SIMD model without addressing autonomy if the partitioning is done uniformly as every PE will be executing the same instruction on operands at the same memory address in each subcycle. The number of virtual processors should be a multiple of the number of physical processors. Every instruction will be slowed down by a factor equal to the VP ratio.

In communication instructions, the address of the destination PE denotes both a physical PE and a memory partition within it. Unless orchestrated carefully, general communications and near neighbor communications will need

to access different partitions of the local memory in different PEs while virtual processing. While the time to store the received value in the local memory is a constant on machines with addressing autonomy, it could take a worst case time equal to the VP ratio on machines with no addressing autonomy. Machines with addressing autonomy can also speedup the conditional instruction execution when only a small number of PEs are active. Different PEs can operate on different partitions of their local memory, not wasting time on inactive virtual PEs. This reduces the slowdown due to virtualization from *VP ratio* to the maximum number of active virtual PEs in any physical PE.

5.3 Simulating MIMD model on SIMD machines

The question of simulating an MIMD model of computation on an SIMD hardware has received considerable attention recently [24]. Many researchers are of the opinion that it is easier and more cost effective to build SIMD style parallel hardware than MIMD ones. Bridges et al. define a measure of CPU utilization and establish that massively parallel SIMD computers utilize their CPU better than MIMD computers [6]. The MIMD emulator of Dietz and Cohen achieved 25% of the peak machine performance on a MasPar MP-1 at a fraction of the cost of a comparable MIMD computer [9]. They expect to achieve 50% performance in the next version of the emulator. Wilsey et al. demonstrate how MIMD performance can be extracted out of an existing SIMD hardware and suggest minor hardware modifications to boost the performance of MIMD code [32].

Emulating an MIMD machine on an SIMD architecture is done by storing the program in the local memory of each PE. The PE acts like a fetch-and-execute unit that executes the code in its local memory. It is easy to see how processor autonomy can ease this task. Dietz and Cohen [9] and Wilsey et al. [32] conclude that the independent addressing, i.e. the addressing autonomy, of MasPar MP-1 critical to their simulator. Wilsey et al. also suggest that register indirect addressing capability will further enhance the performance of their MIMD simulator.

Operation autonomy takes SIMD machines closer to the MIMD model and can help simulate such a model. Common subexpression induction is a technique that transforms MIMD code into code that is more amenable to SIMD processing [8]. Limited operation autonomy described in Section 3.6 can eliminate the need for common subexpression induction in some cases. On an SIMD machine where the controller issues a generic logical instruction with the specific operation selected locally, there is no need to transform logic instructions to fit the SIMD paradigm as all logical instructions can be executed simultaneously. Wilsey et al. recommends having multiple controllers to issue multiple instruction streams, with each PE selecting one locally. This also is a form of operation autonomy, as mentioned in Section 3.6.

6 Conclusions

In this paper, we examined the processor autonomy of the SIMD class of machines and categorized the SIMD class into six subclasses on its basis. These subclasses differ importantly from one another as we demonstrated using code that distinguish each type of autonomy. We also provided examples of machines belonging to each category and discussed how autonomies can be simulated on machines without them.

The study of the capabilities of the autonomy classes is important to the design of future high performance processor arrays. We demonstrated the usefulness of addressing autonomy using several examples; MasPar machines demonstrate the feasibility of providing addressing autonomy. Designers of the processor arrays of the future need to take a serious look at these results.

The machines with operation autonomy are interesting and need to be studied in detail. The cost of (limited) local selection of the instruction to be executed is not significant, particularly if the PEs are horizontally microcoded. Several issues related to it need addressing. How can we design the generic (issued by the controller) and specific (selected locally) instruction set for such a machine? How can we design algorithms that effectively use the operation autonomy? Will this produce a model of computation that is a good mix of the SIMD and MIMD models at a significantly reduced cost?

Acknowledgements: The author wishes to thank Prof. Larry Davis of the University of Maryland, College Park for many discussions on this topic. Author also thanks the NASA Goddard Space Flight Center, Greenbelt, MD for the access to their MasPar MP-1 machine.

References

[1] Thinking Machines Corporation. Connection Machine: Model CM-2 Technical Summary. Technical report, Thinking Machines Corporation, May 1989.

[2] K. E. Batcher. The Design of a Massively Parallel Processor. *IEEE Transactions on Computers*, 29:836–840, 1980.

[3] K. E. Batcher. The Architecture of Tomorrow's Massively Parallel Computer. In J. Fischer, editor, *Proceedings of the First Symposium on the Frontiers of Massively Parallel Scientific Computing*, pages 151–157. NASA, September 1986.

[4] T. Blank. The MasPar MP-1 Architecture. In *Proceedings of the 35th IEEE Computer Society International Conference*, pages 20–24, 1990.

[5] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. S. Tseng, J. Sutton, J. Urbanski, and J. Webb. iWarp: An integrated solution to high-speed

parallel computing. In *Proceedings of the Supercomputing Conference*, pages 330–339, 1988.

[6] T. Bridges, S. W. Kitchel, and R. M. Wehrmeister. A CPU Utilization Limit for Massively Parallel MIMD Computers. In *The Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 83–92, 1992.

[7] E. W. Davis and J. H. Reif. Architecture and Operation of the BLITZEN Processing Element. In *Proceedings of the Third International Conference on Supercomputing*, 1990.

[8] H. G. Dietz. Common Subexpression Induction. Technical Report TR-EE 92-5, School of Electrical Engineering, Purdue University, January 1992.

[9] H. G. Dietz and W. E. Cohen. A Massively Parallel MIMD Implemented by SIMD Hardware. Technical Report TR-EE 92-4, School of Electrical Engineering, Purdue University, January 1992.

[10] M. J. B. Duff and T. Fountain. Enhancing 2-D Meshes. In *Proceedings of the International Conference on Pattern Recognition*, pages 654–659, 1990.

[11] M. J. Flynn. Very High Speed Computing Systems. *Proceedings of the IEEE*, 54(12):1901–1909, Dec. 1966.

[12] T. Fountain. *Processor Arrays: Architecture and Applications.* Academic Press, London, 1987.

[13] T. J. Fountain. The Development of the CLIP7 Image Processing System. *Pattern Recognition Letters*, 1:331–339, 1983.

[14] T. J. Fountain. Introducing Local Autonomy to Processor Arrays. In H. Freeman, editor, *Machine Vision: Algorithms, Architectures, and Systems*, pages 31–56. Academic Press, San Diego, 1988.

[15] E. R. Komen. Efficient Parallelism Using Indirect Addressing in SIMD Processor Arrays. *Pattern Recognition Letters*, 12:279–289, 1991.

[16] S. Y. Lee and J. K. Aggarwal. Parallel 2-D Convolution on a Mesh Connected Array Processor. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 9:590–594, 1987.

[17] H. Li and M. Maresca. Polymorphic-torus architecture for computer vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11:233–243, 1989.

[18] M. Maresca and H. Li. Connection Autonomy in SIMD Computers: A VLSI Implementation. *Journal of Parallel and Distributed Computing*, 7:302–320, 1989.

[19] P. J. Narayanan. Analysis of Replicated Data Algorithms on Processor Array Architectures. In *Proceedings of Supercomputing '91*, Albuquerque, 1991.

[20] P. J. Narayanan. Single Source Shortest Path Probelm on Processor Arrays. In *Proceedings of The Fourth IEEE Symposium on the Frontiers of Massively Parallel Computing*, 1992.

[21] P. J. Narayanan and L. S. Davis. Rank Order Filtering on SIMD Machines. In *Proceedings of the 11th International Conference on Pattern Recognition*, The Hague, August 1992.

[22] D. Nassimi and S. Sahni. Finding Connected Components and Connected Ones on a Mesh Connected Parallel Computer. *SIAM Journal of Computing*, 9(4):744–757, November 1980.

[23] J. R. Nickolls. The Design of the MasPar MP-1: A Cost Effective Massively Parallel Computer. In *Proceedings of the 35^{th} IEEE Computer Society International Conference*, pages 25–28, 1990.

[24] M. Nilsson and H. Tanaka. MIMD Execution by SIMD Computers. *Jouran of Information Processing*, 13(1):58–61, 1990.

[25] S. F. Reddaway. DAP - A Distributed Array Processor. In *First Annual Symposium on Computer Architecture*, pages 61–65, 1973.

[26] A. Rushton. *Reconfigurable Processor Array: A Bit-sliced Parallel Computer.* MIT Press, Cambridge, MA, 1989.

[27] H. J. Siegel. *Interconnection Networks for Large Scale Parallel Processing: Theory and Case Studies.* Lexington Books, Lexington, MA, 1984.

[28] Thinking Machines Corporation. Connection Machine CM-5 Technical Summary. Technical report, Thinking Machines Corporation, May 1989.

[29] S. Tomboulian and M. Pappas. Indirect Addressing and Load Balancing for Faster Solution to Mandelbrot Set on SIMD Architectures. In J. Ja'Ja', editor, *The Third Symposium on the Frontiers of Massively Parallel Computation*, pages 443–450, College Park, Maryland, October 1990.

[30] J. D. Ullman. *Computational Aspects of VLSI.* Computer Science Press, Rockville, MD, 1984.

[31] C. C. Weems, S. P. Levitan, A. R. Hanson, and E. M. Riseman. The Image Understanding Architecture. *International Journal of Computer Vision*, 2(3):251–282, January 1989.

[32] P. A. Wilsey, D. A. Hensgen, N. B. Abu-Ghazaleh, C. E. Slusher, and D. Y. Hollinden. The Concurrent Execution of Non-communicating Programs on SIMD Processors. In *The Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 29–36, 1992.