

Optimizing Neural Networks on SIMD Parallel Computers

Andrea Di Blas* Arun Jagota Richard Hughey

Department of Computer Engineering
Baskin School of Engineering
University of California, Santa Cruz

February 22, 2005

Abstract

Hopfield neural networks are often used to solve difficult combinatorial optimization problems. Multiple restarts versions find better solutions but are slow on serial computers. Here, we study two parallel implementations on SIMD computers of multiple restarts Hopfield networks for solving the maximum clique problem. The first one is a fine-grained implementation on the Kestrel Parallel Processor, a linear SIMD array designed and built the University of California, Santa Cruz. The second one is an implementation on the MasPar MP-2 according to the “SIMD Phase Programming Model”, a new method to solve asynchronous, irregular problems on SIMD machines. We find that the neural networks map well to the parallel architectures and afford substantial speedups with respect to the serial program, without sacrificing solution quality.

Keywords: Combinatorial Optimization, Hopfield Neural Networks, Single Instruction-Multiple Data Parallel Computer, Maximum Clique, Parallel Programming Models for Irregular Problems

Introduction

A *clique* in a graph is a subgraph that is completely connected. The maximum clique problem is the NP-hard problem of finding the largest clique in a given graph [12].

*Corresponding author: e-mail: andrea@cse.ucsc.edu

This problem has many practical applications in science and engineering [7]. These include computer-aided circuit design, computer vision [3], protein structure matching [38], distributed fault diagnosis in multiprocessor systems [5], and constraint satisfaction problems [20].

Not only is the problem difficult to solve exactly, it is also NP-hard to solve near-optimally. For this reason, there are no algorithms that solve the problem (not even near-optimally) in reasonable time. Hence, heuristic methods abound.

In this paper, we consider a discrete Hopfield neural network approach to the maximum clique problem. A sequential implementation of this network was presented by Jagota [21]. Here we design parallel implementations. We exploit the fact that the neural network is well-suited to parallel implementation on a Single Instruction–Multiple Data (SIMD) computer. Specifically, we design and evaluate two parallel implementations, one on a MasPar parallel computer, and one on a specialized array processor, Kestrel, designed and built at UCSC. We find that both these implementations afford great speedups over the serial implementation.

Previous Work

Over the past few decades, there have been a number of different approaches to the maximum clique problem [7]. It is impossible to discuss them all here, hence we restrict ourselves to those that use neural networks and to previous work on parallel implementations of neural networks. More broadly, Smith offers a review of the research in the field of neural networks for combinatorial optimization, with a comprehensive bibliography [36].

The earliest encoding of the maximum clique problem in a Hopfield network [17] can be found in the work by Ballard et al. [4]. They employ a discrete Hopfield model in which r of the N units are updated simultaneously, thus spanning from fully asynchronous ($r = 1$) to fully synchronous ($r = N$). Their focus, however, is on studying the convergence rate as a function of r , measured by the number of parallel updates, rather than on the quality of solution. Ramanujam and Sadayappan [31] proposed an encoding that employs non-binary weights and admits invalid solutions, but no experimental results are reported. In the work by Lin and Lee [25] an N -vertex graph is mapped onto a “maximum neural network” with $2(N + 1)$ neurons. Even though the network is said to admit an energy function, the energy function is not used to enforce the constraints in the evolution.

The algorithm described in this paper is based on the work of Jagota [20, 21, 23]. The main differences with previous works can be summarized as follows: i) an N -vertex graph is mapped onto an N -node network, ii) weights and states are binary, iii) no infeasible solutions are admitted, and iv) the evolution is based on an energy function.

The parallel implementation of neural networks has been an active research area

because neural networks intrinsically exhibit concurrency. Lin et al. present a model for mapping multi-layer perceptron neural networks onto SIMD parallel computers [26]. Prechelt [30] explores compiler techniques to increase the efficiency of neural network classifiers on parallel computers. He offers results obtained on a MasPar MP-1 SIMD parallel computer. Misra has written a good survey with a rich bibliography, on parallel implementations of different neural network [27]. There do not appear to be any SIMD implementations of neural network solutions to the maximum clique problem.

Bertoni, Campadelli and Grossi [6] present an interesting algorithm for finding a large clique using a Hopfield network. They also provide a small-scale parallel implementation of this algorithm. The reported experimental results are quite good. For all of the above reasons, we think that this work is important to review in detail, which is what we do in the next few paragraphs.

Their algorithm is inspired by an algorithm developed by one of us (Jagota [21]) called ρ -annealing, so let us review that first. Like most other neural algorithms, ρ -annealing maps the clique problem to a Hopfield network whose node set equals the node set of the given graph. If nodes i and j are adjacent in the given graph, then the weight w_{ij} in the network is set to 1, otherwise it is set to $\rho < 0$. When $\rho < -2n$, where n is the number of nodes in the graph, the stable states of the network correspond to maximal cliques in the graph. The ρ -annealing algorithm starts from a ρ close to 0 and gradually decreases ρ until it becomes less than $-2n$.

More precisely, let the annealing schedule be $0 > \rho_0 > \rho_1 > \dots > \rho_m$. The network may be set to any state initially. At the ρ setting ρ_0 , it is then driven to a stable state (which is typically not a clique). Let S_i denote the stable state reached by the network during the setting $\rho = \rho_i$. S_i then becomes the initial state for the setting $\rho = \rho_{i+1}$. The stable state S_m reached under the setting $\rho = \rho_m$ is a maximal clique.

Bertoni, Campadelli and Grossi proposed a somewhat similar algorithm that they called Iterative Hopfield Nets (IHN). The Hopfield network structure is the same as in the previous paragraph; the difference is in the weights. In IHN, the initial state is set to “all vertices in the given graph”. The initial energy function E_0 has two terms — one that penalizes pairs in S that are non-adjacent in the graph and one that rewards large $|S|$ when S is a clique. The network is driven to a energy minimizing state S at E_0 . If some pairs in S are non-adjacent in the graph, a new energy function E_1 is derived from E_0 in which the penalties of these specific pairs are increased. The network is again driven to a minimum, of this energy function, starting from the same S . And so on. The authors have implemented a small version of IHN (32 nodes) in FPGAs.

Our work in the present paper is related to that of IHN in that both employ Hopfield networks. The main difference is that we use multiple restarts (with and without adaptation). Over the years, we have formed a growing appreciation for

stochastic strategies, especially ones employing multiple restarts. We have found them to work better than our earlier strategies. Moreover, our approach provides an easy way to control the trade-off between solution quality and running time (by controlling the number of restarts). On the other hand, we should note that the solution qualities found by IHN experimentally seem to be quite good — as good or in some cases better than those of ours in the present paper, in comparable instances (see the section Experimental Results for a more detailed comparison). Still, by increasing the number of restarts we expect to be able to match or exceed their results.

Now let us briefly compare our work in this paper with IHN from the parallel implementation point of view. First, we note that the IHN hardware implementation is algorithm-specific, whereas ours of the present paper maps the algorithm onto a generic SIMD computer. Next we note that in its present form the FPGA implementation of IHN is very small, whereas we can handle problems of up to the size of the parallel machine. Finally, we observe that using multiple restarts lends itself nicely to parallelization. Multiple independent restarts can be run independently in parallel; multiple adaptive restarts require occasional (or frequent) global synchronization which make the parallelization somewhat interesting.

We now turn our attention to non-neural parallel approaches to the maximum clique problem. These include the work by Pardalos et al. in which they implement an exact search algorithm on a network of four Sun4 workstations using MPI in Fortran-77 [29]. No results on standard DIMACS graphs are reported. The work by Shinano et al. [34] proposes another exact solution to maxclique based on branch-and-bound and its parallel implementation on a network of IBM RS/6000 workstations. They offer some solution quality and run time results on few DIMACS graphs. Finally, it is worth mentioning the work by Alizadeh [1] in which a linear programming technique is parallelized to obtain sub-linear parallel run time on perfect graphs (for which computing the maximum clique exactly is polynomial [24]). No experimental results are reported.

The work presented in this paper combines and expands our previous work on SIMD implementations of asynchronous problems [9] and a specific work on SIMD implementations of Hopfield networks for Maximum Clique [10].

The Optimizing Neural Network

Given an undirected graph $G = \{V, E\}$, composed of a set of $N = |V(G)|$ vertices and a set of $|E(G)|$ edges, we build a corresponding discrete Hopfield network [21] as in Fig. 1. The Hopfield network is completely connected, and weights w_{ij} are assigned to the edges according to the following rule: $w_{ij} = 0$ if vertices i and j are connected by an edge, $w_{ij} = -1$ otherwise. A bias input to the nodes is initialized to $I_i = \frac{1}{2}, \forall i$.

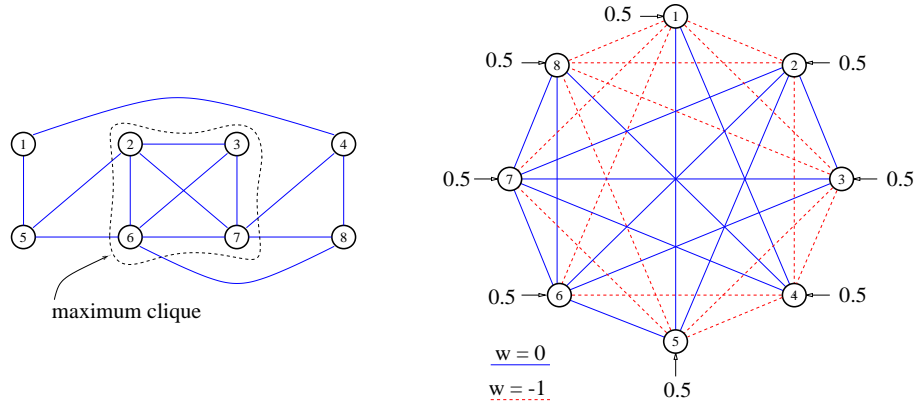


Figure 1: A sample graph (left) and the corresponding Hopfield neural network used to approximate its maximum clique (right).

A state $S_i \in \{0, 1\}$ associated with each node indicates that the node is included in the clique under construction when $S_i = 1$. The input to node i is

$$n_i = \sum_{j \neq i} w_{ij} S_j + I_i$$

Any serial update of the form:

$$S_i(t+1) = \begin{cases} 1 & \text{if } n_i(t) > 0 \\ 0 & \text{if } n_i(t) < 0 \\ S_i(t) & \text{otherwise} \end{cases}$$

minimizes the network energy:

$$E(\vec{S}) = -\frac{1}{2} \sum_{ij} w_{ij} S_i S_j - \sum_i I_i S_i$$

with lower energy minima corresponding to larger cliques. At each step, a node $i : \Delta E_i < 0$ is picked by a random selection, with probability

$$p_i = \frac{\Delta E_i}{\sum_{j: \Delta E_j < 0} \Delta E_j}$$

where

$$\Delta E_i(t) = -(S_i(t) - S_i(t-1))n_i$$

is the network energy decrease caused by the (possible) switch of node i .

```

maxclique()
{ read_graph() // read graph description into  $\vec{W} = w_{i,j}$ 
   $\forall i, I_i \leftarrow 0.5$  // initialize nodes' bias
   $\alpha_r \leftarrow 1, \alpha_p \leftarrow 1$  // initialize reward and penalize coefficients
   $C_{best} \leftarrow \vec{\Phi}$  // initialize best clique to empty
  for  $r = 1$  to RESTARTS do
  { find_a_clique() // call the energy descent function
    if ( $|C| > |C_{best}|$ ) //  $C$  is the clique just found
    {  $\forall i \in C, I_i \leftarrow I_i + \alpha_r * (|C| - |C_{best}|)$  // larger clique found: reward nodes in  $C$ 
       $C_{best} \leftarrow C$ 
    }
    else
       $\forall i \in C, I_i \leftarrow I_i - 1 - \alpha_p * (|C_{best}| - |C|)$  // penalize nodes in  $C$ 
    normalize_bias() // normal. all nodes' bias to ]0,1[
     $\alpha_r \leftarrow \alpha_r * 1.001, \alpha_p \leftarrow \alpha_p * 0.999$  // adjust adaptation coefficients
  }
  return( $C_{best}$ )
}

```

Figure 2: The maximum clique algorithm with adaptive restarts.

The complete algorithm (Fig. 2) uses multiple restarts to avoid local minima. Moreover, the restarts are *adaptive*, to focus on the most promising areas in the network. If a larger clique is found in a restart, then the bias input of each node in the clique is increased (the clique is *rewarded*), so that in the next restart the search will focus on its neighborhood. Otherwise, the bias inputs of the corresponding nodes are decreased (the clique is *penalized*), so that the search can gradually unfocus from a given clique [21, 22].

```

find_a_clique()
1: {  $\vec{S} \leftarrow \vec{\Phi}$  // initialize clique to empty
2:  $\vec{n} \leftarrow \vec{I}$  // initialize inputs to bias
  do
3: {  $\forall i, V_i \leftarrow \begin{cases} 1 & \text{if } n_i > 0 \text{ and } S_i = 0 \\ 0 & \text{otherwise} \end{cases}$  // node  $i$  can be selected
// node  $i$  can not be selected
4:  $\forall i, \Delta E_i \leftarrow -V_i * n_i$  // compute energy variations
5:  $k \leftarrow F(\vec{\Delta E})$  //  $F()$  picks next node
6: if ( $k = 0$ ) return // no more selectable nodes, clique found
7:  $S_k \leftarrow 1$  // add node  $k$  to clique under construction
8:  $\vec{n} \leftarrow \vec{n} + \vec{w}_k$  // and update all nodes' inputs
  } while(1)
}

```

Figure 3: The network evolution function that finds a clique.

The basic algorithm would apparently require $O(N^2)$ computation time for each restart loop. However, it can be easily worked out so that the total network energy does not need to be evaluated at each step. Moreover, the inputs to the nodes are updated upon a node status change, rather than re-computed every time. These two modifications bring the complexity of a cycle down to $O(N)$ (Fig. 3, lines 4 and 8) at the expense of adding a “switch vector” \vec{V} , whose values are also computed in $O(N)$ time (line 3). The function $F()$ (line 5) performs the random selection and returns the selected node or 0 if no nodes can be added to the clique. Finally, the algorithm in Fig. 3 has been stripped of the instructions that take into account the possible *removal* of a node from the clique under construction. This condition can only happen when the network (the \vec{S} vector) is initialized to some non-empty state where some nodes are not part of the clique. The selection mechanism always selects nodes that are in $N(C)$ (where C is the clique under construction), therefore, starting from an empty clique, no node removal is necessary.

Two Parallel Implementations

We evaluated two parallel implementations of the neural network algorithm: a fine-grained one that involves mainly parallelizing the vector computations, and a coarse-grained one in which each PE executes its own copy of the entire algorithm independently on its own copy of the entire graph.

The UCSC Kestrel parallel processor

The UCSC Kestrel parallel processor was originally developed at the University of California, Santa Cruz, with the intent of speeding up string comparison algorithms (such as the Smith and Waterman algorithm [37]) used for DNA and protein sequence analysis [18, 13, 33]. A database search engine based on the Kestrel system is currently available on the web [15].

Kestrel’s architectural model is a 512-PE SIMD linear array implemented as a single-board system [8, 14]. The system is composed of a PCI interface unit, an instruction memory, the array controller, and the PE array (Fig. 4, left).

Input and output of data to and from the PE array can take place only at the array ends, through the array controller, with input and output queues providing synchronization with the PCI bus.

Each PE (Figure 4, right) is composed of 3 basic blocks: an operating unit, a local memory and two banks of “systolic shared registers” [19]. The 8-bit operating unit groups an ALU, a multiplier, a bit-shifter (also used for conditional operation and PE masking), a comparator and some flag logic. The local memory is a 256-byte static RAM (operating at the same speed as the ALU) that can be addressed using local indirection. A unique inter-PE communication solution makes Kestrel different from other linear PE arrays. Both the left and the right register banks

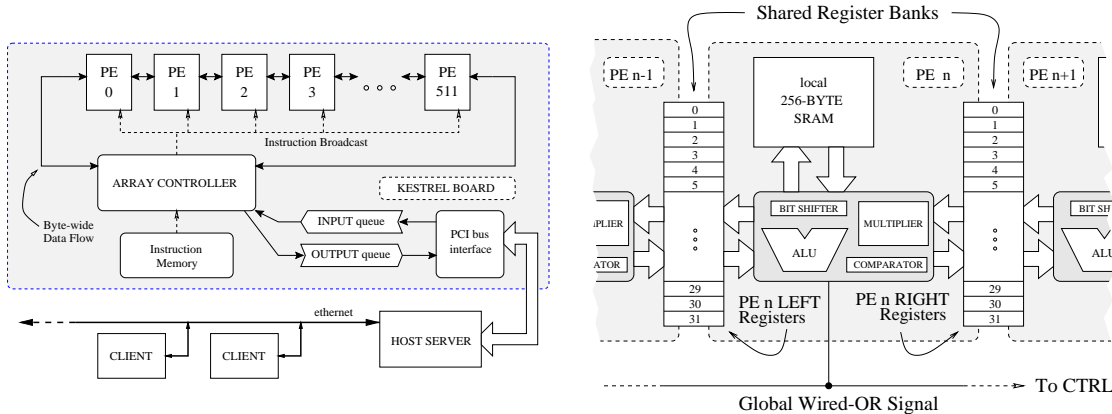


Figure 4: The Kestrel system high-level structure (left) and a detail of the PE array (right).

can be used by the PE as the source or the destination of any operation. In this way, 1-step inter-PE communication is performed simply by choosing the correct register banks as source and destination of an operation. A global *wired-or* signal (like the `globalor` in MasPar MPL) is detected by the controller and used to poll the PEs to perform global branching in the instruction sequence, including program termination.

Fine-grain Kestrel implementation

The mapping of the neural network algorithm onto the Kestrel parallel computer led to a “classic” fine-grain implementation, with one network node per PE. Therefore, the largest graph that can be solved with the current Kestrel system has 512 nodes. The key point in this implementation is that the steps at lines 1, 2, 3, 4 and 8 (Fig. 3) are performed in parallel on all N nodes (PEs), bringing down their time complexity to $O(1)$. The delicate point in the loop remains the selection function $F()$, whose complexity is still $\Theta(N)$.

The MasPar MP-2

The MasPar MP-2 is a well-known parallel processor and does not need to be described in detail here. We only mention the fact that it is a SIMD bi-dimensional array composed of 1k, 2k, 4k, 8k or 16k PEs (our programs were tested using a 4k-PE MP-2). The MasPar MP-2 architecture is essentially identical to the MasPar MP-1 architecture [28], except that the internal word length is 32 bits instead of four and the bandwidth to local memory is doubled.

Coarse-grain MasPar implementation

Figure 5 depicts the high-level mapping of our algorithm to the MasPar. Initially, the graph is read and a copy of it is stored into each PE's local memory. Then, each PE (the dashed boxes in Fig. 5) executes the algorithm on its own copy of the graph and performs the adaptation on its own copy of the weights. Different PEs follow a different stream of random numbers, used by the function F to select the nodes, and therefore explore a different region of the solution space. A synchronization barrier (the dash-dotted line) is used to compare the individual solutions and output the largest clique found.

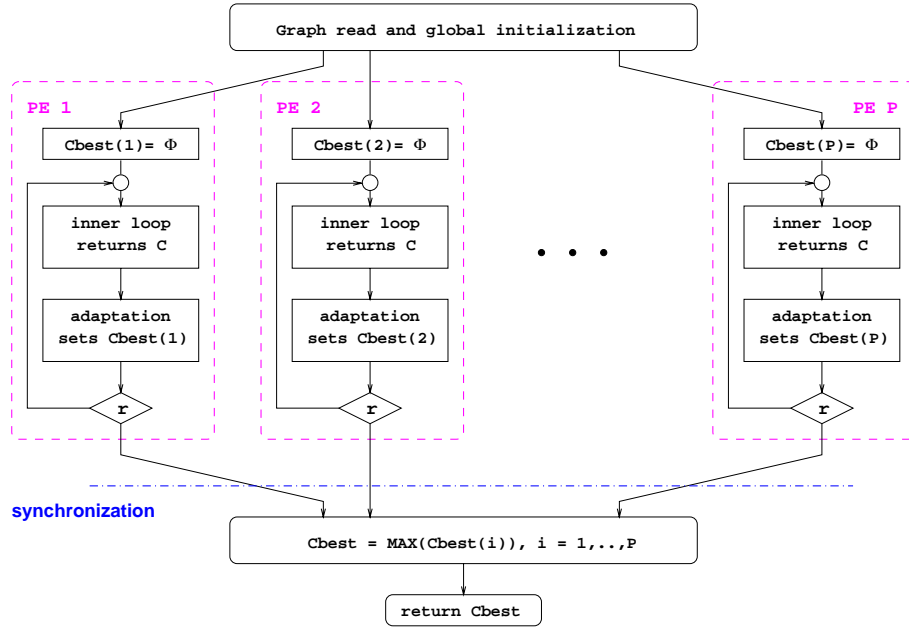


Figure 5: The MasPar "SPPM" implementation with individual adaptation.

Figure 6 shows a variant of this in which the adaptation is performed globally. In this case, there is a single set of weights. Each PE gets its own copy of these weights. A synchronization barrier is placed after the `find_a_clique()` function returns, and the largest clique found among all the PEs is broadcast to the whole array. Each PE then adapts its copy of the weights using this clique. This leads to a new setting of the weights that is identical with each PE. Again, in the subsequent operation, PEs will evolve differently based on individual sequences of random numbers used by the node selection function.

With respect to the mapping as in Fig. 5, this mapping achieves better results, on average, when only a few restarts are performed. This allows all the PEs to

immediately focus on a promising area in the graph.

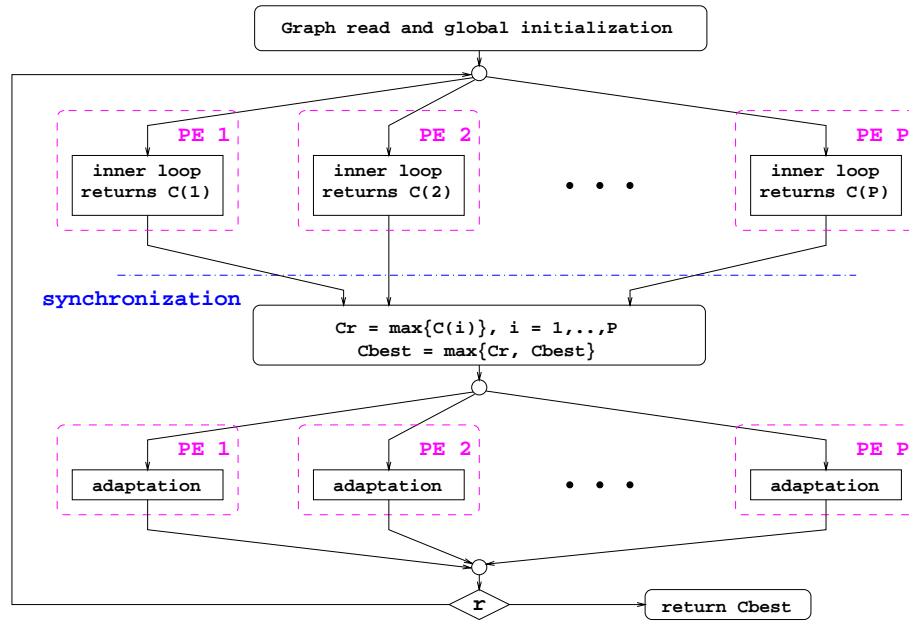


Figure 6: The MasPar “SPPM” implementation with global adaptation.

The “SIMD Phase Programming Model”

Our coarse-grained implementation was to make every PE execute independently its own copy of the entire algorithm on its own copy of the graph. A coarse-grained direct mapping of the algorithm onto a SIMD computer would be, in principle, straightforward.

In a direct mapping, however, all PEs will start the inner loop at the same time, but they will finish at different times, since they are likely to find cliques of different sizes. Therefore, all PEs will have to wait idle for the PE that finishes last. This is not very efficient. Therefore rather than using the direct mapping, here we use a more sophisticated mapping called “SIMD Phase Programming Model” (SPPM), that was specifically developed to handle this type of situation on SIMD platforms [9]. In this mapping, PEs that finish earlier are able to restart without having to wait for the others to finish, which yields a more efficient parallel implementation overall. Moreover, the SIMD Phase Programming Model also has the advantage of simplifying the decomposition of the sequential algorithm into efficiently parallelizable code.

In a SIMD parallel computer only a single instruction at a time is issued by the global controller, so it is impossible for different PEs to actually execute different

instructions at the same time. The “SIMD Phase Programming Model”, however, uses the ability of the PEs to turn themselves on and off to simulate individual execution flows.

The computational model for SIMD Phase Programming can be viewed as an enhanced “instruction-level approach” [35, 39]. A parallel computation will be split into a set of *phases*, “atomic” sequences of instructions serially executed by the PEs. A simple way of describing the parallelization process is to start from a flow chart description of the algorithm that we wish to parallelize (Fig. 7).

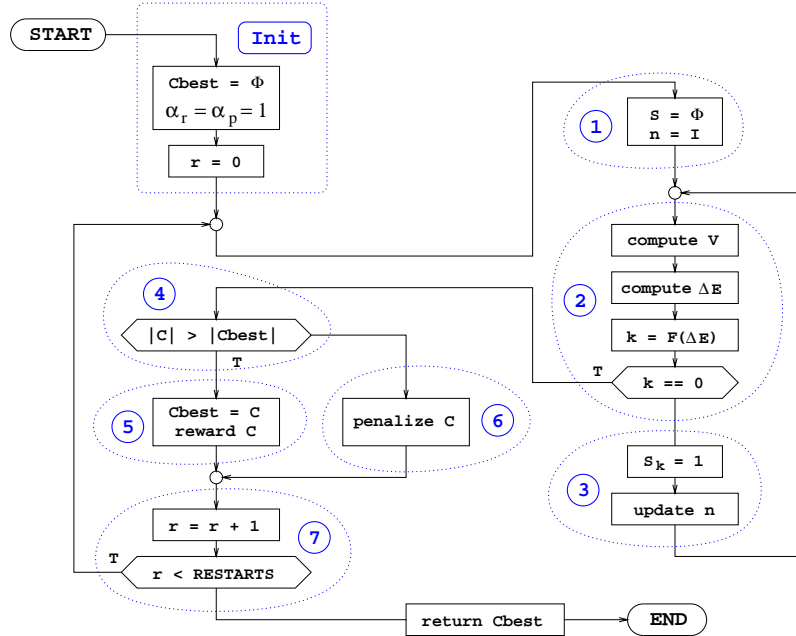


Figure 7: Flow chart of the complete algorithm, partitioned into phases.

The algorithm is then partitioned into “phases” (the dotted regions in Fig. 7), where a “phase” is defined as a block followed by a one- or multi-way branching condition (Fig. 9). The partitioning of the algorithm into phases is not unique; there are multiple choices. For example, phases 4, 5, 6, and 7 could be grouped into a single phase. However, mapping to separate phases the two blocks following the branching condition in phase 4 turns out to be more efficient in the parallel implementation. As a further example, phases 1, 2, and 3 could not be grouped into a single phase, since branching out of a phase is only allowed at the very end of a phase.

Figure 8 shows the algorithm’s flow chart at the phase level, with the branching conditions indicated on the edges.

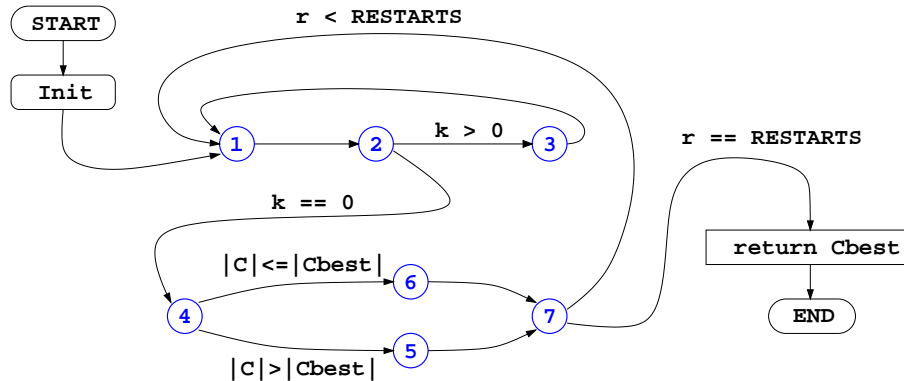


Figure 8: Phase-level flow chart.

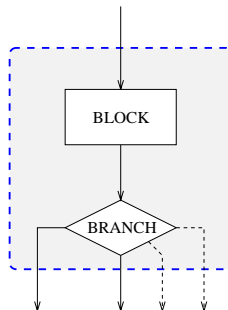


Figure 9: The structure of a phase.

In the parallel execution, the controller has a schedule over phases (Fig. 10). The phases in this schedule are broadcast to all PEs, in sequence. Each PE knows which phase to execute next, and executes it in its entirety or not at all. For example, after executing phase 4, the PEs for which $|C| \leq |C_{best}|$ will not execute the instructions in phase 5.

A group of phases that the controller broadcasts repeatedly is called a “cycle”. Figure 10 shows a single-cycle implementation of the entire algorithm. Depending on the algorithm, it might be more efficient to use multiple cycles. In our case, most of the computing time will be spent in phases 1, 2, and 3 — the inner loop — since PEs will be active in phases 4 to 7 only when a clique has been found. This suggests the two-cycle organization described in Fig. 11. The controller will keep looping through phases 1, 2, and 3, and only broadcast phases 4 to 7 when some PEs have found a clique. In this way we avoid broadcasting phases in which PEs are active only rarely. When all PEs have performed the given number of restarts, the program ends. The reader is referred to the previous paper for more details on

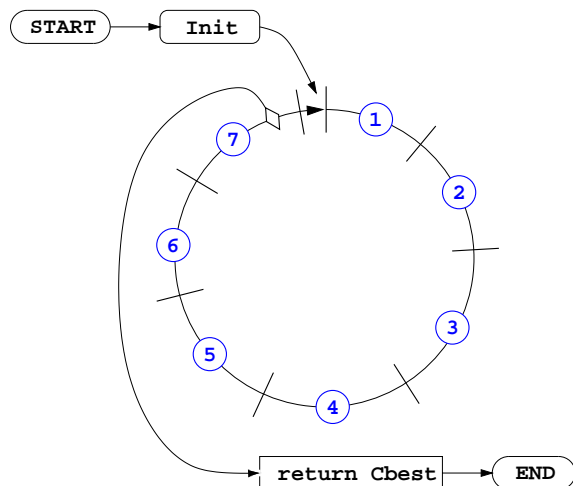


Figure 10: Phase-clock diagram of a single-cycle implementation.

this programming model [9].

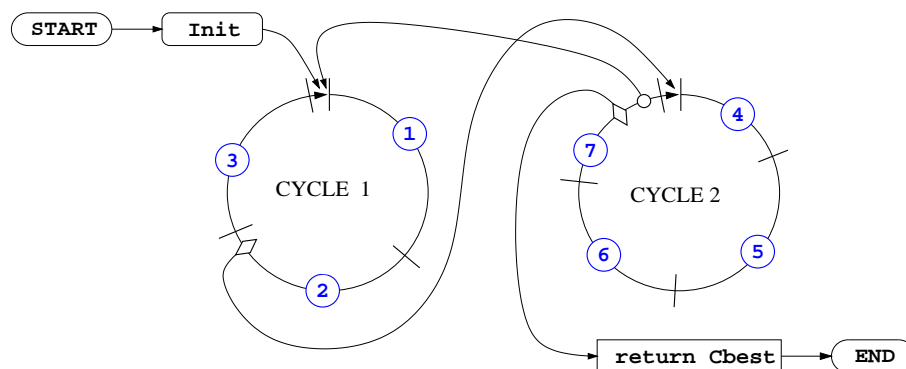


Figure 11: Phase-clock diagram of a more efficient two-cycle implementation.

Experimental Results

Table 1 shows results on some of the DIMACS benchmark graphs [16]. For comparison, timings from a serial machine (143 MHz SUN Ultra-SPARC 1 with 256 MB RAM) running the same algorithm are also reported. Even though the serial machine is not a “state-of-the-art” computer, these results still prove that high speedups can be achieved using relatively slow parallel processors (Kestrel runs at

20 MHz and the MasPar at 12.5 MHz). All experiments were run with 8K total restarts, which, on the MasPar, means two restarts using 4K PEs.

The Kestrel implementation achieves an average speedup of about 20 with respect to the serial program and the MasPar implementation achieves a speedup of about 40, both on the graphs in Tab. 1 and on all 46 DIMACS maximum clique benchmarks with $N \leq 512$. Both parallel implementations maintain the same average quality with respect to the serial program, with small differences accounting for the random mechanism of the algorithm.

| | | | | Kestrel | | MP-2/512 | | MP-2/4096 | | Serial | |
|---------------|-------|--------------|------------|----------------|-------|-----------------|----------|------------------|----------|---------------|-------|
| | | # processors | | 512 | | 512 | | 4096 | | 1 | |
| Graph | $ V $ | $ E $ | γ^* | γ_k | t_k | γ_{m1} | t_{m1} | γ_{m2} | t_{m2} | γ_s | t_s |
| MANN_a27 | 378 | 70551 | 126 | 125.2 | 29.5 | 124.9 | 64.6 | 125.2 | 15.9 | 124.8 | 582.5 |
| MANN_a9 | 45 | 918 | 16 | 16.0 | 8.1 | 16.0 | 1.7 | 16.0 | 0.8 | 16.0 | 12.4 |
| brock200_1 | 200 | 14834 | 21 | 18.2 | 4.9 | 19.6 | 6.9 | 19.8 | 2.8 | 19.2 | 42.2 |
| brock400_1 | 400 | 59723 | 27 | 21.6 | 7.4 | 22.3 | 17.4 | 23.0 | 8.8 | 21.8 | 94.4 |
| c-fat200-5 | 200 | 8473 | 58 | 58.0 | 16.4 | 58.0 | 15.7 | 58.0 | 3.2 | 58.0 | 157.3 |
| c-fat500-10 | 500 | 46627 | 126 | 126.0 | 41.7 | 126.0 | 80.3 | 126.0 | 15.3 | 126.0 | 838.2 |
| hamming6-2 | 64 | 1824 | 32 | 32.0 | 3.8 | 32.0 | 3.4 | 32.0 | 1.0 | 32.0 | 20.5 |
| hamming8-4 | 256 | 20864 | 16 | 16.0 | 4.2 | 16.0 | 7.6 | 16.0 | 3.4 | 16.0 | 33.5 |
| johnson16-2-4 | 120 | 5460 | 8 | 8.0 | 3.3 | 8.0 | 2.5 | 8.0 | 1.3 | 8.0 | 16.9 |
| johnson32-2-4 | 496 | 107880 | - | 16.0 | 8.2 | 16.0 | 22.0 | 16.0 | 13.7 | 16.0 | 109.4 |
| keller4 | 171 | 9435 | 11 | 11.0 | 2.6 | 11.0 | 4.2 | 11.0 | 2.0 | 11.0 | 22.2 |
| p_hat300-2 | 300 | 21928 | 25 | 21.3 | 3.4 | 23.9 | 11.4 | 24.6 | 4.4 | 22.7 | 58.2 |
| p_hat500-3 | 500 | 93800 | 50 | 40.4 | 7.7 | 44.2 | 34.7 | 45.6 | 14.3 | 40.9 | 182.4 |
| san200_0.7_1 | 200 | 13930 | 30 | 29.0 | 4.4 | 30.0 | 9.4 | 30.0 | 3.1 | 30.0 | 43.5 |
| san400_0.9_1 | 400 | 71820 | 100 | 80.2 | 8.1 | 100.0 | 53.4 | 99.4 | 13.4 | 69.7 | 230.2 |
| samr200_0.7 | 200 | 13868 | 18 | 16.4 | 4.5 | 16.9 | 6.4 | 16.8 | 2.8 | 16.8 | 36.6 |
| samr400_0.7 | 400 | 55869 | - | 18.9 | 6.8 | 19.1 | 15.8 | 19.7 | 8.3 | 19.3 | 80.8 |

Table 1: Average performance of parallel and serial programs, for 8K total restarts on different graphs. When known, the optimal clique γ^* is indicated.

The MasPar implementation permits a more elaborate experimentation. In our Kestrel fine-grain implementation there are always $N = |V|$ active PEs, but with the SPPM implementation a program can be run with any number of PEs. The plots in Fig. 12, left, show a typical behavior of the solution quality as a function of the number of PEs used, for different numbers of restarts on a sample graph. With only one restart, there is no adaptation, and solution quality is generally poor. With two restarts, i.e. with a little adaptation, the solution quality substantially increases. Execution time, however, is only weakly dependent on the number of PEs, but increases linearly with the number of restarts (Fig. 12, right). Therefore, when the number of PEs is much larger than the number of nodes in the graph, doing more

| Geometric average values on: | Kestrel | | MP-2/512 | | MP-2/4096 | |
|---------------------------------|---------------------|-----------|------------------------|--------------|------------------------|--------------|
| | γ_k/γ^* | t_s/t_k | γ_{m1}/γ^* | t_s/t_{m1} | γ_{m2}/γ^* | t_s/t_{m2} |
| graphs in Tab. 1 | 0.95 | 21.24 | 0.98 | 10.42 | 0.98 | 37.87 |
| all DIMACS maxclique benchmarks | 0.93 | 21.25 | 0.94 | 10.44 | 0.94 | 37.40 |

Table 2: Summarizing the performance: average solution quality (relative to the optimal solution) and execution times (relative to the serial machine) for the graphs in Tab. 1 and for all 46 official DIMACS maxclique benchmark graphs with $N \leq 512$.

than a few restarts is time-consuming without leading to a proportionally better solution. It is helpful to note that an exact branch-and-bound solver takes 856.65 seconds on 51 IBM RS/6000 workstations to find a maximum clique in p-hat500-3 [34].

We now briefly compare solution quality results with those on IHN by Bertoni, Campadelli and Grossi [6]. From their Table 1, we excerpt out the solution qualities obtained by IHN on those DIMACS graphs that are common between their Table 1 and our Table 2. These graphs and the IHN results, in parentheses, are MANN9 (16), MANN27 (125), p.hat300-2 (25), p.hat500-3 (49), c-fat200-5 (58), san200-0.7.1 (30), san400-0.9.1 (100), sanr200-0.7 (17), and sanr400-0.7 (21). Comparing these results with those of ours from Table 2, we note that IHN’s solution qualities on these graphs match or improve on ours. Having said that, we expect to be able to improve our results by increasing the number of restarts in our approach. The question will then become “how to compare our results with those of IHN (or other algorithms) taking both solution quality and running time into account”. This makes comparisons hard in principle (which is more important, solution quality or running time, depends on the application context). Even if we could resolve this satisfactorily there is the practical issue of comparing times across different implementations.

As with other asynchronous applications, the “SIMD Phase Programming Model” implementation exhibits a good parallel speedup

$$S(n) = \frac{t_1 \text{ PE}}{t_n \text{ PEs}}$$

and efficiency

$$\mu(n) = S(n)/n$$

Moreover, in this case, super-linear speedup is observed (Fig. 13). This phenomenon is common to depth-first parallel searches, when the algorithm returns just the first solution found and not all the possible solutions. This behavior is strongly dependent on the distribution of the solutions and on the random nature of the search [32].

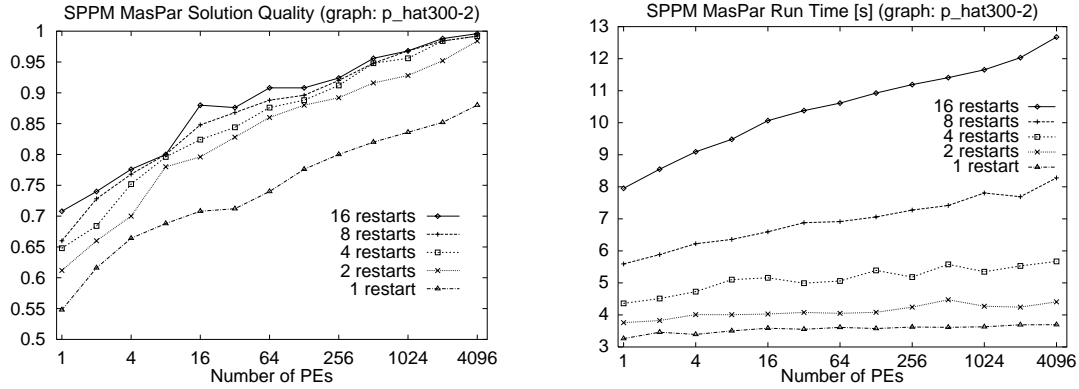


Figure 12: MasPar solution quality (as the ratio between the size of the average clique found and the optimal clique) and execution time as a function of the number of PEs, for different number of restarts on a sample graph.

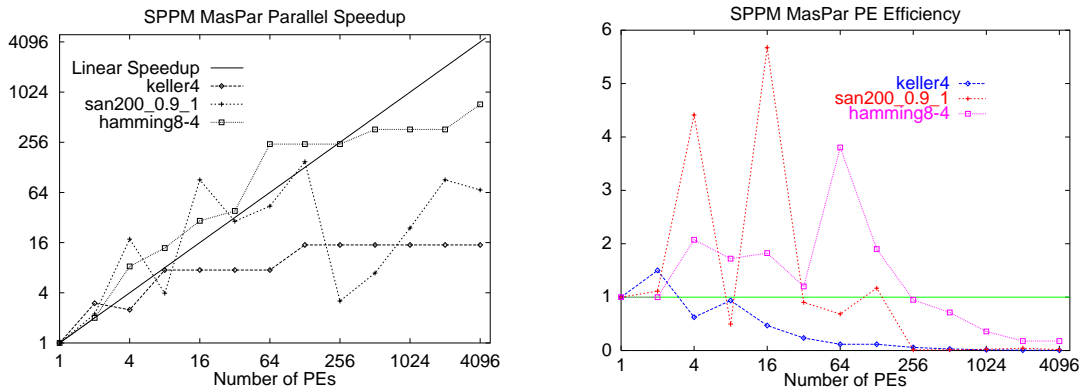


Figure 13: MasPar parallel speedup (left) and efficiency (right) for three sample graphs.

Conclusion

We have presented two different parallel implementations on two SIMD computers of a discrete Hopfield neural network approach to approximating the maximum clique in a graph. The Kestrel fine-grain implementation exploits the intrinsic parallelism of the algorithm, and therefore performs better, with respect to the serial program, when graphs are large and with large cliques. The MasPar SPPM implementation offers a more flexible approach because all available PEs can be active and contribute to the solution at the same time, regardless of the number of nodes

in the graph. The maximum clique problem has applications in many fields, and its parallel neural implementation can extend the scale of problems that can be efficiently solved. Moreover, the neural network algorithm itself is somewhat more generic (the problem-specific knowledge is encoded in weights and bias only), which can make it applicable to other combinatorial problems as well. Indeed, we have applied the same basic approach (in its serial form) to three variants of the graph coloring problem with good results [11].

Acknowledgments

This work was supported in part by the NSF grants CDA-9115268 and EIA-9722730.

References

- [1] F. Alizadeh, “A sublinear-time randomized parallel algorithm for the maximum clique problem in perfect graphs,” in *Second ACM-SIAM Symp. on Discrete Algorithms*, pp. 188–194, Society for Industrial and Applied Mathematics, 1998.
- [2] R. B. Altman *et al.*, eds., *Pacific Symposium on Biocomputing 2001*. London: World Scientific, 2000.
- [3] D. H. Ballard and M. Brown, *Computer Vision*. Prentice-Hall, 1982.
- [4] D. H. Ballard, P. C. Gardner, and M. A. Srinivas, “Graph problems and connectionist architecture,” tech. rep., Dep. Comp. Sci., Univ. of Rochester, 1987.
- [5] P. Berman and A. Pelc, “Distributed fault diagnosis for multiprocessor systems,” in *20th Annual Int. Symp. on Fault-Tolerant Computing*, pp. 340–346, 1990.
- [6] A. Bertoni, P. Campanelli, and G. Grossi, “A neural algorithm for the maximum clique problem: Analysis, experiments and circuit implementation,” *Algorithmica*, vol. 33, no. 1, pp. 71–88, 2002.
- [7] I. Bomze, M. Budinich, P. Pardalos, and M. Pelillo, *The maximum clique problem*, vol. 4 of *Handbook of Combinatorial Optimization*. Boston, MA: Kluwer Academic, 1999.
- [8] D. Dahle, J. D. Hirschberg, K. Karplus, H. Keller, E. Rice, D. Speck, D. H. Williams, and R. Hughey, “Kestrel: Design of an 8-bit SIMD parallel processor,” in *Seventeenth Conf. on Advanced Research in VLSI*, pp. 145–162, Sept. 1997.
- [9] A. Di Blas and R. Hughey, “Explicit SIMD programming for asynchronous applications,” in *IEEE Int. Conf. on Application-specific Systems, Architectures and Processors (ASAP2000)*, pp. 258–267, IEEE Comp. Soc. Press, July 2000.
- [10] A. Di Blas, A. Jagota, and R. Hughey, “Parallel implementations of optimizing neural networks,” in *Proc. of ANNIE 2000*, pp. 153–158, Nov. 2000.
- [11] A. Di Blas, A. Jagota, and R. Hughey, “Energy function-based approaches to graph coloring,” *IEEE Trans. on Neural Networks*, vol. 13, pp. 81–91, Jan. 2002.
- [12] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: Freeman, 1979.

- [13] L. Grate, M. Diekhans, D. Dahle, and R. Hughey, "Sequence analysis with the Kestrel SIMD parallel processor," in *Pacific Symposium on Biocomputing 2001* (R. B. Altman *et al.*, eds.), (London), pp. 263–274, World Scientific, 2000.
- [14] J. D. Hirschberg, D. M. Dahle, K. Karplus, D. Speck, and R. Hughey, "Kestrel: A programmable array for sequence analysis," *J. of VLSI Signal Processing*, vol. 19, pp. 115–126, 1998.
- [15] Home page of *The UCSC Kestrel Parallel Processor*
<http://www.cse.ucsc.edu/research/kestrel>.
- [16] Home page of Center for Discrete Mathematics and Theoretical Computer Science (DIMACS)
<http://dimacs.rutgers.edu>.
- [17] J. Hopfield and D. Tank, "Neural computation of decisions in optimization problems," *Biological Cybernetics*, vol. 52, pp. 141–152, 1985.
- [18] R. Hughey, "Parallel hardware for sequence comparison and alignment," *CABIOS*, vol. 12, no. 6, pp. 473–479, 1996.
- [19] R. Hughey and D. P. Lopresti, "B-SYS: A 470-processor programmable systolic array," in *Int. Conf. on Parallel Processing*, vol. 1, pp. 580–583, Aug. 1991.
- [20] A. Jagota, "Optimization by reduction to maximum clique," in *IEEE Int. Conf. on Neural Networks*, pp. 1526–1531, Mar. 1993.
- [21] A. Jagota, "Approximating maximum clique with a Hopfield network," *IEEE Trans. on Neural Networks*, vol. 6, pp. 724–735, May 1995.
- [22] A. Jagota, "An adaptive, multiple restarts neural network algorithm for graph coloring," *European J. of Operational Research*, vol. 93, pp. 257–270, 1996.
- [23] A. Jagota, L. Sanchis, and R. Ganesan, "Approximately solving maximum clique using neural network and related heuristics," in *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, vol. 26, pp. 169–204, American Mathematical Society, 1996.
- [24] D. E. Knuth, "The sandwich theorem," *Electronic J. of Combinatorics*, vol. 1, 1994. Article 1, approx. 48 pp. (electronic).
- [25] F. Lin and K. Lee, "A parallel computation network for the maximum clique problem," in *1st Int. Conf. on Fuzzy Theory Tech.*, Oct. 1992.
- [26] W. Lin, V. K. Prasanna, and K. W. Przytula, "Algorithmic mapping of neural network models onto parallel SIMD machines," *IEEE Trans. on Computers*, vol. 40, pp. 1390–1401, Dec. 1991.
- [27] M. Misra, "Parallel environments for implementing neural networks," *Neural Computing Surveys*, vol. 1, pp. 46–80, 1997.
- [28] J. R. Nickolls, "The design of the Maspar MP-1: A cost effective massively parallel computer," in *COMPCON Spring 1990*, pp. 25–28, IEEE Computer Society Press, Feb. 1990.
- [29] P. Pardalos, J. Rappe, and M. Resende, "An exact parallel algorithm for the maximum clique problem," in *High Performance Algorithms and Software in Nonlinear Optimization* (P. P. R. De Leone, A. Murl'i and G. Toraldo, eds.), Kluwer, 1998.
- [30] L. Prechelt, "Exploiting domain-specific properties: Compiling parallel dynamic neural network algorithms into efficient code," *IEEE Trans. on Parallel and Distributed Systems*, vol. 10, pp. 1105–1117, Nov. 1999.

- [31] J. Ramanujam and P. Sadayappan, "Optimization by neural networks," in *Int. Conf. on Neural Networks*, vol. 2, pp. 325–332, 1988.
- [32] V. N. Rao and V. Kumar, "On the efficiency of parallel backtracking," *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, pp. 427–437, Apr. 1993.
- [33] E. Rice and R. Hughey, "Molecular fingerprinting on the SIMD parallel processor Kestrel," in *Pacific Symposium on Biocomputing 2001* (R. B. Altman *et al.*, eds.), (London), pp. 323–334, World Scientific, 2000.
- [34] Y. Shinano, T. Fujie, Y. Ikebe, and R. Hirabayashi, "Solving the maximum clique problem using PUBB," in *Int. Parallel Processing Symp. and Symp. on Parallel and Distributed Systems (IPPS/SPDP)*, pp. 326–332, IEEE, 1998.
- [35] W. Shu and M. Wu, "Asynchronous problems on SIMD parallel computers," *IEEE Trans. on Parallel and Distributed Systems*, vol. 6, pp. 704–713, July 1995.
- [36] K. Smith, "Neural networks for combinatorial optimization: A review of more than a decade of research," *INFORMS Journal on Computing*, vol. 11, no. 1, pp. 15–34, 1999.
- [37] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *J. of Molecular Biology*, vol. 147, pp. 195–197, 1981.
- [38] M. Vingron and P. Pevzner, "Motif recognition and alignment for many sequences by comparison of dot matrices," *J. of Molecular Biology*, vol. 218, pp. 33–43, 1991.
- [39] M. Wu and W. Shu, "MIMD programs on SIMD architectures," in *The Sixth Symp. on the Frontiers of Massively Parallel Computation*, pp. 162–170, Oct. 1996.