Evaluating Parallel Simulated Evolution Strategies for VLSI Cell Placement

SADIQ M. SAIT, MUSTAFA IMRAN ALI and ALI MUSTAFA ZAIDI
Computer Engineering Department, King Fahd University of Petroleum & Minerals, Dhahran-31261, Saudi Arabia. email: {sadiq, mustafa, alizaidi}@ccse.kfupm.edu.sa

Abstract. Simulated Evolution (SimE) is an evolutionary metaheuristic that has produced results comparable to well established stochastic heuristics such as SA, TS and GA, with shorter runtimes. However, for optimization problems with a very large set of elements, such as in VLSI cell placement and routing, runtimes can still be very large and parallelization is an attractive option for reducing runtimes. Compared to other metaheuristics, parallelization of SimE has not been extensively explored. This paper presents a comprehensive set of parallelization approaches for SimE when applied to multiobjective VLSI cell placement problem. Each of these approaches are evaluated with respect to SimE characteristics and the constraints imposed by the problem instance. Conclusions drawn can be extended to parallelization of SimE when applied to other optimization problems.

Keywords: optimization, parallel algorithms, evolutionary metaheuristic, simulated evolution, VLSI cell placement, cluster computing

1. Introduction

Simulated evolution (SimE), proposed by Kling and Banerjee [1], belongs to the class of general purpose stochastic metaheuristics. It has been applied to a variety of optimization problems in VLSI design automation, computer network design, and other domains [2]. The SimE algorithm is based on the principles of evolution. However, unlike Genetic Algorithms (GA), only a single solution is evolved instead of a population of solutions. Also, unlike Simulated Annealing (SA) and Tabu Search (TS), each move in SimE is a compound move and the element(s) perturbed are selected probabilistically based on their fitness values and not entirely randomly.

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 [4, 5, 6]. From a computational point of view, metaheuristics are algorithms from which functional and data parallelism can be extracted. However, metaheuristics usually operate

upon irregular data structures, such as graphs, or upon data with strong dependencies among different operations and as such remain difficult to parallelize using only data and functional parallelism [4]. Furthermore, 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 parallelizability of both the cost computation and perturbation functions while the latter issue requires that 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.

In this paper the parallelization of SimE is explored when it is applied to a multiobjective VLSI cell placement problem with the goal of achieving scalable speed-ups for the best solution qualities obtained with the serial algorithm. To this end, various parallelization approaches are investigated and a comparison amongst them with respect to SimE metaheuristic characteristics and problem instance interaction is presented. The paper is organized as follows: Section 2 explains the combinatorial optimization problem. In section 3.1, a brief description of the SimE algorithm is given and an analysis of sequential implementation's runtime is given. In Section 4, the work is put in context of previous work while Section 5 describes parallel strategies and experimental results. General observations are given in Section 6, and Section 7 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 [3].

2.2. Wirelength Cost:

Interconnect Wire length of each net in the circuit is estimated and then total wire length is computed by adding the individual estimates:

$$Cost_{wire} = \sum_{i \in M} l_i \tag{1}$$

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

2.3. Power Cost:

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

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

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:

$$p_i \simeq C_i \cdot S_i \tag{3}$$

The capacitance C_i of cell i is given as:

$$C_i = C_i^r + \sum_{j \in M_i} C_j^g \tag{4}$$

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 3 can be written as:

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

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

$$Cost_{power} = \sum_{i \in M} p_i = \sum_{i \in M} (l_i \cdot S_i) \tag{6}$$

2.4. 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:

$$T_\pi = \sum_{i=1}^{k-1} (CD_i + ID_i) \tag{7}$$

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:

$$ID_i = (LF_i + R_i^r) \times C_i \tag{8}$$

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 4 .

The delay cost function can be written as:

$$Cost_{delay} = max\{T_\pi\} \tag{9}$$

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:

$$Width - w_{avg} \leq \alpha \times w_{avg} \tag{10}$$

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.

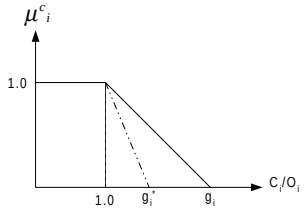

Figure 1. Membership functions

The above rule is translated to *and-like* OWA fuzzy operator [9] and the membership $\mu(x)$ of a solution x in fuzzy set *GOOD solution* is given as:

$$\mu(x) = \begin{cases} \beta \cdot \min\limits_{j=p,d,l} \{\mu_j(x)\} + (1-\beta) \cdot \frac{1}{3} \sum\limits_{j=p,d,l} \mu_j(x); \\ \qquad\qquad \text{if} \quad Width - w_{avg} \leq \alpha \cdot w_{avg}, \\ 0; \qquad\quad \text{otherwise.} \end{cases} \tag{11}$$

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 12-15:

$$O_l = \sum_{i=1}^{n} l_i^* \quad \forall v_i \in \{v_1, v_2, ..., v_n\} \tag{12}$$

$$O_p = \sum_{i=1}^{n} S_i l_i^* \quad \forall v_i \ \in \ \{v_1, v_2, ..., v_n\} \tag{13}$$

$$O_d = \sum_{j=1}^{k} CD_j + ID_j^* \qquad \forall v_j \ \in \ \{v_1, v_2, ..., v_k\} \ in \ path \ \pi_c \tag{14}$$

$$O_{width} = \frac{\sum_{i=1}^{n} Width_i}{\#\ of\ rows\ in\ layout} \tag{15}$$

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. Simulated Evolution Algorithm

3.1. Description of Metaheuristic

The structure of the SimE algorithm is shown in Figure 2. 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 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 [7] 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 an earlier work [7] as 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* [2]. 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 [1]. In this work, the 'sorted individual best fit method' [7] has been used.

3.2. Runtime Analysis of Sequential Algorithm

To determine the contribution of each of the cost functions and SimE operators to overall execution time, the serial implementation was profiled using gprof (GNU profiler) tool. Two separate versions of programs were analyzed for various test cases executed for same number of iterations. Of the two versions, the first optimized only wirelength and power while the other focused on all three objectives. The results obtained showed that for first and second versions respectively 98.4% and 98.5% of time was spent in the allocation function, 0.6% and 0.5% of time was spent in wirelength calculation (excluding wirelength re-calculation calls made in allocation routine), 0.2% and 0.4% of time was spent in goodness evaluation, and 0.2% of time was spent in delay calculation in the second version. Thus, it is obvious that for the given problem instance with the 'sorted individual best-fit' method, allocation routine heavily influences the runtime of the algorithm. The impact of this is discussed in Section 5.

ALGORITHM $Simulated\_Evolution(B, \Phi_{initial})$
NOTATION
B: Bias Value. Φ: Complete solution.
m_i: Module i. g_i: Goodness of m_i.
$ALLOCATE(m_i, \Phi_i)$: Allocates m_i in partial solution Φ_i
Begin
INITIALIZATION;
Repeat
EVALUATION:
ForEach $m_i \in \Phi$ evaluate g_i;
SELECTION:
ForEach $m_i \in \Phi$ **DO**
begin
IF $Random > Min(g_i + B, 1)$
THEN
begin
$S = S \cup m_i$; Remove m_i from Φ
end
end
Sort the elements of S
ALLOCATION:
ForEach $m_i \in S$ **DO**
begin
$ALLOCATE(m_i, \Phi_i)$
end
Until *Stopping Condition is satisfied*
Return Best solution.
End ($Simulated\_Evolution$)

Figure 2. Simulated evolution algorithm.

4. Related Work

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 [4, 5, 6], which present a global view of the field and generalize the various strategies used into broad classes. To put the exploration approach taken in this work into context, the parallel approaches attempted for SA, GA and TS are briefly discussed.

4.1. Simulated Annealing

4.1.1. *Move Acceleration*

Several efforts to determine and exploit parallelism have focused on move computation, as this is a fundamental component performed numerous times during each annealing run. The underlying idea is to partition different, non-interacting portions of the move evaluation task across several processors in parallel. Each individual move is evaluated faster by breaking up the overall task into subtasks such as selecting a feasible move, evaluating the cost changes, deciding to accept or reject, and perhaps updating a global database. Concurrency is obtained by delegating these individual subtasks to different processors.

Such a strategy, referred to as *move-acceleration* or *move-decomposition*, involves a close interaction between processors, and has less potential for parallelism in terms of the amount of parallel work performed and the number of processors that can be employed. Such methodologies are largely restricted to shared memory architectures and preserve all the properties of the serial algorithm [8].

4.1.2. *Parallel Moves*

In this method, moves are computed independently and in parallel by several processors. Since the global system state is partitioned across the processors, the independent computation and subsequent state update of interacting moves causes the locally held view of the global system state in each processor to become inconsistent with the local views in other processors. Consequently, errors are introduced in move evaluation. The impact of such errors may be kept at a minimum through frequent exchanges of state-update information between processors. However, this approach implies significantly increased inter-processor communication, thereby restricting its application in a cluster-of-workstations environment.

4.1.3. *Multiple Markov Chains*

Multiple Markov chains (MMC) call for the concurrent execution of separate simulated annealing chains with periodic exchange of solutions [20]. This approach is particularly promising since it has the potential to use parallelism to increase the quality of the solution.

4.1.3.1. *Non-interacting Scheme* The algorithm can be understood if the sequential simulated annealing procedure is considered as a search path where moves are proposed and either accepted or rejected depending on particular cost evaluations and also a starting random seed. The search path is essentially a Markov chain, and parallelization is

accomplished by initiating different chains (using different seeds) on each processor. Each chain then explores the entire search space by independently performing the perturbation, evaluation, and decision steps. After each processor has completed the annealing schedule, the solutions are compared and the best is selected.

4.1.3.2. *Periodic Exchange Scheme (Synchronous MMC)* In this scheme, processing elements (PEs) exchange local information including the intermediate solutions and their costs after a fixed time period. Then, each PE restarts from the best of the intermediate solutions. Compared to the non-interacting scheme, a communication overhead in this periodic exchange scheme would be introduced. However, each PE can utilize the information from other nodes thereby reducing unproductive computations and idle time. With such communication, these independent Multiple Markov chains can collectively converge to a better solution.

4.1.3.3. *Dynamic Exchange Scheme (Asynchronous MMC)* The statistical data collected during execution may be utilized to adaptively control the SA process in each Markov Chain to further reduce the execution time. For example, the acceptance rate which is closely related to the annealing state can control communication instances. The periodic exchanges that were discussed earlier may introduce unnecessary and untimely communication, thereby wasting time. Moreover, an intermediate solution derived at an insufficiently cooled state can hamper the convergence of other communicating Markov chains.

Soo-Young and Kyung proposed an asynchronous MMC model, which adaptively determines when information is to be exchanged [20]. Communication is permitted based on satisfying certain conditions. First, a certain period of time has to elapse, i.e., to allow each PE sufficient independent annealing. Second, these working nodes exchange information only when necessary, rather than at a fixed schedule, e.g., when other PEs have arrived at a significantly better solution. In this way, these processing elements can more efficiently guide each other to a higher quality solution. This is known as the dynamic exchange scheme, and is an asynchronous MMC model.

4.2. Genetic Algorithms

Over the years, parallel Genetic Algorithms have been broadly classified into the following three models [14, 15]:

1. Global single-population master-slave GAs

2. Single-population fine-grained GAs, and

3. Multiple-population coarse-grained GAs

4.2.1. *Global Single-Population Master-Slave ParGA*
This model follows the Master-Slave paradigm where a single population is maintained on the Master, while evaluation of fitness and/or the application of Genetic Operators is distributed among several slave processors. In this model, selection and crossover consider the entire population and hence it is also known as the Global Parallel GA. As in the serial GA, each solution may compete and mate with any other in the population; also the selection operation determines the new population from the complete set of older population and their offsprings.

The most common operation that is parallelized is the evaluation of solutions, as the fitness of each chromosome is independent of any other. A fraction of the population is assigned to each processor, and communication occurs only as each slave receives its subset and returns the fitness values. The method does not affect the behavior of the GA algorithm, and follows the same search pattern as serial GAs.

4.2.2. *Single-Population Fine-Grained ParGA*
Fine-Grained Parallel GAs maintain only one population, but have a spatial structure that limits interaction between individual solutions. An individual can only compete and mate with its neighbors, i.e., selection and crossover is restricted to a small neighborhood. However, these neighborhoods overlap, thus allowing good solutions to disseminate across the entire population. This model is suitable for massively parallel computers with the ideal case of having only one individual solution for each processing element available.

4.2.3. *Multiple-Population Coarse-Grained GAs*
Multiple-Population GAs provide a more sophisticated parallelization strategy wherein several subpopulations evolve independently on individual processors and exchange individuals periodically. This exchange of solutions is called *migration* and is a core aspect of this parallel model. Multi-population GAs are known with different names. They are referred to as Multi-deme parallel GAs (drawing on the analogy of natural evolution), Distributed GAs (as they are often implemented on distributed parallel architectures), and Coarse-grained GAs (since the computation to communication ratio is usually high). This model of parallel GAs is very popular, but also the most difficult to understand

due to the effect of migration and a large number of influential parameters. There is no hierarchical master-slave structure but rather, a peer model with migration between demes controlled by various parameters.

4.3. Tabu Search

Parallel TS has drawn the attention of many researchers, especially in comparison with similar acceleration strategies applied to other heuristics. The first reported studies were published in the early 90's [16, 17, 18]. Crainic et al. [19], classified the different parallel tabu search heuristics based on a taxonomy along three dimensions as enumerated below.

- The first dimension is **Control cardinality**, where the algorithm is either *1-control*, where one processor executes the search and distributes tasks to other processors or *p-control*, where each processor is responsible for its own search and communicates with other processors.

- The second dimension is **Control and communication type**, where the algorithm can either follow a *rigid synchronization (RS)* and *knowledge synchronization (KS)* approach or it can be *Collegial (C)*, and *Knowledge Collegial (KC)*. The former is a synchronous operation mode where the process is forced to establish communication and exchange information at specific, explicitly defined points. The latter is an asynchronous operation mode where the processors can independently decide on communication depending on the global characteristics of good solutions, the search strategy and the possible content of that communication

- The third dimension is **Search differentiation** where the algorithm can be *SPSS (Single Point Single Strategy), SPDS (Single Point Different Strategies), MPSS (Multiple Point Single Strategy)*, or *MPDS (Multiple Point Different Strategies).*

In addition to this type of classification, a more general category based on processor communication is also used. This divides various approaches as either *Synchronous* or *Asynchronous*. In the former, various processors working with the same solution, communicate in a synchronous manner, where the managing processor orchestrates the activities of all others. In asynchronous strategies, each processor communicates independently of other nodes using either the master-slave or peer-to-peer model.

4.3.1. *Synchronous Parallel Tabu Search*
In this approach, the master is primarily in-charge of controlling movement in the search process, while the slaves are used for distributing workload. Depending on the variants of this strategy, slave processors may start with either the same or different initial solution. After searching its allocated part of the current neighborhood, each slave process reports its best move back to the master. The master process selects the best among these, subject to tabu conditions. If the stopping criteria are met then the search stops; otherwise the master determines a new set of moves and distributes them among the slaves which continue with the search. This approach, by Crainic's classification, would be a 1-control, RS, SPSS algorithm.

4.3.2. *Asynchronous Parallel Tabu Search*
In this approach, each processor explores a subset of the neighborhood of its current solution. Each of these is competing with its neighbors (its adjacent processors) in finding a superior solution. When the stopping criteria are met, every processor reports its best solution. Similar asynchronous parallel tabu search implementations for the traveling salesman and quadratic assignment problems have been reported in [16]. Its classification is p-control, C, MPSS algorithm.

4.4. This work

In this paper follows the approach taken in [4] and classify the various attempted strategies into three comprehensive types according to the source of parallelism. These are [4]:

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.

Unlike SA, GA, TS and many other metaheuristics, parallelization of SimE has not been explored extensively and no comparison among strategies has been made. The only parallelization strategy reported [1]

was for a single objective (wirelength) VLSI cell placement that can be classified under type II. In this paper, a more complex multiobjective cost function is used and comparion of the parallel strategies is presented considering the complete spectrum of parallelization types discussed here.

5. Parallel Strategies and Experiments

The parallel SimE strategies were implemented in C along with MPICH ver.1.2.5 Messsage Passing Interface library. A dedicated cluster was used comprising of eight 2GHz Pentium-4 machines with 256MB RAM, connected with fast ethernet, and running RedHat Linux ver.7.2. The strategies were tested on ISCAS-89 benchmark circuits. They are of various sizes in terms of number of cells and paths, and thus offer a variety of test cases. In all the results tables, runtimes are in seconds and the solution qualities, denoted by $\mu(s)$, is the fuzzy cost measure discussed in Section 2.

5.1. Type I Parallelization

As stated earlier, a type I parallelization aims to speed up the sequential algorithm without modifying its search behavior. For a type I parallel SimE strategy, parallelization of goodness evaluations seems intuitive as it is done at the level of individual elements, although the dependencies among elements has to be taken into account to ensure correctness. However, the allocation routine has a sequential dependence among its operations and it cannot be partitioned without deviating from the sequential algorithm behavior. Hence, the implemented SimE type I parallelization focuses only on distribution of cost calculations and goodness evaluation.

In the multi-objective cost computation, the calculation of wirelength of each net must precede the calculation of power and delay. The wirelength calculation of each net is independent of other nets and thus can be performed in parallel. The same applies to power computations. The calculation of delay costs involves operating on given critical paths, finding the delay of each and then finding maximum delay among all paths. These can also be performed in parallel. This results in a fairly clean partitioning as long as cost computations are concerned. However, the complications lie in goodness evaluations for wirelength, power and delay.

The calculation of wirelength and power goodness values of each cell requires that the wirelength of all fan-in cells be known [7]. This complicates the partitioning of cells among processors; if a processor needs

```
ALGORITHM TypeI_Parallel_SimE_Master_Process
NOTATION
 (* Φ is the complete solution. *)
Begin
 INITIALIZATIONS;
 Repeat
    EVALUATION:
      (* For each slave process. *)
      ParFor
        Slave_ Process(Φ)
        (* Broadcast Current Placement. *)
      EndParFor
      ParFor
        Receive_Partial_Goodness_Values
      EndParFor
    SELECTION;
    Sort the elements of S;
    ALLOCATION;
 Until (Stopping Criteria is Satisfied)
 Return (Best_Solution)
End. (*Master_Process*)
```

Figure 3. Outline of Master Process for Type I Parallel SimE Algorithm.

to calculate the wirelength of cells not in its partition, the potential gain of cost computation division is reduced to the extent of duplicate calculations performed. The situation is worse for delay goodness calculations as all the cells on an assigned long path may not lie in the same assigned partition, resulting in many duplicate calculations across processors. In addition, all processors need to know the computed delay of all long paths in the circuit to calculate the delay goodness of cells in its partition, requiring additional costly communication. Furthermore, during *allocation* at the master node, additional cost calculations may be required when calculating the goodness gains for those cells which are not the members of partition at the master node.

Since delay goodness partitioning has complex communication requirements, and secondly, profiling results indicate that most of the time is spent for wirelength/power cost and goodness calculations, the type I parallel algorithm was implemented for only wirelength and power optimization to observe the results of partitioning. Figures 3 and 4 show the outline of the type I parallel SimE algorithm. The partial cost and goodness computations are carried out by all processors including the master processor, which then receives goodness values

ALGORITHM *TypeI_Parallel_SimE_Slave_Process*(Φ)
NOTATION
(* Φ is the complete solution. *)
(* $\Phi^s$is the partition assigned to slave s. *)
(* m_i is module i in Φ^s. *)
(* g_i is the goodness of m_i. *)
Begin
Receive_Placement
Calculate_Partial_Costs
ForEach $m_i \in \Phi^s$ evaluate g_i **EndForEach**;
Send_Partial_Goodness_Values
End. (*Slave_Pocess*)

Figure 4. Outline of Slave Process for Type I Parallel SimE Algorithm.

from all processors and performs selection and allocation. The slave processors are then updated with the new solution.

The results of type I implementation are shown in Table I. Due to lack of space, the solution quality for each circuit is not shown as it doesn't vary between serial and parallel versions. The results show that there is no benefit of type I parallelization because of poor workload division owing to duplicate calculations. Furthermore, there is an increase in the runtime of parallel algorithm as the parallelization overheads well exceed the little workload distribution. Also, no change in runtimes is observed with increasing processors. Interestingly, the ratio of serial to parallel runtimes remains almost the same across the different test cases and processor counts.

Table I. Results for Type 1 Parallel SimE

Ckt Name	Cells	Seq. Time	Times for Parallel			
			p=2	p=3	p=4	p=5
s1196	561	92	130	130	130	130
s1488	667	187	263	263	263	263
s1494	661	190	268	268	273	270
s1238	540	91	127	129	131	130
s3330	1561	3750	5480	5463	5467	5453

ALGORITHM TypeII_Parallel_SimE_Master_Process
NOTATION
(* k_s: Set of row indices for each process s. *)
(* Φ: The complete current solution. *)
INITIALIZATIONS;
Begin
Repeat
ForEach $s \in m$ Generate_Row_Indices k_s **EndForEach**;
(* For each slave process. *)
ParFor
Slave_ Process(Φ, k_s)
(* Broadcast cur. placement and row-indices. *)
EndParFor
ParFor
Receive_Partial_Placement_Rows
EndParFor
Construct_Complete_Solution
Until (Stopping Criteria is Satisfied)
Return Best_Solution.
End. (*Master_Process*)

Figure 5. Outline of Master Process for Type II Parallel SimE Algorithm.

5.2. Type II Parallelization

The domain decomposition method involves the partitioning of a complete solution into smaller domains to be optimized in parallel. For SimE, this implies the parallelization of all its operators, including *Allocation*. Hence, the search behavior of the parallel algorithm will differ from the serial algorithm. *Allocation* function division requires that alterations performed by the individual sub-allocation functions on the sub-solutions should not overlap, thus allowing the concurrent relocation of several selected cells at a time. After each iteration, the sub-solutions are merged to avoid missing parts of the search space and then re-partitioned. The elements are partitioned row wise among the m processors. This type of partitioning facilitates the adaptation of SimE to type II parallelization as each row can be easily processed independently. 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 2 for comparison).

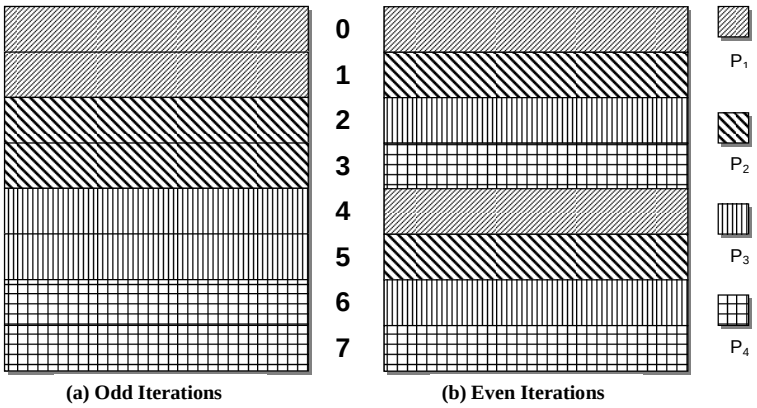

Figure 6. The row allocation pattern when 8 rows are allocated to 4 processors during (a) odd numbered iterations (partitioning pattern 2); (b) even numbered iterations (partitioning pattern 1).

This type of parallelization strategy has been attempted earlier for standard cell placement on a network of workstations [1]. The row allocation pattern that was proposed in [1] is made up of two alternating sets. In the even iterations, each slave gets a slice of $\lceil \frac{K}{m} \rceil$ rows, (where m is the number of processors, and K is the total number of rows in the placement) while in the odd iterations the j^{th} slave gets the set of rows j, $j + m$, $j + 2m$, and so on. Figure 6(a) shows the row distribution pattern for odd-numbered iterations while Figure 6(b) is the partitioning for the even-numbered cycles when 8 rows are used with 4 processors. It was stated that with this fixed pattern of assigning rows to slaves in alternate steps, each cell can move to any position on the grid in at most two steps [1].

The pseudocode of the type II parallel SimE is given in Figures 5 and 7. As can be seen, one of the processors (the master) is in-charge of running SimE on a particular partition as well as performing the following tasks periodically at the end of each iteration: (1) receive the partial placements from all other processors and combine them into a new solution, (2) obtain a new row allocation, and finally, (3) distribute the new solution and row allocation among the processors. The number of rows assigned to each processor depends on the size of the placement and the number of processors. This is repeated for all iterations until the termination condition is met.

The consequence of *Allocation* parallelization, however, 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

Table II. Results for Wirelength-Power Type II Parallel SimE Strategies.

Ckt. Name	$\mu(s)$	Seq. Time	Processors	Fixed Row Pattern	Random Row Pattern
s1196	0.684	92	2	45	50
			3	36 (95)	38
			4	33 (94)	32
			5	29 (89)	31
s1488	0.673	186	2	105	102
			3	60 (98)	65
			4	37 (94)	45
			5	43 (92)	36
s1494	0.650	49	2	42	44
			3	60	35
			4	176	29
			5	196 (94)	25
s1238	0.719	72	2	95	32
			3	116 (96)	23
			4	167(94)	20
			5	185 (93)	14(95)
s3330	0.699	2765	2	1900	1091
			3	930 (99)	574
			4	748	373
			5	724 (97)	378

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.

To observe if a different row allocation pattern than the one mentioned earlier [1] can lead to a different behavior, random row allocation [11] was also attempted in the experimentation. Two parallel multiobjective algorithms, a wirelength-power only and the other including delay optimization as well, were implemented using two types of row allocation patterns for each. No division of wirelength and delay cost calculations was done because of little potential gain as evident by the profiling results and type I parallelization.

Table III. Results for Wirelength-Power-Delay Type II Parallel SimE Strategies.

Ckt. Name	$\mu(s)$	Seq. Time	Processors	Fixed Row Pattern	Random Row Pattern
s1196	0.634	134	2	96	85
			3	37	70
			4	36	55
			5	43(98)	30
s1488	0.523	244	2	54	50
			3	39	80
			4	76	45
			5	70	50
s1494	0.626	253	2	116 (88)	235
			3	73 (87)	93
			4	110 (86)	115
			5	103 (87)	96 (98)
s1238	0.666	187	2	38	110
			3	78	75
			4	83	35
			5	34 (98)	78
s3330	0.674	13007	2	4676 (90)	3171
			3	2604 (87)	1658 (90)
			4	2062 (83)	1105 (86)
			5	1336 (80)	1031 (86)

Tables II and III show the results of type II parallel SimE for two and three objectives optimization respectively. For the results in Table II, the serial algorithm was run for 3500 iterations while the parallel runs were done starting at 4000 iterations and 500 additional iterations added with every additional processor. In Table III, the serial version ran for 5000 iterations and 1000 more iterations for each additional processor were done. This was done because additional iterations are required for the type II parallel algorithm to converge because of partitioning. In cases where the parallel algorithm failed to achieve the highest serial quality, the time shown is for the percentage of serial quality indicated in brackets. The tables show that the speed-up trend and solution qualities are better in case of random row allocation for both optimization versions. It is evident that parallelization of alloca-

ALGORITHM TypeII_Parallel_SimE_Slave_Process(Φ, k_s)
NOTATION
(* Φ^s are the rows assigned to slave s. *)
(* m_i is module i in Φ^s. *)
(* g_i is the goodness of m_i. *)
Begin
Receive Placement_ And_ Indices
EVALUATION:
ForEach $m_i \in \Phi^s$ evaluate g_i **EndForEach**;
SELECTION:
ForEach $m_i \in \Phi^s$ **DO**
Begin
If *Random* $> Min(g_i + B, 1)$
Then
Begin
$S^s = S^s \cup m_i$; Remove m_i from Φ^s
End
End
Sort the elements of S^s
ALLOCATION:
ForEach $m_i \in S^s$ **Do**
Begin
Allocate($m_i, {\Phi_i}^s$)
(* Allocate m_i in local partial solution rows ${\Phi_i}^s$. *)
End
Send_Partial_Placement_Rows
End. (*Slave_Process*)

Figure 7. Outline of Slave Process for Type II Parallel SimE Algorithm.

tion function in type II strategy, which constitutes more than 95% of runtime (Section 3.2), leads to significant speed-ups, though at the cost of achieving lower than maximum serial qualities in some cases.

5.3. Type III Parallelization

Type III parallelization or parallel searches aim for a concurrent exploration of the search space with parallel threads that may or may not interact (by exchanging some kind of information). In the simplest form of parallel search, each thread independently performs a separate search with a different randomization. However, it has been observed that there is seldom any speed-up in this method as this is equivalent to multiple independent runs of the serial algorithm. Strategies in which threads communicate with others have shown promising results for SA,

GA and TS [4, 5, 6]. Hybrid algorithms have also been proposed in which, for instance, GA is used with parallel threads of SA or some other metaheuristic or vice versa.

Parallel searches are effective if the search subspaces of the various threads do not overlap (or have minimal overlap) so that all threads should concurrently search distinct parts of the solution space (ideally). In case of SimE, although the selection operator is non-deterministic, the outcome is highly dependent upon the goodness values. With two threads of SimE using the same solutions but with different randomization, the set of cells selected will not differ much. As such, this does not guarantee the required non-overlapping concurrent exploration of different areas of a search space. Also, the SimE allocation operator that has the greatest impact on final solution quality is deterministic. Compared to this, SA, TS, and GA, exhibit more randomness in their search operators and thus lend themselves to different randomization with parallel searches as compared to SimE.

To explore type III parallelization of SimE, a parallel SimE was implemented on the lines of asynchronous multiple Markov chain parallel simulated annealing [8], where a central processor keeps track of the best solutions found so far among all threads. Since there is no workload division in parallel searches, the only way to achieve any speed-up is to enable threads to assist each other in rapidly reaching better solutions and by minimizing the time wasted in iterations in which no good solutions are found. It is observed that initially the solution rapidly evolves to a certain quality after which successive good solutions are found after a number of inferior ones. The interval of exchanges of best solution with the central processor was varied. Each thread keeps track of the number of successive times it fails to improve the current solution and resets this counter every time a better solution is found. After a certain set limit, called the retry threshold, is exceeded, the thread starts checking at the central processor if a better solution is available. The master either provides a better solution or accepts the solution of the requesting processor if it is better than what master already has. Furthermore, to keep the master updated with the best solution found so far among all threads, so that any requesting thread may be benefited, each processor always communicates the best solution found recently to the master. Thus the parallel algorithm tries to ensure that each processor is given a chance to diversify and evolve solution on its own while a better solution is made available if present. The outline of a slave thread in type III parallel SimE algorithm is given in Figure 8.

The results for Type III parallel SimE are shown in Table IV. The processors start from at least 3 as one processor is required as a central store. Both the serial and parallel algorithms were run for 2500

```
ALGORITHM TypeIII_Parallel_SimE_Slave_Process
NOTATION
 (* Count is the current retry value. *)
Begin
 INITIALIZATIONS;
 Repeat
    EVALUATION;
    SELECTION;
    Sort the elements of S
    ALLOCATION;
    Calculate_Costs;
    If Cur_Cost > Best_Cost
    Then
      Begin
      Inform_Master;
      Count = 0;
      End
    Else
      Count = Count + 1
    EndIf
    If Count > Retry_Threshold
    Then
      Begin
        If Cost_master < Cost_cur
          Then Get_New_Placement
      End
 Until (Stopping Criteria is Satisfied)
End. (*Slave_Process*)
```

Figure 8. Structure of the Type III Parallel Simulated Evolution Algorithm.

iterations at each processor. All runs were performed using the same starting solution but with different randomization seeds. Four different retry values of 50, 100, 150 and 200 iterations were tested. The runtimes show little deviation from the serial runtime. This indicates that the search derives negligible benefit from cooperating processes. Since there is no workload division, the results are virtually identical to the serial algorithm runs, though for higher threshold values consistently higher quality results, sometimes exceeding the serial quality, were obtained. These results strongly relate to the property of SimE that independent searches are not diversified enough when based solely on different randomizations to assist each other in reaching better solutions in less time than the serial algorithm.

Table IV. Results for Type III Parallel SimE

Ckt. Name	$\mu(s)$	Seq. Time	Retry Val.	Time for Parallel p=3	p=4	p=5
s1494	0.673	121	50	130	122	130
			100	118	113	115
			150	125	120	115
			200	110	119	110
s1238	0.719	72	50	70	71	68
			100	64	60	62
			150	70	66	70
			200	71	60	60

6. General Observations

Based on results of the three parallelization strategies, some overall observations can be made. Although it appears that the structure of the generic SimE algorithm lends itself easily to a low-level parallelization, the nature of cost functions (problem instance) and the type of allocation method used dictate the degree of parallelism possible. Type I parallelization would be suitable if goodness calculation is computationally intensive, there is a sparse data dependence among elements and/or the allocation function can be parallelized without affecting its outcome. Secondly, domain decomposition implicitly divides the solution and parallelizes all SimE operators, but the ability to achieve high quality solutions depends again upon the problem instance or the design of allocation operator to cope with parallel domains, i.e., maintaining the algorithm's convergence properties. Lastly, parallel searches are not beneficial to SimE due to its metaheuristic search behavior, as mentioned in Section 5.3, unless some mechanism to diversify the search are introduced additionally. Use of a different allocation function at each thread can be one way of achieving this, whereby the searches are directed in different directions by exploiting the different ways of optimizing the given problem with different allocation functions. Another promising idea might be the use of concepts borrowed from population based evolutionary metaheuristics, such as GA, in conjunction with parallel SimE threads. For instance, solutions from independent, parallel threads may be combined intelligently using crossover operators that

take advantage of SimE goodness measure to produce better starting solutions for the next SimE iterations in each of the parallel threads.

Parallel SA [10], GA [13] and TS [12] were also implemented for the same optimization problem and it was found that parallel cooperative searches best suited SA and GA, while a type I parallelization of TS gave the best speed-ups. At present, a fair and thorough comparison among these different parallelized metaheuristics is being explored.

7. Conclusions

The paper explored parallel SimE strategies for a multiobjective VLSI cell placement, studying the applicability of each class of parallelization to the SimE algorithm structure with a given problem instance. Comparing strategies in an identical setup, it was identified why one particular strategy is more suitable than the other for SimE parallelization using the placement problem as an example of a large optimization instance. The paper identifies the generalities of SimE parallelization that can be extended to other problem instances as well.

Acknowledgements

The authors thank King Fahd University of Petroleum & Minerals (KFUPM), Dhahran, Saudi Arabia, for support under Project Code COE/CELLPLACE/263.

References

1. Kling, R. M. and Banerjee, P.: ESP: Placement by Simulated Evolution. *IEEE Transaction on Computer-Aided Design*, **3(8)**, 245–255 (1989)
2. Sait, S. M. and Youssef, H.: *Iterative Computer Algorithms with Applications in Engineering: Solving Combinatorial Optimization Problems*. IEEE Computer Society Press, California (1999)
3. Sait, S. M. and Youssef, H.: *VLSI Physical Design Automation: Theory and Practice*. World Scientific Pubishers (2001)
4. Crainic, T. G. and Toulouse, M.: Parallel Strategies for Metaheuristics. In Glover, F. W. and Kochenberger, G. A. (eds.) *Handbook of Metaheuristic*, pages 465–514, Kluwer Academic Publishers (2003)
5. Cung, V.-., Martins, S. L., Ribeiro, C. C. and Roucairol, C.: Strategies for the Parallel Implementation of Metaheuristics. In Ribeiro, C. C. and Hansen, P. (eds.) *Essays and Surveys in Metaheuristics*, pages 263–308, Kluwer Academic Publishers (2001)

6. Ekşioğlu, S. D., Pardalos, P. M. and Resende, M. G. C.: Models for Parallel and Distributed Computation - Theory, Algorithmic Techniques and Applications. In Corrêa, R., Dutra, I., Fiallos, M. and Gomes, F. (eds.) *Parallel Metaheuristics for Combinatorial Optimization*, pages 179–206, Kluwer Academic Publishers (2002)
7. Sait, S. M. and Khan, J. A.: Simulated Evolution for Timing and Low Power VLSI Standard Cell Placement. *Elsevier Engineering Applications of Artificial Intelligence*, **16(5-6)**, 407–423 (2003)
8. Chandy, J. A., Kim, S., Ramkumar, B., Parkes, S. and Banerjee, P.: An Evaluation of Parallel Simulated Annealing Strategies with Application to Standard Cell Placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **16(4)**, 398–410 (1997)
9. Yager, R. R.: On Ordered Weighted Averaging Aggregation Operators in Multicriteria Decisionmaking. *IEEE Transaction on Systems, MAN, and Cybernetics*, **18(1)**, 183–190 (1988)
10. Sait, S. M., Zaidi, A. and Ali, M. I.: Asynchronous MMC based Parallel SA Schemes for Multiobjective Standard Cell Placement. In: *Proceedings of the IEEE International Symposium on Circuits and Systems*, Kos, Greece, 21 - 24 May 2006
11. Sait, S. M., Ali, M. I. and Zaidi, A.: Multiobjective VLSI Cell Placement using Distributed Simulated Evolution Algorithm. In: *Proceedings of the IEEE International Symposium on Circuits and Systems*, Kobe, Japan, 23 - 26 May 2005, pages 6226–6229
12. Minhas, M. R. and Sait, S. M.: A Parallel Tabu Search Algorithm for Optimizing Multiobjective VLSI Placement. In *Springer-Verlag, Lecture Notes in Computer Science Series*, pages 587 – 595 (2005)
13. Sait, S. M., Faheemuddin, M., Minhas, M. R. and Sanaullah, S.: Multiobjective VLSI Cell Placement using Distributed Genetic Algorithm. In: *Proceedings of the Genetic and Evolutionary Computation Conference*, Washington, D.C. USA, 24 - 29 June 2005, pages 1585 – 1586
14. Adamidis, P: Review of Genetic Algorithms Bibliography. *Technical Report*, Aristotle University of Thessaloniki, Greece (1994)
15. Cantú-Paz, E.: A Survey of Parallel Genetic Algorithms. In *Calculateurs Parallles, Reseaux et Systems Repartis* (1998)
16. De Falco, I., Del Balio, R., Tarantino, E. and Vaccaro, R.: Improving Search by Incorporating Evolution Principles in Parallel Tabu Search. In: *Proceedings of the First IEEE Conference on Evolutionary Computation*, Orlando, Florida USA, 27 - 29 June 1994, pages 823 – 828
17. Taillard, E.: Some Efficient Heuristic Methods for the Flow Shop Sequencing Problem. *European Journal of Operational Research*, **417**, 65–74 (1990)
18. Garica, B.-L., Potvin, J.-Y. and Rousseau, J.-M.: A Parallel Implementation of the Tabu Search Heuristic for Vehicle Routing Problems with Time Window Constraints. *Computers & Operations Research*, **21(9)**, 1025–1033 (1994)
19. Crainic, T. G., Toulouse, M. and Gendreau, M.: Towards a Taxonomy of Parallel Tabu Search Heuristics. *INFORMS Journal of Computing*, **9(1)**, 61–72 (1997)
20. Lee, S.-Y. and Lee, K. G.: Synchronous and Asynchronous Parallel Simulated Annealing with Multiple-Markov Chains. *IEEE Transactions on Parallel and Distributed Systems*, **7(10)**, 993–1008 (1996)