

On P Systems and Distributed Computing

Apostolos SYROPOULOS

Greek Molecular Computing Group

366, 28th October Str.

GR-671 00 Xanthi, GREECE

E-mail:apostolo@obelix.ee.duth.gr

Abstract

This paper gives an affirmative answer to the following intriguing question: “*Is there any relationship between P systems and distributed computing*”? In particular, we introduce a new form of P systems capable of modelling the basic functionality of distributed systems. Also, we introduce a new kind of processor that can be used to implement the functionality of a compartment of a given P system. These processors can be used to form a purely distributed architecture. Combined with previous experience gained from the implementation of functional languages on parallel hardware, we can use this architecture to provide an implementation of functional languages on distributed architectures.

1 Introduction

P systems [5] are a new promising model of computation inspired by Nature. In particular, P systems form an abstraction of the way cells live and function. In addition, they are built around the notion of nested compartments surrounded by porous *membranes* (hence the term *membrane computing*). Recently, it has been demonstrated that P systems can be quite naturally simulated in a distributed computational environment [1], [10]. Thus, it seems that there is a relationship between P systems and distributive computing, which deserves a deeper study. But what exactly is distributed computing?

If a single computer can solve a problem in ten seconds, can ten computers solve the same problem in one second? This, and similar questions, have intrigued computer scientists since the earliest days of electronic computers. Thus, this fascination led to the development of the scientific discipline of distributed computing. Roughly speaking, distributed computing is a computing method that gives an affirmative answer to the question above. More formally, distributed computing is the process of running a single computational task on more than one distinct computer. Sometimes, distributed computing is confused with cluster computing. However, the two computing methods differ in that computers in a distributed computing environment are typically not dedicated to distributed computing, whereas clusters are almost always comprised of dedicated hardware. This makes distributed computing very attractive because it can utilize computational resources that would otherwise be unused, or it can make it possible to have resources for special computational

purposes shared among users. It also means that high volumes of data that would otherwise require the power of supercomputers can be processed at low cost. In order to demonstrate these remarks, we will present a concrete example of a widely known distributed application.

SETI@home (SETI at home) is a distributed computing project for home computers, hosted by the University of California, Berkeley. SETI is an acronym for the *Search for ExtraTerrestrial Intelligence*. SETI@home's purpose is to analyze data incoming from the Arecibo radio telescope, searching for possible evidence of radio transmissions from extraterrestrial intelligence. With nearly 5 million users worldwide, the project is the most successful example of distributed computing to date. It performs three main tests:

- searching for Gaussian rises and falls in transmission power, possibly representing the antenna passing over a radio source,
- searching for pulses possibly representing a narrowband digital-style transmission, and
- searching for triplets, three pulses in a row.

Since 1999, the project has logged over 1 million years of aggregate computing time. On September 26, 2001, SETI@home reached the ZettaFLOP (10^{21} floating point operations) mark—a world record! While the project has not found any conclusive signs of extraterrestrial intelligence, it has identified several candidate spots for further analysis.

The SETI@home distributed computing software, available for all major operating systems, runs either as a screensaver or continuously while a user works, converting otherwise wasted processor power into useful research.

SETI@home, in addition to its altruistic use to aid SETI, is quite useful as a stress testing tool for computer workstations. Since it uses error-correction algorithms to verify the results of the computations, SETI@home is often used to check on the reliability of a computer configuration when overclocking.

The rest of the paper In what follows we present a brief introduction to concepts related to distributed computing. We continue by showing how the basic ingredients of every distributed system can be modelled by means of P systems. Next, we introduce a new computing processor, namely P-processors, that can be used to implement any P system on real hardware and the relationship between P systems and functional programming. We conclude with some thoughts concerning future work and research.

2 Basic Characteristics of a Distributed System

In order to establish a (formal or at least a semi-formal) relationship between the theory of P systems and distributed computing, we need to know what are the basic ingredients of such systems. Note that we are not going to restrict ourselves to distributed operating systems, but distributed systems, in general. Typically, a

distributed system is characterized by its network structure, the way data flow in the system, the way remote processes are coordinated, the file system it uses, etc.¹ However, not all of these characteristics are interesting in order to establish the relationship we are seeking. For example, whether data flow is encrypted or not in a system is not of great importance. After all data are just data. Let us now present the main ingredients that are common to a wide range of distributed systems.

Network Structures A distributed system consists of a number of interconnected nodes. The way these nodes are connected characterizes the *topology* of the network. Typically, nodes can be fully or partially connected. In the first case, all nodes have direct connections with all the nodes that make up the network. In the second case this is not true—a particular node is connected only to a subset of the nodes that make up the system. Let us now review the basic types of networks:

- In a *hierarchical* network, the nodes are organized as a tree. Each node (except the root) has a unique parent, and some (possibly zero) number of children. This organization is very common and the largest hierarchical network is the Internet itself.
- In a *star* network, one of the nodes in the system is connected to all other nodes, while none of the rest of the nodes are connected to any other.
- In a *ring* network, each node is connected to exactly two other nodes, thus resembling a ring (hence the name). The connection can be unidirectional (i.e., data can flow in only one direction) or bidirectional (i.e., data can flow in both directions).
- In a *multiaccess bus* network, there is a single shared link (the bus). All nodes in the system are directly connected to that link, which may be a straight line or a ring.
- A *hybrid* network is one that consists of sub-networks of different types that are connected together.

Programming Tools The Internet has changed everything, and yet there are few tools for writing distributed programs that run over an internet of computers. Nowadays, Java plus its *Remote Method Invocation* (or RMI for short) and related technologies, such as the Simple Object Access Protocol (or SOAP for short), are the tools of choice for distributed computing. On the other hand, some purely distributed operating systems, such as Plan 9 from Bell Labs or Tanenbaum's Amoeba,² have a quite limited hardware compatibility list, thus making essentially impractical the development of distributed applications on such systems. Hopefully, work on new operating systems, such as the E1 operating system (see <http://www.e1os.org/eng/index.html> for details), will remove this serious drawback.

¹For a detailed presentation of these and other less important characteristics, the reader should consult any good book on the theory of operating systems (e.g., see [8, 11]).

²An amoeba is a unicellular organism, and cells are the "driving force" behind P systems!

File “Systems” Typical distributed applications like WinMX are based on the idea of file sharing. This simply means that a large number of computers share computer resources and in particular disk space. Each node specifies a directory from which remote users can download files, while the same area serves as a means to store incoming data. Roughly speaking, all nodes participating in the network have access to a *distributed file system*. Of course this is not a file system in the strict sense of the word. The reason being the fact that disk space on various computers with different operating systems and, thus, different native file systems, is accessible to all nodes participating in the network. Another, more restrictive, way to share disk space is Sun’s *Network File System*, which allows disk space sharing among nodes running some flavor of the Unix operating system.

Coordination Coordination is a major issue in the design and implementation of reliable distributed systems. For example, event ordering and deadlock handling are problems that have to be solved in order to make a distributed system reliable. Up to now many researchers in the field of distributed computing have proposed various solutions to the coordination problem. For instance, Leslie Lamport, a pioneer in the field of distributed computing, has proposed the *Paxos Algorithm* [3] and more recently the *Disk Paxos* [2] algorithm.

Distributed Architectures Currently, there are two basic models to implement distributed applications: the client-server architecture and the peer-to-peer architecture. In a client/server system, there is a central authority that dictates the flow of computation. Clients can connect to the server in order to post or retrieve data. In a peer-to-peer system there is no physical or logical central authority—all nodes act both as servers and clients. Some argue that peer-to-peer computing is a post-client/server paradigm where every client is also a server and vice versa (e.g., think of the way WinMX operates).

3 P Systems and Distributed Computing

In the previous section we reviewed the basic characteristics of distributed systems. Evidently, readers acquainted with P system have identified a number of features that are common to P systems. For example, a hierarchical network resembles the membrane structure of a typical P system, while the maximal parallelism principle is a form of system coordination. Ergo, in order to establish a relationship between P systems and distributed computing we need to examine closely all aspects of distributed computing in order to find their counterparts in the theory of P systems.

3.1 Network Structure

As we have noted earlier, the structure of a hierarchical network can be easily represented as a membrane structure. This argument is valid by noticing that any concrete hierarchical network structure is just a concrete representation of a generalized tree structure. Thus, for any hierarchical network structure, we can build an equivalent membrane structure. Similarly, a star network structure has actually

the restricted form of a tree structure with only two levels, hence all nodes can be considered as compartments that are placed inside a skin membrane. However, in the most general case, a network structure is actually a graph. Therefore, only P systems with a graph membrane structure are suitable to describe the network structure of such systems.

P systems with a membrane structure that is actually a graph have been discussed in [6, Chapter 6]. In particular, Păun describes tissue-like membrane systems with symport/antiport rules and neural-like networks of membranes, which are capable of describing the network structure of a distributed system. In spite of this, these systems are too elaborate for our needs. What we actually need is a “stripped-down” version of these systems. In other words, we need symbol-object membrane systems with a membrane structure that is actually a graph. Obviously, we cannot specify a graph membrane structure with a string of matching square brackets, but graphs can be easily represented using an *adjacency-matrix representation*.³ It is not difficult to see that an adjacency-matrix is actually a relation in $X \times X$, where $X = \{1, \dots, n\}$. Thus, we can use either a relation or an adjacency-matrix to describe a network structure. But this is not enough—we need to extend the “scope” of the evolution rules. In a symbol-object membrane system data can flow only to parent or child membranes. Thus, we have to relax this restriction and allow data to arrive to any compartment that is directly connected to the “departing” compartment. Let us now summarize:

Definition 3.1 *A graph-structured symbol-object membrane system is a set of (possibly nested) compartments that can be specified as follows:*

$$\Pi = (O, g_m, w_1, \dots, w_m, R_1, \dots, R_m, i_0).$$

Here:

- (i) O is a set of objects that denote elements of information;
- (ii) g_m is a relation in $\{1, \dots, m\} \times \{1, \dots, m\}$, describing the network structure of the system;
- (iii) w_i , $1 \leq i \leq m$, are strings representing multisets of objects that are elements of O and denote conglomerations of information;
- (iv) R_i , $1 \leq i \leq m$, are finite sets of **evolution rules** over O ; R_i is associated with compartment i ; an evolution rule is of the form $u \rightarrow v$, where u is a string over O and v is a string over O_{NTAR} , where $O_{\text{NTAR}} = O \times \text{NTAR}$, and $\text{NTAR} = \{\text{here}, \text{out}\} \cup \{\text{to}_j | 1 \leq j \leq m\}$;
- (v) $i_0 \in \{1, 2, \dots, m\}$ is the label of the **output compartment**.

The symbol “to _{j} ” should be used to directly place an object from the host compartment to compartment j . The rule is applicable only if $(i, j) \in g_m$ or $(j, i) \in g_m$. Apart from these differences, these P systems function exactly like their counterparts with a tree membrane structure.

³Given a graph with n vertexes, we build an $n \times n$ matrix with the following recipe: entry (i, j) is 1 if there is an edge from vertex i to vertex j , otherwise the entry is 0. Especially for undirected graphs, given that there is an edge between i and j , both (i, j) and (j, i) are 1.

3.2 P Systems and Distributed Programming

The π -calculus [4] is a well-known model of concurrency that has been shown to have at least the same expressive power as the λ -calculus.⁴ Practically this means that one can encode at least any general recursive function in the π -calculus. Thus, if P systems are to be used as a foundation for distributed computing, one should be able to encode any general recursive function as a P system. In general, it is a fact that P systems have the computational power of Turing machines. In other words, any Turing machine can be simulated by a corresponding P system and vice versa. However, until recently it was not clear how one can encode a general recursive function as a P system. Fortunately, quite recently, two different encodings of general recursive functions as P systems [9, 7] were proposed. Let us now briefly review the consequences of this new development in the field of membrane computing.

It is known for a long time that functional programming languages can be easily implemented on parallel hardware [13]. Now, assume that F is a functional programming language that has been implemented on parallel hardware. In addition, assume that one implements a purely sequential algorithm A in F . The compiled version of A will be optimized to fully utilize the underlying architecture. Practically, this means that one writes a (purely) sequential program in F and after compilation he/she gets a highly sophisticated parallel program. By following a similar line of thought, we can expect the design and implementation of compilers capable of generating machine code for some distributive architecture based on P systems. The net benefit of this endeavor will be that computer programmers will not have to think about the peculiarities of a particular distributive architecture. Currently, one has to tackle a number of issues when designing programs that are supposed to be executed on distributed architectures. For example, one must choose a particular data dispatch method to ensure that data will be exchanged safely.

3.3 On File Systems

A file system is a concrete mechanism that allows people to store data on a storage medium in an effective way. As such it is a property of the underlying operating system. Since we are not proposing a concrete implementation of a distributed system, but rather a conceptual framework, we believe it makes no sense to discuss this issue.

4 Programming with P Systems

The simulator described in [10] relies heavily on our ability to make use of a network of computers. If we assume that each computer is just a computer processor, we end up with a network of computer processors. Now, if each processor can handle the primitive (computational) actions that take place in a P system, then it will be possible to simulate the behavior of any P system, provided there are enough resources. For example, this means that there is enough memory and the bandwidth

⁴Wegner and Eberbach claim that the π -calculus is actually more powerful than the λ -calculus [12].

is fast enough to accomodate the simulator. Ergo, the first step towards the design of a P-computer is to specify the functionality of these processors and then to find a suitable computing method that can be utilized to express “ordinary” computations in terms of these new processors.

4.1 A New Kind of Computing Processor

The design of a new kind of processor is a two stage operation. First, we need to define the underlying intruction set and then to actually implement it using electronic circuits. It is not our intention to describe a concrete design for the new proposed hardware, but it is definetely our intention to (informally) specify the instruction set of the proposed hardware. After all, it is possible to “implement” a new processor by means of virtual machines. Indeed, this is a common practice in computer science (think of the Java virtual machine or the UCSD p-Code machine). Before we proceed with the specification of the new processor, which we call *P-processor*, we have to make clear that the task of each processor will be to implement the functionality of a compartment of a particular P system.

Each P-processor must be able to communicate with other P-processors. This means that we need instructions that will allow a particular processor to send and receive data. Naturally, when we send data, we need to specify to which processor the data will go. Since the various P-processors will form a network, it is obvious that each of them will be able to receive data from any other processor. However, before a processor will send data to another processor, the former must be sure that the later is available. These remarks suggest the introduction of the following instructions:

send d, i	Send all “ d ’s” to processor “ i ”
isalive i	Check whether P-processor i is alive

Obviously, each P-processor has each own memory and a command of the form **send** d, i should be used to dispatch all elements d to the processor i . The data can be stored in dynamic hash tables where the name of each object d is a key and the value of the entry $D\{d\}$ will denote the number of occurrences of the element d .

Each P system consists of a multiset of data and a set of production rules. Thus, we need to be able to specify the initial data of a compartment and the associated rules. This leads to the introduction of the following instructions:

intro o, n, i	Introduce to the i th compartment n copies of o
addrule R, i	Associate rule R with processor i
delrule R, i	Disassociate rule R from processor i

Note that we need to be able not only to associate a rule with a P-processor, but also to disassociate from it. Since it is not really useful to fully specify the rule itself, R will be actually a memory location that will point to a block of instructions. Naturally, the instructions that make up the block can be any of the instructions presented so far plus any of the following:

replace o, o', i	Replace each occurrence of o in compartment i with o'
delete o, i	Delete each occurrence of o in compartment i
ifempty o, i	If the compartment i is empty, then place a copy of o at it
halt n, i	Halt processor i after n cycles
ihalt i	Immediately halt processor i
joinsys i	Processor i joins the system
exec	Start execution
noop	A do-nothing instruction

Although a **replace** instruction is not absolutely necessary as its functionality can be described in terms of the **delete** and **intro** instructions, still we believe it is useful mainly for reasons of clarity. The instruction **ihalt** corresponds to dissolution of a compartment, while the instruction **joinsys** corresponds to the introduction of a new compartment. Clearly, when a processor joins a system, we need to specify its data and evolution rules as well as its position in the overall graph structure of the system. However, we have not figured out yet how this can be done.

Another problem that we have not addressed at all is how one can actually enumerate the various P-processors. Clearly, this is crucial in order to make our system reliable. However, we believe this is a really technical issue that can be handled by the introduction of additional instructions (e.g., such instructions would allow one to specify the network structure, etc.). As far it regards the implementation of the maximal parallelism requirement, we can safely rely on the techniques developed for the simulator described in [10].

4.2 P Systems and Functional Programming

In the previous subsection we described the instruction set of a new kind of processor that can be used to implement the functionality of a compartment of a P system. Also, it is known that general recursive functions can be computed by P systems. In addition, we have at our disposal at least two different ways to encode any general recursive function as a P system. Furthermore, it is known that all general recursive functions can be encoded in Church's λ -calculus, which forms the basis for the so-called functional programming. The remarks lead quite naturally, to the conclusion that we can use a primitive functional programming language as a high level programming notation. Thus, we do not actually need a new programming method, instead, we can safely rely on the experience gained from the implementation of functional programming languages on parallel architectures.

5 Conclusions

We have demonstrated that P systems are indeed directly related to distributed computing. Thus, we have given an affirmative answer to the question "*Is there any relationship between P systems and distributed computing?*" In addition, we described a new kind of processor capable of handling the primitive programming actions that take place in any compartment of any *arbitrary* P system. However, the design of this processor is far from being complete and we expect much work

to be done towards the formal specification of the processors and, consequently, the implementation of the specification.

References

- [1] G. Ciobanu, G. Wenyuan, P Systems Running on a Cluster of Computers, Lecture Notes in Computer Science 2933, Springer, Berlin, 2004, 123–139.
- [2] E. Gafni, L. Lamport, Disk Paxos, Distributed Computing, 16, 1 (2003), 1–20.
- [3] L. Lamport, The Part-Time Parliament, ACM Transactions on Computer Systems, 16, 2 (1998), 133–169.
- [4] R. Milner, Communicating and Mobile Systems: The π -calculus, Cambridge University Press, 1999.
- [5] Gh. Păun, Computing with Membranes, Journal of Computer and System Sciences, 61, 1 (2000), 108–143.
- [6] Gh. Păun, Membrane Computing: An Introduction, Springer, Berlin, 2002.
- [7] A. Romero-Jiménez, M.J.Pérez-Jiménez, Computing Partial Recursive Functions by Transition P Systems, Lecture Notes in Computer Science, 2933, Springer, Berlin, 2004, 320–340.
- [8] A. Silberschatz, G. Gagne, P.B. Galvin, Operating System Concepts, John Wiley and Sons, New York, 2002.
- [9] A. Syropoulos, S. Doumanis, K.T. Sotiriades, Computing with P Systems, manuscript, 2004.
- [10] A. Syropoulos, E.G. Mamatas, P.C. Allilomes, K.T. Sotiriades, A Distributed Simulation of Transition P Systems, Lecture Notes in Computer Science 2933, Springer, Berlin, 2004, 357–368.
- [11] A.S. Tanenbaum, M. van Steen, Distributed Systems: Principles and Paradigms, Prentice Hall, 2002.
- [12] P. Wegner, E. Eberbach, New Models of Computation, The Computer Journal, 47, 1 (2004), 4–9.
- [13] R. Wilhelm, M. Alt, F. Martin, M. Raber, Parallel Implementation of Functional Languages, *5th LOMAPS Workshop, Analysis and Verification of Multiple-Agent Languages*, M. Dam, ed., Lecture Notes in Computer Science 1192, Springer, Berlin, 1997.