Collaborative Computation in Self-organizing Particle Systems

Alexandra Porter[1(✉)] and Andrea Richa[2]

[1] Stanford University, Stanford, USA
amporter@stanford.edu
[2] Arizona State University, Tempe, USA
aricha@asu.edu

Abstract. Many forms of programmable matter have been proposed for various tasks. We use an abstract model of self-organizing particle systems for programmable matter which could be used for a variety of applications, including smart paint and coating materials for engineering or programmable cells for medical uses. Previous research using this model has focused on shape formation and other spatial configuration problems (e.g., coating and compression). In this work we study foundational computational tasks that exceed the capabilities of the individual constant size memory of a particle, such as implementing a counter and matrix-vector multiplication. These tasks represent new ways to use these self-organizing systems, which, in conjunction with previous shape and configuration work, make the systems useful for a wider variety of tasks. They can also leverage the distributed and dynamic nature of the self-organizing system to be more efficient and adaptable than on traditional linear computing hardware. Finally, we demonstrate applications of similar types of computations with self-organizing systems to image processing, with implementations of image color transformation and edge detection algorithms.

1 Introduction

The concept of *programmable matter* was first defined by Toffoli and Margolus as a computing medium which can be used dynamically and in arbitrary amounts, controlled by both internal and external events [20]. Examples of programmable matter exist in nature, such as proteins closing wounds, bacteria building colonies, and the construction of coral reefs. These examples indicate potential applications of programmable matter, such as smart paint or coating materials for engineering, programmable cells for medical purposes, or adaptable and recyclable building blocks for everyday objects. These applications require tasks for which programmable matter is uniquely capable, such as shape formation and coating. However, they also require computations resembling those

This work was supported in part by NSF under Awards CCF-1353089 and CCF-1422603, and matching NSF REU awards; this work was conducted while the first author was an undergraduate student at ASU.

© Springer International Publishing AG, part of Springer Nature 2018
S. Stepney and S. Verlan (Eds.): UCNC 2018, LNCS 10867, pp. 188–203, 2018.
https://doi.org/10.1007/978-3-319-92435-9_14

done by traditional computers to process information and make decisions. Work so far using the geometric amoebot model for self-organizing particle systems has focused on spatial configuration, including demonstrating efficient programmable matter algorithms for shape formation, coating, and compression (e.g., [2,8,9]).

We introduce solutions using the amoebot model for basic computational tasks exceeding the capabilities of a single particle, including counting or number storage, and matrix-vector multiplication. Basic constructions for computational tasks can then be used as building blocks to solve more complex problems. Self-organizing particle systems have the potential to increase efficiency of algorithms by using dynamic spatial configurations of these computational building blocks to minimize communication costs. We describe and analyze a binary counter algorithm and a matrix-vector multiplication algorithm using the amoebot model. In order to illustrate how our algorithms can be used as part of more complex systems, we discuss concrete applications of our matrix-vector multiplication approach to the image processing tasks of color transformations and edge detection.

1.1 Amoebot Model

In the amoebot model, we represent the particle system as a subset of an infinite, undirected graph $G = (V, E)$, where V is the set of all possible positions a particle can occupy, and E is the set of all possible transitions between positions in V [7]. In the *geometric amoebot model* we impose an underlying geometric structure for G in the form of the *equilateral triangular grid*, as shown in Fig. 1(a). Each particle occupies either a single node (i.e., it is *contracted*) or a pair of two adjacent nodes (i.e., it is *expanded*) on the graph, and each node can be occupied by at most one particle at any point in time, as shown in Fig. 1(b). Two distinct particles occupying adjacent nodes are connected by a *bond* and we refer to such particles as *neighbors*. The bonds ensure the particle system forms a connected structure and are used for exchanging information.

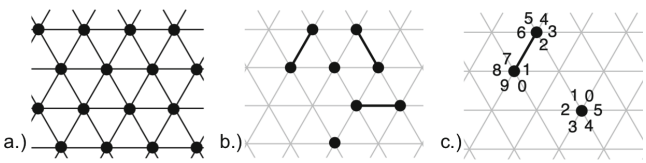

Fig. 1. (a) A section of G, where nodes of G are shown as black circles; (b) five particles on G; the underlying graph G is depicted as a gray mesh; a contracted particle is depicted as a single black circle and an expanded particle is depicted as two black circles connected by an edge; (c) labeling of bonds for an expanded particle and a contracted particle.

Each particle is *anonymous*, meaning it has no globally unique identifier. Particles may communicate with each neighbor by reading and writing to their

shared constant sized memory, which can equivalently be considered as the ability to pass a limited number of bounded-size tokens to adjacent particles.[1] Particles move by asynchronously executing a series of *expansions* and *contractions*. If a particle occupies only one node, it is contracted and can expand to an unoccupied adjacent node. An expanded particle can then contract to occupy only one of the two nodes it occupied while expanded.

We assume a compass-free model, meaning there is no global sense of orientation shared by the particles, and we assume that the particles do not share any underlying coordinate system in G. In the case of the triangular grid, each particle p fixes an arbitrary head direction, which specifies an adjacent edge e_{head} to p. We assume particles have shared *chirality* (sense of clockwise direction) and so they can label their ports in a consistent direction (note that in the presence of gravity, chirality follows naturally). Ports are labeled from 0 to 5 or from 0 to 9 depending on if the particle is expanded. Possible labelings for two nodes are shown in Fig. 1(c).

We assume an asynchronous, concurrent system of particles, where conflicts of movement (e.g., two particles trying to expand into the same empty node location) or shared memory (e.g., two adjacent particles trying to write concurrently onto their shared memory) are resolved arbitrarily so that at most one of the particles involved in the conflict "wins". Thus we can rely on the seminal results for the classical asynchronous model in distributed computing (see, e.g., [14]) that state that any asynchronous execution of the system, where conflicts are resolved arbitrarily, produces an equivalent outcome as a sequence of *atomic particle activations*. Hence, we can assume, without loss of generality, that at most one particle is active at any point in time. Under this model, we define:

Definition 1. *An* asynchronous round *is given by the elapsed time until each particle has been activated at least once.*

In our context, when a particle is activated it can perform an arbitrary bounded amount of computation using its local memory and the shared memory of its neighbors, and at most one movement.

1.2 Related Work

There are a number of existing solutions for programmable matter, which can be categorized as active and passive systems. In passive systems, the computational units have no ability to control their motion, so they move and bond only based on their structure and environmental conditions. Passive systems include DNA computing and tile assembly models, in which computation occurs as a result of tiles bonding together in ways controlled by the tile attributes (see, e.g. [16,22]. Work on tile assembly considers computational problems similar to those we study, including demonstration of a binary counter [18]. However, the specifications of those systems (passive motion, unlimited supply of tiles of any type, etc.) differ considerably from ours. Active systems consist of computational units

[1] For more details on our message sharing model, please refer to [7].

that control their actions, motions, and communications to accomplish specific tasks. Applications of active systems, including shape formation, coating, and compression, have been explored using robotic implementations (see, e.g. [13]). These applications have also been explored using abstract models (see, e.g. [5]), including the amoebot model (see [8] and the referencestherein), which is an active system. The amoebot system has also been used for the application of building convex hulls [6], which requires a binary counter or similar computational primitive.

Classical algorithms for distributed matrix multiplication include Fox's [10] and Cannon's [1]. These divide matrices into consecutive blocks to perform multiplication. More recent algorithms, including the Scalable Universal Matrix Multiplication Algorithm (SUMMA) [21] and Distribution-Independent Matrix Multiplication Algorithm (DIMMA) [4], further reduce the number of necessary operations. In SUMMA, the matrix is divided into rows and columns of blocks, and values are then broadcast down columns and across rows. DIMMA improves on this by adding pipelining to communication and taking advantage of a Least-Common Multiple strategy to reduce computation requirements. Our simpler algorithm for matrix-vector multiplication broadcasts values down columns of the matrix in a way similar to how values are broadcast in SUMMA and DIMMA.

In the field of computer vision and image processing, matrix multiplication is used to apply operators for fundamental tasks including determining gradient (see, e.g. [19]) and measuring color invariants such as luminance [11]. Basic color transformations operators, such as adjustments to brightness, saturation, and hue are also often used in image editing [12].

An application of these image operators is edge detection, which is an important problem due to its applications in feature extraction and recognition. The edge detection algorithm introduced by Canny uses a series of steps including smoothing, filtering, and thresholding to extract edges from an image [3]. Research has been done into how to implement this method efficiently, including a distributed GPU implementation [15].

1.3 Our Contributions

We address the very basic and general problems of *counting* and *matrix-vector multiplication*. We describe the image processing applications of *edge detection* and *color transformations*, as examples of applications that can use and benefit from our matrix-vector and matrix-matrix multiplication setup and algorithms. We assume each instance of these problems is fed into our particle system as a sequence of values passed through a seed particle. Results are stored distributed across the system, and can be output by each particle individually or passed to the seed to output the result as a data stream.

We present an algorithm for a basic *binary counter* using the amoebot model, and show that it counts to a value v in $O(v)$ *asynchronous rounds*. We also present a two-part algorithm for *matrix-vector multiplication* using the amoebot model. The first part of the algorithm is to self-organize particles to set up the input matrix and vector and the resulting vector entries. The second part of

the algorithm distributedly performs the actual multiplication (note that these two algorithmic components run concurrently and there is no need for synchronization). Let h and w denote the number of rows and columns of the matrix (unknown to the set of particles). We show that the *number of asynchronous rounds* it takes to *set up* the matrix and vector entries is $O(hw)$ and the *number of rounds required for matrix-vector multiplication* is $O(h + w)$. Extending this result by executing a sequence of matrix-vector multiplications, the *number of rounds required for matrix-matrix multiplication* is $O(y(h + w))$ with a second matrix of height w and width y, for a total of $O(hw+y(h+w))$ rounds including setup.

As an example of an application of our approach, we describe and analyze a simple implementation of *Canny edge detection* in image processing, which utilizes the setup algorithm introduced for matrix-vector multiplication. We show this implementation requires $O(1)$ *rounds to complete edge detection* after the $O(hw)$ setup is completed (again no synchronization between these two algorithmic phases is needed). Another sample application of our approach in image processing is that of *color transformation*, which is setup in the same way with $O(hw)$ rounds and then requires $O(y(h + w))$ rounds for multiplication. We also provide experimental results on actual implementations of the Canny edge detection algorithm and the color transformation algorithm we consider.

2 Preliminaries

In each of the problems considered here, we categorized particles as being either in the *structure* built for the operation or as *free* particles.

Definition 2. *At any point during the execution of the algorithm the* structure *refers to the set of particles recruited for use in some operations and assigned a specific role and position for that operation. They are in one of the states* {seed, matrix, vector, counter, prestop, result}.

Definition 3. *At any point during the execution of the algorithm, the set of free particles consists of those particles that are not yet assigned a specific purpose. They are in one of the states* {leader, follower, inactive}.

Free particles may eventually become part of the structure or remain available for other uses. As free particles they actively move to make themselves available to extend the structure if needed, but may continue moving indefinitely if they are not recruited. Particle states are defined as the corresponding algorithms are presented in Sects. 3, 4, and 5.

Tokens are small structures (of constant size) of data which are held by exactly one particle at a time during their existence. Tokens are treated as units or allowed to carry a value within the constant range determined by their storage size, depending on the algorithm. Respecting the particles' memory constraints, each particle holds at most a constant number of tokens at any time. Configurations and schedules are defined for a set of particles and will be used to analyze the progress of the entire system toward the final goal.

Definition 4. *A* configuration *of the particle system at a point in time consists of the set of state variables P_j for each particle j, including position, current state, and tokens held.*

We use $p_C(t)$ to describe the position of token t at configuration C: If particle j holds t in configuration C, then $p_C(t) = j$ (ownership of t is indicated in P_j). Tokens travel through a predefined sequence of nodes, regardless of which particles occupy those nodes during the execution of the algorithm.

Definition 5. *A* token path *of length m is a set of particles $P_{k_1}, P_{k_2}, ..., P_{k_m}$ such that P_{k_l} is adjacent to $P_{k_{l+1}}$ and one or more tokens travel from P_{k_x} to P_{k_y} passing through only particles in the path for some x, y with $1 \leq x < y \leq m$.*

We consider a configuration C to be valid if the system is connected (including both the structure and free particle set) and each particle is either contracted or expanded into adjacent positions with no single position occupied by two particles. When clear from context, we will refer to the particle j and P_j indistinctly.

In an asynchronous execution, the system progresses through a sequence of *asynchronous rounds* (Definition 1). When a particle P_j is activated during an asynchronous round, if it holds a token t it can pass t to any neighbor which has available token capacity at the time of the current activation of P_j.

3 Binary Particle Counter Algorithm

The first computational application of the amoebot model we analyze is a binary counter. The binary counter we describe here will also be used as a primitive for the matrix-vector multiplication algorithm presented in Sect. 4. In this implementation, the system contains only the seed particle and a set of initially inactive particles, already forming a line with the seed at the end at round 0.[2]

We denote the non-seed particles $P_0, ..., P_{n-1}$ such that P_0 is a neighbor of the seed particle, denoted S, and labeling follows the line of particles moving away from the seed. Each non-seed particle represents a digit of the counter, with the particle in line closest to the seed representing the least significant bit of the counter. Each P_j with $j < n-1$ receives counting tokens (treated as units) only from P_{j-1} (or S if $j = 0$). When P_j reaches its token capacity, here defined as two, it discards one token and attempts to send the other, representing a carryover, to P_{j+1}. The value of the system as a whole can then be calculated using the state of each digit particle to determine the value it represents.

The seed behaves as an interface to the counter. It receives activations from an external source to increment the counter, upon which it constructs new tokens and sends those to P_0 if there is space in the shared memory with P_0. Due to space limitations, the pseudocode describing this procedure appears in the full arXiv paper [17].

[2] If a line of particles is not readily available, one can easily build one following the algorithm presented in [9] concurrently with the binary counting procedure – i.e., there is no need for synchronization of the phases.

3.1 Runtime Analysis

All of our algorithms, presented in Sects. 3, 4, and 5, follow an asynchronous execution. However, for the analyses of these algorithms, we considered executions according to parallel schedules, since those are easier to handle and will provide a worst-case scenario in terms of number of rounds for asynchronous schedules. In a parallel execution, the system progresses through a sequence of *parallel rounds*.

Definition 6. *During one* parallel round *starting with configuration C and resulting in configuration C^*, one of the following is true for each particle p:*

1. *p occupies the same node(s) in C and C^*,*
2. *p occupies one node in C and expands to an additional adjacent node during the round,*
3. *p occupies two adjacent nodes in C and contracts to a single node during the round, leaving the other node empty in C^*, or*
4. *p occupies two adjacent nodes in C and contracts in a handover such that in C^* a different particle has expanded into one of the nodes p occupied in C.*

Additionally, for each token t, let P_k be such that $k = p_C(t)$. Then at the end of the parallel round one of the following is true:

1. *$p_{C^*}(t) = p_C(t)$,*
2. *if a particle $P_{k'}$ adjacent to P_k is below capacity in C, $p_{C^*}(t) = k'$, or*
3. *if there is a token path length d (labeled as particles $P_{k_1}, ..., P_{k_d}$), for each $1 \leq l \leq d-1$ the particle P_{k_l} in the path has a token t_l (such that $t = t_l$ for some l) which needs to move to $P_{k_{l+1}}$, and P_{k_l} has available token capacity, then $p_{C^*}(t_l) = p_C(t_l) + 1$ for each $1 \leq l \leq d-1$.*

Definition 7. *A movement schedule $(C_0, C_1, ...C_f)$ is a* parallel schedule *if each C_i is a valid configuration and for each $i \geq 0$, C_{i+1} is reached from C_i in exactly one parallel round.*

In asynchronous execution, the system progresses through a sequence of *particle activations*, meaning only one particle is active at a time. When activated, a particle can perform an arbitrary bounded amount of computation (including passing tokens) and make at most one movement. An *asynchronous round* is the elapsed time until each particle has been activated at least once. When a particle P is activated, if it holds a token t it can pass t to any neighbor which has available token capacity at the time of the current activation of P.

Definition 8. *A movement schedule $(C_0, C_1, ...C_f)$ is an* asynchronous schedule *if each C_i is a valid configuration and for each $i \geq 0$, C_{i+1} is reached from C_i by execution of one asynchronous round.*

We now provide a brief, high-level sketch of the proof that shows that a counter with n particles can count to v (where $v \leq 2^n - 1$) in $\Theta(v)$ asynchronous rounds (the proofs and more details can be found in our full arXiv paper [17]).

Lemma 1. *For any asynchronous particle activation sequence A, there exists a parallel schedule $\mathcal{P}$ such that the number of asynchronous rounds needed by the binary counter algorithm according to A is at most equal to the number of parallel rounds required by the algorithm following $\mathcal{P}$.*

We can then count the total number of bit flips that occur in the counter to get the result:

Lemma 2. *The parallel binary counter algorithm counts to the value v in $O(v)$ parallel rounds.*

Combining these two results, we get:

Theorem 1. *The asynchronous binary counter counts to the value v in $\Theta(v)$ asynchronous rounds.*

4 Particle Matrix Multiplication Algorithm

The next computational problem we solve using the amoebot model is matrix-vector multiplication. As before, the seed acts as a source of external input into the system. We suppose the system is initially unaware of the dimensions or values of the matrix and vector to be multiplied, so they will enter the system through the seed particle. The stream of information entering the system from the seed can contain values of matrix or vector entries (we assume each fits on a single particle), end of column markers, and end of vector markers. The seed particle at no point computes the dimensions of the problem since it receives values online in sequence from an external source. The seed then passes values, encapsulated in tokens, into the system as the algorithm proceeds. As these values are passed, the system "recruits" particles to represent the different matrix, vector, and result entries, by having the particles occupy the respective position in the system. We describe how the necessary matrix-vector result structure is built in Sect. 4.1. Below we give an abstract description of how the matrix-vector multiplication proceeds, assuming we have the necessary particles in place to perform the respective operations.

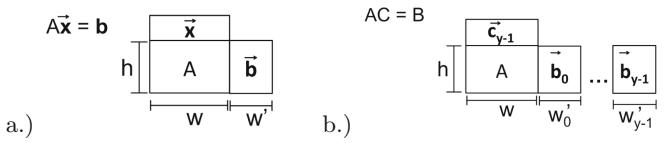

Fig. 2. (a) General matrix-vector multiplication A$\boldsymbol{x}$ setup for $h \times w$ matrix A and $w \times 1$ vector $\boldsymbol{x}$; (b) general matrix-matrix multiplication AC for $h \times w$ matrix A and $w \times y$ matrix C. Shown during final matrix-vector multiplication $A\boldsymbol{c}_{y-1}$.

Let A be a $h \times w$ matrix and $\boldsymbol{x}$ be a $w \times 1$ vector for some nonzero integers h and w. The result of the matrix vector multiplication $A\boldsymbol{x}$ is then $\boldsymbol{b}$, which is

stored using a set of the binary counters described in Sect. 3. The problem is streamed into the system in the order: values for each matrix column from top to bottom, left to right, followed by the values of $\boldsymbol{x}$ ordered from top to bottom. As vector values reach their final positions, vector particles also generate result counter tokens, which are passed along to determine how many particles should position themselves to store the results of the multiplication.

As shown in Fig. 2(a), particles assigned to represent values of $\boldsymbol{x}$ are positioned across the top of those representing matrix A, such that the line of matrix particles directly below a vector particle is the corresponding matrix column. The vector value is then passed down the column and used by each matrix particle it reaches to produce an individual product. Products are then passed across the row of matrix columns to where the set of result particles are positioned to store the product totals.

This algorithm can also be extended to complete matrix-matrix multiplication. To multiply matrices A and C, the setup is the same as before but with the first column of C, $\boldsymbol{c}_0$ replacing the vector $\boldsymbol{x}$. If C has a width of y, after each column $\boldsymbol{c}_i$ is multiplied by A, for $i < y$, we add a new set of results particles to store the vector $\boldsymbol{b}_i$. Thus the entire result matrix B can be stored as series of vectors $\boldsymbol{b}_0, \boldsymbol{b}_1, ..., \boldsymbol{b}_{y-1}$, as shown in Fig. 2(b).

The matrix, vector, and result particles do not know their indices relative to the whole system but can orient themselves such that they know which direction is across the matrix row and which direction is down the matrix column. To multiply a matrix by multiple vectors in a stream, this setup only needs to be executed once. If a finished notification is sent to the seed after each matrix-vector multiplication completes, an additional vector can be used without any changes to the matrix.

4.1 The Algorithm

We refine the notation of a configuration from Sect. 2 to specify the particles' functions in the final system. Let $C_i = (M_{0,0}, M_{0,1}...M_{0,w-1}, M_{1,0}...M_{h-1,w-1}, R_{0,0}, R_{0,1}, ..., R_{0,w'}, ...R_{1,0}...R_{h,w'}, V_0, V_1...V_{w-1})$ be the configuration at round i where $M_{u,v}$ is the configuration of the particle which will eventually be the matrix particle at position (u, v), $R_{u,q}$ will be a result particle at position (u, q) in the results matrix, and V_v is the vector particle at index v in vector $\boldsymbol{x}$. Let c be the token capacity of matrix, vector and result particles, and let m be the maximum value of a matrix or vector entry. We then use w' to denote the number of columns of results particles constructed, so $0 \leq u < h$, $0 \leq v < w$, and $0 \leq q < w'$. Enough result columns are constructed to hold the maximum possible number of tokens generated, so $w' = \lceil \log_c(m^2 w) \rceil$. Finally, we denote the minimum number of particles necessary to complete setup as n', so $n' = hw + w + hw'$. Since particles are given tasks on a first-come, first-serve basis, particles that remain free particles throughout execution do not have any effect on the correctness of the system.

Particles are categorized in configurations based on their final location, but are all initially free particles (except for the seed particle). At the start of exe-

cution, the spanning forest algorithm in [9] is used to organize the connected system of free particles into trees rooted on the seed particle (for completeness, we present the full spanning forest algorithm in [17] as well). Free particles adjacent to the seed are called *leaders* and all other free particles are *followers*, so each leader is the root of a tree of followers. Leader particles move along the surface of the structure (initially consisting of just the seed particle) until they are assigned a role and position in the structure. As leader particles move, they pull along their attached trees of follower particles. When follower particles become adjacent to the structure, they also become leaders and begin moving along the surface.

Flags are set from the seed, vector, and matrix particles to point to where a new particle needs to be added to the structure. As a free particle moves along the surface, it will stop and become part of the structure when one of these flags points to it. Result particles are similarly recruited by setting flags to point to where a particle may be needed based on the maximum possible values of the matrix and vector, but result particles have the option to leave the structure after multiplication has completed if they are not needed to represent the result.

Tokens travel in a predetermined direction in the set of matrix, vector, and result particles. For clarity, we extend the range of the position function $p(t)$ for token t to be ordered pairs representing position in a two-dimensional arrangement of system particles. Input matrix and vector entries are bounded such that an individual token can carry an input vector or matrix value.

Figure 3(a) conceptually shows a system in the process of executing the setup algorithm. Note that any notions of "up/down" and "left/right" are relative to the orientation passed to the system from the seed particle, and do not assume any absolute orientation of the system. At the depicted point in time, each of the matrix values $m_{0,0}, m_{1,0}... \ m_{h,0}, m_{0,1}, ...m_{u-4,v}$ has been streamed into the system through the seed, and assigned to a corresponding particle. For example, the value $m_{0,0}$ is assigned to particle $M_{0,0}$ at the upper left corner of the matrix. Additional matrix tokens (squares labeled t) hold the next three matrix values to be assigned positions.

The next value to be assigned to a token, $m_{u,v}$ is shown at the head of the stream of values entering the seed particle. A token holding $m_{u,v}$ will follow the same path as the other tokens depicted, across the row of vector particles $(V_0, ...V_v)$ to the furthest particle, V_v, that has been recruited so far, and then down the corresponding matrix column. The most recently added matrix particle, $M_{u-4,v}$, will be responsible for recruiting a new matrix particle from the set of free particles (not shown) to be $M_{u-3,v}$ and hold the value $m_{u-3,v}$. This process will continue until the last column is completed.

The last part of the value stream, shown in the left half of the stream entering the seed in Fig. 3(a), is the set of vector values. Vector values are assigned to the first vector particle they reach which does not yet have a value. As each vector value is assigned, a result counter token (treated as a unit) is generated and passed down the vector away from the seed. In Fig. 3(b) these are

Symbol	Meaning
V_u	Vector particle u
v_u	Value at u in vector
$M_{u,v}$	Matrix particle at row u, column v
$m_{u,v}$	Value at row u, column v of matrix
e_i	End of column token for column v
f_k	End of vector token for vector k
$R_{u,q}$	Result particle at column w , row u
r_q	Result particle counter digit w

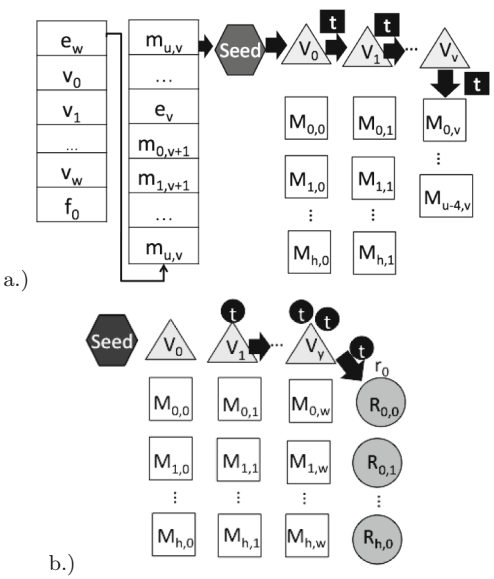

Fig. 3. Diagram of system setup and notation used. Shapes labeled $V_v, M_{u,v}$, or $R_{u,q}$ are particles and small squares/circles labeled t are tokens. In (a) the square tokens hold the matrix values $m_{u-3,v}$, $m_{u-2,v}$, and $m_{u-1,v}$ while in (b) the circular tokens are unit tokens without values. Free particles are not shown.

the circular tokens which are passed from V_{w-1} to $R_{0,0}$ such that $R_{0,0}, R_{1,0}, ...$ acts as a counter. When the farthest vector particle receives or generates result counter tokens it begins to recruit particles to start forming the result segment of the structure. When using multiple matrix-vector multiplications to perform a matrix-matrix multiplication, the existing result particles at the end of each matrix-vector product stop performing operations other than passing tokens. Then new sets of result particles are recruited for each matrix-vector multiplication in the sequence. Note that all phases of the algorithm are running *concurrently*, and there is *no synchronization between phases*. In order to prove the correctness and runtime of our algorithm, we will show that the different phases of our algorithm eventually correctly terminate in order.

Once the first end of vector marker, f_0, is received by the seed, setup will be completed and the multiplication can be executed, as summarized by the following steps:

1. each vector particle V_u passes its value v_u in a token to matrix particle $M_{u,0}$,
2. each matrix particle $M_{u,v}$ with value $m_{u,v}$ computes the product $m_{u,v} \cdot v_u$,
3. $M_{u,v}$ passes the vector value v_u to $M_{u+1,v}$ (if $M_{u+1,v}$ exists) so the vector value continues to move down the column,
4. $M_{u,v}$ passes a total of $m_{u,v} \cdot v_u$ result counter tokens to $M_{u,v+1}$ (or $R_{u,0}$ if $M_{u,v+1}$ does not exist), i.e. to the right across the row, and

5. each result particle $R_{u,v}$ accepts result counter tokens until at capacity, and then clears its counter and passes a carry over token to $R_{u+1,v}$ (executing the binary counter algorithm relative to its row of result particles).

Once multiplication has completed, the excess particles recruited to be result particles can be released back to being free, so that the final system configuration is minimal. Detailed pseudocode descriptions of the algorithms can be found in [17].

4.2 Correctness and Runtime Analysis

Similarly to the binary counter case, in order to show bounds on the runtime of the matrix multiplication system, we show bounds for a parallel schedule (Definition 7) and show that such a parallel schedule is dominated by the asynchronous schedule. For comparisons of progress in a system, we look at how close particles and tokens are to their final position nodes of the graph G. We give a high-level sketch of the proof here; please see [17] for the full proof.

Each matrix value token's final position is at the particle in the input matrix structure corresponding to the value. Each vector value token's final position is at the bottom matrix particle in the column under the vector particle corresponding to their value. Each product token's final position is in the counter representing the value of the result vector corresponding to the matrix row in which the product token originated. By comparing progress of tokens and particles toward their final destinations, we show:

Lemma 3. *For any asynchronous particle activation sequence A, there exists a parallel schedule $\mathcal{P}$ such that the number of asynchronous rounds needed by the matrix-vector multiplication algorithm according to A is at most equal to the number of parallel rounds required by the algorithm following $\mathcal{P}$.*

We first consider the setup phase, which includes particles moving into the structure configuration of *matrix*, *vector*, and *result* particles and the passing of tokens corresponding to inputted matrix and vector values. To show that system setup completes in $O(n')$ parallel rounds, we first show that our modified spanning tree primitive supplies particles to construction as necessary, so that:

Lemma 4. *Each matrix and result particle column takes $O(h)$ rounds to fill with particles in the parallel execution.*

Since $w + w' = O(w)$ columns need to be filled, we get:

Lemma 5. *The parallel matrix system setup completes in $O(n')$ rounds.*

Lemma 5, together with Lemma 3, implies:

Theorem 2. *The streaming matrix system setup completes in $\Theta(n')$ rounds.*

We next consider the actual matrix-vector multiplication process. Multiplication is initiated by each vector particle sending a token representing the value corresponding to its position down the column of the matrix, such that it is

seen by the matrix particle at each position which directly multiplies with that vector value. The amount of computation for the multiplication step is bounded by the time for tokens to travel down matrix columns and across matrix and result rows, so we have:

Lemma 6. *The parallel matrix-vector multiplier completes in $O(h+w)$ rounds.*

Theorem 3. *The asynchronous matrix-vector multiplier completes calculations in $\Theta(h+w)$ rounds.*

We can extend the result of Theorem 3 for matrix-matrix multiplications, namely:

Theorem 4. *The asynchronous matrix-matrix multiplier for matrices of dimensions $h \times w$ and $w \times y$ completes calculations in $\Theta(y(h+w))$ rounds.*

5 Image Processing Applications

Both the setup and multiplication steps of the matrix-vector multiplication algorithm can be used in image processing applications. Individual particles can be assigned to store individual pixels or small grids of pixels of an image, and their proximity to particles holding the corresponding adjacent pixels makes a number of localized image processing algorithms highly efficient.

We first discuss using the amoebot model to execute the Canny edge detection algorithm on a single channel image, meaning with a single scalar value for each pixel. Pixel values are streamed into the system as matrix values and a grid is established in the same way as in matrix-vector multiplication setup (Sect. 4.1), but without the requirement of result particles. Thus matrix particles store the image and can independently start to execute the algorithm as soon as they receive a value. The Canny edge detection algorithm includes local comparisons between pixel values and a matrix convolution operation to identify pixels most likely to be on the edges (see [17] for the full algorithm). Since these operations do not require information to travel between particles further than a constant distance, we have that:

Theorem 5. *Edge detection will complete in constant time after image setup.*

We next discuss how to use the amoebot model to execute image color transformations that use matrix-matrix multiplications. In this application, the input matrix has a row corresponding to each pixel of the original image and three columns corresponding to red, green, and blue. The transformation matrix is then streamed into the system as a sequence of vectors, each of which is multiplied by the matrix. The values in the transformation matrix determine the operation, such as filtering or saturation changes.

6 Simulation Results

As expected, Fig. 4(a) shows that the number of rounds required for the binary counter to reach a value v increases linearly with v. The results shown are each for a set of 10 particles arranged in a line before the system begins to execute. Value counted is the number of distinct counter tokens fed into the system by the seed particle.

For matrix-vector multiplication, the experiments in Fig. 4(b) show an approximately linear increase in the number of rounds for system setup and execution as the number of particles for the matrix-vector structure, n', increases.

In Fig. 4(c) we show two examples of edge detection on small images. The implementation discards an outer boundary at each step rather than using an inference method to fill in nonexistent values around edge pixels, so the images are padded with borders of zero-value pixels before inputted into the system. Results of edge detection are shown for a simple 10×10 shape and a more complex 16×16 image of a coin.

Figure 4(d) shows the results of color transformations by multiplying an image matrix by a 3×3 operator. The upper right example shows increased saturation, the bottom left shows conversion to grayscale, and the lower right shows color filtering.

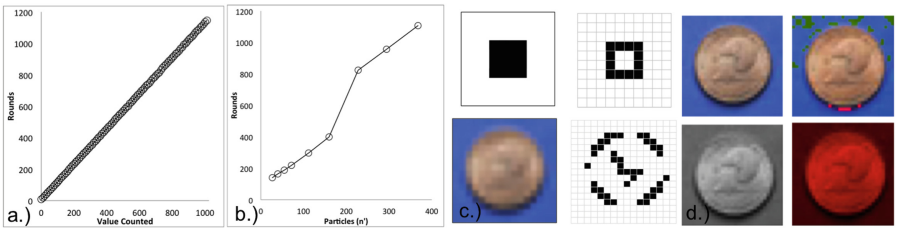

Fig. 4. (a) Asynchronous rounds per value of v counted in the binary counter; (b) asynchronous rounds per vector dimension in matrix-vector multiplication; (c) edge detection results (using red component of RGB); (d) color transformation results. (Color figure online)

7 Discussion

We have described basic computational algorithms that can be used in much larger computing applications, such as image processing tasks or building convex hulls [6]. Due to the limitations of the system receiving input through a seed particle, the binary counter requires $\Theta(v)$ asynchronous rounds to count to a value of v. The setup of the matrix-vector multiplication system is similarly limited by the input and time to assemble the structure of particles, so it requires $\Theta(n')$ rounds to setup the n' particles used to represent the matrix, vector,

and the vector of the product. However, the actual matrix-vector multiplication operations benefit from the parallelism of the system and each matrix-vector multiplication requires only $\Theta(h + w)$ asynchronous rounds (recall that h is the matrix height and w is the matrix width). This is especially beneficial for a matrix-matrix multiplication which requires only one execution of the setup algorithm (excluding the setup of additional results particles) to multiply an input matrix by each column of the other input matrix.

References

1. Cannon, L.E.: A cellular computer to implement the Kalman Filter Algorithm. Technical report, DTIC Document (1969)
2. Cannon, S., Daymude, J.J., Randall, D., Richa, A.W.: A Markov chain algorithm for compression in self-organizing particle systems. In: ACM PODC, pp. 279–288 (2016)
3. Canny, J.: A computational approach to edge detection. IEEE Trans. Pattern Anal. Mach. Intell. **6**, 679–698 (1986)
4. Choi, J.: A new parallel matrix multiplication algorithm on distributed-memory concurrent computers. In: HPC Asia 1997, pp. 224–229. IEEE (1997)
5. Das, S., Flocchini, P., Santoro, N., Yamashita, M.: On the computational power of oblivious robots: forming a series of geometric patterns. In: SIGACT-SIGOPS, pp. 267–276. ACM (2010)
6. Daymude, J.J., Gmyr, R., Hinnenthal, K., Kostitsyna, I., Richa, A.W., Scheideler, C.: Convex hull formation in self-organizing particle systems (2018). Manuscript in preparation
7. Daymude, J.J., Richa, A.W., Scheideler, C.: The Amoebot model (2017). https://sops.engineering.asu.edu/sops/
8. Derakhshandeh, Z., Gmyr, R., Richa, A.W., Scheideler, C., Strothmann, T.: Universal shape formation for programmable matter. In: ACM SPAA, pp. 289–299 (2016)
9. Derakhshandeh, Z., Gmyr, R., Strothmann, T., Bazzi, R.A., Richa, A.W., Scheideler, C.: Leader election and shape formation with self-organizing programmable matter. DNA **21**, 117–132 (2015)
10. Fox, G.C., Otto, S.W., Hey, A.J.: Matrix algorithms on a hypercube I: matrix multiplication. Parallel Comput. **4**(1), 17–31 (1987)
11. Geusebroek, J.M., Van den Boomgaard, R., Smeulders, A.W.M., Geerts, H.: Color invariance. IEEE Trans. Pattern Anal. Mach. Intell. **23**(12), 1338–1350 (2001)
12. Haeberli, P.: Matrix operations for image processing (1993). http://www.sgi.com/graca/matrix/index.html
13. Kernbach, S.: Handbook of Collective Robotics: Fundamentals and Challenges. CRC Press, Boca Raton (2013)
14. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)
15. Ogawa, K., Ito, Y., Nakano, K.: Efficient canny edge detection using a GPU. In: ICNC, pp. 279–280. IEEE (2010)
16. Patitz, M.J.: An introduction to tile-based self-assembly and a survey of recent results. Nat. Comput. **13**(2), 195–224 (2014)
17. Porter, A., Richa, A.W.: Collaborative computation in self-organizing particle systems. arXiv preprint arXiv:1710.07866 (2017)

18. Rothemund, P.W., Winfree, E.: The program-size complexity of self-assembled squares. In: ACMSTOC, pp. 459–468. ACM (2000)
19. Sobel, I.: An Isotropic 3 × 3 Image Gradient Operator. Machine Vision for Three-dimensional Scenes, pp. 376–379 (1990)
20. Toffoli, T., Margolus, N.: Programmable matter: concepts and realization. Physica D **47**, 263–272 (1991)
21. Van De Geijn, R.A., Watts, J.: SUMMA: scalable universal matrix multiplication algorithm. Concurrency Pract. Experience **9**(4), 255–274 (1997)
22. Woods, D.: Intrinsic universality and the computational power of self-assembly. Phil. Trans. R. Soc. A **373**(2046), 20140214 (2015)