ACCELERATING MULTIOBJECTIVE VLSI CELL PLACEMENT WITH PARALLEL EVOLUTIONARY/TABU SEARCH HEURISTICS

Sadiq M. Sait *Mahmood R. Minhas* *Mustafa I. Ali* *Ali M. Zaidi*

College of Computer Sciences & Engineering
King Fahd University of Petroleum & Minerals
Dhahran - 31261, Saudi Arabia
{sadiq, minhas, mustafa,alizaidi}@ccse.kfupm.edu.sa

ABSTRACT

Multiobjective combinatorial optimization problems in various disciplines remain to be the focus of extensive research due to their inherent hard nature and difficulty of finding near-optimal solutions. Iterative heuristics like Tabu Search (TS) and Simulated Evolution (SimE) have successfully been employed to solve a range of such optimization problems [1]. These heuristics are able to obtain high quality solutions, but for most real-life large size problems they may have huge runtime requirements. Parallelization of these heuristics is one of the adopted practical approach to achieve the solutions within acceptable runtimes. In this paper we address a hard multiobjective optimization problem namely, VLSI cell placement [2] with three possibly conflicting objectives: interconnect wirelength, power dissipation, and timing performance. Two heuristics namely, parallel tabu search (TS) and parallel simulated evolution (SimE) are presented. Fuzzy rules are used to design a multiobjective aggregate cost function. The parallel TS implementation is a based on a synchronous candidate list partitioning model, whereas the parallel SimE implementation is based on random distribution of rows to processors [3, 4]. For comparison purposes, a parallel genetic algorithm (GA) based on the island model [5], and a parallel simulated annealing (SA) based on the asynchronous multiple-Markov chain [6] are also implemented. Results of experiments on ISCAS-85/89 benchmark circuits are presented, with solution quality and speedup used as metrics for the comparative/relative evaluation of the presented heuristics.

1. INTRODUCTION

Multiobjective VLSI cell placement is a hard problem to solve due to the sheer complexity of modern circuit density. Conventional constructive techniques often prove inadequate for achieving near optimal solutions. Iterative heuristics on the other hand have proven to be extremely useful in reaching very good quality placement solutions. A primary advantage of iterative heuristics that gives them an upper edge over the former is their probabilistic ability to escape from local minima [1]. However, one of the hindrances in using these heuristics is their excessively large runtime requirements, and hence there is a need for accelerating them. Of the various acceleration approaches employed, parallelization has exhibited the most potential. With parallelization, not only is it possible to reduce runtimes but also to reach better quality solutions.

In this work, we aim at accelerating the solution of multiobjective VLSI placement problem by parallelizing two well-known heuristics: TS and SimE. Both of these heuristics have successfully been applied to this problem [7, 8]. We begin with the formulation of the placement problem and cost functions for the three objectives namely: interconnect wirelength, power consumption, and timing performance (delay).

The rest of the paper is organized as follows. In the following section, we give details of the multi-objective VLSI cell placement problem, and models for estimating the costs for the various objectives to be optimized. In Section 3 we review some previous work related to parallelization of TS and SimE. In Section 4, details of the parallel algorithms are presented. Experimental setup, results obtained on ISCAS benchmark circuits and other observations are given in Section 5, followed by Conclusion in Section 6.

2. VLSI CELL PLACEMENT PROBLEM & COST FUNCTIONS

Due to its complexity, VLSI design process is divided into several intermediate levels of abstraction. Placement is a phase in physical level, and is a process of arranging circuit cells (components) on a layout surface with a goal of optimizing certain design objectives while meeting the given constraints [2].

VLSI cell placement 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 objectives are optimized subject to given constraints.

In standard cell placement all the cells are constrained to have the same height, while their widths are variable and depend upon the complexity [2]. Cells are arranged in rows with routing channels between the rows. A typical standard cell layout is shown in Figure 1. Due to varying width of cells, row widths may be unequal depending on the type and number of cells placed in a row. An approximation would be to treat cells as points, but in order to have a more accurate estimate of wirelength, widths of cells are taken into account. Heights of routing channels are estimated using the vertical constraint graphs constructed during the channel routing phase. With this information, a fairly accurate estimate of power dissipation, delay and total wirelength can be obtained [2].

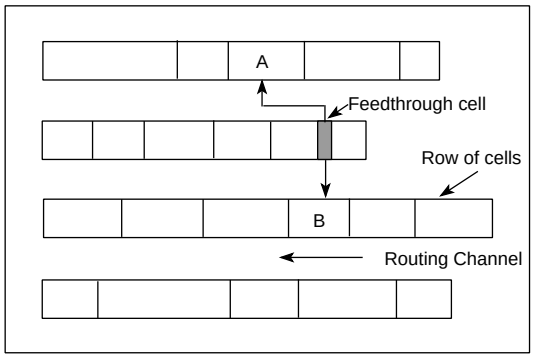


Fig. 1. Standard Cell Layout

2.1. Cost Functions

Now we formulate cost functions for the three said objectives and for the width constraint.

Wirelength Cost: Interconnect wirelength of each net in the circuit is estimated using Steiner tree approximation. Total wirelength is computed by adding all these 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 (which is the same as number of modules for single output cells).

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

$$p_i \simeq C_i \cdot S_i \tag{2}$$

where C_i is total capacitance of net i, and S_i is the switching probability of net i. C_i depends on wirelength of net i, so Equation 2 can be written as:

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

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

$$Cost_{power} = \sum_{i \in M} p_i = \sum_{i \in M} (l_i \cdot S_i) \tag{4}$$

Delay Cost: The delay of any given path is computed as the summation of the delays of the nets $v_1, v_2, ..., v_k$ belonging to that path and the switching delays of the cells driving these nets. Delay 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:

$$T_\pi = \sum_{i=1}^{k-1} (CD_i + ID_i) \tag{5}$$

where CD_i is the switching delay of the cell driving net v_i and ID_i is the interconnect delay of net v_i. The placement phase only affects ID_i because CD_i is technology dependent parameter and is independent of placement. The delay cost function can be written as:

$$Cost_{delay} = max\{T_\pi\} \tag{6}$$

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 average 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:

$$Width - w_{avg} \leq \alpha \times w_{avg} \tag{7}$$

2.2. Fuzzy Aggregate Cost Function:

Since, we are optimizing three objectives simultaneously, we need an aggregate cost function to evaluate the overall cost of a solution. Some traditional methods of aggregation include weighted sum and goal programming. The weighted sum method works by adding the individual costs using different weights for each of them as follows:

$$Cost = \sum_{i=1}^{k} w_i f_i c_i \tag{8}$$

Where c_i are constant multipliers that will properly scale the objectives. The best results are usually obtained if $c_i = \frac{1}{f_i^0}$, where f_i^0 are the values of cost functions in the first iteration of the search algorithm. The main weakness of this approach is the difficulty to determine the appropriate weights [9].

In this work, we resorted to the use of fuzzy logic for integrating three costs. Fuzzy logic allows one to describe the objectives in terms of linguistic variables and use fuzzy rules to find the overall cost of a solution. Three linguistic variables namely, wirelength, power dissipation, and delay are defined and are mapped to the membership values in fuzzy sets *small wirelength*, *low power dissipation*, and *short delay* respectively. The range of acceptable solutions is depicted in Figure 2(a). Membership values are computed using the fuzzy membership functions shown in Figure 2(b). The following fuzzy rule is used to combine the objectives.

Rule: IF a solution has *small wirelength* AND *low power dissipation* AND *short delay* **THEN** it is a good solution.

The above rule is translated to *and-like* OWA fuzzy operator [10] and the membership $\mu(x)$ of a solution x in fuzzy set *GOOD solution* is given as:

$$\mu(x) = \beta \times min(\mu_p(x), \mu_d(x), \mu_l(x)) + (1 - \beta) \times \frac{1}{3} \sum_{j=p,d,l} \mu_j(x) \tag{9}$$

where $\mu_j(x)$ for $j = p, d, l$ 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. As layout width is a constraint, therefore if a solution violates this constraint, it is not a valid solution and is hence discarded. However, for an objective, by increasing and decreasing the value of g_i (see Figure 2), its preference can be varied in the aggregate membership function. The lower bounds O_j for three objectives are computed as given in Equations 10-13:

$$O_l = \sum_{i=1}^{n} l_i^* \quad \forall v_i \in \{v_1, v_2, ..., v_n\} \tag{10}$$

$$O_p = \sum_{i=1}^{n} S_i l_i^* \quad \forall v_i \in \{v_1, v_2, ..., v_n\} \tag{11}$$

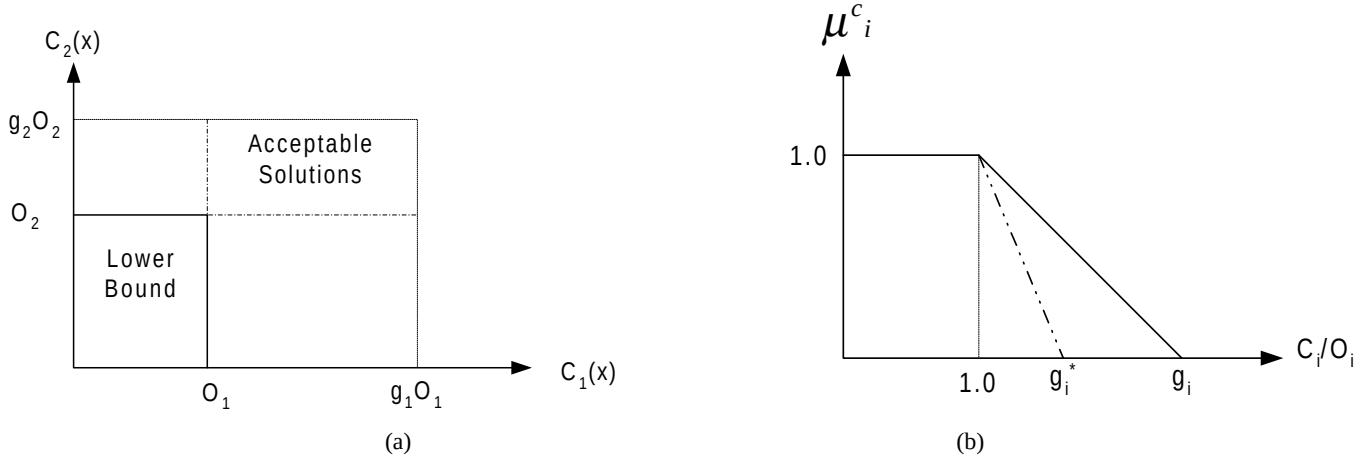


Fig. 2. (a) Range of acceptable solution set. (b) Membership functions.

$$O_d = \sum_{j=1}^{k} CD_j + ID_j^* \quad \forall v_j \in \{v_1, v_2, ..., v_k\} \; in \; path \; \pi_c \tag{12}$$

$$O_{width} = \frac{\sum_{i=1}^{n} Width_i}{\#\; of \; rows \; in \; layout} \tag{13}$$

where O_j for $j \in \{l, p, d, width\}$ are the optimal costs for wirelength, power, delay and layout width respectively, n is the number of nets in layout, l_i^* is the optimal wirelength 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. REVIEW OF PARALLEL ITERATIVE HEURISTICS

General Non-deterministic Iterative Heuristics such as Tabu Search, Simulated Evolution, Simulated Annealing, and Genetic Algorithms are getting more widely adopted to obtain near optimal solutions to numerous hard problems [1]. For small problems, stochastic heuristics have reasonable runtime requirements. For instance, a VLSI cell placement problem with few hundred modules, it is possible to find very good solutions in reasonable time. However, most practical circuits are very large and require several hours of computer time to solve [7]. One way to adapt iterative techniques to solve large problems and traverse larger search spaces in reasonable time is to resort to parallelization [4, 11].

In this paper we address the problem of parallelizing non-deterministic iterative heuristics to solve the multiobjective VLSI standard cell placement problem by using a cluster of low cost PCs. The eventual goal being to achieve either lower run-times for same quality solutions, or higher quality solutions in the same amount of time as the serial approach.

This however is easier said than done: an effective parallelization strategy must consider issues such as proper partitioning of the problem to facilitate uniform distribution of computationally intensive tasks, while also enabling a more thorough traversal of the complex search space. It is worth mentioning, that in certain design problem, such as the one discussed in this paper, sometimes even the smallest savings

in time are worth several times the cost, or conversely, the importance of achieving a specific higher quality becomes essential, irrespective of time taken or resources used. In the subsequent paragraphs we present a brief review of earlier efforts towards the parallelization of TS and SimE.

Parallelization of TS: A number of parallelization techniques for TS have been reported in literature [12]. In the most straightforward and widely adopted approach k tabu search processes are spawned and run concurrently on k processors where each processor carries out independent search [12, 13]. One suggested approach was to link independent searches, where each slave runs a copy of a serial TS but with different parameter settings [14]; after a specified time the slave processes are halted and the main process selects the best solution found and broadcasts to all slave processes to start the entire search again from this new solution.

Another approach to parallelize search within an iteration is when each process is given a task of exploring a subset of the neighborhood of current solution. Two approaches are followed: *synchronous* and *asynchronous*. In the synchronous approach the various processes are always working with the same solution, but exploring different partitions of the current local neighborhood. A *master* process orchestrates the activities of the *slave* processes [13]. In the asynchronous approach, all processes are peer and usually are not all working with the same current solution [12]. Both approaches require that the set of possible moves be partitioned among the available processors so that each processor will be exploring a distinct sub-region of the current solutions neighborhood.

Suggestions to increase efficiency of TS by parallelizing also include partitioning the search space, which is difficult, or partitioning the problem into smaller sub-problems, determining the best moves for each sub-problem, and then performing a compound move [12]. Attempts to solve several practical NP-hard problems have been reported in literature. Examples include, parallel implementations for the vehicle routing problem [13], and quadratic assignment problem (QAP) [15]. For the QAP, the parallelization strategy was implemented on the Connection Machine CM-2, a massively parallel SIMD machine. A reduction in the runtime per iteration was achieved when compared to other sequential and parallel implementations [16, 15].

Parallelization of SimE: Unlike other iterative heuristics the parallelization of SimE has not been the subject of much research. Three ways of speeding up the SimE algorithm have been suggested in literature [3, 4]: (1) a distributed memory MIMD parallel algorithm, (2) a shared memory MIMD parallel algorithm that is based on an extension of basic SimE using the concepts of windowing and hierarchy, and (3) hardware acceleration which consists of implementing time consuming parts in hardware (namely, goodness computation).

A parallelization strategy for VLSI cell placement for a single objective (wirelength) was attempted on a network of workstations [3], where each station is assigned a number of rows of the placement problem, in a pre-determined order. The station executes one iteration of the SimE algorithm on the cells of the rows assigned to it. In each iteration, the rows are redistributed among the processors in a predetermined order [3].

4. PARALLELIZATION STRATEGIES AND IMPLEMENTATION

In this section we present the proposed parallelization strategies and the implementation details of the heuristics. We discuss how the computationally intensive tasks are handled in each heuristic to distribute the workload amongst the available processors.

4.1. Parallel Tabu Search

A generic intuitive strategy for parallelization is to partition the data into small subsets that are distributed among the processors. Each processor is responsible for a data subset and implements a sequential version of the concerned function (or the heuristic) over this data subset.

The sequential implementation of TS was analysed using profiling tools (GNU profiler) to obtain insight into determining the time consuming operations of the code and the usage of resources. For the circuits experimented on, between 60-80% of time was spent on computation of cost of the objectives and their fuzzification. Furthermore, experiments with parameters revealed that for our hard optimization problem with conflicting multiobjectives, large sizes of candidate list (upto 120) were required to obtain high quality solutions. Since the computation of cost for all moves in the candidate list was the most time consuming operation, (in each iteration) the algorithm was designed to partition this workload.

Therefore, the parallel Tabu Search strategy adopted in this work employs dividing the operations within a TS iteration. According to taxonomy given by Crainic et. al [17], our approach can be classified as a synchronous master-slave (one master and remaining slaves), 1-control (each process is responsible for its search), Rigid Synchronous (RS) (all processes are forced to establish communication and exchange information at specific points) and Single Point Single Strategy (SPSS) (all the processes start with the same initial solution and follow the same strategy).

Algorithm MasterProcess;
Begin
(* S_0 is the initial solution. *)
(* $BestS$ is the best solution. *)
(* PCL is the Partial Candidate List. *)
(* p is the number of slave processors. *)
(* OBM is the Overall Best Move. *)
Generate S_0 and p number of $PCLs$;
Send S_0 and a PCL to each slave process;
While *iteration-count* $<$ *max-iterations*
Receive best move and cost from each slave;
Find OBM subject to tabu restrictions;
Generate P number of $PCLs$;
Send OBM and a PCL to each slave process;
Update $BestS$; /*by applying OBM on $BestS$*/;
EndWhile
Return ($BestS$)
End. /*MasterProcess*/

Fig. 3. The master process in parallel TS.

In this implementation, there is an initialization step during which the master process (Figure 3 generates and sends an initial solution and a disjoint (non-overlapping) partial candidate list (PCL) to each slave process. A move in a PCL assigned to a slave in a particular iteration does not appear in PCLs assigned to other slaves. Each slave process searches its local neighborhood by trying each move in the partial candidate list on the initial solution and computes gains due to them. Then it sends the best move and its corresponding cost (or gain) to the master process. The master process selects the overall best move (OBM) among the moves it received from slave processes subject to tabu restrictions.

Then in each subsequent iteration, the master process sends the overall best move and a new partial candidate list to each slave process. Each slave process now starts by performing the received overall best move so that all the slave processes start their iteration from the same solution. Each slave process

Algorithm SlaveProcess;
Begin
Receive S_0 and a PCL from the master process;
$CurS = S_0$; (* Current Solution *)
While *iteration-count* $<$ *max-iterations*
Try each move in PCL and compute cost;
Send the best move and its cost to the master process;
Receive OBM and a PCL from the master process;
Update $CurS$ /* by applying OBM on $CurS$ */;
EndWhile
End. /*SlaveProcess*/

Fig. 4. The slave process in parallel TS.

searches its local neighborhood and sends the best move and its cost to the master process. The pseudo code of the slave process is given in Figure 4.

4.2. Parallel Simulated Evolution

The parallelization of the SimE algorithm is carried out by partitioning the workload among available processors. The partitioning is done according to rows. The workload for each slave in the cell placement problem is the computation of SimE operations of Evaluation, Selection, and Allocation on it's assigned rows [3].

The row allocation pattern that was proposed in [3] is made up of two alternating sets of rows. In the even iterations, each slave gets a slice of $\lceil \frac{K}{m} \rceil$ rows, (where m is the number of slaves, 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 has been shown that with the above 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 [3]. The consequence of row partitioning however is that the each processors has only a partial view of the placement. This hinders free cell movement, making it more difficult for cells to reach their optimal locations. Results from implementing this strategy on our multiobjective optimization problem revealed that even when given a large amount of time, the best solution obtained was poorer than one achieved by the serial implementation.

Though the lack of a global placement view will always exist in case of a parallel algorithm, the effects of restrictive cell movement can be alleviated by using a better row allocation pattern. The use of a pattern that facilitates a variety of combination among the rows sounds intuitively better. In this work we propose an enhanced random row allocation scheme. The pseudo code of the parallel simulated evolution is illustrated in Figures 5 and 6. 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 and evaluate its fitness, (2) partition the new solution to obtain a new row allocation, and finally, (3) distribute the resulting sub-populations among the processors. The number of rows randomly assigned depends on the size of the placement and the number of processors. This is repeated for all iterations until the termination condition is met.

Algorithm Slave_Process($CurS, \Phi_s$)
Notation
(* B is the bias value. *)
(* $CurS$ is the current solution. *)
(* 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;
Selection:
ForEach $m_i \in \Phi_s$ **DO**
Begin
If $Random > Min(g_i + B, 1)$
Then
Begin
$S = S \cup m_i$; Remove m_i from Φ_s
End
End
Sort the elements of S
Allocation:
ForEach $m_i \in S$ **Do**
Begin
Allocate(m_i, Φ_s)
(* Allocate m_i in local partial solution rows Φ_s. *)
End
Send_Partial_Placement_Rows
End. (*Slave_Process*)

Fig. 5. Structure of the Distributed Simulated Evolution Algorithm.

5. EXPERIMENTS & RESULTS

5.1. Experimental Setup

The experimental setup consists of the a homogeneous cluster of 8 machines, x86 architecture, Pentium-4 of 2 GHz clock speed, and 256 MB of memory. These machines are connected by 100Mbit/s Ethernet switch. The operating system used in RedHat Linux 7.3 (kernel 2.4.7-10). The paradigm used for parallelization is MPI (Message Passing Interface). Specifically, MPICH (a portable implementation of MPI standard 1.1) is used in our implementation. In terms of GFlops measure, the maximum performance of the cluster, with NAS Parallel Benchmarks was found to be 1.6 GFlops, (using NAS's LU, Class A, for 8 processors). Using this same benchmark for a single processor, the individual performance of one machine was found out to be 0.3 GFlops. The maximum bandwidth that was achieved using PMB was 91.12 Mbits/sec, with an average latency of 68.69 μsec per message.

5.2. Results & Discussion

In this section we present the performance of the implemented heuristics. ISCAS-85/89 circuits are used as performance benchmarks for evaluating the proposed parallel TS placement technique. These circuits are of various sizes in terms of number of cells and paths, and thus offer a variety of test cases.

For comparison purposes, a parallel genetic algorithm (GA) which is a derivative of a standard distributed GA and follows the island model, with independently evolving sub-populations and periodic ex-

```
Algorithm Parallel_Simulated_Evolution
   Read_User_Input_Parameters
   Read_Input_Files
Begin
   Construct_ Initial_ Placement
   Repeat
   Generate Random_ Row-Indices
        ParFor
          Slave_ Process(CurS, Φ_s)
   (* Broadcast Cur Placement And Row-Indices. *)
        EndParFor
        ParFor
          Receive_Partial_Placement_Rows
        EndParFor
      Construct_Complete_Solution
      Calculate_Cost
   Until (Stopping Criteria is Satisfied)
 Return Best_Solution.
End. (*Parallel_Simulated_Evolution*)
```

Fig. 6. Outline of Overall Parallel Algorithm.

changes of solutions through migration [5, 18], was implemented. A pseudo-diversity approach is taken, wherein similar solutions are not permitted in the population at any time. This diversity serves to widen the search, while limiting the possibility of premature convergence in local minima solution space. The initial population is constructed at the master process and distributed among N slave processes which start running serial GA on their allocated population for a predefined number of iterations called the Migration Frequency (MF). Then each slave process sends MR (Migration Rate) number of its best solutions to the master process, which selects MR overall best solutions and broadcasts them to all slave processes. Each slave process absorbs the incoming best solutions into its population (if they are not already found) by replacing the weakest solutions. Each slave process then continues with the serial GA for another MF iterations. Standard PMX crossover is used to generate offsprings [1].

We begin with a comparison of TS and GA. The quality of solution obtained and runtime required using different number of processors for both TS and GA are tabulated in Table 1. For each circuit, the number of cells are given in the table. The '$\mu(s)$ TS' and '$\mu(s)$ GA' columns show the aggregate fuzzy membership of solution obtained by TS and by GA respectively, whereas 'p' denotes the number of processors used. It should be noted that runtimes shown are for achieving a certain fixed quality.

For large circuits, Parallel TS clearly outperformed Parallel GA, both in terms of quality and runtimes. As can be observed, in case of large circuits, parallel GA was unable to find a reasonable quality solution even after running for a large amount of time. Even for smaller circuits, the solution quality obtained using TS is significantly superior to that obtained using GA. The proposed parallel TS has shown a consistent trend in terms of speedup with increasing number of processors. Figure 7 shows a speedup plot for some selected large circuits, and it can be seen that linear speedup was obtained. On the other hand, parallel GA did not show such performance or trend.

Table 2 illustrates the quality of solution obtained for the two SimE parallelization schemes, namely random row distribution strategy and fixed row distribution strategy. The table illustrates the amount of time for the taken to reach a predefined fuzzy membership with increasing number of processors. For the strategy proposed in this paper, as can be seen, there is a constant decrease in runtime, for all

Table 1. Run times and solution quality $\mu(s)$ for achieving a target membership for serial and parallel TS/GA approaches. UH indicates unreasonably high runtime requirement.

Circuit	# of	$\mu(s)$	Time for	Time for Parallel TS						$\mu(s)$	Time for	Time for Parallel GA		
Name	Cells	TS	Serial TS	p=2	p=3	p=4	p=5	p=6	p=7	GA	Serial GA	p=3	p=5	p=7
s386	172	0.688	52	28	20	17	16	15	14	0.504	15	9.9	5.7	6.7
s641	433	0.785	934	472	332	239	205	171	151	0.616	793	307	390	289
s832	310	0.644	74	40	33	23	22	20	19	0.479	128	43	37	39
s953	440	0.661	195	98	71	53	46	41	36	0.511	309	136	91	108
s1196	561	0.653	374	187	132	97	88	78	67	0.484	988	327	262	205
s1488	667	0.603	259	131	93	69	63	55	49	0.482	1883	677	435	418
s1494	661	0.601	268	137	96	72	65	57	51	0.496	1405	847	638	479
c3540	1753	0.665	2142	1146	703	547	440	370	344	-	UH	UH	UH	UH
s3330	1961	0.699	1186	590	451	313	245	210	184	-	UH	UH	UH	UH
s5378	2993	0.669	1850	914	601	467	371	312	264	-	UH	UH	UH	UH
s9234	5844	0.631	5571	2855	2006	1525	1272	1062	849	-	UH	UH	UH	UH

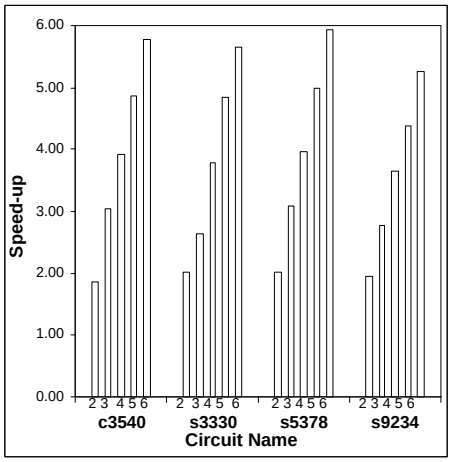
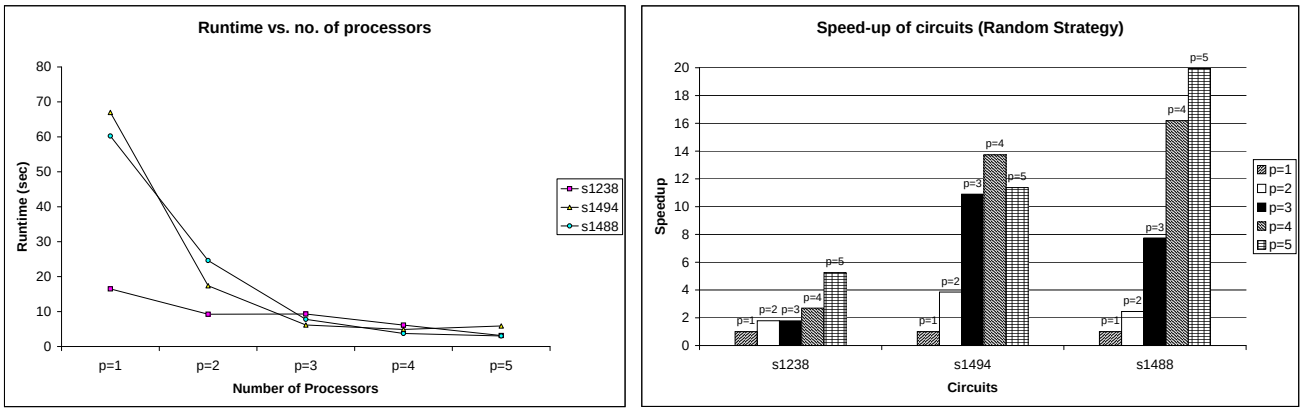


Fig. 7. Speedup obtained for selected large circuits.



a b

Fig. 8. (a) The decrease in runtime to reach a pre-defined fitness objective with increasing number of processors; (b) Speedup versus number of machines.

circuits, with increase in number of processors. Better trends are observed for medium to large circuits, than for smaller ones, as can be seen in Figure 8(a). Speedup is also illustrated in the bar-chart given in Figure 8(b). Due to space restrictions, and scaling factor limitations, not all results have been included in the same figure for sake of clarity.

The fitness values achieved with the proposed random row allocation are consistently higher in all test cases when compared to the fixed row allocation scheme, as shown by the *Qual Fixed* column in Table 2, the fixed row allocation never equals 100% of the solution quality obtained by the proposed scheme. Further, the run times are far better, and the speedup is super linear in most cases for the random row distribution strategy. This can be attributed to modified working space of the selection and allocation operators on each slave, as in each iteration different sets and combination of rows are addressed. This has resulted in even more reduced times to obtain desired solution quality than with workload partitioning alone.

Table 2. Run times of Parallel SimE for a achieving a target fitness, for serial and parallel implementations, for both random and fixed row allocation strategies. UH indicates unreasonably high times.

Circuit	# of	Time for Random Row Distribution					Qual	Time for Fixed Row Distribution			
Name	Cells	p=1	p=2	p=3	p=4	p=5	Fixed	p=2	p=3	p=4	p=5
s641	433	UH	4.99	4.97	3.99	3.87	79.7%	9.14	1.08	0.76	0.55
s1238	540	16.5	9.24	9.29	6.12	3.14	95.8%	17.83	8.47	11.30	5.71
s1494	661	67	17.4	6.15	4.88	5.89	82.3%	2.77	1.85	1.76	4.34
s1488	667	60.23	24.6	7.78	3.72	3.02	96.6%	22.0	4.89	5.1	16
s3330	1961	UH	678.02	115	108.5	49.14	33.8%	316	215	4.6	3.4
s5378	2993	UH	1620	338.2	286.6	178.6	46.8%	UH	UH	124.3	95.0

When SimE was compared to other heuristics, the following was observed. For GAs, the time for completion to obtain solutions of a certain pre-specified quality were exorbitantly high. And in some cases, for the given run-time, acceptable solutions could not be obtained. For example, for the S1494, the serial GA implementation took 1883 Seconds, and when the parallel version was executed on 7 processors the best time was 418 Seconds (with 8% inferior quality than that obatined by SimE).

When comparing Parallel SimE with parallel TS, better quality was obtained in some cases at the cost of high computation time using TS, for the same quality the run-time requirements for TS were over three times more than that required by parallel SimE. For example, for s1494, the time taken by serial TS was 268 Seconds, and when parallel TS was run on 6 processors, the runtime was 57 Seconds, (compared to 5 Seconds by SimE) with slightly better quality, and TS took over 15 Seconds to obtain solutions of same quality as SimE. A similar trend was seen for all circuits.

Table 3. Run times and solution quality $\mu(s)$ for achieving a target membership for serial and parallel SA approaches.

Circuit	# of	$\mu(s)$	Time for	Time for Parallel SA					
Name	Cells	SA	Serial SA	p=3	p=4	p=5	p=6	p=7	p=8
s386	172	0.659	18	9.5	7.2	6.7	5.5	5.0	4.7
s641	433	0.755	221	189.3	169.7	129.6	125.2	120.5	117.7
s832	310	0.638	45	35.0	25.3	19.6	15.6	14.9	14.2
s953	440	0.704	122	96.6	83.4	60.1	56.5	54.5	53.4
s1196	561	0.675	190	145.9	130.9	110.3	96.9	98.2	94.8
s1488	667	0.650	275	151.4	118.4	112.6	98.8	94.0	92.6
s1494	661	0.647	214	131.4	116.2	101.9	98.1	92.3	89.1
c3540	1753	0.734	2445	2153.0	1950.5	1510.2	1306.2	1288.2	1124.4
s3330	1961	0.793	2137	1875.5	1658.6	1572.9	1419.5	1254.5	1081.2

For simulated annealing, the asynchronous multiple-Markov chain parallelization strategy was chosen [6]. In this strategy, each slave processor runs the Metropolis loop for a fixed number of iterations. Upon completion of the loop, the slave replaces the best solution in the master only if it has found a better one, else, it receives the solution at the master to be used as an initial solution for subsequent iterations. Since all slaves are running the same Metropolis loop for the same number of iterations, the decrease in the runtime to find a solution of a specified quality is not significant, however a decreasing trend in required runtime is observed. As can be see from Table 3, given enough time, parallel SA was also able to achieve slightly better quality solutions than SimE. However, for a fixed slightly lower quality, SimE was seen to be increasingly faster than SA as processors were increased. For instance, for s1494, with 3 processors SA took 131 Seconds to achieve the desired quality, while SimE took only 6.1 Seconds. With 5 processors, SA required 101 Seconds on average, while SimE needed only 5.9 Seconds. Similar trends are seen for most circuits.

6. CONCLUSIONS

In this work, we presented two parallel iterative heuristics namely, tabu search and simulated evolution for accelerating the solving of the multiobjective VLSI standard cell placement problem. Fuzzy rules were used to design a multiobjective aggregate cost function for integration of the individual costs of various objectives. For comparison purposes, a parallel genetic algorithm (GA) and a parallel simulated annealing (SA) algorithm are also implemented. Results of experiments on ISCAS-85/89 benchmark circuits revealed that the proposed heuristics significantly reduced the runtime requirements for reaching very good quality solutions to our multiobjective problem.

Our study also included a comparison of the proposed parallelization heuristics with other known algorithms. From our observations, it was found that while TS exhibited linear speedups, it was possible to obtain super-linear speedups with SimE. The quality of solutions obtained from Parallel TS and

Parallel SA was slightly superior to other techniques. While Parallel SA required the largest run times, it provided the best quality solutions. However, the efficiency (defined as Speedup/p, where p is the number of processors) was far below 100% for SA. Finally, it can be concluded, that for appreciable quality solutions, simulated evolution with random row distribution has exhibited dramatic speedups with increase in number of processors, even when compared to other, more established heuristics. The results obtained suggest that in scenarios where placement quality considerations are overridden by design time constraints, the proposed parallel SimE algorithm with Random Row Distribution should be favored.

Acknowledgment

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

7. REFERENCES

[1] Sadiq M. Sait and Habib Youssef. *Iterative Computer Algorithms and their Application to Engineering*. IEEE Computer Society Press, December 1999.

[2] Sadiq M. Sait and Habib Youssef. *VLSI Physical Design Automation: Theory and Practice*. World Scientific, Singapore, 2001.

[3] Ralph M. Kling and Prithviraj Banerjee. ESP: A new standard cell placement package using simulated evolution. *Proceedings of 24th Design Automation Conference*, pages 60–66, 1987.

[4] Prithviraj Banerjee. *Parallel Algorithms for VLSI Computer-Aided Design*. Prentice Hall International, 1994.

[5] M. Toulouse, T. G. Crainic, and M. Gendreau. Issues in Designing Parallel and Distributed search Algorithms for Discrete Optimization Problems. *Publication CRT-96-36, Centre de recherche sur les transports, Université de Montréal, Montréal, Canada*, 1996.

[6] John A. Chandy, Sungho Kim, Balkrishna Ramkumar, Steven Parkes, and Prithviraj 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*, 16:398–410, April 1997.

[7] Sadiq M. Sait, Mahmood R. Minhas, and Junaid A. Khan. Performance and low-power driven VLSI standard cell placement using tabu search. *Proceedings of the 2002 Congress on Evolutionary Computation*, 1:372–377, May 2002.

[8] Junaid A. Khan, Sadiq M. Sait, and Mahmood R. Minhas. Fuzzy Biasless Simulated Evolution for Multiobjective VLSI Placement. *IEEE Congress on Evolutionary Computation, CEC2002, Honolulu*, May 2002.

[9] C. A. C. Coello. A comprehensive survey of evolutionary-based multiobjective optimization. *Knowledge and Information Systems*, 1(3):269–308, 1999.

[10] Ronald R. Yager. On ordered weighted averaging aggregation operators in multicriteria decision making. *IEEE Transaction on Systems, MAN, and Cybernetics*, 18(1), January 1988.

[11] Van-Dat Cung, Simone L. Martins, Celso C. Riberio, and Catherine Roucairol. Strategies for the Parallel Implementation of metaheuristics. *Essays and Surveys in Metaheuristics*, pages 263–308, Kluwer 2001.

[12] I. De Falco, R. Del Balio, E. Tarantino, and R. Vaccaro. Improving search by incorporating evolution principles in parallel tabu search. In *Proc. of the first IEEE Conference on Evolutionary Computation- CEC'94*, pages 823–828, June 1994.

[13] Bruno-Laurent Garica, Jean-Yves Potvin, and Jean-Marc Rousseau. A parallel implementation of the tabu search heuristic for vehicle routing problems with time window constraints. *Computers & Operations Research*, 21(9):1025–1033, November 1994.

[14] M. Malek, M. Guruswamy, M. Pandya, and H. Owens. Serial and parallel simulated annealing and tabu search algorithms for the traveling salesman problem. *Annals of Ops. Res.*, 21:59–84, 1989.

[15] J. Chakrapani and J. Skorin-Kapov. Massively parallel tabu search for the quadratic assignment problem. *Annals of Operations Research*, 41:327–341, 1993.

[16] E. Taillard. Robust tabu search for the quadratic assignment problem. *Parallel Computing*, 17:443–455, 1991.

[17] T. G. Crainic, M. Toulouse, and M. Gendreau. Towards a taxonomy of parallel tabu search heuristics. *INFORMS Journal of Computing*, 9(1):61–72, 1997.

[18] Erick Cant-Paz. A survey of parallel genetic algorithms. *Calculateurs Parallles, Reseaux et Systems Repartis*, 1998.