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
{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 problems with a very large set of elements to optimize, such as in VLSI placement and routing, runtimes can still be very large and parallelization is an attractive option. 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 other SimE based optimization problems.

1. Introduction

Simulated evolution (SimE), proposed by Kling and Banerjee [5] in 1987, 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 [10]. 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 [2, 3, 4]. 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 [2]. 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 we explore the parallelization of SimE when 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, we present various parallelization approaches and a comparison amongst them with respect to SimE metaheuristic characteristics and problem instance interaction. The paper is organized as follows: Section 2 explains our combinatorial optimization problem. Section 3 briefly describes the SimE algorithm. Section 4 presents an analysis of our sequen-

1-4244-0054-6/06/$20.00

tial implementation's runtime. In Section 5 we relate to previous work while Section 6 describes parallel strategies and experimental results. General observations are given in Section 7, and we conclude in Section 8.

2. Optimization Problem and Cost Functions

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

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

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

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

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

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

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. By assuming a fixed supply voltage and clock frequency, and since C_i varies with wirelength of net i, we have:

$$p_i \simeq l_i \cdot S_i$$

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

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

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

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

where CD_i is the switching delay of the cell driving net vi and ID_i is the interconnect delay of net vi. The placement phase affects ID_i because CD_i is technology dependent parameter and is independent of placement.

The delay cost function can be written as:

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

Width Cost: Width cost is given by the maximum of all the row widths in the layout. We have constrained layout width 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, we can express width constraint as below:

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

Overall Fuzzy Cost Function: Since we are optimizing three objectives simultaneously, we need to have a cost function that represents the effect of all three objectives in the form of a single quantity. We propose the use of fuzzy logic to integrate these multiple, possibly conflicting objectives into a scalar cost function. The justification for using our fuzzy aggregating function and its details can be found in [9]. The resulting quality measure for a solution s is denoted as $\mu(s)$ and is a value between 0 and 1, with 1 representing an optimal solution.

3. Simulated Evolution Algorithm

The structure of the SimE algorithm is shown in Figure 1. 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 and retaining the near-optimal 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 we are optimizing three objectives, we use a multiobjective goodness measure [9].

The second step of the SimE algorithm is *Selection*. *Selection* takes as input a bias value B, the solution Φ together with the estimated *goodness* of each element. 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 decision whether to assign an element m_i to the set S is based solely 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. We use a biasless selection function developed in earlier work [9].

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* [10]. The goal of *Allocation* is to favor improvements over the previous generation, without being too greedy. Different techniques can be used here [5] and we use the 'sorted individual best fit method' [9].

4. 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

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

Figure 1. Simulated evolution algorithm.

the 'sorted individual best-fit' method, allocation routine heavily influences the runtime of the algorithm. The impact of this is discussed in Section 6.

5. 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 [2, 3, 4], some of which paint a global view of the field and generalize the various strategies used into broad classes. In this paper we follow the approach taken in [2] and classify the various attempted strategies into three comprehensive types according to the source of parallelism. These are [2]:

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 [5] was for a single objective (wirelength) VLSI cell placement that can be classified under type II. In this paper we use a more complex multiobjective cost function and compare the parallel strategies along the complete spectrum of types discussed here.

6. 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.

6.1. 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, our SimE type I parallelization focuses only on distribution of cost calculations and goodness evaluation.

For our 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.

```
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. (*Mater_Process*)
```

Figure 2. Outline of Master Process for Type I Parallel SimE Algorithm.

The calculation of wirelength and power goodness values of each cell requires that the wirelength of all fan-in cells be known [9]. This complicates the partitioning of cells among processors; if a processor needs 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, we implemented a type I parallel algorithm for only wirelength and power optimization to observe the results of partitioning. Figures 2 and 3 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 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 1. 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

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 3. Outline of Slave Process for Type I Parallel SimE Algorithm.

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.

6.2. 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 1 for comparison).

Table 1. 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 4. Outline of Master Process for Type II Parallel SimE Algorithm.

This type of parallelization strategy has been attempted earlier for standard cell placement on a network of workstations [5]. The row allocation pattern that was proposed in [5] 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. 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 [5].

The pseudocode of the type II parallel SimE is given in Figures 4 and 5. 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 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

Table 2. Results for Wirelength-Power Type II Parallel SimE Strategies.

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

Table 3. Results for Wirelength-Power-Delay Type II Parallel SimE Strategies.

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

```
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 ∈ Φ^s evaluate g_i EndForEach;
 SELECTION:
    ForEach m_i ∈ Φ^s DO
      Begin
        If Random > Min(g_i + B, 1)
        Then
          Begin
            S^s = S^s ∪ m_i; Remove m_i from Φ^s
          End
      End
 Sort the elements of S^s
 ALLOCATION:
    ForEach m_i ∈ S^s Do
      Begin
        Allocate(m_i, Φ_i^s)
    (* Allocate m_i in local partial solution rows Φ_i^s. *)
      End
 Send_Partial_Placement_Rows
End. (*Slave_Process*)
```

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

positions.

To observe if a different row allocation pattern than the one mentioned earlier [5] can lead to a different behavior, we also experimented with random row allocation [7]. 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.

Tables 2 and 3 show the results of type II parallel SimE for two and three objectives optimization respectively. For the results in Table 2, 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 3, 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 allocation function in type II strategy, which constitutes more than 95% of runtime (Section 4), leads to significant speed-ups, though at the cost of achieving lower than maximum serial qualities in some cases.

6.3. 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 [2, 3, 4]. 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

```
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 6. Structure of the Type III Parallel Simulated Evolution Algorithm.

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, we implemented a parallel SimE on the lines of asynchronous multiple Markov chain parallel simulated annealing [1], 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. We varied the interval of exchanges of best solution with the central processor. Each thread keeps track of the number of successive times it fails

Table 4. 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

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 6.

The results for Type III parallel SimE are shown in Table 4. 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 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.

7. General Observations

Based on results of the three parallelization strategies, we make some overall observations. 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 6.3, 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.

We have also implemented parallel SA [11], GA [8] and TS [6] for the same optimization problem and found that parallel cooperative searches best suited SA and GA, while a type I parallelization of TS gave the best speed-ups. Currently we are working on a fair and thorough comparison among these different parallelized metaheuristics.

8. Conclusion

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.

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

References

[1] J. A. Chandy, S. Kim, B. Ramkumar, S. Parkes, and P. Banerjee. An Evaluation of Parallel Simulated Annealing Strategies with Application to Standard Cell Placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(4):398 – 410, April 1997.

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

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

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

[5] R. M. Kling and P. Banerjee. ESP: Placement by Simulated Evolution. *IEEE Transaction on Computer-Aided Design*, 3(8):245–255, March 1989.

[6] M. R. Minhas and S. M. Sait. A Parallel Tabu Search Algorithm for Optimizing Multiobjective VLSI Placement. In *LNCS*, volume 3483, pages 587 – 595. Springer-Verlag GmbH, January 2005.

[7] S. M. Sait, M. I. Ali, and A. M. Zaidi. Multiobjective VLSI Cell Placement using Distributed Simulated Evolution Algorithm. In *Proceedings of the International Symposium on Circuits and Systems*, pages 6226 – 6229, May 2005.

[8] S. M. Sait, M. Faheemuddin, M. R. Minhas, and S. Sanaullah. Multiobjective VLSI Cell Placement using Distributed Genetic Algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1585 – 1586. ACM Press, 2005.

[9] S. M. Sait and J. A. Khan. Simulated Evolution for Timing and Low Power VLSI Standard Cell Placement. *Elsevier Engineering Applications of Artificial Intelligence*, 16:407 – 423, August - September 2003.

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

[11] S. M. Sait, A. M. Zaidi, and M. I. Ali. Asynchronous MMC based Parallel SA Schemes for Multiobjective Standard Cell Placement. In *Proceedings of the International Symposium on Circuits and Systems*, May 2006.