Parallel Computing With Water

Alec Henderson[1], Radu Nicolescu[1], Michael J. Dinneen[1], TN Chan[2], Hendrik Happe[3], and Thomas Hinze[3]

[1] School of Computer Science,
University of Auckland, Auckland, New Zealand
[2] Compucon New Zealand, Auckland, New Zealand
[3] Department of Bioinformatics,
Friedrich Schiller University of Jena, Jena, Germany

Abstract. We further the work on a recently proposed membrane computing model which utilises decentralised water tanks interconnected by pipes with water flow controlled by valves. We demonstrate that such a systems is able to construct 'efficiently': 1) A programmable sequential random access machine (RAM) which we then extend to construct: 2) a programmable exclusive read exclusive write (EREW) parallel random access machine (PRAM).

Keywords: Water-based computing · Membrane systems · RAM machine · Fluidics · PRAM

1 Introduction

P systems first proposed by Gheorghe Păun in [1] are a parallel and distributed model of computation inspired by biological membranes. P systems are decentralised and typically evolve based on the content of a membrane. A water based system was proposed by [2] which contained many properties similar to that of membrane computing and was therefore described as a P system. The system was decentralised and the content of each tank would evolve in parallel similar to how P systems evolve.

In this work we demonstrate that the water system proposed in [2] and extended in [3] can construct a PRAM machine. This model has been shown to be Turing complete inherently proving that a RAM model can be built. However, this construction does not consider the complexity of such a construction. In this work we demonstrate that both a programmable RAM machine and a programmable PRAM machine can be constructed 'efficiently'.

2 Background

In this section we start by defining saturated arithmetic. We then define a RAM machine and show how the euclidean algorithm can be implemented in RAM instruction codes. Ending the section with the definition of the water computing model which we utilise throughout the remainder of the paper.

2.1 Saturated arithmetic

As discussed in [4] saturation arithmetic restricts operations to a fixed range. If a result exceeds the upper or lower limit the result is that limit. For example, if the range was [0,10] then, 5 * 5 = 10 and 10 - 20 = 0. If the upper and lower limits are $+\infty$ and $-\infty$ then it is standard arithmetic. For simplicity, we denote saturating addition as $\oplus$ and saturating subtraction as $\ominus$. Throughout the remainder of the text without loss of generality, we assume all saturated arithmetic except for control tanks, to have lower bound 0, and upper bound $+\infty$ where, control tanks have 0 and 1.

2.2 RAM

There have been many models of computation proposed with the 'sequential machines' typically being the deterministic Turing machine and the random access memory (RAM) machine [5]. In this work we shall focus on the RAM machine as they usually have more practical uses than the traditional Turing model.

A RAM machine consists of a finite program of m lines and a sequence of registers $r_1, r_2, ...r_n$ (perhaps an infinite sequence). The program of the RAM will consist of a sequence of operation codes and parameters. Based on the definition in [6], a RAM has the following operations:

1. $r_i \leftarrow C$: assign a constant value C to register i.
2. $r_i \leftarrow r_j \oplus r_k$: add the value of two registers j and k and assign to register i.
3. $r_i \leftarrow r_j \ominus r_k$: subtract from register j the value stored in k and assign to register i.
4. $r_i \leftarrow r_{r_j}$: get the value y from register j, then get the value from register y and assign to register i
5. $r_{r_i} \leftarrow r_j$: get the value y from register j, then get the value x from register i and assign y to register x.
6. TRA m $r_i > 0$ go to program line m if r_i greater than 0, otherwise go to the next line.

Where we use the item number as the op code as seen in Table 1.

We make the assumption that the output will be the values stored in the registers once the program has halted. A program halts when it has gone to a line number in the program which is not defined. For example, consider the following Euclidean algorithm pseudocode, for positive integers:

```
1 function gcd(a,b)
2     while (a ≠ b)
3         if (a > b) then
4             a ← a ⊖ b
```

Table 1. Operations and there corresponding opcodes.

Operation	Opcode
$r_i \leftarrow C$	1 i C
$r_i \leftarrow r_j \oplus r_k$	2 i j k
$r_i \leftarrow r_j \ominus r_k$	3 i j k
$r_i \leftarrow r_{r_j}$	4 i j
$r_{r_i} \leftarrow r_j$	5 i j
TRA m $r_i > 0$	6 m i

```
5           else
6               b ← b ⊖ a
7       return a
```

This pseudocode translates into the RAM code presented in Table 2.

Table 2. Euclidean algorithm implemented for RAM machine.

Line number	Operation	Opcode	Comment
1	$r_3 \leftarrow r_1 - r_2$	3 3 1 2	
2	TRA 6 $r_3 > 0$	6 6 3	Go to line 6 if $a > b$
3	$r_3 \leftarrow r_2 - r_1$	3 3 2 1	
4	TRA 8 $r_3 > 0$	6 8 3	Go to line 8 if $b > a$
5	TRA 10 $r_1 > 0$	6 10 1	Halt $a = b$ result is in r_1
6	$r_1 \leftarrow r_1 - r_2$	3 1 1 2	$a \leftarrow a - b$
7	TRA 1 $r_1 > 0$	6 1 1	Start the loop again
8	$r_2 \leftarrow r_2 - r_1$	3 2 2 1	$b \leftarrow b - a$
9	TRA 1 $r_2 > 0$	6 2 1	Start the loop again

We note that our RAM model has a fixed size program with m lines. This is able to implement RAM programs of length less or equal m. Noting that the machine halts when it gets to line $m + 1$. Hence if a program had less than m lines an additional line could be added to jump to line $m + 1$.

As the RAM model we have discussed is inherently sequential one can extend it to be parallel. A parallel RAM machine (PRAM) is one of the most well known parallel computing models. As defined in [7, 8] a PRAM consists of a set of processors $p_1, p_2, ...$ and a set of shared registers $r_1, r_2, ...$Each processor can be viewed as an individual RAM with its own program and sequence of registers.

Processors can only communicate via the shared memory via read or write operations. The processors execute in a synchronous manner with each step taking the same amount of time. Typically there are three models of PRAM discussed in the literature to model the read and write of the shared memory they are:

– Exclusive read exclusive write (EREW): only one processor can read or write to each shared register at a time.

– Concurrent read exclusive write (CREW): Any number of processors can read from the same shared register at the same time but only one can write to a each shared memory location.

– Concurrent read concurrent write (CRCW): Any number of processors can read and write the same shared register at the same time.

The CRCW PRAM is typically further defined based on different ways of handling concurrent writes these include[9]:

– Collision: A collision symbol is written. This does not give details about which processors caused the collision or what they were attempting to write.

– Common: Successful write only if all processors writing to the same location are writing the same value.

– Arbitrary: Only one arbitrary attempt is successful but, this choice will result in the same final result.

– Priority: The processor with the lowest IDs write is successful.

2.3 Water model

The water based system originally proposed in [2] and extended in [3] works by having a set of tanks interconnected with pipes which flow is controlled by valves. This model contains no central control as well as no loops (water flows in one direction). Formally the model is defined as:

$$\mathit{\Pi} = (T, T', F, E, R, L, C, V, S, P)$$

With its components:

– T finite set of tank identifiers.

– $T' \subset T$ finite set of control tank identifiers.

– $F \subset T'$ set of control tanks that when full indicate termination of the system.

– $E \in T \setminus T'$ the unique infinite sink of the system.

– $R \in T \setminus T'$ the unique infinite source in the system.

– $L : T \to \mathbb{N}_+$ The level which the tanks are built, the lower the number the conceptually higher the tank. Water can only flow from a tank with a lower number to one with a higher number. r is at 0 and s at ∞.

– $C : T \to \mathbb{N}_+ \cup \{\infty\}$ capacity of the tanks. Where we assume that value tanks are able to be unbounded. Of course for practical cases all value tanks will need to have finite capacity. Control tanks all have capacity 1 (they act as Boolean values).

– V finite set of valve identifiers.

– $S : V \to (T = \mathbb{N}_+ \cup T \neq \mathbb{N}_+)$ An expression from a valve identifier to check whether or not a tank has a certain volume.

– $P \subset T \times T \times \mathcal{P}(V)$ ($\mathcal{P}(V)$ denotes the power set over V) finite set of pipes where water flows from the first element to the second. A pipe (i, j, v) must have $L(i) < L(j)$, meaning water only flows in one direction ('down').

Typically this is modelled by either a set of equations or diagrams.

2.4 Examples

For self containment, we present four examples of water based functions, for a more in detailed description of these functions see [3]. These functions are then used in the construction of a RAM.

– Increment $f(x) = x \oplus 1$ (cf. Figure 1): The increment operator drains the two input tanks (value and control) into the result. Once the inputs are empty the output control tank is filled.

– Addition $f(x, y) = x \oplus y$ (cf. Figure 2): The addition operator works similar to the increment operator but, instead of the control tank draining to the result the value y does. Once all the inputs are empty the output control tank is filled

– Subtraction $f(x, y) = x \ominus y$ (cf. Figure 3): The subtraction operator drains from both x and y into the infinite sink until, y is empty. Once y is empty any remaining water drains from x into the result tank. Once all inputs are empty the output control tank is filled.

– Inplace copy $f(x) = x$ (cf. Figure 4): Input tank x drains into result tank x_1. Whilst x drains tank a fills from the infinite source. Once x is empty a stops filling and the control tank is drained into the infinite sink. Once the control tank is drained the content of a (which stores the value of the original input) is drained into the original input tank x. Once a is empty again the output control is filled.

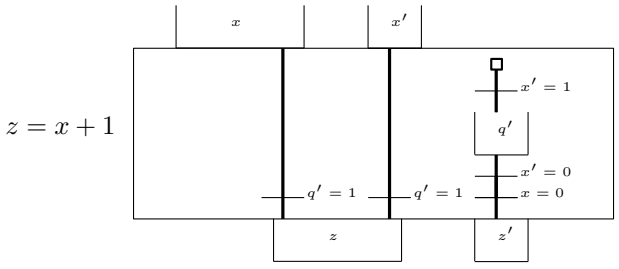

Fig. 1. A diagram representing the increment operator function $z = x \oplus 1$.

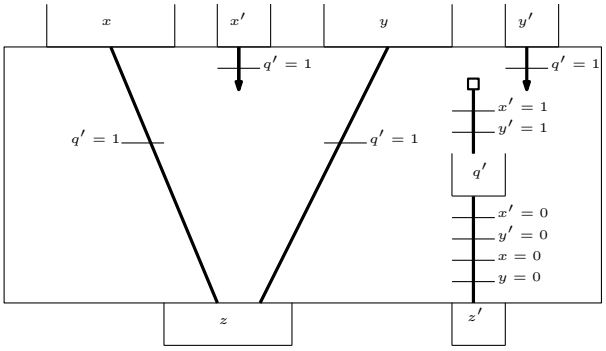

Fig. 2. A diagram representing the controlled saturating addition operator $z = x \oplus y$.

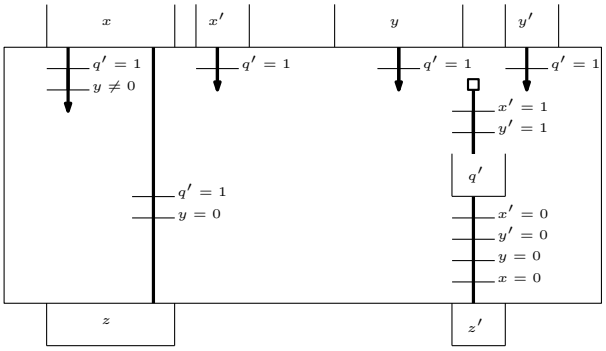

Fig. 3. A diagram representing the controlled saturating subtraction operator $z = x \ominus y$.

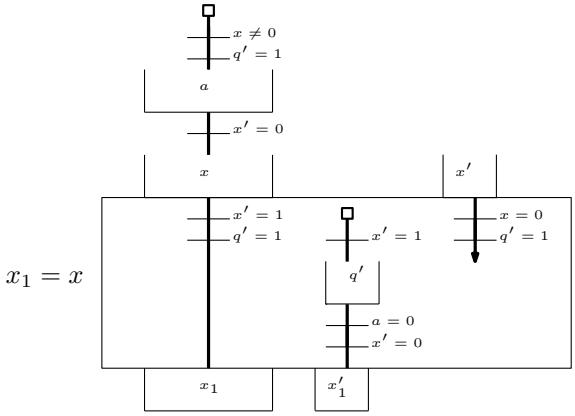

Fig. 4. A diagram representing the inplace copy function $i(x) = x$.

3 Constructing a programmable RAM machine

Constructing a programmable RAM machine using water can be broken into phases. The first phase is to read the line number that we are up to. Using the op code for that line run a function which executes that function. After that has been done check whether the line number exceeds the program line count; if it does then the RAM halts, otherwise it will do the process again. To make it easier to follow we shall also break up the construction into modules which can then be pieced together to form the entire RAM.

3.1 Execute line L of the program

Here we assume that the program being executed will be stored in water tanks $p_{1,1}, p_{1,2}, p_{1,3}, p_{1,4}, p_{2,1}, p_{2,2}, p_{2,3}, p_{2,4}, ..., p_{m,1}, p_{m,2}, p_{m,3}, p_{m,4}$. Where $p_{i,j}$ denotes the ith line with parameter j, noting that we assume that each line of code will have one op code followed by three parameters; the third tank contents will be ignored for operations of two parameters.

To start the program at line L and continue executing until we reach line $m+1$ we utilise the tank system presented in Figure 5. For brevity we have presented an arbitrary program line $p_{o,j}$ but this can be expanded for all program lines by changing the valve for $p_{i,k}$ to $L = i$ for the ith line. For example, the tanks presented in Figure 5 for the euclidean algorithm would initially contain the volumes presented in Table 3.

Noting that to run a function we use the tank system presented in Figure 6. The tank system presented in Figure 6 is not repeated, only one instance exists for a RAM machine. But the line inputs $p_{i,j}$ all drain into the inputs p_1, p_2, p_3, p_4.

Table 3. Initial volumes of water for tanks presented in Figure 5 for Euclidean algorithm. Where we denote a tank i stores volume j by $i(j)$

$L(1)$ $L'(1)$ $a'(0)$ $s(0)$ $s'(0)$ $d'(0)$ $c'(0)$ $q'(0)$ $x'(0)$ $f'(0)$ $h'(0)$
$p_1(0)$ $p'_1(0)$ $p_2(0)$ $p'_2(0)$ $p_3(0)$ $p'_3(0)$ $p_4(0)$ $p'_4(0)$ $l(0)$ $l'(0)$

i	$p_{i,1}$	$p'_{i,1}$	$p_{i,2}$	$p'_{i,2}$	$p_{i,3}$	$p'_{i,3}$	$p_{i,4}$	$p'_{i,4}$
1	3	0	3	1	0	0	2	0
2	6	0	6	0	3	0	0	0
3	3	0	3	0	2	0	1	0
4	6	0	8	0	3	0	0	0
5	6	0	10	0	1	0	0	0
6	3	0	1	0	1	0	2	0
7	6	0	1	0	1	0	0	0
8	3	0	2	0	2	0	1	0
9	6	0	2	0	1	0	0	0

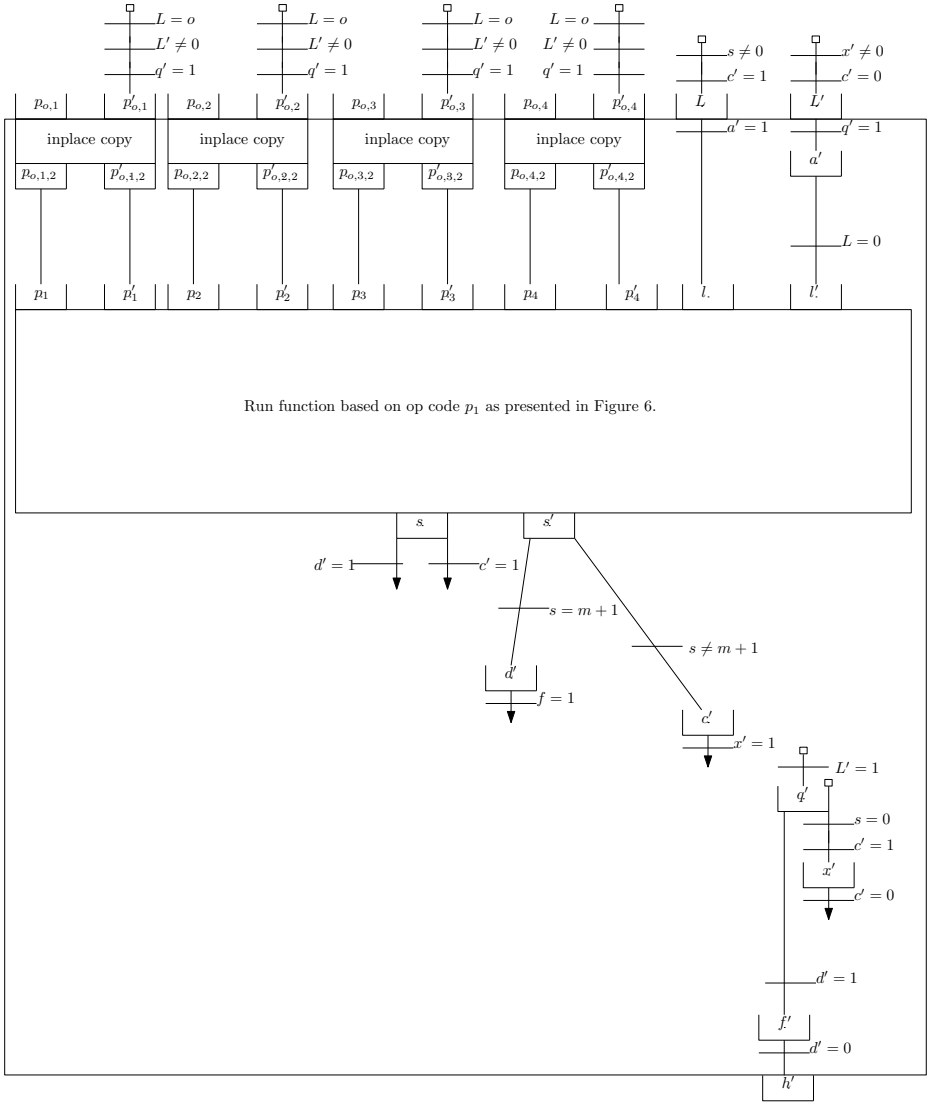

Fig. 5. A diagram representing the an outer loop of a RAM. Where the input L is the line to start the programs execution (typically line 1). After the operation has completed tank s will contain the next line to execute. Tank s will then be drained and L filled with that contents. Once the program has ended tank h' will be full. Noting that we have shown only for one arbitrary program line o.

3.2 Executing an operation code

Once the line number is read and operation number selected it is passed into the inner function presented in Figure 6. This will utilise the op code stored in p_1 and decide which operation to execute. For simplicity, we have presented an arbitrary operation X, this can be expanded for all operations by taking $X = 1, 2, 3, 4, 5, 6$.

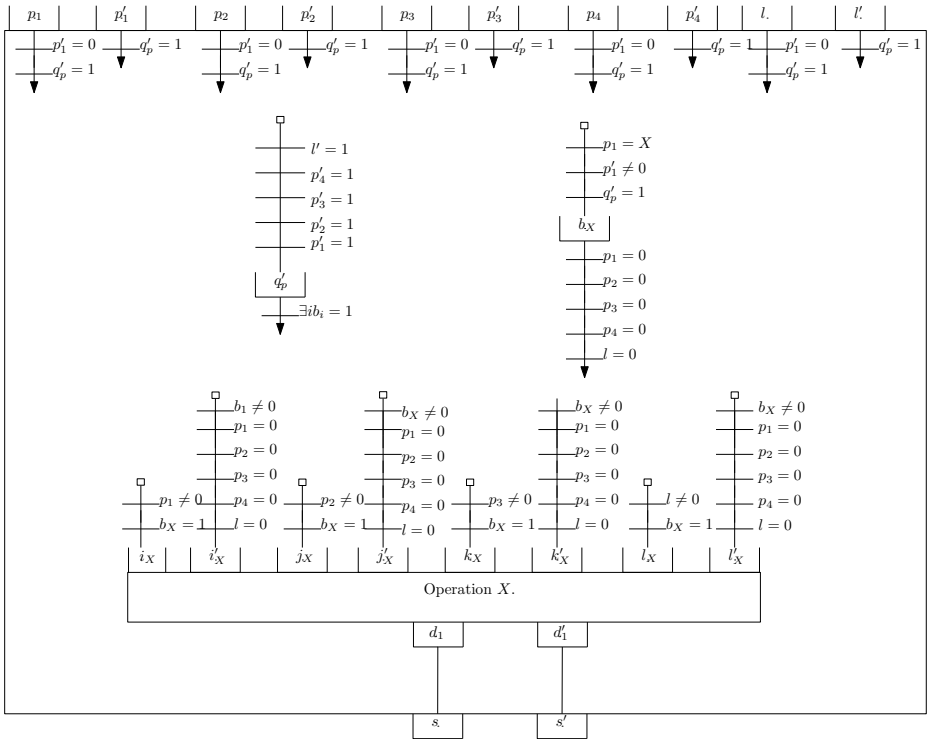

Fig. 6. A diagram representing which of the 6 functions will be executed. Where for simplicity we have shown for an arbitrary opcode X where $X \in \{1, 2, 3, 4, 5, 6\}$. Selected operations are presented: Operation 1 in Figure 8, Operation 2 in Figure 9, Operation 4 in Figure10 and Operation 6 in Figure 11. Noting that $\exists i b_i = 1$ is a shorthand to describe six pipes from q'_p each with one valve as shown in Figure 7.

As described earlier a RAM can be constructed using six basic operations. However, to simplify these operations we utilise the fact that these operations can be viewed as a sequence of read and writes from registers.

Utilising read and write operators, we construct the base operations. For brevity, we only show functions 1, 2, 4, and 6; functions 3 and 5 can be straightforwardly derived from functions 2 and 4, respectively.

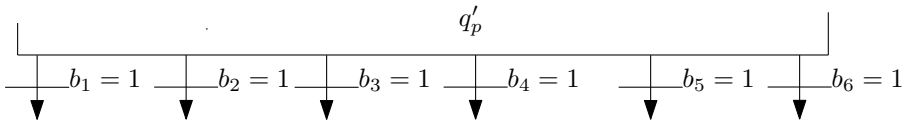

Fig. 7. Diagram to show the expanded version of the shorthand $\exists i b_i = 1$ for tank q'_p.

Although operation one is near identical to the write operation it is important to note that the operations need to also return the new line number after completing hence we use the increment function. Operation 1 can be viewed in Figure 8.

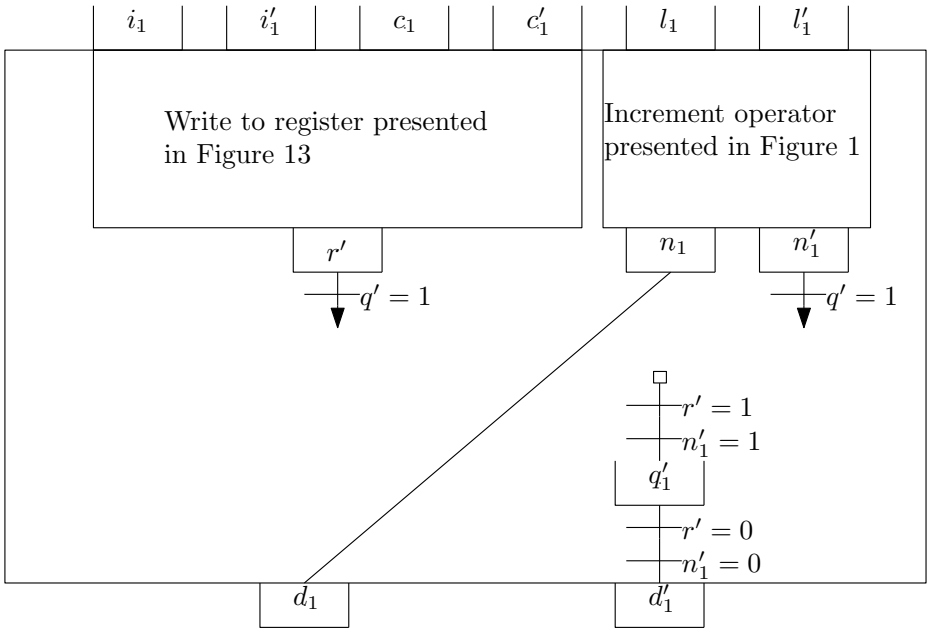

Fig. 8. A diagram representing operation 1: $r_i \leftarrow C$. Returns $l \oplus 1$ where l is the program line number.

Operation 2 can be viewed in Figure 9 where we have left out the line number increment. The line increment can be done just as we did in the first operation.

Operation 4 can be viewed in Figure 10.

Operation 6 can be viewed in Figure 11.

We implement the read and write functions in Figure 12 and Figure 13 respectively. We note that these two functions are reading and writing to the same set of registers so the registers $r_1, ...r_n$ are shared between the figures.

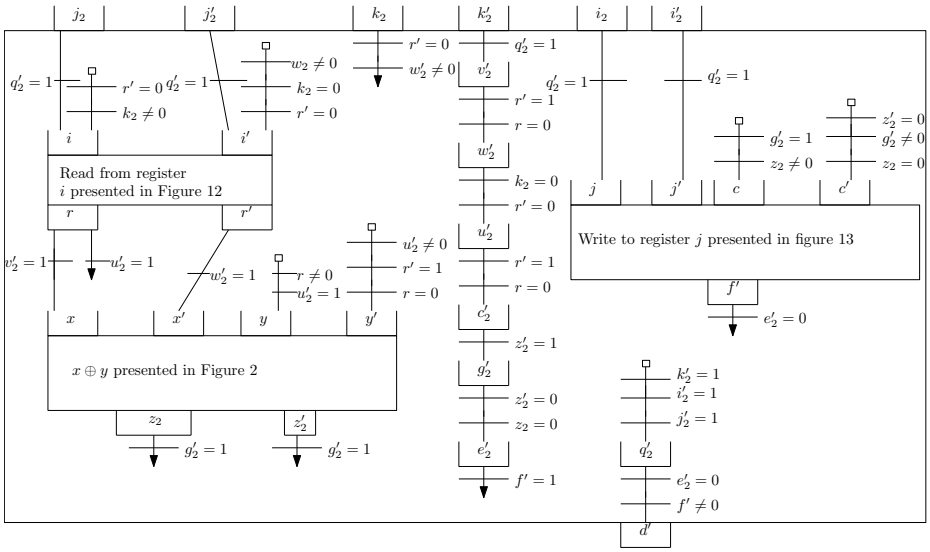

Fig. 9. A diagram representing operation 2: $r_i \leftarrow r_j \oplus r_k$.

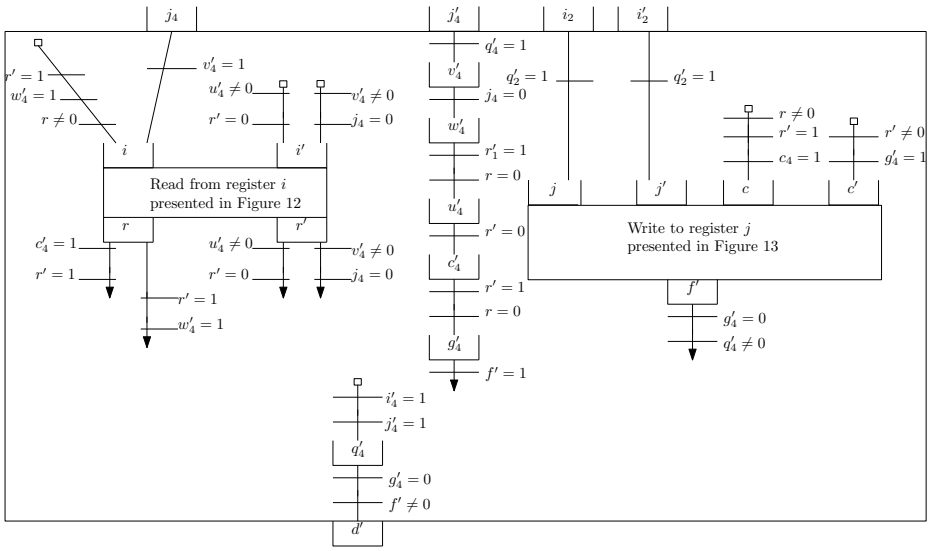

Fig. 10. A diagram representing operation 4: $r_i \leftarrow r_{r_j}$.

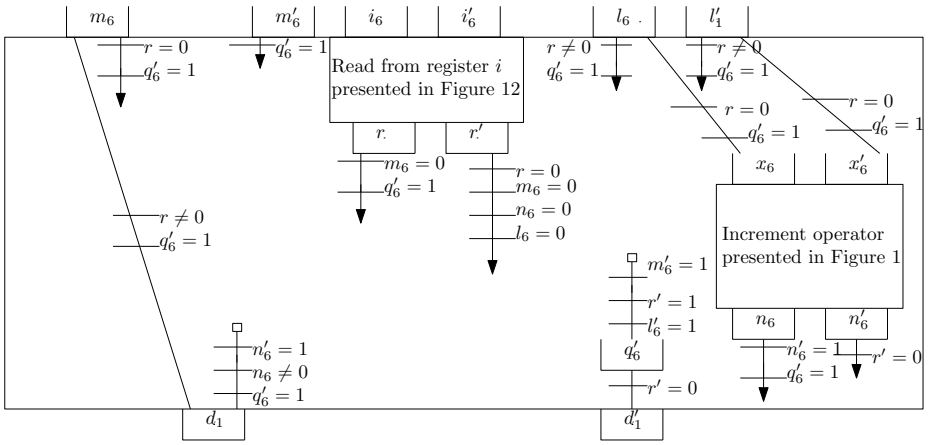

Fig. 11. A diagram representing operation 6: TRA $m\ r_i > 0$.

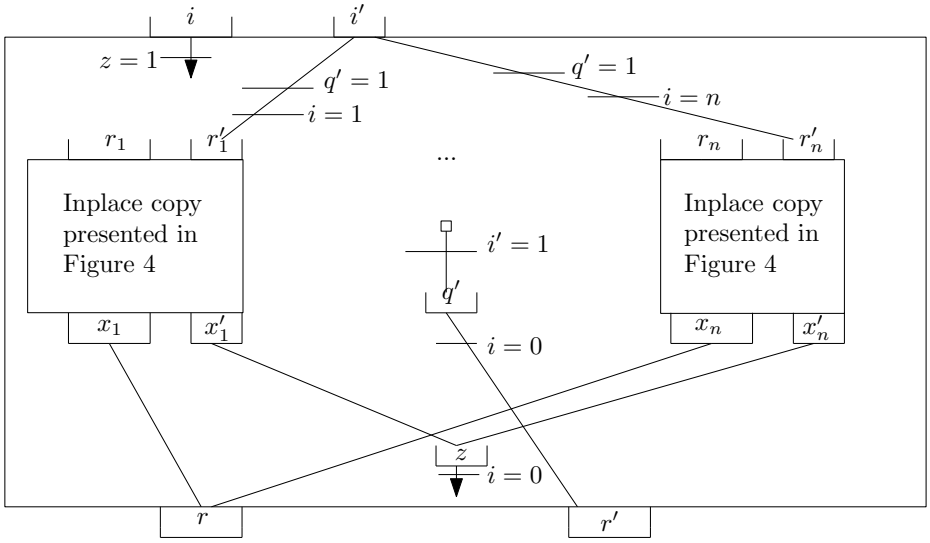

Fig. 12. A diagram representing read from register i.

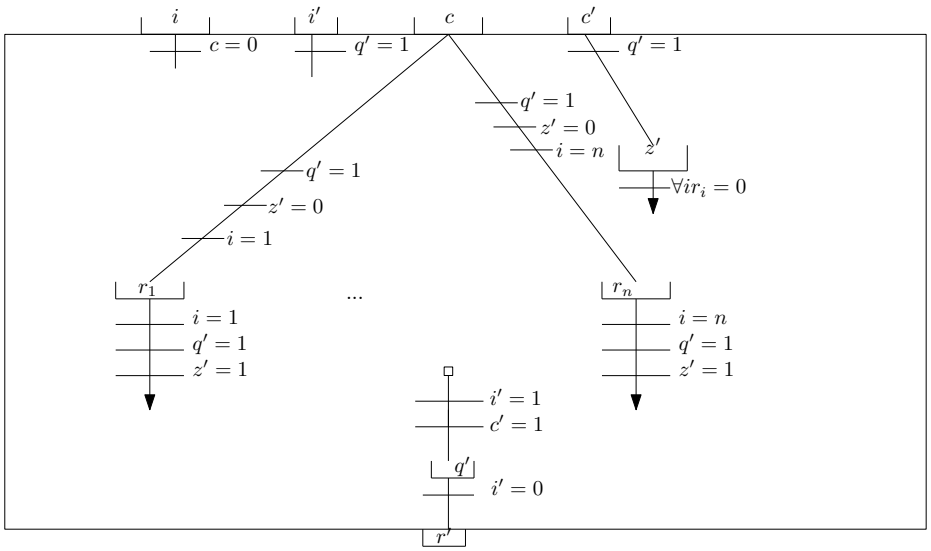

Fig. 13. A diagram representing write to register i the value $c : r_i \leftarrow c$. Where the $\forall$ is a shorthand to mean n valves with each valve being $r_i = 0$.

4 Extending to PRAM

Extending to a EREW PRAM we note the following:

1. The only way for processors to communicate is via read/writes to the shared memory.
2. The output of the PRAM will be the content stored on the shared memory once all processors halt.
3. At each time step if any two processors try to read and or write from the same shared memory location the result will be undefined.
4. Each operation will be run synchronously, i.e. at every time step each processor fully complete one program line.
5. We assume that each processor has its own finite program and a set of local registers.

In this section we extend our previously constructed RAM model to be a single processor. We note currently our RAM model defined earlier does not have operations to read/write to shared memory. But more importantly it does not satisfy the synchronous behaviour required for operations. In this section we shall first describe adding shared read and writes to our previously discussed model. We shall then describe adding a synchronous lock to ensure each processor evaluates each line of its program at the same time.

Denoting shared registers ρ we define the following additional operators:

– $r_i \leftarrow \rho_j$: Read from shared memory and store in local register.

– $\rho_i \leftarrow r_i$ Read from local register and write to shared register.

To define these new functions we note that they are a simpler version of the indirect operation presented in Figure 10. With only one read but, instead of the read from local register r we define a read from shared memory ρ.

We extend the reading and writing of the local registers presented in Figure 12 and Figure 13. To extend it we have each processor $1, 2, ..., p$ have its own input, output, and control tanks for the shared read and writes. The diagram presented in Figure 14 shows how an arbitrary register X reads from shared memory.

The extension to the read can similarly be used for the write which we omit for brevity. It is important to note for a EREW PRAM at any one time step only one of the processors may read or write to a register. If multiple reads, writes or a combination are done on the same shared memory the computation will have an unexpected result.

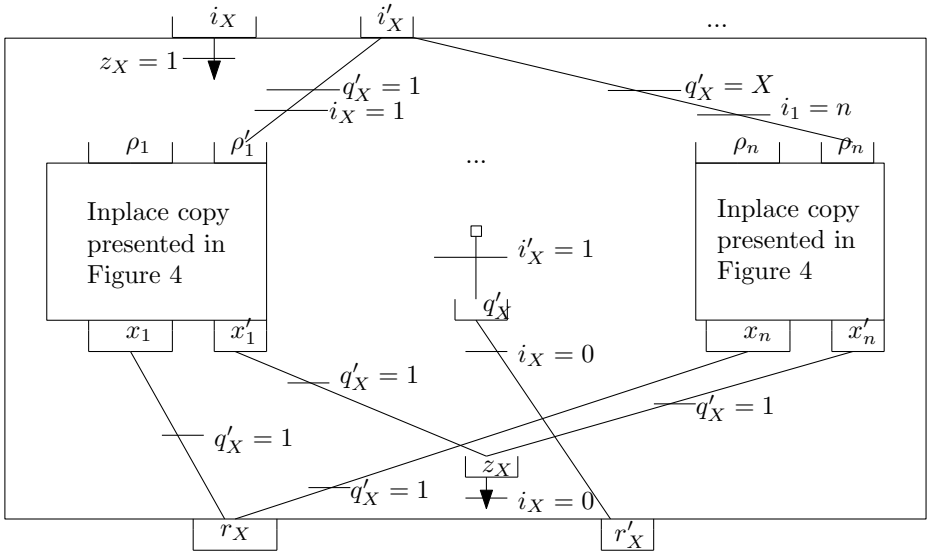

Fig. 14. Parallel read for arbitrary processor X.

To ensure that each line of code is run synchronously each processor must wait until all other processors have completed there current line. To achieve this we alter the outer program presented in Figure 5 to that presented in Figure 15 and Figure 16.

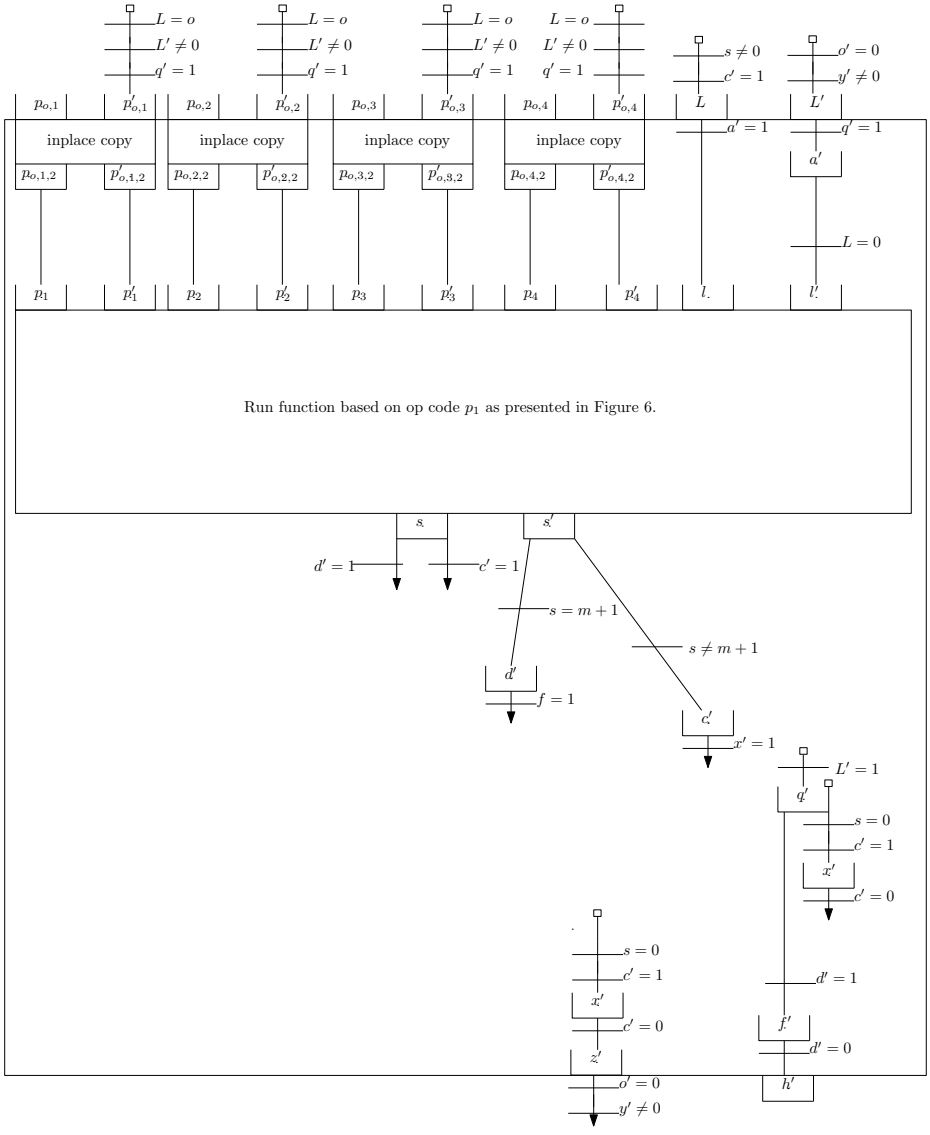

Fig. 15. Parallel outer program that will only loop based on the synchronisation presented in Figure 16

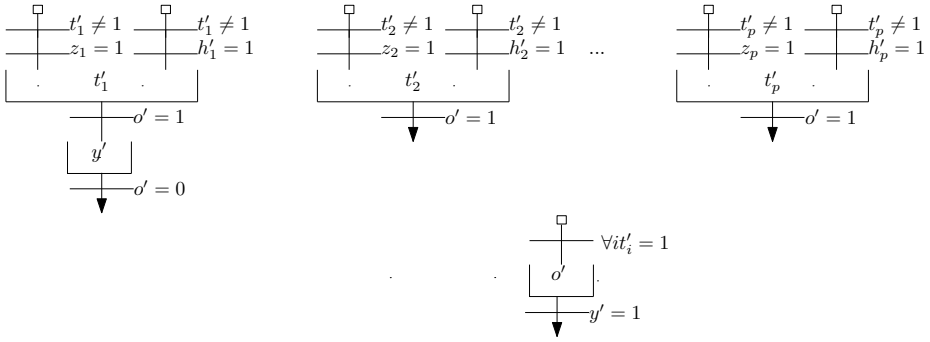

Fig. 16. The synchronisation scheme which ensures that only after all other processors have done an operation may they go to the next line. Noting that we use the shorthand $\forall it'_i = 1$ which expands to the valves presented in Figure 17

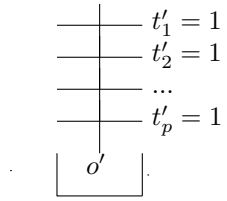

Fig. 17. Expanded version of $\forall it'_i = 1$.

5 Conclusion and future work

Using the previously defined water system [3] we have constructed: 1) a programmable RAM machine and 2) a programmable EREW PRAM machine. This demonstrates the non centrally controlled model is inherently parallel and can model one of the most well known parallel computing models. However, we have only modelled the least powerful of the PRAM models leaving future work to look at allowing concurrent reads and/or writes. Of course with this modelling many well known results of traditional parallel computing transfer to this model.

Another open problem is the cost-based minimisation of the number of valves and pipes.. Future work could look at physical realisations of this system and determining which is more 'expensive' based on a well defined measurement of cost. Physical realisations of the system could also have many other benefits such as for education.

References

1. G. Păun, "Computing with membranes," *Journal of Computer and System Sciences*, vol. 61, no. 1, pp. 108–143, 2000.
2. T. Hinze, H. Happe, A. Henderson, and R. Nicolescu, "Membrane computing with water," *Journal of Membrane Computing*, vol. 2, no. 2, pp. 121–136, 2020.

3. A. Henderson, R. Nicolescu, M. J. Dinneen, T. Chan, H. Happe, and T. Hinze, "Turing completeness of water computing," report CDMTCS-554, Centre for Discrete Mathematics and Theoretical Computer Science, University of Auckland, Auckland, New Zealand, July 2021.
4. F. Zappa and S. Esculapio, *Microcontrollers. Hardware and Firmware for 8-bit and 32-bit devices*. LIGHTNING SOURCE Incorporated, 2017.
5. I. Parberry, "Parallel speedup of sequential machines: A defense of parallel computation thesis," *SIGACT News*, vol. 18, p. 54–67, Mar. 1986.
6. S. A. Cook and R. A. Reckhow, "Time bounded random access machines," *Journal of Computer and System Sciences*, vol. 7, no. 4, pp. 354–375, 1973.
7. E. Gafni, J. Naor, and P. Ragde, "On separating the EREW and CREW PRAM models," *Theoretical Computer Science*, vol. 68, no. 3, pp. 343–346, 1989.
8. P. B. Gibbons, "A more practical PRAM model," in *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pp. 158–168, 1989.
9. F. E. Fich, P. Ragde, and A. Wigderson, "Relations between concurrent-write models of parallel computation," *SIAM Journal on Computing*, vol. 17, no. 3, pp. 606–627, 1988.