Parallel Stochastic Evolution Algorithms for Constrained Multiobjective Optimization

Abstract

Stochastic evolution (StocE) is an evolutionary metaheuristic that has shown to achieve better solution qualities and runtimes when compared to some other well established stochastic metaheuristics. However, unlike these metaheuristics, parallelization of StocE has not been explored before. In this paper, we discuss a comprehensive set of parallel strategies for StocE using a constrained multiobjective VLSI cell placement as an optimization problem. The effectiveness of the proposed strategy is demonstrated by comparing its results with results of parallel SA algorithms on the same optimization problem.

1. Introduction

Evolutionary metaheuristics are increasingly being applied to a variety of combinatorial optimization problems, especially with vast multi-modal search spaces, which cannot be efficiently navigated by deterministic algorithms. Stochastic evolution (StocE) [7] is a randomized iterative search algorithm, similar to the other well known metaheuristics such as Simulated Annealing (SA), Genetic Algorithms (GA) and Tabu Search (TS) [9]. It is inspired by the alleged behavior of biological processes. Compared to SA, each move in StocE is a compound move. Also, it allows controlled uphill moves throughout the search process, unlike SA behavior in the cold regime. StocE has demonstrated improvements in runtime and solution quality over the more established metaheuristics when applied to the same problem instance [10].

With large problem complexities and conflicting optimization objectives, the runtime requirement for achieving near optimal solutions can be prohibitively high. Parallelization of metaheuristics aims to solve complex problems and traverse larger search spaces in a reasonable amount of time. The goals of parallelization can be to achieve either lower runtimes for the same quality solutions as the sequential algorithm or higher quality solutions in a limited amount of time [3]. However, determining an appropriate parallel approach can be a non-trivial exercise. The factors to be considered include the nature of the problem domain (solution landscape), the metaheuristic structure, and the parallel environment. Parallelization of metaheuristics is an actively researched topic [3], but unlike SA, GA, and TS [2, 4], parallelization of StocE has not been studied. In this work, parallel algorithms for StocE are presented considering a complete spectrum of parallel models [2]. VLSI cell placement is used as an optimization problem and the goal is to achieve scalable speed-ups using a low-cost cluster environment. It is found that parallelization of StocE increases its effectiveness in solving large, multi-objective optimization problems. A comparison with parallel SA strategies is also given, which further highlights the performance gains.

This paper is organized as follows: Section 2 briefly discusses the optimization problem and costs functions. This is followed by a description of StocE algorithm in Section 3 and its sequential runtime analysis in Section 4. Section 5 presents the proposed strategies, while experimental results and comparison are given in Section 6. Section 7 concludes the paper.

2. Optimization Problem and Cost Functions

This paper addresses the problem of VLSI standard cell placement with the objectives of minimizing wirelength, power consumption, and timing performance (delay), while considering the layout width as a constraint.

Wirelength Cost: Interconnect wirelength of each net in the circuit is estimated using Steiner tree and then total wirelength is computed by adding the individual estimates:

$$Cost_{wire} = \sum_{i \in M} l_i$$

where l_i is the wirelength estimation for net i and M denotes total number of nets in circuit.

Power Cost: Power consumption p_i of a net i in a circuit can be given as:

$$p_i \simeq l_i \cdot S_i$$

where, S_i is the switching probability of net i. The cost function for estimate of total power consumption in the circuit can be given as:

$$Cost_{power} = \sum_{i \in M} p_i = \sum_{i \in M} (l_i \cdot S_i)$$

Delay Cost: This cost is determined by the delay along the longest path in a circuit. The delay T_π of a path π consisting of nets $\{v_1, v_2, ..., v_k\}$, is expressed as:

$$T_\pi = \sum_{i=1}^{k-1} (CD_i + ID_i)$$

where CD_i is the switching delay of the cell driving net vi and ID_i is the interconnect delay of net vi. The delay cost function can be written as:

$$Cost_{delay} = max\{T_\pi\}$$

Width Cost: Width cost is the maximum of all the row widths in the layout. Formally, width constraint is expressed as:

$$Width - w_{avg} \leq \alpha \times w_{avg}$$

where, α is a constant and w_{avg} is the minimum possible layout width.

Overall Fuzzy Cost Function: In this work, fuzzy logic is used to integrate multiple, conflicting objectives into a scalar cost function [10]. The resulting quality measure for a solution s is denoted as $\mu(s)$ and is a value between 0 and 1, with 1 representing an optimal solution.

```
AlgorithmStocE($S_0$, $p_0$, $R$);
Begin
    $BestS$= $S = S_0$;
    $BestCost$= $CurCost$= $Cost(S)$;
    $p = p_0$;
    $\rho = 0$;
    Repeat
        $PrevCost$= $CurCost$;
        $S = PERTURB(S, p)$;
    /* perform a search in the neighborhood of s */
        $CurCost$= $Cost(S)$;
        $UPDATE(p, PrevCost, CurCost)$;
    /* update p if needed */
        If ($CurCost$< $BestCost$) Then
            $BestS$=$S$;
            $BestCost$= $CurCost$;
            $\rho = \rho - R$;
    /* Reward the search with $R$ more generations */
        Else
            $\rho = \rho + 1$;
        EndIf
    Until $\rho > R$
    Return ($BestS$);
End
```

Figure 1. The Stochastic Evolution algorithm.

3. Sequential StocE Algorithm

The StocE algorithm seeks to find a suitable location $S(m)$ for each movable element $m \in M$, which eventually leads to a lower cost of the whole state $S \in \Omega$, where Ω is the state space. A general outline of the StocE algorithm is given in Figure 1. The inputs to the StocE algorithm are, an initial state (solution) S_0, an initial value p_0 of the control parameter p, and a stopping criterion parameter R. Throughout the search, S holds *the current state (solution)*, while $BestS$ holds *the best state*. If the algorithm generates *a worse state*, a uniformly distributed random number in the range $[-p, 0]$ is drawn. The new uphill state is accepted if the magnitude of the loss is greater than the random number, otherwise the current state is maintained. Therefore, p is a function of the average magnitude of the uphill moves that the algorithm will tolerate. The parameter R represents the expected number of iterations the StocE algorithm needs until an improvement in the cost with respect to the best solution seen so far takes place, that is, until $CurCost \leq BestCost$. If R is too small, the algorithm will not have enough time to improve the initial solution, and if R is too large, the algorithm may waste too much time during the later generations. Experimental studies indicate that a value of R between 10 and 20 gives good results [7]. Finally, the variable ρ is a counter used to decide when to stop the search. ρ is initialized to zero, and $R - \rho$ is equal to the number of remaining generations before the algorithm stops.

After initialization, the algorithm enters a **Repeat** loop **Until** the counter ρ exceeds R. Inside the **Repeat** body, the cost of the current state is first calculated and stored in $PrevCost$. Then, the **PERTURB** function (see Figure 2) is invoked to make a compound move from the current state S. **PERTURB** scans the set of movable elements M according to some apriori ordering and attempts to move every $m \in M$ to a new location $l \in L$. For each trial move, a new state S' is generated, which is *a unique* function $S' : M \rightarrow L$ such that $S'(m) \neq S(m)$ for some movable object $m \in M$. To evaluate the move, the gain function $Gain(m) = Cost(S) - Cost(S')$ is calculated. If the calculated gain is greater than some randomly generated integer number in the range $[-p, 0]$, the move is accepted and S' replaces S as the current state, assuming a minimization problem. Since the random number is ≤ 0, moves with positive gains are always accepted. After scanning all the movable elements $m \in M$, the **MAKE_STATE** routine makes sure that the final state satisfies the state constraints. If the state constraints are not satisfied then **MAKE_STATE** *reverses* the fewest number of latest moves until the state constraints are satisfied. This procedure is required when perturbation moves that violate the state constraints are accepted.

```
FUNCTION PERTURB($S, p$);
Begin
    ForEach ($m \in M$) Do
/* according to some apriori ordering */
        $S' = MOVE(S, m)$;
        $Gain(m) = Cost(S) - Cost(S')$;
        If ($Gain(m) > RANDINT(-p, 0)$) Then
                $S = S'$
        EndIf
    EndFor;
    $S$ =MAKE_STATE($S$);
/* make sure $S$ satisfies constraints */
    Return ($S$)
End
```

Figure 2. The PERTURB function.

The new state generated by **PERTURB** is returned to the main procedure as the current state, and its cost is assigned to the variable $CurCost$. Then the routine **UPDATE** (Figure 3) is invoked to compare the previous cost ($PrevCost$) to the current cost ($CurCost$). If $PrevCost$= $CurCost$, there is a good chance that the algorithm has reached a local minimum and therefore, p is increased by p_{incr} to tolerate larger uphill moves, thus giving the search the possibility of escaping from local minima. Otherwise, p is reset to its initial value p_0.

At the end of the loop, the cost of the *current state* S is compared with the cost of the *best state* $BestS$. If S has a lower cost, then the algorithm keeps S as the best solution ($BestS$) and decrements R by ρ, thereby rewarding itself by increasing the number of iterations (allowing the search to live R generations more). This allows a more detailed investigation of the neighborhood of the newly found best solution. If S, however, has a higher cost, ρ is incremented, which is an indication of no improvements.

```
PROCEDURE UPDATE($p$, $PrevCost$, $CurCost$);
Begin
    If ($PrevCost$=$CurCost$) Then
/* possibility of a local minimum */
        $p = p + p_{\text{incr}}$;
/* increment $p$ to allow larger uphill moves */
    Else
        $p = p_0$; /* re-initialize $p$ */
    EndIf;
End
```

Figure 3. The UPDATE procedure.

4. Sequential StocE Analysis

Prior to formulating parallelization strategies, the profiling of sequential StocE is presented to identify the time intensive routines and performance bottlenecks, thus serving as a basis to engineer effective parallel approaches. The profiling was done using the GNU 'gprof' utility. The percentage of time taken by problem-specific cost computations versus all remaining functions is documented in Table 1. The profiling results clearly demonstrate that more than 90% of time is spent in the cost function calculations of wirelength, power and delay, thereby highlighting the computational complexity involved.

Table 1. Runtime profile of sequential StocE

Circuit Name	Number of Cells	Cost Functions calculation	Other Functions
s1494	661	93.15%	6.85%
s3330	1961	92.86%	7.14%
s5378	2993	93.43%	6.57%
s9234	5844	92.87%	7.13%
s15850	10383	89.64%	10.36%

5. Parallel StocE Algorithms

Given the profiling results, an intuitive approach would be to parallelize the cost functions to achieve a low level parallelization. However, due to the nature of data dependencies involved, this strategy is not suited to a coarse grained parallel environment, where node-to-node communications are high [8]. Another parallelization approach that can be considered is one that divides the solution into independent domains, each to be operated in parallel. This strategy seems attractive as the total cost calculations will be distributed across all processors. Finally, a third type of parallelization that can be attempted is known as cooperative parallel searches (Asynchronous Multiple Markov Chains), where parallel threads each running a complete StocE process cooperate with each other to quickly reach good solutions.

5.1. Asynchronous Multiple Markov Chains (AMMC)

This strategy exploits the capability of multiple, concurrent threads to cooperatively navigate the search space. With this parallel approach, there is no workload division as each processor runs the whole StocE algorithm. It has been reported as a successful strategy for SA parallelization, demonstrating good runtime trends [6]. A similar approach is adopted in this work for StocE, utilizing the advantages of AMMC in terms of relaxing the synchronization requirements among individual processors. Since StocE is strictly sequential in nature, the asynchronous feature of AMMC reduces the inter-processor communication cost, and can be intuitively considered to perform well. Moreover, StocE follows a search path based on randomization, which determines the acceptance/rejection of moves. Hence, each of these paths can be viewed as as a separate Markov chain exploring a different region of the solution space (by using different random seeds). Moreover, the search process is biased by propagating the best solution among all processors. Thus, whenever any processor reaches a solution

```
Algorithm_Parallel_StocE_AMMC_Master_Process
Notation
 (* $CurS$ is the current solution. *)
 (* $Cost(S)$ returns the cost of solution. *)
 (* $BestS$ is the best solution. *)
Begin
 Read_User_Input_Parameters( )
 Read_Input_Files
 Construct_Initial_Placement
 CurS = S0; //only master has the initial Solution
 BestS = CurS;
 CurCost = Cost(CurS);
 BestCost = Cost(BestS);
      Broadcast(CurS);
 Repeat
          Receive_frm_Slave(BestCost);
          Send_to_Slave(verdict);
          If (verdict == 1)
             Receive_frm_Slave(BestS);
          Else
             Send_to_Slave (BestS);
          EndIf
      Until (All Slaves are done);
      Return(BestS);
 EndIf
End. (*Master_Process*)
```

Figure 4. Master Process for Parallel AMMC StocE Algorithm.

better than the others, it is communicated to all participating nodes, thus intensifying exploration around that region of search space. This AMMC approach uses a master-slave topology, the details of which are shown in Figure 4 and Figure 5. The slave processor upon reaching a better solution sends the cost metric to the master node. The master compares this with the current best received by it has. If found better, the slave is instructed to send the entire solution; else, the master sends the solution it has to the slave.

5.2. Rows Division

In contrast to AMMC, this strategy attempts reduction in effective workload by assigning a non-overlapping subset of rows to each processor. A similar parallelization approach was first reported for Simulated Evolution [5]. In this approach, every node is responsible for perturbing cells only within its assigned subset of rows in the overall solution. Two different row allocation patterns are alternated between the successive iterations. This ensures that a cell has the freedom to move to any place in the solution. Figure 6 shows the allocation pattern of rows among three processors. The left and right patterns show the distributions in odd and even numbered iterations, respectively.

Figures 7 and 8 show the parallel algorithms for the master and slave nodes, respectively. As shown in these figures, each processor has the same initial solution and ir calls the ***PERTURB*** function on its allocated subset of non-overlapping rows. The placement generated by a node is termed as a partial solution. These are sent to the master, which combines all partial placements to generate a new complete solution. The master then evaluates this new solu-

```
Algorithm_Parallel_StocE_AMMC_Slave_Process
Notation
 (* $CurS$ is the current solution. *)
 (* $Cost(S)$ returns the cost of solution. *)
 (* $BestS$ is the best solution. *)
Begin
 Read_User_Input_Parameters( )
 Read_Input_Files
      Receive_Initial_Sol(CurS);
 CurS = S0;
 BestS = CurS;
 CurCost = Cost(CurS);
 BestCost = Cost(BestS);
 Repeat
          $S = PERTURB(S, p)$;
          /* perform a search in the neighborhood of s */
          $CurCost$= $Cost(S)$;
          $UPDATE(p, PrevCost, CurCost)$; /* update $p$ if needed */
          If ($CurCost$< $BestCost$) Then
             $BestS$=$S$;
             $BestCost$= $CurCost$;
             $\rho = \rho - R$;
             /* Reward the search with $R$ more generations */
          Else
             $\rho = \rho + 1$;
          EndIf
          Send_to_Master(BestCost);
          Receive_frm_Master(verdict);
          If (verdict == 1)
             Send_to_Master (BestS);
          Else
             Receive_frm_Master(BestS);
          EndIf
      Until $\rho > R$
      Return ($BestS$);
 End
```

Figure 5. Slave Process for Parallel AMMC StocE Algorithm.

tion and depending on the new cost, either increments rho or decrements it by R. This new solution is then again broadcasted to all slaves. This process continues till the target fitness value is achieved or termination criteria is met.

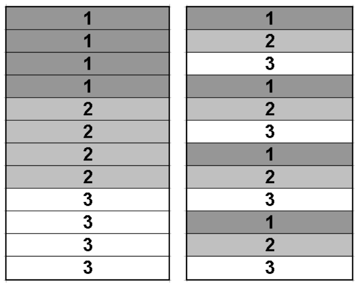

Figure 6. Rows Division

6. Experiments

The programs were written in C using the MPI library (MPICH 1.2.5). A dedicated cluster of eight 2GHz Pentium 4 machines, with 256MB of RAM, connected with 100 Mbps Ethernet, running Redhat Linux 7.2 (kernel 2.4.7-10) was used. ISCAS-89 benchmarks circuits are used, which

```
Algorithm Parallel_StocE_Master_Process
Notation
  (* $CurS$ is the current solution. *)
  (* $\Phi_s$ is the partition selected to work upon. *)
Begin
  Read_User_Input_Parameters( )
  Read_Input_Files
  Construct_Initial_Placement
  Repeat
          ParFor
              Slave_ Process($CurS$)
              (* Broadcast Cur Placement. *)
          EndParFor
          $S = PERTURB(S, p)$;
          /* perform a search in the restricted neighborhood of s */
          (* For each slave process. *)
          ParFor
              Receive_Partial_Solutions
          EndParFor
          Make_Complete_Solution
          $CurCost$= $Cost(S)$;
          $UPDATE(p, PrevCost, CurCost)$;
          /* update $p$ if needed */
          If ($CurCost$< $BestCost$) Then
              $BestS$=$S$;
              $BestCost$= $CurCost$;
              $\rho = \rho - R$;
              /* Reward the search with $R$ more generations */
          Else
              $\rho = \rho + 1$;
          EndIf
      Until $\rho > R$
  Return (Best_Solution)
End. (*Master_Process*)
```

Figure 7. Master Process for Rows Division Parallel StocE Algorithm.

contain a set of circuits of various sizes in terms of number of gates and paths.

Table 2 shows the performance of the AMMC approach. It can be seen that the parallel algorithm achieves negligible, if any, reduction in runtime with increasing processors. The reasons for such behavior are discussed later in this section. The results of rows division strategy are given in Table 3. The maximum number of processors that can be used is limited by the number of rows, and hence for small circuits, results are presented accordingly. As seen, this workload division approach delivers excellent runtime gains, especially for larger circuits. This is reflected in Figure 6, which shows the speedup achieved with increasing number of processors for different circuits.

The intelligence and effectiveness of any optimization algorithm is determined by how it navigates the search space with a strong bias towards higher quality solutions. The algorithmic intelligence of StocE lies in its perturbation

Table 2. Results for AMMC parallel strategy.

Circuit Name	# Cells	$\mu(s)$	Seq. Time	Parallel Times				
				p=2	p=3	p=4	p=5	p=6
s1494	661	0.6	94	32	32	32	32	34
s3330	1961	0.6	186	96	95	89	92	95
s5378	2993	0.6	450	268	270	265	270	268
s9234	5844	0.6	1143	799	802	800	799	799
s15850	10383	0.6	2103	1908	1903	1908	1905	1908

```
Algorithm Parallel_StocE_Slave_Process($CurS, \Phi_s$)
Notation
  (* $CurS$ is the current solution. *)
  (* $\Phi_s$ is the partition caculated by the slave $s$ to work upon. *)
  (* $m_i$ is module $i$ in $\Phi_s$. *)
Begin
  Read_User_Input_Parameters( )
  Read_Input_Files
  Construct_Initial_Placement
  Repeat
  Receive Placement
      $S = PERTURB(S, p)$;
      /* perform a search in the restricted neighborhood of s */
          Send_Partial_Solution
    Until Fitness_Value_not_achieved
End. (*Slave_Pocess*)
```

Figure 8. Slave process for Rows Division Parallel StocE Algorithm.

function, method of calculating and updating the control parameter p, and in selecting an appropriate termination criteria represented by R. Though the AMMC approach works well with SA [6], the results with StocE are very limited, showing only runtime gains for upto two processors. The apparent reason for this limited performance is that each thread of StocE performs a compound move that optimizes the solution to a large extent. In addition, the self awarding criteria of StocE, triggered on finding good solutions, relaxes the termination criteria. This, in effect, gives each processor enough time to keep improving the solution when in local minima and thus the cooperation from other processors gives no noticeable benefit. As a result, StocE fails to show any benefit using a parallel search.

Table 3. Results for rows division strategy.

Circuit Name	$\mu(s)$	Time Serial	Runtime for Parallel StocE				
			p=2	p=3	p=4	p=5	p=6
s1494	0.6	60	49	55	112	-	-
s3330	0.7	1087	355	214	190	186	170
s5378	0.65	1047	495	365	311	305	293
s9234	0.65	2140	1261	917	704	616	615
s15850	0.65	3538	2876	1841	1543	1423	1167

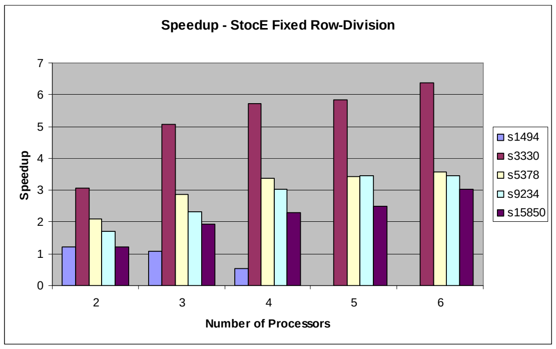

Figure 9. Speedup trend for Parallel StocE with Rows Division

In the rows division parallelization approach, the workload is efficiently distributed by dividing the solution among multiple processors, without disturbing the intelligence of the perturbation mechanism. Here, the Master processor remains in charge of updating and controlling parameters. This direct workload distribution was the primary reason behind the favorable speedup trends seen with this strategy.

A comparison between parallel StocE algorithm using rows division and parallel SA using AMMC as well as rows division is now presented. The focus is on the speed-up achieved for the best fitness values achieved by StocE. The speed-up is defined as follows [1]: Let t_1 denote the worst case running time of the fastest known sequential algorithm for the problem, and let t_p denote the the worst case running time of the parallel algorithm using p processors. Then, the speedup provided by the parallel algorithm is

$$S(1,p) = \frac{t_1}{t_p} \tag{1}$$

Figure 10 graphically depicts the results of comparison using the s15850 ISCAS-89 benchmark. The figure shows reduction in runtime against increasing number of processors as well as the corresponding speedups achieved. All the speedups have been calculated using the best sequential time available, which is the sequential time of StocE. It can be seen that for SA, AMMC gives better results as compared to rows division parallelization model. In case of StocE, since the AMMC model fails to produce any speedup (Table 2), the results are not depicted. The rows division method for StocE gives the best results compared to all other strategies, as it achieves significant workload division without affecting algorithm's intelligence. As can be seen for the benchmark circuit s15850 in Figure 10, StocE rows division outperforms all the parallel versions by achieving the target solution quality in 1167 seconds with 6 processors. To achieve the same solution quality SA using AMMC model took 1500 seconds with 6 processors. SA with rows division model stood last by achieving the targeted solution quality in 3473 seconds with 6 processors and thus failed in producing any speedup. Another interesting feature, making StocE rows division model outstanding is the performance enhancement using less resources. StocE rows division gives speedup of nearly 2 employing 3 processors only and still keeps improving on addition of extra processors while SA AMMC used 5 processors to produce a slightly better speedup than what StocE has achieved with 3 processors. Thus, the rows division model understandably works best for StocE on large circuits with extensive number of rows and cells. The comparisons were made across all the circuits listed in Table 4 and similar trends were seen. It was found that as the circuit size increases, the difference among the performance of parallel StocE and SA implementations becomes more pronounced. The results shown and discussed here represent the behavior of the parallel strategies on a large size problem.

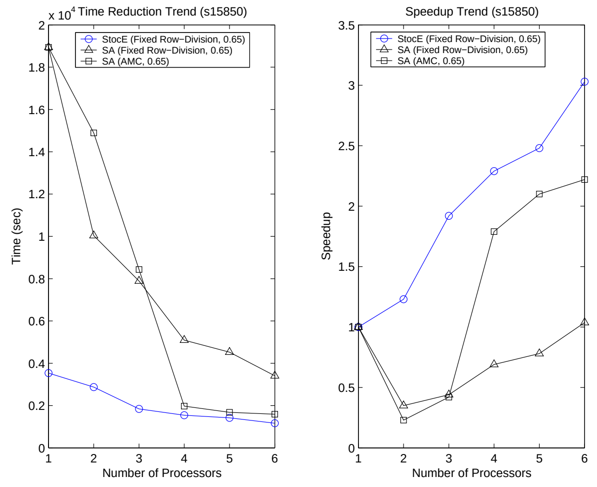

Figure 10. StocE Vs SA. The left and right figures respectively show the average run-time reduction and average speedup.

7. Conclusions

In this paper, two applicable parallelization models for StocE were presented for the VLSI cell placement problem. Low-Level parallelization appeared as an ineffective approach, given the coarse grain parallel environment. A parallel search model using AMMC approach was implemented and was found to give very limited speedups while a domain decomposition parallelization using rows division strategy gave excellent results. Runtimes were compared against the AMMC and rows division parallel SA models and it was found that parallel StocE performs better than parallel SA implementations.

References

[1] S. G. Akl. *Parallel Computation: Models And Methods*. 1997.

[2] T. G. Crainic and M. Toulouse. *Handbook of Metaheuristics*, volume 57, chapter Parallel Strategies for Metaheuristics, pages 465 – 514. Kluwer Academic Publishers, 2003.

[3] V.-D. Cung, S. L. Martins, C. C. Ribeiro, and C. Roucairol. *Essays and Surveys in Metaheuristics*, volume 15, chapter Strategies for the Parallel Implementation of Metaheuristics, pages 263 – 308. Kluwer Academic Publishers, 2001.

[4] S. D. Ekşioğlu, P. M. Pardalos, and M. G. C. Resende. *Models for Parallel and Distributed Computation - Theory, Algorithmic Techniques and Applications*, chapter Parallel Metaheuristics for Combinatorial Optimization, pages 179 – 206. Kluwer Academic Publishers, June 2002.

[5] R. M. Kling and P. Banerjee. Esp: Placement by simulated evolution. *IEEE Transaction on Computer-Aided Design*, 1989.

[6] S.-Y. Lee and K.-G. Lee. Synchronous and asynchronous parallel simulated annealing with multiple markov chains. *IEEE Transactions on Parallel & Distributed Systems*, October 1996.

[7] Y. G. Saab and V. B. Rao. Stochastic evolution : A fast effective heuristic for some generic layout problems. *27th ACM/IEEE Design Automation Conference*, pages 1–6, 1990.

[8] S. M. Sait, M. I. Ali, and A. M. Zaidi. Evaluating Parallel Simulated Evolution Strategies for VLSI Cell Placement. In *20th International Parallel and Distributed Processing Symposium*, April 2006.

[9] S. M. Sait and H. Youssef. *Iterative Computer Algorithms and their Application to Engineering*. IEEE Computer Society Press, December 1999.

[10] S. M. Sait, H. Youssef, J. A. Khan, and A. El-Maleh. Fuzzified iterative algorithms for performance driven lowpower vlsi placement. *IEEE Proceedings of the International Conference on Computer Design*, 2001.