Parallel Adaptive Finite State Automata

Ricardo L. Rocha
Computer Engineering Department, USP – Universidade de São Paulo
Av. Prof. Luciano Gualberto s/n Travessa 3 nº 158 - Cidade Universitária
São Paulo, SP ZIP 05508-900, Brazil
luis.rocha@poli.usp.br

and

César E.C. Garanhani
Computer Engineering Department, USP – Universidade de São Paulo
Av. Prof. Luciano Gualberto s/n Travessa 3 nº 158 - Cidade Universitária
São Paulo, SP ZIP 05508-900, Brazil
cgaranhani@yahoo.com.br

Abstract

The interest on parallelism has grown in many areas of technology. Hardware development has evolved greatly in the last years, leaving to software developers the goal of building better tools and compilers for parallel computation. Also, symbolic computation must take advantage of parallel computation. The proposal contained in this paper is to use functional languages as a tool to implement adaptive automata using the concepts of symbolic computation.

Keywords: Adaptive Automata, symbolic computation, parallel computation.

1 INTRODUCTION

Processors, nowadays, have reached its maximum power or are up to reach it, for the limiting factor is speed light, which we cannot overcome with current studies. The way of increasing computation power has been making many processors work together, in parallel.

Many results on parallel hardware development has been reached, what leaves to software researchers the task of creating tools and models that meet the parallelization requirements [1].

Symbolic computation also has the same strategy: divide and conquer [1]. For that reason, the proposal of this paper is to find a better implementation for computation models called adaptive automata, increasing its power and speed.

2 FUNCTIONAL PARALLEL PROGRAMMING

Many physical and mathematical researches have computational problems, because the widely used single sequential computer has significant limitations. It has been consent that the only way to solve this problem is totally different machine architecture, and the most accepted solution nowadays is the use of parallel computation machines [2].

Significant development in parallel processing has taken place in the last years. Many parallel computers have turn into reality on commercial or academic environments. These parallel architectures include message-passing computers, shared memory multiprocessors, systolic arrays, massively SIMD machines, among others. These machines are the promise for future improvements on computation, mainly because the speed up on computation they offer [3].

One of the simplest parallel architecture is a multiprocessor system, connected with a common data via or a net, communicating with each other by message passing. This kind of architecture is easily built, is extensible and, the most important, avoids the famous “von Neuman bottleneck” [1].

Although the technological advances for the construction of parallel microprocessors and computers, the development of algorithms and programming languages for those machines has not evolved in the same speed [2]. Today, there is a software lack, in comparison with hardware, in the development of parallel computation. For this reason, the most critical phase in the development of a parallel system is software development. The future of parallel computation depends on the creation of simple, but powerful, programming models that makes transparent for the programmer the nuances of hardware project. Many researchers say that the imperative programming model is inadequate for those systems, since those languages are intrinsically tied to the “von Neuman model” of one operation at a time [2].

One alternative to solve this is a parallel programming language based on the functional paradigm for computer programming, or the *para-functional* computation. In this solution, the multiprocessor system

is seen as a single autonomous system where the program is mapped, instead of a bunch of independent processors working together, with complex message-passing mechanism and synchronizations. This way, the same code could be used to execute on a single processor or a multiprocessor system [2].

2.1 Functional paradigm and parallel computation

It is not natural to use programming languages based on the imperative paradigm to build a program for a parallel, multiprocessor based system. Many functions and directives must be used to achieve parallelism. Besides, the programmer must know exactly what is going on the program (on which processor the chunk is running, how many processors are being used, which processor is loaded and which is free, etc.) [2]. All those factors bring difficulties to the system builder and the programmer. Makes more sentence using a programming language that is not a sequence of commands to the processor to build this system, and one of the best candidates is the functional paradigm.

In a pure functional language, the execution of a function does not produce side effects (such as assignment), meaning that no matter the order of execution of the program, the result will always be the same. This property adds great value to such a language when we think of parallel programming, because the development and debugging of a program may be done on a sequential, single-processed machine (what makes the debugging easier) and then be run on a multi-processed one (that increases the computation power) [2].

The key point in this discussion is that parallelism is implicit on functional languages and its semantics supports it [2]. There is no need of constructions for message passing or any kind of synchronization, or constructions and directives to explicitly make a chunk of code parallel, like **forall**, or **parbegin** … **parend**.

Unfortunately, we do not live on a perfect world. The implementations of functional languages, such as LISP and Scheme, have some constructions that produce side effects [2], undesirable for parallel computation. Nevertheless, besides the constructions with side effects being an inheritance of the "von Neuman" architecture, a language with no side effects at all seems too radical for us, human beings [1].

2.2 Functional programming

The functional paradigm is made only of mathematical functions, that is, the base element of those languages is function. A program built under this paradigm consists of definitions and expressions, and the computer to evaluate them. Therefore, developing a program is to build a function that will solve a problem. The computer, then, works as a calculator machine, that takes the parameters of the problem and evaluates the expressions by reducing them to a normalized form.

We can say there are three classes of functional languages:

- **Strict languages**, where the arguments of a function are evaluated before the function itself;

- **Non-strict languages**, on which the arguments are evaluated only if their results are needed (known as *lazy evaluation*); and

- **Hybrid languages**, on which simple arguments, as integers, are evaluated before the function call, but complex arguments, like lists and vectors, are evaluated only if needed, as in non-strict languages.

On functional programming, parallelism can be achieved implicitly or explicitly. Achieving parallelism in strict languages is very simple. Whenever a function has more than one parameter they can be evaluated in parallel, since the parameters are expressions and must be evaluated before the evaluation of the function itself.

However, on non-strict languages the implicit parallelism can be achieved by evaluating the arguments, in parallel, when they are needed. The problem is that the nature of the arguments cannot be foreseen before the function is evaluated, what makes it difficult to do a correct parallel evaluation of the parameters.

Lazy evaluation is a very important feature of non-strict functional languages. By assuring the arguments will be evaluated only if its value is needed, it allows functions to have infinite number of arguments. For example, the expression

$$w = ((x.xx)(x.xx))$$

does not have a normal form, meaning that no matter the number of reductions, we can never find a form that cannot be reduced.

On the other hand, the normal form of the expression

$$w = (y.z)((x.xx)(x.xx))$$

is z. In this example, if the expression were evaluated strictly x would be reduced before w, what ties up w to x, and the reducing processes would go forever.

Lazy evaluation is based on the normal form of the expression to evaluate the expression. It always evaluates the most left outer expression, avoiding infinite computation. But this characteristic makes it difficult to reduce the expressions in parallel, because it limits the parameters to one reduction each time. For single processing systems, it is just fine.

Also, on a non-strict functional language, it may be impossible to evaluate an expression that never ends. It is because the *semantic criterion* of those languages does not allow infinite type of evaluation, unless it is undefined when called. So, to do secure parallelism on those languages, it is important to

define it on its semantic criterion.

Besides the implicit parallelism, we can define structures that explicitly evaluate expressions in parallel with other evaluations. This way, we can define structures that explicitly start a thread, or make parallel loops.

One way to do this on functional programs is the form called *promise*. Promises are, like the name says, a promise that the computation will be done, but it does not mean it will be done immediately. We can say that a *promise* is a place-holder for the value resulting from the evaluation of the promise. At a first sight, it seems that a promise is just a kind of lazy evaluation. And it is! The difference is that a promise computes its evaluation in a different process than the one running the calling function. We can say that a promise is a parallelization of a lazy evaluation.

Promises work like this: when the program reaches the promise function and reduces it, the promise creates a new process and returns immediately. When a promise finishes its computation, the resulting value of the expression is cached. If the calling function needs the promised value, the cached value is returned.

If the calling function needs the promised value before its computation has finished, the calling processes sleeps until the promise computation finishes. Then, the process of the calling thread receives a signal to continue and the cached value of the promise is used. If the promised value is needed any other time, the cached value is returned. This way, the semantics aspect of the program does not change in any way besides the introduction of concurrency.

3 SYMBOLIC PARALLEL PROGRAMMING

Computer programs are different in many aspects and dimensions. In one of these dimensions, the programs can be predominantly numeric in one hand and symbolic in the other. The difference of numeric and symbolic computation suggests that different approaches are made for each one [1].

Lately, the parallel numeric programming has had much attention of researchers and many results were reached: numeric parallel computers with high performance and different degrees of concurrency; compilers and tools that automatically infer parallelism to the program, or that lets the user explicitly specify parallel blocks on the program, etc.

This kind of computation emphasizes the arithmetical computation and its main function can be defined as passing numbers to an arithmetical unity that will compute the result. Those programs are, in general, independent of the data flow [1]. Matrices and vectors are common data structures on those programs and SIMD computers and compute them in parallel.

In contrast, the symbolic computation emphasizes the ordering of data. For that reason, symbolic programs are written with functional languages, which can deal with those data structures a lot easier

than conventional (imperative) programming languages [1].

In a general analysis, we can say that the main function of symbolic computation is to reorganize data sets so the information contained on them can be retrieved easily. Data mining, compilers, database management, symbolic algebra, and programs that learn are good examples of primordially symbolic algorithms.

The structure of symbolic computation suggests that its sequence of operations, usually, depends on the data being manipulated and its response to compilation directives is meaningless. For that reason, SIMD machines cannot help increasing performance on those programs. Besides, it does not make sentence parallelizing loops, because functional paradigm rather makes use of recursive functions instead of loops [1].

Therefore, we must find another kind of parallelization on symbolic computation. One way of doing this is trying to parallelize the logic of the model. For example, in a search tree, we can parallelize the data insertion, trying to insert one data o each side of the tree. In the case of adaptive automaton, the proposal is to parallelize its execution [1].

4 ADAPTIVE FINITE STATE AUTOMATON

Adaptive finite state automata are self-modifying automata. They were idealized to solve problems that **finite state automata** (FSA) are not able to do, the context-dependent languages. That means adaptive automaton has the computational power of a Turing machine, that is, it can represent any computational problem [5].

Like FSA, the adaptive automata are behavior models consisting of states, which represent information about the past, that is, it reflects the input changes from the system start to the present moment; transitions, that consists of an initial state, a condition (or symbol) and an exit state. A transition indicates a state change in response to a fulfilled condition (matching symbol); and adaptive actions, which are functions that may change the automaton structure during a transition.

The adaptive automaton works much like a FSA, taking input symbols and running transitions, unless it reaches a syntactical construction not yet seen. When this happens, the adaptive automaton launches an adaptive function that modifies the automaton initial model. The execution of the automaton then keeps going on normally [4].

After the execution of an adaptive function, the original automaton model is lost and the execution of the automaton continues on the new model generated [4].

An adaptive automaton is represented as

$$AA = (S, SM, \Sigma, \Gamma, T, z_0, s_0, F, E, \Phi),$$

where, like in the FSA, S is the state set, Σ is the input symbol set (alphabet), T is the set of transitions, s_0 is the initial state and F is the set of acceptance states. Different from a FSA, the adaptive automaton has a set of adaptive functions, represented with Γ. The other elements of the set are specific for structural construction of the adaptive automaton and they will not be discussed here [5].

An adaptive function can be evocated before or after (or both times) when a transition happens. The functions evocated before the transition are called **pre adaptive function**; likewise, the functions that happen after the transition are called **post adaptive function**.

An adaptive function can only execute a few **elementary actions**, and they must obey a certain order of execution. They are:

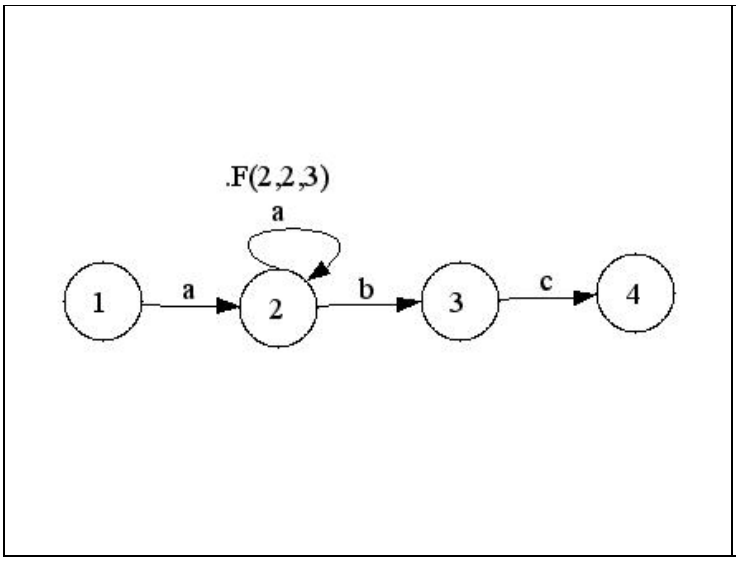
- Search transitions: ?[transition];
- Remove transitions: -[transition];
- Inserting transitions: +[transition].

Besides, inside a function it is also possible to create states through **generators.** Generators are auxiliary structures that can create new states on the adaptive automaton and name then uniquely. They are represented in the adaptive function with an asterisk (*) sing after its name [4].

A good example of an adaptive automaton is the one that solves the language L = $\{a^n b^n c^n\}$. This language is classified as a context dependent language. It means that finite state automatons, with a stack or not, cannot accept this language. But the adaptive automaton can. The initial configuration of the automaton would be like figure 1.

The •F on the transition means that the function F will be called before the transition happens, that is, it is a pre adaptive function on this transition. Pos adaptive functions are represented as F•.

So, let *w=abc* be the input string. The automaton, then, works as a normal FSA, taking the path where there is no adaptive function. Next, let *w=aabbcc* be the input string. The automaton, as it is, will not accept this string, so when it reads the second symbol *a* it will execute the adaptive function F, which is defined in the table below. The resulting automaton is represented in figure 2.



Adaptive functions:

```
F( s1, s2 ,s3 ):
{
  g*  // generators. Each one
  h*  // generates a new state.
  j*:
  -[(s1,a):F→s1];
  -[(s2,b)→3];
  -[(s3,c)→4];
  +[(s1,a)→g*];
  +[(g*,a)→s2];
  +[(s2,b)→h*];
  +[(h*,b)→3];
  +[(s3,c)→j*];
  +[(j*,c)→4];
  +[(g*,a):F(g*,h*,j*)→g*]
}
```

Figure 1: Adaptive automaton that accepts the input $a^n b^n c^n$: Initial configuration.

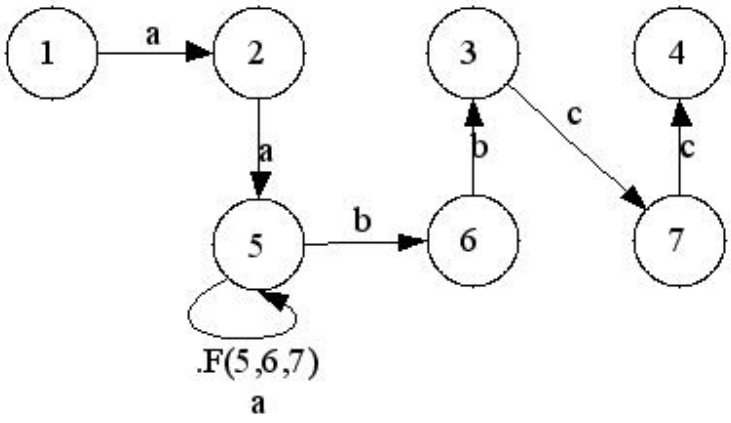


Figure 2: Adaptive automaton that accepts the input $a^n b^n c^n$: Configuration after reading the input aabbcc.

The adaptive automaton is non-deterministic by definition, because any time you run an adaptive function, the automaton structure may change and it is not possible to determine when or if its computation will end.

4.1 Parallelism in adaptive automata

An important feature of an automaton is the ability it has to resolve ambiguity. Ambiguity in automata happens anytime the automaton can take more than one path after reading (or not – empty transitions) a symbol from the input string. Whenever it happens, the automaton must decide what to do, where to go.

Intuitively, we can keep track of the path the automaton is taking. Looking at the graph that represents the automaton, we can say what path to take easily. However, when we ask the computer to do the same, it is not as easy. The implementation is very important. One technique largely used for automaton acceptance implementation is the mechanism of *backtracking*. This mechanism works as this: each transition made is put on a stack. When an ambiguous path is seen, the program decides for one of them and goes on with the execution normally, putting the chosen path on the stack. If the automaton reaches a dead-end path and does not recognize the input string, it goes back (backtracks) to the first ambiguous path on the stack and takes the other one. It does this until it recognizes the input string or until all the possible paths are run.

This algorithm is very slow, because whenever it finds more than one path, it has to run them forward and backward.

The same happens with the adaptive automata. Whenever it reaches an ambiguous path, it has to choose among them. Even worse, whenever it finds an adaptive function that changes the automaton structure, it has to keep the old model, because if the new model does not accept the rest of the input string, the algorithm can take back the old one and continue the execution.

The proposal of parallel adaptive automata is to make an implementation on which anytime an ambiguity happens, a new process is created to each ambiguous path but one. So, each one of the paths are executed in a different process. The same happens if an adaptive function is found: a process is created to execute the function and finish the execution of the new model, while the old process keeps the execution of the old model.

In theory, this method can execute the automaton faster because it avoids the backtracking mechanism. Also, this implementation is closer to the theorical definition of the automaton, which suggests the ambiguous paths should be executed in parallel, not one at a time.

The implementation of such adaptive automaton using a functional language may be done using structures implemented on the language such as promises.

5 IMPLEMENTATION

All of this discussion leads to an implementation of an adaptive automaton using a LISP like language, for all the facilities described on functional programming.

Adaptive functions:	Transitions:

F(s1, s2 ,s3):
{
  g*
  h*
  j*:
  sub[(s1,a):F -> s1]
  sub[(s2,b) -> 3]
  add[(s3,c) -> 4]
  add[(s1,a) -> g*]
  add[(g*,a) -> s2]
  add[(s2,b) -> h*]
  add[(h*,b) -> 3]
  add[(s3,c) -> j*]
  add[(j*,c) -> 4]
  add[(g*,a):F(g*,h*,j*) -> g*]
}
	[(1,a):F(2,2,4) -> a]
	[(1,a) -> 2]
	[(2,b) -> 3]
	[(3,c) -> 4]
	
	initial: 1
	accept: 4

Figure 3: Definition of an adaptive finite state automaton in the program.

In this implementation, to create the initial adaptive finite state automaton, the user must only give the production and the functions set to the program, which will then define the automaton formally, filling the parameters to construct the automaton. That is, giving the definition of the function and transitions of figure 3 defines the adaptive finite state automaton

$$ASA = (\{1,2,3,4\}, SM, \{a,b,c\}, \{ f(s1,s2,s3) \}, T, z_0, 1, \{4\}, E, \Phi),$$

where T is the set of transitions defined in the figure.

What happens if this automaton tries to accept the symbols "*a b c*"? On sequential implementations of such automaton, when you read the first symbol *a*, the automaton would first run the adaptive function *f*, as it is the first transition of this automaton, create a new automaton (backing up the symbol read and the original automaton, to be able to backtrack). This new automaton would look like the one in figure 2. When the second symbol (*b*) is read, the automaton cannot go forward, since there is no transition from *5* with symbol *b*. The program would have to recover the original automaton and try the second transition.

On the parallel implementation, the first two transitions run at the same time. This is done by launching a thread that will run the first transition, while the second will run on the main thread. Also, yet another thread will be launched when the adaptive function is run, because in this implementation, every time an ambiguous path is seen, or whenever an adaptive function is run, a new thread is created.

6 CONCLUSION

The future of parallel programming depends on creation of simple, but efficient models, such that it makes it transparent for the programmer the system architecture where he will develop or run the program. A natural choice for this task is the functional programming languages, whose features meet almost exactly these requirements.

Furthermore, the higher level of functional programming languages offer make much easier to reproduce the concepts of symbolic computation. Yet, the computational power of symbolic models on those languages has increased a lot with the inclusion of constructions to parallelize the language. Thus, programming adaptive automata using this kind of parallel language has many advantages.

Like in every computer application, the parallel implementation of automata has advantages and weaknesses. The advantages are the that it is a simpler method of implementation, since you do not have to worry about saving the automaton configurations every time there is ambiguity or automaton structure changes. Also, it is a faster algorithm than sequential ones with backtracking.

In the other hand, one of the drawbacks of this algorithm is that the programmer must now worry about thread management. For example, suppose you have an automaton with some ambiguous paths and that it recognizes the entrance sequence on the shortest path. The other running threads must stop, since the automaton has already recognized the sequence. In this implementation, this is done by keeping all the created threads on a list and, when a thread reaches a valid end, a simple

```
(map kill-thread (list-of-threads))
```

is needed.

Another point that could be a negative aspect on this implementation is that, depending on the automaton, too many ambiguous paths may appear and, thus, too many threads will be created, demanding more system resource to be spent.

For future works on the parallel adaptive automaton, a statistical study of how thread creation can affect the performance of the automaton must be done to validate the theoretical performance enhancement of this algorithm. Knowing the result of this study can make it possible to implement more parallelism on this automaton to improve its performance, such as running searches, deletion and addition of transitions inside an adaptive function in parallel.

7 REFERENCES

1. Halstead, Robert H., "*Parallel Symbolic Computing*", The institute of Electrical and Electronics Engineers, Inc., 1986.
2. Hudak, Paul, "*Para-funcional Programming*", The institute of Electrical and Electronics Engineers, Inc., 1986.
3. Kumar, Vipin, "*Introduction to Parallel Computing – Design and analysis of Algorithms*", The Benjamin/Cummings Publishing Company, Inc., 1994.
4. Neto, João José and Pariente, César Alberto Bravo, "*Adaptive Automata - a Revisited Proposal*"**.** Lecture Notes in Computer Science. J.M. Champarnaud, D. Maurel (Eds.): Implementation and Application of Automata 7th International Conference, CIAA 2002, 2002, Vol.2608, Tours, France, July 3-5, Springer-Verlag, 2002, pp. 158-168.
5. Neto, J. J. "*Adaptive Automata for Context -Sensitive Languages*", SIGPLAN NOTICES, Vol. 29, n. 9, pp. 115-124, September, 1994.
6. Sebesta, Robert W. *"Concepts of Programming Languages"*, Addison Wesley, 2003.
7. Paulson, C. Lawrence, *"Foundations of Function Programming"*, C. Lawrence Paulson, 1995.