Journal of Universal Computer Science, vol. 14, no. 15 (2008), 2471-2490
submitted: 18/2/08, accepted: 30/6/08, appeared: 1/8/08 © J.UCS

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,miali}@kfupm.edu.sa)

Abstract: This paper discusses the parallelization of Stochastic Evolution (StocE) metaheuristic, for a distributed parallel environment. 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 (SimE). The role of the two mentioned underlying factors is discussed in parallelization of StocE.

Key Words: Parallel metaheuristics, combinatorial optimization, Stochastic Evolution, Simulated Evolution, VLSI cell placement, Cluster computing
Category: F.1.2, I.2.8, I.2.11

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) [Saab and Rao 1991] and Simulated Evolution (SimE) [Kling and Banerjee 1989] 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) [Sait and Youssef 1999]. 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 [Sait et al. 2001].

Parallelization of metaheuristics aims to solve complex problems and traverse larger search spaces in a reasonable amount of time [Crainic and Toulouse 2003, Alba 2005]. 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. 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, and indirectly the runtime due to its effect on algorithm's convergence. Parallelization of metaheuristics is an actively researched topic [Crainic and Toulouse 2003, Alba 2005]. 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 strategies [Crainic and Toulouse 2003]. The approaches are also compared with parallel SimE [Sait et al. 2007] due to the similarities between the two heuristics to evaluate the effectiveness of StocE parallelization. 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 related work, proposed parallel strategies, experimental results and comparison. Section 6 concludes the paper.

2 Optimization Problem and Cost Functions

In this section, the optimization problem is formulated along with the cost functions and constraint used in the optimization process.

2.1 Problem Formulation

This work addresses the problem of VLSI standard cell placement with the objectives of optimizing power consumption, timing performance (delay), and wirelength while considering layout width as a constraint. Semi-formally, the problem can be stated as follows:

A set of cells or modules $M = \{m_1, m_2, ..., m_n\}$ *and a set of signals* $S = \{s_1, s_2, ..., s_k\}$ *is given. Moreover, a set of signals* S_{m_i}*, where* $S_{m_i} \subseteq S$*, is associated with each module* $m_i \in M$*. Similarly, a set of modules* M_{s_j}*, where* $M_{s_j} = \{m_i | s_j \in S_{m_i}\}$ *is called a signal net, is associated with each signal* $s_j \in S$*. Also, a set of locations* $L = \{L_1, L_2, ..., L_p\}$*, where* $p \geq n$ *is given. The problem is to assign each* $m_i \in M$ *to a unique location* L_j*, such that all of the considered objectives are optimized subject to the constraints* [Sait and Youssef 2001].

Following is the description of the cost functions modeling used for estimating the values of three objectives and the constraint as stated above.

2.2 Wirelength Cost:

A Steiner tree approximation, which is fast and fairly accurate in estimating the wire length is adopted [Sait et al. 1999]. To estimate the length of net using this method, a bounding box, which is the smallest rectangle bounding the net, is found for each net. The average vertical distance Y and horizontal distance X of all cells in the net are computed from the origin which is the lower left corner of the bounding box of the net. A central point (X, Y) is determined at the computed average distances. If X is greater than Y then the vertical line crossing the central point is considered as the bisecting line. Otherwise, the horizontal line is considered as the bisecting line. Steiner tree approximation of a net is the length of the bisecting line added to the summation of perpendicular distances to it from all cells belonging to the net. Steiner tree approximation is computed for each net and the summation of all Steiner trees is considered as the interconnection length of the proposed solution.

$$X = \frac{\sum_{i=1}^{n} x_i}{n} \qquad Y = \frac{\sum_{i=1}^{n} y_i}{n} \tag{1}$$

where n is the number of cells contributing to the current net.

$$Steiner\ Tree = B + \sum_{j=1}^{k} P_j \tag{2}$$

where B is the length of the bisecting line, k is the number of cells contributing to the net and P_j is the perpendicular distance from cell j to the bisecting line.

$$Interconnection\ Length = \sum_{l=1}^{m} Steiner\ Tree_l \tag{3}$$

where m is the number of nets.

2.3 Power Cost:

In VLSI circuits with well designed logic gates, the dynamic power consumption contributes the 90% to the total power consumption [Devadas and Malik 1995, Chandrakasan et al. 1992]. Minimizing the dynamic power consumption is among the objectives as mentioned before. Power consumption p_i of a net i in a circuit can be given as:

$$p_i \simeq \frac{1}{2} \cdot C_i \cdot V_{DD}^2 \cdot f \cdot S_i \cdot \beta \tag{4}$$

where C_i is total capacitance of net i, V_{DD} is the supply voltage, f is the clock frequency, S_i is the switching probability of net i, and β is a technology dependent constant.

Assuming a fix supply voltage and clock frequency, then power dissipation of a cell depends on its capacitance and its switching probability. Hence, the above equation reduces to the following:

$$p_i \simeq C_i \cdot S_i \tag{5}$$

The capacitance C_i of cell i is given as:

$$C_i = C_i^r + \sum_{j \in M_i} C_j^g \tag{6}$$

where C_j^g is the input capacitance of gate j and C_i^r is the interconnect capacitance at the output node of cell i.

At the placement phase, only the interconnect capacitance C_i^r can be manipulated while C_j^g comes from the properties of the cell from the library used and is thus independent of placement. Moreover, C_i^r depends on wirelength of net i, so Equation 5 can be written as:

$$p_i \simeq l_i \cdot S_i \tag{7}$$

The cost function for estimate of total power consumption in the circuit can be given as:

$$Cost_{power} = \sum_{i \in M} p_i = \sum_{i \in M} (l_i \cdot S_i) \tag{8}$$

2.4 Delay Cost:

A digital circuit comprises a collection of paths. A path is a sequence of nets and blocks from a source to a sink. A source can be an input pad or a memory cell output, and a sink can be an output pad or a memory cell input. The longest path (*critical path*) is the dominant factor in deciding the clock frequency of the circuit. A critical path makes a problem in the design if it has a delay that is larger than the largest allowed delay (period) according to the clock frequency. Thus, 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:

$$T_\pi = \sum_{i=1}^{k-1} (CD_i + ID_i) \tag{9}$$

where CD_i is the switching delay of the cell driving net vi and ID_i is the interconnect delay of net vi. The overall circuit delay is equal to T_{π_c}, where π_c is the longest path in the layout (most critical path). The placement phase affects ID_i because CD_i is technology dependent parameter and is independent of placement. Using the RC delay model, ID_i is given as:

$$ID_i = (LF_i + R_i^r) \times C_i \tag{10}$$

where LF_i is load factor of the driving block, that is independent of layout, R_i^r is the interconnect resistance of net v_i and C_i is the load capacitance of cell i given in Equation 6 .

The delay cost function can be written as:

$$Cost_{delay} = max\{T_\pi\} \tag{11}$$

2.5 Width Cost:

Width cost is given by the maximum of all the row widths in the layout. The layout width is constrained not to exceed a certain positive ratio α to the average row width w_{avg}, where w_{avg} is the minimum possible layout width obtained by dividing the total width of all the cells in the layout by the number of rows in the layout. Formally, width constraint can be expressed as below:

$$Width - w_{avg} \leq \alpha \times w_{avg} \tag{12}$$

2.6 Overall Fuzzy Cost Function:

Since three objectives are being optimized simultaneously, there should be a cost function that represents the effect of all three objectives in form of a single quantity. In this work, the use of fuzzy logic is proposed to integrate these multiple, possibly conflicting objectives into a scalar cost function. Fuzzy logic allows us to describe the objectives in terms of linguistic variables. Then, fuzzy rules are used to find the overall cost of a placement solution. The following fuzzy rule has been used:

IF a solution has *SMALL wirelength* **AND** *LOW power consumption* **AND** *SHORT delay* **THEN** it is an *GOOD* solution.

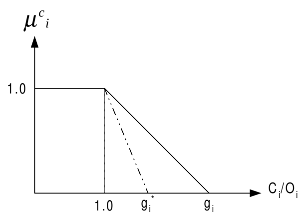

Figure 1: Membership functions

The above rule is translated to *and-like* OWA fuzzy operator [Yager 1988] and the membership $\mu(x)$ of a solution x in fuzzy set *GOOD solution* is given as:

$$\mu(x) = \begin{cases} \beta \cdot \min_{j=p,d,l} \{\mu_j(x)\} + (1-\beta) \cdot \frac{1}{3} \sum_{j=p,d,l} \mu_j(x); \\ \qquad\qquad \text{if } \; Width - w_{avg} \leq \alpha \cdot w_{avg}, \\ 0; \qquad\qquad \text{otherwise.} \end{cases} \tag{13}$$

Here $\mu_j(x)$ for $j = p, d, l, width$ are the membership values in the fuzzy sets *LOW power consumption, SHORT delay*, and *SMALL wirelength* respectively. β is the constant in the range $[0, 1]$. The solution that results in maximum value of $\mu(x)$ is reported as the best solution found by the search heuristic.

The membership functions for fuzzy sets *LOW power consumption, SHORT delay*, and *SMALL wirelength* are shown in Figure 1. The preference of an objective j in overall membership function can be varied by changing the value of g_j . The lower bounds O_j for different objectives are computed as given in Equations 14-17:

$$O_l = \sum_{i=1}^{n} l_i^* \quad \forall v_i \; \in \; \{v_1, v_2, ..., v_n\} \tag{14}$$

$$O_p = \sum_{i=1}^{n} S_i l_i^* \quad \forall v_i \; \in \; \{v_1, v_2, ..., v_n\} \tag{15}$$

$$O_d = \sum_{j=1}^{k} CD_j + ID_j^* \qquad \forall v_j \; \in \; \{v_1, v_2, ..., v_k\} \; in \; path \; \pi_c \tag{16}$$

$$O_{width} = \frac{\sum_{i=1}^{n} Width_i}{\# \; of \; rows \; in \; layout} \tag{17}$$

where O_j for $j \in \{l, p, d, width\}$ are the optimal costs for wire-length, power, delay and layout width respectively, n is the number of nets in layout, l_i^* is the optimal wire-length of net v_i, CD_i is the switching delay of the cell i driving net v_i, ID_i is the optimal interconnect delay of net v_i calculated with the help of l_i, S_i is the switching probability of net v_i, π_c is the most critical path with respect to optimal interconnect delays, k is the number of nets in π_c and $Width_i$ is the width of the individual cell driving net v_i.

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 Ω is the state space. A general outline of the StocE algorithm is given in Figure 2 for a minimization problem. The inputs to the StocE algorithm are,

```
ALGORITHM Stochatic_Evolution(S0, p0, R);
Begin
    BestS= S = S0;
    BestCost= CurCost= Cost(S);
    p = p0;
    ρ = 0;
    Repeat
        PrevCost= CurCost;
    /* perform a search in the neighborhood of S */
        S = PERTURB(S, p);
        CurCost= Cost(S);
    /* update p if needed */
        UPDATE(p, PrevCost, CurCost);
        If (CurCost< BestCost) Then
            BestS=S;
            BestCost= CurCost;
    /* Reward the search with R more generations */
            ρ = ρ − R;
        Else
            ρ = ρ + 1;
        EndIf
    Until ρ > R
    Return (BestS);
End
```

Figure 2: The StocE algorithm.

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 [Saab and Rao 1991]. 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 (Figure 3) 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 ∈ 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;
/* make sure S satisfies constraints */
    S =MAKE_STATE(S);
    Return (S)
End
```

Figure 3: The StocE 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 4) 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_incr;
/* increment p to allow larger uphill moves */
    Else
        p = p_0; /* re-initialize p */
    EndIf;
End
```

Figure 4: The StocE UPDATE procedure.

```
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 DO evaluate g_i;
      SELECTION: ForEach m_i ∈ Φ DO
                 IF Random > Min(g_i + B, 1) THEN
                        S = S ∪ m_i; Remove m_i from Φ
                 Sort the elements of S
      ALLOCATION: ForEach m_i ∈ S DO ALLOCATE(m_i, Φ_i)
  Until   Stopping Condition is satisfied
  Return Best solution.
End
```

Figure 5: The SimE algorithm.

3.2 Simulated Evolution (SimE)

The structure of the SimE algorithm is shown in Figure 5. SimE assumes that there exists a solution Φ 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 Φ. 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 fuzzy goodness measure developed in [Sait et al. 2001] is used.

The second step of the SimE algorithm is *Selection*. *Selection* takes as input the solution Φ 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 Φ into two disjoint sets; a selection set S and a partial solution Φ_p of the remaining elements of the solution Φ. 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 [Sait et al. 2001] 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 Φ_p and generates a new complete solution Φ' with the elements of set S mutated according to an allocation function *Allocation* [Sait and Youssef 1999]. 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 [Kling and Banerjee 1989]. In this work, the *sorted individual best fit* method [Sait et al. 2001] has been used.

4 Sequential Algorithms' Profiling

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. Note that these computations are part of StocE *PURTURB* routine (Section 3.1). For the sequential SimE, on average 98.85% of time is spent in the *Allocation* function.

Table 1: Sequential algorithms' runtime profile.

Circuit	# of Cells	# of Rows	StocE Cost Functions	StocE Others	SimE Allocation Function	SimE 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 Strategies and Experiments

The field of parallel metaheuristics has rapidly expanded in the past ten to fifteen years and parallel versions of metaheuristics have been increasingly proposed. Several excellent surveys, taxonomies and syntheses have also been published [Crainic and Toulouse 2003, Alba 2005], which present a global view of the field and generalize the various strategies used into broad classes. The various approaches can be classified into three comprehensive types according to the source of parallelism [Crainic and Toulouse 2003]. These are:

1. Type I (Low-Level Parallelization): The limited functional or data parallelism of a move evaluation is exploited or moves are evaluated in parallel. This strategy, called low-level parallelism, aims to simply speed-up the sequential algorithm without changing the search space traversal path taken by the algorithm.

2. Type II (Domain Decomposition): This approach obtains parallelism by partitioning the set of decision variables. The partitioning reduces the size of solution space, but it needs to be repeated to allow the exploration of the complete solution space. The traversal is different than the sequential algorithm.

3. Type III (Parallel Searches): Parallelism is obtained from multiple concurrent explorations of the solution space.

This work explores StocE parallelization considering the complete spectrum of parallelization types discussed here. The effectiveness of a parallel strategy is analyzed in terms of workload division and its effect on metaheuristic's intelligence.

Following is the description of the parallel strategies and their experimental results. All parallel programs were written in C using the MPI library (MPICH 1.2.5). A dedicated cluster of 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. In all the results tables, runtimes are in seconds and the solution qualities, denoted by $\mu(s)$, is the fuzzy cost measure (Section 2).

5.1 Low-Level Parallelization

Given the StocE profiling results, parallelizing cost functions to achieve work load division may seem intuitive. However, with fine grained dependencies in cost computation/perturbation functions coupled with a high node-to-node communication cost, this strategy is not well suited to the given parallel environment. This was confirmed by the results obtained when this strategy was applied to parallelize SimE for the same problem [Sait et al. 2007].

5.2 Parallel Searches

In cooperative parallel searches approach, parallel threads each running a complete StocE/SimE process cooperate with each other (by exchanging good solutions) to quickly converge. This strategy exploits the capability of multiple, concurrent threads to cooperatively navigate the search space. 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 Asynchronous Multiple Markov Chains (AMMC) SA [Sait et al. 2006].

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 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 6 and Figure 7. The slave processor upon reaching a better solution sends the cost metric to the master node. The master compares this received metric with the current best it has. If found better, the slave is instructed to send the entire solution; else, the master sends the better solution it has to the slave.

Table 2 shows the performance of the AMMC approach. It can be seen that the parallel algorithm achieves no gain beyond using 2 processors. The reason for

```
ALGORITHM TypeIII_Parallel_StocE_Master_Process
Begin
    Read_User_Input_Parameters;
    Read_Input_Files;
    Construct_Initial_Placement;
/* Only master has the initial Solution */
    CurS = S0;
    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);
End (*Master_Process*)
```

Figure 6: Master Process for Parallel AMMC StocE Algorithm.

```
ALGORITHM TypeIII_Parallel_StocE_Slave_Process
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;
        /* Reward the search with R more generations */
            ρ = ρ − R;
        Else
            ρ = ρ + 1;
        EndIf
        Send_to_Master(BestCost);
        Receive_frm_Master(verdict);
        If (verdict == 1)
            Send_to_Master(BestS);
        Else
            Receive_frm_Master(BestS);
        EndIf
    Until ρ > R
    Return (BestS);
End
```

Figure 7: Slave Process for Parallel AMMC StocE Algorithm.

this limited performance 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 [Sait et al. 2007].

Table 2: Results for Parallel AMMC StocE.

Circuit Name	Number of Cells	$\mu(s)$ StocE	Time for Sequential StocE	Time for Parallel StocE			
				p=2	p=3	p=4	p=5
s1494	661	0.6	94	32.78	32.72	32.73	32.79
s3330	1961	0.6	186	96.92	95.87	89.07	92.66
s5378	2993	0.6	479.93	268.98	270.78	265.89	270.59
s9234	5844	0.6	1143	799.36	802.63	800.83	799.42

5.3 Domain Decomposition

Domain decomposition parallelization divides the solution into independent domains, each to be operated in parallel [Crainic and Toulouse 2003]. 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. 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 8 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.

Figures 9 and 10 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 ρ or decrements it by R. This new solution is then again broadcasted to all the slaves. This process continues till the target fitness value is achieved or termination criteria

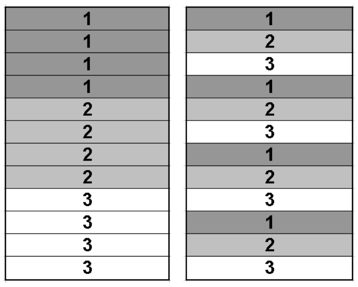

Figure 8: Rows distribution in even and odd numbered iterations.

is met. It should be noted that the search behavior of the parallel algorithm will differ from the serial algorithm owing to this partitioning.

```
ALGORITHM TypeII_Parallel_StocE_Master_Process
Begin
    Read_User_Input_Parameters;
    Read_Input_Files;
    Construct_Initial_Placement;
    Repeat
        /* Broadcast current placement */
        ParFor
            Slave_Process(S, p);
        EndParFor
        /* For each slave process */
        ParFor
            Receive_Partial_Solutions;
        EndParFor
        S = Make_Complete_Solution;
        CurCost= Cost(S);
        /* Update p if needed */
        UPDATE(p, PrevCost, CurCost);
        If (CurCost< BestCost) Then
            BestS=S;
            BestCost= CurCost;
        /* Reward the search with R more generations */
            ρ = ρ − R;
        Else
            ρ = ρ + 1;
        EndIf
    Until ρ > R
    Return (Best_Solution)
End. (*Master_Process*)
```

Figure 9: Outline of master process for rows division parallel StocE.

Similar to the parallel StocE, for domain decomposition parallel SimE, the elements are partitioned row wise among the m processors. A processor s, $1 \leq s \leq m$ would be assigned a subset Φ^s of the solution Φ. Then, each processor s will evaluate the goodness of each element in Φ^s and run the *Selection* step to partition Φ^s into a selection subset S^s and a partial solution of remaining cells

${\Phi_p}^s$ (See the serial algorithm in Figure 5 for comparison).

```
ALGORITHM TypeII_Parallel_StocE_Slave_Process(S, p)
Begin
    Receive_Placement;
    /* perform a search in the restricted neighborhood of S */
    S = PERTURB(S, p);
    Send_Partial_Solution(S);
End. (*Slave_Pocess*)
```

Figure 10: Outline of slave process for rows division parallel StocE.

The results of rows division strategy for StocE and SimE are given in Table 3 and Table 4, respectively. Up to 5 processors are used as no significant gains are observed beyond this number due to the size of benchmarks. The $\mu(s)$ values represent the highest solution quality achieved by sequential algorithm. The results shown are the average of 10 runs per set of processors. For circuits s3330, s5378 and s9234 the maximum standard deviation was 25 seconds. Due to the relatively small size of benchmark circuit s1494, no gains are observed beyond 2 processors. Also the standard deviations for this circuit were high.

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 4. 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 with increase in processors.

Table 3: 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

Figure 11 depicts a comparison between parallel StocE and SimE implemen-

Table 4: Results of rows division parallel SimE.

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

tations using the s9234 ISCAS-89 benchmark. s9234 was the largest common benchmark among the two algorithms and was thus selected for comparison. As can be observed in Table 3, sequential StocE achieves much higher fitness values when compared to sequential SimE. This is due to the difference in stochastic natures and evolutionary search approaches employed by the two algorithms. It is important to mention here that, for comparison, restricting StocE to target low fitness values, as achieved by SimE, resulted in poor speed-up trends by StocE. This is due to the fact that parallel StocE employing just 2 processors achieved the attempted low fitness values quite early in its search process while exploiting the minimum advantage of parallelization. Therefore, increasing the number of processors did not result in any further time reduction.

Thus, the focus is on the speed-ups for the fitness values achieved by both the algorithms when they are close to the steady state and the variance in results is minimal. This comparison highlights the two important points. It shows that StocE achieves better solution qualities than SimE as well as it achieves these high qualities in runtimes quite lower than what SimE requires for achieving low quality solutions.

The speed-up is defined as follows [Akl 1997]: 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 11, 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.

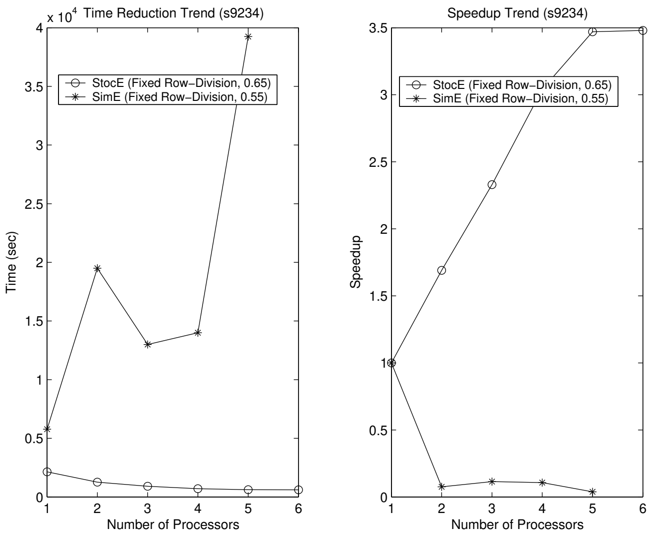

Figure 11: StocE Vs SimE. The left and right figures show the average run-times trend and average speedup, respectively.

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 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 in StocE, 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 VLSI cell placement optimization problem. A comprehensive set of parallel strategies were considered and these 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.

Acknowledgment

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

References

[Akl 1997] Akl, S. G. :"Parallel Computation: Models and Methods"; Prentice-Hall, Inc., New Jersey (1997).

[Alba 2005] Alba, E.:"Parallel Metaheuristics: A New Class of Algorithms"; Wiley-Interscience, (2005).

[Chandrakasan et al. 1992] A. Chandrakasan and T. Sheng and R. W. Brodersen :"Low Power CMOS Digital Design"; Journal of Solid State Circuits,(1992)

[Crainic and Toulouse 2003] Crainic, T. G., Toulouse, M.:"Parallel Strategies for Metaheuristics in Handbook of Metaheuristics"; Springer, (2003).

[Devadas and Malik 1995] Srinivas Devadas and Sharad Malik :"A Survey of Optimization Techniques Targeting Low Power VLSI Circuits"; 32nd ACM/IEEE DAC,(1995)

[Kling and Banerjee 1989] Kling, R. M., Banerjee, P.: "ESP: Placement by Simulated Evolution"; IEEE TCAD, 8, 3 (Mar 1989), 245-256.

[Saab and Rao 1991] Saab, Y. G., Rao, V. B.: "Combinatorial Optimization by Stochastic Evolution"; IEEE TCAD, 10, 4 (Apr 199l), 525-535.

[Sait and Youssef 1999] Sait, S. M., Youssef, H. :"Iterative Computer Algorithms with Applications in Engineering: Solving Combinatorial Optimization Problems"; IEEE Computer Society Press, California (1999).

[Sait and Youssef 2001] Sait, S. M. and Youssef, H. :"VLSI Physical Design Automation: Theory and Practice"; World Scientific Pubishers, (2001)

[Sait et al. 1999] Sadiq M. Sait and H. Youssef and Ali Hussain :"Fuzzy Simulated Evolution Algorithm for Multiobjective Optimization of VLSI Placement"; IEEE Congress on Evolutionary Computation,(1999), 91-97

[Sait et al. 2001] Sait, S. M., Youssef, H., Khan, J. A., El-Maleh, A.: "Fuzzified Iterative Algorithms for Performance Driven Low Power VLSI Placement"; Proc. ICCD'01, IEEE Comp. Society, Washington DC (2001), 484-487

[Sait et al. 2006] Sait, S. M., Zaidi, A. M., Ali, M. I. :"Asynchronous MMC Based Parallel SA Schemes for Multiobjective Standard Cell Placement"; Proc. ISCAS'06, IEEE (2006)

[Sait et al. 2007] Sait, S. M., Ali, M. I., Zaidi, A. M. :"Evaluating Parallel Simulated Evolution Strategies for VLSI Cell Placement"; Springer JMMA, 6, 3 (Sept 2007), 433-454.

[Yager 1988] Yager, R. R.:"On Ordered Weighted Averaging Aggregation Operators in Multicriteria Decisionmaking"; IEEE Transaction on Systems, MAN, and Cybernetics, (1988), 183-190