PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT. © ICOT, 1984

SYSTOLIC PROGRAMMING: A PARADIGM OF PARALLEL PROCESSING

Ehud Shapiro

Department of Applied Mathematics
The Weizmann Institute of Science
Rehovot 76100, Israel

ABSTRACT

Even though the systolic approach shows great promise for realizing massive parallelism, it has been viewed so far only as a method for designing special-purpose attached processors for conventional computers. We claim that the systolic approach has a much greater potential. That it can lead to an algorithm design and programming methodology for general purpose, self-contained, high-level language parallel computers.

This paper proposes a framework for realizing this potential, termed systolic programming. The framework comprises an abstract machine, a programming language, a process-to-processor mapping notation, and an algorithm development and programming methodology.

An implementation of this framework using available technology is currently under investigation.

1 PROLOGUE: A TRIBUTE TO OUR FOREFATHERS' WISDOM

Socrates: I understand from your note that you came to tell me about a parallel computer architecture you are thinking of...

Gera: Yes, it is a rectangular grid of...

Socrates: ...but before that, since I know so little about this topic, could you tell me how to evaluate your new idea?

Gera: Well, an important criteria for evaluating parallel architectures is scalability. It says that for twice the money you should get twice the computer. Or that the architecture should remain feasible as the number of processors goes to infinity.

Socrates: Why is this criteria important?

Gera: First, from an aesthetic point of view, a scalable architecture is more elegant and robust. From a practical point of view, we would not want to re-solve the parallel processing problem every two years afresh, when the number of processing elements per wafer doubles.

Socrates: Sounds reasonable. Are there any implications you can draw from this criteria? Properties that hold for any scalable parallel architecture?

Gera: Yes. Non-uniform costs of communication and memory reference. An architecture in which every processor is "close", in some natural sense, to all other processors, is not scalable. For example, cross-bar switches and their approximations are not scalable. So in a scalable architecture a processor would have "neighbors", and "non-neighbors".

Socrates: So it seems that some pairs of processors will have a hard time talking to each other?

Gera: Yes. So I think that ensuring the locality of communication is the critical problem of parallel processing.

Socrates: Why? Are you sure that all these processors will have so much to talk about? Won't they work most of the time on their own, and only occasionally communicate?

Gera: I don't think so. If you have a scalable architecture, and try to exploit parallel processing to its fullest, you end up breaking your problem into smaller and smaller subproblems. The resulting algorithms involve a lot of communication.

Socrates: And I thought that the real CPU-killers are number crunching programs, that, presumably, involve a lot of computation, and very little communication.

Gera: My statement is true even of number-crunching problems. Consider for example systolic algorithms. They seem to be the most promising approach to highly-parallel numeric algorithms. They show that, even for a compute-bound problem, a highly parallel algorithm involves an awful lot of communication. If this is true of compute-bound problems, it is even more so of communication-bound problems.
In other words, it seems that a highly parallel solution to any problem is communication-bound.

Socrates: You probably wouldn't make these statements without thinking you have some method for localizing communication. But I heard that the major problem of parallel processing is load-balancing. And it seems that dynamic load-balancing and maintaining the locality of communication don't mesh very well, do they? I mean, if you start moving processes around, it becomes difficult to make sure that processes that talk to each other stay near each other, right? So your load-balancing algorithm must be pretty smart, is it?

Gera: No.

Socrates: No? Oh, I see... You probably don't want to do the analysis of who talks to whom every time you spawn a process; you'd rather do it once and for all at compile time. Make the compiler map processes to processors so that they are both evenly spread and perform only local communications. So, a smart compiler, is that what you've got?

Gera: No.

Socrates: Good. I see you've learned something from me. It raises my blood-pressure when someone tells me he has a "smart gadget" for solving a problem that smells NP-hard miles away. Theory aside, it seems that designing efficient process-structures for a parallel program is as difficult as designing efficient data-structures for a sequential program. And we don't have any "smart gadgets" for the latter problem. Do we?

Gera: No, I don't know of any widely used ones.

Socrates: I am getting curious. What's the pigeon in your hat?

Gera: I don't have any, except for good old programmers and algorithm designers.

Socrates: ???

Gera: I mean that if designing efficient process structures for parallel programs and mapping them effectively on the target architecture are important and difficult problems, then we have to solve them. And we should have a way to tell the computer what our solution is. So we must have a language for describing process structures, and for describing how to map these structures on our computer. I guess that's the idea I wanted to tell you about.

Socrates: Hmmm...

Gera: Consider this: you know the systolic band-matrix multiplication algorithm of Kung and Leiserson, do you?

Socrates: Yes. I think it's ingenious, even though moving transparencies around to define it involves too much hand-waving for my age.

Gera: You would like to implement this algorithm, or this kind of algorithms, on a parallel general purpose computer, would you?

Socrates: Probably.

Gera: This algorithm wouldn't make sense if the systolic system is spread all over the place, would it?

Socrates: Probably not.

Gera: And we both agree that a general-purpose method that can automatically identifies the hexagonal process structure of that algorithm from its formal definition (if Kung succeeds in coming up with one) and finds a way to map it onto the plane is, today, out of the question.

Socrates: Well, I would not propose to halt research on automatic programming. Besides, mapping process structures seems a slightly easier then designing them.

Gera: But even if we discover such a method 2 or 20 years hence, it would need to specify the mapping of process structures somehow, right? So we're back to square one. Whether we specify the mapping of process structure manually, or let a "smart compiler" do the job, the kernel programming language of a scalable multiprocessor needs a notation for mapping processes to processors. Quod erat demonstrandum.

Socrates: Hold it, we're not over yet. Ignoring the futuristic "smart compiler", isn't your proposal a major setback for programming, in some sense? Isn't programming difficult as is? Won't it become horrendous if the programmer will have to control explicitly a new dimension, the definition of process structures and the mapping of processes to processors?

Gera: That's precisely the point. Parallel processing introduces a new dimension to programming. In addition to time and space usage, we must also control communication costs. And the thrust of my argument is that if we want to break the sequentiality barrier, we must bite the bullet. We must solve the communication problem at the algorithm design and programming level, and not rely on the hardware architect to solve it for us, because he can't.

Socrates: I like your enthusiastic style of arguing, but you didn't answer my question.

Gera: Well, there are several answers, which are substantiated by my experience with programming using a mapping notation.

Socrates: Experience? Don't tell me you've already built your computer?

Gera: No, not yet. But I have implemented a software simulator which I have been programming extensively. Its just as good.

Socrates: Good for writing a paper.

Gera: No, just as good for my argument. My answers are as follows.

One is that programming in higher-level languages is easier. When switching from sequential to parallel processing, the 2- or 10-fold speed difference between conventional languages and very high-level language programs is shadowed by the difference in the ease of programming. If the choice is between programming a von Neumann machine using a conventional language, and programming a parallel machine using a very high-level language, augmented with a mapping notation, then I choose the latter.

The second answer is that I have noticed a correspondence between the complexity of data-structures in sequential programs and communication structures in parallel programs (this is *not* a theorem). An array of data of a sequential program is mapped into an array of processes in the corresponding parallel program, where each process has only simple data-structures (scalars and I/O streams). A tree data-structure of a sequential program is mapped into a process-tree of the parallel program, where nodes are processes and edges are communication channels. The complexity of defining the data-structures of a sequential program is mapped into the complexity of defining the process structures of a parallel program and their mapping, so it seems that overall program complexity is preserved.

The third answer is that the mapping notation can be developed independently of the main program, and is relatively easy to specify (some say they are kids' stuff).

The fourth answer is that I don't see any other viable alternative.

Socrates: I am still uncomfortable with your solution. Actually, I am a bit surprised to hear it from you: haven't you learned anything from Lisp and Prolog? Isn't explicit allocation of storage an anachronism? Why is allocation of processes to processors different?

Gera: You certainly have a point. However, process-to-processor mapping has much more profound effects on the performance of a parallel program than memory-allocation strategy on a sequential program. And it is a much more difficult problem to solve automatically. Even in dynamic storage allocation, the programmer is the one who determines the data-structures used, although he does not control their location in memory. However, as you have pointed out, mapping is the simpler part. It is conceivable that common process structures could be mapped automatically, and mapping them manually in the meantime is not such a big burden.

Socrates: Maybe. But it seems to me that some load-balancing is still necessary, at least in a multi-user environment.

Gera: I agree. Such a load-balancing algorithm, however, should specify the origin, orientation, and perhaps also the density (number of processes per processor) of a systolic system. But I don't think it should twiddle with the internal mapping of the process structure. Besides, the person to implement this load-balancing algorithm should be the systems hacker, not the hardware architect, so the kernel programming language should have a mapping notation. Quod erat demonstrandum...

Socrates: This argument is getting boring. I resign.

Gera: So my architecture is...

Socrates: Before diving into the details, are their any general implications that hold for any scalable architecture programmed with a mapping notation?

Gera: Let me see. I think that the interconnection pattern of such a machine should be simple and regular. If it is not simple and intuitive, programmers will have hard time utilizing it. I also think it should be of general-purpose, and match the structure of many types of problems. Otherwise algorithm designers will have difficulties in mapping problems into algorithms with efficient communication patterns.

Socrates: Simple, regular, intuitive, huh? I like such conclusions. Motherhood and apple-pie. And what about the programming language?

Gera: Clearly it should be a high-level, expressive concurrent programming language. It should be complete, self-contained, simple, and amenable to efficient implementation. It should be easy to debug. It should have clear semantics. It...

Socrates: If you emit another buzz-word, I quit.

Gera: Sorry. I apologize.

Socrates: What makes a concurrent programming language expressive? What makes it high-level?

Gera: A concurrent programming language should be able to specify process creation, communication, synchronization, and indeterminacy. It is expressive if it can implement easily a broad range of algorithms, and if it lends itself to a rich set of programming techniques. I don't think the expressiveness of the language depends so much on its vocabulary, as on the richness of its programming idioms.

Socrates: Programming idioms? Do you want to get your computer do something useful, or to become another Shakespeare?

Gera: A language is high-level if... Well, high-level is sort of a buzz-word. I guess it means that it supports convenient ways for structuring data, and methods for treating data abstractly. It has control structures such as recursive process and function invocation. Dynamic storage allocation and reclamation. These sorts of things.

Socrates: What makes it complete?

Gera: This is another loaded term. A language is complete if it can implement its own programming environment, the tools that are necessary to use the language conveniently. If you can easily implement an interpreter for the language in the language, then many software tools, particularly debugging tools, are also easy to implement. If you can implement a complete operating system in that language, all the better.

Socrates: I think the remaining properties are rather straightforward. So let's enumerate what we have concluded to be essential properties of a general-purpose parallel computer:

1. It is scalable.
2. It has a simple and regular interconnection pattern.
3. It executes a high-level concurrent programming language, augmented with a mapping notation.
4. It is the responsibility of the application programs, not the architecture or the operating system, to ensure locality of interprocess communication.

I guess the rest is just incidental details, and I am a bit tired now, so let's hear them tomorrow.

2 INTRODUCTION

Systolic algorithms were developed by Kung and his colleagues (Kung 79, 82, Leiserson 83) in order to exploit the parallelism inherent in computational problems. The systolic approach is one of the few that shows how to put into effective use the capabilities of emerging VLSI technology. These algorithms were designed for direct implementation in hardware and therefore were rigidly constrained: they assume a synchronous array of microprocessors of a fixed size and interconnection pattern. Each processor in the array has limited processing and communication capability and a local memory that can accommodate a small program and a few registers.

As a result, an implementation of a systolic algorithm via a systolic array can solve problems of up to a fixed size only. Complex interfaces with a conventional von Neumann machine have to be developed, and methods for breaking problems to subproblems of a size that fits the systolic processor array and for combining the partial results to form the solution are also necessary. Modifying the function of a systolic array is either not possible, or can be done only by halting its execution and downloading a different microprogram (Fisher et al. 83). In addition, programming systolic arrays to execute a desired algorithm turned out in practice to be more difficult then expected, despite the relative simplicity of the algorithm each systolic processor implements.

The systolic approach was introduced as a hardware-oriented methodology for constructing special-purpose attached processors to conventional von Neumann machines. We believe that this view unnecessarily limits its inherent potential. In this paper we propose a framework for constructing general-purpose parallel computers, which incorporates the systolic approach as a software-oriented methodology for algorithm design and implementation.

The proposed framework, termed systolic programming, comprises an abstract machine, a programming language, a process-to-processor mapping notation, and an algorithm development and programming methodology. The abstract machine is an infinite processing surface (Martin 79). The programming language is Concurrent Prolog (Shapiro 83a), augmented with LOGO-like Turtle programs (Pappert 80) as a process-to-processor mapping notation. The algorithm development and programming methodology identifies two separate, though interrelated activities: the design and implementation of process structures and process behaviors (Kung 79), and the design and implementation of the mapping of these structures into the processing surface.

Algorithms suitable for this methodology are best defined in terms of a dynamically changing collection of software processes, synchronized by dataflow. They might be called *soft-systolic* algorithms, in contrast to the classical *hard-systolic* algorithms, which are typically defined in terms of a static collection of synchronous hardware processors (Vijay Saraswat, personal communication).

Some key aspects of the systolic programming approach are:

(1) It is potentially applicable to general-purpose, multi-tasking, multi-user computers.

(2) Arbitrarily large approximations to the infinite abstract machine are realizable using current technology. These machines are scalable with respect to the programming methodology, since the performance of most systolic algorithms improves linearly with the ma-

chine's size.

(3) Many important computational problems have efficient hard-systolic solutions. It is conceivable that even more have soft-systolic solutions. Also, parallel solutions to problems which, historically, are not associated with the systolic approach, such as monte-carlo simulations and array relaxation on mesh-connected computers, are also easily implementable within this framework.

(4) Programming a processing surface using Concurrent Prolog and Turtle programs is comparable to (and perhaps even easier than) programming a von Neumann machine using a conventional language.

The rest of the paper explains in more detail the components of the systolic programming approach — the abstract machine, programming language, mapping notation, and algorithm design and programming methodology — and illustrates it with several programming examples.

3 THE ABSTRACT MACHINE

The abstract machine we propose is an infinite "processing surface" (Martin): A regular arrangement of processors, each of which has some local memory, can timeshare between several software processes, and can communicate with the neighbors connected to it. For concreteness we assume that the processing surface is an infinite rectangular grid of processors, each connected to its four neighbors, but other arrangements are also possible. One suitable building block for a processing surface is INMOS's forthcoming Transputer (INMOS 83).

Various torus-based architectures that can approximate an infinite processing surface have been proposed in the literature (Martin 79, Hewitt 80, Sequin 81, Fiat et al. 84). They differ in the ease in which process arrays of various dimensions can be mapped into them with even load and with no communication penalty.

A naive implementation of a virtual infinite processing surface is obtained by folding a finite rectangular array of processors into a torus. A torus can be mapped onto the plane using constant length wires and a small number of crossovers, as shown by Zippel and Halstead (Hewitt 80). The twisted-torus, and then the doubly-twisted torus, were suggested as improvements to the simple torus by Martin (Martin 79) and Sequin (Sequin 81). A further improvement, termed Polymorphic Arrays, was proposed by Fiat, Shamir and Shapiro (Fiat et al. 84). In that paper a detailed comparison of these architectures is carried out.

A process structure larger than the actual processing surface can be spawned on it using two complementary techniques: folding and condensation. Folding treats the processing surface as infinite and, in case it is approximated by a torus-like structure, effectively folds the process structure several times around it. Since each processor can timeshare between several processes this folding will affect only the level of multi-tasking on each processor. Using folding the program can view the machine as a virtual infinite processing surface, maintaining ignorance of the particular way in which it is being approximated.

However, simple folding will not always give optimal performance, in case the program is communication bound. If the communication-to-computation ratio required by the software processes is larger then the one provided by the hardware processors, then it is better to increase the density of the mapping, by grouping adjacent processes into one processor, thus increasing the ratio of inta-processor to inter-processor communications.

It turns out that for some soft-systolic algorithms an optimal mapping density, in which computation and communication are balanced, can be computed analytically, as will be elaborated in a subsequent paper. This optimal density has the property that the performance of the algorithm on a very large (even infinite) processing surface will not improve even if the process structure is mapped with with lower density (thus reducing the computation load per processor) neither will it improve on a very small processing surface even if spawned with higher density (thus reducing the communication load per channel). In other words, in some cases process structures can be mapped on a virtual infinite processing surface in an optimal way, regardless of the actual dimensions of the finite processing surface that approximates it.

Current technology is sufficient for building a processing surface. One of the more difficult questions the architecture of a general purpose processing surface must address is the interface to and usage of external I/O devices. One feasible approach is to connect devices such as a disk drive and a local-area network interface to the processing surface in regular intervals.

4 THE PROGRAMMING LANGUAGE

In principle, the systolic programming approach is insensitive to the particular programming language chosen, as long as it is expressive enough and is amenable to efficient implementation. A programming language for a processing surface should be able to specify the dynamic creation of processes, the formation of inter-process communication structures, and process behaviors. Process behaviors include process communication, synchronization and indeterminate actions, as well as conventional control and manipulation of data.

It seems that at least five major candidates may claim to have this ability, namely CSP-based languages (Hoare 78), dataflow languages (Ackerman 82), functional languages (Friedman and Wise 78, Henderson 82), Actor languages (Hewitt 80), and logic languages (Kowalski 74, Clark and Gregory 81, 84, Shapiro 83a).

A detailed comparison of the expressiveness of these languages and their amenability to efficient implementation is outside the scope of this paper. However, two points of reference will be made.

On the expressiveness side, the example Concurrent Prolog programs in (Edelman and Shapiro 84, Kusalik 84, Furukawa et al. 83, Hellerstein 84, Hellerstein and Shapiro 84, Hirakawa 83, Hirakawa et al. 83, 84, Shafrir and Shapiro 83, Shapiro 83a, 83b, 84, Shapiro and Takeuchi 83, Shapiro and Mierowsky 84, Takeuchi and Furukawa 83, Suzuki) and in this paper form an outstanding challenge for any concurrent programming language. These programs demonstrate several powerful programming techniques, including the recursive formation of complex communication structures; the use of incomplete messages; the design of fail-safe system hierarchies; and meta-programming, and also several more conventional ones, such as side-effect free treatment of I/O, monitors, interrupts and priorities, and bounded- and unbounded-buffer communication.

On the efficiency side, OCCAM (INMOS 84), a CSP-based language developed at INMOS, seems to have the most efficient implementation on uniprocessors to date among concurrent programming languages, and will probably be the first to have a high-performance multi-processor implementation, as soon as the Transputer is manufactured successfully. Concurrent Prolog, as well as the other languages mentioned, have yet to manifest their amenability to efficient parallel implementation.

These points are not sufficient to justify Concurrent Prolog being the language of our choice for systolic programming. The course of development, however, was the other way round. The systolic programming approach was conceived as a consequence of the experience accumulated in programming in Concurrent Prolog, and as a result of attempts to devise a suitable parallel architecture for it. Furthermore, the systolic programming style was not enforced on Concurrent Prolog, but rather was a natural development, almost a discovery.

After implementing Concurrent Prolog (Shapiro 83a) I started playing with the language. I could not help but notice that many straightforward, innocent-looking logic programs exhibit sytolic-like behavior once viewed through the execution model of Concurrent Prolog. Two phases were identified in the behavior of these programs: a "spawning"-phase, in which the program spawns a set of communicating processes, and a "systolic"-phase, in which these processes behave as a systolic system, overlapping computation and communication. The programs that exhibited this behavior were not composed in an attempt to implement systolic algorithms. Rather, they were the most naive and natural way to express a solution to a problem in Concurrent Prolog. Some of them where not even original Concurrent Prolog programs, but common Prolog programs that were transliterated into Concurrent Prolog by adding read-only annotations and the commit operator at the appropriate places. Examples are the insertion-sort and matrix-multiplication programs shown below.

Following these experiences, I have attempted to implement systolic algorithms explicitly. One of the more challenging systolic algorithms is the Band-matrix multiplication algorithm of Kung and Leiserson (Leiserson 83). To implement it in Concurrent Prolog this time-synchronous algorithm had to be converted into a data-flow synchronized one. The resulting program is 40 lines of code long (Shapiro 83b) and as far as I know it is one of the few working implementations of this algorithm.

These experiences suggests that there is some natural relationship between concurrent logic programming and systolic systems. That systolic systems provide a natural behavioral reading to many logic program, in addition to the familiar declarative and procedural (or problem-reduction) readings (Kowalski 74).

It is fair to note that even though all programs shown in this paper are written in Concurrent Prolog, it seems that they can be implemented in PARLOG (Clark and Gregory 84) as well without great difficulty.

5 THE MAPPING NOTATION

Languages like OCCAM, which specify process arrays, may benefit from a mapping notation that assigns processes to processors based on horizontal and vertical coordinates. I have found after trial and error that a recursive language such as Concurrent Prolog can make better use of a different mapping notation, namely LOGO-like Turtle programs (Pappert 80).

Each process, like a Turtle, has a position on the processing surface and a heading. A fixed-instruction Turtle program T can be associated with any process invocation P, as in P@T (read 'execute P at T'). The initial position and heading of the invoked process P is inherited from the invoker. Its new position and heading are determined by 'executing' the turtle program T.

A fixed-instruction Turtle program is a finite sequence of Turtle commands. An example of such a program is

(forward(1),right(90),forward(1),left(90))

which can be used by a recursive Concurrent Prolog program to define a walk along a grid's diagonal. The *forward* and *back* Turtle commands take a distance as an argument, and change the position of a process accordingly. The *left* and *right* commands take angles as arguments, and change the heading of a process.

In the following *forward* is a shorthand for *forward(1)*, and *right* is for *right(90)*.

The mapping notation can be added and debugged independently of the debugging of the main program. However, we found that graphical simulator of a processing surface, which shows the actions of the processes in each processor, is one of the better debugging tools for the program itself. Such a simulator, written in Prolog, was used to debug the programs in this paper. My experience using Turtle programs so far is that mapping simple structures — vectors, arrays, and H-trees — is rather straightforward. However, once the tool is there, the temptation to implement more and more sophisticated process structures and mappings is present. Debugging the Turtle programs that mapped the Aleph-trees and the Dynamic H-trees discussed below was not a trivial task.

Since the architecture is completely distributed, when a process is to be executed on a remote processor it must be sent there together with its associated program. This is not a major source of inefficiency as it seems on first thought. A heavily-used program, that dynamically spawns a process structure of a certain size for every input, can be converted into a program that spawns a similar process structure once, and then processes an entire stream of inputs.

6 ALGORITHM DESIGN AND PROGRAMMING METHODOLOGY

Given our observations on Concurrent Prolog and systolic systems the following approach to algorithm design may be concluded:

(1) Compose a pure logic program that defines the desired input/output relation.

(2) Make it a Concurrent Prolog program by adding the appropriate read-only annotations and commit operators.

(3) Observe the program's behavior: you've just discovered a new systolic algorithm!

Even though this methodology is not proposed completely seriously, it is not a complete joke either, as the novel systolic readings of the familiar logic programs below demonstrate.

On a more serious note, even for sequential, von Neumann computers, algorithm design is more an art then a science. Nevertheless, the basic tools for designing sequential algorithms, namely data-structures, are quite well understood. We have found that process-structures serve a similar role for soft-systolic algorithms. Many algorithms share the same process structures, and vice versa: when attempting the design of a new algorithm, a rich arsenal of process structures is an invaluable asset.

The relationship between sequential data-structures and parallel process-structures is even deeper. We have found that a sequential algorithm that operates on a certain data-structure often has a parallel counterpart that uses a similar process structure: a list of data is mapped into a list of processes; a tree of data is mapped into a process tree; and an array of data is mapped into a process array. This correspondence is manifest in all example programs below.

The algorithm development strategy implied by the systolic programming approach is similar to that of designing conventional systolic algorithms. A solution to a problem is defined in terms of a collection of processes that overlap computation with communication. The communication structure should be designed so it does not introduce bottlenecks, and can be mapped into the plane without much penalty. In contrast to hard-systolic algorithms, a detailed design and analysis of the timing of communication is unnecessary for obtaining a correct algorithm and can be deferred until fine-tuning for performance is necessary, since operations are synchronized via dataflow.

The claim for the greater freedom of soft-systolic over hard-systolic algorithms can be substantiated by two types of evidence. One is showing that known hard-systolic algorithms are easier to specify and implement when viewed as soft-systolic ones. This should be expected, since it is easier to simulate a synchronous system with a dataflow synchronized system then vice versa. Examples include the Concurrent Prolog implementation of a systolic algorithm for Gaussian elimination, shown below, and the algorithms implemented in (Shapiro 83b, Shafrir and Shapiro 83, Hellerstein and Shapiro 84).

Another type of evidence is novel algorithms which are easier to conceive or implement under the soft-systolic approach. Examples are also starting to accumulate (Edelman and Shapiro 84, Hellerstein 84, Shapiro 83b, Shapiro and Mierowsky 84).

7 PROGRAMMING EXAMPLES

The example programs below demonstrate the specification and mapping of various process structures using Concurrent Prolog as the programming language and Turtle programs as the mapping notation.

The first example is the sieve of Eratosthenes.

```
primes(Ps) :-
    integers(2,Is), sift(Is?,Ps)@forward.

integers(N,[N|Is?]) :-
    N1:=N+1, integers(N1,Is).

sift([Prime|In],[Prime|Out1?]) :-
    filter(In?,Prime,Out),
    sift(Out?,Out1)@forward.

filter([N|In],Prime,Out) :-
    0=:=N mod Prime | filter(In?,Prime,Out).
filter([N|In],Prime,[N| Out?]) :-
    0=\=N mod Prime | filter(In?,Prime,Out).
```

Program 1: Sieve of Eratosthenes

Variants of this program are abundant in the logic-programming folklore. Logically, it defines the relation {*primes(Ps): Ps* is the (infinite) list of primes.} Behaviorally, it defines a soft-systolic algorithm. Using an auxiliary *sift* process, it spawns a dynamically growing set of linearly-connected *filter* processes, one for each prime number found, as shown in Figure 1.

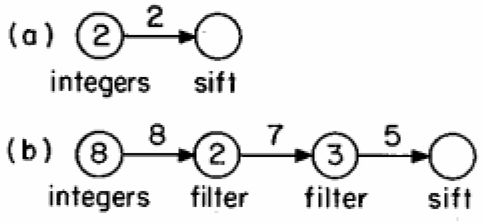

Figure 1: Spawning a pipe of *filter* processes

A *filter* process copies its input stream to its output stream, filtering out multiples of its local prime number. The *sift* process also collects the prime numbers in its output stream.

Program 2 is one of the simpler ways to define the *sort* relation. Its corresponding Prolog program behaves like the ordinary recursive insertion-sort algorithm. The Concurrent Prolog program behaves as follows: it first spawns a linearly connected sequence of *insert* processes, one for every element in its input stream, as shown in Figure 2. The last *insert* process is spawned with the empty input stream, due to the clause *sort([ ],[ ])*. At this time the pipe starts propagating data backwards: starting from the last process, each process copies its ordered (or empty, in case of the last process) input stream to its output stream, inserting its local number in the output stream so it remains ordered.

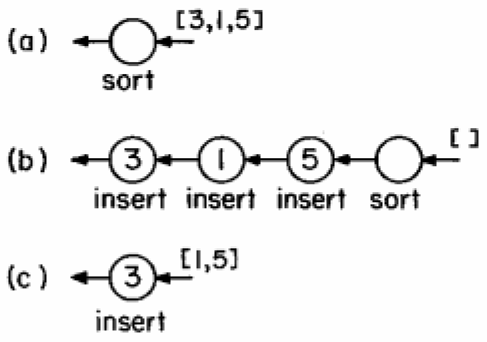

Figure 2: Insertion sort

```
sort([X|Xs],Ys) :-
    insert(X,Zs?,Ys), sort(Xs?,Zs)@forward.
sort([ ],[ ]).

insert(X,[Y|Ys],[Y|Zs?]) :-
    X>Y | insert(X,Ys?,Zs).
insert(X,[Y|Ys],[X,Y|Ys]) :-
    X=<Y | true.
insert(X,[ ],[X]).
```

Program 2: Insertion Sort

Under reasonable assumptions, this program runs in linear time using a linear number of processors. The Concurrent Prolog implementation of merge-sort (Shapiro 83b), whose code is about twice as long, runs in linear time using only a logarithmic number of processors.

Our third example is matrix multiplication. A simple systolic algorithm for matrix multiplication is shown in Figure 3. Two matrices are fed into an array of processors. Each processor in the array computes the internal product of the two vectors that pass through it. At the end of the computation, each processor contains one element of the resulting matrix.

An almost identical behavior is achieved by the following logic program *mm*, which defines the relation {*mm(X,Y,Z) :- Z* is the result of multiplying the matrix *X* with the transposed matrix *Y*}. When called with two matrices, *mm* spawns an array of *ip* processes, each computing the internal-product of two vectors (represented as lists or streams).

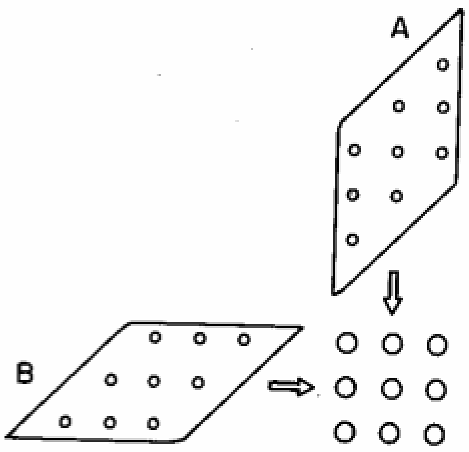

Figure 3 : A systolic algorithm for matrix multiplication

```
mm([ ],_,[ ]).
mm([X|Xs],Ys,[Z|Zs]) :-
     vm(X,Ys?,Z)@right,
     mm(Xs?,Ys,Zs)@forward.

vm(_,[ ],[ ]).
vm(Xs,[Y|Ys],[Z|Zs]) :-
     ip(Xs?,Y?,Z), vm(Xs,Ys?,Zs)@forward.

ip([X|Xs],[Y|Ys],Z) :-
     Z:=(X * Y)+Z1, ip(Xs?,Ys?,Z1).
ip([ ],[ ],0).
```

Program 3: Matrix Multiplication

The Turtle programs of *mm* spawn a vector of *vm* processes, whose heading is orthogonal to the spawning direction. Each *vm* process, in turn, spawns a vector of *ip* processes. A snapshot of the spawning phase is shown in Figure 4. The final result is a rectangular array of *ip* processes, one for each inner-product that needs to be computed.

The behavior of this program is different from the systolic algorithm in Figure 3, in that the vectors are not pipelined between processes, but are shipped independently to each process. In (Shapiro 83b) Program 3 is transformed to pipeline the vectors between processors. This is achieved by adding two output streams to each *ip* process, one for copying each input stream, and connecting the *ip* processes accordingly.

Devising an algorithm that copies bindings from one processor to another in a way that utilizes this pipelining is still a research question. Given such an algorithm, the modified program can multiply two $n \times n$ in time linear in n, using n^2 processors.

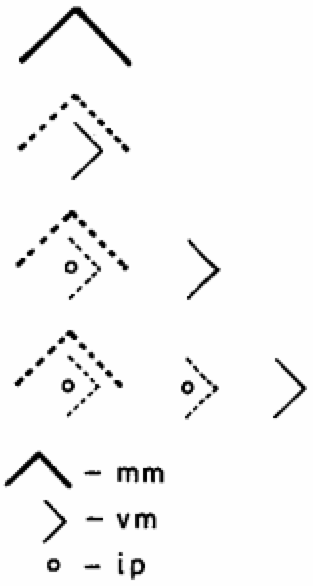

Figure 4 : A snapshot of spawning a process array

The fourth example shows how to spawn the process structure of a divide-and-conquer program using H-trees (Leiserson 83). It computes the relation {*hanoi(N,A,B,X)*:- the sequence of moves *X* can be used to move *N* disks from peg *A* to peg *B*, such that no disk is ever placed on a smaller one}. It is adapted from a similar Prolog program by H. Yasukawa. The notation *p(N++,...) :- ...* is a shorthand for *p(N1,...) :- N1>0 | N:=N1-1,...*.

```
hanoi(0,A,B,(A,B)).
hanoi(N++,A,B,(Before?,(A,B),
  After?)):-
  free(A,B,C),
  hanoi(N,A,C?,Before)@(left,forward(2↑(N/2))),
  hanoi(N,C?,B,After)@(right,forward(2↑(N/2))).

free(a,b,c). free(a,c,b). free(b,a,c).
free(b,c,a). free(c,a,b). free(c,b,a).
```

Program 4: The Towers of Hanoi

Program 4 spawns a tree of *free* processes, one for each step in the solution. The computation of these processes is trivial: given two pegs, they compute the remaining peg. If the overhead associated with spawning a remote process is very high, then spawning only a partial H-tree may be a better solution. Spawning a static H-tree to solve several problems, a possibility mentioned earlier, is not applicable in this case: who would want to solve the same Towers of Hanoi problem more than once (or even once)?

The mapping strategy used by Program 4 requires to know the depth of the computation tree in advance, which may be a limitation for

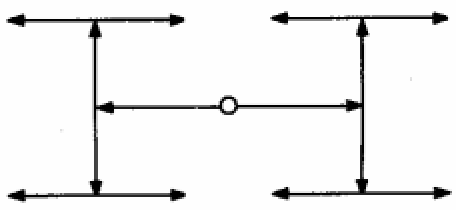
Figure 5: Spawning an H-tree

some divide-and-conquer algorithms. A method for spawning computation trees on a processing surface which does not suffer from this problem is described in (Martin 79, Sequin 81). An implementation of another solution, termed dynamic H-trees, is shown in (Shapiro 83b).

Another problem with H-trees is that they are not evenly spread on the processing surface, therefore may utilize it ineffectively. In the portion of the processing surface covered by an H-tree, roughly one third of the processors contain leaf processes, one third contain non-leaf processes, and one third are either idle or serve only as communication-relays. This may cause load-imbalance if the tree is fairly static and performs heavy computations, e.g. a parallel search tree. An alternative to H-trees, called Aleph-trees, evenly maps complete binary trees of even depth on a processing surface by allocating one leaf-process and one internal-process to each processor (except one). Due to space limitations it will be reported upon in a subsequent paper.

```
gauss([[A0|As]|Columns],[B0|Bs], [X|Xs]) :-
    pivot(A0,B0,Bs?,Bs1,As?,Factors),
    X:=(B0-Sum)/A0,
    row(Factors?,Columns?,Columns1,Xs,Sum)
            @(right,forward),
    gauss(Columns1?,Bs1?,Xs)@(fd,rt,fd,lt).
gauss([ ],[ ],[ ]).
pivot(A0,B0,[B|Bs],[B1|Bs1],[A|As],[Factor|Fs]) :-
    Factor:=A/A0,
    B1:=B-(B0*Factor),
    pivot(A0,B0,Bs?,Bs1,As?,Fs).
pivot(A0,B0,[ ],[ ],[ ],[ ]).
row(Factors,[[A|As]|Columns],
    [As1|Columns1],[X|Xs],Sum1) :-
    cell(Factors,A,As?,As1),
    Sum1:=Sum+(A*X),
    row(Factors,Columns?,Columns1,Xs,Sum?)
            @forward.
row(Factors,[ ],[ ],[ ],0).
cell([Factor|Fs],A0,[A|As],[A1|As1]) :-
    A1:=A-(A0*Factor),
    cell(Fs?,A0,As?,As1).
cell([ ],A,[ ],[ ]).
```
Program 5: Gaussian Elimination

Program 5 is a Concurrent Prolog implementation of a systolic algorithm for Gaussian elimination. It demonstrates the ability of Concurrent Prolog to form complex communication structures, and also the need for a graphical notation that supports the construction and understanding of systolic algorithms.

Program
5 defines the relation {*gauss(A,B,X)*:- *X* is the list of solutions to the linear equation defined by the coefficient matrix *A* and vector of values *B*}. It spawns a lower-triangular array of processes that has *pivot* processes on the diagonal and *cell* processes below it. The recursive construction of the process array is shown in Figure 6. It is a graphical representation of the first clause of Program 5.

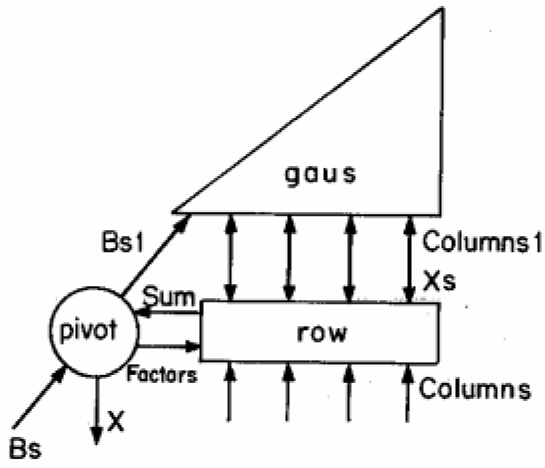

Figure 6 : Recursive construction of the Gauss process array

The computation has three phases: one spawning-phase and two systolic phases — elimination and back-substitution. The execution of the spawning phase and the elimination phase overlap. The general information flow in the two systolic phases is shown in Figure 7.

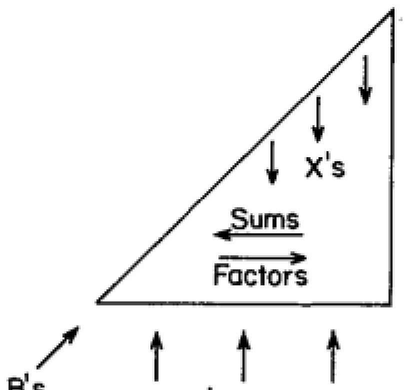

Figure 7 : Information Flow in Gaussian Elimination

The program spawns the n^{th} row with the coefficients of the n-th equation, in which the first $n-1$ variables have been eliminated. The n^{th} row eliminates the coefficients of the n^{th} variable from all subsequent equations in the elimination phase, and computes the value of that variable in the back-substitution phase.

A row is composed of a *pivot* process and a *cell* process. The behavior of the *pivot* process is shown in Figure 8.

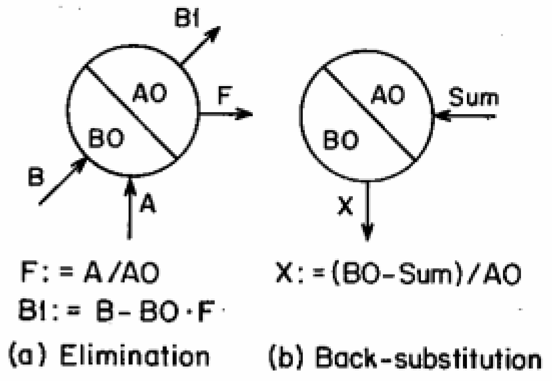

Figure 8: Behavior of a *pivot* process

To eliminate the coefficient A of the first remaining variable in an equation, the pivot computes the factor $F:=A/A0$, where $A0$ is the coefficient of the pivot variable. The factor F is broadcasted to the rest of the row, which subtracts itself, multiplied by the factor, from the equation, and forwards the modified equation to the next row. The pivot also modifies the B value of the equation accordingly. During back-substitution, the pivot receives *Sum* from the row, and computes the value of X.

The behavior of a *cell* process is shown in Figure 9.

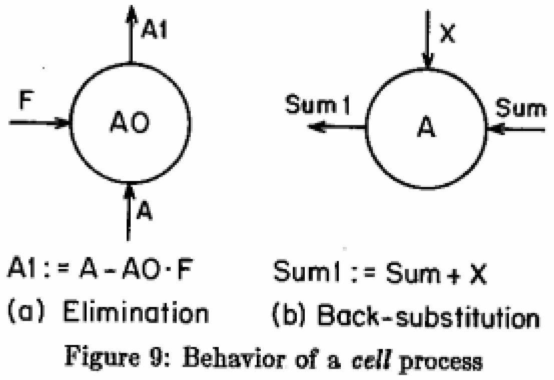

Figure 9: Behavior of a *cell* process

When receiving a factor F, each cell process computes $A1:=A-(F*A0)$, where A is the coefficient received from south and $A0$ is the local coefficient, and sends the result north. Note that the processes that compute the back-substitution, $X:=(B-Sum)/A$, and $Sum1:=Sum+(A*X)$, are created during the spawning phase, and are suspended on their *Sum* variables until the elimination phase is complete.

Program 5 can solve n equations in n variables in time linear in n on a processing surface whose size is linear in n^2. It can be extended to handle zero-pivots.

8 COMPARISON WITH OTHER WORK

Systolic programming shares the belief that architectures based on global communication, such as NYU's Ultracomputer (Gottlieb et al. 83), TRAC (Sejnowiski et al. 80), Alice (Darlington and Reeve 81), and others, are not the ultimate approach to parallel processing. Such architectures are reactionary, since they capitalize on our ignorance of parallel computations. Their attractiveness will decrease steadily as our understanding of parallelism increases and we know more about designing and programming systolic and other local-communication based algorithms.

Several similar architectures based on local interconnections have been proposed, including the Apiary (Hewitt 73), the CHiP computer (Snyder 83), FAIM (Davis 84), the Bagel (Shapiro 83b), and Martin's original Processing Surface (Martin 79).

They differ mainly in their programming methods. The CHiP Computer relies on a mapping-compiler (Bokhari 81) to allocate statically processes to processors, and to compile the amalgamated programs of processes mapped to the same processor into code that timeshares between them (Berman and Snyder 84). This approach is feasible for programming an attached processor, but not for a general-purpose computer. A similar direction was pursued with the Cm*, (Schwans 82).

On the other extreme, the Apiary allows the dynamic creations of processes ("objects"), and relies on algorithms (or heuristics?) for runtime migration of processes to ensure load-balancing and locality of communication (Hewitt and Lieberman 84). Like the global-communication approach, the Apiary accepts ignorance of the behavior of computations as a basic assumption. To date, all progress in science resulted from assuming, in spite of superficial evidence to the contrary, that there is some order in the world, and that science's goal is to uncover it. Resigning before the battle has begun by declaring the world to be unpredictable would be a self-fulfilling prophecy, but would not result in scientific progress.

The systolic programming approach advocated in this paper is in some sense a mixture of the two, and in another sense more conservative then both. It allows the dynamic creation of processes, but relies on the program to map itself on the processing surface. Although it is conservative it is also future-proof. Should a successful mapping-compiler be developed, it can be incorporated as a preprocessor to mapping-less programs. Should some techniques for runtime load-balancing be found useful for certain applications, they can be implemented by adding a layer of interpretation, which may be compiled away for specific programs using techniques of partial evaluation. Needless to say, a statically-allocated distributed meta-interpreter is the best tool for experimenting with load-balancing algorithms.

9 CONCLUSIONS

Many discussions on parallel processing go in circles: "to exploit parallellism fully we must spread the computation as much as possible", "but if we spread it too much we will generate an unmanageable amount of communication". The systolic approach is the only one to date that show how to cut this Gordian knot — to allow massive parallelism without communication bottlenecks.

So far the systolic approach was viewed as having only limited applications. We hope that systolic programming will help to carry its insight into the realm of general purpose parallel computing.

ACKNOWLEDGEMENTS

Part of this research was carried while visiting ICOT, and I am grateful for this stimulating research environment.

The ideas in this paper were developed in collaboration with many. Walter Wilson suggested that one should not worry about small multiprocessors, but rather design an infinite multiprocessor first. Marc Snir pointed out the importance of a mapping notation. Ken Kahn suggested the use of fixed-instruction Turtle programs, instead of a weaker mapping notation I have been using. Amos Fiat and Leon Sterling helped with the design of the Gaussian elimination algorithm. Leon also gave useful comments on the paper.

I would also like to thank Sarah Fliegelmann, who was successful in shielding the mysteries of TEX from me for so long.

REFERENCES

Ackerman, W.B. Data flow languages, IEEE *Computer* **15(2)**, 15-25, 1982.

Berman, F. and Snyder, L. *On mapping parallel algorithms into parallel architectures.* International Conference on Parallel Processing, 1984.

Bokhari, S.H. On the Mapping Problem. IEEE *Transactions on Computers* **C-30(3)**, 207-214, 1981.

Clark, K.L. and Gregory, S. A Relational Language for Parallel Programming. In *Proceedings of the ACM Conference on Functional Languages and Computer Architecture*, October, 1981.

Clark, K.L. and Gregory, S. *PARLOG: Paralle Programming in Logic.* Research Report DOC 84/4, Department of Computing, Imperial College of Science and Technology, April 1984.

Darlington, J. and Reeve, M. Alice: a multiprocessor reduction machine for the parallel evaluation of applicative languages. *Proceedings of the ACM Conference on Compiler Construction*, pp.65-75, July 1981.

Davis, A. *FAIM: Fairchild's AI Machine.* Lecture at the Workshop on Implementation of Concurrent Prolog, The Weizmann Institute of Science, April 1984.

Edelman, S. and Shapiro, E. *Quadtrees in Concurrent Prolog.* Weizmann Institute Technical Report CS84-19, August 1984.

Fiat, A., Shamir, A. and Shapiro, E. *Polymorphic Arrays: An architecture for a Programmable Systolic Machine.* Weizmann Institute Technical Report CS84-20, 1984.

Fisher, A.L., Kung, H.T., Monier, L. and Dohi, Y. Architecture of the PSC: A Programmable Systolic Chip. In *Proceedings of the 10th Annua International Symposium on Computer Architecture*, pp. 48-58, IEEE Computer Society Press, 1983.

Friedman, D.P. and Wise, D.S. Aspects of applicative programming for parallel processing, *IEEE Transactions on Computers* **C-27(4)**, 289-296, 1978.

Furukawa, K., Takeuchi, A. and Kunifuji, S. *Mandala: A Concurrent Prolog Based Knowledge Programming Language/System.* ICOT Technical Report TR-029, 1983.

Gottlieb, A., Grishman, R., Kruskal, C.P., McAuliffe, K.P., Rudolf, L. and Snir, M. The NYU Ultracomputer – Designing an MIMD Shared Memory Parallel Computer, *IEEE Transactions on Computers* **C-32(2)**, 175-190, 1983.

Hellerstein, L. *A Concurrent Prolog Based Region Finding Algorithm.* Honors Thesis, Computer Science Department, Harvard University, May 1984.

Hellerstein, L. and Shapiro, E. Algorithmic Programming in Concurrent Prolog: The MAXFLOW Experience. *Proceedings of the 1984 International Symposium on Logic Programming*, pp.99-118, IEEE, February 1984.

Henderson, P. Purely Functional Operating Systems. In *Functional Programming and its Applications*, P. Henderson and D.A. Turner (eds.), Cambridge University Press, 1982.

Hewitt, C. A universal modular Actor formalism for artificial intelligence. In *Proceedings of the Third International Joint Conference on Artificial Intelligence*, IJCAI, 1973.

Hewitt, C. The Apiary Network Architecture for Knowledgeable Systems. *Proceedings of the Lisp Conference*, Stanford, CA, 1980.

Hewitt, C. and Lieberman, H. Design Issues in Parallel Architectures for Artificial Intelligence. *Proceedings of COMPCON 84*, pp.418-422, IEEE, 1984

Hirakawa, H. *Chart Parsing in Concurrent Prolog.* ICOT Technical Report TR-008, 1983.

Hirakawa, H. et al. *Implementing an Or-Paralle Optimizing Prolog System (POPS) in Concurrent Prolog.* ICOT Technical Report TR-020, 1983.

Hirakawa, H., Chikayama, T. and Furukawa, K. Eager and Lazy Enumerations in Concurrent Prolog. *Proceedings of the Second internationa Logic Programming Conference*, Stan-Ake Tarnlund (ed.), pp.89-101, Uppsala 1984.

Hoare, C.A.R. Communicating Sequential Processes. *Communications of the ACM* **21(8)**, 666-677, 1978.

INMOS Limited. *IMS T424 Transputer Advance Information.* INMOS, 1983.

INMOS Limited. *OCCAM Programming Manual.* Prentice Hall International Series in Computer Science, 1984.

Kowalski, R.A. Predicate logic as a programming language. In *Information Processing* **74**, 569-574. North-Holland, Amsterdam, 1974.

Kung, H.T. Let's Design Algorithms for VLSI Systems. *Proceedings of the Conference on Ver Large Scale Integration: Architecture, Design, Fabrication*, California Institute of Technology, pp. 65-90, January 1979.

Kung, H.T. Why systolic architectures? IEEE *Computer* **15(1)**, 37-46, 1982.

Kusalik, A.J. Bounded-Wait Merge in Shapiro's Concurrent Prolog. *Journal of New Generation Computing* **2(2)**, 1984.

Leiserson, C.E. *Area-Efficient VLSI Computation.* MIT Press, 1983.

Martin, A.J. The Torus: An Exercise in Constructing a Processing Surface. *Proceedings Conference on Very Large Scale Integration: Architecture, Design, Fabrication*, California Institute of Technology, pp. 52-57, January 1979.

Pappert, S. *Mindstrorms: Children, Computers, and Powerful Ideas.* Basic Books, New York, 1980.

Schwans, K. *Tailoring Software for Multiple Processor Systems.* Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University.

Sejnowiski, M.C., Upchurch, E.T., Kapur, R.N., Charlu, D.P.S. and Lipovski, G. An overview of the Texas Reconfigurable Array Computer. *Proceedings of the AFIPS Conference*, pp.631-641, 1980.

Sequin, C.H. Doubly twisted torus networks for VLSI processor arrays. *Proceedings of the Eighth International Conference on Computer Architecture*, IEEE, pp. 471-480, 1981.

Shafrir, A. and Shapiro, E. *Distributed Programming in Concurrent Prolog.* Weizmann Institute Technical Report CS83-12, August 1983.

Shapiro, E. *A Subset of Concurrent Prolog an Its Interpreter.* ICOT Technical Report TR-003, February 1983. Also Weizmann Institute Technical Report CS83-12, August 1983a.

Shapiro, E. *Lecture Notes on the Bagel: A Systolic Concurrent Prolog Machine.* ICOT TM-0031, November, 1983b.

Shapiro, E. *Systems programming in Concurrent Prolog.* Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages, pp.93-105, ACM, January 1984.

Shapiro, E. and Mierowsky, C. Fair, Biased, and Self-Balancing Merge Operators: Their Specification and Implementation in Concurrent Prolog. *Proceedings of the 1984 International Symposium on Logic Programming*, pp.83-91, IEEE, February 1984.

Shapiro, E. and Takeuchi, A. Object-Oriented Programming in Concurrent Prolog. *Journal o New Generation Computing* **1(1)**, July 1983.

Snyder, L. Introduction to the Poker Programming Environment. *Proceedings of the 1983 International Conference on Parallel Processing.*

Suzuki, N. Experience with Specification and Verification of Complex Computer Hardware Using Concurrent Prolog. To appear in: *Logic Programming and its Applications*, D.H.D Warren and M. van Caneghem (Eds.).

Takeuchi, A. and Furukawa, K. Interprocess Communication in Concurrent Prolog. *Proceedings of the Logic Programming Workshop 83*, pp. 171-185, Albufeira, Portugal, June 1983. Also ICOT TR-006.

Weiser, U. and Davis, A. A Wavefront Notation Tool for VLSI Array Design. In *VLSI Systems and Computations*, H.T. Kung, R.F. Sproull and G. L. Steele, Jr. (eds.), Carnegie-Mellon University, Computer Science Press, pp. 226-364, 1981.