Design and Evaluation of Parallel String Matching Algorithms for Network Intrusion Detection Systems

Tyrone Tai-On Kwok and Yu-Kwong Kwok*

Department of Electrical and Electronic Engineering
The University of Hong Kong, Pokfulam Road, Hong Kong
ykwok@hku.hk

Abstract. Network security is very important for Internet-connected hosts because of the widespread of worms, viruses, DoS attacks, etc. As a result, a network intrusion detection system (NIDS) is typically needed to detect network attacks by packet inspection. For an NIDS system, string matching is the computation-intensive task and hence the performance bottleneck, since every byte of the payload of packets must be checked against numerous predefined signature strings, which may occur arbitrarily in the payload. In this paper, we present the design and evaluation of parallel string matching algorithms targeting hardware implementation on FPGAs and software implementation on multi-core processors. Experimental results show that, on a multi-processor system, the multi-threaded implementation of the proposed parallel string matching algorithm can reduce string matching time by more than 40%.

1 Introduction

Network security is gaining more and more concern for Internet-connected hosts because of the widespread of worms, viruses, DoS attacks, etc. As illustrated in Figure 1, like hardware firewall systems, a network intrusion detection system (NIDS) can detect/prevent network attacks by packet inspection/filtering. However, unlike conventional firewall systems, which perform only protocol analysis by inspecting the header of packets, an NIDS also inspects the payload of packets. By adopting this kind of content-based security checking, an NIDS can significantly reduce much more security threats that cannot be detected by conventional firewall systems, such as buffer overflow attacks.

Snort [1] is an open source software NIDS that is widely adopted by the research community as a prototyping platform to investigate different intrusion detection techniques. As a lightweight and yet efficient NIDS, Snort is also used in small networks to detect various network attacks. To increase the security level of Internet-connected hosts, we can even install Snort on each host. Snort relies on a number of predefined rules to filter possible attack packets. For example, one of the signs of the Nimda worm attack is the occurrence of string "readme.eml"

* Corresponding author.

K. Li et al. (Eds.): NPC 2007, LNCS 4672, pp. 344–353, 2007.

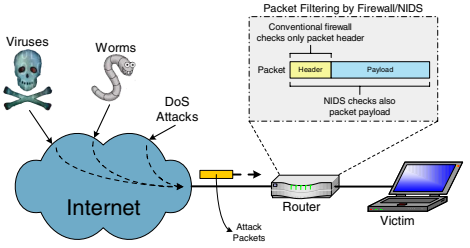


Fig. 1. Network intrusion detection scenario

in a packet. Then, a rule for the NIDS system might be to filter all packets having this particular string in the payload part. Figure 2 shows the actual Snort rule which filters incoming attack packets of the Nimda worm. The signature string "window.open|28 22|readme.eml|22|" is the string that Snort needs to check against every incoming packet (i.e., by performing a string matching on the signature string) so as to eliminate the Nimda worm attack. On the other hand, it should be emphasized that the signature string might appear arbitrarily in the payload of an attack packet, meaning that every byte of the payload must be checked against the signature string.

```
alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any
(msg:"WEB-CLIENT readme.eml autoload attempt"; flow:to_client,established;
 content:"window.open|28 22|readme.eml|22|"; nocase;
 reference:url,www.cert.org/advisories/CA-2001-26.html;
 classtype:attempted-user; sid:1290; rev:10;)
```

Fig. 2. The Snort rule for filtering incoming attack packets of the Nimda worm

At the time of writing, there are 8868 Snort rules like the one shown in Figure 2. Then, it follows that, for each packet, thousands of signature strings need to be checked against the payload of the packet. Thus, the string matching process is a computation-intensive task in Snort. In fact, on profiling the performance of Snort 1.6.3, it is found that 31% of the Snort processing is due to string matching [2], which is the bottleneck that new string matching algorithms should be developed so as to further increase the efficiency of Snort.

To increase the performance of Snort, various string matching schemes have been proposed for incorporation in Snort, for example, the classic Boyer-Moore [3] and Aho-Corasick [4] algorithms. Based on the ideas of the two algorithms, Coit *et al.* proposed a string matching algorithm that can improve the performance of Snort by 1.02–3.32 times when compared with the standard Boyer-Moore implementation [5]. The Wu-Mander multi-pattern matching algorithm [6] and the E^2xB algorithm [7] are another algorithms that have been implemented in Snort.

All the approaches mentioned above are software-based, and it is hardly that they can achieve a packet inspection rate on the order of Gb/s (in practice, a

rate of about 750Mb/s is typically achieved [8]). In order to support fast packet inspection (e.g., at the rate of 10Gb/s), researchers have investigated different hardware-based approaches. Specifically, a string matching engine which implements the Snort rules for packet inspection is realized in hardware. In this regards, FPGAs (Field Programmable Gate Arrays) have become more and more popular in the realization of this kind of high-speed NIDS systems, because we can implement massively parallel circuits in FPGAs (FPGAs are computation-efficient), and more importantly, FPGAs can be dynamically reconfigured to incorporate new rules on-demand (FPGAs are flexible).

Knuth-Morris-Pratt algorithm (KMP) is an efficient string matching algorithm [9], and has been implemented in FPGAs by Baker *et al.* [10]. Other FPGA implementations of string matching algorithms are mostly based on hashing [11] or brute-force (i.e., using discrete comparators or NFAs/DFAs) [12] implementations. However, traditional string matching algorithms such as KMP and those based on hashing are designed based on the von Neumann load/store processor architecture, which cannot utilize the highly spatial parallelism of FPGAs. To this end of the problem, we propose that *cellular automata (CA)* [13,14], a highly parallel computational model as proposed by von Neumann as the *universal machine* [13], is desirable to tackle the string matching problem of NIDS systems.

Despite that current software-based implementation of Snort can hardly achieve a packet inspection rate on the order of Gb/s, we believe that the shortcoming is due to string matching algorithms adopted, not because of the performance of current processors. In fact, the power of current multi-core processors has not been fully harnessed, since the current implementation of string matching algorithms in Snort uses only one processor (i.e., single-threaded). Thus, we propose to design a parallel string matching algorithm with multi-threaded implementation. As multi-threaded programs can fully utilize a processor, we believe that such a multi-threaded implementation in Snort can achieve a packet inspection rate on the order of Gb/s.

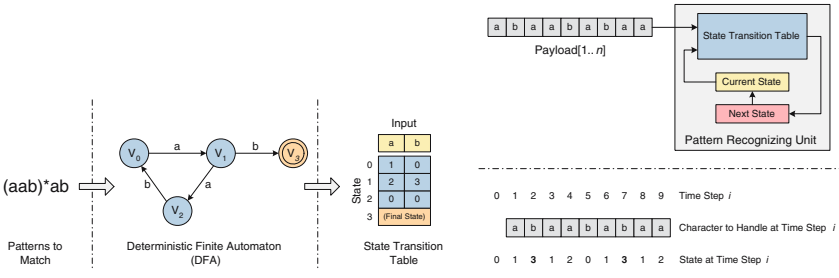
The rest of the paper is organized as follows. In the next section, we present some preliminaries to help understand our proposed algorithms. In Section 3, we present the design and analysis of our proposed parallel multi-pattern string matching algorithms. Experimental evaluation is presented in Section 4. Finally, we conclude in Section 5.

2 Preliminaries

2.1 Multi-pattern String Matching Based on DFA

To recognize a set of string patterns, we can combine the patterns and form a string matching automaton, called deterministic finite automaton (DFA). Figure 3 shows the detailed process. As can be seen, the process is divided into two parts, construction of state transition table and pattern recognizing.

For simplicity, in Figure 3(a) we only illustrate with one pattern $((aab)^*ab)$ [1]. After a state transition table is constructed, as Figure 3(b) illustrates, the pattern recognizing unit (e.g., a processor) can use the table to recognize patterns in a text string (e.g., the payload of a packet). Specifically, each time a character (i.e., a byte) of the text string is taken, which is then used together with the current state of the pattern recognizing unit to look up the next state. If the next state is a final state, then we can know that a pattern is matched, and the pattern recognizing unit can stop or continue to find out other patterns in the rest of the text string.



(a) construction of state transition table (b) pattern recognizing

Fig. 3. The process of multi-pattern string matching based on DFA

2.2 One-Dimensional Cellular Automata (CA)

A one-dimensional cellular automaton consists of a linear array of identical cells, which can be regarded as some lightweight processing elements. Each cell can be in one of a finite number k of states. The state of cell i at time t is $s_t^i \in \Sigma = \{0, 1, ..., k-1\}$. As illustrated in Figure 4 [14], at each time step, all the cells update their state simultaneously according to a local update rule ϕ. This update rule takes as input the local neighbourhood configuration η^i of a cell, which consists of the states of the cell i itself and its $2r$ nearest neighbours, i.e., $\eta^i = (s^{i-r}, ..., s^i, ..., s^{i+r})$. Specifically, we can use the update rule ϕ to obtain the new state of cell i as follows: $s_{t+1}^i = \phi(\eta_t^i)$.

3 Design of Parallel String Matching Algorithms

3.1 Parallel Multi-pattern String Matching Based on CA

To recognize signature strings in the payload of a packet, each character of the payload is handled by a cell of a one-dimensional cellular automaton, and all the cells run in parallel and interactively, as illustrated in Figure 5(a). Specifically,

[1] The signature strings to be checked against by an NIDS are typically specified in a regular expression format, for example, $(aab)^*ab$. Then the NIDS alerts the user when it finds strings like "*ab*", "*aabab*" or "*aabaabaabab*" in a packet.

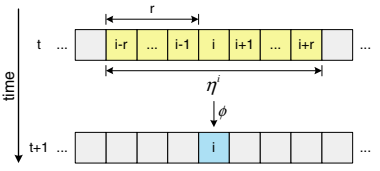
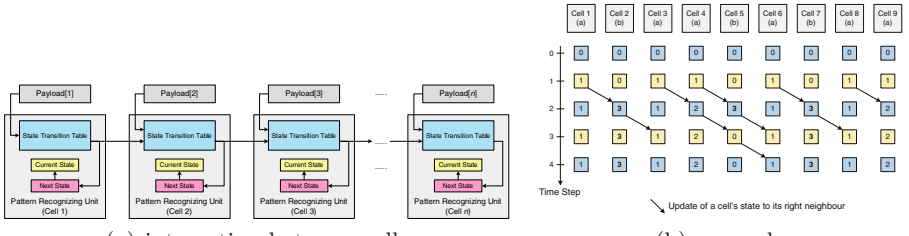


Fig. 4. The update process for cell i in a one-dimensional cellular automaton [14]

each cell (except the leftmost cell) gets the state of its left neighbour and uses this state information together with the character it is handling to look up the state transition table. After the table lookup, the state of the cell is updated and the new state is sent to the right neighbour of the cell. Algorithm 1 describes the update procedure for each cell i of the proposed string matching algorithm, Parallel Multi-Pattern Matching Based on Cellular Automata (CAMP). On the other hand, suppose that the payload of a packet is "*abaababaa*", Figure 5(b) illustrates the time evolution of the cellular automaton for recognizing strings belonging to the pattern $(aab)^*ab$.



(a) interaction between cells (b) example

Fig. 5. The working principle of the proposed multi-pattern string matching algorithm based on cellular automata

3.2 Multi-threaded Design of CAMP (MT-CAMP)

Handling each byte of the payload of a packet by a cell is the ideal case for CAMP, which can achieve the most efficiency. However, since the payload of a packet can have a length of around 1460 bytes, it is practically infeasible to create 1460 processing elements simultaneously on a processor-based system. To mitigate this problem, we develop a multi-threaded version of CAMP (MT-CAMP) for implementation on a multi-core-processor-based system. Algorithm 2 describes the details of the multi-threaded version of CAMP. Specifically, given that the length of payload is n and we want to create $threadNumber$ threads, then each thread implements $\frac{n}{threadNumber}$ cells (i.e., handles $\frac{n}{threadNumber}$ bytes of the payload). On the other hand, it should be noted that the $createThread()$ function in Algorithm 2 does not always create new threads for different packets. To

reduce thread creation overhead, threads are reused. In actual implementation of MT-CAMP, we found that the performance is not satisfactory due to significant communication overhead between threads. We will discuss this issue in more detail in Section 4.

3.3 Parallel Multi-pattern String Matching with No Communication

The poor performance of MT-CAMP on a multi-processor system motivated us to develop a communication-less string matching algorithm, called Parallel Multi-Pattern Matching with Overlapping Region (ROMP). Figure 6 illustrates the idea of multi-threaded design of ROMP (MT-ROMP). Specifically, if there are $threadNumber$ threads created, then the payload of a packet is divided into $threadNumber$ regions, and each thread uses the DFA approach as mentioned in Section 2 to carry out string matching in its region. However, since a string belonging to a particular pattern can span across different regions of the payload, a thread will also perform string matching in its right neighbour's region. Hence, there is an overlapping region where two threads will perform string matching on it. As we will discuss in subsequent section, the length of this overlapping region is small when compared to the length of a payload. Algorithm 3 describes the string matching procedure in each thread of MT-ROMP.

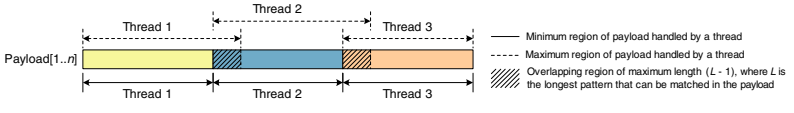


Fig. 6. Payload division in MT-ROMP

3.4 Performance Analysis and Comparison

Suppose that the longest pattern that can be matched in a payload is of length L, then the time complexity of CAMP is $O(L)$. The reason is that L cells are involved in order to recognize the pattern. Referring back to Figure 5(b), if the evolution process can stop at time step 2, then the longest pattern is "ab", meaning that $L = 2$. Similarly, if the evolution process has to stop at time step 4, then the longest pattern is "aabab" and $L = 5$. With a study of the Snort rules, the value of L is typically in the range $1 < L < 20$.

As for MT-CAMP, the time complexity is $O(\frac{n}{threadNumber} \cdot L)$, since each thread needs to implement $\frac{n}{threadNumber}$ cells and the cells within each thread are evolved one by one. The time complexity of MT-ROMP is $O(\frac{n}{threadNumber} + (L-1)) = O(\frac{n}{threadNumber} + L)$, as the longest pattern string can span across two regions of the payload. Table 1 shows the time complexity comparison of CAMP, MT-CAMP, and MT-ROMP with other existing algorithms.

Algorithm 1. CAMP—Update procedure for each cell i.

```
cellUpdate(payload[i], transitionTable)
 1: currentState ← stateLookup(payload[i], transitionTable) /* current state */
 2: newState ← NULL /* new state of the cell */
 3: leftNeighbourState ← NULL /* state of the cell's left neighbour */
 4: while (TRUE) do
 5:    leftNeighbourState ← getLeftNeighbourState()
 6:    newState ← stateLookup(payload[i], leftNeighbourState, transitionTable)
 7:    if (newState ≠ NULL and newState ≠ currentState) then
 8:       currentState ← newState
 9:       if (isFinalState(currentState, transitionTable)) then
10:          report("Matched at payload position i!")
11:       end if
12:    end if
13: end while
```

Algorithm 2. MT-CAMP—Creation of threads and update procedure of cells in each thread.

```
MTcreateThreads(threadNumber, payload[1..n], transitionTable)
 1: cellsPerThread ← n/threadNumber; head ← 1; tail ← 1
 2: while (head < n) do
 3:    tail ← head − 1 + ((head + cellsPerThread − 1 ≤ n)?cellsPerThread : (n − head))
 4:    createThread(MTcellsUpdate(payload[head..tail], transitionTable))
 5:    head ← head + cellsPerThread
 6: end while
MTcellsUpdate(payload[j..k], transitionTable)
 1: m ← k − j + 1 /* number of cells per thread */
 2: currentState[1..m] ← NULL /* current state of cells */
 3: newState[1..m] ← NULL /* new state of cells */
 4: for i = 1 to m do
 5:    currentState[i] ← stateLookup(payload[j + i − 1], transitionTable) /* initial state */
 6: end for
 7: while (TRUE) do
 8:    for i = 1 to m do
 9:       leftNeighbourState ← getLeftNeighbourState(i, currentState[1..m])
10:       newState[i] ← stateLookup(payload[j + i − 1], leftNeighbourState, transitionTable)
11:       if (newState[i] ≠ NULL and newState[i] ≠ currentState[i]) then
12:          currentState[i] ← newState[i]
13:          if (isFinalState(currentState[i], transitionTable)) then
14:             report("Matched at payload position (j + i − 1)!")
15:          end if
16:       end if
17:    end for
18: end while
```

Algorithm 3. String matching procedure in each thread of MT-ROMP.

```
MT-ROMP(payload[j..k], n, transitionTable)
 1: currentState ← stateLookup(payload[j], transitionTable) /* current state */
 2: newState ← NULL /* new state of the cell */
 3: for (i = j + 1; i <= n; i + +) do
 4:    if (i > k and currentState == NULL) then
 5:       break /* escape from the overlapping region */
 6:    end if
 7:    newState ← stateLookup(payload[i], currentState, transitionTable)
 8:    if (newState ≠ currentState) then
 9:       currentState ← newState
10:       if (isFinalState(currentState, transitionTable)) then
11:          report("Matched at payload position i!")
12:       end if
13:    end if
14: end for
```

Table 1. Time complexity comparison

Algorithm	Complexity
CAMP	$O(L)$
MT-CAMP	$O(\frac{n}{threadNumber} \cdot L)$
MT-ROMP	$O(\frac{n}{threadNumber} + L)$
Boyer-Moore [3]	$O(n)$
Aho-Corasick [4]	$O(n)$
KMP [9]	$O(n)$

4 Experimental Evaluation

To implement the ideal design of CAMP (i.e., each byte of the payload is handled by an individual processing element), a viable approach is to use FPGAs, since we can implement massively parallel circuits on FPGAs. More importantly, the communication overhead between cells will be insignificant since the communication is deterministic and at wire speed. On the contrary, the communication between threads is not deterministic, which accounts for the significant communication overhead. Currently, we have not implemented CAMP on FPGAs yet. In this section, we would like to evaluate the performance of MT-CAMP and MT-ROMP on a multi-processor system (particularly, a multi-core system) using multi-threaded implementation. Specifically, our focus is to study the scalability of the proposed parallel string matching algorithms, while maintaining a fast string matching rate.

Since the string patterns that we intend to match are the signature strings in Snort, the first step of evaluation is to convert the signature strings into a state transition table. This process can be carried out by an open source tool called JFlex [15], which is a lexical analyzer generator (or scanner generator). For performance evaluation, we do not use all the signature strings in Snort. In fact, we randomly choose 400 signature strings.

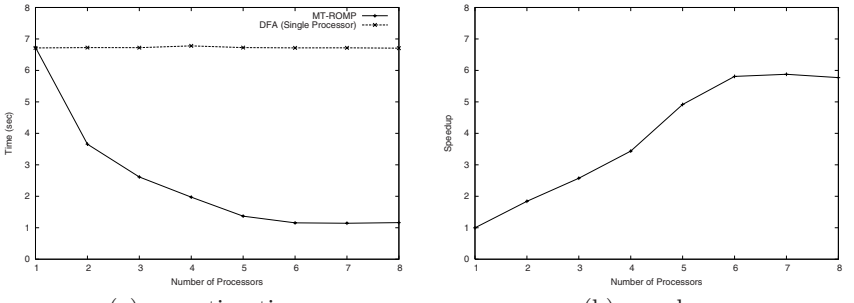
To evaluate the performance of our proposed parallel string matching algorithms, synthetic packets of payload of 1460 bytes are generated. Specifically, for each of the 400 string patterns chosen, we randomly put the string pattern in the payload, and the rest of the payload is filled with random bytes. Using this way, we generate 10 packets for each string pattern and the positions of the string pattern are different. Effectively, 4000 different packets are generated. For our experiments, these 4000 packets are duplicated 100 times to form a packet stream of 400K packets and the packets are stored in the main memory of the system under test.

As a comparison algorithm, the DFA string matching approach as mentioned in Section 2 is chosen. Since the DFA approach shares the same design mechanism as the Aho-Corasick algorithm [4], which is well-known to be a fast multi-pattern string matching algorithm, its speed is of the same grade as the Aho-Corasick algorithm.

First of all, we evaluated the performance MT-CAMP. Using a single thread, we found that MT-CAMP is about 16 times slower than the DFA approach. The deficiency of MT-CAMP comes from the fact that there is excessive communication

overhead between cells. When using more threads, we also found that there is significant communication overhead in using thread-specific synchronization functions such as mutexes.

The poor performance of MT-CAMP motivated us to develop a communication-less parallel string matching algorithm for a multi-processor system, namely MT-ROMP. When inspecting the 400K packet stream under a Pentium D 3.4GHz system with Ubuntu Linux 6.10 installed, the DFA approach uses 3.40s, while MT-ROMP uses 4.08s and 2.06s [2] when one processor core and two processor cores are utilized, respectively. Hence, there is 40% reduction in string matching time when the two processor cores of the dual-core microprocessor are utilized. Since we were not able to obtain a quad-core system to study the scalability of MT-ROMP, we have resolved to evaluate its performance under an SMP machine with eight processors (Sun4u/Sparc with SunOS 5.9 installed). The results are shown in Figure 7. As can be seen from Figure 7(a), there is about 45% reduction in string matching time when two processors are used. This result is quite matched with that obtained from the dual-core machine. On the other hand, Figure 7(b) shows that the speedup is quite linear. However, the speedup is saturated when more than six processors are used. This is due to frequent context switching of the system.



(a) execution time (b) speedup

Fig. 7. Performance comparison of MT-ROMP and the DFA approach

The lesson learnt from our experience of designing a parallel string matching algorithm on a multi-processor system is that it is extremely important to reduce communication between threads. As discussed in detail in [16], the frequent use of mutexes, semaphores, monitors, etc., to achieve synchronization between threads has adverse effect on the performance of multi-threaded programs. With the emergence of multi-core microprocessors, if we expect parallel programming

[2] This corresponds to a string matching rate of about 2.1Gb/s. However, the actual value for real-life packet inspection should be lower than 2.1Gb/s, as the 400K packets are stored in the main memory of the system. We have not taken into consideration the preprocessing time of packets once they are received from the network interface.

to become mainstream, Lee suggested in [16] that we should construct parallel programming models that are much more predictable and understandable than threads.

5 Conclusions and Future Work

In this paper, we have presented the design and evaluation of several parallel multi-pattern string matching algorithms for an NIDS system. Our future work is to implement CAMP on FPGAs to investigate how fast string matching rate it can achieve in hardware implementation. As for MT-ROMP, we would like to integrate it with Snort, and then carry out more detailed performance evaluation with real-life network traffic.

References

1. Snort Official Web Site: `http://www.snort.org/` (2007)
2. Fisk, M., Varghese, G.: Fast content-based packet handling for intrusion detection. Technical Report, CS2001-0670, University of California, San Diego (2001)
3. Boyer, R., Moore, J.: A fast string match algorithm. Communications of the ACM 20, 762–772 (1977)
4. Aho, A., Corasick, M.: Fast pattern matching: An aid to bibliographic search. Communications of the ACM 18, 333–340 (1975)
5. Coit, C.J., Staniford, S., McAlerney, J.: Towards Faster String Matching for Intrusion Detection or Exceeding the Speed of Snort. In: DARPA Information Survivability Conference and Exposition II, vol. 1, pp. 367–373 (2001)
6. Wu, S., Mander, U.: A fast algorithm for multi-pattern searching. Technical Report, TR-94-17, University of Arisona (1994)
7. Anagnostakis, K., Markatos, E., Antonatos, S., Polychronakis, M.: E^2xB: A Domain-Specific String Matching Algorithm for Intrusion Detection. In: The 18th IFIP International Information Security Conference (2003)
8. Norton, M.: Optimizing pattern matching for intrusion detection. White Paper, Sourcefire Inc. (2004)
9. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT Press, Cambridge (2002)
10. Baker, Z.K., Prasanna, V.K.: Time and Area Efficient Pattern Matching on FPGAs. In: FPGA 2004, pp. 223–232 (2004)
11. Dharmapurikar, S., Krishnamurthy, P., Sproull, T., Lockwood, J.: Deep Packet Inspection using Parallel Bloom Filters. In: Symposium on High Performance Interconnects (HotI), pp. 44–51 (2003)
12. Sourdis, I., Pnevmatikatos, D.: Fast, Large-Scale String Match for a 10Gbps FPGA-based Network Intrusion Detection System. In: Cheung, P.Y.K., Constantinides, G.A. (eds.) FPL 2003. LNCS, vol. 2778, pp. 880–889. Springer, Heidelberg (2003)
13. Neumann, J.V.: Theory of Self-Reproducing Automata. University of Illinois Press (1966)
14. Hordijk, W.: Dynamics, Emergent Computation, and Evolution in Cellular Automata. PhD thesis, The University of New Mexico (1999)
15. JFlex—The Fast Scanner Generator for Java (2007), `http://www.jflex.de/`
16. Lee, E.A.: The problem with threads. IEEE Computer Magazine 39, 33–42 (2006)