Virtual Parallelism Allows Relaxing the Synchronization Constraints of SIMD Computing Paradigm

M. Migliardi♠ , P. Baglietto♥
DIST - University of Genoa
via Opera Pia 11a - 16145 Genoa, Italy
Tel. +39-10-353.2709 Fax +39-10-353.2154

and M. Maresca
DEI - University of Padova
Via Gradenigo - 35131 Padova, Italy
Tel. +39 49 827.7820 Fax +39 49 827.7826

Abstract

In this paper we propose to introduce execution autonomy in the SIMD paradigm to overcome its rigidity while preserving the advantages of its synchronous programming model and we show that Virtual Parallelism support is a necessary condition to the profitable application of execution autonomy. We define execution autonomy as the capability of each processing element of a massively parallel computer to execute the instructions in a block of code of a single common program autonomously and asynchronously. We define virtual parallelism as the capability to emulate a n processors array on a m processor array with n/m performance degradation. In past related works the relaxation of SIMD synchronization has been already proposed, nevertheless its relation with Virtual Parallelism has never been studied.

keywords: *SIMD, Computational Paradigms, Execution Autonomy, Virtual Parallelism, Data-parallelism.*

1 INTRODUCTION

The SIMD computational model has progressively lost its popularity mainly because of its rigidity. In particular the following two implementation constraints limit the performance given by computers based on the SIMD paradigm. First the same clock signal has to be distributed to all the Processing Elements (PE). This fact introduces problems of crosstalk and clock skewing, and forces the whole computer to adopt a

♠ Contact author: E-mail om@dist.unige.it

♥ Presenting author: E-mail prp@dist.unige.it

low frequency clock. Second, the need to broadcast each single instruction from the central controller to the PEs limits once again the clock frequency.
However SIMD massively parallel computers have demonstrated their suitability for a number of computing tasks such as graph analysis, numerical analysis and computer vision. Besides, it is much easier to extract massive parallelism by means of data-parallelism than using control-parallelism and a single program that is executed in a synchronous fashion on many PEs makes the task of developing and debugging parallel algorithms on an SIMD machine by far simpler than it is on an MIMD machine where many different programs interact asynchronously.
These facts have lead to two considerations:

- it is possible to separate the ***data-parallel*** programming model [4] from the SIMD computational model;
- the SIMD implementation of the data-parallel programming model is ***over synchronized***.

In past related works it is possible to observe three different tendencies.
In order to improve the SIMD computational model, a first approach has been to propose three different PEs autonomies [7], namely operation autonomy (OA), addressing autonomy (AA) and connection autonomy (CA). OA provides the massively parallel architecture with the capability of executing more than one operation among the PEs; we can cite as examples of OA the activity bit of the Polymorphic Torus and the multi-bit register of the CLIP7 system. AA allows each PE to independently generate the data memory address or to independently modify the data address broadcast from the central controller. We can cite as examples of massively parallel SIMD computer with this capability the MasPar MP machines. CA allows the massively parallel SIMD computer to modify the configuration of the underlying inter-processor communication network at execution time in order to optimally match the communication patterns of the algorithm; examples of massively parallel computational models with CA are the Polymorphic Processor Array [6] and the Mesh with Reconfigurable Busses[8].
In order to exploit the advantages of the data-parallel programming model without mapping it on the SIMD computational model a second approach has been to go in the direction of mapping it on a different computational model. Both the definition of the SPMD computational model [9], the CM-5 system from the Thinking Machines Corp. [5] can be classified in this stream.
Finally, following a third approach, other scientists have proposed to introduce a limited degree of relaxation of the SIMD paradigm synchronization in such a way as to keep it invisible to the machine users [1][10][12].
In our opinion the first approach has only partially solved the problems previously mentioned in that it proposes enhancements that do not deal with the implementation problems of the clock distribution and the instruction distribution. At the same time the second approach has lead to computational models that are often either biased toward MIMD or MIMD altogether and that discard positive aspects of the SIMD computational model like its simplicity of architecture, hardware and programming style. In order to preserve these positive aspects, it is necessary to stick to the third approach, nevertheless past works have not formalized

	SIMD (e.g. MP-1)	SBMD	SPMD (e.g. PASM
Programming Model	Implicitly Synchronous	Implicitly Synchronous	Usually Expli Synchroniz
Program Control Flow	Global	Local	Local
CPU Architecture	Very Simple. Many on a single Chip.	Simple. Many on a single Chip	Usually off-the CPUs
Communication	Completely Synchronous	Block Synchronized	Barrier Synchro
Execution Level	Synchronous	Asynchronous	Asynchrono

Table 1 Features Summarize

the programming feasibility of this approach and have not studied its relation with Virtual Parallelism.
Thus we propose to introduce a fourth kind of autonomy for SIMD computers that we name Execution Autonomy (EA) [1], and we study its relationship with the concept of Virtual Parallelism. EA allows each PE of a massively parallel computer to execute the instructions in a block of code of a single common program autonomously and asynchronously. We categorize this enhanced SIMD model between the classical SIMD computational paradigm and the SPMD computational paradigm, and we call it Single Block of instructions Multiple Data (SBMD) paradigm [1] [1]. As it is pointed out in table 1 the SBMD computational paradigm is different from both the traditional SIMD and the SPMD computing paradigm.
In the next section of the paper we explore in more depth the model proposed defining it in a formal way and showing its requirements; in section three we study its architectural requirements; in section four we formalize its programming requirements and we propose a solution to those programming requirements; finally, in section five we provide some concluding remarks.

2 REQUIREMENTS OF THE SBMD COMPUTATIONAL MODEL

On an SIMD machine the time **T** necessary to execute **n** instructions can be calculated using the following formula:

$$\mathbf{T = n\ (t1 + t2)} \qquad \mathbf{(1)}$$

where **n** is the number of instructions executed, **t1** is the time needed to broadcast a single instruction and **t2** is the time needed to execute a single instruction
The technological solutions affordable in past years were such as to make $\mathbf{t_1} \sim= \mathbf{t_2}$, thus the necessity of broadcasting an instruction at each clock cycle did not constitute a bottleneck in the general machine architecture. Besides, the similarity

[1] It is interesting to observe that the Data Flow computational paradigm has gone through a similar process evolving from early single instruction Data-Flow machines like the Manchester Data Flow or Sigma-1, to macro-block Data-Flow machines like EM-4 [11th].

between the values of $\mathbf{t_1}$ and $\mathbf{t_2}$ made possible to pipeline broadcasting and execution of a single instruction, allowing to overlap the two activities in order to achieve the result of an execution time given by the following formula:

$$\mathbf{T = n * max(t1 , t2)} \qquad (2)$$

In these years the CPUs have quickly evolved both in architectural and in technological aspects becoming by far faster than before. This development has made the assumption $\mathbf{t_1}$ ~= $\mathbf{t_2}$ very distant from the reality, in fact it is quite common nowadays to have $\mathbf{t_1}$ at least one order of magnitude larger than $\mathbf{t_2}$. As an example the clock frequency of the MasPar MP2 system is 12.5MHz, while the clock frequency of current RISC systems is about 600MHz.

Thus it would be very desirable to eliminate the direct proportionality to the $\mathbf{t_1}$ factor from the calculation of the time necessary to execute **n** instructions.

In order to achieve this result it is important to notice that in a sequence of **n** instructions some instructions are executed more than once (e.g. loops), thus in a sequence of **n** instructions there are only **m** different instructions (with $\mathbf{n} \geq \mathbf{m}$).

Definition 1

*We define a **block of m instructions** any slice of the compiled program containing m instructions.*

Definition 2

*We define **reusability factor** of a compiled program the value of the ratio n/m averaged over any block of m instructions.*

If we broadcast a whole block of **m** instructions with a **reusability factor** of n/m before starting to execute it we obtain an execution time that can be calculated with the following formula:

$$\mathbf{T = m*t1 + n*t2} \qquad (3)$$

If we pipeline the broadcasting of a block and its execution, the previous formula can be transformed in:

$$\mathbf{T = max(m*t1 , n*t2)} \qquad (4)$$

Given that usually **n** is larger than **m** and $\mathbf{t_1}$ is larger than $\mathbf{t_2}$, we want to achieve the result of having $\mathbf{n*t_2} > \mathbf{m*t_1}$. In this latter case the formula can be simplified to:

$$\mathbf{T = n*t2} \qquad (5)$$

This last result demonstrates that the computing speed of the machine can be made limited by the single PE computing speed and no more by the time required to broadcast the instructions. As a matter of fact if the number of executed instructions can be made large enough then the time required to broadcast to the PEs the instructions results completely hidden. Thus, any increase in the PEs computing speed due to a faster clock or a better PE architecture will result in increased machine performance.

In order to make this promising result usable and profitable, it is necessary to fulfill the following requirements:

a) $t2 < t1$
b) $n > m$
c) $n*t2 > m*t1$.

Name	Brief Program Description	max BB	mean BB	min BB
p3pack	Matrix operations	13	4.3	1
p3wrap	Matrix operations	2	1.1	1
spfft	Data Parallel FFT	19	3.06	1
fftwap	FFT support and test	1	1	1
mpltest	FFT support and test	5	1.39	1
realtest	FFT support and test	5	1.28	1
dynamics	Molecular dynamics simulation	87	4.81	1
p_convolve	Snake sweep convolution	5	2	1
jplfdct	JPEG Discrete Cosine Transform	180	119.67	1
jplquant	JPEG Quantization of DCT coefficients	5	1.7	1
phuff	JPEG Huffman coding	32	3.72	1

Table 2 Programs used to measure the dimension of basic blocks in data parallel programs and maximum, minimum and mean dimension of their basic blocks.

3 PE EXECUTION AUTONOMY

One of the main obstacles that prevents an SIMD machine from having a short clock cycle is the necessity of broadcasting the clock signal to every single PE of the machine. In fact at higher clock frequencies the phenomena of crosstalk and clock skewing become evident.

We propose as a solution to this problem to enhance the SIMD computing paradigm with **execution autonomy**.

Definition 3

*We define **m-execution autonomy** the capability of the PEs of a massively parallel system to execute a block of m instructions without any need to synchronize among each other.*

The concept of m-execution autonomy is quite important; in fact its length corresponds to the granularity of autonomous and asynchronous execution allowed in the machine.

In a machine with **m-execution autonomy** any PE can work using a locally generated clock signal for the number of consecutive clock cycles necessary to complete the execution of a block of m instructions. Thus it is possible to make $\mathbf{t_2} < \mathbf{t_1}$ and to fulfill requirement **a)**.

Improving SIMD computers with EA requires to introduce a local instruction memory and a local program counter in each PE to allow the independent execution of block of instructions and subprograms previously received from the central controller. The program is still the same for all the PEs but the control flow can be different in each PE depending on the local data flow and the conditional instructions. Thus EA allows each PE to use a clock signal locally generated instead of a clock signal generated in the program controller and then distributed.

Besides, the use of EA and the support for the SBMD paradigm overcomes the rigidity of the traditional SIMD computational model allowing to avoid to have large sets of PEs inactive during the execution of conditional statements.

A local program counter and the independent execution of subprograms allow to overlap the local execution of the instructions with the distribution of the instructions and of the data broadcast by the central controller. Instructions and data broadcast can be overlapped to the instruction execution through memory interleave. The local memory can be divided into two banks alternatively used locally during instruction execution and for I/O operations with the central controller and the external memory. The main additional cost of EA enhanced PEs with respect to traditional SIMD PEs is the presence of a local program memory. This memory needs to be large enough to store the m instructions that each PE is able to execute asynchronously. Nevertheless, this memory is not large, in fact the basic blocks of data parallel programs are small. In order to have a quantitative analysis of this phenomenon we measured the number of instruction of basic blocks of several programs implementing a set of tasks on a MP-1 machine. In table 2 you can see a list of the program we measured with a brief description of each one. We define a basic block of a data parallel program any block of instructions containing only PE instructions and no instructions that need to be executed by the sequential controller or communication primitives. In fact each inter-PE communication primitive forces a synchronization among the PEs. In table 2 we can see the dimension measured in instructions of the largest and smallest basic block of each program as well as the mean number of instructions of each program basic blocks. As table 2 shows, in all but one of the programs, these values are smaller than one hundred; this fact means that in a massively parallel SIMD architecture such as the MP-2, common IPPR programs do not execute more than one hundred parallel PEs instructions consecutively, without a sequential controller instruction or a call to a communication primitive. It is also interesting to notice that the smallest basic block is always just one instruction long. This means that, in every program, it is possible to find at least on occurrence of at least two consecutive sequential controller instructions or calls to a communication primitive.
The measurements we made seem to suggest that, if we adopt a local instruction memory dimensioned in such a way as to contain one hundred instructions, the machine is almost always able to store a program basic block inside the PEs. The ability to never split a single basic block is important because by splitting a basic block we may be unable to use some of the code reusability. In fact (see figure 1 in next page) if we have a loop ten statements long and an instruction memory only five statements long we will not be able to execute asynchronously ten times the loop body inside each PE; on the contrary the controller will need to broadcast a new half of the loop body every five statements.

4 A RESTRICTED VERSION OF THE REUSABILITY FACTOR

A sufficient condition to fulfill requirement **b)** is to have a rather large value of the **reusability factor** for the program to be executed. Anyway, this condition is too restrictive and unnecessarily difficult to realize.

Definition 4

We define restricted reusability factor the value of the ratio n/m calculated on a single block of m instructions.

In the SBMD model the controller divides the compiled program in blocks of code of length **m**. The controller then broadcasts to the PEs blocks of instructions instead of single instructions.

Definition 5

We define **single shot block** *an autonomously executable block of code of maximum length and broadcast by the controller in a single transmission.*

The definition of single shot blocks is tightly coupled with the definition of the m-execution autonomy, in fact it requires that the transmitted block is of the maximum length autonomously executable.

In order to fulfill the requirement **b)** it is enough that any single shot block of code of length **m** has a value of the restricted reusability factor larger than one. As a matter of fact, we overlap the execution of a single shot block with the broadcast of the next one. Thus we don't need to impose any constraint on a non single shot block, i.e. on any block of code that is not transmitted or executed in a single step.

In a program, a high reusability factor is mainly due to the presence of loops. In sequential programs, iterations are very common and so it is possible to have high values of the reusability factor in compiled programs even without the use of smart compilation techniques. In data-parallel programs, the loops are usually unrolled on the PEs of the machine, and the iterative constructs are by far less frequent than in sequential programs.

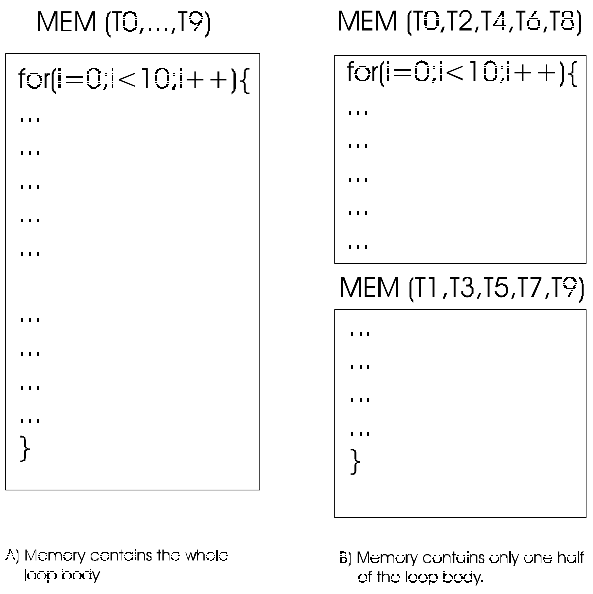
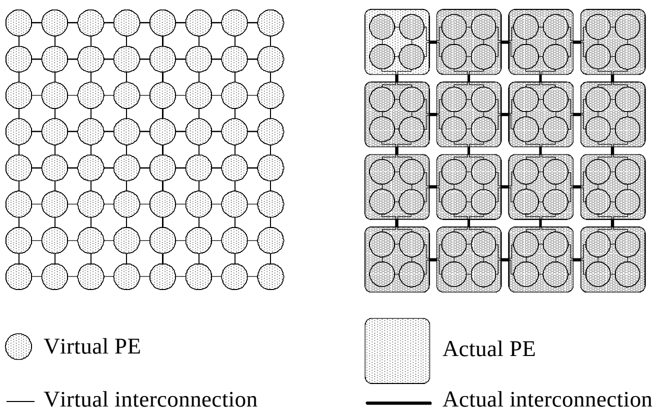


Figure 1 Split basic blocks may reduce code reusability introduced by iterative structures.

These facts seems to imply that it is not possible to have an adequate value of the reusability factor, not even in its restricted form, in data-parallel programs.

However, in data-parallel programs the amount of data that can be processed in parallel is usually much larger than the number of physical PEs of the machine. As a matter of fact, while to process data matrices of size 512×512, or even larger, is quite common, one of the largest data-parallel machine ever realized,



a) A virtual processor array of 8×8 Pes. b) Mapping a virtual processor array on a physical processor array of 4×4 PEs

Figure 2 Contraction mapping. In b) the physical processor are showed as square boxes.

the CM-2, has only 256×256 PEs in its largest configuration and the MasPar systems have 128×128 PEs in their largest configuration. Thus it is necessary to simulate the operations of many virtual PEs on a single actual PE, i.e. to map many virtual PEs on a single actual PE. The choice of how to do this mapping may result particularly critical for some applications, because it affects the way in which inter processor communication among the virtual PEs is to be realized.
In past related works [3] different kind of mapping have been proposed, in this paper we focus our attention only on contraction mapping.

Definition 6
k-way contraction mapping: we define k-way contraction mapping of a virtual processor array of size n on a physical processor array of size m the mapping of a group of virtual processor on a single physical processor in a way such that virtual processor $P_{i,j}$ is mapped onto physical processor $P_{s,t}$ (where i, j = 0, 1, ... , $\sqrt{n}$-1 and s,t = 0, 1, ... , $\sqrt{m}$-1) if and only if $s \leq \frac{i}{\sqrt{k}} \leq s+1$ and $t \leq \frac{j}{\sqrt{k}} \leq t+1$ (where $k = \frac{n}{m}$ and $\sqrt{n}$, $\sqrt{m}$, $\sqrt{k}$ are integers).
This means that the virtual mesh of processor is divided in sub-meshes of constant dimension and each one of them is simulated inside a single real processor (see figure 2).
Mapping many virtual PEs on a single actual PE increases the value of the reusability factor of data-parallel programs, namely a n-way contraction mapping increases the reusability factor of each one-shot block n times. Besides, if we adopt contraction mapping a large number of inter-PE communications become intra-PE-memory data movements. This transformation trades some of the actual load of the interconnection network with an increased load of the local memory system of each PE. This increment of the memory system load can be prevented in the case of an architecture supporting AA. In fact, if each PE is able to generate locally its memory addresses, each memory copy due to the simulation of inter-virtual-PE communication can be translated into the recalculation of some pointer. The reduction of the actual number of inter-PE communication primitives allows

reducing the bandwidth and the cost of the interconnection network of the architecture.
In order to make the guaranteed reusability of instructions due to an n-way contraction mapping profitable it is necessary that the repetition of the program instructions n times does not produce a slow down of more than n times. We call this property of a machine *virtual parallelism support.*
We claim that a **n-**way contraction mapping and the support for virtual parallelism are sufficient condition for allowing a PEs clock speed-up of **n** on execution autonomy enhanced machines.
The presence of a n-way contraction mapping guarantees that each instruction of a data-parallel program needs to be executed no less than n times, thus each single-shot block of the program is guaranteed to have a restricted reusability factor not smaller than n. This property holds true even if the instruction memory dimension forces us to split some basic blocks of the program, in fact this reusability does not come from program iterative structures. This fact guarantees that the PEs can execute instructions at a rate up to n times faster than the rate at which the central controller broadcasts them without starving.
At the same time the support of virtual parallelism guarantees that each PE will be able to complete the execution each single shot block of the program in no more than n*p cycles, where p is the number of cycles required to execute the single-shot block in parallel on n PEs. In fact if the sequentialized execution of single-shot bloc S requires more than n*p cycles the complexity of the partially sequentialized algorithm containing S is more than n times larger than the complexity of the parallel algorithm and this is in contradiction with the definition of virtual parallelism support.

5 CONCLUDING REMARKS

In this paper:

- we have proposed a new autonomy in the SIMD computational paradigm, namely Execution Autonomy;
- we have proposed the SBMD computational paradigm based on Execution Autonomy;
- we have discussed some implementation problems of the SBMD paradigm.
- we have shown that virtual parallelism support is a necessary condition to exploit a relaxation of the SIMD strict synchronous execution;
- we have shown that an m-way contraction mapping and the support for virtual parallelism constitute a sufficient condition for the exploitation of m-execution autonomy.

The execution speed and the performance of machines based on the SBMD computing paradigm are not bound by the time needed to broadcast instructions and data as they are in the SIMD computing paradigm.
The EA allows the PEs of a machine based on the SBMD computing paradigm to adopt a faster clock than the PEs of a machine based on the SIMD computing paradigm.

Given these results, it is possible and convenient to increase the performance of the PEs, both from the technological and from the architectural point of view; as a matter of fact any such increase directly increases the machine performance because the time needed to broadcast the instructions does not bind it any more.
Besides, the use of EA and the support for the SBMD paradigm overcomes the rigidity of the traditional SIMD computational model and it allows avoiding to have large sets of PEs inactive during the execution of conditional statements.

6 References

[1] P. Baglietto, M. Maresca, M. Migliardi, *Introducing Execution Autonomy in the SIMD Computing Paradigm*, Proc. of the International Conference on Massively Parallel Processing Applications and Development, Delft (The Netherlands), June 21-23, 1994.

[2] G. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick and R. A. Stokes, *The ILLIAC IV computer*, IEEE Trans on Computer, C-17 (8), pp.746-757, Aug. 1968.

[3] Y. Ben-Asher, D. Gordon and A. Schuster, *Optimal Simulations in Reconfigurable Processor Arrays*, Technion ITT technical report, Haifa (Israel), 1992.

[4] W. D. Hillis and G. L. Steele Jr., *Data-parallel Algorithms*, Communications of the ACM, Vol. 29, No. 12, pp. 1170-1183, Dec. 1986.

[5] W. D. Hillis and L. W. Tucker, *The CM-5 Connection Machine: a Scalable Supercomputer*, Communications of the ACM, Vol. 36 No. 11, pp. 31-40, Nov. 1993

[6] M. Maresca, *Polymorphic Processor Array*, IEEE Trans. on Parallel and Distributed Systems, Vol. 4, No. 5, pp. 490-506, May 1993

[7] M. Maresca, M. A. Lavin and H. Li, *Parallel Architectures for Vision*, Proceedings of the IEEE, Vol. 76, No. 8, August 1988.

[8] R. Miller, V. K. Prasanna-Kumar, D. Reissis e Q. F. Stout, *Parallel Computations on Reconfigurable Meshes*, IEEE Trans. on Computers Vol. 42, No. 6, giugno 1993, pp. 678-692.

[9] M. A. Nichols, H. J. Siegel and H. G. Dietz, *Data Management and Control Flow Aspects of an SIMD/SPMD Parallel Language/Compiler*, IEEE Trans. on Parallel and Distributed Systems, Vol. 4, No. 2, pp. 222-234, February 1993.

[10] S. Rehfuss and D. Hammerstromm, *Comparing SFMD and SPMD Computation for On-chip Multiprocessing of Intermediate Level Image Understanding Algorithms*, Proc. of the Fourth IEEE International Workshop on Computer Architecture for Machine Perception, Cambridge, Massachusetts, October 20-22, 1997.

[11] M. Sato, Y. Kodama, S. Sakai, Y. Yamaguchi and Y. Koumura, *Thread-Based Programming for the EM-4 Hybrid Dataflow Machine*, Proc. of ISCA92, pp 145-155, 1992.

[12] C. Weems, *Asynchronous SIMD: An Architectural Concept for High Performance Image Processing*, Proc. of the Fourth IEEE International Workshop on Computer Architecture for Machine Perception, Cambridge, Massachusetts, October 20-22, 1997.