ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY
Volume **12**, Number 1, 2009, 35–49

The *Berkeley Motifs* and an Integral Parallel Architecture

Mihaela MALIŢA
Saint Anselm College, NH.
E-mail: `mmalita@anselm.edu`

Gheorghe ŞTEFAN
Politehnica University of Bucharest, Romania *&* BrightScale, Sunnyvale, CA
E-mail: `gstefan@arh.pub.ro`

Abstract. The Integral Parallel Architecture (IPA) developed by ***BrightScale*** is a low-power & low-area one-chip solution to solve *intense* computational problems using data-parallel, time-parallel and speculative-parallel mechanisms. ***BrightScale*** technology is presented and analyzed from the point of view of each of the 13 computational motifs proposed in *The Berkeley's View* [1]. IPA emerges from the Stephen Kleene's computational model of the partial recursive functions [3] as *the simplest* parallel architecture, a good starting point for a true science of parallel computation. We briefly investigate how such an *elementary parallel architecture* performs, for the main computational motifs, in solving the problems of programmability, portability, flexibility, data movement between computational cells, and between cells and the main memory.

1. Introduction

While the mono-processor architectures are very well rooted in the Turing's computational model, multi- and many-processor architectures do not have a similar foundation in a well grounded theoretical model. Rather, all the solution provided for parallel computation – shared memory multiprocessors, distributed memory multiprocessors, vector systems, massively parallel processors, SIMD systems, symmetric multiprocessors – look like a non-formal extension, obtained by interconnecting more than one processing units in networks having *ad-hoc* or too general topologies. We have a lot of parallel organizations, but we are not yet able to provide a *true parallel* *architecture* which makes transparent to the programmer the actual organization of the machine – the nightmare of any application developer. In fact, we are waiting for a true science of parallel computing.

In [8] & [5] is for first time suggested the foundation of parallel computation based on the Kleene's [3] computational model. The partial recursive functions model is a promising starting point in defining an ***elementary parallel architecture***. This elementary concept was called **Integral Parallel Architecture** (IPA) and its simplest associated organization was described.

In this paper, the concept of IPA is revisited in order to be used to describe its first implementation: the ***BrightScale***'s technology, conceived as an accelerator for computational intense applications. Then, the technology is evaluated against the 13 computational motifs emphasized in *The Berkeley's View* [1].

Because the ***BrightScale***'s technology has a very simple, maybe the simplest, implementation possible for a parallel machine, the main goal of our approach is to make preliminary investigations in order to understand how far is this *elementary parallel organization* from a machine able to solve efficiently all the 13 computational motifs considered by the Berkeley's seminal report.

2. The Simplest Parallel Architecture

In [5] the Kleene's computational model is proposed as starting point for a true parallel computing science, and is proved that the partial recursive functions are computed using only various forms of composition (primitive recursive and minimalization rules are forms of the composition rule). In this theoretical framework IPA is defined (see [9] [4])as an architecture featured with three kinds of parallel mechanisms:

data-parallel computing : this form uses operators that take vectors as arguments and returns vectors, scalars (by reduction operations) or streams (input values for time-parallel computations). This form is very similar to Flynn's SIMD machine [2].

time-parallel computing : this form uses operators that take streams as arguments and return streams, scalars, or vectors which can be used as input values for data-parallel computations. This form is akin to MIMD machines, but refers to the computation of a single function (described by a vector of functions) instead of multi-threading computation.

speculative-parallel computing : in this form, operators take scalars as arguments and return vectors reduced to scalars using selection. This form is used mainly to speed up time-parallel computations, and it contains a true MISD-like structure. This form has no real implementations in Flynn's taxonomy.

In these context an actual structure with IPA is built starting from the general form of composition:

$$f(x_0, \ldots x_{n-1}) = g(h_0(x_0, \ldots x_{n-1}), h_1(x_0, \ldots x_{n-1}), \ldots h_{m-1}(x_0, \ldots x_{n-1}))$$

implemented as an actual system as it is represented in Fig. 1, where a computing **cell** h_i is used for each function $h_i(x_0, \ldots x_{n-1})$ and for the reduction function $g(y_0, \ldots y_{m-1})$ specific network g is considered.

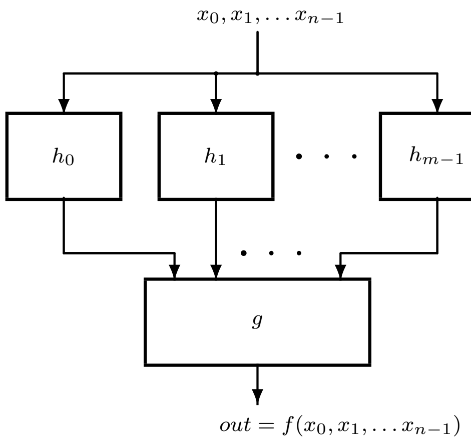
$x_0, x_1, \ldots x_{n-1}$
h_0
h_1
h_{m-1}
g
$out = f(x_0, x_1, \ldots x_{n-1})$

Fig. 1. The physical structure associated to the composition rule. The composition of the function g with the functions $h_0, \ldots, h_{m-1}$ implies a two-level system. The first level, performing in **parallel** m computations is **serially** connected with the second level which performs a reduction function.

From the theoretical model (see Fig. 1) to an actual machine few specific features must be added. These features must allow the following pairs of limit cases:

1. **data-parallel** computation vs. **time-parallel** computation, where:
   - data-parallel means $h_i = h$ for $i = 0, \ldots n-1$, the input is a stream of vectors $\{X_j\} = \{x_0, \ldots x_{n-1}\}$, and the output is another stream of vectors $\{Y_j\} = \{y_0, \ldots y_{n-1}\}$
   - time-parallel means $n = 1$ and the computation looks like a pure pipelined process ($f = g(h_0)$) having the general form of $f([X]) = f_0(f_1(f_2(\ldots f_{p-1}([X])\ldots)))$, where $[X] = [x_0, \ldots x_{n-1}]$ is the input stream of scalars, resulting as output another stream $[Y] = [y_0, \ldots y_{n-1}]$
2. strong data-**locality** vs. **non-locality**, where:
   - strong data-locality means each cell uses only one component of the impute vector, $h_i = h(x_i)$, and sometimes data from a small neighborhood (example: $h_i = h(x_{i-2}, x_{i-1}, x_i, x_{i+1}, x_{i+2})$)
   - non-locality means that each cell will use in one stage of the computation all components of each input vector or at least data resulting from processing each component of each vector (example: FFT)

3. **locally predicated** vs. **globally predicated** execution, where:
   - locally predicated means each cell is featured with a Boolean machine used to manage a finite stack of predicates used to decide how to execute each received instruction
   - globally predicated means that the entire array of cells is catched in a global loop which receives a selected predicate from each cell and sends back to each cell information about the state of the entire array of cells.

Results the ***elementary parallel organization*** represented in Fig. 2, where it is easy to recognize the shape of the theoretical composition circuit (see Fig. 1). But, some additional structure are necessary to obtain a real machine, thus an EPO will be composed by:

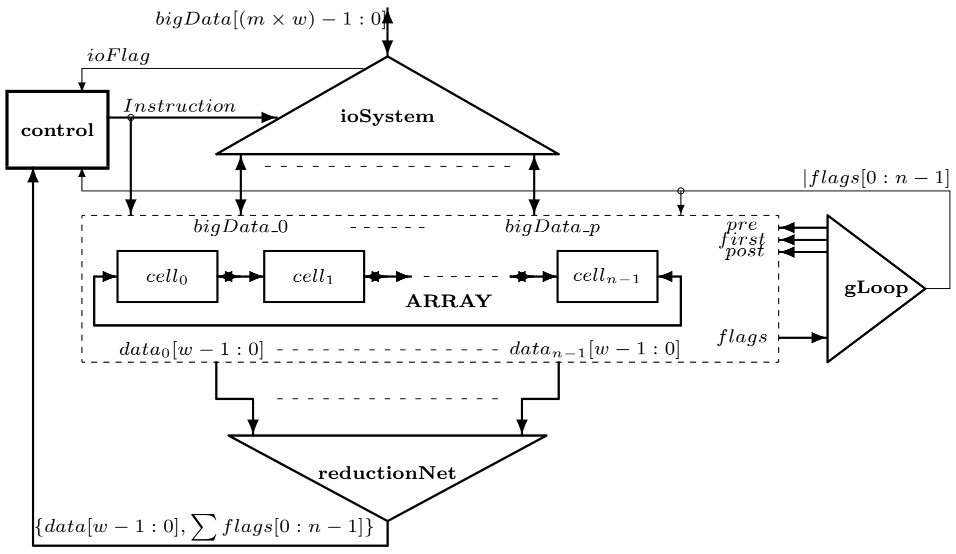

Fig. 2. Elementary parallel organization. Where:
$bigData\_0 = bigData_0[(m \times w) - 1 : 0], \ldots,$
$bigData\_p = bigData_{(\sqrt{n}/m)-1}[(m \times w) - 1 : 0],$
and *pre, first, post, flags* are n-bit connections.

ARRAY : a linearly connected array of n cells each containing an w-bit ALU, a file register, a Boolean machine, and a local memory used:

- to store local data, when the cell is used as an execution unit (EU) for data-parallel computation
- to store local data and the local program, when the cell is used as a processing element (PE) for time-parallel computation

ioSystem : transfers $(m \times w)$-bit data between the array and the external (memory) system; for each input vector n/m clock cycle are used (the *latency* of the system can be "absorbed" if a big number of vectors are transferred); local buffers can be used to make *in-fly* permute on the transferred data

reductionNet : extracts for the use of the **control** module various data from the array

gLoop : takes form each cell the selected predicate and returns to each cell four independent bits: `first` (the cell is the first with the selected predicate on 1), `pre` (the cell is positioned before the *first* cell), `post` (the cell is positioned after the *first* cell), `|flags` (at least one cell has the selected predicate on 1); the last bit is sent also to **control**

control : sends in each clock cycle an instruction to array, and it is used to configure the module **ioSystem**; it contains the program memory for data-parallel processing.

BrightScale's already implemented in $65nm$, using standard-cell library, a version of the above described structure. In this approach data-parallel section, with $n = 1024$, is implemented separated from the time-parallel section, with $n = 8$. The main parameters are:

computation:	400 GOPS[1] at 400 MHz (peak performance)
external bandwidth:	6.4 GB/sec (peak performance)
internal bandwidth:	800 GB/sec (peak performance)
power:	< 3 Watt
area:	$< 50mm^2$.

3. *BrightScale*'s Integral Parallel Architecture

While IPA performs data-parallel, time-parallel and speculative parallelism, almost all parallel computations are data parallel, and only some of them involve time parallel processes supported by speculative computations, if needed.

3.1. Data parallel engine

The computational structure performing data parallel computation is a fine-grain network of *small & simple* execution units (EU) working as a many-cell machine.

BrightScale many-core section (see Figure 3) is a linear array of 1024 EUs. It provides the **spatial dimension** of the array. Each EU is a 16-bit machine, with a 1 KB local data memory. This memory allows to store in the array 512 1024-component

[1]16-bit Giga Operations Per Second

vectors, generating the **temporal dimension** of the array. The processing array works in parallel with an IO plan (IOP) used to transfer data between the array and the external memory. The array is controlled by the stack machine Instruction Sequencer (IS), while the IOP transfers data under the control of a similar machine called IO Controller (IOC). Thus, data processing and data transfer are two independent processes performed in parallel. Data exchange between the processing array and the IO plan is performed in one clock cycle and is synchronized by interrupt mechanisms defined between the two controllers, IS and IOC.

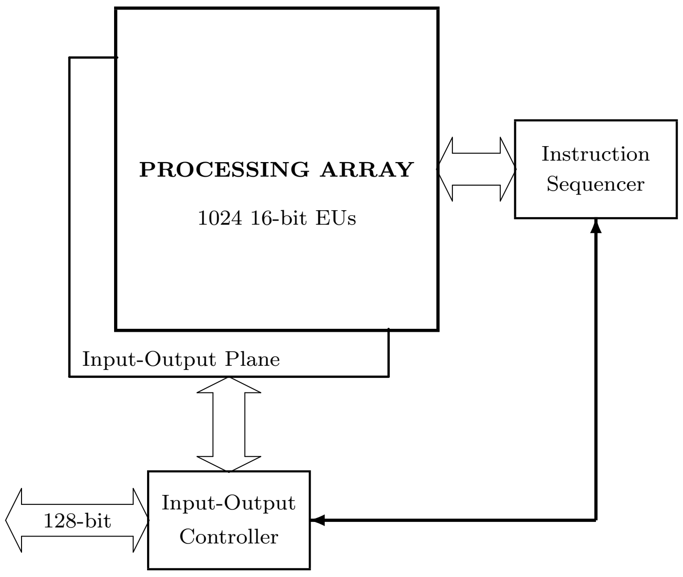

Fig. 3. ***BrightScale*** **data parallel engine.** The processing array is paralleled by the IO Plane which performs *data transfers* transparent to the *processing.*

For time parallel computation a dynamically reconfigurable network of 8 PEs is provided.

Speculative computation is performed in both networks.

3.2. Data parallel architecture

The user's view of the data parallel machine is represented in Figure 4. A linear cellular machine containing 1024 EUs (the *spatial dimension* of the architecture), each storing in its data memory up to 512 scalars (the *temporal dimension* of the architecture), performs parallel, constant number of cycle operations on randomly accessed vectors. Some generic operations are exemplified in the following:

PROCESSING OPERATIONS performed in the processing array under the control of IS

full vector operation: {`carry, v5`} `= v4 + v3;`
the corresponding integer components of the two operand vectors (`v4` and `v3`) are added, and the result is stored in the result vector `v5` and in the Boolean vector `carry`

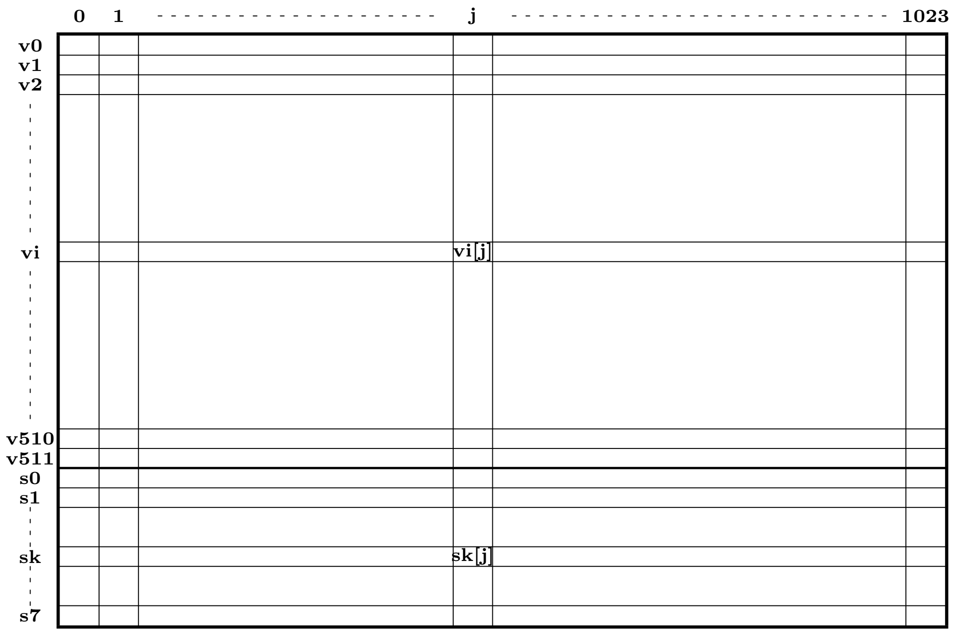

Fig. 4. The internal state of *BrightScale* data parallel machine. There are 512 integer vectors, each having 1024 16-integer components (`vi[j]` is a 16-bit integer), and 8 selection vectors, each having 1024 Booleans (`sk[j]` is a Boolean).

Boolean operation: `s3 = s3 & s5;`
the corresponding Boolean components of the two operand vectors are ANDed and the result is stored in the result vector

predicated execution: `v1 = s2 ?  v3 - v2 :  v1;`
in any positions **where** `s2 = 1` the corresponding components are operated, while in the rest (i.e., **elsewhere**) the content of the result vector remains unchanged (it is a *"spatial" if*)

vector rotate: `v7 = v7 >> n;`
the content of vector `v7` is rotated `n` positions right, i.e.,
`v7[i] = v7[(i+n)mod1024]`

INPUT-OUTPUT OPERATIONS performed in IOP under the control of IOC

strided load: `load v5 address burst stride;`
the content of `v5` is loaded with data from the external memory accessed starting from the address `address`, using bursts of size `burst`, stridden with `stride`

scattered load: `sload v3 high_address (v9 address stride);`
the content of `v3` is loaded with data from the external memory indirectly accessed using the content of the **address vector** `v9`, whose content is used starting from the index `address`, strided with `stride`; the address vector is structured in pairs of 16-bit words; each of the 512 resulting 32-bit word is organized as follows:
{`dummy, burst[5:0], address[24:0]`}
where: if `dummy == 1`, then a burst of {`burst[5:0], 1'b0`} dummy bytes are loaded, else a burst of data from the address {`high_address, address, 1'b0`} is loaded (*indirect* load)

strided store: `store v7 address burst stride;`

gathered store: `gstore v4 high_address (v3 address stride);`
(it is a sort of *indirect* store).

3.3. *VectorC*: the programming language for data parallel architecture

BrightScale's data parallel engine is programmed in ***VectorC***, a C language extension for parallel processing defined by ***BrightScale*** [6]. The extension is made by adding new primitive data types and by extending the existing operators to accept the new data types. In the VectorC programming language the conditional statements have become predication statements. The new data primitives are:

int vector: vector of integers (stored as a pair of 16-bit integer vectors)

short vector: vector of shorts (stored as a 16-bit integer vector)

byte vector: vector of bytes (two byte vectors are stored as a 16-bit integer vector)

selection: vector of Booleans

In order to explain how VectorC works let be the following variable declarations:

```
int i1, i2, i3;
bool b1, b2, b3;
int vector v1, v2, v3;
selection s1, s2, s3;
```

Then a VectorC statement like: `v3 = v1 + v2;` stands for:

```
for (int i = 0; i < VECTOR_SIZE; i++)
    v3[i] = v1[i] + v2[i];
```

and `s3 = s1 && s2;` stands for:

```
        for (int i = 0; i < VECTOR_SIZE; i++)
            s3[i] = s1[i] && s2[i];
```

The scalar statement: `if (b1) i3 = i1 + i2;` becomes in VectorC the following vector predication statement:

```
        WHERE (s1) {v3 = v1 + v2};
```

with the meaning:

```
        for (int i = 0; i < VECTOR_SIZE; i++)
            if (s1[i])
                v3[i] = v1[i] + v2[i];
```

Similarly, `i3 = (b1)?  i1 :  i2;` is extended to accept vectors:

```
        v3 = (s1)? v1 : v2;
```

Here is an example in VectorC computing the absolute difference of two vectors.

```
    vector absdiff(vector V1, vector V2);
    int main() {
        vector V1 = 2;
        vector V2 = 3;
        vector V;
        V = absdiff(V1, V2);
        return 0;
    }
    vector absdiff(vector V1, vector V2) {
        vector V;
        V = V1 - V2;
        WHERE (V < 0) {
            V = -V;
        }
        ENDW
        return V;
    }
```

3.3. Time parallel architecture

Time parallel computation is supported in ***BrightScale***'s technology by a small configurable network or processing elements called **Stream Accelerator** (SA). The network works like a pipe of processors in which in any point two or more machines are parallel connected to support speculation (see Figure 5). In the actual implementation 8 machines are used. Functions like CABAC decoding procedure, a highly sequential and strong data dependency computation, are efficiently executed.

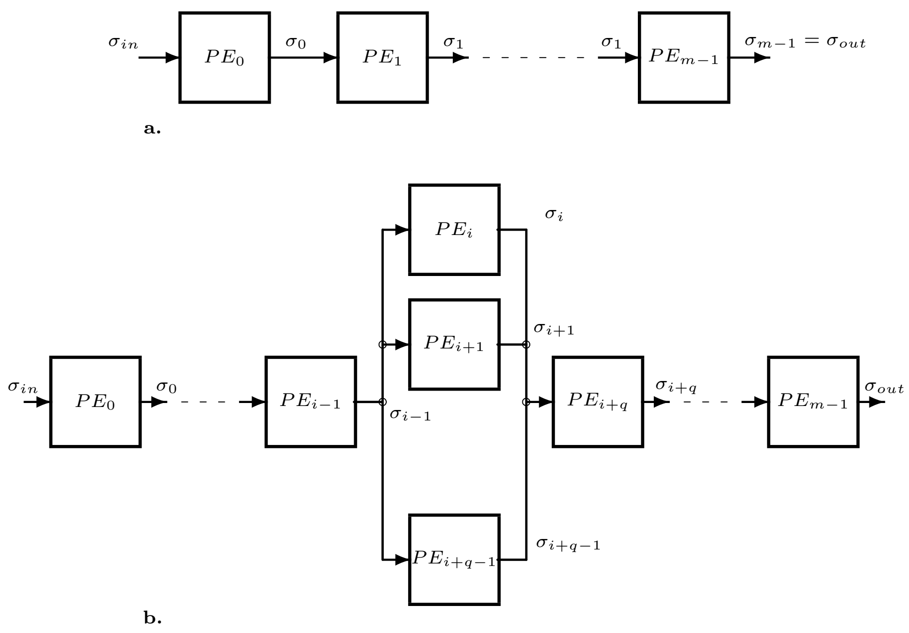

Fig. 5. ***BrightScale* time-parallel engine. a.** The pipe without speculation. **b.** The pipe with speculation. The i-th stage in pipe is computed by q PEs dynamically configured in a speculative linear array. PE_{i+q} selects dynamically as input only one of the outputs of the speculative array.

The computation of SA is defined by two mechanisms:

stream of functions containing m programs $p_j(\sigma)$:

$$S(\sigma_{in}) = < p_0(\sigma_{in}), p_1(\sigma_0), \ldots p_{m-1}(\sigma_{m-2}) >= \sigma_{out}$$

applied to a stream of scalars σ_{in}, generating a stream of scalars σ_{out} as output, where: $p_j(\sigma)$ is a program which processes the stream σ and generates the stream σ_j; *it is a sort of MIMD computation*

vector of functions containing q programs $p_j(\sigma)$:

$$V(\sigma_{in}) = [p_0(\sigma_{in}), \ldots p_{q-1}(\sigma_{in})]$$

applied to a stream of scalars σ_{in}, generating a stream of q-component vectors; *it is a true MISD computation.*

The latency introduced by a stream of functions is m, but the stream is computed in real time. Vector of functions are used to perform speculation when the computation requests it in order to preserve the possibility of real time computation. For example:

$$< p_0(\sigma_{in}), p_1(\sigma_0), \ldots p_{i-1}(\sigma_{i-2}), V(\sigma_{i-1}), p_{i+q}(\sigma_?), \ldots p_{m-1}(\sigma_{m-2}) >$$

is a computation which performs a speculation in the stage i of the pipe. The program $p_{i+q}(\sigma_?)$ selects from the vectors generated by $V(\sigma_{i-1})$ only one component as input.

4. Berkeley's View & ***BrightScale***'s Performance

Berkeley's View [1] provides a comprehensive presentation of the problems to be solved by the emerging actor on the computing market: the ubiquitous parallel paradigm. Many decades an academic topic, parallelism becomes an important actor on the market after 2001 when the clock rate race stopped. This research report presents 13 computational *motifs*[2] which cover the main aspects of the parallel computing. They are defined unrelated with a specific parallel architecture. In the next section we will make a preliminary evaluation of them in the context of ***BrightScale***'s IPA.

BrightScale's cellular network has the simplest possible interconnection network. This is both an advantage and a limitation. On one hand, the area of the system is minimized, and it is ease to hide the associated architecture to the user, with no lose in programmability and in the efficiency of compilation. The limitation acts depending on the application domain. Follows short comments about how the ***BrightScale*** architecture works for each of the 13 *Berkeley's View* motifs.

4.1. ***Motif*** 1: Dense linear algebra

The computation in this domain operates mainly on $n \times m$ matrixes. The operations performed are: matrixes addition, scalar multiplication, transpose of a matrix, dot product of vectors, matrixes multiplication, determinant of a matrix, (forward & backward) Gaussian elimination, solving systems of linear equations, inverse of a $n \times n$ matrix.

Depending on the product $n \times m$ the internal representation of the matrixes is decided. If the product is small enough (usually, no bigger than 128), each matrix can be associated to one EU, resulting 1024 matrixes represented by $n \times m$ 1024-element vectors. But, if the product $n \times m$ is big, then p EUs are associated with each matrix, resulting $1024/p$ matrixes represented on $(n \times m)/p$ 1024-element vectors.

For all the operations above listed the computation is usually accelerated 1024 times, and no time it is under $(1024/p)$ times. This is possible because special hardware is provided for reduction operations (for example: adding 1024 16-bit numbers takes 20 clock cycles).

4.2. ***Motif*** 2: Sparse linear algebra

There are two types of sparse matrixes: (1) randomly distributed sparse arrays (represented by few types of lists), (2) band arrays (represented by a list of vectors).

[2]Initially called *dwarfs*, they are renamed as *motifs* in [7].

For small random sparse arrays, converting them internally into dense array is a good solution. For big random sparse arrays the associated list is operated using efficient search operations. For band arrays systolic-like solution are proposed.

4.3. *Motif* 3: Spectral methods

The typical examples are: FFT or wavelet computation. Because of the "butterfly" data movement FFT computation is implemented depending on the length of the sample. The *spatial* and the *temporal* dimensions of the processing array helps in adapting the data representation in order to achieve an almost linear acceleration. For example: 32 1024-sample FFTs can be processed in parallel, because the initial real samples are stored in 32 vectors, so as for each FFT 32 EUs are allocated. Each set of 1024 samples represents a 32×32 array in the internal state of the ***BrightScale*** architecture (see Figure 4).

4.4. *Motif* 4: *N*-Body method

This method fits perfect on ***BrightScale*** architecture, because for $j = 0$ to $j = n - 1$ must be computed:

$$U(x_j) = \sum_{i=0}^{n-1} F(x_j, X_i).$$

Each function $F(x_j, X_i)$ is computed by another EU, and then the sum is a *reduction* operation linearly accelerated by the array. Depending on the value of n, the data is distributed in the processing array using the spatial dimension only, or both, the spatial and the temporal dimension.

4.5. *Motif* 5: Structured grids

The grid is distributed on the two dimensions of our array: the spatial dimension and the temporal dimension. Each processor is assigned a line of nodes (on the spatial dimension). It performs each update step locally and independently of other line of nodes. Each node only has to communicate with neighboring nodes on the grid, exchanging data at the end of each step. The system works as a cellular automata. The computation is accelerated almost linearly.

4.6. *Motif* 6: Unstructured grids

Unstructured grids problems are described as updates on an irregular grid, where each grid element is updated from its neighbor grid elements. Parallel computation is disturbed when problem size is large, and the non-uniformity of the data distribution ask for special access mechanisms. In order to solve the non-uniformity problem a preprocessing step is required.

The algorithm for preprocessing the n-element unstructured grid representation starts from an initial list of grid elements $G = \{g_0, \dots g_{n-1}\}$ and provide the minimum number of vectors, following the steps sketched here:

1. the $n \times n$ interconnection matrix for n grid elements is generated
2. interchanging elements in the list G a minimal band matrix is generated
3. each diagonal of the band represents a vector loaded into the processing array
4. results a grid with some dummy elements, but each actual grid element has its neighborhood located in few adjacent PEs.

4.7. *Motif* 7: Map reduce

The typical example map reduce computation is the *Monte Carlo method.* This method consists in many completely independent computations working on randomly generated data. This type of computation is highly parallel. Sometimes it requests the *add reduction* function, for which ***BrightScale*** architecture has special accelerating hardware.

4.8. *Motif* 8: Combinational logic

There are a lot of very different problems falling in this class. We list here only the most important and the most frequently used ones:

1. blocks processing, exemplified by AES encryption algorithms; it works in 4×4 arrays of bytes, each array is loaded in one EU, and the processing is completely SIMD-like with linear acceleration
2. stream processing, exemplified by convolutional methods which do not use blocks, processing instead a continuous bitstream; it is computed very efficient in the time parallel accelerator (SA) with no speculation involved
3. image rotation for black & white or color bit mapped images; it is performed (1) loading $m \times m$ array of pixels into the processing array on both dimensions (spatial and temporal), (2) executing a local transformation, and (3) restoring the transformed image in the appropriate place
4. route lookup, used in networking; it supposes three data-base like operations: *longest match, insert, delete*, all performed very efficiently by the ***BrightScale*** processing array.

4.9. *Motif* 9: Graph traversal

The array of 1024 machines can be used as a big "speculative device". Each EU starts with a full graph stored in its data memory, and the computation provides the result when one EU, if any, finds the solution. Limitations are generated by the dimension of the data memory of each EU. More investigation is needed to evaluate the actual power of ***BrightScale*** technology in solving this problem.

Some problems related with graphs are easy solved if matrix computation is involved (example: computing the distance between all the elements of a graph).

4.10. ***Motif*** 10: Dynamic programming

Viterbi decoding is the example presented in [1]. It requests the modular feed-forward architecture of SA, built as a distinct network (like in the actual implementation) or integrated into the main data parallel processing array. Very long stream of bits are parallel computed on line by the pipeline structure of SA.

4.11. ***Motif*** 11: Back-track and branch & bound

Motif under investigation (even *"Berkeley's View"* is silent regarding this *motif*).

4.12. ***Motif*** 12: Graphical models

Motif under investigation (even *"Berkeley's View"* is silent regarding this *motif*).

4.13. ***Motif*** 13: Finite state machine

The authors of *"Berkeley's View"* claim that for this *motif* "nothing helps". But, we consider that a pipe of machines featured with speculative resources [5] helps a lot. In fact, SA solves the problem if its speculative resources are activated. ***BrightScale*** technology offers SA as the first implementation of a machine able to deal with this rebellious *motif*.

5. Concluding Remarks

1. *BrightScale* technology covers almost all motifs. Excepting the motifs 11 and 12 (work on them in progress), possibly 9, the ***BrightScale*** technology performs very well. Therefore, we can claim that the *elementary parallel architecture* and the associated organization, based on Kleene's model, is a promising start for a true *parallel computing science*.

2. The linear network is not a limitation. Because the intense computational problems are characterized by an *advanced locality*, the simplest interconnection network is not a major limitation. The temporal dimension of the architecture helps many times to avoid the limitations imposed by the two simple interconnection networks.

3. The spatial & temporal dimension are doing a good job together. The user's view of the machine is a two-dimension array. Actually one dimension is in space (the 1024 EUs), and the other dimension is in time (the 512 16-bit words stored in each local memory). These two distinct dimensions allow to optimize area, while the locality and the degree of parallelism are both kept at high values.

4. Time parallelism is rare, but unavoidable. Almost anytime in a real complex application all kinds of parallelism are involved. Some pure sequential processes represent sometimes uncomfortable corner cases solved only by the time parallel resources provided in ***BrightScale*** architecture (see the 13th *motif*).

5. *BrightScale*'s organization is transparent. Because the interconnection network is simple the internal organization of the machine is easy to be made transparent to the user. The elegant solution offered by the *VectorC* language is a good proof of the high organizational transparency of the ***BrightScale*** technology.

References

[1] ASANOVIC K. *et al.*, *The Landscape of Parallel Computing Research: A View from Berkeley*, *Technical Report No. UCB/EECS-2006-183*, December 18, 2006. (click here to find the report: `http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf`

[2] FLYNN M. J., *Some computer organization and their affectiveness*, *IEEE Trans. Comp.* **C21**:9 (Sept. 1972), pp. 948–960.

[3] KLEENE S., *General Recursive Functions of Natural Numbers*, in *Math. Ann.*, 1936.

[4] MALIŢA M., ŞTEFAN G., THIEBAUT D., *Not Multi-, but Many-Core: Designing Integral Parallel Architectures for Embedded Computation*, in *ACM SIGARCH Computer Architecture News*, Volume **35** , Issue 5, Dec. 2007, Special issue: *ALPS '07 - Advanced low power systems; communication at International Workshop on Advanced Low Power Systems held in conjunction with 21st International Conference on Supercomputing June 17, 2007 Seattle, WA, USA.*

[5] MALIŢA M., ŞTEFAN G., *On the Many-Processor Paradigm*, in H. R. Arabina (Ed.), *Proceedings of the 2008 World Congress in Computer Science, Computer Engineering and Applied Computing, vol. PDPTA'08 (The 2008 International Conference on Parallel and Distributed Processing Techniques and Applications)*, 2008.

[6] MÎŢU B., *C Language Extension for Parallel Processing*, research report, BrightScale, 2008. (click here to find it: `http://arh.pub.ro/gstefan/Vector%20C.ppt`)

[7] PATTERSON D. A., *The Parallel Computing Landscape: A Berkeley View 2.0*, keynote lecture at *The 2008 World Congress in Computer Science, Computer Engineering and Applied Computing*, Las Vegas, July, 2008.

[8] ŞTEFAN G., *Integral Parallel Computation*, in *Proceedings of the Romanian Academy*, Series A: Mathematics, Physics, Technical Sciences, Information Science, vol. **7**, no. 3, Sept.-Dec. 2006, pp. 233–240.

[9] ŞTEFAN G., *The CA1024: SoC with Integral Parallel Architecture for HDTV Processing*, invited paper at *4th International System-on-Chip (SoC) Conference & Exhibit*, November 1 & 2, 2006, Radisson Hotel Newport Beach, CA.