1

Implementation of Parallel Arithmetic in a Cellular Automaton

Richard K. Squier, Ken Steiglitz, Mariusz H. Jakubowski

Abstract

We describe an approach to parallel computation using particle propagation and collisions in a one-dimensional cellular automaton using a particle model — a Particle Machine (PM). Such a machine has the parallelism, structural regularity, and local connectivity of systolic arrays, but is general and programmable. It contains no explicit multipliers, adders, or other fixed arithmetic operations; these are implemented using fine-grain interactions of logical particles which are injected into the medium of the cellular automaton, and which represent both data and processors. We give parallel, linear-time implementations of addition, subtraction, multiplication and division.

1: Introduction

The goal of this paper is to use Particle Machines (PMs) to incorporate the parallelism of systolic arrays [3] in hardware that is not application-specific and is easy to fabricate. The PM model, introduced in [8, 6, 7], uses colliding particles to encode computation. A PM can be realized in VLSI as a Cellular Automaton (CA), and the resultant chips are locally connected, very regular (being CA), and can be concatenated with a minimum of glue logic. Thus, many VLSI chips can be strung together to provide a very long PM, which can then support many computations in parallel. What computation takes place is determined entirely by the stream of injected particles: there are no multipliers or other fixed calculating units in the machine; the logic supports only particle propagation and collisions. So, while many algorithms for a PM mimic systolic arrays and achieve their parallelism, these algorithms are not hard-wired but are "soft" in the sense that they do not use any fixed hardware structures.

An interesting consequence of this flexibility is that the precision of fixed-point arithmetic is completely arbitrary and determined at run time by the user. Variable precision arithmetic is exploited in a linear-time, arbitrary-precision division algorithm described in [10] and illustrated in Section 3.

The recent paper [7] shows that FIR filtering of a continuous input stream, and arbitrarily nested combinations of fixed-point addition, subtraction, and multiplication, can all be performed in one fixed CA-based PM in time linear in the number of input bits, all with arbitrary precision.

The description of a particular PM includes its collision rule set, which determines the results of collisions of particles. Because these rules only partially specify their input, there are rule sets whose rules conflict on the outcome of some collisions: that is, one rule states

*Richard Squier is with the Computer Science Department at Georgetown University, Washington DC 20057. Ken Steiglitz and Mariusz Jakubowski are with the Computer Science Department at Princeton University, Princeton NJ 08544.

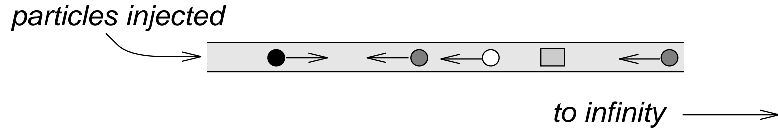

Figure 1. *The basic conception of a particle machine*

that some particle is present in the outcome, and another rule states that it is not. If, given a particular set of inputs to the PM, one of these conflicting collisions occurs, we say the rule set is not *compatible* with respect to that set of inputs. The PM model is Turing equivalent, so the general question of compatibility is undecidable. However, if the set of inputs is sufficiently constrained, as is usually the case, the constraints can be used to test compatibility in time polynomial in the number of particles and rules. See [10] for a proof of undecidability and a discussion of a practical solution to the problem.

We define the PM model in the next section. We then conclude with high-level descriptions or simulation examples of parallel, linear-time PM implementations of addition, subtraction, multiplication, and division.

2: Particle machines

Figure 1 shows the general arrangement of a PM. Particles are injected at one end of the one-dimensional CA, and these particles move through the medium provided by the cells. When two or more particles collide, new particles may be created, existing particles may be annihilated, or no interaction may occur, depending on the types of particles involved in the collision.

The state of cell i of a 1-d CA at time $t+1$ is determined by the states of cells in the *neighborhood* of cell i at time t, the neighborhood being defined to be those cells at a distance, or *radius*, r or less of cell i. Thus, the neighborhood of a CA with radius r contains $k = 2r+1$ cells and includes cell i itself.

We think of a cell's n-bit state vector as a binary *occupancy vector*, each bit representing the presence or absence of one of n particle types (the same idea is used in lattice gasses; see, for example, [2]). The operation of the CA is determined by a rule, called the *update* rule, which maps states of the cells in the neighborhood of cell i at time t to the state of cell i at time $t+1$.

Figure 2 illustrates some typical collisions when binary addition is implemented by particle collisions. This particular method of addition will be one of two described later when we develop arithmetic algorithms. The basic idea is that each addend is represented by a stream of particles containing one particle for each bit in the addend, one stream moving left and the other moving right. The two addend streams collide with a ripple-carry adder particle where the addition operation takes place. The ripple-carry particle keeps track of the current value of the carry between collisions of subsequent addend-bit particles as the streams collide least-significant-bit first. As each collision occurs, a new rightmoving result-bit particle is created and the two addend particles are annihilated. Finally, a trailing “reset” particle moving right resets the ripple-carry to zero and creates an additional result-bit particle moving right.

We need to be careful to avoid confusion between the bits of the arithmetic operation and the bits in the state vector. The ripple-carry adder is represented by two particle types, the bits of the rightmoving addend and the rightmoving result are represented by two more particle types, the leftmoving addend bits are represented by another two types, and the

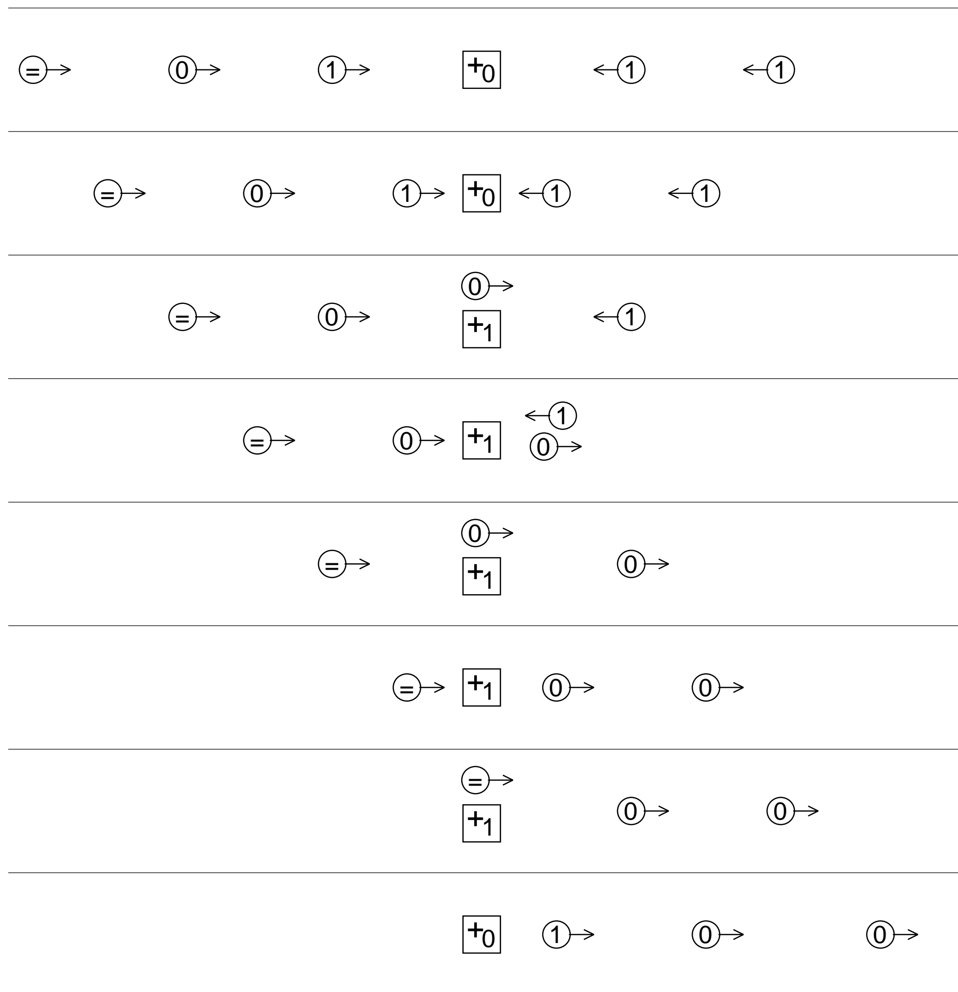

Figure 2. *An example illustrating some typical particle collisions, and one way to perform addition in a particle machine. What is shown is actually the calculation* $01+11 = 100$*, implemented by having the two operands, one moving left and the other moving right, collide at a stationary "ripple-carry" particle. When the leading, least-significant bits collide, in going from row 2 to 3, the ripple-carry particle changes its identity so that encodes a carry bit of* 1*, and a rightmoving sum particle representing a bit of* 0 *is created. The final answer emerges as the rightmoving stream* 100*, and the ripple-carry particle is reset by the "equals" particle to encode a carry of* 0*. The bits of the two addends are annihilated when the sum and carry bits are formed. Notice that the particles are originally separated by empty cells, and that all operations can be effected by a CA with a neighborhood size of 3 (a radius of 1).*

reset particle is represented by one additional type. Thus, the operations shown in Fig. 2 use seven bits of the state vector. We'll denote by C_i the Boolean state vector variable for cell i. The individual bits in the state vector will be denoted by bracket notation: for instance, the state vector bit corresponding to a rightmoving zero particle in cell i is denoted $C_i[0_R]$. The seven Boolean variables representing the seven particles are:

$C_i[0_R]$	rightmoving zero
$C_i[0_L]$	leftmoving zero
$C_i[1_R]$	rightmoving one
$C_i[1_L]$	leftmoving one
$C_i[+_0]$	ripple-carry adder w/ zero carry
$C_i[+_1]$	ripple-carry adder w/ one carry
$C_i[=_R]$	rightmoving adder reset

All the particle interactions and transformations shown in the example can be effected in a CA with radius one; that is, using only the states of cells $i-1$, i, and $i+1$ to update the state of cell i. A typical next-state rule (as illustrated in the first collision in Fig. 2) therefore looks like

$$C_i[0_R]^{(t+1)} \Leftarrow (C_{i-1}[1_R] \wedge C_i[+_0] \wedge C_{i+1}[1_L]\ )^{(t)} \tag{1}$$

which says simply that if the colliding addends are 1 and 1, and the carry is 0, then the result bit is a rightmoving 0.

Notice that using two state-vector bits to represent one data bit allows us to encode the situation when the particular data bit is simply not present. (Theoretically, it also gives us the opportunity to encode the situation when it is both 0 and 1 simultaneously, although the rules are usually such that this never occurs.) It can be very useful to know that a data bit isn't present.

In [10] we estimate that 300 cells of a CA implementing a PM with 100 rules and 36 particles will fit on a cm^2 100 MHz CMOS chip. Such a PM can implement all four arithmetic operations in linear time.

3: Linear-time arithmetic

We conclude with sketches of linear-time PM implementations of the four basic arithmetic operations. For details, see [6, 7]. Note that in all of these implementations, we can consider velocities as relative to an arbitrary frame of reference. We can always change the frame of reference by appropriate changes in the update rules.

Figure 3 shows in diagrammatical form the scheme already described in detail in Fig. 2. Figure 4 shows an alternate way to add, in which the addends are stationary, and a ripple-carry particle travels through them, taking with it the bit representing the carry. We can use either scheme to add, simply by injecting the appropriate stream of particles. The choice will depend on the form in which the addends happen to be available in any particular circumstance, and on the form desired for the sum. Note also that negation can be performed easily by sending a particle through a number to complement its bits, and then adding one — assuming we use two's-complement arithmetic.

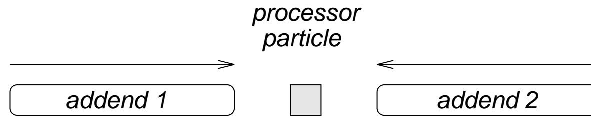

Figure 3. *The particle configuration for adding by having the addends collide bit by bit at a single processor particle.*

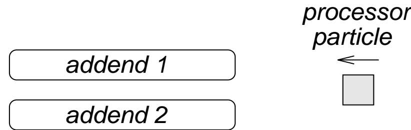

Figure 4. *An alternate addition scheme, in which a processor travels through the addends.*

Figure 4 also illustrates the use of “tracks”. In this case two different kinds of particles are used to store data at the same cell position, at the cost of enlarging the particle set. This turns out to be a very useful idea for implementing multiply-accumulators for FIR filtering, and feedback for IIR filtering [7]. The idea is used in [10] for implementing division, adapting an algorithm due to Leighton [4].

Figure 5 shows the particle arrangement for fixed-point multiplication. This mirrors the well known systolic array for the same purpose, but of course the structure is “soft” in the sense that it represents only the input stream of the PM which accomplishes the operation.

Figure 6 shows a picture generated by a simulation of the division example described in [10]. The simulated PM uses 38 types of particles and 112 rules, and is capable of realizing all the applications mentioned in this paper, including FIR and IIR filtering.

Finally, Fig. 7 shows a completely different algorithm implemented on exactly the same machine. This example computes the infinite series $1/(1-x) = 1 + x(1 + x(1 + ...)...)$, for the case $x = 0.5$. Four bits of precision are used in the calculation, and a test particle and associated rules are used to detect the termination condition, which is that no change occur in the four most significant bits of the intermediate result.

The reader is referred to [6, 7, 10] for more detailed descriptions and a discussion of nested operations and digital filtering.

4: Discussion

In a way, a PM is a programmable parallel computer without an instruction set — a NISC (No Instruction Set Computer). What happens in the machine is determined by the

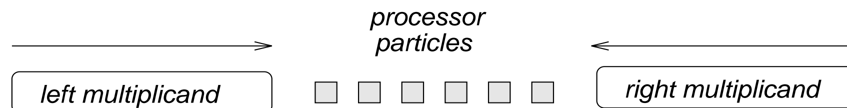

Figure 5. *Multiplication scheme, based on a systolic array. The processor particles are stationary and the data particles collide. Product bits are stored in the identity of the processor particles, and carry bits are stored in the identity of the data particles, and thereby transported to neighbor bits.*

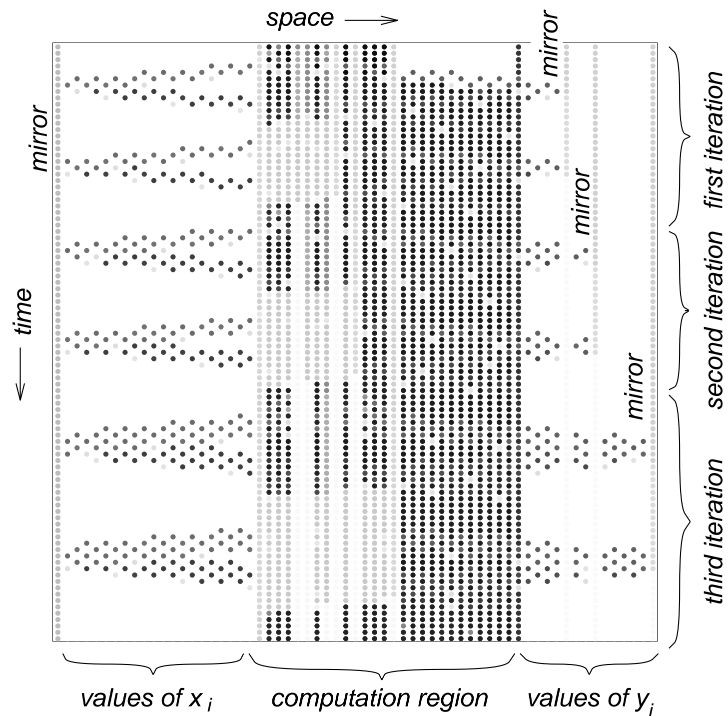

Figure 6. *PostScript generated by a simulation of the division implementation. Each cell is represented by a small circle whose shading depends on which particles are present in that cell. For clarity, only every seventh generation is shown. The example is 1/7.*

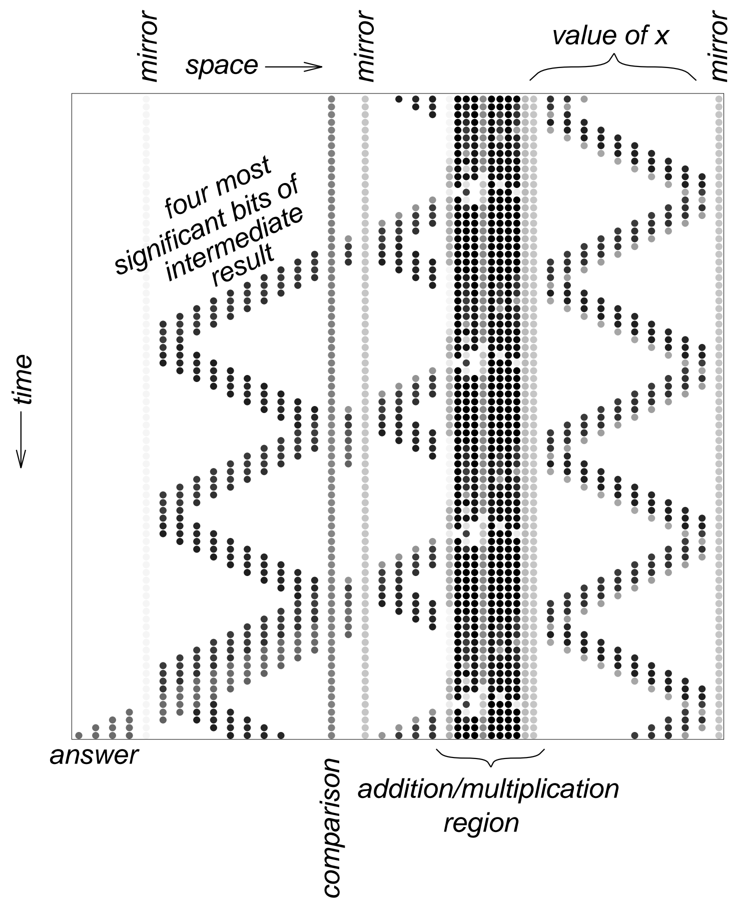

Figure 7. *Simulation of a different algorithm on the same machine used for division. The computation is the evaluation of the power series for $1/(1-x)$ by Horner's Rule. Every second generation is shown.*

stream of input particles. At this point we have accumulated tricks for translating systolic arrays and other structures into particle streams, but general problems of programming a PM, such as designing higher-level languages and building compilers, are unexplored.

The three main advantages of PMs for doing parallel arithmetic are the ease of design and construction, the high degree of parallelism available through simple concatenation, and the flexibility of word length — which depends, of course, only on the particle groups entering the machine.

In summary, the particle model gives us a new way to think about parallel computation. The medium that supports the particles need not be a CA [5], and even if it is, the implementation need not be in VLSI.

5 Acknowledgement

This work was supported in part by NSF grant MIP-9201484, and a grant from Georgetown University.

References

[1] Cappello, P. R., "Towards an FIR Filter Tissue," Proc. ICASSP 85, pp. 276-279, Tampa, FL, Mar. 1985.

[2] U. Frisch, D. d'Humie'res, B. Hasslacher, P. Lallemand, Y. Pomeau, and J. P. Rivet, "Lattice gas hydrodynamics in two and three dimensions," *Complex Systems* **1** (1987), pp. 649-707.

[3] H. T. Kung, "Why systolic architectures?" *IEEE Comput.* **15** 1 (Jan. 1982), pp. 37-46.

[4] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures*, Morgan Kaufman Publishers, San Mateo, CA, 1992.

[5] N. Margolus, "Physics-like models of computations," *Physica* **10D** (1984), pp. 81-95.

[6] R. K. Squier and K. Steiglitz, "Subatomic particle machines: parallel processing in bulk material," submitted to *Signal Processing Letters*.

[7] R. K. Squier and K. Steiglitz, "Programmable Parallel Arithmetic in Cellular Automata using a Particle Model," Tech. Rept. CS-TR-478-94, Computer Science Dept., Princeton Univ., Dec. 1994. To appear in *Complex Systems*.

[8] K. Steiglitz, I. Kamal, and A. Watson, "Embedding computation in one-dimensional automata by phase coding solitons," *IEEE Trans. on Computers* **37** 2 (1988), pp. 138-145.

[9] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*, Addison-Wesley, Reading, MA, 1985.

[10] R. K. Squier, K. Steiglitz, and M. H. Jakubowski, "General Parallel Computation without CPUs: VLSI Realization of a Particle Machine," Tech. Rept. CS-TR-484-95, Dept. of Computer Science, Princeton University, Feb. 1995.