Architectural Principles for One-Chip Parallel Computer

MIHAELA MALIŢA
Saint Anselm College
Department of Computer Science
100 Saint Anselm Dr., Manchester
New Hampshire, USA
mmalita@anselm.edu

GHEORGHE M. ŞTEFAN
Politehnica University of Bucharest
Faculty of Electronics
1-3 Maniu Bd. Bucharest
ROMANIA
gstefan@arh.pub.ro

Abstract: This paper starts from the programmer's view and states architectural principles for designing the one-chip many processor computer. John Backus's and Gary Sabot's old visions about how a parallel computer must be programmed are followed in the context of the current technologies. The proposed architectural principles are exemplified with the Connex chip, a 1024-cell SPMD engine able to provide 120 GOPS/Watt, 6 GOPS/mm^2 and 60 GOPS/$.

Key–Words: Parallel computation, computer architecture, parallel programming, functional programming, many-cell computer.

1 Introduction

Programming parallel engines is currently the main challenge of parallel computation. Even if a good programming model is provided, the problem of generating efficient code remains a big issue, because designing a transparent architecture is not an easy task. The bet on a good fit between a hardware structure and a programming model support the effort of defining a transparent architecture, the first condition for a friendly programming environment.

What means a good architecture? For a sequential engine is: the total transparency. For a parallel engine we are not yet in the position to provide a definitive answer. A tentative answer is possible. Maybe, we should give up some restrictions and accept to manage explicitly aspects related with the communication.

The paper starts from two solutions proposed, in an old different technological environment, by John Backus in 1978 [2] and Gary Sabot in 1988 [6]. We reconsider their approach because of the current technological improvements. Backus with its functional forms suggests the five forms of parallelism of an Integral Parallel Architecture, while the Paralation model of Sabot goes deeply makeing explicit distinction between computation and communication, exercising, in the same time, explicit control over locality.

The Connex Architecture [4] and the associated hardware, the Connex System [9] [10], seems to match pretty well the features requested by both, functional forms of Backus the paralations of Sabot.

2 The Functional Forms of Backus

The Functional Programming Systems (FP Systems) is proposed by John Backus as an alternative to the *von Neumann style of programming*. But, ***FP Systems can be seen also as a possible definition for parallel system from the programming point of view***. An FP System consists of *objects (atoms & sequences), functions* and *functional forms*. We claim that the functional forms define the way the parallel execution works in an Integral Parallel Architecture [10].

Apply to all is the functional form used to defines *data parallelism*. It is represented by:

$$\alpha f :< x_1, \ldots, x_p > \rightarrow < f : x_1, \ldots, f : x_p >$$

Usually, unary (x_i is an atom, and function f is `not`, `inc`, ...) and binary (x_i is a pair of atoms, and function f is `add, mult, compare,` ...) elementwise functions are performed. But, n-ary functions are also considered (if x_i is a sequence, then f could be `fft`).

Insert defines the *reduction parallelism*. The sequential recursive definition proposed by Backus

$/f :< x_1, \ldots, x_p > \rightarrow$
$f :< x_1, /f :< x_2, \ldots, x_p >>$

can be implemented in parallel by a *log*-depth tree. Frequently used functions are: `select`, defined by: $i :< x_1, \ldots, x_p > \rightarrow x_i$, which returns the i-th element of a sequence, `add` (returns the sum of the elements), `max` (returns the maximum value).

Construction defines the *speculative parallelism*, defined by:

$$[f_1, \ldots, f_p] : x \rightarrow < f_1 : x, \ldots, f_p : x >$$

There is a special case when $f_i(x) = g(i, x)$ when the construction is implemented as an *Apply to all* which uses the index sequence $ix =< 1, \ldots, p >$.

Threaded construction defines the *thread parallelism*. It is a form inspired from *construction*:
$\theta[f_1, \ldots, f_n] :< x_1, \ldots, x_p > \rightarrow$
$< f_1 : x_1, \ldots, f_n : x_p >$

Composition defines the *time parallelism* as a pipelined application of a sequence of functions, f_1,f_2, ..., f_q, to x, as follows:
$(f_1 \circ f_2 \circ \ldots \circ f_q) : x \equiv ((f_1 : f_2 : \ldots : (f_q : x) \ldots))$
Composition supports the transition from vector processing to stream processing.

Condition allows both, the conditioned sequence of operations applied on a stream of sequences (requested by the Single Program Multiple Data – SPMD [5]):
$(p \rightarrow f; g) : x \rightarrow$
$if\ ((p : x) = 1)\ f : x;\ else\ g : x;$
and the conditioned execution along a sequence:
$(p \rightarrow f; g) :< x_1, \ldots, x_p > \rightarrow$
$< (p \rightarrow f; g) : x_1, \ldots, (p \rightarrow f; g) : x_p >$

3 The Sabot's Paralation

The Paralation programming model proposed by Sabot for parallel architectures had an undeserved little impact in the world of parallel computing. It is an elegant approach for parallelism which uses only one data structure and three operators. We claim that in the domain of one-chip parallel computation the main concerns about Paralation model are removed.

The only data structure is the ***paralation*** (**paral**lel rel**ation**) which is a two-dimension array organized *horizontally* on **fields** and *vertically* on **sites**. It contains at least the `index field`. When a new paralation is initialized its first field index is created. For example, a 3-field paralation with 8 sites (the first field is the index field) is:

```
((0 1 2 3 4 5 6 7)
 (a b c d e f g h)
 (C B A E F D H G))
```

Elementwise evaluation is the first operation in the Paralation model. Let be the paralation:

```
P1=(index A B)=((0 1 2 3 4 5 6 7)
                (1 0 1 2 4 7 9 2)
                (4 7 9 2 1 0 1 2))
```

The conditioned elementwise evaluation `C = A<B ? A:B` generate a new field in `P1`:

```
P1=(index A B C)=((0 1 2 3 4 5 6 7)
                  (1 0 1 2 4 7 9 2)
                  (4 7 9 2 1 0 1 2)
                  (1 0 1 2 1 0 1 2))
```

In the approach proposed by Backus the elementwise evaluation is *Apply to all*.

Move is a parallel assignment which moves the content of one field to another. The general form of this operation specifies which element in the source field goes into which element in the destination field. The simplest mapping is when each element in the source field keeps its position unchanged in the destination. Another mapping is specified by a permutation field, as in the following example, where S is the source, P is the permutation and D is the destination:

```
P1=(index S P D)=((0 1 2 3 4 5 6 7)
                  (a b c d e f g h)
                  (4 7 3 2 1 5 0 6)
                  (e h d c b f a g))
```

Match is the third operator. Mapping is a partial and multi-valued operation. Default values are provided by a default field (DF). A general mapping combines sometimes some elements into a single one using a *combining function* (`add` in our example). Two fields, mapping source (MS) and mapping target (MT), are used to define the mapping. Let us illustrate the mapped move using the match operation:

```
(MS MT DF S D)=((a b c d a b e a)
                (b e c f d c)
                (d d d d d d)
                (3 6 4 1 7 2 2 5)
                ((6+2=8) 2 4 d 1 4))
```

To the first element in MT point the 2nd element and the 6th element from MS, then the first element in D is the sum between the 2nd element and the 6th element from S. To the second element in MT points only the 7th element from MS, then the second element in D is the 7th from S, and so on. The mapping is partial, because the 4th element in D receives nothing from S, and is multi-valued because the first element in D receives two elements from S, which are *reduced* to one element using the combining function `add`.

4 Connex Architeecture

4.1 The hardware

Connex System (Figure 1) is a one-chip many-core parallel engine, designed for general purpose or embedded applications. It consists of:
Linear Array of Cells : a p-cell one-dimension array; each cell contains a processing element with local memory and is connected to the left and the right cell.
Controller : a RISC processor which issues in each cycle an instruction, accompanied by an atom and/or an address if needed, received by each cell in the array; it runs also the sequential code.
Trans : is the transfer unit used to exchange, transparently to the computation process, scalar vectors with the external memory.
Broadcast : is a log-dept pipelined tree used to distribute to each cell the triplet {`instruction, address, atom`} issued by the Controller.

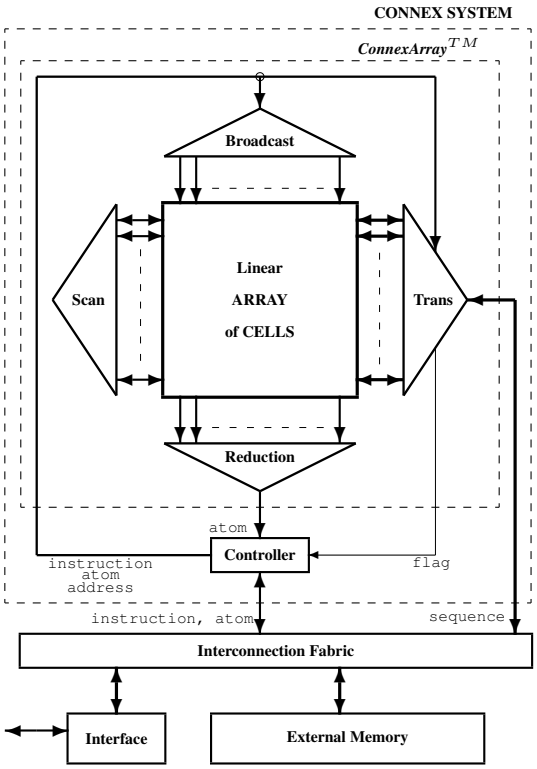

Figure 1: The Connex System.

Reduction : performs reduction operations on the atoms issued by each active cell.
Scan : is the global loop closed over the array; it processes vectors received from the array.

4.2 The user view of the system

For the user of the system a simple image is provided. It is kept as simple as possible, "but not simpler", in order to help the programmer to optimize its approach. It consists in the internal memory – a two-dimension array of scalars – and the external memory. The notations used for the internal memory of vectors are:
$ix =< 1, \ldots i, \ldots p >$
$v_0 =< s_{11}, s_{12}, \ldots s_{1i}, \ldots s_{1p} >$
$v_1 =< s_{21}, s_{22}, \ldots s_{2i}, \ldots s_{2p} >$
$\ldots$
$v_j =< s_{j1}, s_{j2}, \ldots s_{ji}, \ldots s_{jp} >$
$\ldots$
$v_{n-1} =< s_{n1}, s_{n1}, \ldots s_{ni}, \ldots s_{np} >$
$a =< b_1, b_2, \ldots b_i, \ldots b_p >$
while for the external RAM:
$s =< s_0, s_1, \ldots s_i, \ldots s_{m-1} >$
where:

- ix is a constant vector called index; each cell is identified by one component of this vector
- $v =< v_1, v_2, \ldots v_j, \ldots v_n >$ is a two-dimension array of $p \times n$ scalars; each line v_j, called ***horizontal vector***, is distributed along the cells (one component per cell), while each column, $w_i =< s_{1i}, s_{2i}, \ldots s_{ji}, \ldots s_{ni} >$, ***called vertical vector***, is stored in the local memory of the i-th cell
- a is the activation vector of Booleans distributed along the cell; the i-th cell is active if $b_i = 1$
- s is the external memory of scalars.

4.3 Connex ISA

The instruction set of Connex (Connex ISA) is defined in SCHEME. A vector is always a horizontal one. Connex ISA is partially described here, with emphasis on the functionality related with our purpose:

(Store aAddr mAddr) : store the vector v_{aAddr} in the external memory starting with s_{mAddr}.

(Load aAddr mAddr) : load the vector v_{aAddr} with atoms from the external memory starting with s_{mAddr}.

(*unary*Op vector) : is an element wise unary operation (`unary` = {`Inc`,`Dec`,`Not`,...}).

(*binary*Op op1 op2) : is an element wise binary operation (`binary` = {`Add`,`Mult`,`Div`,`And`,...}); at least one of `op1` or `op2` is a vector.

(*red*Op vector) : is the reduction operation (`Op` = {`Add`,`Max`,...}) which returns the `Op` operation applied on the active atoms of the vector `vector`.

(*test* x y) : is a test operation (`test` = {`Eq`,`Lt`,`Gt`,...}) which returns a Boolean vector; at least one of `x` or `y` is a vector.

(Where vector) : acts on active cells and sets inactive all the cells where the vector `vector` is 0.

(ElseWhere) : restores the state of cells before the last `Where` and applies `Where` with the complemented vector `vector` used by the last `Where`.

(EndWhere) : sets active all the cells inactivated by the last `Where` or the last `ElseWhere`, i.e., restore the activation shape modified by the last `Where`.

(Permute vector1 vector2) : permutes the vector `vector1`, according to the indexes provided by the vector `vector2`; performed in the Scan module.

(Search element vector) : on the active components of the vector `vector` search the atom `element`; only the components equal with `element` remain active.

4.4 In silicon validation of Connex

The system described is validated by three silicon implementations [8]. The last one in 2008, is BA1024, a 1024-cell array of 16-bit cells, using standard cell design for 65 nm. BA1024 was designed to run at 400 MHz for HDTV market and dimensioned for decoding two H264 HD streams. It provided 120 GOPS/Watt (measured on real chips running frame rate conversion programs), 6 GOPS/mm^2 and 60 GOPS/$ (GOPS: 16-bit Giga Operations Per Second).

Compared with IBM Cell BE , AMD/ATI 2900 or NVIDIA 8800 GPU, the Connex architecture provide much more GOPS/Watt, GOPS/mm^2, GOPS/$ with a simpler organization and the simplest programming environment. The main reason is the theoretical distinction and actual segregation made in the Connex approach between complex computation and intense computation (more details in [9]).

5 Functional Forms on Connex

All the five forms of parallelism (data-parallelism, reduction-parallelism, speculative-parallelism, time-parallelism, and thread-parallelism) which result from the seminal paper of John Backus, are supported efficiently by a Connex-like architecture. When the composition functional form is added, the most promising form of one-chip parallelism are supported: the stream processing executed on the SPMD model. Indeed, the forms of an IPA are supported as follows:

- data-parallelism is supported mainly by (*unary*Op vector) and (*binary*Op op1 op2)
- reduction-parallelism is supported by RedOp vector
- speculative-parallelism is supported by index dependent executions or running locally stored code
- time-parallelism is supported by an enough big local memory able to store intermediary results, besides of organizing local buffers of data, thus allowing a true stream execution in SPMD mode
- thread-parallelism is supported mainly by the multi-threaded features implemented at the Controller level (the local memory in each cell can be used for multi-threaded executions)

Because almost any real application requests all forms of parallelism, for one-chip parallel computing an IPA guaranteed by Backus is a must.

6 Paralation on Connex

The low-level, architectural model provided by Backus is detailed by the Paralation Model proposed by Sabot. Both provide a solid foundation for what the one-chip parallelism must be.

A paralation is represented on Connex Architecture by the index vector ix and the content of a subset of horizontal vectors from v. For example:
$< ix, v_3, v_4, v_5 >$
is a paralation with 4 fields. Each site is a vertical vector of the following form:
$< i, s_{3i}, s_{4i}, s_{5i} >$
which is associated to the i-th cell. Connex offers a direct implementation for a paralation, because each horizontal vector supports one or few fields and each vertical vector supports one or few sites.

6.1 Elementwise evaluation operator

The elementwise evaluations are performed in Connex System by operations (*unary*Op vector) and (*binary*Op op1 op2). The conditioned elementwise evaluations use (Where vector), (ElseWhere) and (EndWhere) operations. For example:

```
(Where (Lt v3 v4))  ; Lt is less than
   (SetVector v5 v3)
 (ElseWhere)
   (SetVector v5 v4)
(EndWere)
```

6.2 Move operator

Move & match operations are strongly related. The Connex System performs these operations more or less efficiently, depending on the structural resources it involves. The simplest move operation in the Paralation model, let us call it *direct move*, is performed in Connex System by moving the content of v5, as it is, in v3:

```
(SetVector v3 v5)
```

A more complex way to move data from a field to another is to rearrange the components in the destination field according to a specified permutation. Let us call it *permuted move* operation. Permuted move is defined by a permutation vector/field v5, used to permute the content of v4 with the result in v3:

```
(SetVector v3 (Permute v4 v5))
```

The hardware resources to perform this operation determines the execution speed. The fastest version, provided by a solution which uses a Beneš-Waxman permutation network, provides a time in $O(log\, n)$, where n is the length of the field. The circuit size is in $O(n \times log\, n)$. Taking into account the frequency of

this kind of move, the cost is too big for a general purpose engine, because for $n > 1024$ this size competes with the size of the execution units distributed in the array. Next section will provide an efficient solution in the context of Connex Technology.

6.3 Match operator

For the most complex move, the *matched move*, which uses the match operation to provide the mapping, let us revisit the example already provided in section 3. Let be the initial state in Connex System with the destination vector/field filled up, for the sake of simplicity, with default value $d = 0$:

```
(MS S MT D)=((a b c d a b e a)
            (3 6 4 1 7 2 2 5)
            (b e c f d c)
            (0 0 0 0 0 0))
```

The informal description of the algorithm running on Connex System is:

1. the i-th element from the field MS is searched in MT (using `Search` operation, if only atoms are involved or `Search` and `CondSearch` operations if the elements of the field are sequences of atoms)

2. in all the matched *sites* the field D is incremented with the value stored in the i-th site of the field S

3. the previous two steps are performed for each component of the field MS.

For our example, the content of the vector/field D evolves as follows:

```
inital    (0 0 0 0 0 0)
step 1    (0 0 0 0 0 0) // no 'a' in MT
step 2    (6 0 0 0 0 0)
step 3    (6 0 4 0 0 4)
step 4    (6 0 4 0 1 4)
step 5    (6 0 4 0 1 4) // no match
step 6    (8 0 4 0 1 4)
step 7    (8 2 4 0 1 4)
step 8    (8 2 4 0 1 4) // no match
```

The advantage of the Connex implementation is that the mapping defined by the *from-field* MS to the *to-field* MT is "extracted" and "executed" in the same time, so as the generation of the explicit forms for the fields *to-key* and *from-key* (see [7]) is avoided. The search capabilities of the Connex System are used to accelerate the match operation, allowing a linear dependency of the execution time by the length of the data moved using the mapping mechanism.

7 Why Paralation, Backed by Backcus's Functional Forms?

There are various reasons for using Connex as the target architecture for any Paralation Model implementation in one-chip parallel (embedded) applications, despite the fact that the model failed in the 1980's to be successful. The failure was due technological limitations. One-chip many core approach was at that moment only a dream. We consider that the Paralation Model, backed by the Functional Forms emphasized by Backus, benefits now from a completely new technological environment. A one-chip many core can be the perfect fit for reconsidering Paralation Model, proposed by Sabot in 1988, in the larger context of FP Systems proposed by John Backus in 1978. Few short motivations are listed bellow.

Data structure is a natural match because the structure of a Sabot's paralation or Backus's sequences are directly implemented in Connex System. The field corresponds with the horizontal vector, and the site with the vertical vector.

***Apply to all* or elementwise evaluation operation are Connex operations** . In a distributed, multichip or multi computer system this kind of approach is so inefficient that it does not make sense. It was a strong reason for the failure of Sabot's proposal.

Permuted move operation is supported in Connex depending on the application domain.

For applications which use this operation frequently, in the *Scan* module (see Figure 1) a Beneš-Waxman permutation network can be implemented for an array having no more than 1024 cells. A bit-serial version could be efficient for bigger arrays. But, the Connex approach offers the opportunity to perform, with no cost, the permutation on a vertical vector, inside a site, instead of performing the permutation on a horizontal vector, inside a field.

Example: Let be the computation of 256 256-sample FFTs performed on a 256-cell Connex System. There are two solutions to make the computation. The first solution loads the samples for each FFT as fields/horizontal vectors, while the second loads the samples for each FFT as sites/vertical vectors. The first, horizontal computation working at one FFT at a time, performs all the additions and multiplications in parallel very efficiently, but, after each stage of add & multiply, the resulting field/vector must be permuted. The permutations request big hardware or big time. The second, vertical computation working on 256 FFTs at a time, performs the add & multiply stage with the same efficiency, but the permutation is avoided because in each site the components of the vertical vectors can be randomly accessed in the local memory of each cell.

Experimental results show [3] that for even a small number of samples, 64, the vertical solution is 4.3 times faster than the horizontal one in floating point applications, while in fix point is 11 times faster.

But, what can be done for a big number of samples? An intermediate solution is possible, because the principle of the Cooley-Tukey algorithm allows to compute a FFT of composite size $N \times M$ in terms of FFTs of size N and M. We can use the structure of the 2D FFT algorithm, with a twiddle multiplication between the vertical and horizontal stages. 1D FFT of size $N \times M$ is computed in three steps: (1) compute N vertical FFTs of size M, (2) multiply the array with twiddle factors, (3) compute M horizontal FFTs of size N. Instead of step (3) which supposes the inefficient horizontal FFT computation, the array is ***transposed*** and only then M vertical FFTs are computed.

In [3] 1024-sample floating point FFTs are computed for horizontal solution and for a 2D shape of 32×32 samples. The acceleration obtained for the 2D compared to the horizontal is 7.3.

Transforming fields in sites, or horizontal vectors in vertical vectors, is a good solution to avoid the slowdown induced by permutations. The transpose function helps to make this transformation. It is easy to optimize this function with few specific circuits.

Matched move is based on Connex's search capabilities represented by the fast search operations. The unconditioned or conditioned search of an atom/element in a vector/field is done in one clock cycle, offering the best conditions for accelerating the matched move. Multiple-write in one cycle allows to move more than one atom/element at a time from one field to another. When the mapping provides multiple vectors pointing to the same cell, we can decide to use the log-time reduction functions of the Connex System.

8 Conclusion

The design of a parallel one-chip computer must start from the image the programmer have about the use of the resulting engine. The image provided by Backus and Sabot was used to impose the architectural features of the one-chip many core engine Connex.

The play between *field* and *site* is benefic for optimizing the communication in a Connex System. In fields/horizontal vectors there is the simplest, but no cost, the locality, while in sites/vertical vectors any locality is possible with the cost of running a sequence of code.

Optimizing the transpose operation minimizes the communication overhead in the system. For this reason the operation is performed, transparently to the main processing.

The impact of Functional Forms and Paralation will grow once implemented for a one-chip parallel engine, because a lot of costly actions in distributed hardware are removed on a compact structure like Connex. Paralation means both, computation and communication. The second one introduces big overheads Paralation is not responsible for, but it paid for with an undeserved little impact in the last 24 years. We hope Paralation backed by Functional Forms implemented in Connex technology will lift up each other.

Acknowledgements: The author got a lot of support from the main technical contributors to the development of the *ConnexArray*^{TM} technology, the *BA1024* chip, the associated language, and its first application: E. Altieri, F. Ho, B. Mîţu, M. Stoian, D. Thiebaut, T. Thomson, D. Tomescu.

References:

[1] K. Asanovic, et. al., *The Landscape of Parallel Computing Research: A View from Berkeley*, Technical Report No. UCB/EECS-2006-183.

[2] J. Backus, Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs, *Communications of the ACM*, August 1978, 613-641.

[3] I. Lorentz, M. Maliţa, R. Andonie, Fitting FFT onto an Energy Efficient Massively Parallel Architecture, *The 2nd Int. Forum on Next Generation Multicore/Manycore Tech.*, June, 2010.

[4] M. Maliţa, G. Ştefan, Parallel RISC Architecture. A Functional Approach Based on Backus's FP language, *Proceedings of the 2011 International Conference on PDPTA*, Las Vegas, 2011, pp 492-498.

[5] M. McCool, *Scalable Programming Models for Massively Multicore Processors*, *Proceedings of the IEEE*, Vol. 96, No. 5, May 2008.

[6] G. Sabot, *The Paralation Model. Architecture-Independent Parallel Programming* (Cambridge, Massachusetts: The MIT Press, 1988).

[7] P. Snively, The Paralatioin Model, *The Journal of Apple Technology*, vol. 8, no. 7, 1992.

[8] G. Ştefan, *The Connex Project*, at http://arh.pub.ro/gstefan/conexMemory.html

[9] G. Ştefan, One-chip teraArchitecture, *Proceedings of the 8th Applications and Principles of Information Science Conference*, Okinawa, 2009.

[10] G. Ştefan, Integral Parallel Architecture in System-on-Chip Designs, *The 6th International Workshop on Unique Chips and Systems*, Atlanta, GA, USA, December 4, 2010, pp. 23-26.