Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

# A TAXONOMY OF PARALLEL SIMULATED ANNEALING TECHNIQUES

Daniel R. Greening

August 1989
CSD-890050

# Parallel Simulated Annealing Techniques

Daniel R. Greening

University of California, Los Angeles, and
IBM T.J. Watson Research Center

November 27, 1989

**Abstract**

Simulated annealing is a stochastic algorithm for solving discrete optimization problems, such as the traveling salesman problem and circuit placement.

To reduce execution time, researchers have parallelized simulated annealing. *Serial-like* algorithms identically maintain the properties of sequential algorithms. *Altered generation* algorithms modify state generation to reduce communication, but retain accurate cost calculations. *Asynchronous* algorithms reduce communication further by calculating cost with outdated information.

Experiments suggest that asynchronous simulated annealing can obtain greater speedups than other techniques. It exhibits the properties of cooperative phenomena: processors asynchronously exchange information to bring the system toward a global minimum.

This paper provides a comprehensive, taxonomic survey of parallel simulated annealing techniques, highlighting their performance and applicability.

# 1 Introduction

Several interesting combinatorial optimization problems are *NP*-hard—that is, they require at least non-deterministic polynomial time to obtain optimal solutions. Mathematicians have exerted considerable effort trying to determine whether $P$ (deterministic polynomial time) = $NP$, with no success. Thus, present-day optimal solutions for *NP*-hard problems require exponential time, rendering them intractable. That explains why sub-optimal, polynomial-time algorithms, like simulated annealing, have attracted interest.

Simulated annealing seeks to minimize a cost function for a system of interacting state variables [1]. Often the cost function presents a difficult landscape, with many local minima. Simulated annealing tries to escape local minima by randomly following cost-*increasing* paths. One cannot guarantee polynomial-time convergence to a global minimum for NP-complete problems (that would prove $P = NP$), however evidence shows that simulated annealing often produces good results.

Simulated annealing is often applied to VLSI circuit placement. In VLSI design, reducing circuit area decreases fabrication price, and shortening wires increases circuit speed. Rearranging circuit elements (called "cells") on a plane will change those values. Optimizing the arrangement is VLSI circuit placement: the cost function typically includes a linear combination of total circuit area and total wire-length.

Variants of VLSI placement fall into three categories. In "gate-array placement," all cells have a uniform rectangular shape. In "row-based placement," cells have a constant height, but varying width. In "macro-cell placement," cells can vary in size and shape.

A popular row-based placement and routing program, called TimberWolfSC, uses simulated annealing [2]. In a benchmark held at the 1988 International Workshop on Placement and Routing, TimberWolfSC produced the smallest placement for the 3000-element *Primary2* circuit—3% smaller than its nearest competitor. Moreover, it completed earlier than all other entrants. TimberWolfSC also routed *Primary2*; no other entrant completed that task.

Even approximate solutions to *NP*-hard problems require substantial time: simulated annealing is no exception. On a Sun 4/260, *Primary2* requires approximately 3 hours to place. Larger circuits require more time; for 30,000-element circuits, placement runs commonly take 36 hours.

Such onerous run times have driven researchers to implement simulated annealing on multiprocessors. Several techniques have been tried. Organized under the taxonomy described in this paper, their similarities should become clear.

Parallel implementations, particularly the asynchronous simulated annealing programs, exhibit properties of cooperative phenomena: individual processors make decisions based on incomplete or delayed information, yet together they approach a common goal [3]. As a result, this paper can suggest research directions and implementation techniques for areas outside simulated annealing.

## 1.1 Problem Formulation

Problems amenable to simulated annealing typically have these features:

1. One can construct an initial solution, or "state," $s_0 \in S$, where $S$ is the set of all feasible states, and evaluate its cost-function $E_{s_0} = f(s_0)$.

2. One can construct an inexpensive mapping, through a neighborhood relation $g$, from a single feasible state $s$ into a set of feasible states $g[\![\{s\}]\!]$.

3. One can inexpensively compute the cost-difference $\Delta E_{s,s'}$ for any state $s' \in g[\![\{s\}]\!]$, so that $E_{s'} = \Delta E_{s,s'} + E_s$.

4. A finite number, $i$, of recursive applications of $g$ to $s_0$, $g \circ \cdots \circ g[\![\{s_0\}]\!]$ covers the entire state space, so $g^i[\![\{s_0\}]\!] = S$.

```
1.    function accept(ΔE, T)
2.        return((ΔE ≤ 0) ∨ (e^(−ΔE/T_i) > random()));

3.    read(P);
4.    s ← some randomly constructed initial state for P;
5.    E ← f(s);
6.    T ← ∞;
7.    loop for i ← 0 to ∞
8.        ŝ ← a randomly selected element of g[[{s}]];
9.        ΔE ← Δf(s, ŝ)
10.       if accept( ΔE, T ) then
11.               s ← ŝ;
12.               E ← E + ΔE;
13.           end if;
14.       if done( T, other statistics ) then
15.               write( s );
16.               stop;
17.           end if;
18.       T ← update( T, other statistics );
19.       end loop;
```

Figure 1: Algorithm SSA, Sequential Simulated Annealing

Sequential implementations follow the general form in Figure 1, Algorithm SSA. It uses a pseudorandom number generator to create a random starting state (line 4), to generate a random state change for consideration (line 8), and to decide whether to accept the generated state (line 10).

The **accept** function (line 1) uses $\Delta E$ to decide whether to keep the new configuration. If $\Delta E$ is negative, the perturbed state $\hat{s}$ is better than $s$, and the program always accepts the new configuration (lines 11–12). If $\Delta E$ is positive, state $\hat{s}$ is worse than state $s$, and the program accepts the new configuration with probability $e^{-(\Delta E/T)}$.

Higher $T$ values and lower $\Delta E$ values increase the likelihood that a cost-increasing configuration will be accepted. However, if $T > 0$, any cost-increasing configuration has some probability of being accepted.

The procedure for updating $T$ is called the *temperature schedule.* The *equilibrium cost* is the mean value of $E$ we would obtain from running the simulated annealing algorithm forever at some fixed temperature $T$. Most program first set $T$ at a high value, then reduce $T$ while attempting to keep $E$ close to the equilibrium cost. One common temperature schedule has $T \leftarrow \gamma T$, where $0 < \gamma < 1$.

Intuitively, simulated annealing first explores the entire state space and then reduces its scope. Each lowering of the temperature restrains state exploration further. While the temperature is high, the algorithm can easily jump out of local minima; at its lowest temperatures, the algorithm usually moves toward lower cost.

## 1.2 Parallel Algorithms

Since a new state contains modifications to the previous state, simulated annealing is often considered an inherently sequential process. However, researchers have eliminated some sequential dependencies, and have developed several parallel annealing techniques. To categorize these algorithms, we ask several questions:

1. How is the state space divided among the processors?

2. Does the state generator for the parallel algorithm produce the same neighborhood as the sequential algorithm? How are states generated?

3. Can moves made by one processor cause cost-function calculation errors in another processor? Are there mechanisms to control these errors?

4. What is the speedup? How does the final cost vary with the number of processors? How fast is the algorithm, when compared to an optimized sequential program?

A parallel algorithm exhibits so-called "superlinear" speedup when the speed improvement over a sequential algorithm exceeds the number of processors. Simulated annealing researchers frequently see this suspicious property.

Three factors can explain most superlinear speedup observations. First, changes to state generation wrought by parallelism can improve annealing speed or quality [4]. If this happens, one can reconcile the sequential algorithm by mimicking the properties of the parallel version [5, 6]. Second, a speed increase might come with a solution quality decrease [7]. That property holds for sequential annealing, as well [8]. Third, annealing experimenters often

begin with an optimal initial state, assuming that high-temperature randomization will annihilate the advantage. But if the parallel implementation degrades state-space exploration, high-temperature may not totally randomize the state: the parallel program, then, more quickly yields a better solution [9].

Knowledge of such pitfalls can help avoid problems. Superlinear speedup in cooperating systems, such as parallel simulated annealing, should raise a red flag: altered state exploration, degraded results, or inappropriate initial conditions may accompany it.

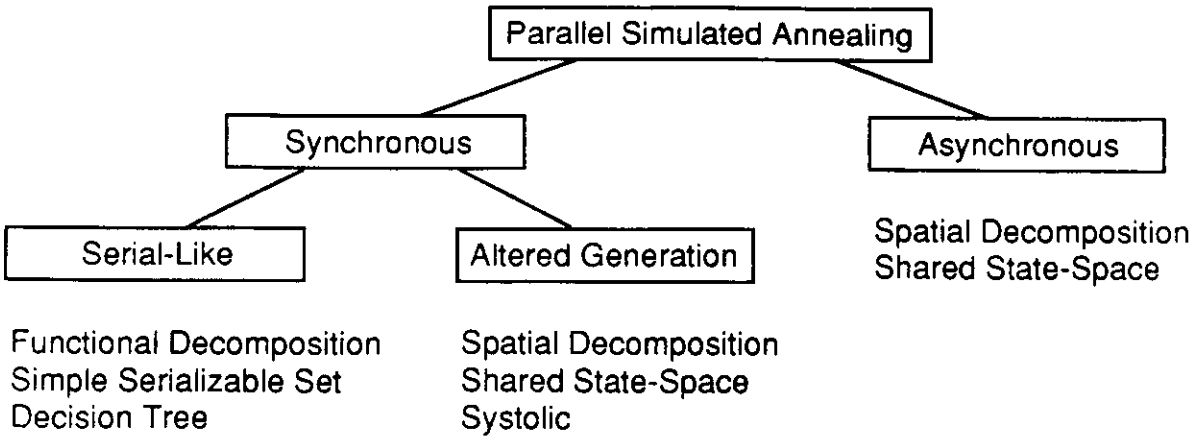



Figure 2: Parallel Simulated Annealing Taxonomy

The taxonomy presented here divides parallel annealing techniques into the three major classes shown in Figure 2: serial-like, altered generation, and asynchronous. We call an algorithm *synchronous* if adequate synchronization ensures that cost function calculations are accurate. Two major classes, *serial-like* and *altered generation*, are synchronous algorithms. *Serial-like convergence* algorithms identically maintain the convergence properties of sequential annealing. *Altered generation* algorithms modify state generation, but retain accurate cost calculations. *Asynchronous* algorithms, the third major class, eliminate some synchronization and tolerate the resulting errors to get a better speedup.

Each class makes some trade-off between cost-function accuracy, state generation, parallelism or communication overhead.

# 2 Serial-Like Algorithms

Three synchronous parallel algorithms preserve the convergence properties of sequential simulated annealing: *functional decomposition*, *simple serializable set*, and *decision tree decomposition*. We call these *serial-like* algorithms.

## 2.1 Functional Decomposition

*Functional decomposition* algorithms exploit parallelism in the cost-function $f$. In the virtual design topology problem, for example, the cost function must find the shortest paths in a graph. One program computes that expensive cost function in parallel, but leaves the sequential annealing loop intact [10]. Published reports provide no speedup information.

Another program evaluates the cost function for VLSI circuit placement in parallel [11]. Simultaneously, an additional processor selects the next state. Figure 3, Algorithm FD, shows the details.

```
1.   m' ← select random state;
2.   loop for i ← 0 to ∞
3.       m ← m';
4.       parallel block begin
5.           m' ← generate( m );
6.           E_0 ← block-length-penalty( m );
7.           E_{1,0} ← overlap for affected cell c_0 before move;
8.           ··· E_{1,j} ← overlap for affected cell c_j before move;
9.           E_{2,0} ← overlap for affected cell c_0 after move;
10.          ··· E_{2,j} ← overlap for affected cell c_j after move;
11.          E_{3,0} ← length change for affected wire w_0;
12.          ··· E_{3,k} ← length change for affected wire w_k;
13.          end parallel block;
14.      ΔE ← E_0 + (E_{1,0} + ... + E_{1,j}) − (E_{2,0} + ... + E_{2,j}) + (E_{3,0} + ... + E_{3,k});
15.      if accept( ΔE, T ) then
16.          parallel block begin
17.              update overlap values;
18.              update blocks and cells;
19.              update wire w_0;
20.              ··· update wire w_k;
21.              end parallel block;
22.          end if;
23.      recompute T, evaluate stop criteria, etc.
24.      end loop;
```

Figure 3: Algorithm FD, Functional Decomposition for VLSI Placement

One can obtain only a limited speedup from Algorithm FD. Ideally, the parallel section from line 4 to line 13 dominates the computation, each process executes in uniform time, and communication requires zero time. One can then extract a maximum speedup of $1 + 2j + k$, where $j$ is the average cells affected per move, and $k$ is the average wires affected per move. Researchers estimate a speedup limitation of 10, based on experience with the VLSI placement program TimberWolfSC [2].

Since cost-function calculations often contain only fine-grain parallelism, communication and synchronization overhead can dominate a functional decomposition algorithm. Load-balancing poses another difficulty. Both factors degrade the maximum speedup, making functional decomposition inappropriate for many applications.

## 2.2 Simple Serializable Set

If a collection of moves affect *independent* state variables, distinct processors can independently compute each $\Delta E$ without communicating. We call this a "serializable set"—the moves can be concluded in any order, and the result will be the same. The simplest is a collection of rejected moves: the order is irrelevant, the outcome is always the starting state.

The *simple serializable set* algorithm exploits that property [11]. At low annealing temperatures, the acceptance rate (the ratio of accepted states to tried moves) is often very low. If processors compete to generate one accepted state, most will generate rejected moves. These can all be executed in parallel.

```
1.    shared variable s, semaphore sema;
          ...
2.    parallel loop for i ← 1 to P;
3.        loop for j ← 0 to ∞
4.            wait( sema );
5.            s_old ← s;
6.            signal( sema );
7.            ⟨ŝ, ΔE⟩ ← generate( s_old );
8.            if accept( ΔE, T ) then
9.                wait( sema );
10.               if s_old = s then
11.                   s ← ŝ;
12.                   T ← new T;
13.                   end if;
14.               signal( sema );
15.               end if;
16.           change T, evaluate stop criterion, etc.
17.           end loop;
18.       end parallel loop;
```

Figure 4: Algorithm SSS. Simple Serializable Set Algorithm

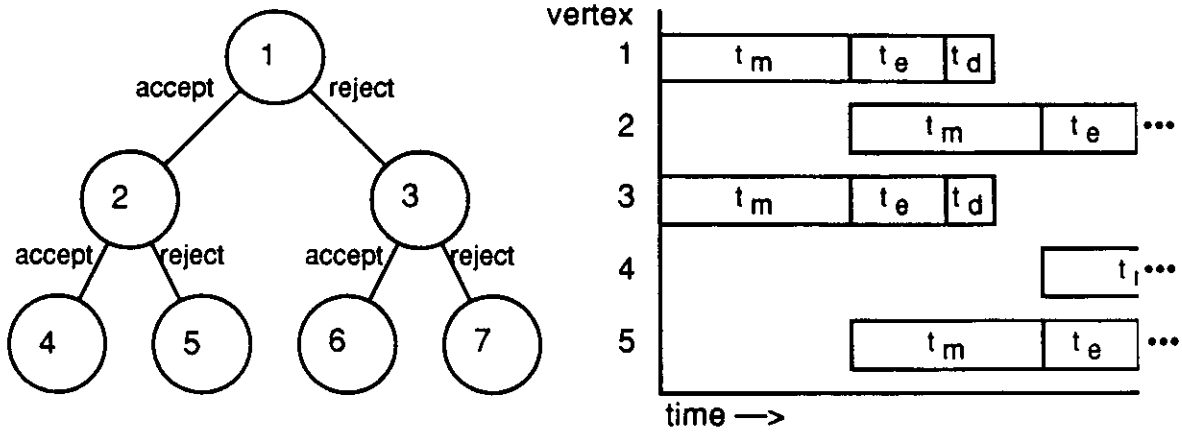
Figure 4, Algorithm SSS, shows such a technique [12]. $P$ processors grab the current state in line 5. Each processor generates a new state at line 7. If the new state is accepted (line 8) *and* the old state has not been altered by another processor (line 10), the move is made. Otherwise the move is discarded.

If the acceptance rate at temperature $T$ is $\alpha(T)$, then the maximum speedup of this algorithm, ignoring communication and synchronization costs, is $1/\alpha(T)$. At high temperatures, where the acceptance rate is close to 1, the algorithm provides little or no benefit. But since traditional annealing schedules spend a majority of time at low temperatures, Algorithm SSS can improve overall performance.

Algorithm SSS has limitations. Some recent annealing schedules maintain $\alpha(T)$ at relatively high values, throughout the temperature range, by adjusting the generation function. Lam's schedule, for instance, keeps $\alpha(T)$ close to 0.44 [8]. With that schedule, Algorithm SSS provides a maximum speedup of approximately 2.3, regardless of the number of processors.

## 2.3 Decision Tree Decomposition

A third serial-like algorithm, called decision tree decomposition, exploits parallelism in making accept-reject decisions [13]. Consider the tree shown in Figure 5a. If we assign a processor to each vertex, cost evaluation for each suggested move can proceed simultaneously. Since a sequence of moves might be interdependent (i.e., not serializable), however, we *generate* the states in sequence.




a. Annealing Decision Tree b. Functional Dependence

Figure 5: Decision Tree Decomposition

Figure 5b shows vertex dependencies. A vertex generates a move in time $t_m$, evaluates the cost in time $t_e$, and decides whether to accept in time $t_d$. Note that vertex 2 cannot begin generating a move until vertex 1 generates its move and sends it to vertex 2.

Research has provided hypothetical execution times, but no experimental confirmation. A simple implementation results in predicted speedups of $\log_2 P$, where $P$ is the number of

processors. By skewing tree evaluation toward the left when $\alpha(T) \geq 0.5$, and toward the right when $\alpha(T) < 0.5$, researchers predict a maximum speedup of $(P + \log_2 P)/2$ [13].

In numeric simulations, however, the speedups fall flat. With 30 processors and $t_m = 16t_e$, the estimated speedup was 4.7. Unfortunately, in VLSI placement problems $t_m \gg t_e$, and in traveling salesman problems $t_m \approx t_e$. Reconciling $t_m$ leads to a speedup of less than 2.5 on 30 processors. As a result, this approach holds little promise for such applications.

# 3 Altered Generation Algorithms

Even if a parallel annealing algorithm computes cost-functions exactly, it may not mimic the statistical properties of a sequential implementation. Often, state generation must be modified to reduce inter-processor communication. These *altered generation* methods change the pattern of state space exploration, and thus change the expected solution quality and execution time.

## 3.1 Spatial Decomposition

In spatial decomposition techniques, we distribute state variables among the processors, and variable updates are transmitted between processors as new states are accepted. Spatial decomposition techniques are typically implemented on message-passing multiprocessors.

In *synchronous* decomposition, either processors must carefully coordinate move generation, or processors must not generate moves that affect other processors' state variables. We call the resulting two techniques *cooperating processors* and *independent processors.*

### 3.1.1 Cooperating Processors

A cooperating processor algorithm disjointly partitions state variables over the processors. A processor that generates a new state notifies other affected processors. Then, those processors synchronously evaluate and update the state. If a proposed move could interfere with another in-progress move, the proposed move is either delayed or abandoned.

One such program minimizes the number of routing channels (the slots where wires lie) for a VLSI circuit [9]. The cost is the total number of routing channels that contain at least one wire; two or more wires can share the same routing channel, if they don't overlap.

The program first partitions a set of routing channels across the processors of an iPSC/2 Hypercube; that processor assignment henceforth remains fixed. Processors proceed in a lockstep communication pattern. At each step, all processors are divided into master-slave pairs. The master processor randomly decides among four move classes:

*Intra-displace* The master and slave each move a wire to another channel in the same processor.

*Inter-displace* The master processor moves one of its wires to a channel in the slave processor.

*Intra-exchange* Each master and slave each swap two wires in the same processor.

*Inter-exchange* The master swaps a wire from one of its channels with a wire in the slave.

Experiments indicate superlinear speedups, from 2.7 on 2 processors to 17.7 on 16 processors. These apparently stem from a nearly-optimal initial state and more-constrained parallel moves, making the reported speedups untenable. However, the decomposition method itself is sound.

### 3.1.2 Independent Processors

In independent processor techniques, each processor generates state changes which affect only its own variables. Under this system, a fixed variable assignment would drastically limit state-space exploration, and produce an inferior result; it requires periodic state variable redistribution.

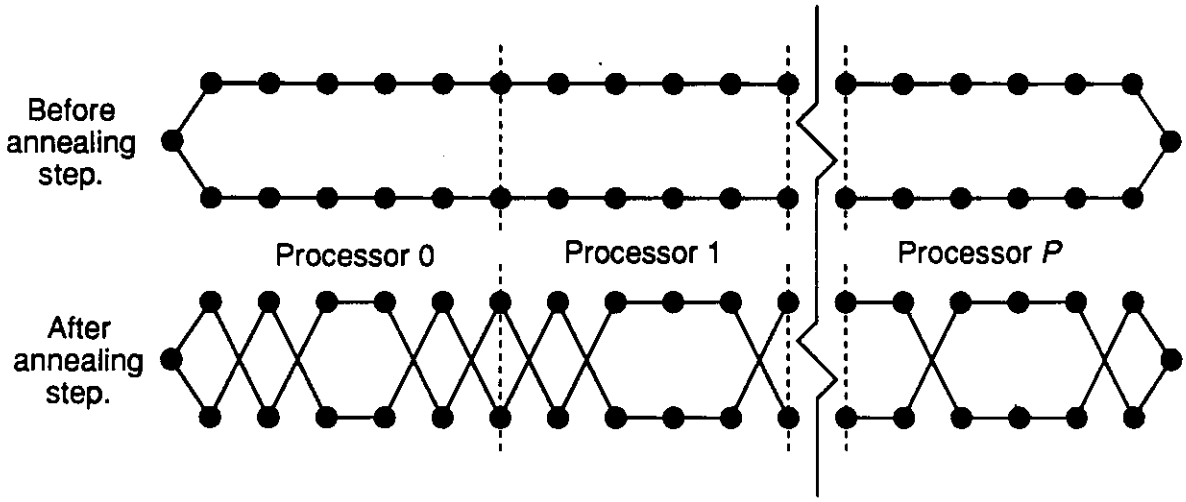



Figure 6: Rubber Band TSP Algorithm

One such technique optimizes traveling salesman problems [14]. A traveling salesman problem (TSP) consists of a collection of cities and their planar coordinates. A tour that visits each city and returns to the starting point forms a solution; the solution cost is its total length.

We construct an initial, poor-quality solution by putting the cities into a random sequence: the tour visits each in order and returns to the first city. We stretch this string of cities out like a rubber band, and evenly divide the two parallel tracks among the processors, as shown in Figure 6. The state variables consist of the endpoints of each two-city segment.

Each processor anneals the two paths in its section by swapping corresponding endpoints. After a fixed number of tries in each processor, the total path length is computed, and a

new temperature and a shift count are chosen. Each processor then shifts the path attached to its top left node to the left, and the path attached to its bottom right node to the right, by the shift count. This operation redistributes the state variables, ensuring that the whole state space is explored. Annealing continues until it satisfies the stopping criterion.

In one experiment, the 30 processor versus 2 processor speedup ranged from about 8 for a 243 city TSP, to 9.5 for a 1203 city TSP. Unfortunately, a single processor example was not discussed. The paper does not show final costs; final cost probably increases as the number of processors increases. Other spatial decomposition techniques exhibit similar behavior and speedups [15, 16].

## 3.2 Shared State-Space

Shared state-space algorithms make simultaneous, independent moves on a shared-memory state-space: no cost-function errors can occur.

One such algorithm optimizes VLSI gate-array placement [7]. Changes in the state generation function, resulting from the locking of both cells and wires, caused the algorithm to generate poor convergence. Maximum speedup was 7.1 for 16 simulated RP3 processors, solving a uniform $9 \times 9$ grid problem. Improving the parallel algorithm's convergence would reduce its speedup below 7.1.

A similar algorithm for minimizing the equal partition cut-set (see section 4.2) obtained a dismal speedup close to 1 on 16 processors [17].

Another shared state-space algorithm constructs conflict-free course timetables [18]. Before evaluating a move, the algorithm must lock the instructors, courses and rooms for two time periods, then swap them. If the locks conflict with an in-progress move, the locks are abandoned and another move is generated. Speedup was compared against an optimized sequential algorithm. With 8 processors, a speedup of 3.2 was obtained in scheduling 100 class periods, while 6.8 was obtained in scheduling 2252 class periods.

## 3.3 Systolic

The systolic algorithm relies on the property that simulated annealing brings a thermodynamic system toward the Boltzmann distribution [19, 20].

Suppose we have $P$ processors, and we maintain the same temperature for a chain of $N$ generated states. We would like to divide these moves into $P$ subchains of length $L = P/N$, and execute them on different processors. Figure 7 shows a corresponding data flow graph for this decomposition.

At any PICK node on processor $p$, we must decide between state $s_{(n-1,p)}$ computed by processor $p$ at temperature $T_{n-1}$, and state $s_{(n,p-1)}$ computed by processor $p-1$ at temperature $T_n$. We make the choice according to the Boltzmann distribution. The relative probability of picking $s_{(n-1,p)}$ is

$$\rho_0 = \frac{1}{Z(T_{n-1})} e^{[f(s1)-f(s_{(n-1,p)})]/T_{n-1}} \tag{1}$$

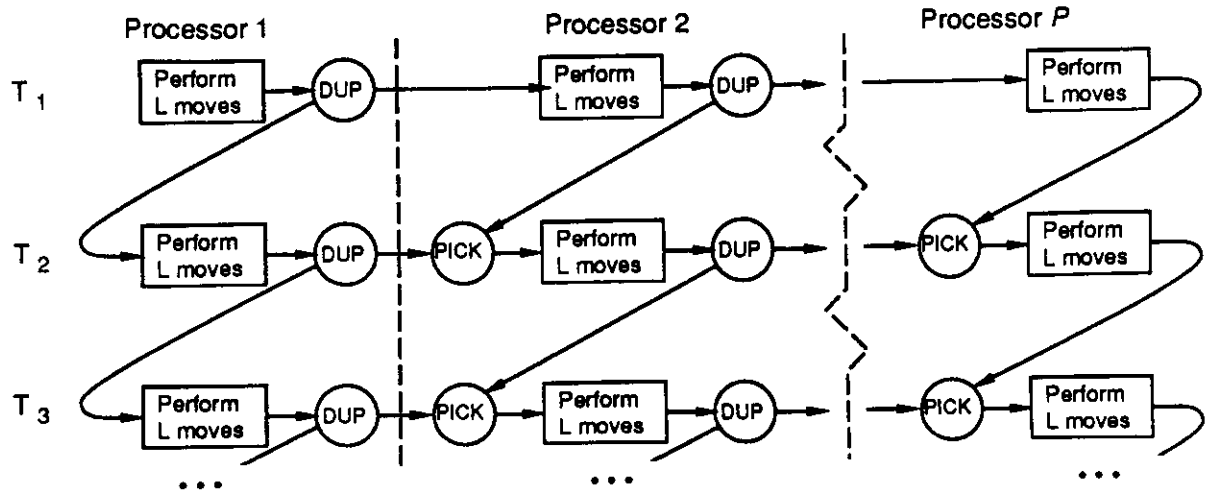



Figure 7: Systolic Algorithm

and the relative probability of picking $s_{(n,p-1)}$ is

$$\rho_1 = \frac{1}{Z(T_n)} e^{[f(s\downarrow)-f(s_{(n,p-1)})]/T_n} \tag{2}$$

where $S$ is the entire state space and $s\downarrow$ is a minimum cost state. $Z(T)$ is the *partition function* over the state space, namely

$$Z(T) = \sum_{s \in S} e^{f(s)/T} \tag{3}$$

The PICK node then selects $s_{(n-1,p)}$ and $s_{(n,p-1)}$ with probabilities

$$p(s_{(n-1,p)}) = \frac{\rho_0}{\rho_0 + \rho_1}, \qquad p(s_{(n,p-1)}) = \frac{\rho_1}{\rho_0 + \rho_1} \tag{4}$$

If we don't know the minimum cost, we can't evaluate $f(s\downarrow)$. A lower bound must suffice as an approximation. Choosing a lower bound far from the infimum will increase execution time or decrease solution quality [8].

The partition function, $Z$, requires the evaluation of every state configuration. The number of state configurations is typically exponential in the number of state variables, making exact computation of $Z$ unreasonable.

As a result, the systolic method uses an approximate $Z$. In the temperature regime where the exponential function dominates, $\rho_0$ and $\rho_1$ are almost completely determined by their

numerators in Equations 1 and 2. The influence of $Z(T)$ thus becomes small, and it can be approximated by the normal distribution.

How does the algorithm perform? With 8 processors operating on a $15 \times 15$ uniform grid of cities, the systolic algorithm obtained a mean path-length of 230, at a speedup of about 6.2, while the sequential algorithm obtained an average of about 228.5. Accounting for the less optimal parallel result, the effective speedup is something less than 6.2.

# 4 Asynchronous Algorithms

Without sufficient synchronization, different processors can simultaneously read and alter dependent state-variables, causing cost-function calculation errors. Such algorithms are *asynchronous*. Imprecise cost-function evaluation accelerates *sequential* simulated annealing under certain conditions [21, 22]; a similar effect accompanies asynchronous parallel simulated annealing.

These algorithms use a method related to chaotic relaxation—processors operate on outdated information [23]. Since simulated annealing randomly selects hill-climbing moves, it can tolerate some error; under the right conditions, annealing algorithms can evaluate the cost using old state information, but still converge to a reasonable solution. This property holds for genetic algorithms, as well [24].

Error tolerance provides a great advantage in multiprocessing: when processors independently operate on different parts of the problem, they need not synchronously update other processors. A processor can save several changes, then send a single block to the other processors. The processor sends less control information and compresses multiple changes to a state variable into one, reducing total communication traffic. In addition, if updates can occur out-of-order, synchronization operations are reduced. Asynchronous algorithms require a minimum synchronization: two processors acting independently must not cause the state to become inconsistent with the original problem.

Figure 8 shows how errors arise in a spatially decomposed traveling salesman problem. In the figure, variables $a_0$ and $a_1$ denote the endpoints of edge $a$. The simulated annealing algorithm swaps endpoints to generate a new state. The algorithm partitions the cities over two processors. A processor may only swap endpoints that point to its vertices, ensuring problem consistency. However, to reduce synchronization time, processors do not lock edges while they evaluate the cost-function.

While processor 0 considers swapping endpoint $a_0$ with $b_1$, processor 1 considers swapping endpoint $a_1$ with $b_0$. Processor 0 sees a path-length change for its move of $\Delta E = 2\,(1-\sqrt{2}) \approx -0.818$. Processor 1 also sees $\Delta E \approx -0.818$, for its move.

Processor 0 makes its move, by swapping $a_0$ and $b_1$. Now, processor 1 makes its move, thinking its $\Delta E \approx -0.818$ (a good move) when the effect is $\Delta E \approx +0.818$ (a bad move). At low temperatures, the error will degrade the final result unless corrected by a later move. So, simulated annealing does not have an unlimited tolerance for errors.

Cost-function errors usually degrade convergence quality, when all other factors are fixed:

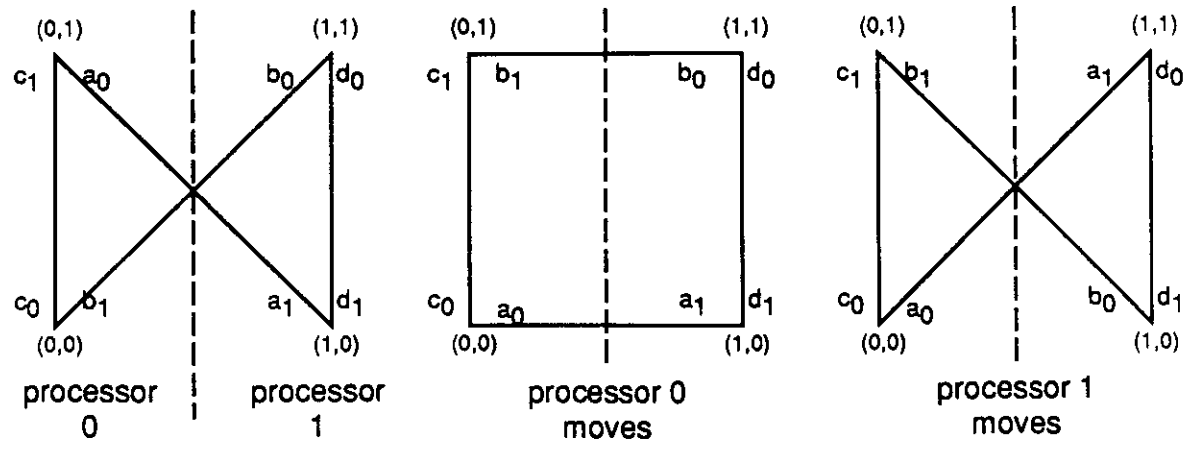



Figure 8: Cost-Function Errors in Spatial Decomposition

note the contrast with altered state generation. For example, experiments have shown that VLSI placement quality decreases as errors increase [4, 25].

Several authors have conjectured that annealing properties might be preserved when the errors are small. Experimental evidence bears this out [4, 7, 26, 27, 28]. However, we can easily construct a problem which converges well under sequential simulated annealing, but will *likely* converge to a bad local minimum in an asynchronous program.

Consider a system with two state variables $x$ and $y$, so some state $s = \langle x, y \rangle \in S$. Let the cost-function be $f(x + y)$, shown in Figure 9. Now put $x$ and $y$ on two separate processors. Each processor proposes a move: processor 0 generates $x \leftarrow x - 1$, while processor 1 generates $y \leftarrow y - 1$. In both cases, $\Delta E < 0$, so each move will be accepted.

The cost-function error causes the state to jump to a high local minimum. At low temperatures, the annealing algorithm probably will not escape this trap.

## 4.1 Asynchronous Spatial Decomposition

Asynchronous spatial decomposition methods, like the synchronous methods in section 3.1, partition state variables across different processors. However, in asynchronous algorithms each processor also maintains read-only copies of state variables from other partitions.

When a processor evaluates a new state, it uses only local copies of state variables. In some programs, when a move is accepted the new state information is immediately sent to other processors [26]. In other programs, a processor completes a fixed number of tries, called a "stream," before transmitting the modifications [4, 25]. Longer streams increase the execution-to-communication ratio, gaining a speedup, but they also increase calculation errors, reducing the solution quality.

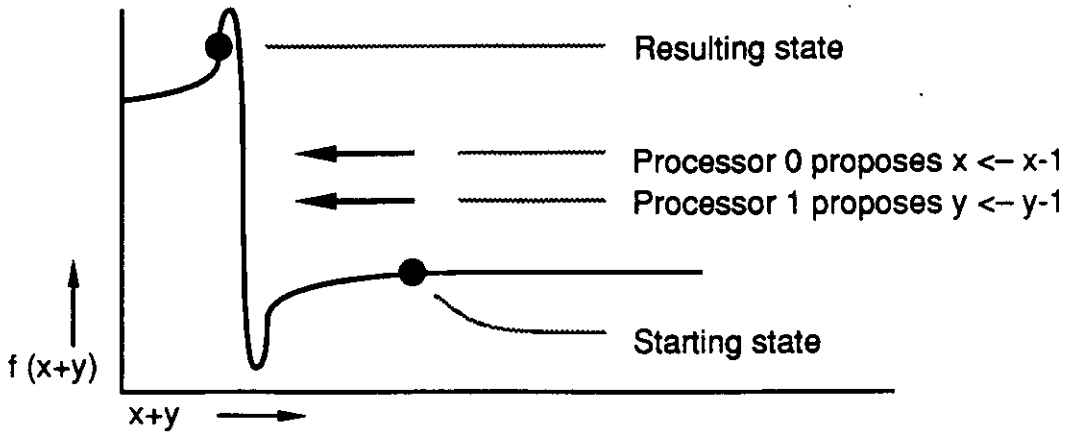



Figure 9: Errors Can Cause Annealing Failure

### 4.1.1 Clustered Decomposition

The clustered decomposition technique solves two concurrent optimization problems: the specified target problem and assigning the state variables to processors.

In one example, the target problem is VLSI macro-cell placement, and the assignment problem is cell partitioning [29]. Overlap penalties in the VLSI cost-function generate the largest errors—when two cells owned by different processors are moved to the same empty location, neither processor will see an overlap, but the overlap error might be huge. This leads to a clustering problem: divide state variables (macro-cells) equally among the processors, while putting dependent variables (adjacent or connected macro-cells) on the same processor.

We compute the assignment cost-function, for VLSI macro-cell placement, as follows. Let $C$ be the set of cells, let $\mathrm{C} = \{C_1, \ldots, C_P\}$ be the partition of $C$ over $P$ processors, let $\bar{c}$ be a cell's vector center and let $|c|$ be its scaler area. For each processor $p$, we can compute the center of gravity $X_p$ of its partition $C_p$

$$X_p = \frac{1}{\sum_{c \in C_p} |c|} \sum_{c \in C_p} \bar{c} \cdot |c| \tag{5}$$

and its inertial moment

$$\Gamma_p = \sum_{c \in C_p} \|\bar{c} - X_p\|^2 \cdot |c| \tag{6}$$

The assignment cost-function for partition C is

$$f_c(\mathrm{C}) = w_c \cdot \sum_{i=1}^{P} \Gamma_i \tag{7}$$

where $w_c$ is a weighting factor.

Experiments used the same temperature for both partitioning and placement: independent temperature schedules would probably improve the result. A 30 macro-cell problem, running on an 8 processor, shared-memory Sequent 8000, reached a speedup of 6.4 against the same algorithm running on a single processor.

Clustering improved convergence. We express a result's *excess cost* as $E_{\text{final}} - E_{\text{min}}$, where $E_{\text{final}}$ is the result's cost, and $E_{\text{min}}$ is the best solution known (presumably close to optimal). Clustering reduced the excess cost in a 101 cell, 265 wire problem by about 15%.

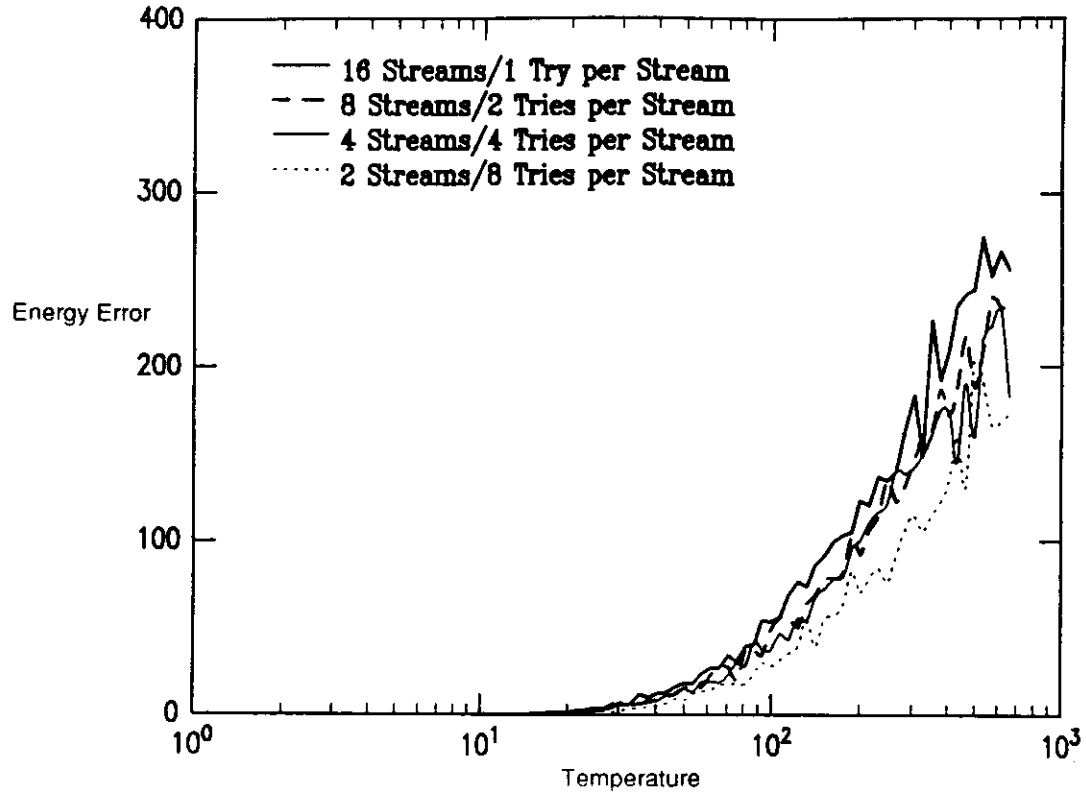



Figure 10: Spatial Decomposition, 16 Tries per Block

### 4.1.2 Rectangular Decomposition

A simpler approach, rectangular decomposition, tries to accomplish the same goals. It divides the grid of a VLSI placement problem into disjoint rectangles, then shifts the boundaries after each stream [4]. At low temperatures, interdependent state variables typically share a rectangle.

Different variants were tried: placing restrictions on the minimum width of a rectangle and "fuzzing" the rectangle boundaries. All rectangular decomposition schemes produced

small errors and converged close to the minimum. In contrast, random landing point assignment on identical problems produced greater errors and converged to a much higher final cost [25].

One rectangular decomposition experiment fixed the number of generated states in a block of $PN$ tries, where $N$ is the stream length and $P$ is the number of processors. Figure 10 displays the resulting errors. The error value is $\varepsilon = |\Delta E - \Delta E_P|$, where $\Delta E$ is the actual cost change after completion of a stream, and $\Delta E_P$ is the sum of the apparent cost changes observed by the processors. Increasing $P$ also increases $\varepsilon$, as one might expect.

## 4.2 Asynchronous Shared State-Space

Asynchronous shared state-space algorithms keep all state variables in shared memory. Processors competitively lock state variables, make moves, and unlock. Unlike synchronous algorithms, processors need not lock all affected state variables; they need only lock those variables required for problem consistency.

One experiment compared synchronous and asynchronous algorithms for VLSI gate-array placement [7]. Under a simulated RP3 environment, three methods were tried. Method A, a synchronous shared state-space algorithm, is described in section 3.2. Each processor locked two circuits and any attached wires before attempting a swap.

In method B1, an asynchronous algorithm, processors lock only the two cells in the proposed move, and calculate the new wire length with possibly-changing information. Each processor maintains a local copy of the histogram, which holds a collection of intermediate cost function variables [1]. A move updates only the local histogram; at the completion of a stream, each processor corrects its histogram with global state information.

Method B2 operates like B1, except that it never corrects the local histograms. Thus, histogram information becomes progressively outdated as the temperature falls.

Method B1 converged well with a maximum of 8 processors. Method B2 converged imperfectly, but surprisingly enough it converged better than a random spatial decomposition technique [25].

Using extrapolated simulation measurements for a 900 cell placement problem running on 90 processors, researchers estimated a speedup of about 45 for Method A, and 72 for Method B1 and B2 [30].

Another experiment compared synchronous and asynchronous shared state-space algorithms for the equal partition cut-set problem [17]. Given a graph with an even number of vertices, such algorithms partition the vertices into two equal sets, and minimize the number of edges which cross the partition boundary. The synchronous algorithm locked both vertices and edges, while the asynchronous algorithm locked only vertices.

On a 250 vertex graph, the synchronous algorithm ran more slowly than a sequential implementation, except at 16 processors where the speedup was close to 1. The asynchronous algorithm ran faster than the sequential algorithm, yielding 16-processor speedups from 5 on a graph with mean vertex degree 10, to 11 on a graph with mean vertex degree 80.

These two experiments indicate that asynchronous execution may be very beneficial in simulated annealing.

# 5 Hybrid Algorithms

Hybrid algorithms recognize that different schemes may be more appropriate at different temperatures. We provide only a cursory review, since previous sections provide algorithmic details.

## 5.1 Modified Systolic and Simple Serializable Set

One hybrid combines a modified systolic algorithm and a simple serializable set algorithm [12]. In the modified systolic algorithm, independent processors copy the current state, then complete a stream of moves at the same temperature. The PICK operation chooses among the results, as per Equations 1 and 2. Equal temperatures for PICK simplify the computations.

At high temperatures, where most moves are accepted, Algorithm SSS provides little benefit—here only the systolic algorithm is used. As the lower temperatures reduce the acceptance rate, the program combines Algorithm SSS with systolic. Finally, at extremely low acceptance rates, the program uses Algorithm SSS exclusively.

Researchers claim this hybrid is slightly faster than the systolic algorithm alone [19].

## 5.2 Random Spatial Decomposition and Functional Decomposition

Another approach combines asynchronous spatial decomposition with functional decomposition [31]. This program randomly distributes the state variables across processors in an iPSC hypercube, to perform VLSI macro-cell placement.

With a 20 macro-cell problem, on a 16 processor iPSC, the algorithm obtained speedups of between 4 and 7.5. Considering the small problem size and the message-passing architecture, the speedup appears very good.

## 5.3 Heuristic Spanning and Spatial Decomposition

One implementation uses heuristic spanning, a non-simulated annealing technique, and asynchronous rectangular decomposition to perform VLSI placement [28].

The heuristic spanning algorithm chooses several random starting states, and iteratively improves each. For the high-cost regime, heuristic spanning shows better convergence behavior than simulated annealing.

In the low cost regime, rectangular decomposition refines the state space to a lower final cost than heuristic spanning could achieve. The rectangular decomposition method showed a speedup of 4.1 on 5 processors, and an extrapolated speedup of 7.1 on 10 processors.

Using the hybrid technique, researchers estimate speedups of 10–13 on 10 processors, when compared to a standard simulated annealing algorithm.

### 5.4 Functional Decomposition and Simple Serializable Set

In another hybrid algorithm, functional decomposition operates at high temperatures, and simple serializable set operates at low temperatures [11]. The poor behavior of Algorithm SSS at high temperatures justifies a different algorithm.

In this early work, researchers sought to avoid convergence problems by using only serial-like algorithms—little was known of altered-generation or asynchronous algorithms. On a 100 cell gate-array placement problem, they achieved a maximum speedup of 2.25 on a 4 processor VAX 11/784.

## 6 Conclusion

We can neatly categorize parallel simulated annealing techniques into serial-like, altered generation, and asynchronous algorithmic classes. Experimental comparisons of these different techniques have appeared only recently, and have been limited in scope [4, 7, 11, 28].

Based on this survey, it appears that asynchronous and altered generation algorithms have provided the best overall speedup, while one serial-like technique, simple serializable set, has been incorporated advantageously at low temperatures. Several experiments indicate promising speedups in asynchronous algorithms.

Fruitful areas of parallel simulated annealing research include the following: empirical comparisons of the three algorithmic classes, using identical problems; characterization of annealing state spaces amenable to altered generation and asynchronous parallel annealing; the development of tuned temperature schedules which compensate for errors in asynchronous algorithms; and adapting work in related areas, such as computational ecologies, to parallel annealing. My colleagues and I are currently exploring these areas.

## Acknowledgements

Miloš D. Ercegovac, Frederica Darema, Stephanie Forrest, Dyke Stiles, Steve R. White, Andrew Kahng, Richard M. Stein, M. Dannie Durand, Andrea Casotto, James Allwright, and Jack B. Hodges reviewed an early draft of this paper, and provided many helpful comments. Responsibility for errors rests with the author.

## 7 References

[1] S. Kirkpatrick, Jr. C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[2] Carl Sechen, Kai-Win Lee, Bill Swartz, Jimmy Lam, and Dahe Chen. TimberWolfSC version 5.4: Row-based placement and routing package. Technical report, Yale University, New Haven, Connecticut, July 1989.

[3] Bernardo A. Huberman and Tad Hogg. The behavior of computational ecologies. In *The Ecology of Computation*, pages 77–113. Elsevier Science Publishers B.V., 1988.

[4] Daniel R. Greening and Frederica Darema. Rectangular spatial decomposition methods for parallel simulated annealing. In *Proceedings of the International Conference on Supercomputing*, pages 295–302, Crete, Greece, June 1989.

[5] Mark Jones and Prithviraj Banerjee. Performance of a parallel algorithm for standard cell placement on the intel hypercube. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 807–813, 1987.

[6] V. Faber, Olaf M. Lubeck, and Andrew B. White, Jr. Superlinear speedup of an efficient sequential algorithm is not possible. *Parallel Computing*, 3(3):259–260, July 1986.

[7] Frederica Darema, Scott Kirkpatrick, and Alan V. Norton. Parallel algorithms for chip placement by simulated annealing. *IBM Journal of Research and Development*, 31(3):391–402, May 1987.

[8] Jimmy Lam. *An Efficient Simulated Annealing Schedule.* PhD thesis, Yale University, New Haven, CT, December 1988.

[9] R. Brouwer and P. Banerjee. A parallel simulated annealing algorithm for channel routing on a hypercube multiprocessor. In *Proceedings of the International Conference on Computer Design*, pages 4–7, 1988.

[10] Joseph Bannister and Mario Gerla. Design of the wavelength-division optical network. Technical Report CSD-890022, UCLA Computer Science Department, May 1989.

[11] Saul A. Kravitz and Rob A. Rutenbar. Placement by simulated annealing on a multiprocessor. *IEEE Transactions on Computer-Aided Design*, CAD-6(4):534–549, July 1987.

[12] F.M.J. de Bont, E.H.L. Aarts, P. Meehan, and C.G. O'Brien. Placement of shapeable blocks. *Philips Journal of Research*, 43(1):1–27, April 1988.

[13] Roger D. Chamberlain, Mark N. Edelman, Mark A. Franklin, and Ellen E. Witte. Simulated annealing on a multiprocessor. In *Proceedings of the International Conference on Computer Design*, pages 540–544, 1988.

[14] James R.A. Allwright and D.B. Carpenter. A distributed implementation of simulated annealing for the travelling salesman problem. *Parallel Computing*, 10(3):335–338, May 1989.

[15] Edward Felten, Scott Karlin, and Steve W. Otto. The traveling salesman problem on a hypercubic, mimd computer. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 6–10, St. Charles, Pennsylvania, 1985.

[16] Srinivas Devadas and A. Richard Newton. Topological optimization of multiple level array logic: On uni and multi-processors. In *Proceedings of the International Conference on Computer-Aided Design*, pages 38–41, Santa Clara, CA, November 1986.

[17] M.D. Durand. Cost function error in asynchronous parallel simulated annealing algorithms. Unpublished manuscript, 1989.

[18] D. Abramson. Constructing school timetables using simulated annealing: Sequential and parallel algorithms. Technical Report TR 112 069, Department of Communication and Electrical Engineering, Royal Melbourne Institute of Technology, Melbourne, Australia, January 1989.

[19] Emile H.L. Aarts, Frans M.J. de Bont, Erik H.A. Habers, and Peter J.M. van Laarhoven. A parallel statistical cooling algorithm. In *Proceedings of the Symposium on the Theoretical Aspects of Computer Science*, volume 210, pages 87–97, January 1986.

[20] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equations of state calculations by fast computing machines. *Journal of Chemical Physics*, 21(6):1087–1091, June 1953.

[21] Saul B. Gelfand and Sanjoy K. Mitter. Simulated annealing with noisy or imprecise energy measurements. to appear in *Journal of Optimization Theory and Applications*, 1989.

[22] Lov K. Grover. Simulated annealing using approximate calculation. In *Progress in Computer Aided VLSI Design, volume 6*. Ablex Publishing Corp., 1989. (also as Bell Labs Technical Memorandum 52231-860410-01, 1986).

[23] D. Chazan and W. Miranker. Chaotic relaxation. *Linear Algebra and Its Applications*, 2(2):199–222, April 1969.

[24] Prasanna Jog and Dirk Van Gucht. Parallelisation of probabilistic sequential search algorithms. In *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, pages 170–176, 1987.

[25] Rajeev Jayaraman and Frederica Darema. Error tolerance in parallel simulated annealing techniques. In *Proceedings of the International Conference on Computer Design*, pages 545–548. IEEE Computer Society Press, 1988.

[26] Prithviraj Banerjee and Mark Jones. A parallel simulated annealing algorithm for standard cell placement on a hypercube computer. In *Proceedings of the International Conference on Computer-Aided Design*, pages 34–37, November 1986.

[27] Lov K. Grover. A new simulated annealing algorithm for standard cell placement. In *Proceedings of the International Conference on Computer-Aided Design*, pages 378–380. IEEE Computer Society Press, November 1986.

[28] Jonathan S. Rose, W. Martin Snelgrove, and Zvonko G. Vranesic. Parallel standard cell placement algorithms with quality equivalent to simulated annealing. *IEEE Transactions on Computer-Aided Design*, 7(3):387–396, March 1988.

[29] Andrea Casotto, Fabio Romeo, and Alberto Sangiovanni-Vincentelli. A parallel simulated annealing algorithm for the placement of macro-cells. *IEEE Transactions on Computer-Aided Design*, CAD-6(5):838–847, September 1987.

[30] Frederica Darema, Scott Kirkpatrick, and V. Alan Norton. Parallel techniques for chip placement by simulated annealing on shared memory systems. In *Proceedings of the International Conference on Computer Design*, pages 87–90, October 1987.

[31] Rajeev Jayaraman and Rob A. Rutenbar. Floorplanning by annealing on a hypercube multiprocessor. In *Proceedings of the International Conference on Computer-Aided Design*, pages 346–349, November 1987.