Membrane Systems and Distributed Computing

Gabriel Ciobanu*, Rahul Desai, and Akash Kumar

National University of Singapore, School of Computing
Department of Computer Science
gabriel@{comp.nus.edu.sg,info.uaic.ro}

Abstract. This paper presents membrane systems as an appropriate model for distributed computing, an efficient and natural environment to present the fundamental distributed algorithms. We support the idea that P systems can become a primary model for distributed computing, particularly for message-passing algorithms. We present the core theory, the fundamental algorithms and problems in distributed computing. We focus on an example describing an immune response system against virus attacks. The example is implemented using a P system library created by the authors to simulate the main functions of a P system, and an MPI program that takes advantage of the highly parallel features provided by the model. The program uses distributed leader election and synchronization algorithms.

1 Membrane and Molecular Computing

Formal language theory has been used as a basis for developing theoretical computational models related to DNA sequences and molecular processes. These developments reveal some theoretical facets of *molecular computing* related mainly to the computational power of the new systems, generative capability, complexity, and universality.

Membrane computing is based on *membrane systems* or P systems, a new class of distributed and parallel computing devices introduced in [7]. The approach is based on hierarchical systems: finite cell-structures consisting of cell-membranes embedded in a main membrane called the *skin*. The membranes determine *regions* where *objects*, elements of a finite set, and evolution rules can be placed. The objects evolve according to given *rules* associated with a region. Objects may also move between regions. A *computation* starts from an initial configuration of the system, and terminates when no further rule can be applied. A software simulator of membrane systems is presented in [3].

A membrane structure is usually represented by a Venn diagram, and it can be mathematically represented by a tree or by a string of matching parentheses. Hierarchical systems are well-known structures in computer science, and the notion of computation based on evolution rules is common. The interpretation of the computation is rather new: the result of a computation is a multiset of objects collected in the output cell or sent out of the system. The behaviour of the

* Corresponding author

G. Păun et al. (Eds.): WMC-CdeA 2002, LNCS 2597, pp. 187–202, 2003.

whole system is obtained by combining the resulting multisets, or considering the multiplicity of objects present in a specific output membrane of a final configuration. Păun has introduced initially three alternatives to look at membrane systems. Starting from these approaches, several variants were considered. Each of these variants has been shown to generate recursively enumerable sets. All these results emphasize a new computing paradigm inspired by a basic function of biomembranes.

2 The Distributed Nature of the Membrane Systems

An interesting aspect of the membrane systems is that they are distributed and parallel computing devices. To complement the overwhelming majority of researches in this area dealing with computability results, this paper aims to show that the membrane systems represent an appropriate model for distributed computing. We emphasize on the algorithmic aspects related to the distributed systems computational power provided by membrane systems. The algorithms are mainly presented in [2]. In this paper we focus on an example describing an immune response system against virus attacks. The example uses various distributed algorithms, and it is implemented using a P system library and an MPI program emphasizing the highly parallel features provided by the model.

Another approach presenting the membrane systems as distributed and parallel computing devices is described in [4], where a new version of P systems called Client–Server P systems is introduced. The new version is devoted to the interaction between components and is similar to the network client-server model. The Client–Server P systems are based mainly on the power of communication between membranes, and they have the same expressive power as Turing machines.

The link between membrane systems and distributed and parallel computing is not difficult to comprehend. If we associate each membrane with a host on a network, then the membrane containing a few such membranes is congruent to a subnet, and subsequently their parent membranes represent larger networks. Finally, the skin membrane could represent the World Wide Web. You can imagine that routing in Internet is similarly to message passing within a P system. Specifically, membranes within the same membrane can directly communicate with each other (by sending objects within the membrane), while if two membranes in different parent membranes need to pass objects to each other, then this scenario is congruent to sending the packet to a router which connects the two networks. Having established the basic premise of the paper, we will conclusively show how P systems provide a natural and efficient representation of distributed systems.

In the context of parallel and distributed systems, the algorithmic issues studied in the sequential model require a fundamental rethinking. In parallel systems, a problem is solved by a tightly-coupled set of processors, while in distributed systems it is solved by a set of communicating (asynchronous) processes. From the viewpoint of the theory of distributed computing, which is restricted to the

Message Passing and Shared Memory model, the P systems model provides a different perspective, which is natural and easy to relate to. In P systems the process of passing objects is similar to message passing; moreover, membranes in the same biological system could have access to the same DNA, or could have access to the same blood stream, making it possible to relate the model to a Shared Memory system as well. Both these extensions are simplistic and natural, and hence it is not difficult to adapt the already existing theory of distributed computing to the P systems model, something that this paper aims to achieve.

We emphasize on the algorithmic aspects related to the distributed systems computational power provided by membrane systems. In membrane systems the process of passing objects through membranes in both directions is similar to message passing. We consider a system of communicating membranes with antiport carriers, and the main meaning regarding this choice is that the membranes *send* and *receive* information. We present some basic algorithms of distributed computing, starting with algorithms for broadcast, convergecast, flooding, leader election, mutual exclusion in distributed systems, the fault tolerant systems and the consensus problem.

We present the fundamentals of distributed computing, starting with algorithms for broadcast, convergecast, the leader election problem, the mutual exclusion problem in a distributed environment, and finally the consensus problem and fault tolerance. Some of the presented algorithms are used in an example describing an immune response system against virus attacks.

3 Basic Algorithms in P Systems

The field of distributed computing is notoriously difficult, mainly due to uncertainties introduced by limited local knowledge, asynchrony, and failures. The fundamental issues underlying the design of distributed systems are related to communication, coordination, synchronization and fault tolerance. Mastering fundamental algorithmic ideas and techniques, someone is able to design correct distributed systems and applications. We present here some basic algorithms over communicating membrane systems, algorithms representing the core theory of distributed computing. For more information and notation about fundamental distributed algorithms, see [1].

Collecting and dispersing information is central to any system. Even though P systems share the same DNA, local information often has to be passed around in the system. This is where the message passing model becomes relevant. Two basic algorithms in any message passing model are *broadcast* and *convergecast*. Another common algorithm discussed in a message passing model is *flooding*. This algorithm constructs a spanning tree when a graph is given. P systems themselves essentially have a spanning tree structure. Each membrane of the tree has exactly one parent, except the skin membrane which represents the root.

Broadcast: Consider a system in which a membrane m_i has to send some object to all the membranes of the system. Clearly, we have two cases of broadcast here - one in which m_i is the skin membrane or the root node, and secondly when m_i is any node. It is not difficult to see that the second case is a generalization of the first one. We shall start with the algorithm for the simple case (when m_i is the skin membrane), for easy understanding.

Very often in a distributed computing model, the root node has to broadcast a certain message, say M. Here is the prose description of the algorithm. The skin membrane, m_s first sends the message to all its children. Upon receiving a message from its parent, the membrane sends the message to all its children, if any. Here is the formal pseudocode for this algorithm.

Algorithm: Skin Membrane Broadcast

Initially M is in transit from m_s to all its children.

Code for m_s:

1. Upon receiving no message:
2. terminate

Code for $m_i, 0 \leq i \leq n-1, i \neq s$:

3. Upon receiving M from its parent:
4. send M to all its children
5. terminate

A skin broadcast algorithm for P systems has a message complexity of $n-1$ and time complexity l, when l levels of membranes are present.

Message Complexity: As evident from the above algorithm, the message is communicated from a parent membrane to a child membrane exactly once. The algorithm terminates after sending M once to all its children. Thus, the total number of messages passed is equal to the number of edges in the spanning tree structure. Since, the number of edges in a spanning tree with n nodes is $n-1$, we obtain the result that $n-1$ messages are passed in a P system with n membranes. Therefore, the message complexity of this algorithm is $O(n)$.

Time Complexity: In every admissible execution of the skin broadcast algorithm, every membrane at level l, i.e. at a distance of l edges from the root node in the spanning tree, receives the message M in l time. Thus a P system with l levels of membranes will have a time complexity of l. This corresponds to a depth of l in a spanning tree configuration. In the worst case, when the spanning tree is a chain, there can be at most $n-1$ levels for a system with n membranes. This shows that the time complexity of any P system for skin broadcast is $O(n)$.

3.1 Generalised Broadcast

Having seen broadcast for the skin membrane, we shall now move on to a more general broadcast in which any membrane can broadcast a message. Each membrane m_i which needs to broadcast sends the object M to its parent and all

children (if any). A membrane m_j, upon receiving a message from its parent, sends it to its children. If it receives the message from its child, then it sends the message to all its other children, and parent. The algorithm is given as follows.

Algorithm: Generalised Broadcast

Say, membrane m_a, $0 \leq a \leq n-1$ needs to send the message to all the membranes in the system.

Code for m_a:

1. if $(a \neq s)$
2. send M to its parent
3. send M to all children

Code for m_i, $0 \leq i \leq n-1, i \neq a$:

4. Upon receiving M from its parent:
5. send M to all children
6. Upon receiving M from its child:
7. if$(i \neq s)$
8. send M to its parent
9. send M to all children

A generalized broadcast algorithm for P systems has a message complexity of $n-1$ and a time complexity of $l+k$, when l levels of membranes are present and the membrane at k^{th} level broadcast. Message complexity is similar to the skin broadcast algorithm, and it is $O(n)$. A P system with l levels of membranes will have a worst case time complexity of $2 \times l$. This means that the time complexity for generalized broadcast is $O(n)$. More details are presented in [2].

3.2 Convergecast

The broadcast problem mentioned above aims at dispersing information held by a membrane to other membranes of the system. Convergecast, on the contrary, aims at collecting information from all the membranes of the system to the skin membrane. Many variants of the problem are available, for example, forwarding the sum of all the values held by membranes, or forwarding the maximum value, etc. In a general convergecast algorithm, instead of the result of a particular operation, all the values are forwarded. In a generalized convergecast variant, the size of message can increase as the message progresses to the skin membrane. For simplicity we shall consider the algorithm of forwarding the sum of all the values held by membranes.

As it can be seen, unlike broadcast which is initiated by the membrane that wishes to disseminate information, convergecast is initiated by the *leaves*, i.e. membranes which contain no inner membranes. This algorithm is recursive, and requires each membrane m_i to forward the sum of values held by its membrane. In other words, the sum of the subtree rooted at it. A membrane collects all

the values held by its inner membranes, and computes the sum including its own values. This sum s_i is then forwarded to its parent membrane. Clearly, each membrane has to receive a sum from each of its children before it can forward the sum to its parent. The pseudocode for the algorithm is given below.

Algorithm: Convergecast

Code for leaf membranes:

1. Starts the algorithm by sending its value x_i to its parent.

Code for non-leaf membranes m_i with k children:

2. Waits to receive messages containing sums $s_{i_1}, s_{i_2}, \ldots, s_{i_k}$ from its children $m_{i_1}, m_{i_2}, \ldots, m_{i_k}$.
3. Computes $s_i = x_i + s_{i_1} + \ldots + s_{i_k}$
4. if $(i \neq s)$
5. Sends s_i to its parent.

The analysis of this algorithm is analogous to the skin broadcast algorithm, since the only difference in the two is the direction of message flow. As for the skin broadcast algorithm, the message complexity of the algorithm is $n-1$. The time complexity of the algorithm is $O(n)$, since at most $n-1$ levels may be present in a P system with n membranes. Therefore there is a convergecast algorithm for P systems with message complexity $n-1$ and time complexity l, when l levels of membranes are present.

4 Leader Election in P Systems

The existence of a leader in a P system can often simplify the task of coordination among the membranes. It might often be useful to have a leader (other than the skin-membrane as the default). It might also be the case that the criterion for leadership may not always be met by the skin-membrane. The leadership election problem, generally refers to the general class of symmetry breaking problems. The most general variant of it requires exactly one node from a system of many initially similar nodes to declare itself the *leader*, while the others recognize the leader and declare themselves *not-elected*.

Theorem 1. *It is impossible to solve the leadership election problem in a system where the membranes are anonymous.*

The idea behind this impossibility result is that the symmetry between the membranes can be maintained forever if the membranes are anonymous (they are very similar). Without some initial asymmetry provided by unique identifiers, symmetry cannot be broken and it is impossible to elect a single leader: if one membrane is elected, then so are all the membranes. Therefore, we assume that every membrane in the system has one unique identifier *id*. An algorithm is said to be *uniform*, if it does not depend on the number of membranes. And conversely, *non-uniform* algorithms rely on the knowledge of the number of membranes.

4.1 A Simple Leader Election Algorithm

The most straightforward way to solve the problem is that every membrane sends a message with the maximum *id* among all its children (and itself) to its parent, and waits for a response from its parent. The skin membrane in turn, would reply to all its children with a message containing the maximum *id* that it received. Ultimately, one membrane (which receives its own id back) will be elected leader.

Algorithm: Leader Election

Initially: *elected* = false, *children* = set of children membrane,
and *parent* = the parent membrane.
For every membrane m_i

1. If *children* = empty, send *id* to *parent*
2. Upon receiving id_j from all children,
   $winner = max(id_1, id_2, \ldots, id_n, id)$
3. If $parent \neq null$
   then send *winner* to *parent*
   else (it is the skin membrane)
   send *winner* as *leader* to all children
4. Upon receiving message *leader* from parent
   if $leader = id$ then *elected* = true
5. If *children* = empty, terminate
   else send *leader* to all children and terminate.

The message complexity of the above algorithm is $O(n)$, since every link between the parent and the child is used to exchange 2 messages. In a system with n membranes, there are $n-1$ such links. Thus, the message complexity is $O(n)$. The leadership algorithms in asynchronous rings have a lower bound of $O(n\,logn)$. Whereas, in the synchronous case, a message complexity of $O(n)$ can be achieved at the cost of the time-complexity [1]. Generally, the P systems are structured like a tree, already providing a sufficient edge to break-symmetry as compared to a ring, where every substructure of the ring is symmetrical.

5 Mutual Exclusion in Shared Memory

Shared memory is another major communication model and we shall see how P systems can be used effectively to model this as well. In a shared memory, processors communicate via a common memory area that contains a set of *shared variables*, which are also referred to as *registers*. In natural computing using P systems, as said above, membranes have access to the same blood stream and mutual exclusion can thus be easily modelled.

5.1 Formal Model of Shared Memory Systems

Before we proceed to understand mutual exclusion algorithms, we need to define the formal model for a shared memory system. We assume we have a system with n membranes $m_0, m_1, \ldots, m_{n-1}$ and m registers or shared variables $R_0, \ldots, R_{m-1}$. Each register (shared variable) has a *type* which can specify the values which a register can take, the operations that can be performed on it, the value returned by each operation, and the updated value of the register as a result of the operation. Beside this, an initial value for the register has to be specified. Another important distinction in shared memory systems comes when analyzing algorithms. Unlike in object passing models, object complexity is meaningless. On the other hand, *space complexity* becomes relevant in this model. Space complexity can be measured in two ways: number of registers used, and number of distinct values the register can take. Measuring time complexity of shared memory algorithms is still a current research area and we only focus on whether the number of steps in the worst case running of the algorithm is infinite, finite, or bounded.

5.2 The Mutual Exclusion Problem

The *mutual exclusion* problem is one where different membranes need access to a shared resource that cannot be used simultaneously. Some relevant terms in the section are given below:

- *Critical Section:* Code segment that has to be executed by at most one membrane at any time.
- *Deadlock:* A situation in which when one or more membranes are trying to gain access to a critical section, and none of them succeeds.
- *Lockout:* When lockout occurs, a membrane trying to enter its critical section never succeeds.

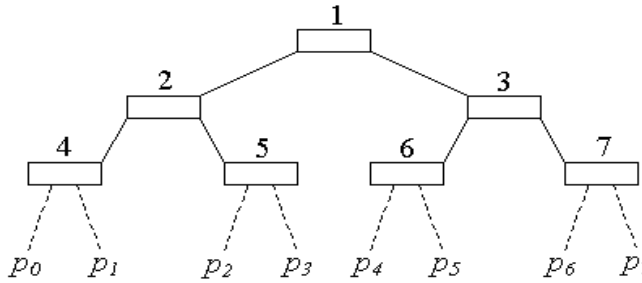
A membrane might need to execute some additional code segments before and after the critical section. This is to ensure mutual exclusion. The relevant sections of the code are:

- *Entry:* Code section where the membrane prepares to enter critical section.
- *Critical Section:* Code section which has to be executed exclusively.
- *Exit:* Code section executed when a membrane leaves the critical section.
- *Remainder:* Remainder of the code.

The desired properties are mutual exclusion, no deadlock and no lockout. *Mutual Exclusion* is achieved when in every configuration of every execution at most one membrane gets access to critical section. *No Deadlock* is achieved in every admissible execution, when membranes are in the *entry* section, at a later stage, a membrane is definitely in the critical section. *No Lockout* is achieved when in every admissible execution, a membrane trying to enter the critical section, eventually gets an entry.

5.3 Achieving Mutual Exclusion

The *test-and-set* and *read-modify-write* are powerful primitives used to achieve mutual exclusion. Another commonly known algorithm is the *Bakery algorithm* which uses *read/write* registers [1]. An appropriate algorithm for P systems would be the *tournament algorithm*. The conventional tournament algorithm can be modified to suit P systems. The tournament algorithm is a bounded mutual algorithm for n processors. It is based on selecting one among two processors at every stage, and thus selecting one among n processors in $\lceil log_2 n \rceil$ stages. The algorithm is recursive and every processor that succeeds at a stage climbs up the binary tree. The processor reaching the root gains entry to the critical section. An example with 8 processors is presented below:



Membranes within a parent membrane can select one among themselves using the tournament algorithm, and the parent can then forward the request to its parent in turn. The one membrane that succeeds to reach the skin level gains entry to the critical section. The number of rounds k in this algorithm will be equal to the number of levels l of the system. The pseudocode for the algorithm is mentioned below. The conventional algorithm is used by the term `tournament` and the list of membranes *list* is passed to it. The algorithm returns the *id* of the membrane that succeeds in the tournament algorithm.

Algorithm: Tournament

l represents the maximum depth of the system,
m_j is the parent membrane of m_i.

```
1. for k = l downto 1
2.     1) all parent membranes m_i at depth k:
3.         if (list ≠ φ)
4.             w = tournament (list)
5.         else
6.             w = −1
7.         end if
8.         send w to its parent, if any.
9.     2) all leaves m_i at depth k:
10.        if (need access to critical section)
11.            w = i
12.        else
```

13. $w = -1$
14. end if
15. send w to its parent m_j, if any.
16. 3) all parent membranes m_i at depth $k-1$
17. for $p = 0$ to $c-1$ (c is the number of children membranes)
18. receive w_{i_p} from m_{i_p}
19. if ($w_{i_p} \neq -1$)
20. add w_{i_p} to $list$
21. end for
22. end for

The first step is not executed in the algorithm when $k = l$. This is because there are no parent membranes at depth l since that is the maximum depth of the tree. The above algorithm is good appropriate for P systems and it provides mutual exclusion with no deadlock and no lockout.

6 Fault Tolerant Consensus

For a system to coordinate effectively, often it is essential that every membrane within the system agree on a common course of action. With the help of leadership election, and a subsequent broadcast/flooding, it is possible for a consensus to be reached. This section discusses the consensus problem and fault tolerance (viz. reaching a consensus despite having failures within parts of the system).

6.1 The Consensus Problem

Consider a system in which each membrane m_i needs to coordinate with the rest of the processors and choose a common course of action, i.e. agree upon a value for the variable *decision*. A solution to the consensus problem must guarantee the following:

- *Termination*: In every admissible execution, all the non-faulty nodes must eventually assign some value to *decision*.
- *Agreement*: In every admissible execution, all the non-faulty nodes must not decide on conflicting values.
- *Validity*: In every admissible execution, all non-faulty nodes must make the correct decision, i.e. must choose the correct value for *decision*. In other words, that if the consensus problem in question is choosing the maximum value from a certain set of numbers, then the *decision* must actually be the maximum value from the given set of input values.

Clearly the consensus problem is an important one, and the process would be disturbed in the presence of nodes which behave in an undesirable manner. However, within certain restrictions, it is possible to achieve a fault-tolerant consensus.

6.2 Failures

A failure is said to occur when a membrane behaves abnormally. There are two basic types of failures. *Simple failures* are when some membranes within the membrane system just stop functioning and do not ever recover, but wrong operations are never performed. The *Byzantine failures* are when some faulty membranes may behave in an unpredictable manner, contrary to a process which would help to reach a consensus.

The Simple Failure Case
The most important parameter which needs to be determined is f, the maximum number of membranes that can fail so that the consensus may still be achieved. Such a system is called an f-resilient system. We have the following results:

Lemma 1. *In every execution at the end of $f + 1$ rounds, all non-faulty membranes have the same set of values to base their decision upon.*

Theorem 2. *It takes an upper bound of $f + 1$ rounds to solve the consensus problem with simple failures in an f-resilient system.*

Algorithm: Consensus
Initially, every membrane m_j has some value x_j which it needs to send to all other membranes and ultimately reach a consensus based on these values.

In every round k, $(1 \leq k \leq f + 1)$, m_i behaves as follows:

1. Send x_i to all membranes within parent's membrane
2. Receive x_j from m_j.
3. Add x_j to an array (vector) V.
4. If $k = f + 1$ make decision based on the values stored in V.

The Byzantine Failure Case
This type of failure is more severe. The case is called the Byzantine failure because of a metaphorical description of a plan of action taken by several divisions of the Byzantine army (i.e. with traitors) to attack an enemy city [5]. In the Byzantine case, the faulty membranes behave arbitrarily and even maliciously. In a f-resilient Byzantine system there exists a subset of at most f "Byzantine faulty" membranes. It becomes difficult to distinguish between a functional and a Byzantine faulty membrane, because unlike a membrane that crashes and simply stops sending objects, a Byzantine faulty membrane continues to send objects which may hamper the consensus process that requires membranes to agree on a common action based on their possible conflicting inputs. It is known that a consensus can be reached only if less than a third of the processors are Byzantine faulty processors [5]. The result is also true for membranes.

Theorem 3. *In a system with n membranes, having f Byzantine membrane, there is no algorithm to solve the consensus problem if $n < 3f$.*

Theorem 4. *In order to reach consensus in an f-resilient system, every non-faulty membrane must send at least $f + 1$ objects to all other non-faulty membranes to meet the requirements of the consensus problem.*

7 Example and Implementation

In this section we present an example describing an immune response system against virus attacks. This example is implemented using a membrane system library to emulate the main functions of a membrane system, and Message Passing Interface library that takes advantage of the highly parallel features provided by the model. Message Passing Interface (MPI) is still the most popular environment in the field of parallel computing, though many new parallel languages and tools were introduced.

When a virus enters a cell, it tries to destroy the host cell and all the surrounding cells by periodical replication and propagation. The human immune system is in charge of producing suitable antibodies which can counter-attack the virus. In the event that the antibody is not present, it needs to be transported (through intercellular communication) from the cell producing it to the place where it is required. The survival of the cell depends on the availability of these antibodies.

7.1 Terms

- Immune Response: The immune response is the way the body recognizes and defends itself against microorganisms, viruses, and substances recognized as foreign and potentially harmful to the body.
- Antibody: Antibodies are special proteins that are part of the body immune system. White blood cells produce antibodies to neutralize harmful germs called antigens.
- Virus: An infectious particle composed of a protein capsule and a nucleic acid core, which is dependent on a host organism for replication.
- Virus Propagation: The process by which the virus multiplies and sends copies of itself to the inner cells.
- Virus Neutralization: The process by which the virus is deactivated (destroyed) by the corresponding antibody.
- Clean cell: A cell which is free from any antigen.
- Infected cell: A cell which has at least one virus present in it.
- Virus Maturity Period: This is the time duration in which the virus is inactive, after which it regularly replicates.
- Virus Propagation Period: A mature virus regularly replicates after a certain number of time units, and this period is defined as the virus propagation period.

7.2 Problem Definition

Given an initial membrane structure with the viruses and antibodies, it is useful to know that when an equilibrium is reached, whether all the cells are *clean* or some cells remain *infected*. From a pharmaceutical perspective, this tells us whether or not the membrane requires any external medicinal supply (or whether it is strong enough to resist the virus). Certain configurations worsen at every

subsequent phases i.e. more viruses survive and antibodies slowly get depleted. The detection of such patterns in early stages would increase the chances of cleaning the cells.

7.3 P System Perspective

The organism which the virus infects is represented as a skin membrane in the P system. The cellular hierarchy of the organism is modelled as the tree structure of the membranes. The virus and antibodies are the objects in the system. The virus and antibody properties (e.g. the type and life) are the symbols of the objects. We use the following rules:

1. The skin membrane has unlimited supply of antibodies of all types.
2. An antibody of type a_i is required to neutralize the virus v_i.
3. For virus propagation:
   - each virus has a maturity period, after which it can reproduce,
   - thereafter, it reproduces regularly after a fixed propagation period,
   - the virus child thus created may be sent to any one of the children membranes in a random manner.
4. A membrane requiring an antibody sends a request to its parent for that particular antibody.

For this problem, a distributed computing approach is given by the following algorithms:

- Leadership Election: Identifying whether a cell is clean or infected is analogous to whether the virus wins or not in an leader election algorithm;
- Synchronization: Communication between membranes to maintain the required balance of antibodies in order to reach a clean state.

7.4 Equilibrium State Determination

Determining whether the virus will survive or not is not a trivial problem. The sufficient condition for the equilibrium state can be written as $M > 2D$, where M is the maturity period of the virus, and D is the maximum distance between a virus and the skin. This is easy to see as it will take exactly $2 \times D$ time for a membrane to receive the antibody after it has requested for it. If the above condition is satisfied, the virus will be destroyed before it reproduces. This condition has to be true for all the membranes. However, the necessary condition is more complicated. Since, some antibodies may be present in earlier stages as well (e.g. one of the membranes other than the skin membrane) the membrane can receive the antibody earlier. Thus, in practice a precise analysis is required. Moreover, each child virus will have its own life-cycle of maturing and reproducing simultaneously, making the mathematical formulation rather complicated.

The following table shows the number n of viruses at different times t. In this example $M = 3$ and the virus propagation period $P = 1$.

Time	Total	New	Mature
0	1	1	0
1	1	0	0
2	1	0	0
3	2	1	1
4	3	1	1
5	4	1	1
6	6	2	2
7	9	3	3
8	13	4	4
9	19	6	6
10	28	9	9

From the table we can observe that $n(t) = n(t-1) + n(t-3)$; generalizing, we get $n(t) = n(t-P) + n(t-M)$. It is possible to express $n(t)$ as a polynomial function of t. The coefficients of the polynomial expression are provided by the the roots of the equation $t^M - t^{M-P} - 1 = 0$. It is difficult to solve this equation manually; a recursive algorithm can easily be implemented at the start of the simulation to decide the state.

7.5 Algorithm: Immune Response

In each round we have

1. An exchange of antibodies:
   (a) every parent sends its antibodies to its children as requested in the previous round;
   (b) every child receives antibodies as sent by parent;
2. a virus propagation:
   (a) increment virus life and check for reproduction;
   (b) send virus to children if required;
   (c) receive virus from parent (if not skin).
3. **Compute leader**: for each round, check if virus dominates or is destroyed.
4. **Synchronization**: send antibody requests to all parents.

7.6 Implementation

For implementing this example, we have used the Message Passing Interface (MPI). MPI is a standard developed to enable portable message passing applications, and though many new parallel languages and environments are introduced, it is still very popular in the field of parallel computing. MPI is a library of functions and macros that can be used in the Fortran, C, C++ and Java programs. MPI programming means that you write your program in C, C++ or Java, and when the time comes for parallel processes to communication or synchronize, you should explicitly call the MPI send or receive function to

help. MPI send function sends a message to the named target process, while in the target process, a correspondent receive function must be set to make the corresponding work. MPI is quite easy to use. You need to master around six commands to write simple programs; they are MPI_Init(), MPI_Comm_rank(), MPI_Comm_size(), MPI_Send(), MPI_Recv() and MPI_Finalize(). To use the MPI system and functions, the header file mpi.h should be included. Different processes are identified with their task ID's; the MPI system assigns each process a unique integer called as its rank (beginning with 0). The rank is used to identify a process and communicate with it. Each process is a member of a communicator; a communicator can be thought of as a group of processes that may exchange messages with each other. By default, every process is a member of a generic communicator environment (it could be interpreted as the skin in a membrane system). Although we can create new communicators, this leads to an unnecessary increase in complexity. The processes can be essentially identical, i.e. there is no inherent master-slave relationship between them. So it is up to us to decide who will be the master and who will be the slaves. A master process can distribute data among the slaves. Once the data is distributed among the slaves, the master must wait for the slaves to send the results and then collect their messages. Packing and decoding is handled by MPI internally. The code for the master as well as the slaves could be in the same executable file. More details can be found in [6,8]. Our implementation is written in C and uses the parallel environment provided by the MPI library.

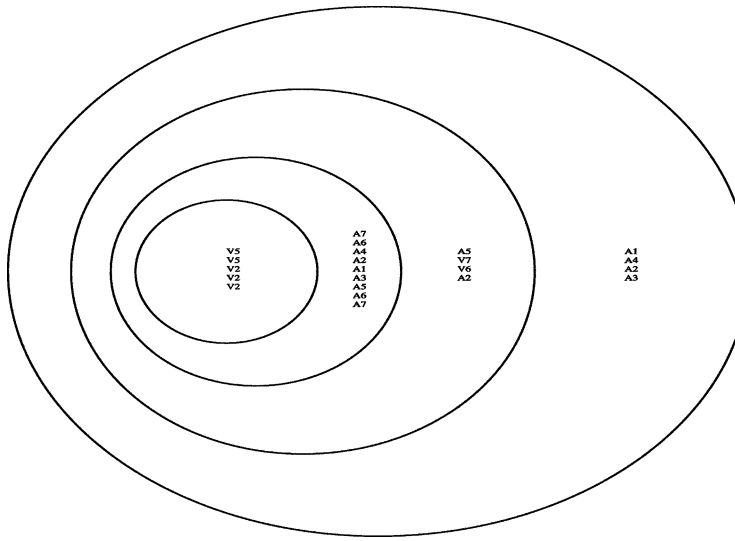


Fig. 1. Example of an output diagram

The implementation steps were:

1. A P system library was written to emulate the functions of a P system. We assume that each membrane is a processor.
2. An MPI program was written for the simulation of the various rounds of the system, implementing the algorithm presented above.
3. An interface was written to create an additional level of abstraction between the library and the MPI program, in order to hide the implementation details of the P system library.
4. An automated graphical output is generated at the end of the simulation making use of XFig and LaTeX. The output at the end of each round is used to encode an XFig diagram, and these are included in a LaTeXpresentation.

References

1. H. Attiya, J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*, McGraw-Hill, 1998.
2. G. Ciobanu. Distributed algorithms over communicating membrane systems, *BioSystems*, Elsevier, to appear.
3. G. Ciobanu, D. Paraschiv. P System Software Simulator, *Fundamenta Informaticae* vol.49 (1-3), 61-66, 2002.
4. G. Ciobanu, D. Dumitriu, D. Huzum, G. Moruz, B. Tanasă. Client-Server P Systems in modeling molecular interaction. In Gh. Păun, G. Rozenberg, A. Salomaa, C. Zandron (Eds.): *Membrane Computing 2002*, Lecture Notes in Computer Science - this volume, Springer, 2002.
5. L. Lamport, R. Shostak, M. Pease. The Byzantine generals problems. *ACM Trans. Program. Lang. Syst.* vol.4(3), 382-401, 1982.
6. P. Pacheco. *Parallel Programming with MPI*, Morgan Kaufmann Publishers, 1997.
7. Gh. Păun. Computing with membranes, *Journal of Computer and System Sciences*, vol.61, 108-143, 2000.
8. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra. *MPI–The Complete Reference vol.1, The MPI Core*, 2nd edition, MIT Press, 1998.