# Parallel Strategies for Stochastic Evolution

Sadiq M. Sait, Khawar S. Khan, Mustafa I. Ali
*Computer Engineering Department*
*King Fahd University of Petroleum & Minerals, Dhahran, Saudi Arabia*
{*sadiq,khawar,mustafa*}*@ccse.kfupm.edu.sa*

## Abstract

*The paper discusses the parallelization of Stochastic Evolution metaheuristic, identifying effective parallelization for a distributed parallel environment. Multiobjective VLSI cell placement is used as an optimization problem. A comprehensive set of parallelization approaches are tested and an effective strategy is identified in terms of two underlying factors: workload division and the effect of parallelization on metaheuristic's search intelligence. The strategies are compared with parallelization of another similar evolutionary metaheuristic called Simulated Evolution. The role of the two mentioned underlying factors is discussed in parallelization of stochastic evolution, the parallelized version of which has not been presented before.*

## 1. Introduction

Evolutionary metaheuristics are being increasingly 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) [6] and Simulated Evolution (SimE) [5] are evolutionary iterative search algorithms, similar to other well known iterative non-deterministic heuristics such as Simulated Annealing (SA), Genetic Algorithms (GA) and Tabu Search (TS) [8]. The two algorithms are inspired by the alleged behavior of biological processes, however, they differ fundamentally in how the principles of evolution are applied. Both algorithms have demonstrated improvements in runtime and solution quality over the more established heuristics when applied to the same problem instance [9].

Parallelization of metaheuristics aims to solve complex problems and traverse larger search spaces in a reasonable amount of time [4, 2]. However, when parallelizing metaheuristics, not only speed-ups are important but also the maximum achievable qualities. Therefore, to achieve any benefit from parallelization requires not only a proper partitioning of the problem for a uniform distribution of computationally intensive tasks, but more importantly, a thorough and intelligent traversal of a complex search space for achieving good quality solutions. The tractability of the former issue is largely dependent on parallelization of both the cost computation and perturbation functions while for the latter issue the interaction of parallelization strategy with the *intelligence* of the heuristic must be considered, as it directly affects the final solution quality obtainable, and indirectly the runtime due to its effect on algorithm's convergence. Parallelization of metaheuristics is an actively researched topic [4, 2]. However, unlike other heuristics, parallelization of StocE has not been studied before. In this work, parallel algorithms for StocE are presented, considering a spectrum of parallel models [4]. The approaches are also compared with parallel SimE due to the similarities in both heuristics [7]. VLSI cell placement is used as an optimization problem and the goal is to achieve scalable speed-ups using a low-cost cluster computing environment. The best parallel strategies for both SimE and StocE are compared with respect to the effectiveness of parallelization in terms of workload division and the effect of parallelization on metaheuristic's intelligence.

This paper is organized as follows: Section 2 briefly discusses the optimization problem and costs functions. This is followed by a description of StocE and SimE algorithms in Section 3 and the sequential algorithms' runtime analyses in Section 4. Section 5 presents the proposed parallel strategies, experimental results and comparison. Section 6 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. The interconnect wirelength of each net in the circuit is estimated using Steiner tree and then total wirelength is computed by adding the individual estimates, i.e., $Cost_{wire} = \sum_{i \in M} l_i$,

0-7695-2976-3/07 $25.00 © 2007 IEEE
DOI 10.1109/ISDA.2007.71

where $l_i$ is the wirelength estimation for net $i$ and $M$ denotes total number of nets in circuit. The power consumption $p_i$ of a net $i$ in a circuit is calculated 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 is $Cost_{power} = \sum_{i \in M} p_i = \sum_{i \in M} (l_i \cdot S_i)$. The delay cost is determined by the delay along the longest path in a circuit. The delay cost is determined by the delay along the longest path in a circuit. The delay $T_\pi$ of a path $\pi$ 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 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 $\alpha$ is a constant and $w_{avg}$ is the minimum possible layout width. Finally, to integrate these multiple conflicting objectives into a scalar cost function, fuzzy logic is used in this work [9]. 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.

## 3. Evolutionary Metaheuristics

### 3.1. Stochastic Evolution (StocE)

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 $\Omega$ 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 [6]. Finally, the variable $\rho$ is a counter used to decide when to stop the search. $\rho$ is initialized to zero, and $R - \rho$ is equal to the number of remaining generations before the algorithm stops.

```
AlgorithmStocE(S_0, p_0, R);
Begin
    BestS= S = S_0;
    BestCost= CurCost= Cost(S);
    p = p_0;
    ρ = 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;
            ρ = ρ − R;
    /* Reward the search with R more generations */
        Else
            ρ = ρ + 1;
        EndIf
    Until ρ > R
    Return (BestS);
End
```

**Figure 1. The StocE algorithm.**

After initialization, the algorithm enters a **Repeat** loop **Until** the counter $\rho$ exceeds $R$. Inside the **Repeat** body, the cost of the current state is first calculated and stored in $PrevCost$. Then, the **PERTURB** function (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 $\leq 0$, 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.

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** 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 $p_{\text{incr}}$ to tolerate larger uphill moves, thus giving the search the possibility of escaping from local minima. Otherwise, $p$ is reset to its initial

```
FUNCTION PERTURB(S, p);
Begin
    ForEach (m ∈ 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.**

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 $\rho$, 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, $\rho$ is incremented, which is an indication of no improvements.

### 3.2. Simulated Evolution (SimE)

The structure of the SimE algorithm is shown in Figure 3. SimE assumes that there exists a solution $\Phi$ of a set $M$ of $n$ (movable) elements or modules. The algorithm starts from an initial assignment $\Phi_{initial}$, and then, following an evolution-based approach, it seeks to reach better assignments from one generation to the next by perturbing some ill-suited components while retaining the remaining ones. A cost function $Cost$ associates with each assignment of movable element $m_i$ a cost $C_i$. The cost $C_i$ is used to compute the goodness (fitness) $g_i$ of an element $m_i$, for each $m_i \in M$. The goodness measure must be strongly related to the target objective of the given problem. Hence in SimE approach, the quality of a solution can be measured as the quality of all its constituent elements.

The algorithm has one main loop consisting of three basic steps, *Evaluation*, *Selection*, and *Allocation*. The three steps are executed in sequence until the solution average *goodness* reaches a maximum value, or no noticeable improvement to the solution *fitness* is observed after a number of iterations.

The *Evaluation* step consists of evaluating the *goodness* $g_i$ of each element $m_i$ of the solution $\Phi$. The *goodness* measure must be a single number expressible in the range $[0, 1]$. It is generally defined as $g_i = \frac{O_i}{C_i}$, where $O_i$ is an estimate of the optimal cost of element $m_i$, and $C_i$ is the actual cost of $m_i$ in its current location. Since three objectives are being optimized, a multiobjective goodness measure developed in [9] is used.

```
ALGORITHM Simulated_Evolution(B, Φ_initial)
NOTATION
B: Bias Value.      Φ: Complete solution.
m_i: Module i.      g_i: Goodness of m_i.
ALLOCATE(m_i, Φ_i): Allocates m_i in partial solution Φ_i
Begin
INITIALIZATION;
Repeat
  EVALUATION:
        ForEach m_i ∈ Φ evaluate g_i;
  SELECTION:
        ForEach m_i ∈ Φ DO
            begin
                IF Random > Min(g_i + B, 1)
                THEN
                    begin
                        S = S ∪ m_i; Remove m_i from Φ
                    end
            end
      Sort the elements of S
  ALLOCATION:
        ForEach m_i ∈ S DO
            begin
                ALLOCATE(m_i, Φ_i)
            end
Until  Stopping Condition is satisfied
Return Best solution.
End (Simulated_Evolution)
```

**Figure 3. The SimE algorithm.**

The second step of the SimE algorithm is *Selection*. *Selection* takes as input the solution $\Phi$ together with the estimated *goodness* of each element, and a bias value $B$ to compensate for non-ideal nature of the calculated goodness values. It partitions $\Phi$ into two disjoint sets; a selection set $S$ and a partial solution $\Phi_p$ of the remaining elements of the solution $\Phi$. Each element in the solution is considered separately from all other elements. The probability of assigning an element $m_i$ to the set $S$ is based on its *goodness* $g_i$. The selection operator has a non-deterministic nature, i.e, an individual with a high *goodness* (close to one) still has a non-zero probability of being assigned to the selection set $S$. It is this element of non-determinism that gives SimE the capability of escaping local minima. In this work, a biasless selection function developed in [9] has been used.

*Allocation* is the SimE operator that has the most important impact on the quality of solution. *Allocation* takes as input the set $S$ and the partial solution $\Phi_p$ and generates a new complete solution $\Phi'$ with the elements of set $S$ mutated according to an allocation function $Allocation$ [8]. The goal of *Allocation* is to favor improvements over the previous generation, without being too greedy. A variety of heuristics can be used in this step [5]. In this work, the 'sorted individual best fit method' [9] has been used.

## 4. Sequential Algorithms' Analyses

Prior to formulating parallelization strategies, the profiling of sequential algorithms 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. For

sequential StocE, the percentage of time taken by problem-specific cost computations versus all remaining functions is documented in columns 4 and 5 of 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 identifying where the computational effort is concentrated. For the sequential SimE, on average 98.85% of time was spent in the allocation function alone. Thus, it is obvious that for the given problem instance with the 'sorted individual best-fit' method, allocation routine heavily influences the runtime of SimE algorithm.

**Table 1. Sequential algorithms' runtime profile.**

| Circuit | # of Cells | # of Rows | StocE | | SimeE | |
|---|---|---|---|---|---|---|
| | | | Cost Functions | Others | Allocation Function | Others |
| s1494 | 661 | 11 | 93.1 | 6.8 | 97.6 | 2.3 |
| s3330 | 1961 | 17 | 92.9 | 7.1 | 99.3 | 0.6 |
| s5378 | 2993 | 22 | 93.4 | 6.6 | 99.2 | 0.7 |
| s9234 | 5844 | 22 | 92.9 | 7.1 | 99.3 | 0.4 |

## 5. Parallel Algorithms and Experiments

Given the StocE profiling results, an intuitive approach is 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. This was confirmed by the results obtained when this strategy was applied to parallelize SimE for the same problem [7].

Cooperative parallel searches is another parallelization approach that can be attempted, where parallel threads each running a complete StocE/SimE process cooperate with each other (by exchanging good solutions) to quickly converge. Parallel search aims to achieve speed-up by enhancing the search behavior rather than workload division. This type of parallelization has reportedly worked well with SA (Asynchronous Multiple Markov Chains) [3]. A similar approach was applied to parallelize StocE but very limited speedups, if any, were obtained (the results are not presented due to space constraint). The reason is that each StocE thread performs a compound move that optimizes the solution to a large extent without any cooperation. Furthermore, the self-rewarding criteria of StocE, triggered on finding good solutions, relaxes the termination criteria (Section 3.1). Due to this, each processor keeps attempting to further improve the solution by itself without cooperation from other processors. The net effect is no noticeable benefit from cooperative parallel searches. It should be noted that similarly poor results were obtained for SimE with this strategy [7].

A third kind of parallelization is one that divides the solution into independent domains, each to be operated in parallel [4]. This strategy seems attractive as it distributes the total cost calculations among the processors. It attempts reduction in workload by assigning a non-overlapping subset of rows to each processor and thus it is termed as *rows division* strategy. A similar parallelization approach was reported for SimE [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 4 shows the allocation pattern of twelve rows among three processors. The left and right patterns show the distributions in odd and even numbered iterations, respectively.

For the domain decomposition parallel SimE, the ele-

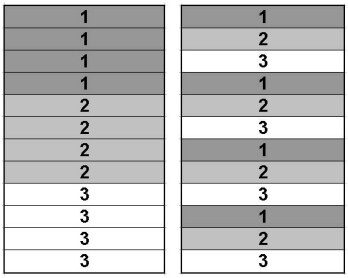


**Figure 4. Rows Division**

**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 5. Outline of master process for rows division parallel StocE.**

**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 6. Outline of slave process for rows division parallel StocE.**

ments are partitioned row wise among the $m$ processors. A processor $s$, $1 \leq s \leq m$ would be assigned a subset $\Phi^s$ of the solution $\Phi$. Then, each processor $s$ will evaluate the goodness of each element in $\Phi^s$ and run the *Selection* step to partition $\Phi^s$ into a selection subset $S^s$ and a partial solution of remaining cells ${\Phi_p}^s$ (See the serial algorithm in Figure 3 for comparison). Figures 5 and 6 show the parallel StocE algorithms for the master and slave processes, respectively, for the rows division approach. Each processor starts with the same initial solution and 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 the partial placements to generate a new complete solution. The master then evaluates this new solution 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. It should be noted that the search behavior of the parallel algorithm will differ from the serial algorithm owing to this partitioning.

The parallel programs were written in C using the MPI library (MPICH 1.2.5). A dedicated cluster of eight 2.8 GHz Pentium 4 machines, with 512MB of RAM, connected with 100 Mbps Ethernet, running Fedora Core Linux was used. Only the large ISCAS-89 benchmarks circuits are used to be able to observe gain from parallelization. Due to space constraints, results for four benchmark circuits are presented. Up to 5 processors are used as no significant gains are observed beyond this number due to the size of benchmarks. The results of rows division strategy for StocE and SimE are given in Table 2 and Table 3, respectively. The $\mu(s)$ values represent the highest solution quality achieved by sequential algorithm. Due to the relatively small size of benchmark s1494, no gains are observed beyond 2 processors. In case of parallel SimE, since there is a degradation in the highest $\mu(s)$ values achieved with increase in processors, the highest $\mu(s)$ achieved and the corresponding time is given for different processor counts in Table 3. Also, the row labeled 'Common' gives the time to achieve the common lowest quality. As can be seen, for parallel StocE the domain decomposition approach delivers significant runtime reductions while achieving the target sequential qualities, especially for larger circuits. On the other hand, a domain decomposition parallel SimE implementation achieves lower than highest achievable sequential solution qualities along with a degradation in maximum solution qualities achievable with increase in processors.

**Table 2. Results of rows division parallel StocE.**

| Circuit Name | $\mu(s)$ | Serial Time | Runtimes for parallel StocE | | | |
|---|---|---|---|---|---|---|
| | | | p=2 | p=3 | p=4 | p=5 |
| s1494 | 0.6 | 60 | 49 | 55 | 112 | - |
| s3330 | 0.7 | 1087 | 355 | 214 | 190 | 186 |
| s5378 | 0.65 | 1047 | 495 | 365 | 311 | 305 |
| s9234 | 0.65 | 2140 | 1261 | 917 | 704 | 616 |

**Table 3. Results for SimE Rows Division Strategy.**

| Circuit | | Sequential | Runtimes for parallel SimE | | | |
|---|---|---|---|---|---|---|
| | | | p=2 | p=3 | p=4 | p=5 |
| s1494 | $\mu(s)$ | 0.54 | 0.54 | 0.54 | 0.54 | 0.54 |
| | **Time** | 368 | 111 | 52 | 62 | 179 |
| | *Common* | 368 | 111 | 52 | 62 | 179 |
| s3330 | $\mu(s)$ | 0.7 | 0.68 | 0.68 | 0.63 | 0.54 |
| | **Time** | 23695 | 30342 | 20533 | 13194 | 6644 |
| | *Common* | 1900 | 5632 | 3776 | 10634 | 6644 |
| s5378 | $\mu(s)$ | 0.7 | 0.67 | 0.64 | 0.62 | 0.6 |
| | **Time** | 44701 | 76650 | 43803 | 20253 | 18493 |
| | *Common* | 2750 | 4691 | 5573 | 9846 | 18493 |
| s9234 | $\mu(s)$ | 0.67 | 0.61 | 0.61 | 0.59 | 0.55 |
| | **Time** | 125311 | 152424 | 71751 | 61864 | 39250 |
| | *Common* | 5774 | 19498 | 13000 | 14000 | 39250 |

Figure 7 depicts a comparison between parallel StocE and SimE implementations using the s9234 ISCAS-89 benchmark. The focus is on the speed-ups for the best fitness values achieved by the two algorithms. 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 given by $S(1, p) = \frac{t_1}{t_p}$. The speedups have been calculated using the best sequential time available, which is that of sequential StocE. As can be seen in Figure 7, StocE rows division outperforms the SimE rows division by achieving the target solution quality of 0.65 in 616 seconds with 5 processors, while SimE achieves a far lesser quality of 0.55 in 39250 seconds with the same number of processors.

The results obtained with parallel StocE and SimE using domain decomposition strategy can be analyzed from the aspects of algorithm's intelligence and workload division. A parallel strategy may effect the metaheuristic's 'decision variables', as in case of domain decomposition, and

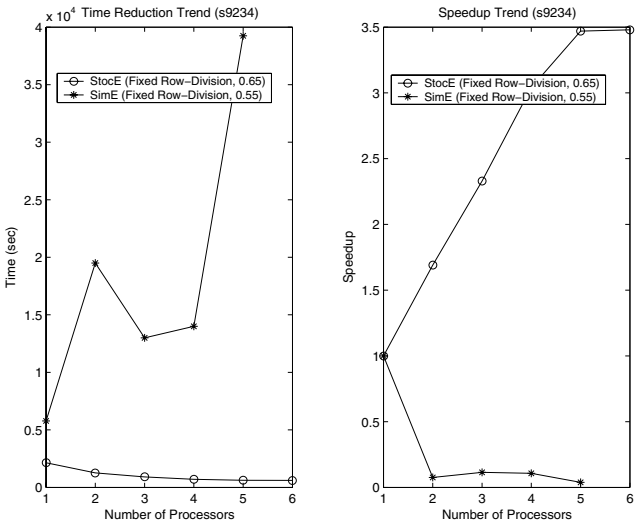


**Figure 7. StocE vs SimE. The left and right figures show the average run-times trend and average speedup, respectively.**

this change can either constrain the search or enhance it. In this respect, if a parallel strategy constrains or at best maintains the sequential algorithm's search behavior, the only way to achieve any speed-up is through effective workload division. In case of strategies that enhance the search behavior, speed-ups are possible without workload division, while workload division can lead to further speed-ups in this case.

In case of domain decomposition based parallel StocE and SimE, there is a significant workload division by dividing the solution among multiple processors because of the parallelization of the perturbation functions in both the cases. However, the consequence of dividing the solution is that each processor only has a limited freedom of cell movement, which reduces even further with increasing number of processors on a given number of total rows. This affects the optimum cell movement, making it more difficult for cells to reach their optimal locations in the same number of iterations as the sequential algorithm. Also, some error in optimum cell position determination is introduced as each processor considers the cells outside its partition as not changing positions. Owing to the largely stochastic nature of *PERTURB* operation, the solution distribution does not negatively effect the algorithm's intelligence. However, in case of SimE, the *Selection* and *Allocation* of elements is more of a deterministic process rather than stochastic as it is determined by the *goodness* values of each element. In SimE, the parallelization of *Selection* and *Allocation* operators constrains the algorithm's intelligence, resulting in lower than sequential algorithm solution qualities with parallelization and a degradation of qualities with increasing the subdivisions (using more processors).

## 6. Conclusions

This paper discussed parallelization of Stochastic Evolution applied to a multiobjective VLSI cell placement optimization problem. A comprehensive set of parallel models were considered and the strategies were compared with parallel Simulated Evolution applied to the same optimization problem. It was found that a low-level parallelization was not applicable because of the structure of optimization cost functions. Also, a parallel search strategy was not found useful for StocE parallelization because of nature of StocE heuristic. The best results were obtained with a domain decomposition approach using rows division, and furthermore, these results far exceeded the best results obtained using a similar parallel SimE approach. The strategy was compared based on two underlying principles of workload division and interaction of parallelization strategy with a heuristic's search intelligence, discussing why parallel StocE achieved an effective parallelization compared to parallel SimE for the same optimization problem.

Acknowledgments: The authors would like to thank King Fahd University of Petroleum & Minerals, Saudi Arabia, for support under project code # COE/CELL PLACE/263.

## 7. References

[1] S. G. Akl. *Parallel Computation: Models and Methods*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[2] E. Alba. *Parallel Metaheuristics: A New Class of Algorithms*. Wiley-Interscience, 2005.

[3] J. A. Chandy, S. Kim, B. Ramkumar, S. Parkes, and P. Banerjee. An Evaluation of Parallel Simulated Annealing Strategies with Application to Standard Cell Placement. *IEEE Trans on CAD of IC Systems*, 16(4):398 – 410, 1997.

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

[5] R. M. Kling and P. Banerjee. ESP: Placement by Simulated Evolution. *IEEE Trans on CAD*, 8(3):245–256, 1989.

[6] Y. G. Saab and V. B. Rao. Combinatorial Optimization by Stochastic Evolution. *IEEE Trans on CAD of IC Systems*, 10(4):525 – 535, 1991.

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

[8] S. M. Sait and H. Youssef. *Iterative Computer Algorithms with Applications in Engineering: Solving Combinatorial Optimization Problems*. IEEE Computer Society Press, 1999.

[9] S. M. Sait, H. Youssef, J. A. Khan, and A. El-Maleh. Fuzzified Iterative Algorithms for Performance Driven Low Power VLSI Placement. In *ICCD '01: Proceedings of the Intl Conf on Computer Design: VLSI in Computers & Processors*, p 484, 2001.