Assessment of Parallelization Strategies of Metaheuristics for Linear Speed-up while Maintaining Quality

Mustafa Imran Ali, *ID# 230203*, Ali Mustafa Zaidi, *ID# 230201*

***Abstract*— Non-deterministic iterative heuristics such as Simulated Evolution (SimE) and Simulated Annealing (SA) are widely utilized to solve a range of hard optimization problems [SY99], [Ban94], [Ban94]. This interest is attributed to their generality, ease of implementation, and their ability to deliver high quality results. However, depending on the size of the problem, such heuristics may have very large runtime requirements. One practical approach to speeding up their execution is parallelization. In this paper, we present an assessment of various parallelization strategies for both Simulated Evolution and Simulated Annealing as applied to the VLSI Standard Cell Placement problem. Our aim is to evaluate the potential of these strategies for providing near-linear speedups, while sustaining the serial quality. We identify the impact of various issues on the successful parallelization of these algorithms. Issues under consideration are the problem instance being dealt with, characteristics of the heuristic algorithms themselves, and the implementation environment. Based on our findings we also propose some enhancements to some of the strategies.**

I. Introduction

GENERAL Stochastic Heuristics such as Simulated Annealing and Simulated Evolution are getting more widely adopted to obtain near optimal solutions to numerous hard problems [SY99]. For small problems, stochastic heuristics have reasonable runtime requirements. However, for most real-world problems, the run-time requirements of these algorithms tend to grow dramatically. One way to adapt iterative techniques to solve large problems and traverse larger search spaces in reasonable time is to resort to parallelization [Ban94], [CMRR01]. The eventual goal being to achieve either much 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.

Achieving the goals of parallelization requires proper partitioning of the problem for a uniform distribution of computationally intensive tasks, while respecting the underlying implementation environment. The tractability of this is largely dependent on both the parallelizability of the cost computation and perturbation functions. But most importantly, achieving good quality solutions requires a thorough and intelligent traversal of a complex search space. Thus, for a parallelization strategy to be truly effective, its interaction with the *'intelligence'* of the heuristic must be considered, as it directly affects the final solution quality obtainable, and also indirectly the runtime because of the time required for convergence.

In this paper, we explore and develop various parallelization strategies for SA and SimE with the aim of achieving near-linear speedups, while maintaining the solution quality achieved by the serial versions of these algorithms. Furthermore, we present a qualitative analysis of the interaction between the chosen parallelization strategies and iterative heuristics, as a means of understanding the observed behavior.

For our problem instance, we use the multi-objective standard cell placement problem. We present results for the serial versions of both algorithms as well as for several parallel strategies. Results are quantified in terms of runtimes and placement solution qualities achieved for our test circuits.

Studies of parallelization strategies for metaheuristics have been undertaken previously [CMRR01], [TC02]. In this paper, we use the classification presented in [TC02], which broadly classifies all types of parallelization attempted according to the source of parallelism exploited. The three types of parallelism for heuristics identified are:

1) Low-Level Parallelization (Type 1): The operations within an iteration of the solution method can be parallelized.
2) Parallelization by Domain Decomposition (Type 2): The search space (problem domain) is divided and assigned to different processors.
3) Multithreaded or Parallel Search (Type 3): Parallelism is implemented as a multiple concurrent exploration of the solution space using search threads with various degrees of synchronization.

This paper is organized as follows. In Section II we give a brief description of the Simulated Annealing and Simulated Evolution algorithms. We then present a formal description of our combinatorial optimization problem and cost functions in Section III. Section IV gives details of the implemented parallelization strategies, while experiments and results are discussed in Section V. This is followed by a detailed analysis of the parallelization techniques and in Section VI and finally we conclude in Section VII.

Manuscript submitted June 12, 2005.

This work was done as part of course work requirements for Parallel and Vector Architectures (CSE-661) Spring 2005, under the supervision of Dr. Mohammad Mudawar.

The authors are Research Assistants with the Computer Engineering Department, College of Computer Sciences & Engineering, King Fahad University of Petroleum & Minerals, Dhahran, Saudi Arabia.

II. Description of Heuristics

A. Simulated Annealing

Simulated Annealing is a general adaptive heuristic for solving combinatorial optimization problems. It operates on a single solution, with each solution consisting of multiple elements. The algorithm has one main loop consisting of three basic steps: Perturbation, Evaluation and Decision. Perturbation involves performing a small change in the current solution. In the context of standard cell placement, perturbation may involve changing the location of one or more cells in the placement, or swapping two cells. Following this, evaluation of the perturbed solution is carried out to determine any change in quality or cost of the solution in terms of the objective(s) being optimized. This is followed by the third and most important step: Decision. It is primarily this step that differentiates Simulated Annealing from both a random search, and a fully greedy constructive heuristic. It is this part of the algorithm that gives SA its hill-climbing ability - or the ability to escape from local minima in the search space.

The decision on whether to accept the new solution generated by the perturb step is made based on the Metropolis Acceptance Criterion. Given the well defined properties of the Simulated Annealing algorithm, the primary impact of this criterion is that initially, when the solution quality is low, the algorithm approximates a random search, and as time progresses, the acceptance criterion tends to become more greedy. Thus, earlier in the optimization process, when solution quality is low, the algorithm is more likely to escape from local minima, but as better solutions are found with time, increased greediness/reduced randomness allows the algorithm to explore the neighborhood of the last good solution more thoroughly. The pseudo-code of the Simulated Annealing Algorithm, and its Metropolis function is given in Figure 1 and 2 respectively.

Algorithm Simulated_annealing($S_0, T_0, \alpha, \beta, M, Maxtime$);
(*S_0 is the initial solution *)
(*BestS is the best solution *)
(*T_0 is the initial temperature *)
(*α is the cooling rate *)
(*β a constant *)
(*$Maxtime$ is the total allowed time for the annealing process *)
(*M represents the time until the next parameter update *)

Begin
$T = T_0$;
$CurS$=S_0;
$BestS$=$CurS$; /* $BestS$ is the best solution seen so far */
$CurCost$=*Cost*($CurS$);
$BestCost$=*Cost*($BestS$);
$Time = 0$;
Repeat
Call Metropolis($CurS$, $CurCost$, $BestS$, $BestCost$, T, M);
$Time = Time + M$;
$T = \alpha T$;
$M = \beta M$
Until ($Time \geq MaxTime$);
Return ($BestS$)
End. (*of Simulated_annealing*)

Fig. 1. Procedure for simulated annealing algorithm.

Algorithm Metropolis($CurS$, $CurCost$, $BestS$, $BestCost$, T, M);
Begin
Repeat
$NewS$= $Neighbor$($CurS$);
$NewCost$=*Cost*($NewS$);
$\Delta Cost$=($NewCost - CurCost$);
If ($\Delta Cost < 0$) **Then**
$CurS$=$NewS$;
If $NewCost < BestCost$ **Then**
$BestS$= $NewS$
EndIf
Else
If ($RANDOM < e^{-\Delta Cost/T}$) **Then**
$CurS$=$NewS$;
EndIf
EndIf
$M = M - 1$
Until ($M = 0$)
End. (*of Metropolis*)

Fig. 2. The Metropolis procedure.

B. Simulated Evolution

Simulated Evolution algorithm (SimE) is a general search strategy for solving a variety of combinatorial optimization problems [KB89]. Simulated Evolution was proposed by Kling and Banerjee in 1987 and is based on an analogy with the principles of natural selection thought to be followed by various species in their biological environments [KB89]. It is an attempt in a series of efforts to design better randomized iterative optimization algorithms that are based on more elaborate heuristic knowledge, which should allow the newly designed algorithm to exhibit superior performance to that of simulated annealing with respect to run time requirements and/or quality of solution.

SimE operates on a single solution, termed as $population$. Each population consists of elements. For instance, in case of the placement problem, these elements are cells to be moved. The algorithm has one main loop consisting of three basic steps, Evaluation, Selection and Allocation.

In the *Evaluation* step, *goodness* of each element is measured as a single number between '0' and '1', which is an indicator of how near the element is from its optimal location. A higher value of $goodness$ means that the element is near its optimal location.

Then comes *Selection*, which is the process of selecting elements which are unfit (badly placed) in the current solution. For that purpose, the goodness of each individual is compared with a random number (in the range [0,1]); if the goodness is less than the random number then the cell is selected. Therefore, an individual having high goodness measure still has a non-zero probability of being *selected*. It is this element of non-determinism that gives SimE the capability of escaping local minima. The last step, *Allocation*, has the most impact on the quality of solution. Its main function is to mutate the population by altering the location of selected cells.

The above three steps are executed in sequence until no no-

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. 3. Structure of the Simulated Evolution Algorithm.

ticeable improvement to the population goodness is observed after a number of iterations, or a fixed number of iterations are completed. The pseudo-code of SimE is similar to that given in Figure 3 [SY99]. Although the illustration depicts the slave process to be discussed later, if the entire set of rows is allocated to a single processor, then the execution of the algorithm is the same as that of the serial SimE.

Similar to simulated annealing algorithm, SimE is a very sound approximation algorithm. It is a general algorithm that is relatively easy to apply to almost any combinatorial optimization problem. However, SimE seems to be more greedy than simulated annealing, thus allowing to reach near-optimal solutions in lesser time than simulated annealing. This claim has been supported by experimental results [KB89].

III. The Placement Optimization Problem and Cost Functions

In this section, we describe the problem and the cost functions for the three objectives and the constraint. Our placement optimization problem is of a multiobjective nature with three design objectives namely, interconnect wire-length, power consumption, and timing performance (delay). The layout width is taken as a constraint. The aggregate cost of the solution is computed using fuzzy rules.

Wire-length Cost: Interconnect Wire length of each net in the circuit is estimated using Steiner tree approximation. Total wire length is computed by adding all these individual estimates:

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

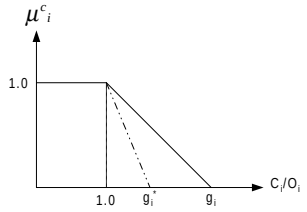


Fig. 4. Membership functions

where l_i is the wire length 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 wire length 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 belonging to that path and the switching delay of the cells driving these nets. The delay T_π of a path π consisting of k nets 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 i and ID_i is the interconnect delay of net i. Delay cost is determined by the delay along the longest path in a circuit.

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

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 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:

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

Fuzzy Aggregate Cost Function: We used fuzzy logic for designing an aggregating cost function, allowing 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 is used:
Rule 1: IF a solution has *SMALL wire length* **AND** *LOW power consumption* **AND** *SHORT delay* **THEN** it is a *GOOD* solution.

The above rule is translated to *and-like* OWA fuzzy operator [Yag88] and the membership $\mu(x)$ of a solution x in fuzzy set *GOOD solution* is given as:

$$\mu(x) = \begin{cases} \beta \cdot \min_{j=p,d,l} \{\mu_j(x)\} + (1-\beta) \cdot \frac{1}{3} \sum_{j=p,d,l} \mu_j(x); \\ \qquad \text{if} \quad Width - w_{avg} \leq \alpha \cdot w_{avg}, \\ 0; \qquad \text{otherwise.} \end{cases} \tag{8}$$

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 wire length* 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 wire length* are shown in Figure 4.

IV. Parallel Strategies

A. Parallel Simulated Annealing Approaches

For SA [SY99] we implement several variations of the Type 3 parallel search approach using asynchronous multiple Markov chains (AMMC) strategy described in [LL96] and applied to the VLSI standard cell placement problem in [CRPB97]. The basic structure of this approach is given in Figure 5. On each available processing element, an SA operation is initiated with the same starting solution, but with different seeds for pseudo-randomizaton. Given below are the specifications of our parallel search implementation of SA:

1) **The Information exchanged:** The entire recent best solution is communicated to slave processes.
2) **Connection Topology:** The parallel processes communicate via a central solution storage area, where the best solution found so far is kept. The master process is reserved for this purpose.
3) **Communication Mode:** Communication is asynchronous. Thus communication time is minimized since there are no synchronization barriers.
4) **Time to Exchange Information:** Each process works on a recent best solution retrieved from the central store for the duration of its Metropolis loop. This can be a varying number of perturbation-decision steps at a constant temperature.

We implement four distinct versions of the AMMC approach: For strategy 1, aside from the above points, there is no difference between the serial version and each of the parallel search processes. Such an approach has been found to improve solution qualities in a fixed amount of time [LL96], and our results corroborate this fact. Strategy 2 however, is an attempt to provide near linear speedup over the serial version. This is accomplished by dividing the number of Metropolis iterations at each process by the total number of processes. Strategy 3 builds upon strategy 2 by implementing different cooling schedules on each processor. this is done in such a way that some of the processors are searching every solution in a greedy manner, while others are still in the high temperature region.

Based on what we have learned from the last three parallel search SA strategies, we proposed certain modifications to

Algorithm Parallel_Simulated_Annealing($S0, T0, \alpha, \beta, M$, $Maxtime, my\_rank, p$)
Notation
(* $S0$ is the initial solution. *)
(* $BestS$ is the best solution. *)
(* $T0$ is the initial temperature. *)
(* α is the cooling rate. *)
(* M is the time until next parameter update. *)
(* $Maxtime$ is the total allowed time for the annealing process. *)
(* my_rank is rank of current process;0 for master,!0 for slaves. *)
(* p is the total number of running processes. *)
Begin
T = T0;
CurS = S0; // only master has the initial Solution
BestS = CurS;
CurCost = Cost(CurS);
BestCost = Cost(BestS);
Time = 0;
If (my_rank == 0) // i.e. Master process
Broadcast(CurS);
Endif

If (my_rank ! = 0) // i.e. Slave process
Repeat
Call Metropolis(CurS, CurCost, BestS, BestCost, T, M);
Time = Time + M;
T = α T;
M = β M;
Send_to_Master(BestCost);
Receive_frm_Master(verdict);
If (verdict == 1)
Send_to_Master (BestS);
Else
Receive_frm_Master(BestS);
EndIf
Until (Time $\geq$ Maxtime);
EndIf
If (my_rank == 0) // i.e. Master process
Repeat
Receive_frm_Slave(BestCost);
Send_to_Slave(verdict);
If (verdict == 1)
Receive_frm_Slave(BestS);
Else
Send_to_Slave (BestS);
EndIf
Until (All Slaves are done);
Return(BestS);
EndIf
End. (*Parallel_Simulated_Annealing*)

Fig. 5. Procedure for Parallel Simulated Annealing using Asynchronous MMC

the cooling schedule of our basic, serial Simulated Annealing algorithm. This adaptive cooling schedule, when implemented for the Type 3 scheme, but using a different set of parameters, yielded our 4th parallel search SA strategy. A brief description of the adaptive cooling schedule is given below:

1) Maintain constant rate of solution quality improvement per Metropolis loop
2) If rate of improvement drops below a certain threshold, increase M incrementally, since not enough time is being spent at each temperature level
3) If rate of improvement is more than 1.5 times the threshold value, decrease M, since an unnecessary amount of

time is being spent at the given quality level.

4) If solution is saturated, i.e. there have been no improvements for the past 'n' iterations, increase the temperature by 10%

Aside from the above-mentioned Type 3 strategies, we have also implemented a Type 1 Parallel SA that divides the cost computation workload over multiple processors. This scheme however is only able to implement the wire length and power cost functions, as they are easily divisible, while the delay cost function has been excluded. The reason for this is that due to the complex interdependencies that needed to be observed to calculate delay meant that a significant number of computations would have to be replicated on each of the processors, thereby negating any speedup observed.

B. Parallel Simulated Evolution

All the three parallelization types have been attempted for Simulated Evolution. For Type 2, three different strategies were implemented, the first being the baseline domain decomposition, which is a variation of one that was originally proposed in [KB89]. The second is the communication optimized version (strategy 2) and while third is communication as well as cost computation optimized (strategy 3) variant of the first. The communication optimization is achieved by reducing the amount of data transferred by taking advantage of the sparse nature of solution matrix data structure. For cost computation division, only the wire length and power computations were targeted because delay computations involve complex dependencies among cell partitions which reduce the potential benefits of such a division. Type 3 attempts a parallel search implementation along the same lines as strategy 1 of parallel SA. Type 1 targets cost computation and goodness computation division among processors, while retaining the *selection* and *allocation* operations at the master.

Unlike Simulated Annealing , Genetic Algorithms and Tabu Search , the parallelization of SimE has not been the subject of much research. Kling and Banerjee suggested a distributed memory MIMD parallel approach for speeding up the SimE algorithm (Type 2 approach) [KB89]. Each process is assigned a number of rows of the placement problem in an alternating block and cyclic row assignment order. Each process executes one iteration of the SimE algorithm on the rows assigned to it. After each iteration, the rows are redistributed among the processors [KB89]. This row assignment allows any cell to migrate to its best desired location in at most two iterations. Our implementation is based on a random-row allocation pattern instead of the original. The structure of Type 2 SimE is given in Figure 6.

For Type 3, the specifications are identical to those of Strategy 1 for parallel SA, save for the fact that information is exchanged between master and slave after every iteration at the slave.

```
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 Type 2 SimE Algorithm.

V. Experiments and Results

In this section, we first describe our experimental environment followed by a detailed description of results.

A. Experimental Setup

The experimental setup consists of a homogenous cluster of 8 Pentium-4 2 GHz machines, and 256 MB of memory. These machines are connected by fast ethernet switch. Operating system used is Redhat Linux 7.3 (kernel 2.4.7-10). The algorithms were implemented in C/C++, using MPICH ver. 1.2.4. ISCAS-89 circuits are used as performance benchmarks for evaluating the parallel metaheuristics.

B. Results: Parallel Simulated Annealing

Table I shows the results obtained from experiments with Strategy 1 for the benchmark circuits listed in column 1. The third column lists the highest quality achieved by the serial version of the algorithm (reference optimal quality = 1.00). The remaining columns list the time taken to achieve the specified quality, with the given number of processors. Using strategy 1, we were always able to exceed the quality achieved by the serial version. Figure 7 (a) shows the speedups achieved by strategy 1, for the same quality, with different number of processors and for different circuits. Here we see that speedup achieved using strategy 1 is poor. Even with 8 processors, we are unable to even achieve a speedup of 3.

Table II shows the results obtained from experiments with strategies 2 and 3. Unlike the previous table, the third column here shows the highest common quality that could be achieved by multiple runs of strategies 2 and 3 for every number of processors. Comparing with column 3 of Table I, we can easily note that there is a roughly 10% drop in achievable quality with this scheme. Figure 7 (b) shows the speedups achieved by strategy 2 as the number of processors is varied. We see that in this case, speedup is almost linear. Note however, that there is no significant improvement in quality between Strategy 2 and 3.

Table **??** shows the results obtained for s1196 by the Type 1 parallel SA scheme. For the parallel runs, we see that although qualities achieved are identical to the serial run, the runtime results are extremely poor in this case - worse than the serial

TABLE I

ASYNCHRONOUS MMC PARALLEL SA RESULTS FOR STRATEGY 1

Circuit Name	Number of Cells	$\mu(s)$ SA	Time for Serial SA	Time for Parallel SA Strategy 1					
				p=3	p=4	p=5	p=6	p=7	p=8
s1196	561	0.675340	190	145.98	130.95	110.31	96.98	98.24	94.89
s1238	540	0.699469	212	183.91	130.32	127.55	117.12	114.66	111.58

TABLE II

ASYNCHRONOUS MMC RESULTS PARALLEL RESULTS FOR STRATEGY 2

Circuit Name	Number of Cells	$\mu(s)$ SA	Time for Serial SA	Time for Parallel SA Strategy 2					
				p=3	p=4	p=5	p=6	p=7	p=8
s1196	561	0.630367	103	44.67	31.32	22.81	18.47	16.46	14.42
s1238	540	0.630573	117	58.03	39.21	26.31	22.31	19.73	15.83

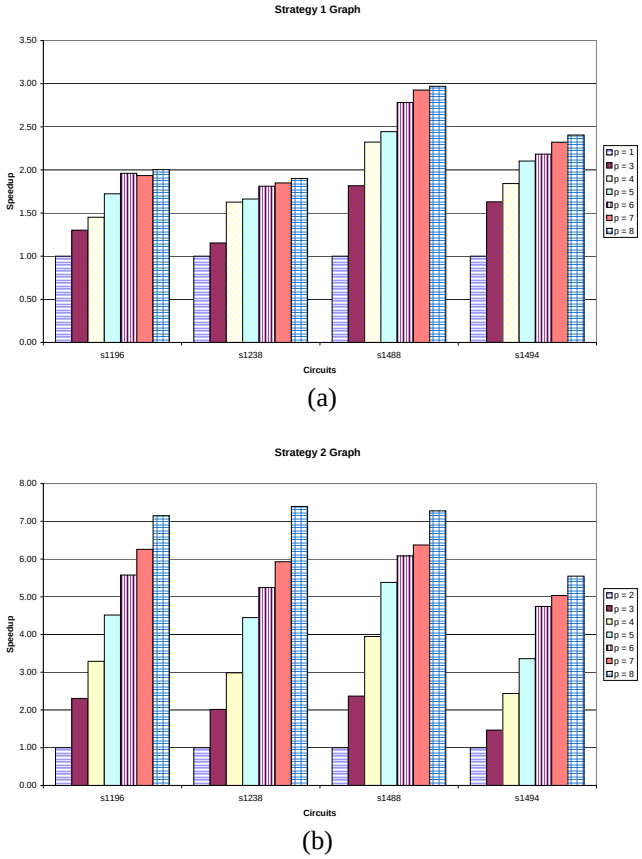


Fig. 7. Speedup versus number of machines for (a) Parallel SA AMMC Strategy 1; (b) Parallel SA AMMC Strategy 2.

runtime by orders of magnitude. Due to lack of time, we have included the runtime results of only 2 runs: parallel with 2 processors, and parallel with 7 processors.

From these results, it became evident that for Simulated Annealing, if any progress is to be made towards achieving our goals of linear run times with sustained quality, an in depth study of the impact of parameter M on achievable solution quality is required. To this end, we ran several experiments on both the serial and parallel (7 processors) versions of Type 3 Parallel SA, keeping all things constant except M, which was divided by 1, 9, 17, 25, and 57 respectively for each new run. Results of the Serial version are given in Figure 8(a), with a close up of the top left region of the curve shown in Figure 8(b). Results for similar Runs of the Type 3 parallel SA on 7 processors are given in Figure 9(a), with a closeup of the active region given in Figure 8(b).

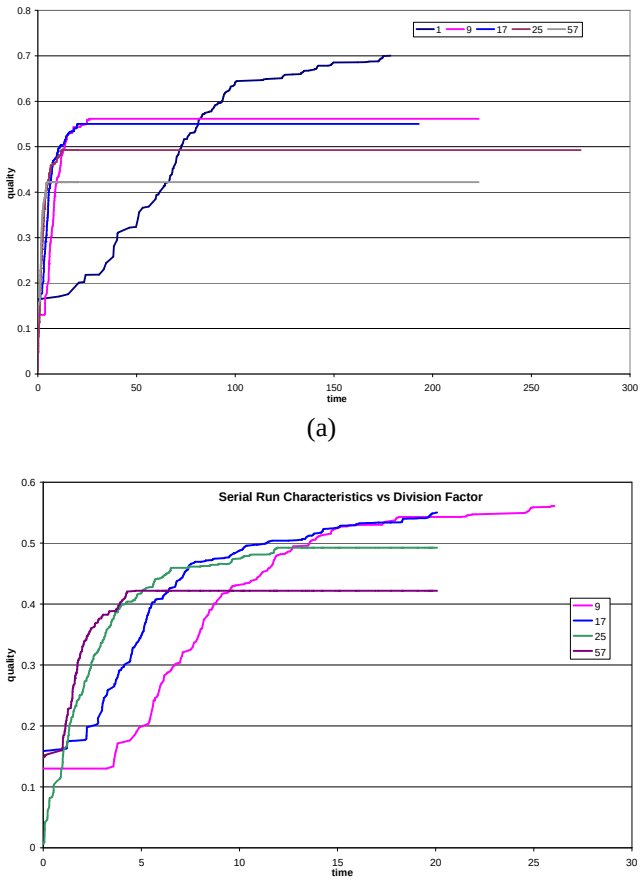


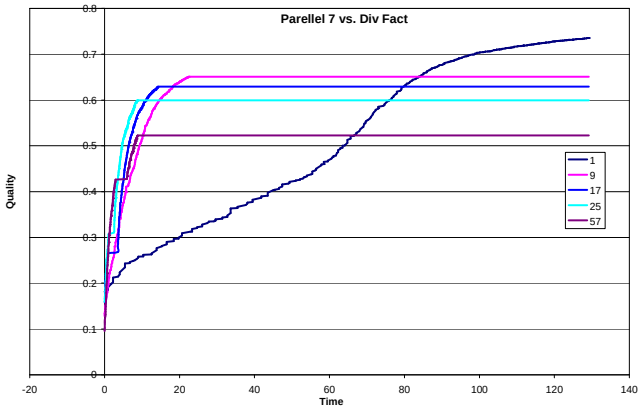
Fig. 8. Quality vs. Runtime results for Serial SA, with different values for M

From the above results, we can see that for both the serial and parallel versions, division of M by a larger number increases the rate at which new solutions are found, at least initially. However, this has the adverse effect that the final solution quality achieved is inversely proportional to the division factor. Intuitively this would suggest that the M factor should start at a small value, and then should increase as solution quality rises. However, a balance is necessary: if M

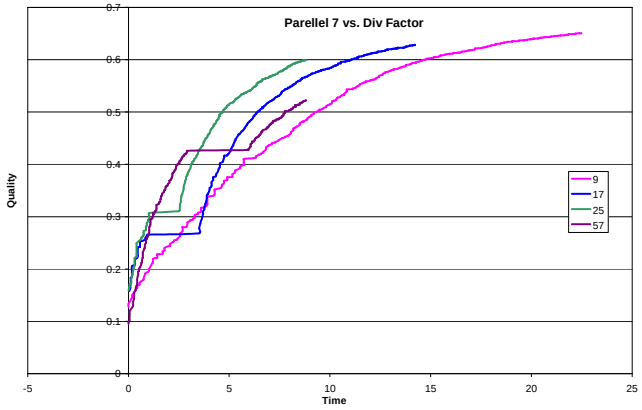
TABLE III
TYPE 1 PARALLEL SA RUNTIME AND QUALITY RESULTS

Circuit Name	Number of Cells	$\mu(s)$ SA	Time for Serial SA	Time for Parallel SA Strategy 2					
				p=2	p=3	p=4	p=5	p=6	p=7
s1196	561	0.566007	42.886	1816.4	-	-	-	-	2216.6

increases too fast, runtime is compromised; if M increases too slowly, achievable solution quality is affected. the answer to this comes from an key observation mad during these runs: during the steep improvement phase the rate of improvements to solution quality is constant per metropolis call - meaning that during the initial phase, the high rate of climb is primarily due to the short time spent in each metropolis call.



(a)



(b)

Fig. 9. Quality vs. Runtime results for Type 3 Parallel SA (7 processors), with different values for M

An adaptive scheme for maintaining the cooling schedule was developed for the serial SA based on the above observations. This is compared with a Type 3 parallel SA implementation with 7 processors in Figure 10. What we see from this comparison is that as the both the search processes approach higher and higher solutions, the gap horizontal gap between the two curves diminishes. We understand this to be a clear indication that as higher and higher qualities are targeted, the potential for speedup diminishes such that for any number of processors, achieving the optimal cost would yield virtually no speedup.

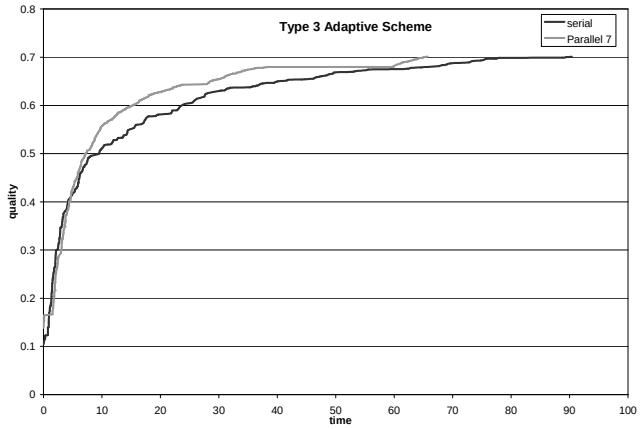


Fig. 10. Solution Quality vs Runtime Trends for Parallel and Serial Adaptive SA

C. Results: Parallel Simulated Evolution

The run times to achieve a fixed solution quality for serial, Type 2 baseline, Type 2 Strategy 2 and Type 3 Strategy 3 parallel SimE are shown in Table IV. Since the qualities achieved by the Type 2 random-row distribution approaches are consistently 10 to 20% below the serial quality, all time values are shown for the highest quality common across all runs. Results are shown only for up to 5 processors since no more processors can be used with the given circuits because of no greater than 10 rows are available to divide among processor. For all the Type II strategies it can be seen that for the chosen circuits, which are relatively small, there is speed-up observed will 3 processors, beyond which the runtimes for all strategies begin to rise up. The effect of communication optimization is visible in lower run times for Strategy 2 compared to baseline, while run times of Strategy 3 show that the division of cost has not resulted in noticeable gains.

The results of Type I implementation are shown in Table VI. The qualities given are the highest achieved by the serial algorithm, which are the same as that achieved by Type I parallel SimE across all processors. For the chosen circuits, the run times keep decreasing till 5 processors , beyond which there is a stagnation of further gains. Also, the speed-ups are non-linear.

The solution qualities obtained by the Type 3 parallel SimE are consistently lower than those of the serial version, while the runtime values are largely inconsistent and show no discernible trend. We discuss and attempt to understand this behavior as well as results of other parallelization types in section **??**.

TABLE IV

RESULTS FOR THE THREE TYPE II PARALLEL SIME STRATEGIES.

Circuit Name	Number of Cells	$\mu(s)$ SimE	Time for Sequential SimE	Times for Baseline Type II		Times for Strategy 2		Times for Strategy 3	
				p=3	p=5	p=3	p=5	p=3	p=5
s1196	561	0.644	54	44	50	34	44	30	41
s1238	540	0.680	60	45	55	39	46	34	42

TABLE V

RESULTS FOR TYPE 1 PARALLEL SIME

Circuit Name	Number of Cells	$\mu(s)$ SimE	Time for Serial SimE	Time for Parallel SimE Type 1		
				p=3	p=5	p=7
s1196	561	0.762	67	53	44	45
s1238	540	0.799	72	49	35	40

TABLE VI

RESULTS FOR TYPE 3 PARALLEL SIME

Circuit Name	Number of Cells	$\mu(s)$ SimE	Time for Serial SimE	Time for Parallel SimE Type 3		
				p=3	p=5	p=7
s1196	561	0.694	58	57	60	61
s1238	540	0.709	65	66	64	67

VI. DISCUSSION & ANALYSIS

The basic goals of parallelization for an iterative heuristic are either the achievement of higher quality for the same runtime, or achieving near-linear speedup for a given quality. For effective parallelization of an iterative heuristic, such that the goals of parallelization are achieved, it is essential to take into account the interaction of the parallelization scheme with: 1) Parallelizability of the solution perturbation operation 2) Parallelizability of the solution quality/cost computation function 3) Characteristics of the parallel environment, and most importantly 4) The intelligence of the heuristic. In this section, we present an analysis of all the results generated from our parallel implementations of these algorithms, with respect to the above factors.

In order to properly evaluate our parallelization schemes and understand the results obtained, we must consider them in light of each of these issues.

A. Cost Computation Function

For the Multi-objective VLSI standard-cell placement problem, computation of solution quality involves individual computation of overall wire-length, delay, and power metrics, followed by their combination using a fuzzy operation (Section **??**). Computing this multi-objective cost function requires the most recent state of the solution to be accurate. Due to inter-dependencies among cells in the netlist, the computation cannot be neatly divided and computed on portions of the solution without operating on other cells not in current the partition. This is specially true for delay computation which takes place on long paths that span across row boundaries and the goodness calculation. For baseline Type 2 parallel SimE, this results in an inefficient division of workload as the work done per processor is not significantly reduced with increase in number of processors. The primary impact of this can be seen in the lack of run-time scalability. Division of cost computation for wire length and power was attempted in Strategy 2 for Type 2. Delay was omitted due to its more complex dependencies resulting in poor gain, if any, by partitioning delay computation The results show minor gains implying that effect of wire length only division coupled with increased communication per iteration for communicating partial costs to all doesn't add up much. Type 1 implementation on the basis of cost and goodness computation division shows speed-ups that are not linear and are limited by circuit sizes with increasing processors.

The Type 3 scheme adopted for both heuristics is immune to this issue, since aside from the sparse solution exchanges, each processing element is undertaking an independent but complete search operation. This means that although the cost computation functions remain undivided, the operate on largely different solutions on different processors, and thus give equivalent performance to the serial algorithm. This assessment is verified from experimental results for all Type 3 versions of parallel SA, but not for the parallel-search SimE. Given that all things are equal amongst the three parallel search implementations vis a vis the cost computation function, it is reasonable to assume that the inconsistent runtime behavior of SimE is most likely caused by one of the other factors discussed below, and not this one.

For the Type 1 schemes for both Simulated Annealing and Simulated Evolution, we have excluded the Delay computation function as it was found to introduce duplicate computations on each processor and thus potentially negating any benefit that would be obtained from the workload division. Now, since computation of wire length and power can easily be partitioned over multiple processors, it is expected that if communication delays are ignored, some speedup should be observed over the serial version. The fact that this is not the case for Type 1 SA has nothing to do with the divisibility of the cost function, as it is perfectly divisible now. Rather, trouble stems from the nature of the parallel environment, as we shall see in the next subsection.

B. Parallelization Environment

In our cluster-of-workstations operating environment, it is essential to minimize the amount of communication in relation to the computation. For Type 2 parallel SimE, barrier synchronization is used and occurs frequently (after every iteration). Thus as the number of processing elements increases, the number of messages to be transmitted increases linearly. Furthermore, with larger benchmark circuits, the size of each message increases as well. This has a direct adverse impact on the run time characteristics of the implementation.

Strategy 2 & 3 of Type 2 SimE tries to mitigate this problem by reducing the data communicated by taking advantage of sparse nature of solution matrix data structure. The results indicate improved performance but trends with increasing processors mentioned before still persist, leading to poor runtime scaling. The high communication times are also a dominating factor in poor speed-ups achieved with Type 1 SimE.

The periodic, asynchronous communication model used for both Type 3 parallel algorithms (all SA and SimE) ensures that communication delays are minimized, and occur only when necessary. Thus the impact of communication delays on the runtime performance of these approaches is minimal. This can be verified from Figure 11, which shows the ratio of communication time to computation time for Type 3 parallel SA strategy 2, when run on 7 processors for circuit s1196. Again we observe that the reason for the inconsistent runtime behavior of Type 3 parallel SimE must not be the parallelization environment.

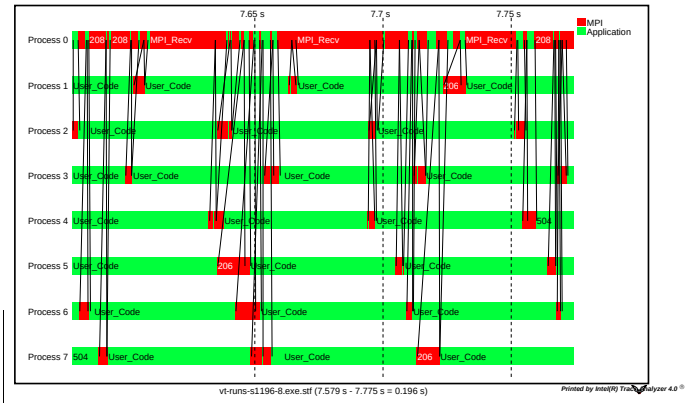
As for the Type 1 parallel SA approach, we get the results shown in Figure 12. We now understand the reason for such poor performance of the Type 1 Parallel SA implementation - communication time drastically overshadows the computation time. There are three reasons for this: (1) Communications are too frequent: each Metropolis call communication to take place because cost of solution is computed once every Metropolis iteration; (2) Two collective communication calls: one to send updates to slaves, and the next to retrieve computed costs from slaves - this use of barrier communication significantly increases the average waiting time for all processes; (3) The amount of work being actually divided is very small as compared to the communication time required.

All three factors combined incur an enormous communication overhead at each Metropolis Iteration. Thus the primary reason for such poor performance being obtained from the Type 1 version is its unsuitability to the distributed memory environment, where communication latencies are so large that communication must only be very sparse, as is the case with the Type 3 parallel implementations of both algorithms.

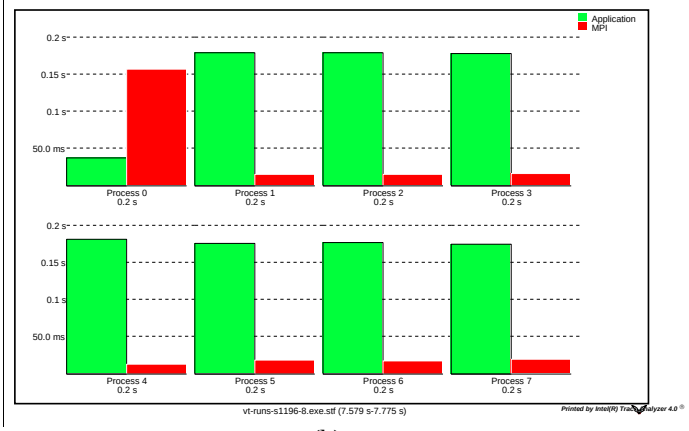
Both of the factors discussed above have a negative impact on runtime scalability, while achievable quality remains largely unaffected.

C. Solution Perturbation and Algorithmic Intelligence

The solution perturbation and next-solution selection operations are where the intelligence of virtually all stochastic



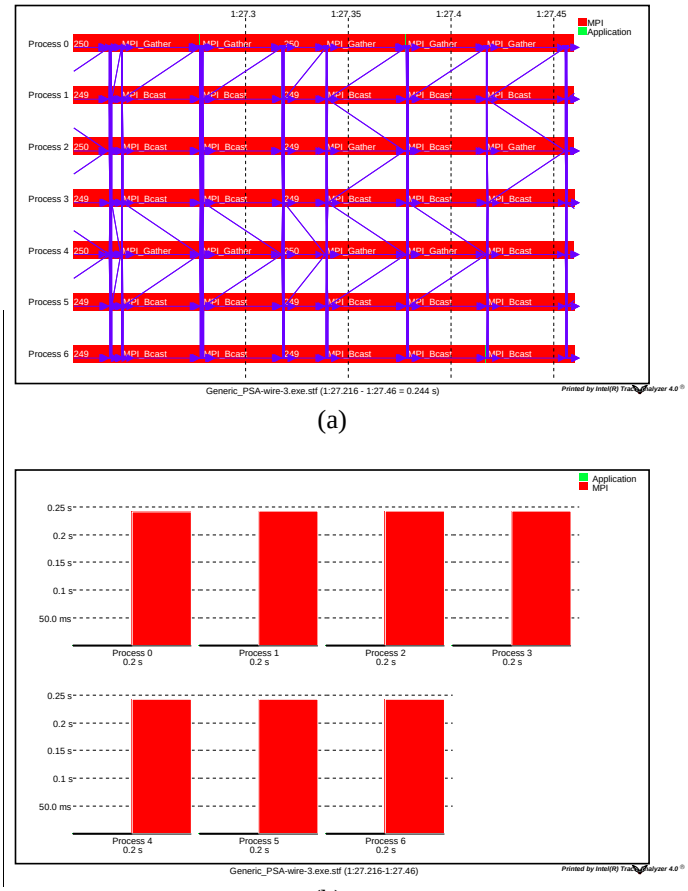
(a)



(b)

Fig. 11. (a) Communication vs. Computation Traces for all processors (b) Ratio of Communication to Computation for each processor

heuristics lies. In SimE both evaluation and selection of individual cells may be carried out independently of all the other cells. The allocation step, however, needs cells to be allocated in a certain order, thereby introducing a sequential dependence. In the domain-decomposition scheme, division of workload is achieved by dividing the whole solution (i.e. the placement) into distinct, roughly equal sized portions, each of which is treated as a complete solution in itself. This allows the division of the allocation function without causing inconsistencies in the solution [KB89]. The drawback of such a division is the lack of a global placement view for each of the nodes during the allocation step. Since parallelization of SimE is achieved by dividing the solution between PEs, movement of individual cells across the solution during the allocation step is restricted due to the lack of global placement view for each PE. This represents a division of the heuristic intelligence and has an adverse effect on algorithm behavior, resulting in a significant drop in the solution quality achievable by the algorithm. Even our attempt to improve cell movement to different parts of the placement by using the random-row distribution scheme falls short of providing solution qualities equivalent to the serial SimE algorithm. In an attempt to maintain solution quality, Type 1 was implemented which gave same solution quality as serial with poor speed-ups due to high communication times as mentioned before. Type 3 implementation was expected to



(a)

(b)

Fig. 12. (a) Communication vs. Computation Traces for all processors (b) Ratio of Communication to Computation for each processor

improve upon the quality issue, but the results were contrary and the reasons for such outcome are mentioned below.

On the other hand, the solution perturbation operation in SA is inherently sequential and, in the chosen parallelization schemes (Types 1 and 3), is left undivided. The intelligence of SA lies instead in its cooling-schedule. In Type 3 parallel SA, each independent parallel search chain periodically starts its search from the best available solution at the time. This, coupled with the ability of SA to escape local minima, allows the parallel search to be focused around a recent best solution, which would be the logical place to look for an even better solution. Thus not only does the algorithmic intelligence remain undivided, it is further enhanced using the Asynchronous MMC approach, allowing the achievement of better solutions in the same or lesser amount of time, as is the case for Strategy 1.

As for Strategies 2 and 3, however, we see that although a division of the workload has a positive impact on runtime, there is an adverse impact on achievable quality. This can be understood by looking at how the intelligence of the algorithm is affected by such a division (achieved simply by dividing the cooling-schedule parameter 'M' by the number of processors). Since SA convergence is highly sensitive to the cooling schedule, it is understandable that such a drastic change to one of its parameters would result in lower quality solutions. Division of 'M' reduces the amount of time each processor spends searching for a better solution in the vicinity of a previous good solution, resulting in a less thorough parallel search of the neighboring solution space. Even the proposed enhancement of varying other parameters across other processors, as done in Strategy 3, is insufficient to counteract the impact of divided 'M'.

Strategy 4 of the Type 3 parallel SA approach was implemented after a careful study of the impact of varying M on achievable solution quality. The adaptive nature of the cooling schedule allows this technique to achieve high quality results in significantly reduced runtimes, when compared with the original implemented algorithms. However, compared to the serial SA version with a similar Adaptive cooling schedule, the performance benefits are less significant, in fact more comparable to the difference between Strategy 1 and the original Serial SA - achieving the same quality solution in slightly lesser time.

The Type 1 parallel SA is able to achieve identical qualities to the serial version of the algorithm, despite the severely degraded runtime. The former is because the intelligence of the approach remains undivided, while the latter as we saw is due to its unsuitablilty for the parallel environment.

We finally discuss the reasons for why our implementation of Type 3 parallel SimE failed to exhibit the robustness generally expected of parallel search techniques [TC02]. Since poor quality and unpredictable runtime results cannot be blamed on either the parallel environment, or the cost computation functions, we are left with the one factor that is different among the Type 3 parallel schemes: heuristic intelligence.

There is a key difference between the Type 3 parallel implementation of SimE and SA. Type 3 parallel SA performs multiple iterations, each involving simple random perturbs, between best solution exchanges; whereas Type 3 SimE exchanges the best solution after every iteration, which involves a single, highly aggressive and greedy compound operation. From our experiments, we deduce that greedy nature of the perturb, combined with overly frequent exchanges of the entire recent best solution among processes leads to an 'over-intensification' of the search, making the heuristic very susceptible to becoming stuck at, or drawn towards local minima. This explains why the Type 3 parallel SimE is unsuccessful in achieving solution qualities similar to, or better than its serial version, unlike the other two heuristics. As for the wildly fluctuating runtimes, these could be a result of the conflict between the inherently rapid convergence rate of SimE and the 'over-intensification' described above. For the expected low qualities, which ever factor is more dominant in any particular run of the algorithm eventually determines the final runtime: sometimes convergence is facilitated by the parallel search, leading to super-linear runtimes, while at other times the search may get stuck at local minima resulting in much larger runtimes.

VII. Conclusion

For the Parallel version of SA we find that the most promising approach for a MIMD-DM environment is the

adaptive Type 3 approach. This can be attributed to the inherent robustness of this scheme, as it is largely immune to any complications arising from either the cost computation function or the parallelization environment. However, despite the observed qualities and improvements of this scheme, we find that a direct trade off exists between achievable solution quality and speedup. This is because as solution quality approaches the optimum value, any further improvements become increasingly difficult to achieve. This means that it is possible to get linear, or even super linear speedups in certain instances, as long as the quality target specified is low enough. But after a certain threshold, speedup inevitably starts to diminish, often to such an extent that for any number of processors, given a high enough quality target, it approaches 1. In case of Parallel SimE approaches we find that only Type 1 is able to attain serial solution qualities but speed-ups are non-linear and do not scale well with increasing processors because of high communication costs and inability to achieve an efficient workload division. In fact, in all SimE schemes that rely on workload division the high communication costs of our parallel environment result in poor runtime performance.

It is clear that parallelization of metaheuristics is a non trivial task. A parallelization scheme should be evaluated in light of the properties of the heuristic (nature of the solution perturbation, interaction with algorithmic intelligence), properties of the problem domain (e.g. complexity/divisibility of cost functions associated with multiobjective standard cell placement), and of the parallelization environment (whether the system is shared or distributed memory). It should be noted that runtime improvements are directly linked to the efficiency of the latter two interactions, while quality improvement or sustenance is primarily determined by the first as shown by the results obtained in this work.

REFERENCES

[Ban94] Prithviraj Banerjee. *Parallel Algorithms for VLSI Computer-Aided Design*. Prentice Hall International, 1994.

[CMRR01] 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.

[CRPB97] 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 CAD*, Vol. 16 No.4, April 1997.

[KB89] R. M. Kling and P. Banerjee. ESP: Placement by Simulated Evolution *IEEE Transactions on Computer-Aided Design*, March 1989, VOLUME 8, NUMBER 3, PAGES 245-255.

[LL96] S.-Y. Lee and K.-G. Lee. Synchronous and Asynchronous Parallel Simulated Annealing with Multiple Markov Chains. *IEEE Transactions on Parallel & Distributed Systems*, 7, October 1996.

[SHAM01] Sadiq M. Sait and Habib Youssef and Aiman H. El-Maleh and Mahmood R. Minhas. Iterative Algorithms for Multiobjective (VLSI) Standard Cell Placement *INNS-IEEE Int'l Joint Conference on Neural Networks*, July 2001.

[SM05] Sadiq M. Sait, and Mahmood R. Minhas. A Parallel Tabu Search Algorithm for Optimizing VLSI Cell Placement. *International Conference on Computational Science and its Applications*, May 2005.

[SMK02] 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.

[SY99] Sadiq M. Sait and Habib Youssef. *Iterative Computer Algorithms and their Application to Engineering*. IEEE Computer Society Press, December 1999.

[Yag88] Ronald R. Yager. On ordered weighted averaging aggregation operators in multicriteria decision making. *IEEE Transaction on Systems, MAN, and Cybernetics*, 18(1), January 1988.

[TC02] M. Toulouse and T. G. Crainic. Parallel Strategies for Metaheuristics *State-of-the-Art Handbook in Metaheuristics*, F. Glover, G. Kochenberger (Eds.), Kluwer Academic Publishers, 2002.