Parallel Programming on Hopfield Nets

Pekka Orponen and **Frédéric Prost**

Department of Computer Science
University of Helsinki
P. O. Box 26, FIN–00014 Helsinki, Finland
E-mail: orponen@cs.helsinki.fi

Département de Mathématiques et d'Informatique
Ecole Normale Supérieure de Lyon
69364 Lyon cedex 07, France
E-mail: fprost@lip.ens-lyon.fr

Abstract.

We describe a simple general purpose condition-action type parallel programming language and its implementation on Hopfield-type neural networks. A prototype compiler performing the translation has been implemented.

Keywords: neural networks, Hopfield nets, parallel programming

1 Introduction

In addition to their uses in specific applications such as pattern classification, associative memory or combinatorial optimization recurrent neural networks are also theoretically universal, i.e., general-purpose computing devices. It was observed in [7] that sequences of polynomial-size recurrent threshold logic networks are computationally equivalent to (nonuniform) polynomial space-bounded Turing machines, and in [11] this equivalence was extended to polynomial-size sequences of networks with symmetric weights, i.e., "Hopfield nets" with hidden units. A corollary to the latter construction shows also that polynomial-size symmetric networks with small (i.e., polynomially bounded) interconnection weights are equivalent to polynomial *time*-bounded Turing machines. In [17] similar characterizations are proved concerning the computational power of asymmetric bounded-size networks with real-valued units and a saturated-linear transfer function. (All the networks discussed here and below have discrete synchronous dynamics, i.e., the states of all the units are updated in parallel discrete steps. However, see [12] for a simulation of synchronous Hopfield nets by totally asynchronous ones.)

An obvious next step after establishing such a universality result is to design a high-level programming language and a compiler appropriate for the model.

For the analog asymmetric network model of [17] this was done in [16]; here we describe our approach to high-level programming of standard discrete symmetric Hopfield nets. Although the idea of high-level programming of neural nets may seem counterintuitive at first, we believe that methods for doing this will eventually be quite useful, considering the rapid progress in neural network hardware (see, for instance, the work presented in the compendium [15]).

A direct approach to the compilation would have been to simply implement the Turing machine simulation scheme from [11]. This would, however, have been a waste of the networks' capabilities, as this simulation forces the potentially highly parallel network to behave in a purely sequential manner. Thus we chose rather to experiment with a very simple parallel language, where a program consists of a set of "*condition* $\rightarrow$ *action*" pairs, to be executed repeatedly in parallel until a "**halt**"-action is encountered. The only operations performed by our programs are elementary arithmetic and left and right shifts on fixed-length integers, and the only conditions allowed are conjunctions and disjunctions of numerical comparisons. The language should obviously be enriched considerably to be of any practical use; nevertheless even this prototype version is computationally universal (within the limits set by the chosen integer length), and suffices to try out the basic compilation principles.

Finally, a few words on the broader motivation of this work. While the idea of high-level programming of neural nets may seem counterintuitive (after all, these are supposedly *learning* devices), there are at least two good reasons to consider the topic. Firstly, learning of concepts with high-level structure purely from raw data is very difficult (see, e.g. [6]), and thus it might be useful to have a method for initializing a network with an approximate solution, and use learning only for the fine-tuning of its parameters. Secondly, work on hardware implementations of neural network models progresses rapidly (see, for instance, the compendium [15]), and eventually ideas about high-level progamming methods for them will be very useful. (We might note that these motivations underlie, at least partly, also the by now quite large literature on neural representation of symbolic knowledge and "hybrid" neural-symbolic models; see, e.g. [4, 8, 9, 10, 14, 18].)

2 Mapping parallel programs onto asymmetric networks

The compilation of programs from our simple parallel language into Hopfield networks proceeds in two stages: first the program is represented as a recurrent threshold logic network with asymmetric interconnections, and then this asymmetric network is "symmetrized" using the construction from [11]. Here we shall first describe the mapping into asymmetric networks, and then briefly the symmetrization procedure.

The syntax of our simple language is as described in Figure 1. As an element-

program	=	**var** *list of names*: **length** *integer*; **pardo** statement {; statement}* **parend**
statement	=	condition → action{, action}*
condition	=	*Boolean combination of* comparison*'s*
comparison	=	variable $\{< \mid \leq \mid = \mid \geq \mid >\}$ variable
action	=	assignment | **halt**
assignment	=	variable := expression
expression	=	*arithmetic expression* | **shl** *variable* | **shr** *variable*

Figure 1: Syntax of a simple parallel language.

ary translation example, let us consider the following program with two 8-bit integer variables x and y. Starting from any pair of nonnegative initial values (satisfying the appropriate length constraints), the program simply adjusts the values of the variables until they are equal, and then halts.

var x, y: **length** 8;
pardo
$x < y \quad \rightarrow \quad x := x + y,\ y := y - 1;$
$x > y \quad \rightarrow \quad x := x - y;$
$x = y \quad \rightarrow$ **halt**
parend.

Schematically, this program would first be translated into the asymmetric recurrent network shown in Figure 2. The boxes labeled x and y represent the 8-bit variable registers, whose contents are first initialized with the input values of the variables, and then repeatedly updated until a halting-test module (not shown) determines that the values are equal and forces the network to converge. The thick lines in Figure 2 correspond to 8-bit data lines around the network, and the thin lines to single-bit controls. Note that *all* the possible assignments for each variable are computed in parallel on each pass of data around the network, and only at the end of the pass the control lines are used to select the correct assignment, within a simple multiplexer (`MPX`) subcircuit. In case none of the selector lines are active, a multiplexer implicitly selects the old value of the variable (the bottom input lines to the x and y multiplexers in Figure 2). If many of the selector lines are active simultaneously, the result of the multiplexing is undetermined.

In addition to the registers and the comparison and multiplexer modules, Figure 2 shows three 8-bit adders (`ADD`), one module (`NEG`) for transforming an 8-bit number into its negative in a two's complement representation, and one constant input module (-1). All these subcircuits are quite easy to build using threshold logic units: for instance the comparisons require just one unit each, and each multiplexer consists of 8 two-way one-bit selectors, each of size 3, plus the additional

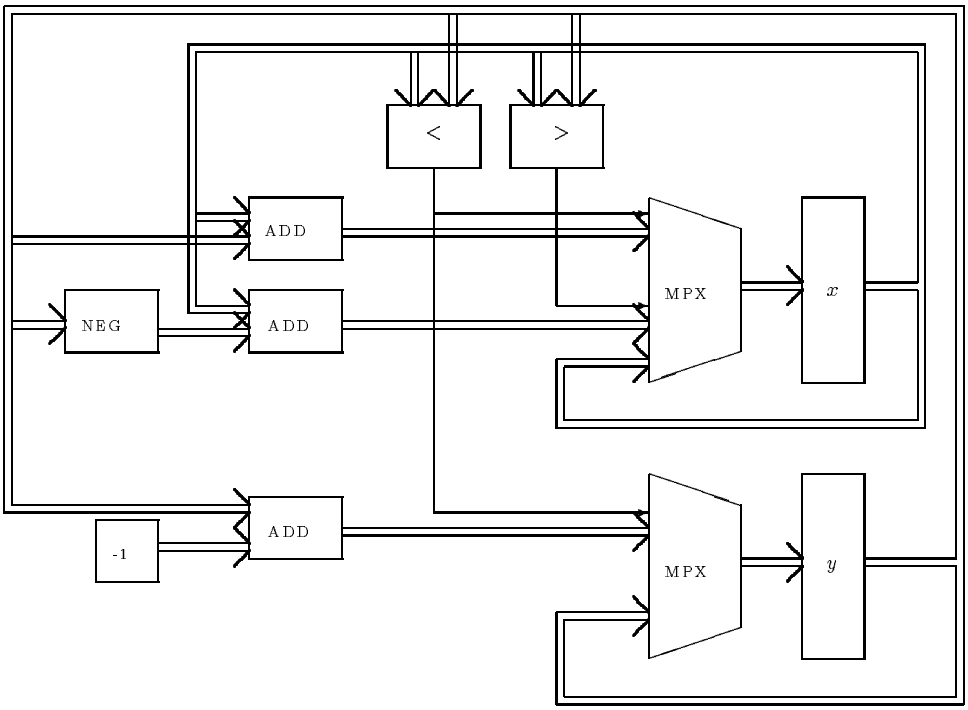

Figure 2: Asymmetric net corresponding to example program.

circuitry needed to implement the default selection. The most complicated modules are the adders, whose implementation we moreover have not even attempted to optimize. For an 8-bit adder we simply chain together 8 single-bit full adders. (This is perhaps the main aspect of our compiler that needs improvement. The current design is bad not only because it consumes a lot of units, but also because it introduces long delays into the network and complicates its sequencing.) In addition to the modules shown in the figure, the network also contains a certain amount of control circuitry, e.g. for implementing the **halt** operation, and ensuring that all the inputs required by a module arrive at the same time.

As an example of threshold logic design, Figure 3 shows a full adder that adds bits x_i, y_i, and c_i, to produce output bit z_i and carry bit c_{i+1}. All the weights in the first layer of the circuit are 1.

3 Mapping asymmetric networks onto symmetric ones

The procedure for transforming convergent asymmetric nets into symmetric ones is discussed in detail in [11]; here we provide only an outline. In the transformation, each asymmetric edge in the original network is first replaced by a sequence of three symmetric edges and their intermediate units, as indicated in Figure 4.

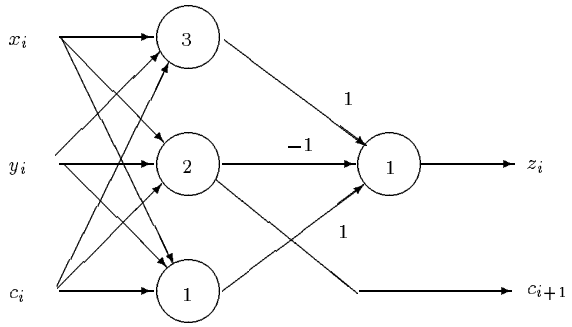

Figure 3: A single-bit full adder subcircuit.

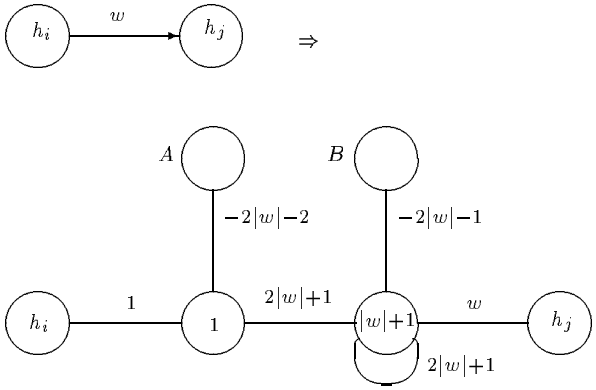

Figure 4: A sequence of symmetric edges corresponding to an asymmetric edge of weight w.

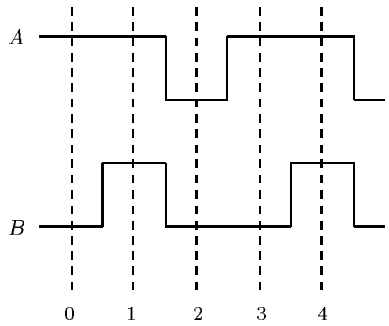

Figure 5: The clock pulse sequence used in the edge simulation.

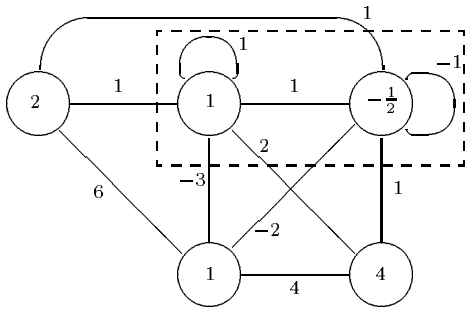

Figure 6: A three-bit binary counter network.

The two intermediate units act like locks in a canal, permitting information to flow only from left to right. The locks are sequenced by clock pulses emanating from the units labeled A and B, in cycles of period three as presented in Figure 5. (This edge simulation technique was originally introduced, in a somewhat more complicated form, by Hartley and Szu in [3].)

Because symmetric nets always converge to either a stable state or a cycle of period two [2], a proper clock of period three cannot actually be constructed in a symmetric net. However, to simulate a *convergent* computation of the original asymmetric net, it suffices to have a clock that produces an exponential number of periods and then stops. (Note that a convergent synchronous computation on n units must terminate within 2^n steps, because otherwise the network repeats a configuration and goes into a cycle.) Such a clock can be obtained from, e.g., a symmetric exponential-transient network designed by Goles and Martínez [1]. The first two stages in the construction of this network are presented in Figure 6.

The idea here is that the n units in the upper row implement a binary counter, counting from all 0's to all 1's (in the figure, the unit corresponding to the least significant bit is to the right). For each "counter" unit added to the upper row, after the two initial ones, two "control" units are added to the lower row. The purpose of the latter is to first turn off all the "old" units, when the new counter unit is activated, and from then on balance the input coming to the old units from the new units, so that the old units may resume counting from zero.

The required period-three clock may now be obtained from the second counter unit of this network by means of an appropriate delay line construction. In building this connection, the weights and the thresholds in the counter network must also be multiplied by some sufficiently large constant so that the rest of the network has no effect back on the clock. For more details of the construction, see [11].

4 Conclusion and further work

We have designed and implemented an experimental compiler from a very simple condition-action type parallel programming language into standard symmetric discrete Hopfield networks with synchronous dynamics. The programming language should be extended considerably to be practical, and some of the design decisions in the compiler (notably the simplistic implementation of arithmetic modules) have turned out to be less than optimal. Nevertheless, this is to our knowledge the first attempt at general-purpose high-level programming on this standard and theoretically well-studied computational model.

References

[1] Goles, E., Martínez, S. Exponential transient classes of symmetric neural networks for synchronous and sequential updating. *Complex Systems 3* (1989), 589–597.

[2] Goles, E., Olivos, J. Comportement périodique des fonctions à seuil binaires et applications. *Discr. Appl. Math. 3* (1981), 93–105.

[3] Hartley, R., Szu, H. A comparison of the computational power of neural networks. In: *Proc. of the 1987 Internat. Conf. on Neural Networks, Vol. 3.* IEEE, New York, 1987. Pp. 15–22.

[4] Hinton, G. E. (ed.) *Connectionist Symbol Processing.* (Reprint of *Artificial Intelligence 46:1–2* (1990).) The MIT Press, Cambridge, Ma., 1991.

[5] Hopfield, J. J. Neural networks and physical systems with emergent collective computational abilities. *Proc. Nat. Acad. Sci. USA 79* (1982), 2554–2558.

[6] Kearns, M., Valiant, L. Cryptographic limitations on learning Boolean formulae and finite automata. *J. Assoc. Comput. Mach. 41* (1994), 67–95.

[7] Lepley, M., Miller, G. Computational power for networks of threshold devices in an asynchronous environment. Unpublished manuscript, Dept. of Mathematics, Massachusetts Inst. of Technology, 1983.

[8] Levine, D. S., Aparicio, M. A., IV (eds.) *Neural Networks for Knowledge Representation and Inference.* Lawrence Erlbaum, Hillsdale, N.J., 1994.

[9] Myllymäki, P., Orponen, P. Programming the Harmonium. In: *Proceedings, International Joint Conf. on Neural Networks (Singapore, November 1991), Vol. I.* IEEE, New York, 1991. Pp. 671–677.

[10] Myllymäki, P., Orponen, P., Silander, T. Integrating symbolic reasoning with neurally represented background knowledge. In: *Proceedings, AAAI-92 Workshop on Integrating Neural and Symbolic Processes (San Jose, Ca., July 1992)*, 168–172.

[11] Orponen, P. The computational power of discrete Hopfield nets with hidden units. *Neural Computation 8* (1996), 403–415.

[12] Orponen, P. Computing with truly asynchronous threshold logic networks. *Theoretical Computer Science,* to appear.

[13] Parberry, I. (ed.) *The Computational and Learning Complexity of Neural Networks: Advanced Topics.* The MIT Press, to appear.

[14] Plate, T. A. Holographic reduced representations: Convolution algebra for compositional distributed representations. In: *Proceedings, 12th International Joint Conf. on Artificial Intelligence (Sydney, August 1991)*, 30–35.

[15] Sánchez-Sinencio, E., Lau, C. *Artificial Neural Networks: Paradigms, Applications, and Hardware Implementations.* IEEE Press, New York, 1992.

[16] Siegelmann, H. T. A neural programming language. In: *Proc. of the 12th National Conf. on Artificial Intelligence (Seattle, Wa., July 1994), Vol. 2.* AAAI Press/The MIT Press, 1994. Pp. 877–882.

[17] Siegelmann, H. T., Sontag, E. D. Analog computation via neural networks. *Theoretical Computer Science 131* (1994), 331–360.

[18] Sun, R., Bookman, L. A. (eds.) *Computational Architectures Integrating Neural and Symbolic Processes.* Kluwer Academic, Boston, Ma., 1995.