Massively Parallel Search for Transition-Tables of Polyautomata

Peter Sanders
Informatik für Ingenieure und Naturwissenschaftler
Universität Karlsruhe, D-76128 Karlsruhe (Germany)
E-mail: `sanders@ira.uka.de`

Abstract

One of the fundamental tasks in automata theory is to look for transition-tables that implement a given specification. In principle, most of this task can be performed by a computer. But a combinatorial explosion in the number of possible transition-tables quickly renders brute force search impractical. This paper demonstrates two approaches to extend the frontier of tractable problem sizes. Firstly, an efficient heuristic technique is used which dramatically prunes the search space without giving up completeness. Secondly, a massively parallel implementation is described which achieves near linear speedup on as many as 16384 Processors. These techniques yield some new results regarding two open problems involving cellular automata and trellis automata.

1 Introduction

A recurring theme in computer science is the wish to automatically infer programs from a problem specification. In general, program inference is undecidable and even in cases where it is decidable its computational complexity makes it prohibitive for most applications. Nevertheless, the idea is so attractive that it appears to be worth looking for niches of applicability.

The programming paradigm investigated in this paper are cellular automata or, more generally, polyautomata. These collections of multiple interacting finite automata attract a lot of interest recently because they are simple and realistic models for parallel computation and many other complex systems. Since programs for polyautomata are simply tables and have a very clear semantics, it is possible to design efficient and reliable search algorithms (Section 2). These ideas date back to [1] but have almost been forgotten since then. Two examples concerning cellular automata and trellis automata for which search techniques have proved useful are presented in Section 3.

The complexity of transition-table search quickly exceeds the computational abilities of a sequential computer. This makes it very attractive to use parallelism to boost performance. Section 4 describes an implementation strategy suitable for massively parallel computers. An implementation on a MasPar MP-1 SIMD computer with 16384 processing elements (PEs) achieves near linear speedup. This result makes the parallel implementation interesting on its own right because so far there have been few successful implementations of parallel tree search on machines of this scale (e.g. [2]), most of which appear to use toy examples like the 15-puzzle.

2 Search Strategies

The idea of searching for a transition-table realizing a given specification can only be carried out if the specification has a form that is tractable and efficient to implement. Therefore, this paper restricts the specifications to use the following types of information:

- A finite set of inputs $\mathcal{C}_0$ for which a given behavior is to be achieved. If the original problem requires an infinite number of potential inputs, the search algorithm can only find candidate solutions which have to be verified separately. However, the search algorithm may be able to refute the existence of a solution.
- A specification of the desired behavior of the automaton for a given input. This information may involve final results, maximal and minimal number of steps and constraints on intermediate results.
- A maximal number of different symbols[1] to be used.
- A priori knowledge about the transition-function. For example, some entries of the transition-table may be known in advance.

An Abstract Search Algorithm

A very simple search strategy is *naive generate-and-test*: Enumerate all transition-tables for a given number of symbols and test all possible inputs for each table. A simple back-of-the-envelope calculation shows that this is prohibitively expensive for all but the most simple cases.

A better algorithm is based on the idea to overlap enumeration of solution candidates and testing of their correctness. The abstract Algorithm 1 takes one input at a time and tries to simulate the automaton's behavior for this input. A cellular automaton for example is simulated by updating one cell at a time, e.g. in left to right order. Whenever the program encounters a transition-table entry that is not yet defined, it has to choose a value for this entry. When the simulation runs into a situation that contradicts the problem specification, one of the previous decisions has to be reverted (by choosing another entry and restarting the simulation from this choicepoint). Eventually, a solution will be found (when all inputs have been processed) or the search runs out of untried choices proving that no solution exists.

[1]In general, automata of different kinds may involve different kinds of alphabets for representing input, output, internal state, etc. For the sake of simplicity, this paper uses the term "symbol" to subsume all these entities.

```
set all unknown entries of σ to ⊥
FOR each C ∈ 𝒞₀ DO
    WHILE C has not been processed completely DO
        IF the next step in simulating the behavior for C is defined THEN
            (*SIMULATION*)
            execute it
            IF the state of the automaton has become incorrect THEN
                backtrack
        ELSE
            (*CHOICE POINT for backtracking*)
            choose a result for the undefined entry of σ
RETURN σ
```

Figure 1: Abstract search algorithm.

Optimizations

Although Algorithm 1 is already vastly more efficient than the naive approach, there are a number of optimizations that turn out to be useful:

It is usually a good idea to try the shortest inputs first since this often facilitates pruning after only a few transition-table entries have been tried.

Given a solution, it is possible to construct isomorphic solutions by swapping the names of symbols which have no predefined meaning. These isomorphic solutions can be excluded from enumeration if, at a choice point, only those alternatives are considered which come from different classes of undistinguishable symbols [1].

When backtracking occurs, Algorithm 1 moves back to the most recent choice point. But it is possible that this choice point did not influence the failure of simulation that provoked backtracking. In this case, the choice point can be removed completely (and so forth). Although this heuristic sounds intuitive, care must be taken to implement it correctly. A choice point may influence a failure indirectly by affecting another choice point which might have been removed from the stack when it ran out of choices. A safe way to make this backtrack decision is to tentatively remove the choice point and then check whether enough information is available to reproduce the simulation failure. This can be done by using a kind of simulation which uses the additional rule that undefined transition-table entries produce a special symbol "?" for undefined. Since this "error simulation" is costly, it is useful to identify special cases (which depend on the type of polyautomata) for which the backtrack decisions are easier to make.

There are many more possible approaches to reducing the search space like non depth-first tree traversal, more intelligent simulation ordering, ... But one has to be careful not to overdo it. Many heuristics that reduce the size of the search tree may increase the actual computing time. (See Section 3.1 for some examples.)

3 Applications

3.1 The Firing Squad Synchronisation Problem

The *firing squad synchronization problem* (FSSP) is a classical problem of cellular automata theory: Determine a class of one-dimensional cellular automata with von Neumann-neighborhood and the following properties:

- The initial configuration of interest has the form $\mathtt{G}\mathtt{Z}_0^{s-1}$ where $\mathtt{G}$ is called the *general state*, $\mathtt{Z}_0$ is the *quiescent state* and s is the size of the cellular array.
- The (local) *transition-function* σ has the property that $\sigma(\mathtt{Z}_0, \mathtt{Z}_0, \mathtt{Z}_0) = \sigma(\mathtt{Z}_0, \mathtt{Z}_0, \#) = \mathtt{Z}_0$ i.e. neither cells in the quiescent state nor the right border[2] are able to initiate any activity by themselves.
- After as few steps as possible the cellular array is to reach the configuration $\mathtt{F}^s$. $\mathtt{F}$ is called the *firing state*.
- No cell fires before all others fire.
- The automaton shall use as few states as possible while working for arbitrary sizes. This excludes trivial solutions using a counter in each cell.

The figurative interpretation is that a general (the leftmost cell) wants to make all his soldiers fire synchronously using neighborhood communication only. In [3] it has been shown that at least $2s - 2$ transitions are necessary to achieve synchronization and time-optimal solutions have been developed in [3], [1], [4], and [5] employing 16, 8, 7, and 6 states respectively (the border state is not counted).

Balzer [1] implemented a search algorithm in order to prove that there can be no four-state solution. However, his interpretation of the backtrack heuristics from Section 2 was not correct, rendering the proof incomplete. The corrected heuristics increases the search space from about 60 000 nodes to about 16 000 000 nodes.[3] But this is no problem on today's machines.

Our implementation employs the heuristics described in Section 2. Many expensive "error simulations" can be saved by using the following observation: If the choice point under consideration was discovered for the same input for which the simulation ran into a dead end, and, within the current space-time diagram, the error-point lies within the event horizont of the choice point, then the choice point is guaranteed to influence the error point (but not vice versa). The parallel implementation needs less than 11 seconds to prove that there is indeed no solution to the FSSP with four states. Figure 2 shows a four-state automaton that works up to array-size eight.

Once this result was obtained, measures were taken in order to solidify it. After all, the new algorithm could still be wrong. First, a stripped-down sequential version along the lines of the basic Algorithm 1 was written. After many hours it yielded the same result. (The source code can be found in [6].) Furthermore, the search algorithm

---

[2] "#" is a border state which is assumed to be immutable and which is only introduced in order to avoid tedious treatment of special cases for the leftmost and rightmost cells.

[3] This is still a small number compared to the $1.2 \cdot 10^{27}$ possibilities, a brute force algorithm would have to consider.

```
#G.# #G..# #G...# #G....# #G.....# #G......# #G.......#
#GG# #GX.# #GX..# #GX...# #GX....# #GX.....# #GX......#
#FF# #XXX# #XXG.# #XXG..# #XXG...# #XXG....# #XXG.....#
     #GGG# #GX.G# #GX.X.# #GX.X..# #GX.X...# #GX.X....#
     #FFF# #XXXX# #XXG.X# #XXG.G.# #XXG.G..# #XXG.G...#
           #GGGG# #GX.GX# #GX.XXG# #GX.XXX.# #GX.XXX..#
           #FFFF# #XXXXX# #XXG.XX# #XXG.G.X# #XXG.G.G.#
                  #GGGGG# #GX.G.G# #GX.XXGX# #GX.XXXXG#
                  #FFFFF# #XXXXXX# #XXG.XGX# #XXG.GGXX#
                          #GGGGGG# #GX.G.GX# #GX.XXGXG#
                          #FFFFFF# #XXXXXXX# #XXG.XGXX#
                                   #GGGGGGG# #GX.G.GXG#
                                   #FFFFFFF# #XXXXXXXX#
                                             #GGGGGGGG#
                                             #FFFFFFFF#
```

Figure 2: Space-time diagrams for a four-state "near miss".

was defined in a more formal way (using rewrite rules). This made it possible to prove the completeness of the search algorithm at least on an abstract level. The remaining question, whether one accepts the output of a C program as a proof, is a philosophical question beyond the scope of this paper.

Naturally, it was also tried to resolve the question if there is a five-state solution which would settle the FSSP for good. But it can be estimated that the best current algorithm would take about 10^{16} times the age of the universe on a MasPar.

It was also tried to devise better algorithms that are able to prune the search space more efficiently. For example, an algorithm has been implemented that can find the effect of a new transition-table entry on *all* inputs quite efficiently. But the reduction in search space is not worth the additional overhead. Balzer [1] tried to apply AI methods like constraint propagation with the same negative result.

Given that there *is* a solution to the five-state problem, some other approaches are possible. First, if there were a very large number of solutions, a randomized search algorithm would probably find one. But a 24-hour run on the MasPar did not produce anything. An approach taken by Balzer is to postulate certain properties of a solution and to search for a solution with these properties. But the wrong heuristics in Balzers implementation made this approach look better than it is. Furthermore, his strongest postulates do not hold for Mazoyer's six-state solution [5].

3.2 Recognition of $\{ww^R\}$ and $\{ww\}$ by Homogeneous Trellis Automata

A very simple, yet interesting model of systolic computation are homogeneous trellis automata [7]. In their simplest form they consist of identical nodes connected in a conceptually infinite, trellis-like triangular structure (Figure 3). Each node takes two inputs from below which belong to a finite alphabet A and computes a single output using a function $g : A \times A \to A$. The output is transmitted to the adjacent nodes in the row above. All nodes work synchronously and they can be used to accept languages over $X \subseteq A$. Words to be tested are input at a level corresponding to their length. A word is accepted iff the top node outputs an accepting symbol from $A_0 \subseteq A$.

It is still an open question what exactly is the computational power of homogeneous trellis automata. A particularly interesting open problem is whether the language $\{ww|w \in \{\mathtt{a},\mathtt{b}\}^+\}$ can be accepted. This is striking because the language is quite simple and the very similar languages $\{ww^R|w \in \{\mathtt{a},\mathtt{b}\}^+\}$ (palindromes) and $\{w@w|w \in \{\mathtt{a},\mathtt{b}\}^+\}$ can be accepted.

The search algorithm was adapted for trellis automata in order to help answer this question. All words over $\{\mathtt{a},\mathtt{b}\}$ up to a certain length are used as test inputs and they are tried in order of increasing length. This has the advantage that a word xwy can be tested very quickly: The automaton's behavior for xw and wy is already known and all that remains to be determined is the output of the top-level node. Since there are only two input symbols, the necessary computations can be done very efficiently using bit arithmetic.

Using this approach it was possible to show that no homogeneous trellis automaton with less than seven symbols can accept $\{ww|w \in \{\mathtt{a},\mathtt{b}\}^+\}$. Furthermore, if a seven-symbol solution exists it will have exactly two accepting symbols.

This result is only meaningful if it can be compared to the number of symbols necessary to accept the palindrome language. It turned out that the known solutions use much more than seven symbols.[4] But the search algorithm was able to find solutions using only five symbols. It yielded 16 candidate solutions which work for all inputs with length up to 20. Figure 4 shows one of these solutions whose correctness has been proved manually for inputs of arbitrary length.

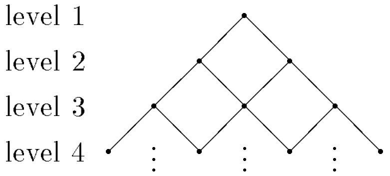

Figure 3: Homogeneous trellis structure.

	a	b	c	d	x
a	x	c	c	x	
b	c	x	x	c	
c	d	c	c	d	a
d	c	d	d	c	b
x			d	c	b

```
         x
        b c
       d x d
      c a d c
     c c x c c
    c c b c c b
   x d d x d d x
  a d c a d c a d
 c x c c x c c x c
a b b a b b a b b a
```

Figure 4: Solution of the palindrome problem with an example.

We therefore have an (admittedly weak) result about the original question: Accepting $\{ww|w \in \{\mathtt{a},\mathtt{b}\}^+\}$ takes at least two symbols more than accepting palindromes. But the solution of the palindrome problem and its proof are also interesting for themselves. It can be claimed that the "ultimate" solution for a classical problem has been found. Furthermore, it is quite interesting to look at a proof of a computer generated algorithm. On the one hand, verifying such an algorithm might be considered as a worst-case reverse-engineering problem since nothing is known about the "ideas" behind the algorithm. On the other hand, it may be hoped that the search "discovered" interesting principles because otherwise the solution would not be so short. It turned out that both is true. On the one hand, the algorithm uses some quite involved information encoding. On the other hand, it was possible to discover a quite simple invariant which forms the backbone of the proof.

[4] How many symbols are used exactly is hard to say, because often a different interconnection scheme is used.

4 Parallel Search

The search algorithm is parallelized using a straightforward general approach. The search-tree is decomposed into disjoint subtrees which can be searched independently [8, 2]. Section 4.1 shows how this decomposition can be made efficient using an appropriate (dynamic) load balancer. Section 4.2 discusses implementation issues which might also be interesting since no comparable implementations on a MasPar are known to the author. Another key issue is how a SIMD computer can effectively support the traversal of independent subtrees in the context of complex control structures. This question, which is orthogonal to the topics discussed here is covered in [6, 9].

4.1 Load Balancing

A sequential search algorithm starts with one single root node. In order to assign work to all PEs, the search-tree has to be expanded to a sufficient degree and the generated nodes should be evenly spread over all PEs. Interestingly, this is possible using only one single broadcast of the root node and $O(\log(P))$ node expansions (if P is the number of PEs). All PEs start with the root node. Whenever a node is expanded, exactly one successor is selected based on the information provided by the PE index. This process is continued until no two PEs work on the same nodes. This mechanism can be implemented using simple **DIV** and **MOD** arithmetics [10].

However, heuristic search trees in general and search trees for polyautomata in particular often have a very irregular structure; therefore, this static load balancing approach can only serve as an initialization method which prepares the ground for the dynamic redistribution schemes described in the following.

General Principles of Dynamic Load Balancing

Dynamic load balancing comes in many guises. Therefore, it makes sense to fix a few simple principles which have proved to make sense for the given application domain:

- In order to keep network traffic low, work is sent from busy to idle PEs only.
- The search tree is partitioned by splitting the bottommost entry of the stack that has open alternatives, because this is the place where large, untouched portions of the search-tree are most likely to be found.
- A work exchange phase is initiated whenever the percentage of busy PEs drops below a certain limit α. This scheme is simple and quite efficient (efficiencies of around 80 % have been achieved).

What remains to be determined is which communication patterns are useful to find partners for work exchange.

Random Permutations

A quite efficient load balancing scheme can be implemented using an almost trivial idea: Idle PEs simply choose a communication partner at random and probe it for work (e.g.

[8]). For the SIMD setting, a slight modification was introduced. Instead of selecting communication partners independently, one global random number $r \in \{1, \ldots, P-1\}$ is generated and an idle PE with number i communicates with PE i **XOR** r. (Assuming that P is a power of 2.) This guarantees that no two PEs will try to fetch work from the same place. (The function $\lambda i.i$ **XOR** r is bijective.) Although not all possible permutations can be generated using this scheme, all PEs are treated equal and every PE can reach all other PEs.

Another advantage is that it is very efficient on a hypercube. Even if all PEs want to communicate and if communication is possible only along one dimension at a time (as for the Connection Machine) there will never be two messages that want to traverse the same link into the same direction. Furthermore, at most one message needs to be stored in a given node at a given time. The proof for all these properties is very simple: Assume for a moment that r is a power of two. Then the permutation simply consists of swapping messages along one hypercube dimension. The permutation for an arbitrary r can be composed of k of these elementary permutations if k is the number of one-bits in a binary representation of r.

Rendezvous

It was possible to speed up the search by another 13 % using a more informed load distribution strategy.[5] The scheme proceeds in three phases: First, the addresses of all idle PEs are stored in the PEs with lowest number. Then the busy PEs retrieve these addresses. If there are less than 50 % idle PEs, those PEs which have the highest load are served first. Load is estimated using the number of times a given piece of work has been split. Finally, the busy PEs that received an address of an idle one share their workload. Figure 5 gives an example of this process.

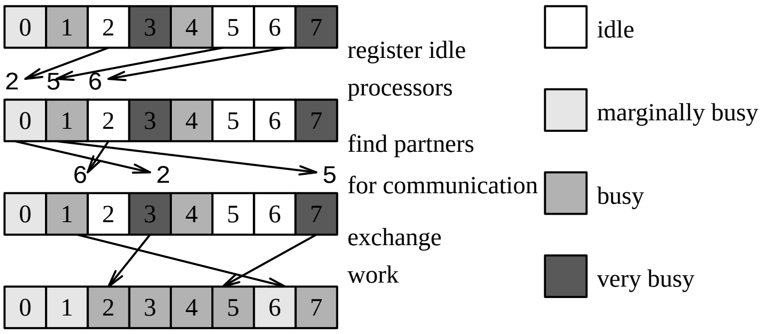

Figure 5: Load distribution by rendezvous with sorting.

A multitude of reasons is responsible for making this *rendezvous* scheme perform better than random permutations:

- As long as 50 % of the PEs are able to share work, all PEs will be busy after a load balancing step.

[5]This scheme was developed independently of the almost identical one in [2].

- Larger pieces of work are transmitted on the average.
- The heuristics used in transition-table search make it possible that PEs traverse parts of the search-tree which would be pruned by the sequential algorithm. This becomes less likely as the search-tree is split nearer to the root.
- The overhead due to sorting and rendezvous can be made smaller than the time spent for transmission of work. In part, this may be due to the fact that accessing locally adressed memory is expensive on current SIMD machines. Accessing the work to be transferred can therefore be as expensive as actually transmitting it.

4.2 Implementation Issues

The implementation was done using the data-parallel ANSI-C extension MPL and all measurements were done on a 16384 PE MasPar MP-1 using the firing squad synchronization problem (see Section 3.1) with four states.

Besides the more fundamental optimizations described above, there were a large number of more mundane but equally important opportunities for tuning: Packing the data before transmission, using low-level router properties, providing appropriate `register` annotations, replacing some `&&` and `||` operators by `&` and `|`, choosing data types of lowest possible precision, hand-coding a multi-dimensional array-access, and replacing a function call by a table lookup.

What makes all these details interesting is that they would not have been necessary on a sequential machine. The optimizations would be irrelevant or could be performed by the compiler. It is not clear which optimizations could be automated by the compiler of a (possibly higher level) parallel language.

All in all, the final version of the program was about 38 times faster than a SPARC-station II. But the first parallel version without optimizations was no faster than a workstation. Furthermore, only one optimization yielded a speedup of more than 100 % (switching from a simple nearest neighbor communication scheme to random permutation load balancing). The remaining order of magnitude in acceleration was due to many small steps which look pretty limited in isolation (note that speedup factors do not add, they multiply).

5 Conclusions

This paper centers around the idea of using heuristic search for determining transition-tables of polyautomata. An abstract algorithm is presented that in principle works for any kind of polyautomaton. Using this approach it is possible to correct Balzer's proof that there is no four-state solution to the firing squad synchronization problem. The same search-method yields the smallest homogeneous trellis automata accepting the palindrome language (involving only five symbols). One solution has been verified manually resulting in some interesting insights into computer generated algorithms. It can also be proved that the very similar language $\{ww|w \in \{\mathtt{a},\mathtt{b}\}^+\}$ requires at least a seven-symbol transition-table.

Like all attempts to infer algorithms automatically, transition-table search suffers from a rapid combinatorial explosion. This can be observed for the five-state FSSP and for seven-symbol trellis automata. Although some simple heuristics prove to be useful in reducing the computational complexity, it often turns out that more sophisticated heuristics incur an excessive overhead. This renders AI-like methods less useful than clever implementation in an imperative language.

Transition-table search can be speeded up further by using parallelism. This proves to be effective for massively parallel computers and even SIMD machines. A key issue for parallel search is load balancing. A simple form of initialization can be realized almost without communication. For dynamic load balancing, rendezvous with sorting by workload proves to be very effective. A method that is similarly efficient uses a special kind of random permutations. It is very simple and has low communication overhead. However, if the search algorithm is to yield a sizeable speedup over a state-of-the-art workstation, careful optimization is necessary.

References

[1] R. Balzer. *An 8-state minimal time solution to the firing squad synchronization problem.* Information and Control 10, 22–42, 1967.

[2] C. Powley, C. Ferguson, R. Korf. *Depth-first heuristic search on a SIMD machine.* Artificial Intelligence 60, 199–242, 1993.

[3] A. Waksman. *An optimum solution to the firing squad synchronization problem.* Information and Control 9, 66–78, 1966.

[4] H.D. Gerken. *Über Synchronisationsprobleme bei Zellularautomaten.* Masters thesis, University of Braunschweig, April 1987.

[5] J. Mazoyer. *A six-state minimal time solution to the firing squad synchronization problem.* Theoretical Computer Science 50, 183-238, 1987.

[6] P. Sanders. *Suchalgorithmen auf SIMD-Rechnern — Weitere Ergebnisse zu Polyautomaten.* Masters thesis, University of Karlsruhe, August 1993.

[7] K. Čulik II, J. Gruska, A. Salomaa. *Systolic trellis automata II.* International Journal of Computer Mathematics 16, 3–22, 1984.

[8] R. Finkel, U. Manber. *DIB— A distributed implementation of backtracking.* ACM Trans. Prog. Lang. and Syst., 9, 235–256, April 1987.

[9] P. Sanders. *Emulating MIMD behavior on SIMD machines.* Int. Conf. on Massively Parallel Processing, Delft, Elsevier 1994 (to appear).

[10] O. El-Dessouki, H.W. Huen. *Distributed enumeration on between computers.* IEEE Trans. on Computers 29, 818–825, 1980.