Evolutionary Computation on the Connex Architecture

István Lőrentz
Electronics and Computers Department
Transylvania University
Braşov, Romania
istvan@splash.ro

Mihaela Maliţa
Computer Science Department
Saint Anselm College Manchester
Manchester, NH, USA
mmalita@anselm.edu

Răzvan Andonie
Computer Science Department
Central Washington University
Ellensburg, WA, USA
and
Electronics and Computers Department
Transylvania University
Braşov, Romania
andonie@cwu.edu

Abstract

We discuss massively parallel implementation issues of the following heuristic optimization methods: Evolution Strategy, Genetic Algorithms, Harmony Search, and Simulated Annealing. For the first time, we implement these algorithms on the Connex architecture, a recently designed array of 1024 processing elements. We use the Vector-C programming environment, an extension of the C language adapted for Connex.

Introduction

Evolutionary Algorithms (EA) are a collection of optimization methods inspired from natural evolution (Bäck 1996), (Back, Fogel, & Michalewicz 1997), (Back, Fogel, & Michalewicz 1999). The problem is formulated as finding the minimum value of an evaluation function over a set of parameters defined on a search space. Well known evolutionary techniques are: Evolution Strategy (ES), Genetic Algorithms (GA), and Evolutionary Programming (EP). These techniques are also related to stochastic search (e.g., Simulated Annealing (SA)), and they share the following characteristics:

- Start with a random initial population.
- At each step, a set of new candidate solutions is generated based on the current population.
- Based on some criteria, the best candidates are selected to form a new generation.
- The algorithm is repeated until the solution is found, or a maximum number of iterations reached.

EAs are *meta-heuristic*, as they don't make many assumptions of the function being optimized (for example, they do not require known derivatives). From a meta-heuristic point of view, the function to be optimized is a 'black-box', only controlled by the input parameters and the output value. Meanwhile, EAs are parallel by their nature. Parallel implementations of optimization algorithms is generally a complex problem and this becomes more challenging on fine grained architectures with inter-processor communication burdens.

Our study focuses on implementation issues of EAs on a recent massively parallel architecture - the Connex Architecture (CA). The CA is a parallel programmable VLSI chip consisting of an array of processors. Functionally, it is an array/vector processor. It is not a dedicated, custom-designed (ASIC) chip, but a general purpose architecture. The CA is now developed by Vebris[1]. An older version was developed in silicon by BrightScale, a Silicon Valley start-up company in (see (Ştefan 2009)).

Several computational intensive applications have been already developed on the CA: data compression (Thiebaut & Ştefan), DNA sequences alignment (Thiebaut & Ştefan 2001), DNA search (Thiebaut, Ştefan, & Maliţa 2006), computation of polynomials (Thiebaut & Maliţa), frame rate conversion for HDTV (Ştefan 2006), real-time packet filtering for detection of illegal activities (Thiebaut & Maliţa 2006), neural computation (Andonie & Maliţa 2007), and Fast Fourier Transform (Lőrentz, Maliţa, & Andonie 2010).

We do not intend to compare the efficiency of different EAs on the CA, but to provide the implementation building blocks. The motivation and novelty of this work are to expose the CA's vector processing capability for meta-heuristic optimization algorithms. We will provide the resulted performance results (instructions/operators) for several optimization benchmarks. The code is written in C++, using Vector-C, available at (Maliţa 2007), a library of functions which simulate CA operations. We use simulation because the floating-point version of the chip is still under development.

Review of Evolutionary Algorithms

We will first summarize the following standard optimization algorithms: Evolution Strategy, Genetic Algorithms, and Harmony Search, and Simulated Annealing. We will describe them in a unified way, in accordance to the EA general scheme from the Introduction.

Genetic Algorithms

In the original introduction of the 'Genetic Algorithm' concept, described by (Holland 1975), the population of 'chromosomes' is encoded as binary strings. Inspired from biological evolution, every offspring is produced by selecting two parents (based on their fitness), the genetic operators are the cross-over and single-bit mutation. The theoretical

[1] `http://www.vebris.com/`

foundation of GA is the Schema Theorem. Since the original formulation, GA evolved into many variants. We will consider here only the standard procedure:

Algorithm 1 Genetic Algorithm

Initialize population, as M vectors over the $\{0, 1\}$ alphabet, of length N.
repeat
Create M child vectors, based on:
1. Select 2 parents, proportionate to their fitness
2. Cross-over the parents, on random positions
3. Mutate (flip) bits, randomly
The created M child vectors will form the new population, the old population is discarded.
until termination criterion fulfilled (solution found or maximum number of iterations reached).

Evolution Strategy

Evolution Strategy is also a population based optimization method, with canonical form written as $(\mu/\rho + \lambda)$-ES. Here μ denotes the number of parents, ρ the mixing number (number of parents selected for reproduction of an offspring), λ the number of offspring created in each iteration (Beyer & Schwefel 2002).

Algorithm 2 Evolution Strategy algorithm (μ, λ)-ES

Initialize population $\mathbf{V}_\mu = \{\mathbf{v}_1, \ldots, \mathbf{v}_\mu\}$. Each individual $\mathbf{v}$ of the parent population represents a vector of N numbers encoding the decision variables (the search space) of the problem. The population is initialized randomly.
repeat
Generate λ offspring $\tilde{\mathbf{v}}$ forming the offspring population $\{\tilde{\mathbf{v}}_1, \ldots, \tilde{\mathbf{v}}_\lambda\}$ where each offspring $\tilde{\mathbf{v}}$ is generated by:
1. Select (randomly) ρ parents from $\mathbf{V}_\mu$.
2. Recombine the selected parents $\mathbf{a}$ to form a recombinant individual $\tilde{\mathbf{v}}$.
3. Mutate the parameter set $\mathbf{s}$ of the recombinant.
Select new parent population (using deterministic truncation selection) from either
- the offspring population $\tilde{\mathbf{V}}_\lambda$ (this is referred to as comma-selection, usually denoted as (μ, λ)-selection), or
- the offspring $\tilde{\mathbf{V}}_\lambda$ and parent $\mathbf{V}_\mu$ population (this is referred to as plus-selection, usually denoted as $(\mu+\lambda)$-selection)
until termination criterion fulfilled (solution found or maximum number of iterations reached).

The specific mutation and recombination operations will be presented later in this paper.

Harmony Search

Harmony Search (HS) is a meta-heuristic algorithm inspired by musical composition (Geem, Kim, & Loganathan 2001). According to (Weyland 2010), HS is a particular case of the $(\mu + 1)$ ES algorithm. In HS, the population, encoded as vectors of real or integer numbers, is stored in a matrix. The population size (number of rows) is fixed. Each new candidate is created by a discrete recombination (identical to the recombination of ES), or as a random individual. Mutation is performed with given probability. A key parameter is the the mutation 'strength' (or bandwidth). The new individual will replace the worst individual in the actual population if it is 'better' than this one.

Simulated Annealing

Inspired from the physical process of annealing, SA allows unfavorable decisions, when a controlling parameter called 'temperature' is high.

Over the iterations, the temperature is decreased and the algorithm will asymptotically approach a stochastic hill climbing. SA (Kirkpatrick *et al.* 1983) can be implemented over a population of (1 parent + 1 descendant), using the uniform mutation presented later in this article.

Algorithm 3 Simulated Annealing

Initialize a random candidate solution V
Set initial temperature, $T = T0$
repeat
mutate (perturb) the existing solution, to create V'
compute $\Delta = f(V') - f(V)$
if $\Delta < 0$ *or* $U(0, 1) < \exp(-\Delta/T)$ **then**
accept new candidate: V = V'
end if
Reduce T
until termination criterion fulfilled (Acceptable solution found or maximum iterations reached)
return V, f(V)

$U(0, 1)$ denotes an uniform random variable between $[0, 1]$.

The Connex-BA1024 chip

We implement the previous optimization algorithms on the CA, a massively parallel architecture known as the Connex BA1024 chip. In this section we briefly introduce some of the hardware characteristics of BA1024. As a first CA implementation example, we will describe a random number generator program. This generator will be used in our subsequent applications.

The CA is a Single Instruction Multiple Data (SIMD) device with 1024 parallel processing elements (PEs), as well as a sequential unit, which allows general purpose computations. It contains standard RAM circuitry at the higher level of the hierarchy, and a specialized memory circuit at the lower level, the Connex Memory, that allows parallel search at the memory-cell level and shift operations.

Several CA chips can be integrated on the same board, extending the length of processed vectors in increments of 1024, while receiving instructions and data from only one controller. A controller oversees the exchange of data between the two levels. Just as regular memory circuits, the

operations supported by the CA can be performed in well-defined cycles whose duration is controlled by the current memory technology, which in today's technology is in the 1.5 ns range.

The 1024 cells are individually addressable as in a regular RAM, but can also receive broadcast/instructions or data on which they operate in parallel at a peak rate of 1 operation per cycle. This general concept fits the Processor-In-Memory paradigm. The cells are connected by a linear chain network, allowing fast shifting of data between the cells, as well as the insertion or deletion of data from cells while maintaining the relative order of all the data. All these operations are performed in a single memory cycle.

The hardware performances of BA1024 are:

- Memory cycle: 1.5 ns.
- Computation: 400 GOPS at 400 MHz (peak performance)
- External bandwidth: 6.4 GB/sec (peak performance)
- Internal bandwidth: 800 GB/sec (peak performance)
- Power: $\approx$ 5 Watt
- Area: $\approx$ 50 mm^2 (1024-EU array, including 1Mbyte of memory and the two controllers).
- 65nm implementation

Using a 16-bit arithmetic, the BA1024 computes the scalar product of a 1024-tuple vector in 37.5 ns (26 million scalar products/sec), and performs 1024×1024 matrix multiplications in 40 ms (25 operations/sec). Adding up to 1024 numbers is done in 5 cycles. Multiplication is done in 10 cycles. The $P = 1024$ processing elements, each containing 512 registers, are interconnected in a ring. From an algorithmic point of view, the chip can be considered as an array of $P = 1024$ columns and $M = 512$ rows. By convention, we represent it as an array of horizontal vectors. In C-style row-major notation, $A[i][j]$ denotes the i'th register inside the j-th processing element.

An important component of evolutionary algorithms is the pseudo-random number generator. An ideal random number generator should be (Quinn 2003): uniformly distributed, uncorrelated, cycle-free, satisfy statistical randomness tests, and reproducible (for debugging purposes). In addition, parallel generators must provide multiple independent streams of random numbers. We used the xorshift generator, introduced by (Marsaglia 2003), with period $2^{128} - 1$. The random seed needs 4 integer vectors $X[0], X[1], X[2], X[3]$ of 1024 elements each. Here is the C++ code of this pseudo-random generator, using the Vector-C library:

```
vector<uint> xor128(vector<uint> X[]) {
 vector<uint> T;
 T  = x[0] ^ (X[0] << 11);
 T ^= (T ^ (T >> 8));
 T ^= X[3] ^ (X[3]  >> 19);
 X[0] = X[1];
 X[1] = X[2];
 X[2] = X[3];
 X[3] = T;
 return T;
}
```

Vectors are in represented in uppercase and initialized with seed values from the host computer (in Linux, `/dev/urandom`). It is essential that each component of the seed vector has a different, independent value. Once initialized, the presented function generates 1024 independent pseudo-random streams.

On the CA, generating in parallel $N <= 1024$ uniformly distributed random numbers results in a linear speedup: $S_{xor128} = T_{sequential}/T_{parallel} = N$, where $T_{sequential}$ is sequential execution time and $T_{parallel}$ is parallel execution time.

The `randvN(`σ`)` function returns a vector. Each component of this vector is an independent random variable with Gaussian distribution, 0 mean and σ standard deviation. The CA lacks trigonometric and logarithmic functions, used by the Box-Muller method for generating normal distributed random numbers. Therefore, we used an approximation method, based on the central limit theorem: $N(0,\sigma) \approx \sigma\left(\sum_{k=1}^{12} U(0,1) - 6\right)$, where $U(0,1)$ is the uniform random number generator in the [0, 1] interval.

Evolutionary operators on the CA

We present the building blocks of an evolutionary algorithm using the CA vector instructions. The control flow of the algorithm is still sequential, but mutation and evaluation operators are vectorized. The population is represented as a matrix. Rows (individuals) are mapped as CA vectors and use vectorial instructions for mutation, recombination, and evaluation. A population is evaluated sequentially. The vector length (max. number of decision variables of the search space) is limited to 1024, while the population size is limited by the number of CA rows. Horizontal mapping allows efficient computation of fitness functions via the parallel CA reduction operator.

Recombination

The recombination operator forms a new individual, based on a set of parents in the existing population. Typically the offspring will get a combination of the parents features. There are many variants for the recombination, we will present the commonly used ones in GA and ES: crossover and discrete recombination.

Crossover The crossover operation creates a new individual by combining the features of two parents. In one-point crossover, elements from the first parent vector are copied up to a random position. Continuing from that position, elements from the second parent vector are further copied. We implement this using a vector selection mask of random length (Fig. 1).

```
vector crossover(vector X, vector Y){
 int position = rand(VECTOR_SIZE) ;
 where( i < position )
  C = X;  elsewhere  C = Y;
 return C;
}
```

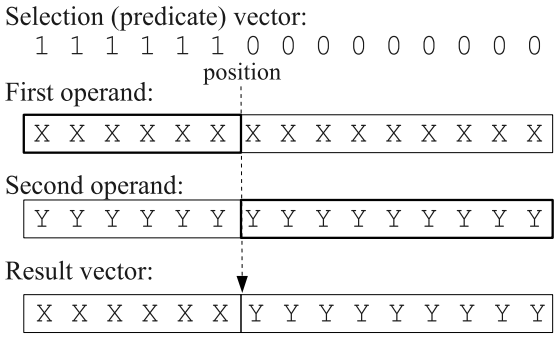


Figure 1: Parallel one-point crossover using predicate vector.

The `rand(n)` scalar function returns a random integer in the range [0, n-1]. The statement **where**(condition) ... **elsewhere** ... is a parallel-if construct available on CA. Index i denotes the processor element. The expression is evaluated in parallel on each PE_i, and a selection flag (predicate) is set, which conditions the execution of the statements inside the `where` block. The `elsewhere` block is executed after the selection predicates are negated. For brevity, we omit the vector element data type, which can be either integer or float.

To obtain a two-point crossover, we need to change the condition inside `where` to use 2 parameters, denoting the start and end splicing points:

```
where ( i>=a && i<b )    C = X;
elsewhere   C = Y;
```

The above code can be generalized for uniform crossover (Sywerda 1989). In this case, for each position, a bit is randomly selected from one of the parents. Uniform crossover can implemented by changing the condition to

```
where (randvb(0.5)) { .... }
```

where `randvb`(p) creates a Boolean vector, each bit having value '1' with probability p.

Discrete Recombination In ES, the recombination operator uses information from ρ individuals. In discrete recombination, each position of the candidate individual vector $\mathbf{v}'$ is copied from the same position of a randomly chosen parent: $\mathbf{v}'(\mathbf{i}) = \mathbf{v_k}(\mathbf{i})$. In this case, the HS algorithm uses a recombination of the entire population.

CA supports matrix-vector addressing (selecting a different cell from each column, to form a new vector), which is used for discrete-recombination.

For $N <= 1024$, the parallel speedup of the two recombination operators is linear: $S_{crossover} = N$.

Mutation

Mutation involves changing a single, random position by a given amount. In horizontal mapping, first we create a selection mask, with a single '1' bit, on the k-th position, then perform a vector + scalar operation, which will add only the elements on the k-th position:

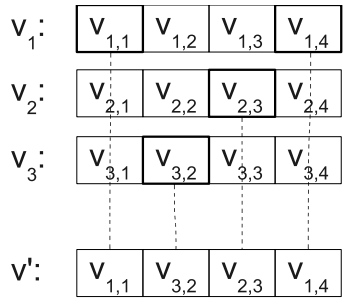


Figure 2: Discrete recombination. A new vector $\mathbf{V}'$ is created from 3 parents.

```
vector mutate(vector X){
 int pos = rand(VECTOR_SIZE);
 float amount = rand11(); // [-1...1]
 where (i == pos)
   X += amount;
 return X;
}
```

In ES, the mutation operator alters the vector by a random amount: $v' = v + N(0, \sigma^2)$, where $N(0, \sigma^2)$ denotes a random variable with normal distribution. Our Vector-C function name is `randvN(sigma)`. The σ^2 variance parameter controls the mutation strength:

```
vector mutateES(vector X){
  return X + randvN(sigma);
}
```

Since the single-bit mutation's serial execution time is constant, there is no speedup achieved by parallelization: $S_{mutate1bit} = 1$. On the other hand, the speedup for ES-mutation is linear: $S_{mutateES} = N$, since each vector element is affected.

Fitness Function Evaluation

In evolutionary techniques, evaluating the fitness functions usually consumes most of the time (compared to the mutation, selection), so it is crucial to implement it most efficiently. The class of functions that can be efficiently computed using vectorial instructions on the CA has the form:

$$f(x_1, x_2, ...x_N) = \bigoplus_{i=1}^{N} h_i(x_{i-k}, ..., x_i, ..., x_{i+k}) \qquad (1)$$

where $\bigoplus$ is the parallel-reduction operator, k defines a fixed-size neighborhood (independently of N). Currently, the CA supports parallel sum reduction. The $h_i()$ function should depend only on the i-th variable and optionally on a small local neighborhood, $i-k, ..., i+k$. This is due to the constrain that processing elements (PEs) are interconnected by a ring bus, so efficient communication is done only by neighboring PEs (data-locality).

In (Maliţa & Ştefan 2009), it is described how to compose such a function on the CA, by combining data-parallel and time-parallel computations, illustrated in Fig. 3. Such

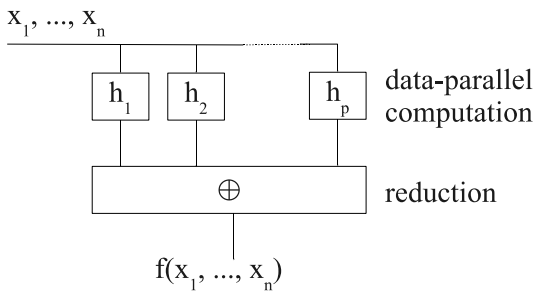


Figure 3: Parallel computation followed by reduction.

generic functions can be evaluated on P processors as follows (t_i are partial results):

for every $i = 1...P$ **do in parallel**
$t_i = h_i(...)$
end for
result = $reduce(t_1, ..., t_p)$

For a one-to-one mapping of $h()$ invocations to processing elements, $f()$ is computed in $T_f = T_h + T_{red}$ time, where T_h is serial time to compute $h()$ and T_{red} is reduction time (which is a CA machine instruction).

Assuming data-parallel computation, for sequentially processing N items the speedup is $S = \frac{N(T_h+T_+)}{T_h+T_{red}}$, where T_+ is serial execution time of the associative operator used for reduction.

Due to the constant parallel evaluation time (up to the maximum vector size 1024), we use functions that can be expressed this way.

Selection

Given the 'horizontal' mapping of the population in the CA, after evaluation, the fitness value (a scalar) is available to the sequential unit. The selection decision operation is not vectorized, it is done by the sequential unit by comparing or sorting the scalar fitness values.

Selection in Simulated Annealing To implement SA on the CA, we use the `mutate()` and `evaluate()` functions already presented. The SA-specific selection operation (to choose between two solutions Vold, Vnew) is:

```
vector selectSA(vector Vold, vector Vnew,
                                    float t)
{
 df = evaluate(Vnew) - evaluate(Vold);
 if ( df < 0 || randf() < exp(-df/t) )
  return Vnew;
 else
  return Vold;
}
```

The $\exp(-df/t)$ scalar function (Boltzmann factor) is evaluated by the CA's sequential unit. Function `randf()` returns an uniform random variable in the [0,1) interval.

Experimental Results

In our experiments, we use two benchmark problems: the generalized Rosenbrock function and the geometric distance problem.

The generalized Rosenbrock function

This is a standard benchmark function used in optimization, illustrated in Fig. 4. The generalized N-dimensional form is (De Jong 1975):

$$f(\mathbf{x}) = \sum_{i=1}^{N-1} \left[(1 - x_i)^2 + 100(x_{i+1} - x_i^2)^2\right] \quad \forall x \in \mathbb{R}^N \tag{2}$$

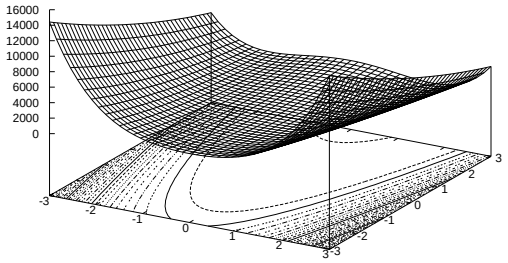


Figure 4: The Rosenbrock function of two variables.

The evaluation of the multi-dimensional Rosenbrock function can be performed using vector arithmetic, shifting and sum-reduction. The following code snippet shows the implementation:

```
float Rosenbrock(vector X) {
 vector A, X2, Xsh;
 Xsh = 0.0;
 where ( i<N )
  Xsh = rotateLeft(X, 1); // x1,x2, ...
 where( i<(N-1) ) {
  X2 = X * X;          // x0^2, x1^2, ...
  Xsh -= X2;           // (x1-x0^2), ...
  Xsh *= Xsh * 100;
  X2 = 1 - X;          // 1-x0, 1-x1, ...
  X2 = X2 * X2;        // (1-x0)^2, ...
  X2 += Xsh;
  return sumv(X2);
}}
```

Geometric distance problem

The geometric distance problem arises in molecular geometry: given a set of distances between pairs of atoms space, determine each atom's (x,y,z) coordinate. Although various solutions exist, the problem can be tackled also as a global optimization problem (Grosso, Locatelli, & Schoen 2009).

We implement a simplified form of this problem, where each coordinate is assumed to take only *discrete* values inside a given bounding rectangle. The aim is to minimize

$$f(x_1, ...x_N) = \sum_{i \neq j} \left(||x_i - x_j|| - d_{ij} \right)^2 ; \qquad (3)$$

for all (i, j) pairs for which d_{ij} is known, where $x_i \in D \subset \mathbb{Z}^3$.

To parallelize the evaluation function, we notice that the list of distances must be distributed for each processing element, since the CA does not support random-access inter-processor communication. The pairs of points for which the distances are known (as input data) represent the edges of an undirected graph. We label the edges as $e_1 ... e_N$ and the vertices as $\mathbf{x}_1, ...\mathbf{x}_V$. Each edge is mapped onto its own processor: $e_p \Leftrightarrow PE_p$.

To compute $f()$, we need for each pair the x_i, x_j, d_{ij} variables. The i, j vertex indexes for processor p are noted by i_p and $j_p, (p = 1...N)$.

Note that some of the vertices must be shared between processors. To implement this sharing, we use the following method: Each PE p will hold the distance d_p and the vertices of the two nodes it connects $\mathbf{x}_{i_p}, \mathbf{x}_{j_p}$. For example, in a simple triangle case with three vertices, we have three edges with labels e_0: A - B, e_1: B - C, e_2: A - C (Fig. 5). To avoid inter-processor communication during the iterations, since each PE stores vertex data into private variables, we must assure that the variables which represents the same vertex on a different processor have identical values. We do this in the following way:

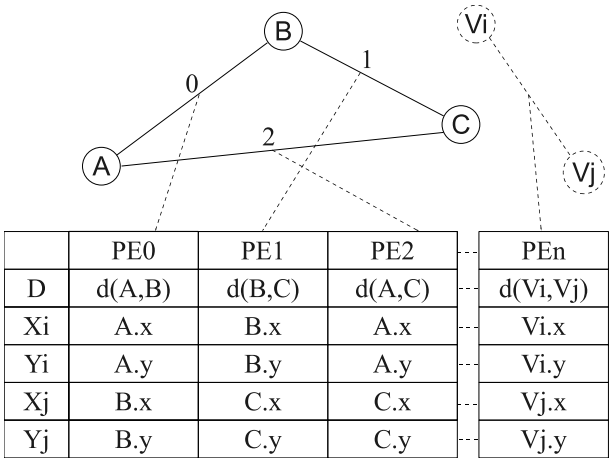
1. The vertices are initialized to random values, at the program initialization.
2. The vertices are distributed to each processor, each processor stores a private copy.
3. Each vertex $\mathbf{x}_i$ will have also associated a random number generator stream r_i.

This data representation allows parallel evaluation of the sum of the distances and parallel mutation of the vertex coordinates. We present the flowchart of the computation in Fig. 6.

For example, to load the graph represented in Fig. 5, we assign to each edge the corresponding PE. PE_0 will receive the data corresponding to edge 0: the coordinates of points A,B and the distance d(A,B).

To evaluate the distances, no inter-processor communication is required. Each PE computes the distance between the vertices it holds and subtracts from the known, input distance. The parallel reduction step computes the sum of squared differences, resulting a scalar fitness value.

```
void evaluateDist(vector Xi,Yi,D)
{
 vector Dx, Dy;
 Dx=Xi[k]−Xj[k];
 Dy=Yi[k]−Yj[k];
 Dx *= dx;  Dy *= dy;
 Dx += dy;
 return sumAbsDiff(Dx,D);
}
```



	PE0	PE1	PE2	...	PEn
D	d(A,B)	d(B,C)	d(A,C)	...	d(Vi,Vj)
Xi	A.x	B.x	A.x	...	Vi.x
Yi	A.y	B.y	A.y	...	Vi.y
Xj	B.x	C.x	C.x	...	Vj.x
Yj	B.y	C.y	C.y	...	Vj.y

Figure 5: Example of a graph loaded into the Connex Array. The edge labels are the indexes, for which the distances are known. When new edges are added, the table extends horizontally, while the number of rows is kept constant. There are also two additional rows (Ri, Rj), not shown in the figure, which contain the seeds for the random generators

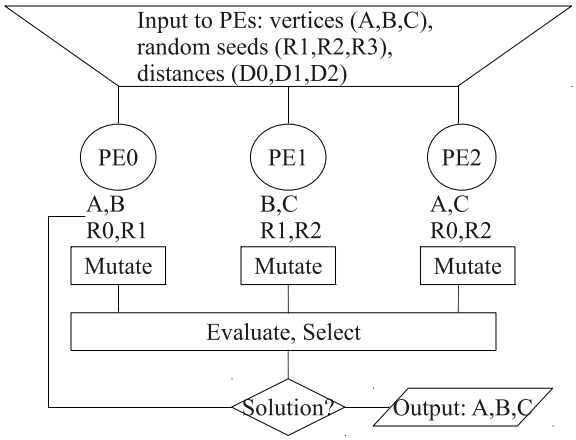


Figure 6: Flowchart of the parallel evolution of vertices. Note that apart from the evaluation (sum-reduction) there is no inter-communication between the processors

In the above listing, the input vectors are:

Xi_p, Yi_p - first vertex belonging to edge p,

Xj_p, Yj_p - second vertex belonging to edge p,

D_p = (known) length, squared, of edge p.

sumAbsDiff(Dx,D) sums the absolute differences of two vectors:

```
float sumAbsDiff(vector A, vector B) {
 vector V;
 V = A − B;
 where(V < 0)
  V= –V;
 return sumv(V);
}
```

Results

We measured the number of vectorial operations, for each specific evolutionary operator, as well as some test functions (see Table 1).

Operation	T_{Par}	T_{Seq}	S
A+=B	1	1024	N
xorshift 128	13	13312	N
sumAbsDiffs	7	4096	0.5 N
1-Point Crossover	3	2048	0.6 N
Uniform Crossover	15	14350	0.9 N
Uniform Mutation	33	21172	0.6 N
HS Mutation	107	71506	0.6 N
Rosenbrock	14	14325	N
evaluateDist	13	10240	0.7 N

Table 1: Vector instruction count by evolutionary operators

T_{par} is parallel execution time, measured in units of vectorial operations, T_{seq} is sequential execution time (number of sequential operations; we used the instruction count instead of physical time). The last column contains S, the speedup T_{seq}/T_{par}, running on $N <= 1024$ processing elements. We use a one-to-one data element - PE mapping.

To accurately interpret these results, we have to emphasize that we used instruction counts instead of cycle counts simply because the floating-point version of the chip is still under development. The results give a theoretical achievable speedup when using the presented algorithms.

Conclusions

The meta-heuristic algorithms presented above are dependent on the way initial data is organized. We used horizontal mapping. Another choice is to map the population vertically, by loading the population data as columns in the CA. The vectorial instructions will operate in this case over the corresponding variables of the entire population. By this transposition, the previous parallel operations will become serial, and parallelism will operate over the entire population. However, in vertical mapping we cannot speed-up the evaluation function by using the parallel sum instruction. Since the evaluation function is the most time-critical, we did not explore further the vertical mapping method, to verify if there are benefits in other evolutionary blocks.

The CA offers vectorial computational facilities which are well suited for the implementation of evolutionary algorithms. We plan to continue our experimental work and test the efficiency of meta-heuristic optimization, including on the CA itself (not just on the simulator).

References

Andonie, R., and Maliţa, M. 2007. The Connex Array™ as a neural network accelerator. In *CI '07: Proceedings of the Third IASTED International Conference on Computational Intelligence*, 163–167. Anaheim, CA, USA: ACTA Press.

Back, T.; Fogel, D. B.; and Michalewicz, Z., eds. 1997. *Handbook of Evolutionary Computation*. Bristol, UK, UK: IOP Publishing Ltd., 1st edition.

Back, T.; Fogel, D. B.; and Michalewicz, Z., eds. 1999. *Basic Algorithms and Operators*. Bristol, UK, UK: IOP Publishing Ltd., 1st edition.

Bäck, T. 1996. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford, UK: Oxford University Press.

Beyer, H.-G., and Schwefel, H.-P. 2002. Evolution strategies A comprehensive introduction. *Natural Computing* 1:3–52.

Ştefan, G. 2009. One-Chip TeraArchitecture. In *Proceedings of the 8th Applications and Principles of Information Science Conference, Okinawa, Japan*.

De Jong, K. A. 1975. *An analysis of the behavior of a class of genetic adaptive systems.* Ph.D. Dissertation, Ann Arbor, MI, USA.

Geem, Z. W.; Kim, J. H.; and Loganathan, G. 2001. A New Heuristic Optimization Algorithm: Harmony Search. *SIMULATION* 76(2):60–68.

Grosso, A.; Locatelli, M.; and Schoen, F. 2009. Solving molecular distance geometry problems by global optimization algorithms. *Comput. Optim. Appl.* 43(1):23–37.

Holland, J. 1975. *Adaptation in natural and artificial systems*. University of Michigan Press.

Kirkpatrick, S.; Gelatt, C. D.; Jr.; and Vecchi, M. P. 1983. Optimization by Simulated Annealing. *Science* 220:671–680.

Lőrentz, I.; Maliţa, M.; and Andonie, R. 2010. Fitting FFT onto an energy efficient massively parallel architecture. In *Proceedings of the Second International Forum on Next-Generation Multicore/Manycore Technologies*, IFMT '10, 8:1–8:11.

Maliţa, M., and Ştefan, G. 2009. Integral parallel architecture & Berkeley's Motifs. In *ASAP '09: Proceedings of the 2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, 191–194. IEEE Computer Society.

Maliţa, M. 2007. The Vector-C library on Connex (A software library for a Connex-like multiprocessing machine). `http://www.anselm.edu/internet/compsci/Faculty_Staff/mmalita/HOMEPAGE/ResearchS07/WebsiteS07/`.

Marsaglia, G. 2003. Xorshift RNGs. *Journal of Statistical Software* 8(14):1–6.

Ştefan, G. 2006. The CA1024: SoC with integral parallel architecture for HDTV processing. In *4th International System-on-Chip (SoC) Conference & Exhibit, November 1-2*.

Quinn, M. J. 2003. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group.

Sywerda, G. 1989. Uniform crossover in genetic algorithms. In *Proceedings of the third international conference on Genetic algorithms*, 2–9. Morgan Kaufmann Publishers Inc.

Thiebaut, M., and Ştefan, G. Ziv-Lempel compression with the Connex Engine. Tech. Rep. 077, Dept. Computer Sci-

ence, Smith College, Northampton, MA, 01063, January 2002.

Thiebaut, M., and Ştefan, G. 2001. Local alignment of DNA sequences with the Connex Engine. In *The First Workshop on Algorithms in BioInformatics WABI 2001*.

Thiebaut, D., and Maliţa, M. Fast polynomial computation on Connex Array. Technical Report 303, Smith College, November 2006.

Thiebaut, D., and Maliţa, M. 2006. Real-time packet filtering with the Connex Array. In *Proceedings of the International Conference on Complex Systems*, 501–506.

Thiebaut, D.; Ştefan, G.; and Maliţa, M. 2006. DNA search and the Connex technology. In *Proceedings of the International Multi-Conference on Computing in the Global Information Technology (ICCGI'06)*.

Weyland, D. 2010. A rigorous analysis of the harmony search algorithm: How the research community can be misled by a "novel" methodology. *Int. J. of Applied Metaheuristic Computing* 1(2):50–60.