1

General Parallel Computation without CPUs: VLSI Realization of a Particle Machine

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

February 6, 1995

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 sketch a VLSI implementation of a PM, and estimate its speed and size. We next discuss the problem of determining whether a rule set for a PM is free of conflicts. In general, the question is undecidable, but enough side information is usually available in practice to answer the question in polynomial time. We then show how to implement division in time linear in the number of significant bits of the result, using an algorithm of Leighton. This complements similar previous results for fixed-point addition/subtraction and multiplication. The precision of the arithmetic is arbitrary, being determined by the particle groups used as input.

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

*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.

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. 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. In Section 6 of this paper we complete this suite of parallel arithmetic operations with an implementation of division that exploits the PM's flexibility by changing precision during computation.

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 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, as we show later in this paper, 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.

The next section gives a description of the PM model. This is followed by an outline of an implementation of the model in VLSI and an estimate of its performance. We then discuss the issue of rule conflict. Finally, we give a detailed description of an implementation of Leighton's Newtonian division algorithm in the model.

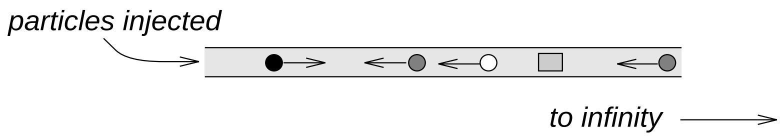

Figure 1: *The basic conception of a particle machine*

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 arith-

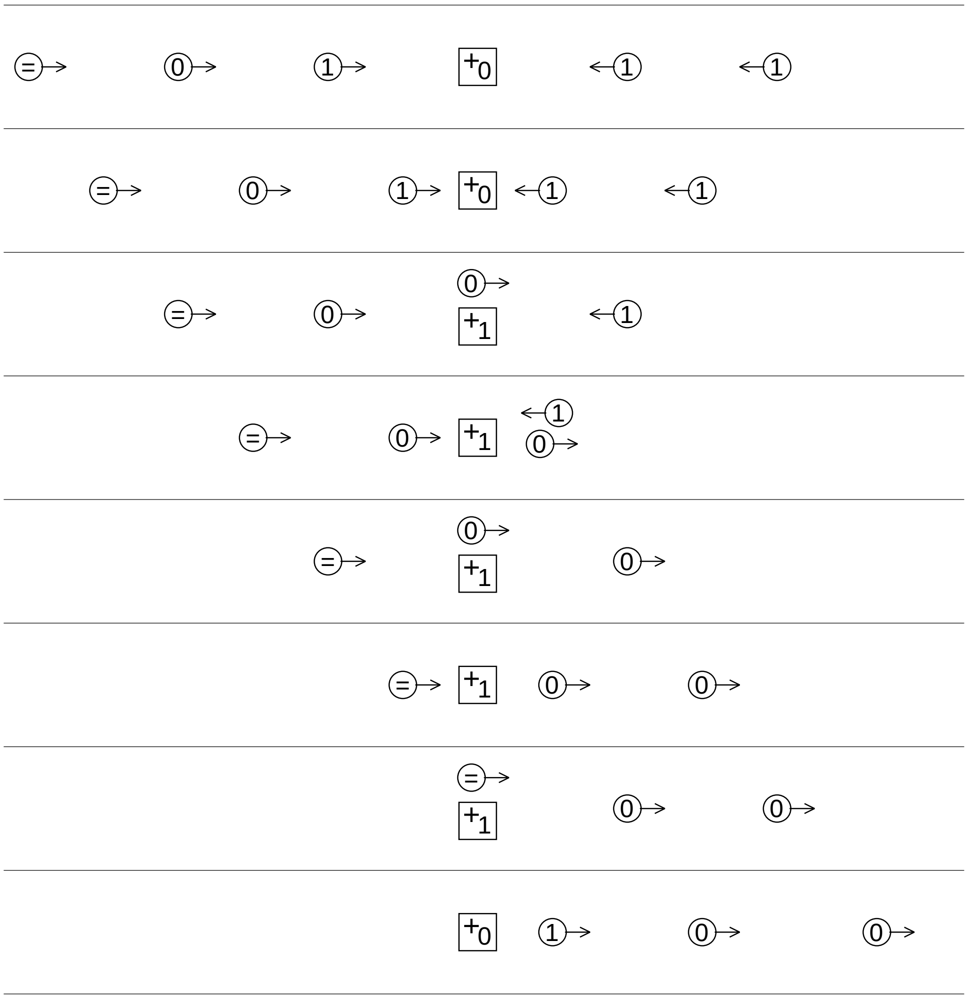

Figure 2: *An example illustrating some typical particle collisions, and one way to perform addition in a particle machine. What is shown is actually the calculation* $01 + 11 = 100$*, implemented by having the two operands, one moving left and the other moving right, collide at a stationary "ripple-carry" particle. When the leading, least-significant bits collide, ingoing 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).*

metic 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 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 a data bit isn't present.

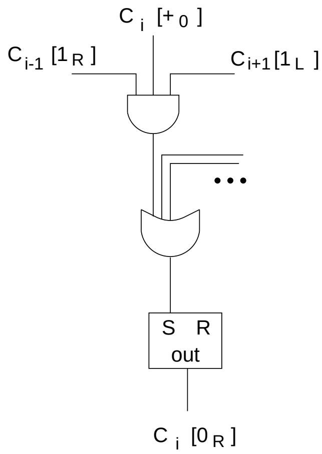

Figure 3: *The logic fragment of an implementation of the rule in Eq. 1. The conditions create a rightmoving 0.*

3 VLSI size and performance estimate

We base our estimate of VLSI size and performance on a straightforward realization of the rule set as suggested by Fig. 3. In that figure, the rule in Eq. 1 is implemented directly in random logic feeding the inputs to an S-R flip-flop that stores one bit of a cell's state vector. This makes it easy to map the rules into logic generally; each rule that creates or destroys a particle sets or resets its corresponding bit. Figure 4 shows the general logic fanning into a state bit.

In a practical VLSI implementation the rules would probably be combined and realized in a PLA. The following area estimate assumes this is done. If PMs prove practically useful, it may be worth investing considerable effort in optimizing the layout of a PM for a rule set that is sufficiently powerful to implement a wide variety of computations. After all, this design need be done only once, and the resulting chip would be useful in many applications. As with memory cells, only a single cell needs to be designed, which can then be replicated. The chips themselves can be concatenated to form very long — and hence highly parallel — machines.

Our layout of a row of cells is shown in Fig. 5. Here each cell has p bits in its state vector, thus supporting p particle types, and contains a logic block, a bank of flip-flops, and wiring connecting logic inputs and outputs to flip-flop inputs and outputs. Our estimate of the area required for these

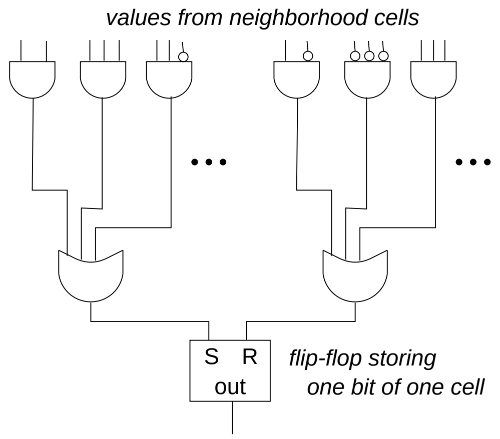

Figure 4: *A possible layout plan for a PM.*

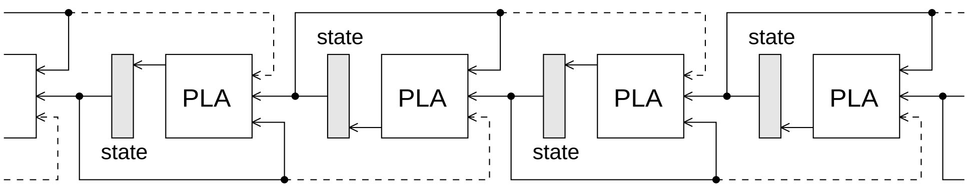

Figure 5: *The general layout of cells in the CA for a PM. The bits in the state vector are shown as shaded registers. Each such register supplies data to the update logic for its own cell, and those of its nearest neighbor cells. Connections to right neighbors are shown as dashed lines.*

elements uses a rough approximation to the space required to route a single signal wire in modern VLSI [9]: about $\alpha = 6\lambda$, where $\lambda = 0.2\mu$. Thus, a wiring channel containing n wires we estimate to be $n6\lambda$ across. We allow four times as much space per signal wire for PLA signal wires, or $\beta = 24\lambda$ per PLA wire.

As laid out in Fig. 5, a cell requires vertical space for two wiring channels. In addition, we must fit in the larger of either a bank of p flip-flops or the vertical span of the PLA. Since we assume a simple layout, the PLA is the larger structure and we ignore the flip-flops. The PLA contains p input wires, $p/3$ from the cell's own flip-flops and $p/3$ each from the two neighbor cells, and p output wires, giving $2p$ total PLA wires, counting vertically. The total height of a cell is then $(2/3)p\alpha + 2p\beta$.

Horizontally, a cell must accommodate the PLA, two wiring channels, and the flip-flops. The PLA requires roughly as many wires as minterms. Estimating an average of four minterms per output wire, we get $4p$ horizontal PLA wires. The width of the flip-flops is about 10 PLA wires. A cell's width is then $(2/3)p\alpha + 4p\beta + 10\beta$.

Let's establish the number of particles required, p, for a general PM. Consider a single track of data. In general for a single track we need data bits that travel in either direction or remain stationary. This requires six particles: 0 and 1 moving left, 0 and 1 moving right, and stationary 0 and 1. Three data tracks suffice for all applications we have tried. For each data track we assume about six operator particles are needed. This gives us $p = 36$ total operator and data particles. Using the above area estimates and given a chip 1 centimeter on a side, we find there is room for about 300 cells on a single chip for a PM supporting 36 particles.

Now let's estimate the potential parallel performance. Using the multiplication scheme shown later in Fig. 8 we need about $2n$ cells to hold a single n-bit operand moving either left or right. The processor particles need $2n$ cells. This gives $6n$ total cells between the beginning of one multiply and the beginning of the next multiply. Supposing we have 16-bit operands, this means we can fit three 16-bit multiplies operating concurrently on a single chip. The cell logic is very simple, so a conservative estimate of clock speed is 100 Mhz. A multiply completes in $2n$ ticks. This gives us about 3 million 16-bit integer multiplies per second per chip. Using logic optimization and other layout and performance refinements in the chip design, we might expect to get a factor of 5 to 10 improvement over this estimate.

4 Compatible collision rules

4.1 Collision rules

A set of rules in a PM is a relation between *preconditions* that determine the rule's applicability to collisions and *effects* that give the outcome of collisions. The domain is a set of pairs of the form (P_D, A_D), where P_D is a set of particles that must be *present* in the neighborhood in order for the rule to apply, and A_D is a set of particles that must be *absent*. We refer to particles in A_D as *particle negations*, and we consider particle negations to collide, even though the actual particles are not present. The range is a set of pairs (P_R, A_R), where the sets P_R and A_R give the particles that are created and destroyed, respectively.

4.2 Compatibility

When particles collide, two or more collision rules may apply simultaneously to determine the results. For our purposes the effects of these rules should not conflict; that is, one such rule should not destroy any particle created by another. If this condition is satisfied, the results of the collision depend only on the colliding particles, not on the order of rule application. We call a rule set of a PM *compatible with respect to a set of inputs* (or simply *compatible*) if every collision that occurs is resolved without conflicts. We use the term *input* to include the initial state of the PM's medium, and all particles subsequently injected.

Given a PM, we want to be able to determine whether or not its rule set is compatible. The general problem of determining compatibility turns out to be undecidable. However, as we will see, if the PM designer provides certain additional information about the particles, the problem is solvable in time polynomial in the number of particles and rules.

4.2.1 Rule compatibility is undecidable

Theorem 1 *The rule compatibility problem is undecidable.*

Proof: A straightforward reduction from the halting problem. Given a Turing machine M with an initial configuration of its tape, we transform M into a PM, and M's tape into this PM's input, in such a way that M halts if and only if the PM's rule set is not compatible.

Let S, Γ, δ, and h denote M's set of states, tape alphabet, transition function, and halt state, respectively. We begin constructing the PM P from M as follows. For each symbol $x \in \Gamma$, introduce a new stationary particle x_p. The tape of the Turing machine then maps directly into the medium of the PM; in particular, the initial configuration of M's tape corresponds to the initial state of P's medium.

We simulate the transition function δ with particles and collision rules. For each state $s \in S$, create a stationary particle, s_N. This particle is designed to perform the function of M's tape head. Assume that the transition function is defined as δ: $S \times \Gamma \rightarrow S \times \Gamma \cup \{L, R\})$, where L and R are special symbols that indicate left and right movement of the tape head. For all states s and tape symbols x such that $\delta(s, x)$ is defined:

- If $\delta(s, x) = (t, y)$, introduce a rule that transforms the stationary state particle s_N to the stationary state particle t_N, and transforms the symbol particle x_p into the symbol particle y_p. This rule simulates M's changing state and writing a symbol on its tape.

- If $\delta(s, x) = (t, L)$, introduce rules that move the state particle s_N one cell to the left and transforms s_N into t_N. These rules simulate M's changing state and moving its tape head to the left.
- If $\delta(s, x) = (t, R)$, introduce analogous rules to simulate tape head movement to the right.

To complete the construction, add the stationary particle corresponding to M's initial state to the cell corresponding to M's initial head position. The above rules must be compatible, because the medium behaves exactly like M's tape and the rules operate according to M's transition function. Finally, choose an arbitrary particle x_p and introduce two conflicting rules. One rule transforms the particle representing the halt state of M into x_p; the other rule transforms it into x_p's negation. The complete set of rules is compatible if and only if M never halts. □

4.2.2 Additional information makes compatibility decidable

Although deciding rule compatibility from only the rule set and the input is not possible in general, all is not lost. If the PM designer provides complete information about which pairs of particles can collide, we can determine compatibility with a simple polynomial-time algorithm. The PM designer usually has a good idea of which particles can collide and which cannot, even though computing this information is in general an undecidable problem. For example, a binary arithmetic algorithm most likely uses two particles representing a 0 and a 1 that never coexist in the same cell.

The information which the PM designer should provide is an exhaustive list $\mathcal{L}$ of pairs of the form (α, β), where α and β are particles or negations of particles, and α and β can collide. We assume that the designer is willing to guarantee the correctness and completeness of this information, so that if the pair (α, β) is not in the list, then α and β can never collide.

An easy way to check for rule compatibility is to ensure that each pair of rules in the rule set satisfies the following condition: if the rules apply to any collision simultaneously, then the effects of the rules do not conflict. The rule effects (P_1, A_1) and (P_2, A_2) conflict if and only if one rule destroys a particle created by the other; that is, if $P_1 \cap A_2 \neq \emptyset$ or $P_2 \cap A_1 \neq \emptyset$. Two rules

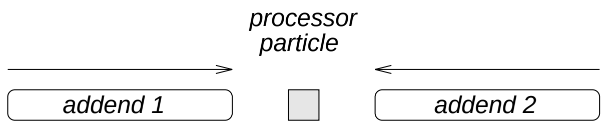

Figure 6: *The particle configuration for adding by having the addends collide bit by bit at a single processor particle.*

with preconditions $(\hat{P}_1, \hat{A}_1)$ and $(\hat{P}_2, \hat{A}_2)$ can be applicable simultaneously only if the following conditions hold:

- The rules do not conflict in their preconditions, that is, $\hat{P}_1 \cap \hat{A}_2 = \emptyset$ and $\hat{P}_2 \cap \hat{A}_1 = \emptyset$.
- The combined preconditions of the rules contain only pairs (α, β) that can collide; that is, $\alpha \in \hat{P}_1 \cup \hat{A}_1$ and $\beta \in \hat{P}_2 \cup \hat{A}_2$ only if $(\alpha, \beta) \in \mathcal{L}$.

It is easy to verify that these conditions can be checked in time polynomial in the number of particles and rules.

5 Linear-time arithmetic

We will conclude this paper with a description of a linear-time PM implementation of Leighton's division algorithm [4]. Before we discuss division, however, we briefly review the implementations of addition and multiplication given in [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 6 shows in diagrammatical form the scheme already described in detail in Fig. 2. Figure 7 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.

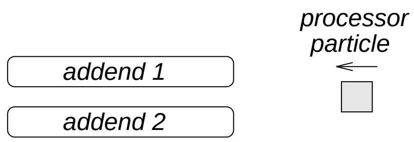

Figure 7: *An alternate addition scheme, in which a processor travels through the addends.*

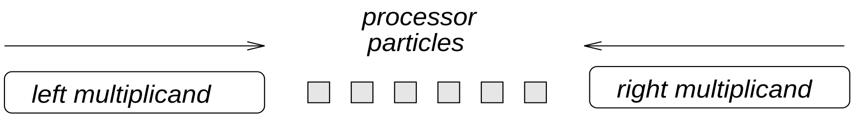

Figure 8: *Multiplication scheme, based on a systolic array. The processor particles are stationary and the data particles collide. Product bits are stored in the identity of the processor particles, and carry bits are stored in the identity of the data particles, and thereby transported to neighbor bits.*

Figure 7 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 the next section for implementing division.

Figure 8 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.

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

6 Linear-time division

Although division is much more complicated than the other elementary arithmetic operations, a linear-time, arbitrary-precision algorithm is possible using the particle model. The algorithm we present here, based on Newtonian iteration and described by Leighton [4][1], calculates the reciprocal of a number x. We assume for simplicity that x is scaled so that

[1] The algorithm described here is actually slightly different in some details, but it is not hard to verify its complexity and correctness.

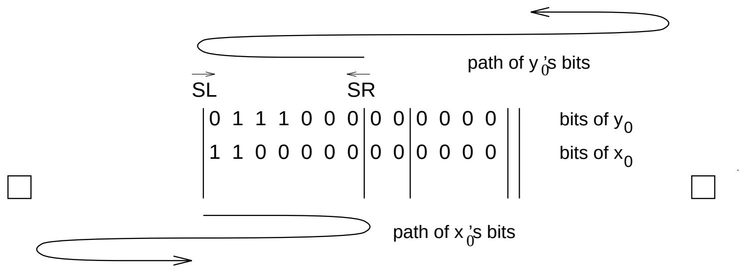

Figure 9: *Initial configuration for division.*

$\frac{1}{2} \leq x < 1$. For an arbitrary division problem, we rescale the divisor by shifting its binary point left or right, calculate its reciprocal, multiply by the dividend, and finally scale the result back. Since each of these steps takes only linear time, the entire division uses only linear time.

The algorithm works as follows. Let N denote the number of bits desired in the reciprocal. For simplicity, assume N is a power of 2. Let x_i represent the i-th approximation of the reciprocal and y_i the divisor, both rounded down to $2^{i+1}+4$ places. Beginning with $x_0 = 1.1$ in binary, the method iterates using $x_{i+1} = x_i(2 - y_i x_i)$, for $0 \leq i < \lg N$. At the end of the i-th iteration, the error is less than $2^{-2^{i+1}}$.

The particle implementation of this algorithm carefully coordinates stationary and moving particles to create a loop. The i-th iteration of the loop performs linear-time, fixed-point additions and multiplications using $2^{i+1}+4$ bits of precision. *Marker* particles delimit the input number and indicate the bit positions that define the successive precisions required by the algorithm.

Figure 9 illustrates this setup, giving the initial template for calculating the reciprocal of the number 7 to 8 bits of precision, that is, with error less than $\frac{1}{256}$. The outer markers enclose the binary numbers $y_0 = 0.111$, which is 7 rescaled to fit the algorithm, and $x_0 = 1.1$, the first approximation to the reciprocal. Additional markers are at bit places 6, 8, and 12 after the binary point, indicating that three iterations are required, at precisions of 6, 8, and 12 bits. Two consecutive markers terminate the number.

The only moving particles in Fig. 9 are *sender* particles, denoted by SL and SR, whose job is to travel through the medium and send data bits to the left and right to begin an iteration of the loop. (The SL and SR particles are created by a chain reaction of previous collisions which will not

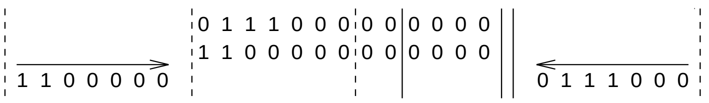

Figure 10: *Configuration just before the first multiplication.*

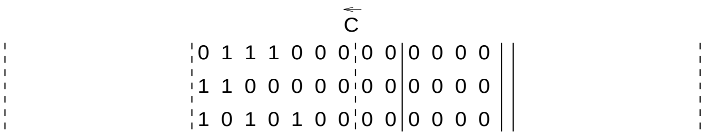

Figure 11: *Configuration just before subtraction.*

be described here.) Also present in Fig. 9 are *mirror* particles, denoted by squares, which "reflect" moving data bits; that is, a collision of a moving bit with a mirror generates a bit moving in the opposite direction and destroys the original bit.

An iteration of the loop proceeds as follows. First, the sender particles SL and SR collide with the two markers, as shown in Fig. 9, to generate two mirrors in place of the markers. Next, SL moves right, sending bits of y_0 to the left; at the same time, SR moves left, sending bits of x_0 to the right. The moving bits of the two numbers bounce off the first mirrors, pass through the second mirrors, and finally bounce off the mirrors represented by squares in Fig. 9. This prepares the medium for the multiplication y_0x_0. At this point all four mirrors are transformed to stationary markers. Figure 10 shows the configuration just before multiplication occurs. When the last bits of x_0 and y_0 collide with markers, the multiplication is finished, as shown in Fig. 11.

The next task is to calculate $2 - y_0x_0$. We do this simply by taking the 2's complement of y_0x_0, since $0 < y_ix_i < 2$. For this purpose we generate a special particle C when the last bit of y_0 collides with a marker (see Fig.

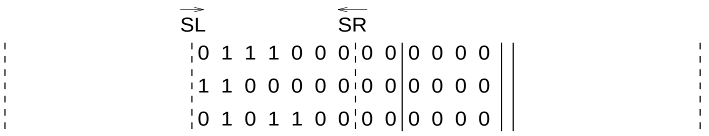

Figure 12: *Configuration just before the second multiplication.*

$\overrightarrow{SL}$							$\overleftarrow{SR}$					
0	1	1	1	0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0	0	0	0
0	1	0	1	1	0	0	0	0	0	0	0	0

Figure 13: *Configuration after the first iteration.*

11). The particle C is similar to a ripple-carry particle (see Figs. 2, 6, and 7). It first adds 1 to the complement of x_0y_0's first bit, then moves left through the rest of the bits, flipping them and adding the carry from the previous bits. Figure 12 shows the result.

All that remains is to multiply this result by x_0. This proceeds in the same manner as the multiplication of x_0 by y_0. Once this is done, an iteration of the loop is complete, and the bits of x_1 replace the bits of x_0. Figure 13 illustrates the configuration after completion of the first iteration.

The remaining iterations proceed exactly as described above, only with higher precisions, as determined by the markers in the template. When two consecutive markers are encountered, the division is finished, and we send a special particle through the template to restore the markers and mirrors for the next division.

Figure 14 shows a picture generated by a simulation of the division example just described. The simulated PM uses 38 types of particles and 79 rules, and is capable of realizing all the applications mentioned in this paper, including FIR and IIR filtering.

7 Discussion

There are three new contributions in this paper. First, we've described a CA implementation and VLSI layout for a PM, estimating speed and area. Then we showed that the general problem of determining whether rules can ever conflict is undecidable, but also that this is not a serious problem in practice. Finally, we showed how to implement a linear-time division algorithm in a PM, complementing a suite of linear-time implementations of basic arithmetic: addition/subtraction and multiplication/division. As pointed out in [7], these operations can be nested and combined for parallel execution in the cells of a PM.

In a way, a PM is a programmable parallel computer without an instruction set. What happens in the machine is determined by the stream of input

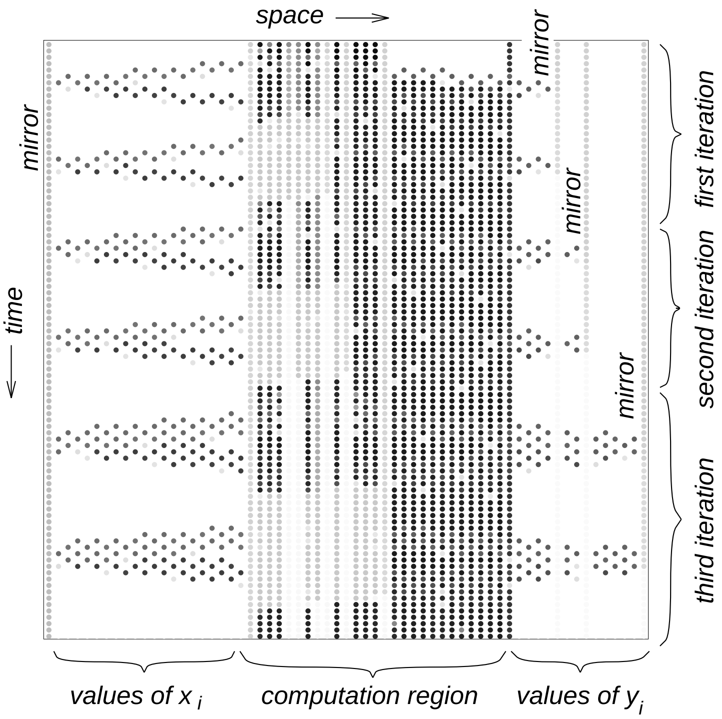

Figure 14: *PostScript generated by a simulation of the division implementation. Each cell is represented by a small circle whose shading depends on which particles are present in that cell. For clarity, only every seventh generation is shown. The example is the one described in the text, 1/7.*

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 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.

8 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," submitted to *Complex Systems*; Tech. Rept. CS-TR-478-94, Computer Science Dept., Princeton Univ., Dec. 1994.
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.