Analog recurrent neural network simulation, $\Theta(\log_2 n)$ unordered search, and bitonic sort with an optically-inspired model of computation

Damien Woods, Thomas J. Naughton and J. Paul Gibson

TASS Research Group,
Department of Computer Science,
National University of Ireland, Maynooth, Ireland.
Email: {dwoods,tomn,pgibson}@cs.may.ie
URL: http://www.cs.may.ie/TASS

Date: 03 September 2001

Technical Report: NUIM-CS-2001-TR-06

Key words: model of computation, unconventional model of computation, analog computation, optical computing, computability, computational complexity, analog recurrent neural network, Fourier transform, unordered search, bitonic sort.

Abstract

We prove computability and complexity results for an original model of computation. Our model is inspired by the theory of Fourier optics. We prove our model can simulate analog recurrent neural networks, thus establishing a lower bound on its computational power. We also prove some computational complexity results for searching and sorting algorithms expressed with our model.

1 Introduction

In this paper we prove some computability and complexity results for an original continuous-space model of computation. The model was developed for the analysis of (analog) Fourier optical computing architectures and algorithms, specifically pattern recognition and matrix algebra processors [12, 13]. The functionality of the model is limited to operations routinely performed by information processing optical scientists and engineers. The model operates in discrete timesteps over a finite number of two dimensional (2-D) images of finite size and infinite resolution. It can navigate, copy, and perform other optical operations on its images. A useful analogy would be to describe the model as a random access machine, without conditional branching and with registers that hold continuous images. It has recently been proven that the model can simulate Turing machines and Type-2 machines [14]. However, the model's exact computational power has not yet been characterised.

In Sect. 2, we define our optical model of computation and give the data representations that will used in Sects. 3 and 4. In Sect. 3 we demonstrate a lower bound on computational power by proving our model can simulate a type of dynamical system called Analog Recurrent Neural Networks (ARNNs) [18, 17]. This simulation result proves our analog model can decide any language in finite time. Our model admits some extremely efficient algorithms for standard searching and sorting algorithms. In Sect. 4, a $\Theta(\log_2 n)$ binary search algorithm that can be applied to certain unordered search problems and a sorting algorithm based on a bitonic sort are investigated.

2 Computational model

Each instance of our machine consists of a memory containing a program (an ordered list of operations) and an input. The memory structure is in the form of a 2-D grid of rectangular elements, as shown in Fig. 1(a). The grid has finite size and a scheme to address each element uniquely. Each grid element holds a 2-D infinite resolution complex-valued image. There is a program start location **sta** and a small number of 'well-known' addresses labeled **a**, **b**, **c**, and so on. The two most basic operations available to the programmer, **ld** and **st** (parameterised by two column addresses and two row addresses), copy rectangular $m \times n$ subsets of the grid into and out of image **a**, respectively. Upon such loading and storing the image contents are rescaled to the full extent of the target location [as depicted in

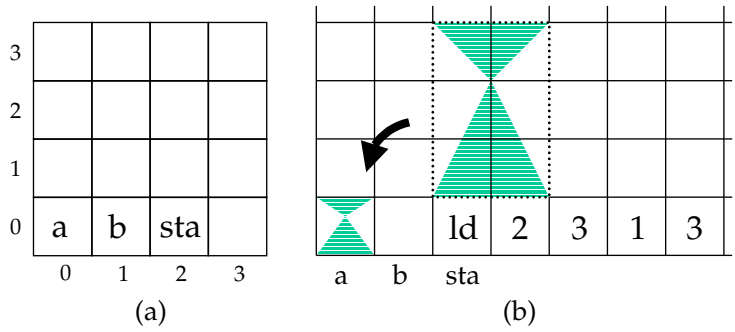

Figure 1: Schematics of (a) the grid memory structure of our model of computation, showing example locations for the well-known addresses **a**, **b** and **sta**, and (b) loading (and automatically rescaling) a subset of the grid into grid element **a**. The program | ld | 2 | 3 | 1 | 3 | . . . | *hlt* | instructs the machine to load into default location **a** the portion of the grid addressed by columns 2 through 3 and rows 1 through 3.

Fig. 1(b)]. The complete set of atomic operations is given in Fig. 2.

In this paper, a complex-valued image is a function $f : \mathbb{R}_0^1 \times \mathbb{R}_0^1 \mapsto \mathbb{C}$ where $\mathbb{R}_0^1 = \{x : x \in \mathbb{R} \wedge 0 \leq x \leq 1\}$. Let $\mathcal{I}$ be the set of such images. Each instance of our machine is a quadruple $M = (D, L, I, P)$, in which

- $D = (m, n)\,,\ \ m, n \in \mathbb{N}$: grid dimensions
- $L = (s_{\mathrm{x}}, s_{\mathrm{y}}, a_{\mathrm{x}}, a_{\mathrm{y}}, b_{\mathrm{x}}, b_{\mathrm{y}}, \ldots)\,,\ \ s, a, b \in \mathbb{N}$: locations of **sta** and the well-known locations
- $I = [(i_{1\mathrm{x}}, i_{1\mathrm{y}}, \psi_1), \ldots, (i_{k\mathrm{x}}, i_{k\mathrm{y}}, \psi_k)]\,,\ \ i \in \mathbb{N},\ \ \psi \in \mathcal{I}$: the k inputs and their locations
- $P = [(p_{1\mathrm{x}}, p_{1\mathrm{y}}, \pi_1), \ldots, (p_{l\mathrm{x}}, p_{l\mathrm{y}}, \pi_l)]\,,\ \ p \in \mathbb{N},\ \ \pi \in \{\{$ld, st, h, v, $*$, $\cdot$, +, br, hlt, /$\} \cup \mathcal{N}\} \subset \mathcal{I}$: the l programming symbols, for a given instance of the machine, and each of their locations. $\mathcal{N}$ is the set of row and column addresses encoded as images.

As might be expected for an analog processor, its programming language does not support comparison of arbitrary image values. Fortunately, not having such a comparison operator will not impede us from implementing a conditional branching instruction (see Sect. 2.3). In addition, address resolution is possible since (i) our set of possible image addresses is finite (each memory grid has a fixed size),

ld $c1$ $c2$ $r1$ $r2$ z_{l} z_{u}	: $c1, c2, r1, r2 \in \mathbb{N}$; $z_{\text{l}}, z_{\text{u}} \in \mathbb{Q}$; copy into **a** the rectangle of images defined by the image at coordinates ($c1$, $r1$) and the image at ($c2$, $r2$). Two additional real-valued parameters $(z_{\text{l}}, z_{\text{u}})$, specifying lower and upper cut-off amplitudes respectively, filter the rectangle's contents by amplitude before rescaling, as defined by $\rho(f(i,j), z_{\text{l}}, z_{\text{u}}) = \begin{cases} z_{\text{l}}, & \text{if } \lvert f(i,j)\rvert < z_{\text{l}} \\ \lvert f(i,j)\rvert, & \text{if } z_{\text{l}} \le \lvert f(i,j)\rvert \le z_{\text{u}} \\ z_{\text{u}}, & \text{if } \lvert f(i,j)\rvert > z_{\text{u}} \ . \end{cases}$ In general we use a filter of $(0, 1)$ in this paper, written as $(0/1, 1/1)$, where the symbol '/' is used to express rationals as the ratio of two integers.
st $c1$ $c2$ $r1$ $r2$ z_{l} z_{u}	: $c1, c2, r1, r2 \in \mathbb{N}$; $z_{\text{l}}, z_{\text{u}} \in \mathbb{Q}$; copy the image in **a** into the rectangle defined by the coordinates ($c1$, $r1$) and ($c2$, $r2$).
h	: perform a horizontal 1-D Fourier transform on the 2-D image in **a**. Store result in **a**.
v	: perform a vertical 1-D Fourier transform on the 2-D image in **a**. Store result in **a**.
$\cdot$	: multiply (point by point) the two images in **a** and **b**. Store result in **a**.
$+$	: perform a complex addition of **a** and **b**. Store result in **a**.
$*$	: replace **a** with the complex conjugate of **a**.
br $c1$ $r1$	: $c1, r1 \in \mathbb{N}$; unconditionally branch to the instruction at the image with coordinates ($c1$, $r1$).
hlt	: halt.

Figure 2: The set of atomic operations permitted in the model.

and (ii) we anticipate no false positives (we will never seek an address not from this finite set). A more formal treatment of this point is given in Sect. 2.2.

2.1 Encoding numerical values as images

There are many ways to encode finite, countable, and uncountable sets as images. We outline a number of techniques that will be used later in the paper. Consider an image that contains a high amplitude peak at its centre and zero everywhere else,

$$|f(x,y)| = \begin{cases} 1, & \text{if } x,y = 0.5 \\ 0, & \text{otherwise} \quad . \end{cases} \tag{1}$$

Such an image encodes the symbol '1'. An empty image $f(x,y) = 0$ encodes '0'. Images encoding symbols '0' and '1' can be combined using a stepwise rescaling technique (an image 'stack') or with a single rescale operation (an image 'list') to encode nonnegative integers in unary and binary notations. These concepts are illustrated in Fig. 3. A stack encoding of the integer 2 in unary could be accomplished as follows. Take an empty image, representing an empty stack, and an image i encoding a '1' that that we will 'push' onto the stack. A push is accomplished by placing the images side-by-side with i to the left and rescaling both into the stack location. The image at this location, a stack with a single '1', would be regarded as a stack encoding of the integer 1. Take another image j encoding a '1', place it to the left of the stack, and rescale both into the stack location once again. The unary stack image contains two peaks at particular locations that testify that it is an encoding of the integer 2. To decrement the value in the stack, a 'pop' operation is applied. Rescale the stack image over any two image locations positioned side-by-side. The image to the left will contain the symbol that had been at the top of the stack and the image to the right will contain the remainder of the stack. The stack can be repeatedly rescaled over two images popping a single image each time. Binary representations of nonnegative integers would be encoded in a similar manner. A unary stack encoding of the integer 2 could be regarded as a binary stack encoding of the integer 3. Our convention is to encode binary strings with the least significant bit at the top of the stack. Therefore, if j in the preceding example had instead encoded a '0' the resulting push operation would have created a binary encoding of the integer 2. As a convenient pseudocode, we use statements such as `c.push('1')` and `c.pop()` to increment and decrement the unary string encoded in well-known image location **c**.

In the 'list' encoding of a unary or binary string, each of the rescaled '0' and

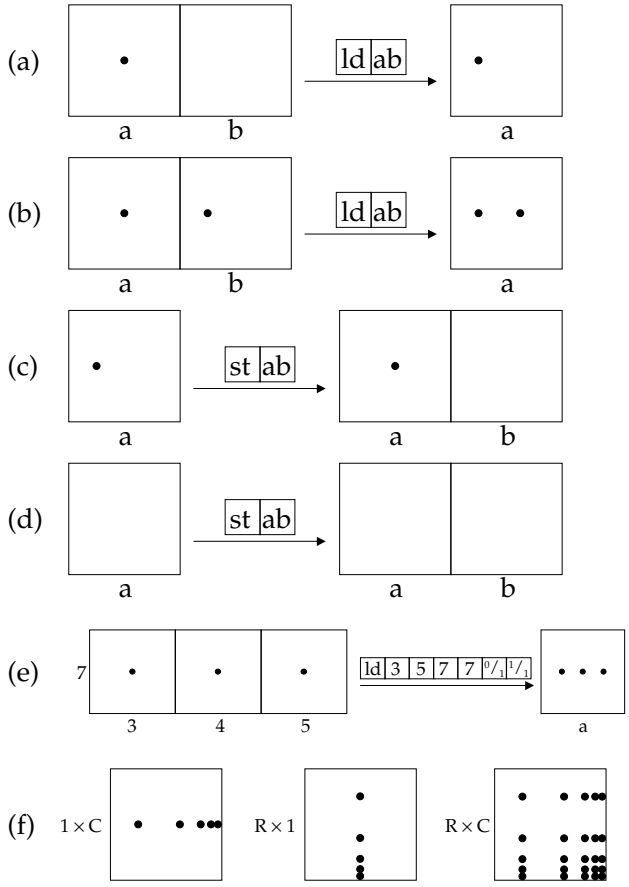

Figure 3: Encoding numbers in images through the positioning of amplitude peaks. In the illustrations, the nonzero peaks are coloured black and the white areas denote zero amplitude. (a) To encode the integer 1 in unary notation we could use a 'stack' structure. We 'push' a symbol '1' (image **a**) onto an empty stack (image **b**) using the command ld ab. The result by default is stored in **a**. (b) Pushing a '1' onto a stack encoding the integer 1 in unary to create the encoding of 2 in unary. (c) 'Popping' a stack encoding the number 1, resulting in a popped symbol '1' (in **a**) and an empty stack in **b**. (d) Popping an empty stack results in the symbol '0' and the stack remains empty. (e) Encoding the integer 3 in unary in a 'list' structure. (f) Illustration of the matrix images used in the ARNN simulation.

'1' images are equally spaced in the list image (in contrast to the Cantor dust encoding used for stacks). A unary list encoding of the integer 3 would involve positioning three '1' symbols side-by-side in three contiguous image locations and rescaling them into a single image in a single step [see Fig. 3(e)]. The choice of encoding (stack or list) is usually driven by computational complexity considerations. It is possible to enhance this encoding by allowing the peaks to have real-valued amplitudes, and by using both dimensions of the image. This enhanced encoding is used in the ARNN simulation of Sect. 3. In this simulation, the stack encoding of a finite number of real-valued amplitude peaks is referred to as a $1 \times C$ matrix image (when encoded in the horizontal direction), and an $R \times 1$ matrix image (when encoded in the vertical direction), and an $R \times C$ matrix image [when both dimensions are used – see Fig. 3(f)].

2.2 Transformation from continuous image to finite address

Our model uses symbols from a finite set in its addressing scheme and employs an address resolution technique to effect decisions (see Sect. 2.3). Therefore, during branching and looping, variables encoded in continuous images must be transformed to the finite set of address locations. In one of the possible addressing schemes available to us, we use symbols from the set $\{0, 1\}$. We choose $B = \{0^i 1 0^j : i, j \geq 0 \wedge i + j = m + n - 1\}$ as our underlying set of address words. Each of the m column and n row address locations will be a unique binary word in the finite set B. An ordered pair of such binary words identifies a particular image in the grid. Each address word will have an image encoding. $\mathcal{N}$ is the set of such image encodings, with $|\mathcal{N}| = m + n$. In order to facilitate an optical implementation of our model we cannot presuppose any particular encoding strategy for the set $\mathcal{N}$ (such as the simple stack or list binary encodings of Sect. 2.1). Our address resolution technique (our transformation from $\mathcal{I}$ to B) must be general enough to resolve addresses that use any sensible encoding.

Given an image $s \in \mathcal{I}$ we wish to determine which address word $s' \in B$ is encoded by s. In general, comparing a continuous image s with the elements of $\mathcal{N}$ to determine membership is not guaranteed to terminate. However, since it will always be the case that (i) $s \in \mathcal{N}$, (ii) that $|\mathcal{N}|$ is finite, and (iii) that $\mathcal{N}$ contains distinct images, we need only search for the single closest match between s and the elements of $\mathcal{N}$. We choose a transformation based on cross-correlation (effected through a sequence of Fourier transform and image multiplication steps) combined with a thresholding operation.

The transformation $t : \mathcal{I} \times \mathcal{I} \mapsto B$ is specified by

$$t(s, P) = \tau\left(P \circledast s\right) \quad , \tag{2}$$

where s encodes an unknown addressing symbol, P is a list image that contains a predefined ordering of the elements of $\mathcal{N}$, $\circledast$ denotes a cross-correlation operation, and τ is a thresholding operation. The cross-correlation operation produces an image

$$f(u, v) = P \circledast s = \int_0^1 \int_0^1 P(x, y) s^*(x + u, y + v) \mathrm{d}x \mathrm{d}y \quad , \tag{3}$$

where (x, y) and (u, v) are the coordinates of the input and output spaces of the correlation operation, respectively. In the theoretical machine, $f(u, v)$ could be produced by the code fragment $\boxed{\text{ld}\,|\,\text{P}\,|\,\text{h}\,|\,\text{v}\,|\,\text{st}\,|\,\text{b}\,|\,\text{ld}\,|\,\text{s}\,|\,\text{h}\,|\,\text{v}\,|\,*\,|\,\cdot\,|\,\text{h}\,|\,\text{v}}$, where a multiplication in the Fourier domain is used to effect cross-correlation. According to Eq. (3), the point of maximum amplitude in $f(u, v)$ will be a nonzero value at a position identical to the relative positioning of the encoded symbol in P that most closely matches s. All other points in $f(u, v)$ will contain a value less than this cross-correlation value. The thresholding operation of Eq. (2) is defined

$$\tau\left(f(u, v)\right) = \begin{cases} 1, & \text{if } |f(u, v)| = \max(|f(u, v)|) \\ 0, & \text{if } |f(u, v)| < \max(|f(u, v)|) \end{cases} \quad . \tag{4}$$

This produces an image with a peak at image coordinates $(\frac{2i+1}{2m+n}, 0.5)$ for one and only one positive integer i in the range $[0, m + n - 1]$. Given the definition of a list encoding of a binary string (Sect. 2.1), we can see that these unique identifiers are exactly the images that encode the integers $\{2^0, \ldots, 2^{(m+n-1)}\}$. This gives us a function from continuous images to a finite set of addresses $t : \mathcal{I} \times \mathcal{I} \mapsto \{0, 1\}^{m+n}$.

2.3 Conditional branching from unconditional branching

Our model does not have a conditional branching operation as a primitive; it was felt that giving the model the capability for arbitrary comparison of continuous images would rule out any possible implementation. However, we can effect indirect addressing through a combination of program self-modification and direct addressing. We can then implement conditional branching by combining indirect

addressing and unconditional branching. This is based on a technique by Rojas [15] that relies on the fact that $|\mathcal{N}|$ is finite. Without loss of generality, we could restrict ourselves to two possible symbols '0' and '1'. Then, the conditional branching instruction "if (α='1') then jump to address X, else jump to Y" is written as the unconditional branching instruction "jump to address α". We are required only to ensure that the code corresponding to addresses X and Y is always at addresses '1' and '0', respectively. In a 2-D memory (with an extra addressing coordinate in the horizontal direction) many such branching instructions are possible in a single machine.

2.4 A general iteration construct

Our iteration construct is based on the conditional branching instruction outlined in Sect. 2.3. Consider a loop of the following general form, written in some unspecified language,

```
SX
while (e > 0)
  SY
  e := e - 1
end while
SZ
```

where variable `e` contains a nonnegative integer specifying the number of remaining iterations, and `SX`, `SY`, and `SZ` are arbitrary lists of statements. Without loss of generality, we assume that statements `SY` do not write to `e` and do not branch to outside of the loop. If `e` is represented by a unary stack image, this code could be rewritten as

```
SX
while (e.pop() = '1')
  SY
end while
SZ
```

and compiled to a machine as shown in Fig. 4. In this machine, **e** specifies the number of remaining iterations in unary and is encoded as a stack image. A second well-known address **d**, unused by the statements in the body of the loop, holds the value popped from **e** and must be positioned immediately to the left of **e**. The fragment `br | 0 | *d` is shorthand for a piece of indirect addressing code, and means "branch to the instruction at the intersection of column 0 and the row specified by the contents of well-known image location **d**".

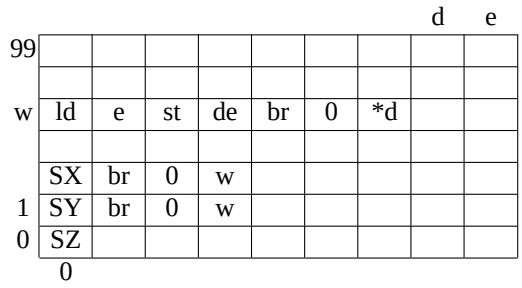

Figure 4: Machine description of a while loop.

2.5 Complexity measures

The standard computational complexity measures of time and space could be used to analyse instances of our machine. TIME would be measured by counting the number of times any of the primitive operations of Fig. 2 was executed. Particular operations could be weighted (**h** could have a different cost to **br**) and parameter values taken into account (**ld** and **st** would have costs proportional to the number of grid elements accessed, and **br** could have a cost proportional to the distance of the jump). This would also accommodate addressing schemes where decoding a numerical value from an image takes TIME proportional to the size of that value. It is important to recognise that the temporal cost of grid element accesses would differ by no more than a constant (worst case being the size of the grid) and does not otherwise depend on the amount or type of information stored in an image.

SPACE could be a straightforward static measure of the number of elements in the grid. Such a measure would include storage of the program and input data. A more standard costing would count the number of grid elements that were overwritten at least once. This latter measure does not count the grid elements reserved for the program or the input. RESOLUTION is a measure of the spatial compression of image data due to rescaling, relative to the input resolution. For optical algorithms that use stack encodings, RESOLUTION is of critical concern. A final measure is RANGE, which is used to describe the amplitude resolution or amplitude quantisation (if applicable) of the images stored in the machine.

In this paper we use a uniform cost measurement for TIME (each primitive operation costs one unit), SPACE is the number of elements in the grid, and RESOLUTION is as defined above.

3 Computability results

In this section we prove our model can simulate ARNNs with real-valued weights.

3.1 Boolean circuits and ARNNs

Let $\Sigma = \{0, 1\}$, let $\Sigma^* = \bigcup_{i=0}^{\infty} \Sigma^i$, and let $\Sigma^+ = \bigcup_{i=1}^{\infty} \Sigma^i$.

Informally, a Boolean circuit, or simply a circuit, is a directed acyclic graph where each node is an element of one of the following three sets: $\{\wedge, \vee, \neg\}$ (called *gates*, with respective in-degrees of 2,2,1), $\{x_1, x_2, \ldots, x_n\}$ ($x_i \in \{0, 1\}$, *inputs*, in-degree 0), $\{0, 1\}$ (*constants*, in-degree 0). A circuit family is a set of circuits $C = \{c_n, n \in \mathbb{N}\}$. A language $L \subseteq \Sigma^*$ is decided by the circuit family C if the characteristic function of the language $L \cap \{0, 1\}^n$ is computed by c_n, for each $n \in \mathbb{N}$. When the circuits are of exponential size (with respect to input word length and where circuit size is the number gates in a circuit) circuit families can decide all languages $L \subseteq \Sigma^*$. It is possible to encode a circuit by a finite symbol sequence, and a circuit family by an infinite symbol sequence. For a more thorough introduction to circuits we refer the reader to [4].

ARNNs are finite size feedback first-order neural networks with real weights [18, 17]. The state of each neuron at time $t + 1$ is given by an update equation of the form

$$x_i(t+1) = \sigma\left(\sum_{j=1}^{N} a_{ij} x_j(t) + \sum_{j=1}^{M} b_{ij} u_j(t) + c_i\right) \quad , \quad i = 1, \ldots, N \qquad (5)$$

where N is the number of neurons, M is the number of inputs, $x_j(t) \in \mathbb{R}$ are the states of the neurons at time t, $u_j(t) \in \Sigma^+$ are the inputs at time t, and $a_{ij}, b_{ij}, c_i \in \mathbb{R}$ are the weights. An ARNN update equation is a function of discrete time $t = 1, 2, 3, \ldots$. The network's weights, states and inputs are often written in matrix notation as $A, B, c, x(t)$, and $u(t)$. The function σ is defined as

$$\sigma(x) := \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } 0 \leq x \leq 1 \\ 1 & \text{if } x > 1 \ . \end{cases} \qquad (6)$$

A subset P of the N neurons, $P = \{x_{i_1}, \ldots, x_{i_p}\}$, $P \subseteq \{x_1, \ldots, x_N\}$, are called the p output neurons. The output from an ARNN computation is defined as the states $\{x_{i_1}(t), \ldots, x_{i_p}(t)\}$ of these p neurons for time $t = 1, 2, 3, \ldots$.

Deciding languages using formal nets

ARNN input/output mappings can be defined in many ways [18]. In this paper we give a simulation of the general form ARNN which has update equation Eq. (5). A specific type of ARNN, called a formal net [18], decides languages. Formal nets are ARNNs with the following input and output encodings. A formal net has two binary input lines, called the input data line (D) and the input validation line (V), respectively. If D is *active* at a given time t then $D(t) \in \{0, 1\}$, where $D(t)$ represents a symbol from the input word to be decided, otherwise $D(t) = 0$. $V(t) = 1$ when D is active, and $V(t) = 0$ thereafter. An input to a formal net at time t has the form $u(t) = (D(t), V(t)) \in \{0, 1\}^2$. A formal net has two output neurons $O_d(t), O_v(t) \in \{x_1, \ldots, x_N\}$ which are called the output data line and output validation line, respectively.

We wish to decide if a word $w \in \Sigma^+$ is in language L. An ARNN to decide L is given w as input. For some t if $O_v(t) = 1$ and $O_d(t) = 1$ then $w \in L$. For some t, if $O_v(t) = 1$ and $O_d(t) = 0$ then $w \notin L$.

In [18] Siegelmann and Sontag prove that for each language $L \in \Sigma^+$ there exists an ARNN N_L to decide L, hence proving the ARNN model to be computationally more powerful than the Turing machine model. N_L contains one real weight. This weight encodes a circuit family that decides L. For a given input word $w \in \Sigma^+$, N_L retrieves the encoding of the corresponding circuit $c_{|w|}$ from its real weight and uses this encoding to decide if w is in L. In polynomial time ARNNs can decide the nonuniform language class $P/poly$. Given exponential time ARNNs can decide all languages.

3.2 Simulation encoding scheme

In our ARNN simulation we use image amplitude values to represent arbitrary real numbers. $r \in \mathbb{R}$ is represented by the amplitude value at coordinates $(0.5, 0.5)$ of an image $f \in \mathcal{I}$; $|f(0.5, 0.5)| = r$. Such an image representing one real number is called a scalar image. A $1 \times C$ matrix image is composed of C amplitude peaks in the horizontal direction, an $R \times 1$ matrix image is composed of R amplitude peaks in the vertical direction, and an $R \times C$ matrix image is composed of $R \times C$ amplitude peaks, as shown in Fig. 3(f).

3.3 ARNN representation

In our notation $\overline{\alpha}$ is the image encoding of symbol α. The ARNN weight matrices A and B are represented by $N \times N$ and $N \times M$ matrix images $\overline{A}$ and $\overline{B}$ respectively. The weight vector c and state vector $x(t)$ are represented by $N \times 1$ matrix images $\overline{c}$ and $\overline{x}(t)$ respectively.

The input vector $u(t)$ is represented by a $1 \times M$ matrix image $\overline{u}(t)$. Before our simulation program begins executing we assume a stack called I encodes all input vectors $u(t)$ for $t = 1, 2, 3, \ldots$. At time t, the top element of stack I is a $1 \times M$ matrix image representing the ARNN input vector $u(t)$. ARNNs are defined over finite length input words w. Therefore I's top n stack elements encode an ARNN input w as n successive $1 \times M$ matrix images and all other elements of the stack encode the value 0.

The p output neurons are represented by an $N \times 1$ matrix image $\overline{P}$. We use $\overline{P}$ to extract the p output states from the N encoded neuron states in $\overline{x}(t)$. $\overline{x}(t)$ contains N amplitude peaks, each at specific coordinates as illustrated in Fig. 3(f). p of these peaks represent the p ARNN output states and have coordinates $\{(x_1, y_1), \ldots, (x_p, y_p)\}$ in $\overline{x}(t)$. In $\overline{P}$ the amplitude values at coordinates $\{(x_1, y_1), \ldots, (x_p, y_p)\}$ each encode the value 1, all other coordinates in $\overline{P}$ have amplitude values encoding 0. We multiply $\overline{x}(t)$ by $\overline{P}$. This multiplication results in an output image $o(t)$ that contains the encoding of the p ARNN outputs at the coordinates $\{(x_1, y_1), \ldots, (x_p, y_p)\}$. $o(t)$ encodes the value 0 at all other coordinates. At each timestep t the simulator pushes $o(t)$ to an output stack O.

3.4 ARNN simulation overview

From the neuron state update equation Eq. (5), each $x_j(t)$ is a component of the state vector $x(t)$. From $x(t)$ we can derive the $N \times N$ matrix $X(t)$ where each column of $X(t)$ is a copy of the vector $x(t)$. $X(t)$ has components $x_{ij}(t)$, where $i, j \in \{1, \ldots, N\}$. From $u(t)$ we can derive the $N \times M$ matrix $U(t)$ where each row of $U(t)$ is a copy of the vector $u(t)$. $U(t)$ has components $u_{ij}(t)$, where $i \in \{1, \ldots, N\}$ and $j \in \{1, \ldots, M\}$. Using $X(t)$ and $U(t)$ we rewrite Eq. (5) as

$$x_i(t+1) = \sigma\left(\sum_{j=1}^{N} a_{ij}x_{ij}(t) + \sum_{j=1}^{M} b_{ij}u_{ij}(t) + c_i\right) \quad , \quad i = 1, \ldots, N \ . \qquad (7)$$

In our simulation we generate $N \times N$ and $N \times M$ matrix images $\overline{X}(t)$ and $\overline{U}(t)$ representing $X(t)$ and $U(t)$ respectively. We then simulate the affine combination in Eq. (7) using our model's $+$ and $\cdot$ operators.

Recall from the model's definition in Sect. 2 that the **ld** and **st** operations effect amplitude filtering. We use this amplitude filtering to simulate the ARNN σ function. From the definition of ρ in Fig. 2, we set $z_{\text{l}} = 0$ and $z_{\text{u}} = 1$ to give

$$\rho(f(i,j),0,1) = \begin{cases} 0, & \text{if } |f(i,j)| < 0 \\ |f(i,j)|\,, & \text{if } 0 \leq |f(i,j)| \leq 1 \\ 1, & \text{if } |f(i,j)| > 1 \ . \end{cases} \tag{8}$$

Using our encoding scheme, $\rho(\overline{x},0,1)$ simulates $\sigma(x)$.

3.5 ARNN simulation algorithm

For brevity and ease of understanding we outline our simulation algorithm in a high-level pseudocode, followed by an explanation of each algorithm step.

(i)	$\overline{u}(t)$	:=	I.pop()
(ii)	$\overline{X}(t)$	:=	push $\overline{x}(t)$ onto itself horizontally $N-1$ times
(iii)	$\overline{AX}(t)$	:=	$\overline{A} \cdot \overline{X}(t)$
(iv)	$\Sigma\overline{AX}(t)$	:=	$\Sigma_{i=1}^{N} \overline{AX}(t)$.pop$_i$()
(v)	$\overline{U}(t)$	:=	push $\overline{u}(t)$ onto itself vertically $N-1$ times
(vi)	$\overline{BU}(t)$	:=	$\overline{B} \cdot \overline{U}(t)$
(vii)	$\Sigma\overline{BU}(t)$	:=	$\Sigma_{i=1}^{M} \overline{BU}(t)$.pop$_i$()
(viii)	affine-comb	:=	$\Sigma\overline{AX}(t) + \Sigma\overline{BU}(t) + \overline{c}$
(ix)	$\overline{x}(t+1)$	:=	ρ(affine-comb$,0,1$)
(x)	O.push$(\overline{P} \cdot \overline{x}(t+1))$		
(xi)	Goto step (i)		

In step (i) we pop an image from input stack I and call the popped image $\overline{u}(t)$. $\overline{u}(t)$ is a $1 \times M$ matrix image that represents the ARNN's inputs at time t. In step (ii) we generate the $N \times N$ matrix image $\overline{X}(t)$ by pushing $N-1$ identical copies of $\overline{x}(t)$ onto a copy of $\overline{x}(t)$. In step (iii), $\overline{X}(t)$ is point by point multiplied by matrix image $\overline{A}$. This single multiplication of two matrix images efficiently simulates (in parallel) the matrix multiplication $a_{ij}x_j(t)$ for all $i,j \in \{1,\ldots,N\}$ (as described in Sect. 3.4). Step (iv) simulates the ARNN summation $\sum_{j=1}^{N} a_{ij}x_j(t)$. Each of the N columns of $\overline{AX}(t)$ are popped and added (using the $+$ operation) to the previous expanded image.

In step (v) we are treating $\overline{u}(t)$ in a similar way to our treatment of $\overline{x}(t)$ in step (ii), except here we push vertically. In step (vi) we effect $\overline{B} \cdot \overline{U}(t)$, efficiently

simulating (in parallel) the multiplication $b_{ij}u_j(t)$ for all $i \in \{1, \ldots, N\}, j \in \{1, \ldots, M\}$. Step (vii) simulates the ARNN summation $\sum_{j=1}^{M} b_{ij}u_j(t)$ using the same technique used in step (iv).

In step (viii) we simulate the addition of the three terms in the ARNN affine combination. In our simulator this addition is effected in two simple image addition steps. In step (ix) we simulate the ARNN's σ function by amplitude filtering using our ρ function with the parameters $(0, 1)$ as described in Sect. 3.4. We call the result of this amplitude filtering $\overline{x}(t+1)$; it represents the ARNN state vector $x(t+1)$. In step (x) we multiply $\overline{x}(t+1)$ by the output mask $\overline{P}$ (as described in Sect. 3.3). The result, which represents the ARNN output at time $(t+1)$ is then pushed to the output stack O. The final step in our algorithm sends us back to step (i). Notice our algorithm never halts as ARNNs are defined for time $t = 1, 2, 3, \ldots$.

Figure 5: Simulating an ARNN on our model of computation. The machine is in two parts for clarity. The larger is a universal ARNN simulator and the smaller is the inserted ARNN. The simulator is written in a compact shorthand notation. The expansions into sequences of atomic operations are shown in Fig. 6 and the simulation program is explained in Sect. 3.6.

(a)

ld	I	→	ld	5	5	99	99	0	/	1	1	/	1
st	t1a	→	st	11	12	99	99	0	/	1	1	/	1
st	I	→	st	5	5	99	99	0	/	1	1	/	1
ld	t1	→	ld	11	11	99	99	0	/	1	1	/	1
st	u(t)	→	st	1	1	99	99	0	/	1	1	/	1
ld	x(t)	→	ld	0	0	99	99	0	/	1	1	/	1
st	t1	→	st	11	11	99	99	0	/	1	1	/	1
ld	t1a	→	ld	11	12	99	99	0	/	1	1	/	1
st	b	→	st	13	13	99	99	0	/	1	1	/	1
ld	A	→	ld	11	11	0	0	0	/	1	1	/	1
st	ab	→	st	12	13	99	99	0	/	1	1	/	1
ld	b	→	ld	13	13	99	99	0	/	1	1	/	1
st	bt2	→	st	13	14	99	99	0	/	1	1	/	1
ld	t2	→	ld	14	14	99	99	0	/	1	1	/	1
st	$\Sigma AX(t)$	→	st	2	2	99	99	0	/	1	1	/	1
ld	u(t)	→	ld	1	1	99	99	0	/	1	1	/	1
st	t3	→	st	12	12	14	14	0	/	1	1	/	1
ld	at3	→	ld	12	12	14	99	0	/	1	1	/	1
ld	B	→	ld	12	12	0	0	0	/	1	1	/	1
ld	$\overline{\Sigma AX}(t)$	→	ld	2	2	99	99	0	/	1	1	/	1
ld	c	→	ld	0	0	13	13	0	/	1	1	/	1
ld	$\overline{P}$	→	ld	0	0	14	14	0	/	1	1	/	1
ld	O	→	ld	6	6	99	99	0	/	1	1	/	1
st	O	→	st	6	6	99	99	0	/	1	1	/	1

(b)

whl	N-2	...	end

Figure 6: Time-saving shorthand conventions. (a) Short-hand instructions used in the simulator in Fig. 5. (b) Expands to initialisation instructions and the while loop code given in Fig. 4.

3.6 Explanation of Figs. 5 and 6

The ARRN simulation with our model is shown in Fig. 5. The numerals (i)–(xi) are present to assist the reader in understanding the program; they correspond to steps (i)–(xi) in the high-level pseudocode in Sect. 3.5. Our ARNN simulator program is written in a shorthand notation (including shorthand versions of the operations **ld**, **st**, and **br** from Fig. 2) that is expanded using Fig. 6. Before the simulator begins executing a simple preprocessor or compiler could be used to update the shorthand to the standard long-form notation.

The machine consists of two parts (separated in the diagram for clarity). The larger is the universal ARNN simulator. Addresses t_1, t_2, and t_3 are used as temporary storage locations during a run of the simulator [note: address t_3 is located at grid coordinates $(12, 14)$]. In the simulator our $\overline{\alpha}$ notation denotes the image encoding of α, and also acts as an address identifier for the image representing α. Locations $\overline{x}(t)$ and $\overline{u}(t)$ are used to store our representation of the neurons' states and inputs during a computation. $\Sigma\overline{AX}(t)$ is a temporary storage location used to store the result of step (iv). Locations $\overline{N}$ and $\overline{M}$ store our representation of the dimensions of $x(t)$ and $u(t)$ (necessary for bounding our while loops). The contents of $\overline{N}$ and $\overline{M}$ must be supplied as input to the simulator. At time t, our representation of the ARNN input $u(t)$ is at the top of the stack I. This input is popped off the stack and placed in memory location $\overline{u}(t)$. The computation then proceeds as described by the high-level pseudocode algorithm in Sect 3.5. The output memory location O stores the sequence of outputs as described in Sect. 3.3. Program execution begins at well-known location **sta** and proceeds according to the rules for our model's programming language which are given in Sect. 2.

The smaller part of Fig. 5 illustrates how an ARNN definition must be inserted into the universal ARNN simulator. The address identifiers $\overline{A}$, $\overline{B}$ and $\overline{c}$ store our encoding of the corresponding ARNN matrices, and $\overline{P}$ stores our mask for extracting the p output states from the N neuron states, as described in Sect. 3.3.

The code fragment 

is shorthand for code to initialise and implement the while loop given in Sect. 2.4. The instructions between `ctr` and `end` are executed `ctr` times. The `whl` routine has TIME complexity $4+\text{ctr}(s+4)$, RESOLUTION complexity $2^{\text{ctr}} + \text{maxres}$, and constant SPACE complexity, where `ctr` $\in \mathbb{N}$ is the number of times the body of the while loop is executed, s is the number of operations in the body of the while loop, and maxres is the maximum resolution of any image accessed during execution of the code for `whl`.

3.7 Complexity analysis of simulation algorithm

In our simulation pushing (or popping) p scalar images to (or from) a $p \times 1$ or $1 \times p$ 1-D matrix image requires TIME $O(p)$, RESOLUTION $O(2^{p-1})$, and constant SPACE. Pushing q $p \times 1$ or $1 \times p$ 1-D matrix images to form a $p \times q$ or $q \times p$ 2-D matrix image requires TIME $O(q)$, RESOLUTION $O(2^{p+q-2})$, and constant SPACE. If the ARNN being simulated has time $t = 1, 2, 3, \ldots$, M as the length of the input vector $u(t)$ and N neurons, and n is the number of image stack elements used to encode the finite input to our simulator, then our simulation program takes TIME

$$T(N, M, t, n) = t(21N + 9M + 16) + 1 \tag{9}$$

Our simulation takes TIME linear in N, M and t, and independent of n. It takes constant SPACE, and exponential RESOLUTION

$$R(N, M, t, n) = \max(2^{(n-t+M-1)}, 2^{(2N-2)}, 2^{(N+M-2)}, 2^{(N+t-1)}) \ . \tag{10}$$

3.8 Deciding languages

Let us assume we are simulating a formal net F and the language decided by F is called L. On input word w, F decides if w is in L in time $t_F(w)$, that is for F's output validation line O_v, $O_v(t_F(w)) = 1$. Simulating F (on input w) with our ARNN simulator takes linear TIME $T(N_F, M_F, t_F(w), n_w)$, exponential RESOLUTION $R(N_F, M_F, t_F(w), n_w)$ and constant SPACE to produce our representation of $O_v(t_F(w)) = 1$. By way of ARNN simulation our model decides all languages with these complexity bounds.

4 Complexity Results

Sorting and searching [10] provide standard challenges to computer scientists in the field of algorithms, computation, and complexity. In this paper we focus on a binary search algorithm (with our model this algorithm can be applied to unordered lists) and an implementation of the bitonic sort, first introduced by Batcher [5] as one of the fastest sorting networks.

4.1 Unordered search

Consider an unordered list of n elements. For a given property, each element could be represented by a bit key denoting whether the element satisfies that property or

```
procedure search(i1, i2)
  e := i2
  c := '0'
  while (e.pop() = '1')
    ab := i1
    select (∫∫a)
    case '1':
      i1 := a
      c.push('0')
    case '0':
      i1 := b
      c.push('1')
    end select
  end while
  a := c
end
```

Figure 7: Algorithm to perform a $\Theta(\log_2 n)$ search on an unsorted binary list.

not. If only one element satisfies that property, the problem of finding its index becomes one of searching an unsorted binary list for a single '1'. This problem was posed by Grover in [9]. His quantum computer algorithm requires $O(\sqrt{n})$ comparison operations on average. Bennett et al. [6] have shown the work of Grover is optimal up to a multiplicative constant, and that in fact any quantum mechanical system will require $\Omega(\sqrt{n})$ comparisons. Algorithms for conventional models of computation require $\Theta(n)$ comparisons in the worst case to solve the problem. We present an algorithm that requires $\Theta(\log_2 n)$, in the worst case, with a model of computation that has promising future implementation prospects.

The algorithm in Fig. 7 takes two inputs, one a list image and the other a stack image. (Unary and binary stack images and list images were introduced in Sect. 2.1.) The first input, **i1**, is the binary list of length n, represented by a list image with n equally spaced amplitude peak positions in the horizontal direction. The image contains only one peak (a '1') and the rest of the positions (and the rest of the image) has zero amplitude ('0's). We assume that n is a power of 2. The number $\log_2 n$ is also supplied as input (**i2**). This is used to bound the iteration and is stored in unary in a stack image. The algorithm uses a well-known location **c** as it constructs, bit by bit, the index of the only '1' in **i2**. This index is stored in binary in a stack image. This index is returned through location **a** when the algorithm terminates.

Figure 8: Machine to perform a $\Theta(\log_2 n)$ search on an unsorted binary list.

An instance of our model implementing this algorithm is shown in Fig. 8. In this machine, the computation begins by branching to location $(0, 3)$ to execute the two assignment statements before evaluating the loop guard. The code in row 2 corresponds to the line "`select (∫∫a)`" from the algorithm above. This operation transforms the list in **a** into the symbol '1' if there had been a '1' anywhere in the list, and into the symbol '0' otherwise. It does this by integrating all of the power in **a** and positioning it at **a**'s centre. The computation halts as soon as the stack at location **e** is exhausted and control branches to location $(0, 0)$.

4.2 Bitonic sort

Bitonic sort is an important algorithm for facilitating efficient parallel implementations because the sequence of comparisons is not data-dependent. Bitonic sort consists of $O(n(\log_2 n)^2)$ comparisons in $O((\log_2 n)^2)$ stages. This is non-optimal – [1] identifies a sorting network with only $O(n \log_2 n)$ comparisons but a very large constant factor. However, bitonic sort continues to be faster than theoretically optimal solutions for all practical problem sizes.

In [3, 2] Akl provides a review of the history of parallel sorting and classifies the bitonic sort as being one of many compare-and-exchange (CAE) based sorts. Such sorts can be efficiently implemented on many different parallel architectures, whose commonality is the CAE component as a fundamental building block. In [16] we see an optimal CAE sort algorithm for mesh-connected computers which requires enormous electronic resources: this solution is complex and difficult to build due to the electronic nature of the underlying implementation architecture. Stirk and Athale [19] provide one of the first treatments of a parallel CAE sort with partial optical implementation. The optics eliminates communication bottlenecks and interconnection problems by providing massive parallelism free of electromagnetic interference. This work is built upon in [7] where the bitonic sort is implemented on a parallel sorting architecture using hybrid optoelectronic technology. This system is proposed as a functional extension of electronic computers and the main originality is in the smart optical interface devices between parallel optical memories and electronic processors (the authors do not advocate the development of an all-optical computer). In [11] a constant time optical CAE sort is introduced. Their claim of constant time suggests that the complexity of the algorithm is constant. However, there are upper bounds on the number of elements that can be sorted.

We propose a novel optical implementation of the bitonic sort on our model. A Java sequential implementation of our bitonic sort is given in Appendix A.

(Bitonic sorting is similar to MergeSort in that it requires a list of elements to be split into two even sublists, recursively sorted and then merged. However, in the bitonic merge step the lists to be sorted are bitonically ordered rather than ordered.)

We followed a technique similar to [8] in order to verify the correctness of our algorithm. The complexity of the sequential implementation is $O(n(\log_2 n)^2)$. The complexity of the implementation on our optical computer is dependent on the complexity of the `bitonicMerge` operation. As with a classical multiprocessor system (where the number of processors required is linear with respect to the size of the lists to be merged) the `bitonicMerge` can be implemented optically with complexity $O(\log_2 n)$, given that we can provide a fully parallel implementation of the CAEs contained within the iteration (see comment in the code in Appendix A). In this case, the depth of our bitonic sort computation is defined by the recurrence relation $d(n) = d(n/2) + \log_2 n$. This corresponds to a $O((\log_2 n)^2)$ implementation – the same as is seen with the multiprocessor implementations referenced above. The advantage of our proposed implementation over the electronic and optoelectronic implementations is that we execute the algorithm on a single (optical) processor, where there is no theoretical bound on the number of elements in our input list. (Although optoelectronic devices may be required for input and output, no electronic circuitry is necessary for the computation.) In the case of the implementations we reviewed that contain electronic circuitry, an upper bound is hard-wired into the computation. Furthermore, our optical implementation can be viewed as a software solution, running on a universal computer, not requiring dedicated hardware resources.

References

[1] M. Ajati, J. Kmolos, and S. Szemeredi. An $O(N \log N)$ sorting network. In *Proc. 25th ACM Symposium on Theory of Computing*, pages 1–9, 1983.

[2] S.G. Akl. *Parallel Sorting Algorithms*. Academic Press Inc., 1985.

[3] S.G. Akl. *Parallel Computation Models and Methods*. Prentice Hall, 1997.

[4] José Luis Balcázar, Josep Díaz, and Joaquim Gabarró. *Structural Complexity*, volume 1. Springer-Verlag, Berlin, 1988.

[5] K.E Batcher. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Computing Conference*, volume 32, pages 307–314, 1968.

[6] Charles H. Bennett, Ethan Bernstein, Gilles Brassard, and Umesh Vazirani. Strengths and weaknesses of quantum computing. *SIAM Journal on Computing*, 26(5):1510–1523, 1997.

[7] F.R. Beyette, P.A. Mitkas, and C.W. Wilmsen. Bitonic sorting using an optoelectronic recirculating architecture. *Applied Optics*, 33(35):8164–8172, 1994.

[8] R. Couturier. Formal engineering of the bitonic sort using PVS. In *2nd Irish Workshop on Formal Methods*, Cork, Ireland, July 1998. BCS electronic workshops in computing (eWiC).

[9] L. K. Grover. A fast quantum mechanical algorithm for database search. In *Proc. 28th Annual ACM Symposium on Theory of Computing*, pages 212–219, may 1996.

[10] D.E Knuth. *The Art of Computer Programming, Vol 3: Sorting and Searching*. Addison-Wesley, 1973.

[11] A. Louri, J.A. Hatch, and J. Na. Constant-time parallel sorting algorithm and its optical implementation using smart pixels. *Applied Optics*, 34(17):3087–3097, 1995.

[12] Thomas Naughton, Zohreh Javadpour, John Keating, Miloš Klíma, and Jiří Rott. General-purpose acousto-optic connectionist processor. *Optical Engineering*, 38(7):1170–1177, July 1999.

[13] Thomas J. Naughton. A model of computation for Fourier optical processors. In Roger A. Lessard and Tigran Galstian, editors, *Optics in Computing 2000*, Proc. SPIE vol. 4089, pages 24–34, Quebec, Canada, June 2000.

[14] Thomas J. Naughton and Damien Woods. On the computational power of a continuous-space optical model of computation. In Maurice Margenstern and Yurii Rogozhin, editors, *Machines, Computations and Universality: Third International Conference*, volume 2055 of *Lecture Notes in Computer Science*, pages 288–299, Chişinău, Moldova, May 2001.

[15] Raúl Rojas. Conditional branching is not necessary for universal computation in von Neumann computers. *Journal of Universal Computer Science*, 2(11):756–768, 1996.

[16] C.P. Schnorr and A. Shamir. An optical sorting algorithm for mesh-connected computers. In *Proc. 18th ACM Symposium on Theory of Computing*, pages 255–261, 1986.

[17] Hava T. Siegelmann. *Neural networks and analog computation: beyond the Turing limit*. Progress in theoretical computer science. Birkhäuser, Boston, 1999.

[18] Hava T. Siegelmann and Eduardo D. Sontag. Analog computation via neural networks. *Theoretical Computer Science*, 131(2):331–360, September 1994.

[19] C.W. Stirk and R.A. Athale. Sorting with optical compare-and-exchange modules. *Applied Optics*, 27(9):1721–1726, 1988.

A Bitonic sort code

```
// Java sequential implementation of bitonic sort
// Parallelisation is commented in code
// The array to be sorted is defined by variable a
// This array must contain 2^k elements

public void sort(){bitonicSort(0, size, INC);}

private void bitonicSort(int low, int high, boolean dir){
     if (high>1){
         int mid=high/2;
         bitonicSort(low, mid, INC); bitonicSort(low+mid, mid, DEC);
         bitonicMerge(low, high, dir);
     }
 }

private void bitonicMerge(int low, int high, boolean dir){
     if (high>1){
         int mid=high/2;
         for (int i=low; i<low+mid; i++)  // in parallel
             CAE(i, i+mid, dir);
         bitonicMerge(low, mid, dir); bitonicMerge(low+mid, mid, dir);
     }
 }
```

```
// The compare and exchange (CAE) fundamental component
private void CAE(int i, int j, boolean dir){
     if (dir==(a[i]>a[j])){
         // if in wrong order then swap
         int temp=a[i];a[i]=a[j];a[j]=temp;
     }
 }
```