BLOB Computing

Frédéric Gruau [†] Yves Lhuillier [†] Philippe Reitz [‡] Olivier Temam [†]

[†] LRI, Paris South University & INRIA Futurs France
{gruau,lhuillie,temam}@lri.fr

[‡] LIRMM, Montpellier University France
reitz@lirmm.fr

ABSTRACT

Current processor and multiprocessor architectures are almost all based on the Von Neumann paradigm. Based on this paradigm, one can build a general-purpose computer using very few transistors, e.g., 2250 transistors in the first Intel 4004 microprocessor. In other terms, the notion that on-chip space is a scarce resource is at the root of this paradigm which trades on-chip space for program execution time. Today, technology considerably relaxed this space constraint. Still, few research works question this paradigm as the most adequate basis for high-performance computers, even though the paradigm was *not* initially designed to scale with technology and space.

In this article, we propose a different computing model, defining both an architecture and a language, that is intrinsically designed to exploit *space*; we then investigate the implementation issues of a computer based on this model, and we provide simulation results for small programs and a simplified architecture as a first proof of concept. Through this model, we also want to outline that revisiting some of the principles of today's computing paradigm has the potential of overcoming major limitations of current architectures.

Categories and Subject Descriptors

C.1.4 [**Processors Architecture**]: Parallel Architectures—*Distributed architectures*; D.3.2 [**Programming Languages**]: Language Classifications—*Concurrent, distributed, and parallel languages*

General Terms

Design, Languages, Performance

Keywords

Scalable Architectures, Cellular Automata, Bio-inspiration

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CF'04, April 14–16, 2004, Ischia, Italy.
Copyright 2004 ACM 1-58113-741-9/04/0004 ...$5.00.

1. INTRODUCTION

The first processors in the 1950s and later on the first microprocessors in the 1970s, like the Intel 4004, were based on the so-called Von Neumann model: a central processing unit with a memory and a set of instructions to guide machine execution. This model was extremely popular and successful because it combined *universality* and *compactness* : the ability to build a general-purpose computing machine capable of executing a large range of programs in a very restricted space (2250 transistors in the Intel 4004 first microprocessor). Compactness means that the Von Neumann architectures are implicitly based on the idea that space is a scarce resource and thus, *they trade space for time*: complex tasks are broken down into a sequence of simple instructions/operations that execute one by one on a single general-purpose central processing unit (ALU), and intermediate results are stored in a memory. Even though technology considerably relaxed this space constraint, today's processors still largely follow the same principles.

It is almost paradoxical that, in the past 30 years, almost all attempts at exploiting the additional *space* brought by technology relied on the "space-constrained" paradigm: (1) parallel computers, either on-chip or true multi-processors, almost all rely on a set of independent such processors, (2) today's high-performance general-purpose processors attempt to squeeze performance out of the original model by altering it in every possible way (pipeline, cache hierarchy, branch prediction, superscalar execution, trace cache and so on). As a result, current high-performance computers are plagued by intrinsic flaws. In parallel computers, it is difficult to efficiently manage communications between many individual processors which must constantly exchange data over complex interconnection networks. The efficiency of high-performance processors decreases as their complexity increases [13]. And as the number of processors increases or the processor complexity increases, it is excessively difficult to write or generate programs that efficiently exploit these architectures [44].

Few research works in computer architecture question the cpu-memory model as the most adequate basis for high-performance computers, even though this model was *not* originally designed to scale with technology and space, since it was proposed before the advent of integrated circuits. The purpose of this article is to introduce a novel computing model, combining both architecture and language, that is intrinsically designed to exploit *space*; we then investigate the implementation issues of a computer based on this model, and we provide some execution simulation results as a first proof of concept. Through this model, we also want to outline that revisiting some of the principles of today's computing paradigm has the potential of overcoming major limitations of current architectures.

The principles of this model are the following. Space is exploited by merging and distributing computations and memory over a large number of automata; the machine implementations of these automata are called *Blobs* because they vary in shape and size within the resource space, and they act both as processing and memory elements. The architecture is regular and homogeneous (and thus easily scalable) and consists in a very large set of hardware elements, called *cells*, each Blob using several cells. There is no central control unit anymore: control is distributed over the whole architecture and takes the form of simple rules that manage Blobs movements over hardware cells by mimicking physical laws like gas pressure. As a result, the *placement* of data and resources is dynamically and automatically managed, and we have formally proved that we achieve optimal VLSI complexity in certain cases [31]. Finally, because cells only interact with their neighbors, there is no communication network per say.

This computing model is not solely an architecture model, it defines both the architecture and the language. In current computing systems, architecture design and compiler design are usually considered separate tasks. As a result, computer architects design architectures which are difficult to compile for, and compilers insufficiently take into account architecture complexity. Still, the major limitation of current compilers lay even more in languages than in compilers themselves. Current programming languages are not architecture-friendly: from the task imagined by the programmer to the program itself, lots of information (semantics), e.g., parallelism and memory usage, are lost because the language is not designed to retain them; and in many cases, such information are critical for an efficient exploitation of the target architecture. To a large extent, the compiler role consists in reverse-engineering the program in order to dig up the lost information.

The *Blob Computing* model comes with a language that is both architecture-independent and architecture-friendly: it provides minimal but essential information on the program to facilitate the task of the architecture but it makes almost no assumption on the architecture itself. The program specifies when and how Blobs are created, and as a result, it implicitly tells the architecture how to exploit space and parallelism. We have already developed a compiler for translating programs based on an imperative language (Pascal) into *Cellular Code* [30], on which the Blob language is based, without significantly increasing the program size, thereby demonstrating the expressiveness of the programming language; the compiler and the associated translation techniques have been patented [25].

Section 2 presents the related work. Note that, because we propose to modify many aspects of the current computing model, we thought it was necessary to position our model with respect to the traditional computer approaches and their limitations, the alternative computing models as well as the emerging technologies, hence the long related work section. In Section 3, we present the Blob computing model, how to program a Blob machine and we investigate the implementation issues of a Blob computer. Finally, in Section 4, we show that some classic examples can be efficiently implemented on a Blob computer and we present some simulation results.

2. RELATED WORK

Moore's law and the exploitation of parallelism have fueled considerable research on improving the original cpu-memory model. At the same time, researchers in academia keep exploring novel computing models either to overcome the limitations of the cpu-memory model or to exploit future and emerging technologies [1, 18]. The first category of research works can be defined as "bottom-up": they attempt to incrementally transform the original paradigm into a paradigm that can better scale with technology and particularly increasing space. The second category can be defined as "top-down": they attempt to elaborate a computing model with the desired properties and, in some cases, they later derive a real machine out of it. While we are closer to the second category, the present study is a joint work by researchers from both categories, and thus, we are focused on defining a model that can effectively become a real machine in the long term. After a first study on the proposed model properties [28, 31], this study is the second step in that direction. A third study related to implementation issues of the proposed model has already been completed [29].

In this section, we outline the current trends in traditional computers, then we highlight the main alternative models, and finally we explain how Blob computing differs from both the traditional and alternative approaches.

2.1 Trends in traditional computers

Processors bet on speed rather than space. In the past 20 years, processor architecture has largely relied on speed to improve performance. Space is essentially spent on implementing the hardware components required for coping with increasing speed (caches, branch prediction tables,...). One of the major motivations of RISC processors was building simple architectures that can be quickly retargeted to the newest technology process, through the use of pipelines to achieve low cycle time and high instruction throughput. Superscalar processors have followed this trend and pushed it to its limits: the Pentium IV has more than 20 pipeline stages [35], and this number is poised to increase, calling for sophisticated instruction fetch techniques [10, 47]; memory latency is now so high that a significant share of the on-chip space is devoted to cache hierarchies as in the Intel Itanium [51]; even communication delays within a chip cannot be hidden within a clock cycle, as shows the *drive* pipeline stages in the Pentium IV [35] or multi-cluster VLIWs [55]. Multi-cluster architectures are also a testimony to the difficulty of scaling up a centralized computing resource. If on-chip space were essentially devoted to increasing computing power, the number of bits processed per time unit would be at least an order of magnitude larger than it actually is [17].

Multiprocessors exploit space but most attempt to hide it. The very idea of multiprocessors consists in using a large number of computing resources, i.e., to trade space/cost for performance. However, for a long time, multiprocessors have also pursued the impossible goal of hiding space to the user, i.e., the distance between computing nodes and the associated long communication latencies: shared-memory multiprocessors attempt to mimic the original cpu-memory model but they do not scale well with the number of processors [42], virtual shared-memory was investigated for distributed-memory multiprocessors in an attempt to hide the memory distribution across space (computing nodes). Not surprisingly, grid computing [40] is popular now because it focuses on applications that need no communication between computing nodes and can thus realize this goal of a large number of parallel computing resources without intra-computation communication issues. Also, SIMD architectures are popular, especially in the embedded domain, for a restricted class of applications with very regular communication and computing patterns, such as network processing [23].

Space is used to build complex architectures, and it is difficult to write programs with high sustained performance for complex architectures. The difficulty to parallelize programs so that computations hide long communication delays is probably one of the major causes for lesser popularity of the massively parallel

computing paradigm in the 1990s. Mainly, properly distributing data and computations across space to minimize communications as the number of computing resources increases (i.e., as space increases) proved a difficult task. Moreover, many algorithms are thought sequentially, written using languages that are not designed to retain information on parallelism, and as a result, digging up parallelism properties using optimizing compilers can be fairly difficult [6], and even more so if programs use irregular communication and computing patterns [11].

As processor complexity increases, the optimization issues for processor architectures are becoming similarly difficult. It took more than 10 years and significant research efforts to embed in some compilers a reasonably accurate cache model that considers both capacity and conflict misses for loop nests with linear array references only [22]. As more components with complex run-time behavior are incorporated in the architecture (branch predictors, Trace cache,…), researchers need to investigate novel and more complex optimization techniques, such as run-time iterative compilation, to harness the additional complexity [7].

Acknowledging space in architectures. It is interesting to note that, in both the processor and multiprocessor domains, a number of important and recent research propositions seem to converge toward fairly regular grid-like architectures. The most notable examples are the TRIPS processor [48] and the IBM Blue Gene/L multiprocessor [39]. In both machines, the core units are simple processing units (respectively functional units and simple processors), and they are linked through a 2-D grid-like network that favors communications with neighbors, i.e., computing nodes that are close to each other in space. Using regular 2-D grids *acknowledges* that the computer is built in a 2-D space, and it simply distributes processing power and memory (for Blue Gene/L only) across space. Similarly, using a network that implements the natural neighbor-to-neighbor communications *acknowledges* that communicating with distant nodes takes more time, i.e., it is not hidden within a complex network that pretends to provide transparent connections between all nodes. Apparently, IBM proposed a similar *cellular architecture* for the embedded engine of the future PlayStation by Sony [4].

Still, if both propositions seem a more natural fit to the notion of space than many previous and more complex architectures, they focus on architecture issues, and do not address the associated programming issues.

Also, it is important to note that many original solutions are emerging in the FPGA domain where exploiting space comes naturally [16]: mapping parts or whole programs onto logic blocks to exploit circuit-level parallelism [64, 8], dynamically adjusting circuit space to program requirements [14], and so on. Still many of these solutions are not designed for general-purpose computing but rather for specific application domains, especially stream-based applications.

2.2 Trends in alternative computing models

The realization that many of the current architecture limitations are rooted in the computing paradigm itself rather than current architecture designs has stemmed an increasing amount of research works on novel computing paradigms.

Most new computing models aim at providing a way to describe and exploit parallelism. The two main factors that distinguish them are (1) *programmability* : the ability of the language to describe large classes of algorithms, while at the same time remaining sufficiently architecturally independent (i.e., the abstraction potential of the execution model), and (2) *space exploitation and scalability* : how the different computing models exploit and scale up with available resources.

Other studies on alternative computing models do not stem from the current computing paradigm but they aim at finding ways to exploit novel technologies, and in certain cases, it also means defining space-scalable paradigms.

Programmability The goal is to design high-level programming languages where the programmer can easily and naturally translate an algorithm into a program, while, at the same time, providing rich information for the target architecture. Since most performance improvements are achieved through the exploitation of parallelism and an efficient management of program data, programming languages designers generally focus on passing information on parallelism and data usage [19, 9, 3].

Traditional imperative programming languages like C or Java are popular because they enable a natural and almost direct translation from the algorithm/task to the program. However, by expressing an algorithm as a sequence of steps, such programming languages introduce a "sequential bias" where some of the parallel properties of the algorithm are lost.

Declarative or synchronous languages such as LUSTRE [32], Esterel [5], Signal [21] or StreamIt [56] address the parallelism expression problem: they explicit data consumption and production and thus implicitly specify parallelism. Yet, these declarative computing models essentially target DSP or ASIC architectures so that their programmability is restricted to regular algorithms. They lack features often used in general-purpose programs, such as dynamic data structures.

Systolic machines or cellular automata [58, 37, 50] represent other computing models where the programmer explicitly describes the spatial configuration of the substrate and communications. These models are suited to fairly regular and massively parallel applications where both parallelism and communications can be easily described. They are less suitable for more complex and irregular applications.

Dataflow computing models [43] address both the parallelism expression and programmability issues. Pure functional languages [19] and more generally, term rewritings systems [36, 3] offer a simple parallel programming model well suited to the dataflow computing model. Term-rewriting languages such as Gamma [3], combine both the implicit parallelism of declarative languages and natural programming features such as control flow constructs and dynamic data structures. The principle is to represent data as a set of t-uples and the program as a set of rules that apply on certain combinations of t-uples, i.e., whenever the combination of t-uples associated with a rule is present, the rule applies and outputs one or several other t-uples. Intuitively, t-uples are a bit like the objects and rules like the methods of object-oriented programming languages, except that the application of rules is only determined by the presence of the input t-uples in a dataflow way, while it is rigidly driven by the program in imperative object-oriented languages like C++ or Java. Still, complex term-rewriting programs have two drawbacks: managing a large unstructured set of rules is difficult for a programmer, and the more t-uples and rules the longer the task of checking which rules apply.

Consequently, Fradet et al. [20] have proposed to encapsulate t-uples and rules within subsets. This type of encapsulation was introduced by Paun with *membranes* in [45]. The program is now a set of interconnected subsets which can pass t-uples to each other, a bit like procedures within an imperative language. Each subset then corresponds to a *task* and the whole program is thus represented as a task graph [33]. Moreover, task graphs have additional assets: it is possible to dynamically replicate the subsets defined by membranes to fully exploit the intrinsic program parallelism; this operation is called *task graph rewriting*. For instance, if a program

enables N simultaneous executions of a given task, replicating the corresponding subset will induce the parallel execution of the N same tasks.

Obviously, such programming languages have many of the features we need to program a space-oriented machine: implicit parallelism as in declarative languages, genericity as in functional languages, yet structured parallelism description as in systolic machines and cellular automata, and finally, dynamic adjustment of program parallelism to the available resources (space) as in task graph rewriting.

Space exploitation and scalability. The same way performance-oriented programming language research focuses on the exploitation of parallelism, performance-oriented novel computing models focus on the exploitation of large computing resources, i.e., space. This rather general formulation of machine scalability implicitly takes into account several other associated technology-independent criteria and constraints: communication time that increases linearly with distance [57], bandwidth which is physically limited per cross section unit, network size which grows in $O(n^2)$ in an all-to-all model where n is the number of processing elements [15] and tolerance to structural defects.

Within the field of alternative computing models, there is a general consensus that cellular architectures represent a major solution for achieving scalability under the abovementioned constraints [62]. Cellular automata are networks of independent automata which evolve depending on their own state and the state of their neighbors only. Thus, communications are only local and there is no need for a communication network as in traditional parallel machines. Cellular automata implement a form of massive parallelism and memory distribution: the distinction between processing and memory disappears and it is replaced with a single component, the cellular automaton, that combines both. Even though cellular automata have been shown to be universal by John Von Neumann in 1952, i.e., one can write any program using cellular automata, all studies or realizations are application-specific: physical simulation, neural networks, and specific computations [63, 38, 53]. Still, the cellular automata paradigm bears several constraints that could limit a true large-scale implementation: they assume a regular structure and thus no structural defect, as well as synchronous evolution of automata.

Amorphous Computing [1] is a novel paradigm based on cellular automata that preserves most of the former properties of cellular automata while getting rid of the latter constraints. This paradigm shows how it is possible to perform simple computations using a cellular automata-like model without assuming structural regularity, a defect-free structure or synchronism. As a result, it is compatible with either large-scale traditional silicon implementations which satisfy the cellular automata constraints, or novel technologies which do not satisfy these constraints, see next paragraph. However, this model does not show how to implement any complex behavior (any program) using unstructured networks of automata, which is one of the main contributions of the Blob paradigm. A Blob computer can be implemented either on a traditional 2-D silicon circuit with all of the abovementioned constraints, or on novel technologies on top of a technology-abstracting paradigm like Amorphous Computing.

2.3 Emerging technologies

With the upcoming end of Moore's law, we need to examine the candidate replacement technologies and decide which ones may provide similar spatial scalability properties.

Chemical-reaction based computing, also called molecule-based computing is a surprisingly old field of research [12]. A chemical reaction represents a function (or an automaton state transition) and different molecules represent different values (or different automaton states). The term-rewriting model is generally the model of choice for this technology [3]: chemical reactions occur in parallel depending on the availability of molecules, much the same way that rules apply on t-uples depending on their presence. Still, this technology is not compatible as is with membrane and task graph computing as molecules are not grouped in structured sets, though chemistry or biology could provide such properties in the future. Chemical-based computing is an attractive technology provided it is possible to exploit the massive parallelism of simultaneous chemical reactions. However, because it is difficult to build many different molecules, widespread adoption is unlikely. Assume one wants to represent a decimal number between 0 and 255; in an electronic circuit, it is possible to represent this number using 8 bits and the decimal value associated with the bit at *position* i, $0 \leq i < 8$, is 2^i. In a chemical solution the notion of *position* does not exist, so we need to use 256 molecules to represent all possible numbers. For large problems, it will not be feasible to generate enough molecules.

Another approach to chemical computing is DNA computing [2] which solves the number of molecules issue by using only four different molecules (the DNA four bases) and creating molecules of huge lengths, i.e., it is a base-4 computing model instead of a binary (electronics) or unary (chemical computing) computing model. DNA computing has been used to solve small-scale function optimization problems [41], but critics point out that solving large-scale problems will require excessive amounts of matters. Still, while DNA computing initially seemed appropriate only for computing large functions, recent research works demonstrated automata computations using DNA, opening up interesting possibilities [46].

Molecular electronics is emerging as a possible candidate for post-lithography electronics [24]. The assets of molecular electronics do not necessarily rest in much smaller transistors but in simpler manufacturing processes relying on molecule growth, and thus fairly large structures at a reduced cost. Since the manufacturing process would induce numerous defects, the corresponding computing paradigm must be defect-tolerant, such as the Amorphous Computing paradigm or more traditional reconfigurable computing paradigms [16]. Still, existing transistors based on molecular electronics, e.g., carbon nanotubes, cannot provide signal amplification and thus, they are not yet compatible with large-scale circuits.

Other very recent research works suggest that it might be possible to design a whole circuit within a single large-scale molecule [54], overriding the signal amplification issues. However in-molecule computing is a very novel technology that is based on different properties than electric signal propagation, and much research work is still needed before an appropriate computing model can emerge.

Quantum computing [18] is both a new technology and a new paradigm. Though it can potentially solve some NP-difficult problems in polynomial times [52], quantum computing is not a spatial-oriented technology and it seems restricted to certain algorithms and may not be appropriate as a general-purpose computing paradigm.

2.4 Blob computing

Blob computing vs. other approaches. Like several current research works on traditional computers, a Blob computer corresponds to a regular grid of processing elements, memory being embedded within each processing element and thus implicitly distributed. The Blob computing model relies on a spatial-oriented

language where a program is a combination of computations and space management operations. Still, the language is sufficiently abstract and machine-independent for implementing many algorithms, the Blob computing model being a general-purpose machine with a special strength for spatial-oriented computations.

The Blob computing model is an alternative model that, like declarative and dataflow models, embeds implicit parallelism in the language semantics. As for functional programming languages, the ability to execute irregular programs with dynamic data structures is key to the model. On the other hand, unlike dataflow models, the Blob computing model specifies the algorithm spatial structure. The general idea is to strike the right balance between the programmer effort and machine/compiler effort while achieving high performance. The Blob computing model retains the spatial orientation of cellular automata-based architectures, but it proposes a more machine-independent and easier to use language that is suited for programs with complex data and parallelism properties.

Even though the Blob computer design investigated in this study is based on traditional electronics, many of the currently emerging technologies can be characterized as “space-oriented” technologies (chemical and DNA computing, molecular electronics, in-molecule computing) and thus, they could ultimately provide an adequate basis for a Blob computer implementation.

Past work on Blob computing. The initial research studies on Blob computing were theoretical research works on defining a language based on cellular development (*Cellular Encoding*) [26], on designing a compiler to translate imperative programs into a cellular code [30], on proving the optimal space/time model performance within a simplified framework [31], and finally on introducing the Blob concept and proving implementation correctness on an arbitrary architecture [28]. The present and upcoming studies aim at presenting the principles of Blob computing, Blob programming and investigating implementation issues of Blob computers.

3. BLOB COMPUTING

The Blob computing model encompasses both a programming model and an architecture model. In the following sections, we progressively introduce the different notions at the root of the Blob computing model and their rationale. In order to explain the programming model and the main principles of Blob computing, we introduce a simplified and abstract representation of a Blob computer called the *Abstract Blob Machine*. We then introduce the main instructions of a Blob computer and illustrate Blob programming with simple examples. Finally, we propose a first implementation of a Blob computer.

3.1 Principles

Network of automata. An abstract Blob machine is composed of a large number of elementary components called *processing elements* (PEs), though their behavior, structure and interactions are very different from the processors found in traditional parallel machines; the PEs are the abstract equivalent of Blobs. A processing element implements a finite-state automaton, and more precisely a Mealy automaton with actions (outputs) specified on the transition links. Links connect processing elements to form a network; the network structure can be irregular but processing elements can only communicate with their *neighbors*.

The links between PEs are oriented bidirectional links identified with labels. PEs can send/receive data to/from the other PEs to which they are linked, i.e., the *neighbor* PEs. A PE can identify one of its links using both its label and its direction, e.g., the outgoing link L_1 denoted $L_1\rightarrow$, see Figure 1, though it can happen that two or more links have the same label and direction.[1]

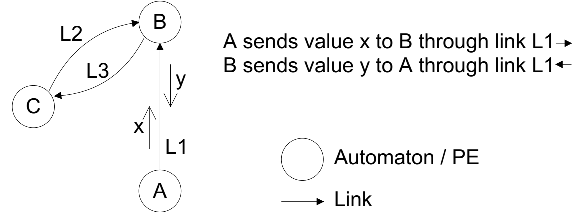


Figure 1: *PEs and links*

The evolution of an abstract Blob machine consists in an infinite sequence of automata state transitions (computations), communications between PEs, transitions again, etc. For the sake of simplicity, we will assume a synchronous model in this study: all computations, and then all communications are performed simultaneously, much like in a clocked SIMD machine. However, the final Blob computer implementation will be an asynchronous machine.

Exploiting space and parallelism. The core asset of a network of automata is its ability to exploit both space and parallelism. In a network of automata there is no distinction anymore between memory and processing units: data are distributed among the different automata which can all perform computations in parallel on their stored data. Intuitively, a network of automata behaves like a processor that performs in-memory computations only, and in a distributed way. The size and structure of a network of automata can be adjusted to accommodate either a greater number of data (more memory) or a greater number of parallel computations (more arithmetic and logic operators). Moreover, space and parallelism are intricately related: by exploiting space to distribute data over several automata, a network of automata enables parallel operations on these data. Consider Figure 2 for instance: once spatially developed, a network of automata can perform parallel computations on values in PEs, see `parallel adds`, and connections between PEs realize dataflow dependencies between PE values, see `sum reduction`.

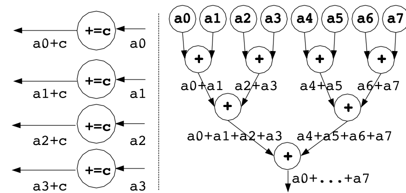


Figure 2: *Sum reduction & Parallel adds on array items*

Self-Developing network. Since exploiting space is at the core of the Blob computing paradigm, a Blob computer has the ability to expand, contract or alter the network of PEs. Therefore, besides sending/receiving data, a PE can duplicate itself creating along the required links, merge with another PE, destroy itself or add/remove any of its links. In other terms, an abstract Blob machine is a *self-developing* network of automata.

[1] In that case, sending a value to any of them will send a value to all of them, and as a result, a PE can receive multiple values simultaneously (unlike in a traditional finite-state automaton).

Consequently, a Blob program performs two types of operations: traditional data and control operations, and network development operations. It is possible to perform either separately or simultaneously both types of operations. In certain cases, it is more efficient to set up the network and then send data at a rapid pace to perform computations, while in other cases, the network needs to adjust dynamically to the amount of data (the equivalent of dynamic allocation) and computations (a program with both control and computing intensive parts).

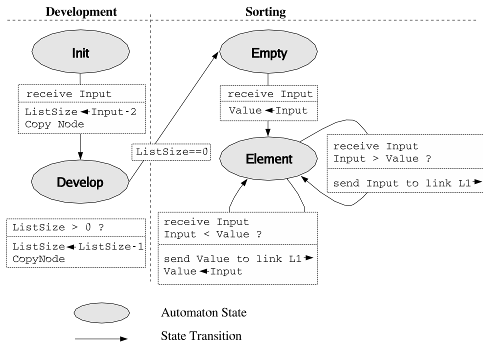


Figure 3: *Separate development and computations (automaton).*

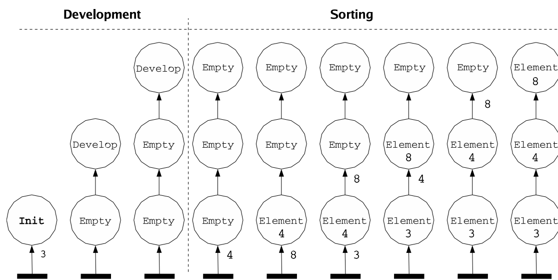


Figure 4: *Separate development and computations (execution).*

Let us illustrate the principles of a Blob computer with a simple example which consists in sorting a list of numbers. In the corresponding automaton, see Figure 3, network development and computations are separated.[2] Consider Figure 4 which represents the development of a network of three PEs used to sort a list of 3 numbers. Each PE can store one number of the list. At the beginning, there is a single empty PE which has a special input link through which values are received. This PE receives a first input value which is the list size. The automaton specifies that the PE replicates and passes the decremented list size to the new PE, and so on until the list size is equal to 0.

After the development, the network receives a sequence of values. Each automaton acts as follows: if it has no value, it stores the incoming value; if its own value is larger than the incoming value, it sends its own value to the outgoing link (to the upper PE) and stores the incoming value. Finally, if its own value is smaller than the incoming one, it sends the incoming value to the outgoing link. The list is sorted when all automata are idle.

[2] A formal description of these automata is given in Section 3.2.

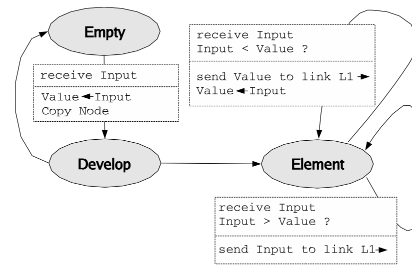


Figure 5: *Simultaneous development and computations (automaton)*

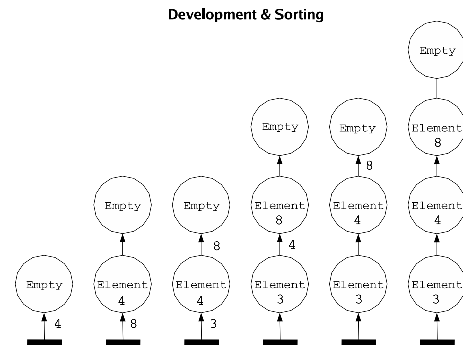


Figure 6: *Simultaneous development and computations (execution)*

Using simultaneous computations and network development, it is possible to write a more efficient program for the same task, and to accommodate an arbitrarily large data set size. Consider again the problem of sorting a list of integers where the list size is now arbitrarily large. The corresponding automaton is shown in Figure 5. At the beginning, there is again a single PE with its special input, but the list size is not provided, see Figure 6. Each automaton performs simultaneously network development and value sorting as follows : when an automaton receives a value, it tests its state and value. If its state is `Empty` it creates a PE between itself and the source of its incoming link and stores the new value in this new PE, remaining in the `Empty` state; the new PE is in the `Element` state. Whenever a PE in `Element` state receives a value, it acts as the previous automaton of Figure 3 (passing the value to the upper PE if it is smaller than its own). Thus, the network of PEs (a 1-dimensional network in this case) expands to accommodate new values and sorts them simultaneously.

Putting it all together. Consider a simple FIR (Finite Impulse Response filter) algorithm that performs a MAC (Multiply-Accumulate) on N parameters, see Figure 7. This example exploits parallelism at different levels: pipeline parallelism since data are streamed into the network in a pipelined manner, parallel multiplications of the N parameters with the current values, and parallel and pipelined additions in a tree-like manner to compute the final result at each step. In this case, there is no need for adjusting the network size during the computations, so the network is developed first.

The development phase of the program consists in creating the

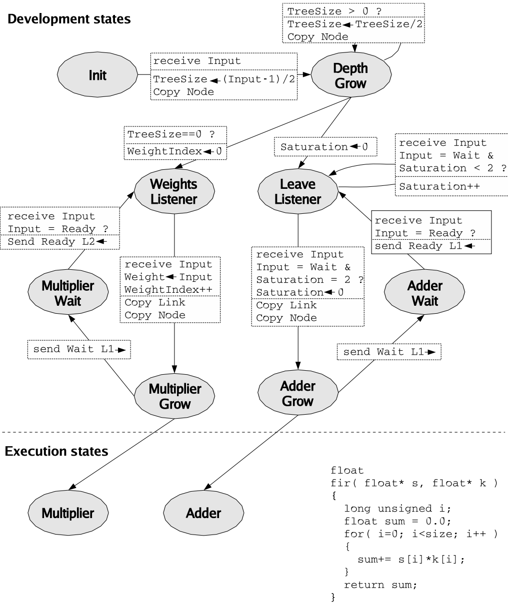


Figure 7: *FIR (C program and automaton)*

N parallel multipliers and for each pair of multiplier PEs to create an adder PE, and again for each pair of adder PEs another adder PE and so on. In Figure 7, the states on the left build the multipliers chain while the states on the right build the adder tree. The resulting network is shown in Figure 8 for $N = 6$ and $N = 8$. The computing phase consists in fetching the lower values for the multiplier PEs and sending the result to the adder PEs, and for the adder PEs in adding the two incoming values and passing the result to the next adder PE, and so on.

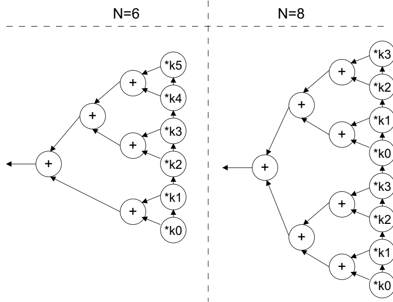


Figure 8: *FIR (network)*

In the FIR algorithm, the greater the number of parameters (N), the greater the filter accuracy. In the C version of the FIR filter in Figure 7, the number of elementary operations ($+$, $\times$) is equal to $2 \times N$. In the Blob version of FIR, the number of PEs is also equal to $2 \times N$, and the execution time is in $O(logN)$. In the C version, greater accuracy comes at the cost of increased running time, while in the Blob version, achieving greater accuracy costs additional space and little additional time.

Note that it is possible to write a parallel version of FIR for a multiprocessor machine with the same execution time complexity, but a Blob program simultaneously expresses both the algorithm implicit parallelism and how to develop the network to exploit space for this program. The role of the Blob computer will then consist in appropriately mapping this PE network on the hardware.

Scalability: Blob computing versus processors and multiprocessors. The execution of the above FIR example on a Blob computer benefits from the pipeline parallelism (signal values are pipelined into the machine) and the fine-grain parallelism (parallel adds) both already exploited in superscalar processors. Blob computing does not propose new ways to improve performance, but rather a programming and architecture paradigm that achieves easier and better *scalability*. For instance, scaling up a traditional superscalar processor to execute 6 instructions in parallel instead of 4 means scaling up the size and capacity of many of its components, see Section 2, and preferably recompiling the program. Scaling up a Blob computer to achieve more parallelism for the same example simply means increasing the amount of space available for Blobs (PEs), without even changing the program.

More generally, a Blob computer can circumvent many of the main performance bottlenecks of today's microprocessors and multiprocessors:

1. By expressing its algorithm as an automaton using *Cellular Encoding*, the program provides implicit parallelism to the architecture, so that dynamic instruction-level parallelism or complex compiler-based static analysis for finding program parallelism are no longer necessary. One may argue that additional efforts are required from the user, but our experience with *Cellular Encoding* tends to show that writing a cellular code does not mean spending special efforts to find parallelism in an algorithm but rather expressing an algorithm using a fairly natural space-oriented perspective, see Section 4 for examples. With respect to parallelism, the biggest benefit comes from avoiding the sequential bias of imperative languages.

2. As a result, a program is no longer a sequence of instructions with a single program counter but a large set of automata that evolve concurrently. Consequently, an `if` statement does not stall a whole program until it is resolved, thus there is no need for complex and hard-to-scale branch prediction mechanisms [34] anymore.

3. Memory latency issues no long exist as such since memory and computations are merged into Blobs (PEs). The problem now consists in making sure that Blobs that need to communicate are located closed to each other to minimize communication time.

4. In fact, placing Blobs within the available space becomes the main issue and it is similar to the difficult issue of distributing tasks over many processing elements in a multiprocessor. However, in a Blob computer, there is no longer a distinction between processing elements and the interconnection network (processing elements *are* the interconnection network). Thus, the problem becomes finding the optimal way to place objects on a continuous space; while this problem is NP-hard, we will see in Section 3.3 that mimicking basic physical laws like pressure and elasticity provides a simple and efficient heuristic. Still, we have yet to perform extensive measurements on the performance of this heuristic.

5. As mentioned above, a Blob computer is a large "continuous" space, i.e., it can be viewed as both a processor or a multiprocessor. As a result, in a Blob program/computer there is no distinction between the fine-grain parallelism (ILP) of superscalar processors and the coarse-grain parallelism of multiprocessors. Consequently, it is no longer necessary to search the right balance between both types of parallelism as processor performance or the number of processors evolve. Only SMT processors [60] can combine the exploitation of fine-grain and coarse-grain parallelism on the same architecture, but SMT processors are based on superscalar-like architectures which are difficult to scale.

3.2 Programming a Blob computer

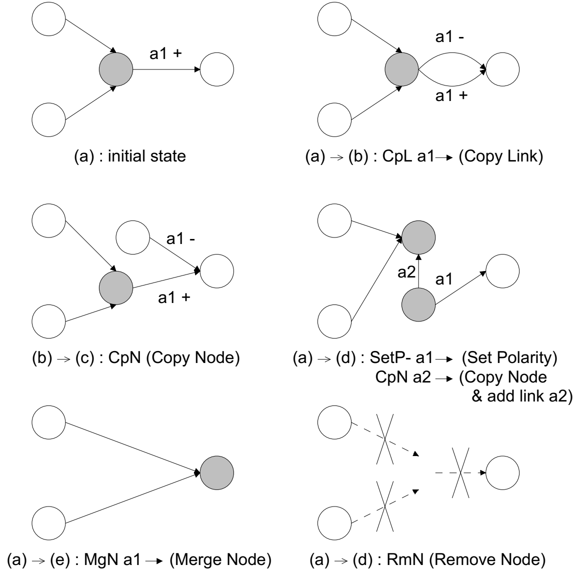


Figure 9: *Blob instructions*

As mentioned above, an automaton of a Blob computer performs two types of operations: traditional arithmetic and logic operations and network development operations. Therefore, in the abstract Blob machine, we introduce instructions to specify traditional arithmetic and logic operations, and we introduce control signals to specify network development operations, see Section 3.3. These instructions and signals are the equivalent of assembly instructions in a traditional processor.

Network development signals either expand or contract the network of automata. The process of duplicating a PE is split into two steps:

1. First, the links to/from the PE are duplicated using the `CpL (Copy Link)` signal. To distinguish between two links of a pair, each link is automatically assigned a *polarity* (+ or -), see Figure 9.

2. Then, the PE is duplicated using the `CpN (Copy Node)` signal which creates two copies of the PE distinguished again by a polarity (+ or -). During the duplication process, duplicated links bind to duplicated PEs depending on their polarity.

Conversely, it is possible to remove a link using the `RmL (Remove Link)` signal or a PE using the `RmN (Remove Node)` signal (the PE self-destructs). The complete set of the abstract Blob machine control signals is listed in Figure 9.

The `SetP (Set Polarity)` instruction enables asymmetric link repartition during PE duplication. Consider a link L to/from a PE: using signal `CpL`, the link L is duplicated and, after the PE is itself duplicated using signal `CpN`, one link copy is associated with each of the PE copies. Now, if we want only one of the two PE copies to have an L link, we would use signal `SetP` to polarize the link, e.g., `SetP+`, instead of signal `CpL`; then, when signal `CpN` is used, the link would be attached only to the PE with the corresponding polarity, e.g., +, see Figure 9, $(a) \rightarrow (d)$.

Note that after PE duplication, links keep their polarity; applying `CpL` to a polarized link still creates two copies with opposite polarity. Even though instances of the same link have the same address and direction, and thus cannot be distinguished, we have shown that it is possible to have a universal and tractable programming model [28].

Another important signal is `MgN (Merge Node)`: when a PE is removed using `RmN`, the links to/from the PE are removed as well, and consequently, the target PEs of these deleted links are not connected anymore. Using signal `MgN` it is possible to remove a PE and still preserve connectivity, see Figure 9, $(a) \rightarrow (e)$.

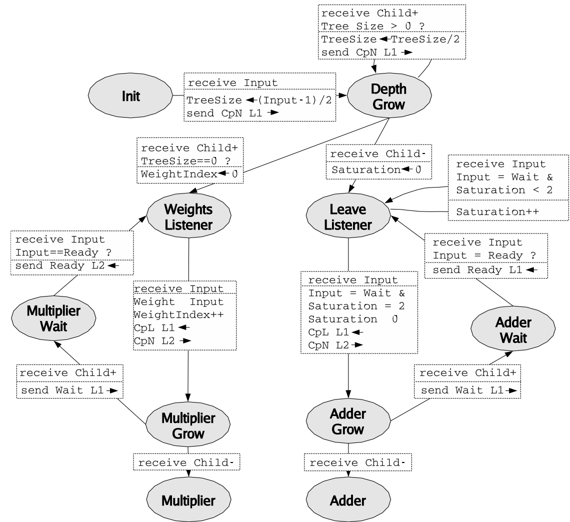


Figure 10: *FIR automaton*

The FIR automaton of Figure 7 using Blob instructions is shown in Figure 10.

3.3 Investigating the implementation issues of a Blob computer

A traditional computer can be considered as an implementation of a Turing machine, replacing the tape with an efficient memory mechanism (register, addressable memory) while retaining the main structural underpinning of the model: the separation between an active processing part and a passive memory part. With respect to the infinite memory size of the abstract Turing model, the limitation of traditional computers are their finite-size memory which restricts the scope and dimension of target applications. Likewise, a Blob computer is an implementation of the abstract Blob machine. In this case, we need to find a mechanism for implementing a network of PEs, while retaining the main concepts of the paradigm:

1. separate processor/memory parts are replaced with a large set of simple hardware components, called *cells*, capable of performing both elementary arithmetic/logic operations and storing small amounts of data; once combined into a network, several such cells provide the same capability as one or several processor+memory pairs,

2. these cells can communicate with their neighbors and thereby form a network, Figure 11 shows an example 2-D grid network topology where each cell has 8 neighbors; there is no other communication links between cells,

3. a PE automaton is the software running on the hardware cell network: the automaton is split into a set of elementary computing or memory operations, called *particles*; one particle is mapped to one cell, but a particle can move on the cell network and change cell; since PEs are made of particles and particles can move, PEs can move as well; moreover a PE can create new particles and map them on available neighbor cells; overall, with this particle/cell implementation, it is possible to create the illusion that PEs can move and duplicate as in a self-developing network,

4. through particles and cells, each PE is in fact assigned a given area of the hardware cell network; consequently, the different PEs of a program rest on distinct areas of the hardware cell network, and thus the PEs automata can execute concurrently; in other terms, implementing PEs using particles and a hardware cell network is enough to exploit the implicit parallelism of a network of PEs.

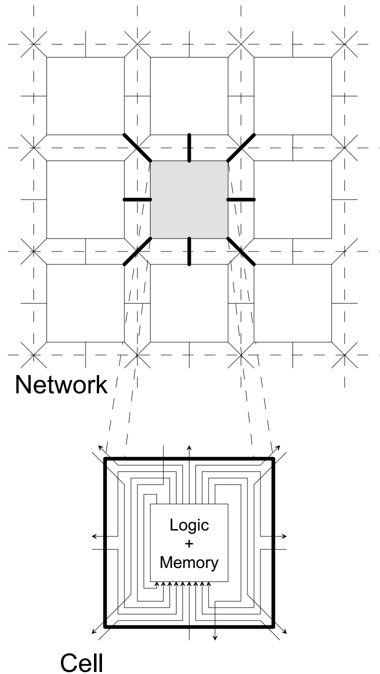


Figure 11: *The hardware cell network*

The abstract Blob model assumes an infinite space size. The limitation of the Blob computer is naturally the finite space size which restricts the scope and dimension of target applications.

Implementing a PE automaton using "particles". In a traditional processor, programs are decomposed into a restricted set of different elementary operations (assembly instructions). Because any program can be decomposed into a long sequence of a few such simple operations, one simple architecture component, the ALU, is enough to execute almost any program. The processor is a generic implementation of a program even though it is less efficient than a dedicated circuit like an ASIC.

For the same reason, we do not use a typical automaton implementation (a state register, a more or less large memory for the state transition table, plus a given number of input and output registers) where the table size and the number of input/output registers are fixed, thus limiting the size and number of variables of individual PE automata. Instead, we decompose a PE automaton into a set of individual state transitions, i.e., particles. Each particle performs one state transition of the automaton and/or one of the arithmetic and logic operations associated with this state transition, see Section 3.2. The cell is the hardware component associated with the particle: one particle per cell. Note again that in a traditional processor the program is spread over time (a sequence of elementary instructions performed on the same CPU), while in a Blob computer it is spread over space (a group of elementary particles).

Consider again the example of Figure 3. The automaton has 6 transitions and almost all transitions except one perform a single operation. The corresponding group of particles is shown in Figure 12. For instance, consider the transition ELEMENT → ELEMENT: IF `Value` > `Store` THEN SEND `Value` TO `Link1` OUTGOING. It is implemented with 3 particles: one particle to perform the test, another particle to send the value to the link and a final particle to broadcast the new state thus finishing the state transition.

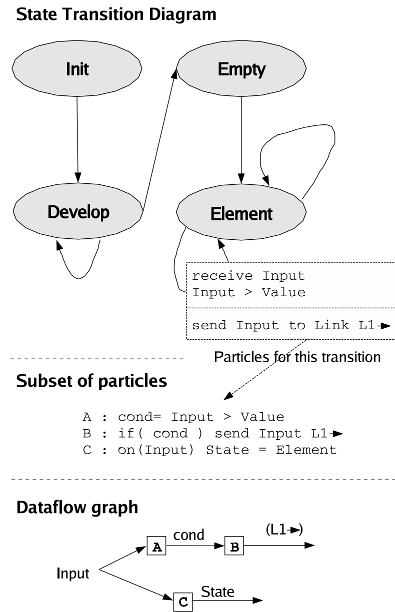


Figure 12: *Coding a state transition using particles*

The set of particles in a PE actually implements a *dataflow graph*, e.g., Figure 12 shows the dataflow graph of the abovementioned transition. Whenever a particle has received all its input variables, it performs its operations and possibly sends a result to other particles. Within a PE, particles communicate using *signals*. These signals are *broadcasted* to all particles, there is no particle address nor networks for propagating signals. Particles can only send signals to the particles in the neighbor cells, and the signals are passed from cell to cell until all PE particles are reached. A signal contains the following information: the transition source state identifier, identifier type (either a PE register or a link), an identifier and a value.

Consider again the same transition and the corresponding set of particles in Figure 12; when receiving `Value` through its link, the PE sends the following internal signal to its particles: STATE=`Element`, TYPE=`variable`, VARIABLE=`Value`, VALUE=`xxx`. The (state, variable) pair corresponds to the expected input of the test particle; since no other variable is expected (variable `Store` is stored inside the particle), it performs the test and if the condition is true it broadcasts the following signal: STATE=(*the target state of the transition since this is the last particle in the dataflow graph; in this case, the target state is* ELEMENT *again*), TYPE=`link`, VARIABLE=(*link address*), VALUE=(*the value contained in Store*). The PE will then send this value to the corresponding link; inter-PE communications are explained later.

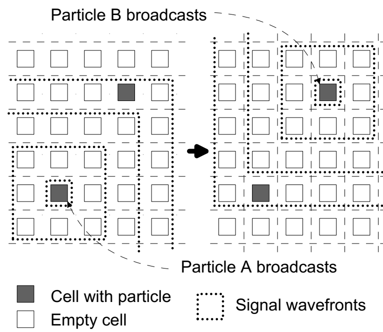


Figure 13: *Signal broadcasts within a Blob*

The Blob: the group of particles in a PE. Each particle in a PE automaton uses a hardware cell; when visualized on a 2-D grid, as in Figure 14, this group of particles look like a blob, especially when it is moving and changing shape when particles move, replicate or collide with other PE particles, hence the term Blob computing.

Whenever a particle reaches a hardware cell, it sets a special bit which indicates that the cell is now part of a Blob. Implicitly, this bit serves to implement a topological definition of the Blob: the Blob is a region of the hardware cell network (a set of connected hardware cells), see Figure 14. This region also defines a local communication area among particles of the same Blob: whenever a broadcasted signal reaches a cell that is not in the Blob, the signal is ignored, i.e., it dies outside the Blob.

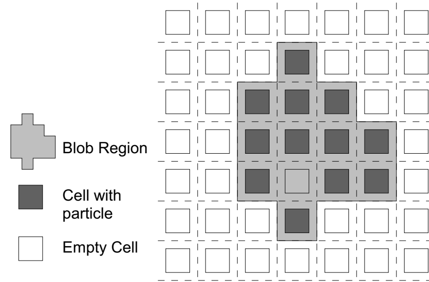


Figure 14: *Implementing a Blob using particles on hardware cells*

Blob movement and division. When Blobs are duplicated, they need additional space, thereby moving neighbor Blobs in the 2-D space. Now, how can we organize the movement and placement of all Blobs and the particles inside ? Mapping a program task graph to a processor network is a central and NP-complete problem of parallel computing [49]. The topology of a Blob computer is particular: instead of using a traditional computer network, it partitions space into neighboring elements that can only interact with each other. Then, the problem consists in finding a proper layout for objects of varying size and shape in a 2-D (or 3-D) space. Moreover, this layout must dynamically adjust to changing Blob positions, shape and population count. One of the key ingredients of the Blob machine is a novel bio-inspired placement algorithm to address this issue; "nature" had to solve a similar placement problem for arranging billions of neurons in the 20 cm^3 of our skull: the development of neuronal cells is genetically programmed but the actual placement of neurons is largely influenced by fundamental physical laws such as pressure exerted among cells.

The Blob computer attempts to emulate this placement method by implementing the following properties:

• A Blob computer simulates some of the traditional laws that guide objects relative placement, movement and shape in nature, such as pressure. Since the program is expressed as a set of physical objects (Blobs) which can move over time within the space (hardware cell network), their placement, movement and shape are ruled by these physical laws. Since these laws are enforced permanently, they can accommodate the dynamic and continuous development of the objects.

• Since, by essence, the same physical laws apply to all parts of the space, to enforce them in the whole system, one only needs to implement them locally at the level of each individual machine component (a hardware cell).

• Besides the basic physical laws, it is possible to add more physical laws to better take into account object properties. For example, the fact that communicating objects need to be placed nearby can be modeled using elasticity.

While the detailed implementation of a Blob computer is still on-going work and is out of the scope of the present study (see [29] for preliminary results and more details), we outline below the main intuitions.

Movement is based on gas particles principles. Gas particles essentially move using two main rules: (1) a gas particle modifies the trajectory of another particle by communicating its *momentum* to the other particle when they collide, (2) while a gas particle can move in a certain direction, its trajectory is not straight but locally erratic (i.e., seemingly random over a short distance). The role of the first rule is to implement the pressure that a Blob exerts on another Blob through its particles. The role of the second rule is to let the particles of a Blob (and thus the Blob itself) find a path by themselves (implicitly using trial and error through their random movement) without having the programmer explicitly specify the Blob path. Figure 15 illustrates the movement of Blob particles (during a Blob division).

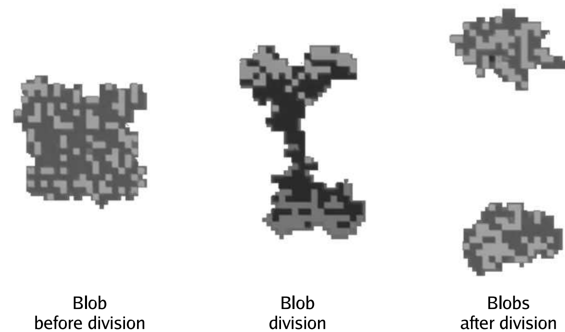


Figure 15: *3 Blob division stages.*

Both rules share an important property: they are *local* rules, i.e., they manage Blob movements without knowledge of what is happening in other space areas. As a result, they are easy to implement and they significantly reduce the architecture and program complexity compared to more traditional computers. The general self-organizing principle of a Blob architecture is summarized by the following observation from Turing in his work on biological morphogenesis [61]: "Global order can arise from local interactions".

Blob division. Similarly, Blob division is managed by a set of local rules. This process is inspired from (though not identical to) biological cells mitosis [29], i.e., the division of biological cells. When a Blob receives a division signal, all particles duplicate; each particle is flagged $-$ or $+$; using a signal wave, particles are gathered in two sets (only $-$ particles and only $+$ particles); then, when the separation is complete, the two sets become two independent Blobs, see Figure 15.

Inter-Blob communications. We have not yet tested the low-level implementation of inter-Blob communications, but we envision them as follows. Blobs are connected to each other using links. Links are implemented using specific Blobs, called *Link Blobs*. At both ends, these link Blobs are attached to standard Blobs using a single special *link particle*. In order to keep as a compact spatial Blob distribution as possible for a given program, these link Blobs are driven by elasticity laws, i.e., they have the same physical behavior as springs or rubber bands: both ends attempt to get close to each other. That property is again implemented as a simple rule on the link particles: they are attracted to the direction of their immediate neighbor particle on the Blob link membrane. Since this particle is in the same direction as the other end of the Blob link, the particle implicitly attempts to go in the direction of the link particle at the other end of the link Blob.

4. EXPERIMENTS AND EXAMPLES OF APPLICATION

4.1 Experimental Framework

We have written two Blob computer simulators: a functional simulator and a performance simulator. The first simulator is an *Abstract Blob Machine* simulator: it interprets simple arithmetic and logic operations, register storage operations and the network development operations described in Section 3.2. To a certain extent, this simulator is a *functional* simulator, i.e., an emulator: it can execute a Blob program but does not provide any information on the program performance. The performance simulator describes a simplified implementation of the Blob computer; even though the notion of pressure and elasticity have been implemented in that simulator, their implementation is much more simple than the rules described in Section 3.3. One physical rule (Blob division) has been implemented thoroughly in the SIMP/STEP cellular automata simulator by Bach and Toffoli [59], and the corresponding results are presented in [29] and an example in Figure 15. In the performance simulator, PEs automata are decomposed into particles and particle computations and communications (signal broadcasts) are simulated. This second simulator provides information on program execution time and space, i.e., Blob computer performance metrics, see Figure 20. The goal of the performance simulator is to validate the concept of "particles" and show they can be used to implement complex computations. Due to the spatial nature of Blob executions, the simulator comes with a graphical user interface for visualizing the development of Blob programs, see Figures 19, 23.

In the next paragraphs, we show how to program three fairly different kinds of problems using a Blob computer: the QuickSort algorithm, the Kung and Leiserson [37] matrix multiply algorithm, and a neural network program used for robot control [27]. Performance metrics are shown for the QuickSort algorithm only. Except for the QuickSort Blob program which mixes network development and computations, only the computational parts of the Blob programs are shown below, the development phase is omitted though illustrated in the figures.

4.2 QuickSort

QuickSort is a recursive "divide & conquer" algorithm and it works as follows. At any recursion level, the algorithm creates three lists: the sets of data larger, smaller than the pivot and equal to the pivot; the process is then recursively called on the "smaller" and "larger" lists, Figure 16.

FUNCTION QUICKSORT → LIST OF NUMBERS
INPUT l: LIST OF NUMBERS
$v \leftarrow l[1]$
$left \leftarrow$ LIST OF THE ELEMENTS OF l STRICTLY SMALLER THAN v
$right \leftarrow$ LIST OF THE ELEMENTS OF l STRICTLY GREATER THAN v
$middle \leftarrow$ LIST OF THE ELEMENTS OF l EQUAL TO v
RETURN (CONCAT(QUICKSORT($left$),
CONCAT($middle$,QUICKSORT($right$))))

FIGURE 16: *QuickSort algorithm*

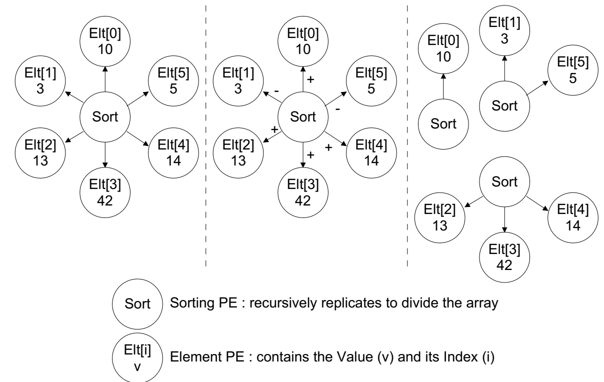


Figure 17: *One iteration of the QuickSort algorithm*

Initially, there is a single list value per PE `Elmt` and all PEs are connected to a central `Sort` PE; each of the `Elmt` PE has a unique index ranging from 0 to $N-1$ where N is the list size, see Figure 17. The `Sort` queries the 0 index and the corresponding list value from all PEs; it then broadcasts this value, which it is the pivot value, to all PEs. Each PE then polarizes its link with the `Sort` PE as follows: if its own value is strictly smaller than the pivot value, it sets the polarity to $-$, or to $+$ otherwise. Then the `Sort` PE divides itself into two polarized PEs, so that all values smaller than the pivot value remain with the $-$ PE, and all other values with the $+$ PE. Using a second division of the $+$ PE, a third `Sort` PE is created that is only linked to the pivot PE. Then, the `Sort` $+$ PE and the `Sort` $-$ PE communicate with their respective list to recompute the index of all their PEs and set a new 0 index in each list. The process then starts again recursively. The corresponding automaton is shown on Figure 18. Note that the full QuickSort Blob program has a final phase that builds a true list by scanning all PEs and destroying them in the process.

The full QuickSort Blob program was run on the particle-level simulator. The execution time complexity is in $O(\sqrt{N})$ as compared to $O(Nlog(N))$ at best on a traditional processor, and the space complexity is in $O(N)$, see Figures 20, 21. The execution time is expressed in number of hardware cell cycles. A single transition operation, passing a signal from one cell to another, or passing a particle from one cell to another all require a single cycle.

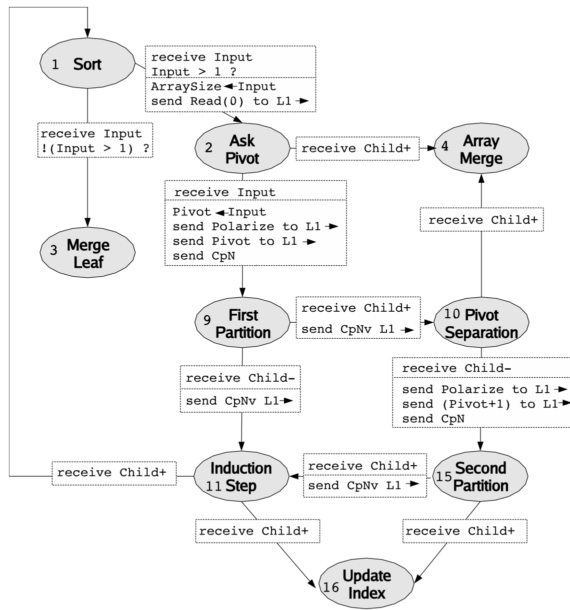


Figure 18: *Quicksort automaton*

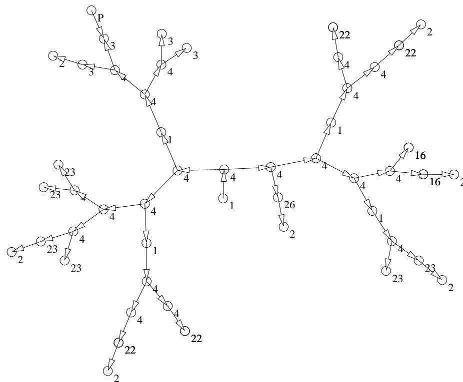


Figure 19: *Simulated execution of the Blob program*

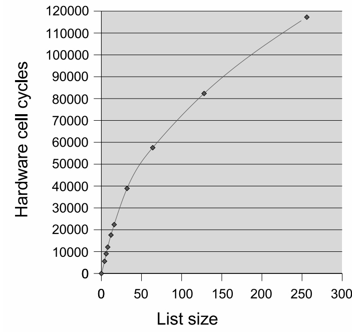


Figure 20: *Time performance*

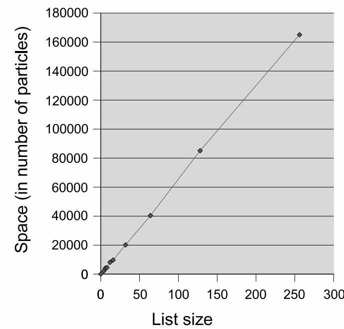


Figure 21: *Space performance*

The number of particles per Blob is equal to 76. The total number of particles may seem large but it is actually equal to the number of memory and register elements (temporary list elements) normally used in the algorithm run on a traditional processor. In a Blob computer, the development is spatially-oriented and all intermediate variables become particles. It is possible to optimize the spatial cost by several orders of magnitude through several tradeoffs that we have not yet explored: the number of state transition operations per particle, and finding an intermediate programming style between the spatial-only Blob programs and the time-only imperative language programs.

4.3 Matrix Multiply

The principle of the Kung and Leiserson matrix multiply algorithm is again to view the process in a spatial, but also a streamed, manner, see Figure 22 (note that in this figure, PE_{ij} actually corresponds to the matrix element c_{ij}). The matrices on the top and the left are the operand matrices, while the matrix in the middle is the result matrix. First, a_{00} and b_{00} are sent to c_{00} which will contain the inner product of a_{0j} and b_{i0} at the end of the algorithm; c_{00} is now equal to $a_{00} \times b_{00}$, and a_{00} is passed to c_{01} while b_{00} is passed to c_{10}. In the next step, c_{00} receives a_{01} and b_{10}, c_{01} receives b_{01} (and a_{00} from c_{00}), and c_{10} receives a_{10} (and b_{00} from c_{00}); and so on.

Again, due to its spatial nature, this algorithm easily translates into a Blob program. The first part consists in developing the network of PEs; in this case, one PE is associated with each c_{ij} element. The development is performed as shown in Figure 23, each PE having the same automaton and the only input is the matrix dimension. The upper and leftmost PEs are attached to input/output ports. Then, each resulting PE only keeps the computational part of the automaton, shown in Figure 24 and the computations are performed as described in the above paragraph. The space requirements are in $O(N^2)$ where N is the matrix dimension, and the execution time is in $O(N)$.

Note that Blob computing and programming is not only dedicated to programs with large data sets. Computing-intensive programs with small data sets also strongly benefit from Blob computing by spatially developing parallel or recursive computations.

4.4 Neural Network

This example neural network controls the locomotion of a 6-legged robot using a simplified model: for each leg, there are two sensory neurons N_a, N_b activated when the leg reaches an extreme position (to know when it must retract or expand), and two motor neurons M_a, M_b to move the leg up/down and forward/backward.

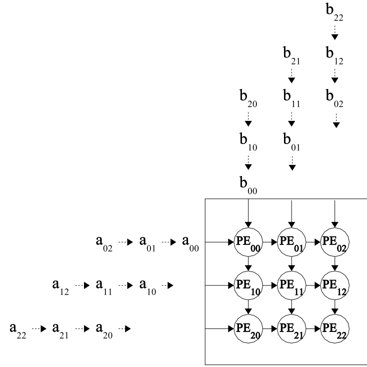
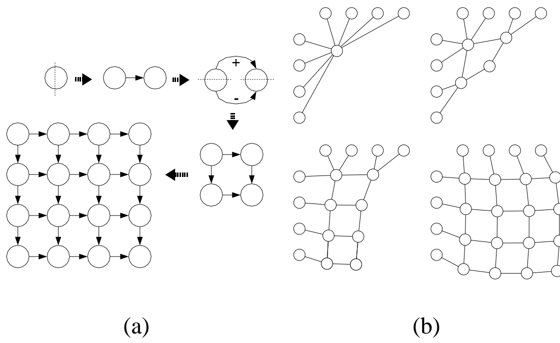


Figure 22: *PE network for the Kung & Leiserson matrix multiply algorithm*



(a) (b)

Figure 23: *Network development: (a) principles, (b) simulation*

The robot moves alternatively three of its legs forward and three of its legs backward. Since there are 12 sensors and 12 actuators, there are 12 input ports and 12 output ports. As for the algorithm of Section 4.3, the Blob program starts with a development phase illustrated in Figure 25. The automaton of one leg control neuron is shown in Figure 26. Since neural networks are also a type of "spatial" computation by nature, they lend well to Blob spatial development: one or a few PEs per neuron and parallel execution of the different neurons/PEs.

5. CONCLUSIONS AND FUTURE WORK

This article presents the Blob computing model which can scale more easily with technology and space than current processors and multiprocessors. These properties are embedded both in the language (*Cellular Encoding*) and the architecture (*Blobs*) which is based on cellular automata. We have also investigated implementation issues of a Blob computer, and presented simple execution examples on a simplified Blob computer implementation.

Future studies will focus on implementation issues of a Blob computer. We also wish to investigate more traditional processor organizations that retain some, though not all, of the principles of Blob computing.

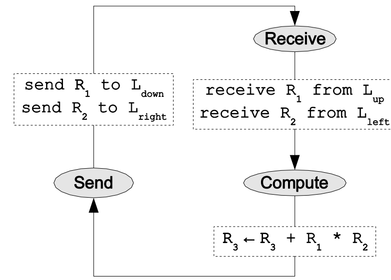


Figure 24: *The computing part of the automaton*

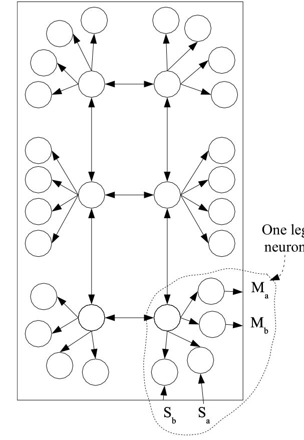


Figure 25: *Neural network development*

6. REFERENCES

[1] H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, T. F. Knight, R. Nagpal, E. Rauch, G. J. Sussman, and R. Weiss. Amorphous computing. *Communications of the ACM*, 43(5):74–82, 2000.

[2] L. Adleman. Molecular computation of solutions to combinatorial problems,. *Science*, 266, November 1994.

[3] J-P. BANÂTRE and D. Le MÉTAYER. Gamma and the chemical reaction model : Ten years after. In J.-M. Andreoli, H. Gallaire, and D. Le Métayer, editors, *Coordination Programming: Mechanisms, Models and Semantics*, pages 1–39, 1996.

[4] M. Baron. Microprocessor report. *Tidbits*, March 2001.

[5] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[6] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoefinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[7] F. Bodin, T. Kisuk, P. Knijnenburg, M. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *Workshop on Profile and Feedback-Directed Compilation at PACT*, 1998.

[8] J. M.P. Cardoso and H. C. Neto. Fast hardware compilation of behaviors into an FPGA-based dynamic reconfi gurable computing system. In *Proc. of the XII Symposium on Integrated Circuits and Systems Design*, pages 150–153. IEEE Computer Society Press, October 1999.

[9] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. DeHon. Stream computations organized for reconfi gurable execution (SCORE). In *FPL*, pages 605–614, 2000.

[10] L. Chen, S. Dropsho, and D. H. Albonesi. Dynamic data dependence tracking and its application to branch prediction. In *Proceedings of*

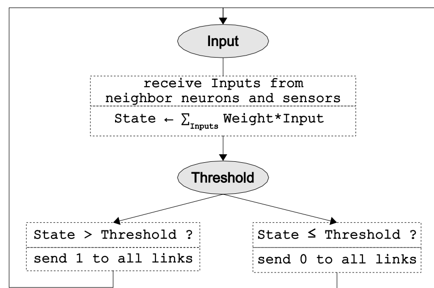


Figure 26: *Neural network automaton*

the 9th International Symposium on High-Performance Computer Architecture, pages 65–77, February 2003.

[11] J.-F. Collard, D. Barthou, and P. Feautrier. Fuzzy array dataflow analysis. In *ppopp*, pages 92–102, Santa Barbara, California, July 1995.

[12] M. Conrad. On design principles for a molecular computer. *Commun. ACM*, 28(5):464–480, 1985.

[13] Z. Cvetanovic and D. Bhandarkar. Performance characterization of the alpha 21164 microprocessor using TP and SPEC workloads. In *HPCA*, pages 270–280, 1996.

[14] A. Dandalis and V. K. Prasanna. Run-time performance optimization of an FPGA-based deduction engine for sat solvers. *ACM Trans. Des. Autom. Electron. Syst.*, 7(4):547–562, 2002.

[15] A. DeHon. *Reconfigurable Architectures for General-Purpose Computing*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1996.

[16] A. DeHon. Very large scale spatial computing. In *Third International Conference on Unconventional Models of Computation*, pages 27–37, October 2002.

[17] André DeHon. The density advantage of configurable computing. *Computer*, 33(4):41–49, 2000.

[18] D. Deutsch and R. Jozsa. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London Ser. A*, A439:553–558, 1992.

[19] Jr. F. H. Carvalho, R. M. F. Lima, and R. D. Lins. Coordinating functional processes with Haskell#. In *Proceedings of the 2002 ACM symposium on Applied computing*, pages 393–400. ACM Press, 2002.

[20] P. Fradet and D. Le Métayer. Structured Gamma. *Science of Computer Programming*, 31(2–3):263–289, 1998.

[21] T. Gautier, P. Le Guernic, and L. Bernard. Signal: A declarative language for synchronous programming of real time system. *Computer Science*, pages 257–277, 1987.

[22] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: an analytical representation of cache misses. In *Proceedings of the 11th international conference on Supercomputing*, pages 317–324. ACM Press, 1997.

[23] P. N. Glaskowsky. Microprocessor report. *Network Processors Mature in 2001*, February 2002.

[24] S. C. Goldstein and M. Budiu. Nanofabrics: spatial computing using molecular electronics. In *Proceedings of the 28th annual international symposium on Computer architecture*, pages 178–191. ACM Press, 2001.

[25] F. Gruau. Process of translation and conception of neural networks, based on a logical description of the target problem, us patent en 93 158 92, december 30, 1993.

[26] F. Gruau. Automatic definition of modular neural networks. *Adaptive Behaviour*, 3(2):151–183, 1995.

[27] F. Gruau. Modular genetic neural networks for 6-legged locomotion. *Artifical Evolution*, pages 201–219, 1995.

[28] F. Gruau and P. Malbos. The Blob: A basic topological concept for hardware-free distributed computation. In *Unconventional Models of Computation (UMC'02), Kobe, Japan*, pages 151–163, 2002.

[29] F. Gruau and G. Moszkowski. Time-efficient self-reproduction on a 2-d cellular automaton. In *1st International Workshop on Biologically inspired Approaches To Advanced Information Technology*, 2004.

[30] F. Gruau, J-Y. Ratajszczak, and G. Wiber. A neural compiler. *Theoretical Computer Science*, 141(1, 2):1–52, April 1995.

[31] F. Gruau and J. T. Tromp. Cellular gravity. In *741*, page 10. Centrum voor Wiskunde en Informatica (CWI), ISSN 1386-3681, 1999.

[32] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[33] C. Hankin, D. Le Métayer, and D. Sands. Refining multiset transformers. *Theoretical Computer Science*, 192(2):233–258, 1998.

[34] T. H. Heil, Z. Smith, and J. E. Smith. Improving branch predictors by correlating on data values. In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 28–37. IEEE Computer Society, 1999.

[35] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, , and P. Roussel. The microarchitecture of the pentium 4 processor. *Intel Technology Journal Q1*, 2001.

[36] J.W. Klop. Term rewriting systems. In *Handbook of Logic in Computer Science*, volume 2. Clarendon Press, 1992.

[37] H.T. Kung and C.E. Leiserson. Systolic array for VLSI. In Addison Wesley, editor, *Introduction to VLSI systems, C. A. Mead and L. A. Conway*, 1980.

[38] J. Mazoyer. Computations on one dimensional cellular automata. *Annals of Mathematics and Artificial Intelligence*, 16:285–309, 1996.

[39] J. Moreira. Memory models for the Bluegene/L Supercomputer. http://www.dagstuhl.de/03431/Proceedings/, 2003.

[40] R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler. A design space evaluation of grid processor architectures. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 40–51. IEEE Computer Society, 2001.

[41] A. Narayanan and S. Zorbalas. DNA algorithms for computing shortest paths. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 718–724, University of Wisconsin, Madison, Wisconsin, USA, 22-25 1998. Morgan Kaufmann.

[42] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. In *IEEE Computer*, pages 52–60, 1991.

[43] G. M. Papadopoulos and D. E. Culler. Monsoon: an explicit token-store architecture. In *Proceedings of the 17th annual international symposium on Computer Architecture*, pages 82–91. ACM Press, 1990.

[44] D. Parello, O. Temam, and J-M. Verdun. On increasing architecture awareness in program optimizations to bridge the gap between peak and sustained processor performance matrix-multiply revisited. In *Proceedings of the Proceedings of the IEEE/ACM SC2002 Conference*, page 31. IEEE Computer Society, 2002.

[45] G. Paun. Computing membranes. *Journal of Computer and System Sciences*, 1(61):108–143, 2000.

[46] J. A. Rose, Y. Gao, M. Garzon, and R. C. Murphy. DNA implementation of finite-state machines. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 479–490, Stanford University, CA, USA, 13-16 1997. Morgan Kaufmann.

[47] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *International Symposium on Microarchitecture*, pages 24–35, 1996.

[48] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ilp, tlp, and dlp with the polymorphous TRIPS architecture. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 422–433. ACM Press, 2003.

[49] V. Sarkar. Partitioning and scheduling parallel programs for multiprocessors. *MIT Press*, 1989.

[50] G. Sassatelli, L. Torres, P. Benoit, T. Gil., C. Diou, G. Cambon, and J. Galy. Higly scalable dynamically reconfigurable systollic ring-architecture for DSP application. In *Design Automation and Test in Europe Conference and Exhibition*, march 2002.

[51] H. Sharangpani and K. Arora. Itanium processor microarchitecture. *IEEE Micro*, 20(5):24–43, october 2000.

[52] P. W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *IEEE Symposium on Foundations of Computer Science*, pages 124–134, 1994.

[53] M. Sipper. Co-evolving non-uniform cellular automata to perform computations. *Physica D*, 92:193–208, 1996.

[54] R. Stadler, S. Ami, M. Forshaw, and C. Joachim. A memory/adder model based on single c_{60} molecular transistors. *Nanotechnology*, 12:350–357, 2001.

[55] A. Terechko, E. Le Thenaff, M. Garg, and J. van Eijndhoven. Inter-cluster communication models for clustered VLIW processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, pages 354–364, February 2003.

[56] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *Computational Complexity*, pages 179–196, 2002.

[57] C. D. Thompson. The VLSI complexity of sorting. *IEEE Transactions on Computers*, C-32(12):1171–1184, 1983.

[58] T. Toffoli. Programmable matter methods. *Future Generation Computer Systems*, 16(2–3):187–201, 1999.

[59] T. Toffoli and T. Bach. A common language for "programmable matter" (cellular automata and all that). *Bulletin of the Italian Association for Artificial Intelligence*, March 2001.

[60] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *25 years of the international symposia on Computer architecture (selected papers)*, pages 533–544. ACM Press, 1998.

[61] A. M. Turing. The chemical basis of morphogenesis. *Phil. Trans. Roy. Soc. of London*, Series B: Biological Sciences(237):37–72, 1952.

[62] P. M. B. Vitanyi. Locality, communication, and interconnect length in multi-computers. *SIAM Journal of computing*, 1988.

[63] J. Wahle, L. Neubert, J. Esser, and M. Schreckenberg. A cellular automaton traffic flow model for online simulation of traffic. *Parallel Computing*, 27(5):719–735, 2001.

[64] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. Chimaera: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 225–235. ACM Press, 2000.