Exploring Asynchronous MMC based Parallel SA Schemes for Multiobjective Cell Placement on a Cluster-of-Workstations

Sadiq M. Sait Ali M. Zaidi Mustafa I. Ali
College of Computer Sciences & Engineering
King Fahd University of Petroleum & Minerals
Dhahran-31261, Saudi Arabia
E-mail: {sadiq, alizaidi, mustafa}@ccse.kfupm.edu.sa

Abstract

Simulated Annealing (SA) is a popular iterative heuristic used to solve a wide variety of combinatorial optimization problems. However, depending on the size of the problem, it may have large run-time requirements. One practical approach to speed up its execution is to parallelize it. In this paper, several parallel SA schemes based on the Asynchronous Multiple-Markov Chain model are explored. We investigate the speedup and solution quality characteristics of each scheme when implemented on an inexpensive cluster of workstations for solving a multi-objective cell placement problem. This problem requires the optimization of conflicting objectives (interconnect wire-length, power dissipation, and timing performance), and Fuzzy logic is used to integrate the costs of these objectives. Our goal is to develop several AMMC based parallel SA schemes and explore their suitability for different objectives: achieving near linear speedups while still meeting solution quality targets, and obtaining higher quality solutions in the least possible duration.

1 Introduction

There is a growing need for obtaining useful/acceptable solutions for combinatorial optimization problems in numerous areas of research and industry. Consequently, there is considerable interest in utilizing iterative stochastic heuristics like Simulated Annealing, that are capable of delivering acceptable or near-optimal solutions to these problems with reasonable runtimes [1]. This is especially true with the often conflicting, multiple objectives that have to be addressed in such problems. However, despite their potential, such heuristics (Simulated Annealing in particular) can still have extremely high runtime requirements if very high solution qualities are required, (or very strong constraints are placed).

One way to adapt iterative techniques such as SA to solve large problems and traverse larger search spaces in reasonable time is to parallelize them [2, 3], with the eventual goal being to achieve either much lower run-times for same quality solutions, or higher quality solutions in a fixed amount of time. From a computational point of view, metaheuristics are algorithms from which functional and data parallelism can be extracted. However, metaheuristics usually operate upon irregular data structures, such as graphs, or upon data with strong dependencies among different operations and as such remain difficult to parallelize using only data and functional parallelism [4]. Furthermore, when parallelizing metaheuristics, not only speed-ups are important but also the maximum achievable qualities. Therefore, to achieve any benefit from parallelization requires not only a proper partitioning of the problem for a uniform distribution of computationally intensive tasks, but more importantly, a thorough and intelligent traversal of a complex search space for achieving good quality solutions. The tractability of the former issue is largely dependent on parallelizability of both the cost computation and perturbation functions while the latter issue requires that the interaction of parallelization strategy with the intelligence of the heuristic must be considered, as it directly affects the final solution quality obtainable, and indirectly the runtime due to its effect on algorithms convergence.

Simulated Annealing Parallelization Issues

The Simulated Annealing Algorithm has an inherent sequential nature since each iteration (consisting of three phases: move, evaluate, decide) depends upon the previous iteration [5, 6]. The decision phase determines what the current solution will be for the start of the next move-evaluate-decide cycle. This inherent sequential nature makes parallelization of this algorithm a non-trivial task.

Parallel Simulated Annealing has been the subject of intensive exploration since it was first proposed. Virtually all known methods of parallelization for Simulated Annealing can be classified into one of two groups: Single Markov-chain and Multiple Markov-chain methods [7]. Most Single Markov chain approaches attempt to exploit parallelism between the three phases. They include move-acceleration, parallel-moves, and speculative annealing and are generally more suitable for shared-memory environments. Approaches based on Multiple Markov chains call for the concurrent execution of separate simulated annealing chains with periodic exchange of solutions [7, 8]. This approach is particularly promising since it has the potential to use parallelism to increase the quality of the solution rather than simply accelerate the algorithm. Theoretically, this approach is not intended to provide speedups since the same amount of work is being done by each processor as in the serial version. However, since a higher fitness solution can be reached in the same amount of time, speedup may be measured as the difference in times taken to achieve the same quality as the serial version. Multiple Markov Chain based parallelization is ideally suited for distributed memory systems, considering that the need for communication between nodes is considerably reduced.

In our work, attempt to solve the multi-objective VLSI standard cell placement problem. We experiment with different versions of the Asynchronous Multiple-Markov Chain Parallel SA (or AMMC PSA) approach described in [7], as this scheme has been found to be well suited to solving this problem in a distributed-memory environment [8]. Our goal is to develop parallel SA implementations that:

1. solve a VLSI standard cell placement problem with multiple, potentially conflicting objectives,

2. are suited for an inexpensive, cluster-of-workstations environment, as opposed to specialized HPC solutions like those utilized in the majority of prior work,

3. can achieve (a) improved quality solutions with runtimes equivalent to the serial algorithm, and/or (b) near-linear speedups without compromising final solution quality.

2 SA Parallelization Strategies in Literature

Several studies of parallelization strategies for meta-heuristics in general have been reported in literature [9, 10]. For our discussion, we use the classification proposed by Toulouse and Crainic [9], which broadly classifies all types of attempted techniques according to how parallel nature is exploited. The three categories of parallel strategies for heuristics are identified as:

1. Low-Level Parallelization (Type 1): The operations within an iteration of the solution method can be parallelized. Such methods seek to divide the computational workload for each iteration across multiple processors, and as a consequence, leave the algorithm characteristics unaffected.

2. Parallelization by Domain Decomposition (Type 2): The search space (problem domain) is divided and assigned to different processors. For trajectory based methods such as Simulated Evolution, Stochastic Evolution and Simulated Annealing, this may involve the partitioning of the solution across available processors so that multiple perturbations/moves may be performed on the solution in each iteration, instead of a single move. This usually implies a conspicuous departure from the functionality and characteristics of the serial algorithm.

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 or information exchange. Such approaches are increasingly proving their worth. These methods allow for increasing the variety of the search threads particularly by having different types of searches - same method with different parameter settings or even different meta-heuristics - proceeding concurrently. Thus, a more thorough exploration of the solution space of a given problem instance becomes possible. As an additional benefit, multithreaded methods appear more robust than their sequential counterparts relative to the differences in problem types and characteristics. Such approaches also offer a relatively easy way to harness the simple and cost effective parallelism provided by an inexpensive network-of-workstations parallel environment.

In this section we discuss several notable parallelization approaches attempted for Simulated Annealing in literature, as well as identify where each approach fits in the above classification. We also identify the pitfalls as well as the potential associated with each technique with respect to our specific problem instance and parallelization environment.

2.1 Move Acceleration

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

Such a strategy, referred to as *move-acceleration* or *move-decomposition* is an example of the Type 1, or low-level parallelization mentioned earlier. It involves a close interaction between processors, and has less potential for parallelism in terms of the amount of parallel work performed and the number of processors that can be employed. Such methodologies are largely restricted to shared memory architectures [8] and preserve all the properties of the serial algorithm. Kravitz and Rutenbar [11] implemented this parallel SA method for cell placement on a shared memory multiprocessor, achieving a speedup of 2 on 4 processors.

2.2 Parallel Moves

An example of the Type 2 or domain decomposition parallelization scheme is the Parallel Moves strategy. In this method, moves are computed independently and in parallel by several processors. Since the global system state is partitioned across the processors, the independent computation and subsequent state update of interacting moves causes the locally held view of the global system state in each processor to become inconsistent with the local views in other processors. Consequently, errors are introduced in move evaluation. The impact of such errors may be kept at a minimum through frequent exchanges of state-update information between processors. However, this approach implies significantly increased inter-processor communication, thereby restricting its application in a cluster-of-workstations environment.

One method to circumvent this problem is to accept a single move from among the set of interacting moves computed in parallel, and discard the rest. This method ensures that no errors are introduced in move evaluation although it is not very efficient. Allowing errors in parallel moves calls for techniques to control their effect on annealing. However, it has been observed that Simulated Annealing is largely error-tolerant and the introduction of a limited amount of error does not drastically affect the convergence properties of the algorithm [12].

Several methods to control the error have been proposed, while in other methods, the algorithm is allowed to proceed with error though occasionally local views of the global state are synchronized across all the processors. Such parallel moves techniques in which error is introduced in a controlled manner create opportunities for exploiting coarse-grained parallelism, and show a greater potential for faster execution. It therefore, becomes very important to understand the nature of these errors and their effect on the quality of the resulting solutions [12, 13].

Kravitz and Rutenbar [11] implemented this approach on a shared memory multiprocessor, achieving a speedup of 3.5 on 4 processors. Banerjee, Jones and Sargent [14] used this approach for standard-cell placement on an iPSC/2 hypercube multiprocessor and proposed several geographical partitioning strategies for the problem specific to the hypercube topology. Speedups of 12 on 16 processors were reported. Casotto et al. [15] worked on speeding up simulated annealing for the placement of macrocells, and achieved speedups of 6 using 8 processors using this approach on a shared memory multiprocessor. Sun and Sechen [16] have shown results achieving near linear speedup on a network of workstations, also using this approach. Chandy and Bannerjee [8] implemented this method for standard cell placement on both a shared-memory Sun 4/690MP as well as a distributed-memory Intel iPSC/860, with the former exhibiting a speedup of approximately 2 on 4 processors, and the latter achieving a maximum speedup of 3.75 on 8 processors. It is important to note at this point that virtually all of the parallel methods listed above exhibited degradation of final solution quality as more processors were added.

2.3 Speculative Execution

Speculative computation attempts to predict the execution behavior of the simulated annealing schedule by speculatively executing future moves on parallel nodes. The speedup is limited to the inverse of the acceptance rate, but being a form of Type 1 parallelization scheme, it does have the advantage of retaining the exact execution profile of the sequential algorithm, and thus the convergence characteristics are maintained.

A sequential simulated annealing schedule is simply a series of move proposals intended to reduce some cost function as related to the particular problem. Each move consists of three parts - the proposal or perturbation, evaluation, and decision. Only after these three parts are completed is the next move started. Since the decision made by the next move is dependent on the current state as determined by prior moves, simulated annealing is almost inherently serial in nature. Consider the decision tree of moves in Figure 1(a). The top node represents a move attempted in a simulated annealing process. There are two possible decisions as a result of this move - acceptance or rejection. Speculative computation will assign two different processors to speculatively work on the two possibilities before the parent move has completed. The reject-processor can start at the same time as the parent, since it will assume that the state has not changed. After the parent has completed the move proposal, it can then relay the new state to the accept-processor.

As the acceptance characteristics of the procedure varies, the shape of the tree can also change. For example, if the acceptance rate is high, it would make sense to generate a linear tree of only acceptance nodes, and on the other hand, a very low acceptance rate would imply the creation of only rejection nodes [see Figure 1(b)].

Speculative computation seems to be a promising avenue to achieve at least some speedup in the high temperature region. However, the work done by Chandy et al.,

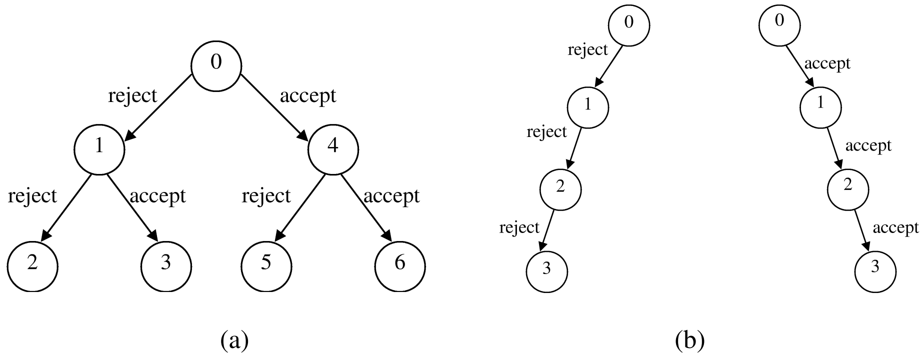


Figure 1: (a) Possible Decision Tree for Speculative Parallel SA, (b) Decision Trees in the Low-temperature and High-temperature regions.

shows that particularly for the standard cell placement problem, speculative execution SA succumbs to a very high overhead and thus is not a feasible option [8].

2.4 Multiple Markov Chains

Multiple Markov chains call for the concurrent execution of separate simulated annealing chains with periodic exchange of solutions [7]. This approach is particularly promising since it has the potential to use parallelism to increase the quality of the solution. All implementations based on this scheme fall under the Type 3 category of parallelization.

Non-interacting Scheme

The algorithm can be understood if the sequential simulated annealing procedure is considered as a search path where moves are proposed and either accepted or rejected depending on particular cost evaluations and also a starting random seed. The search path is essentially a Markov chain, and parallelization is accomplished by initiating different chains (using different seeds) on each processor. Each chain then explores the entire search space by independently performing the perturbation, evaluation, and decision steps. After each processor has completed the annealing schedule, the solutions are compared and the best is selected.

This differs from parallel moves in that each chain is allowed to perform moves on the entire set of cells and not just a subset. Of course, there is no speedup in this approach since each processor is individually performing the same amount of work as the sequential algorithm. To achieve speedup, we must reduce the number of moves evaluated in each chain by a factor of 1/N where N is the number of processors. Since the number of moves determines the run time of the program, a reduction by a factor of 1/N will cause a speedup of N. Obviously, such a reduction alone is not appropriate since the quality will likely decrease accordingly. To take advantage of the fact that multiple processors are being used, some means of interaction or information exchange between the various chains is necessary [8].

Periodic Exchange Scheme: Synchronous MMC

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

Dynamic Exchange Scheme and the Asynchronous MMC Method

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

Soo-Young and Kyung proposed an asynchronous MMC model, which adaptively determines when information is to be exchanged [7]. Communication is permitted based on satisfying certain conditions. First, a certain period of time has to elapse, to allow each PE sufficient independent annealing. Second, these working nodes exchange information only when necessary, rather than at a fixed schedule, e.g., when other PEs have arrived at a significantly better solution. In this way, these processing elements can more efficiently guide each other to a higher quality solution. This is known as the dynamic exchange scheme, and is an asynchronous MMC model.

In order to further improve the performance, asynchronous communication can be centralized by having PEs access a global state repository to reduce overhead and idle time. Each of these processing nodes follows a separate search path and whenever they complete their individual annealing run, they access a global state which consists of the current best solution and its cost. Using this method of managed communication, overhead time can be further reduced substantially. However, an additional master node that holds and communicates the global state is required.

The *master* PE does not perform any computation. When a working node has completed an iteration, it sends its solution metric to the *master* and requests the best solution available. The master PE, on receipt of this request, will determine if the received solution is better than its local "best". If it is, the *master* will ask the requestor to send back its state. The requestor would then do so, and continue with the next set of iterations. If instead, the *master* determines that the local best solution is better than the one received then it would send this current best state to the requesting node. At the cost of dedicating an extra processor for "master" usage, this asynchronous approach can eliminate much of the idle time that was present in earlier schemes.

Chandy and Bannerjee implemented the Asynchronous MMC method for solving the standard-cell placement problem on both a shared-memory Sun 4/690MP as well as a distributed-memory Intel iPSC/860. For the former, a maximum speedup of 2.53 was achieved on 4 processors, and a maximum speedup of 6.26 on 8 processors for the second machine. Both implementations exhibited a mild degradation of final solution quality as the number of processors increased.

The rest of this paper is organized as follows. In Section 3, a detailed description of our Placement Optimization problem and Cost Functions is provided. Next, Section 3.2 we present a brief overview of our experimental setup, followed by details of the attempted parallelization strategies and their results, in Section 4. This is followed by an analysis of these results in Section 5 and finally we conclude in Section 6.

3 The Optimization Problem, Cost Functions and Experimental Setup

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. In this section, we describe the problem and the cost functions for the three objectives and the constraint. The aggregate cost of the solution is computed using fuzzy rules.

3.1 Cost Functions

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:

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

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.

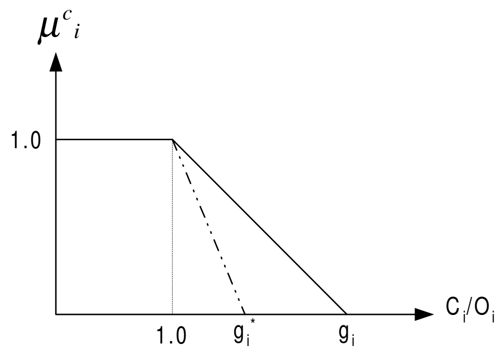


Figure 2: Membership functions

The above rule is translated to *and-like* OWA fuzzy operator [17] and the membership $\mu(x)$ of a solution x in fuzzy set *GOOD solution* is given as:

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

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 2.

3.2 Experimental Setup

The experimental setup consists of a dedicated, homogenous cluster of 8 x 2 GHz Pentium-4 machines, and 256 MB of memory. These machines are connected by 1Gbit/s 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. In terms of GFlops, the maximum performance of the cluster was found to be 1.6 GFlops using NAS Parallel Benchmarks (NAS's LU, Class A, for 8 processors). Using this same benchmark for a single processor, the performance of a single machine was found 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.

In the following section, we present a discussion of each attempted strategy along with its associated results and speedup characteristics. A comparison and discussion of the different strategies is provided in the Sections 4 and 5. ISCAS-89 circuits are used as performance benchmarks for evaluating the parallel strategies. In the results tables below, the target solution quality listed for each benchmark is the lowest common value achieved by all the experimental runs for that benchmark. When generating the results for each of the parallel strategies, at least five runs were made for each circuit and number of processors. The median value of time from each set of five runs is reported. All the runs for a given benchmark circuit had the same initial solution, but different seed values to initialize the pseudo-random number generator.

4 Attempted Parallelization Strategies

Based on the literature studied, it can be concluded that the most promising scheme for parallelization of Simulated Annealing in our inexpensive distributed memory environment is the Asynchronous MMC model [7, 8]. We developed and experimented with several variations of this Type 3 parallel search approach. The primary goals of these experiments were to explore the potential for improvements in both runtime and

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
```

(a)

```
 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 ≥ 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*)
```

(b)

Figure 3: (a) Procedure for Parallel Simulated Annealing using Asynchronous MMC (b) Metropolis Criterion

achievable solution quality by making the most effective utilization of the parallel environment. Each successive parallel strategy attempts to incrementally build upon the knowledge gathered from the previous schemes in order to improve upon their characteristics in terms of runtime and solution quality.

The basic structure of our AMMC PSA implementation is given in Figure 3 below. It is similar to the scheme described in [8]. On each available processing element, an SA operation is initiated with the same starting solution, but with different seeds for pseudo-randomization. The specifications of our AMMC parallel search implementation of SA are given below:

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. Each process communicates with the master independently and compares its own best solution with the solution residing at the master. If the master owns the better solution, the slave starts its next Metropolis loop with this solution, while the master's copy remains unchanged. Conversely, if the slave has the better solution, it continues its work after the master has received this latest best solution, which is then available for comparison by the other slave processes.

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.

We implement four distinct versions of the Asynchronous Multiple Markov Chains approach.

4.1 Asynchronous MMC Parallel SA Strategy 1

For Strategy 1, aside from the above points, there is no difference between the serial version and each of the parallel search processes. This approach is not tuned to provide improved speedup characteristics. Instead, it has been found to improve solution qualities in a fixed amount of time [7], and our results corroborate this fact.

Table 1 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. 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 4 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 sub-linear. Even with 8 processors, we are unable to even achieve a speedup of 3.

Table 1: Results for Strategy 1

Circuit Name	# of Cells	$\mu(s)$ SA	Serial Time	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
s1488	667	0.650381	275	151.46	118.44	112.59	98.94	94.04	92.65
s1494	661	0.647920	214	131.40	116.27	101.89	98.13	92.26	89.10

4.2 Asynchronous MMC Parallel SA Strategy 2

While Strategy 1 is able to meet and even surpass the qualities achieved by the serial algorithm, its runtime characteristics leave something to be desired. Strategy 2 is an attempt to provide near linear speedup over the serial version. This is accomplished by dividing the amount of work done at each of the individual processes by the total number of processes. Specifically, the number of Metropolis iterations at each process is divided by the total number of processes.

Table 2 shows the results obtained from experiments with Strategy 2. Unlike the previous table, the third column here shows the highest common quality that could be achieved by multiple runs of Strategy 2 for every number of processors. Comparing with column 3 of Table 1, we can easily note that there is an average drop in achievable solution quality of approximately 9% with this scheme. Figure 5 shows the speedups achieved by Strategy 2 as the number of processors is varied. In this case we see that speedup is almost linear.

Similar trends are reported in [8] when their AMMC parallel SA is implemented on the distributed-memory Intel iPSC/860. Their results are somewhat different in that they only show a 4% average loss in solution quality instead of 9% for 8 processors. However, our speedup characteristics are slightly better: we achieve an average speedup (over our 4 benchmark circuits) of 6.84 for 8 processors as opposed to their 5.9.

Table 2: Results for Strategy 2

Circuit Name	Number of Cells	$\mu(s)$ SA	Serial SA Time	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
s1488	667	0.582884	101	42.67	25.59	18.77	16.61	15.85	13.88
s1494	661	0.591114	75	51.11	30.79	22.32	15.82	14.9	13.52

4.3 Asynchronous MMC Parallel SA Strategy 3

With Strategy 2, we were able to address the runtime limitations of Strategy 1 in a limited manner. However, this was achieved only with a 9% reduction in solution quality. We see that although a division of the workload has a positive impact on runtime, there

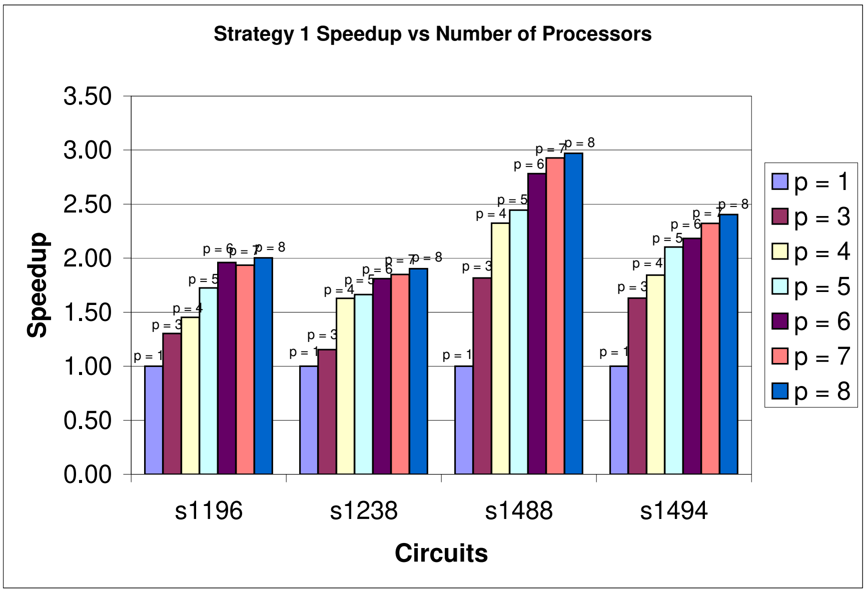


Figure 4: Speedup versus number of machines for Parallel SA AMMC Strategy 1

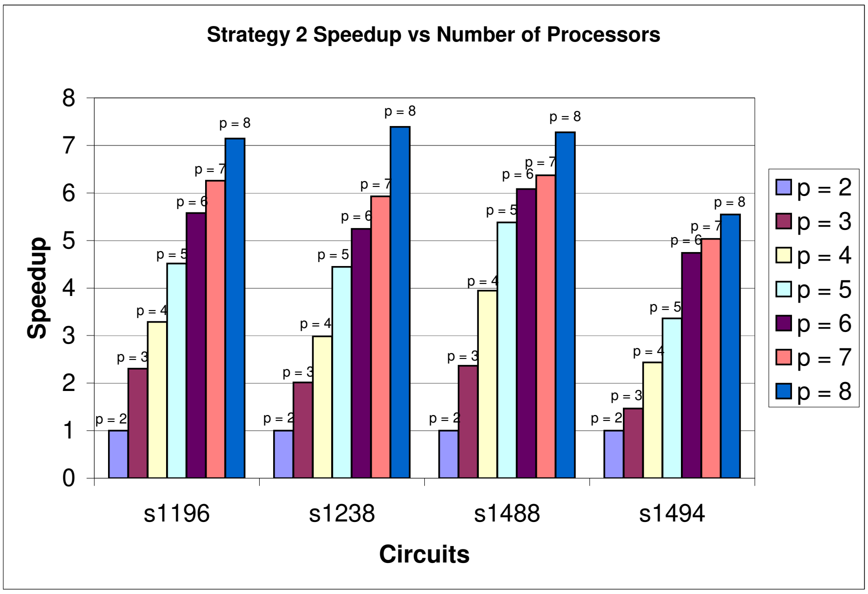


Figure 5: Speedup versus number of machines for Parallel SA AMMC Strategy 2

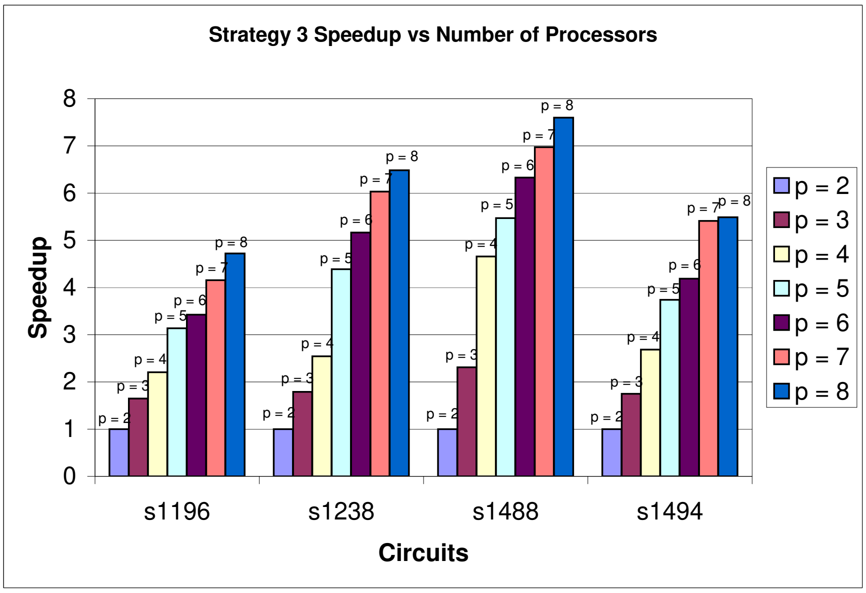


Figure 6: Speedup versus number of machines for Parallel SA AMMC Strategy 3

is an adverse impact on achievable quality. The loss in achievable quality in Strategy 2 can be understood by looking at how the intelligence of the algorithm is affected by division of the factor 'M'. All of the parameters of the cooling schedule were originally optimized for the serial Simulated Annealing. 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.

In Strategy 3, we attempted to offset the negative impact on algorithmic intelligence by introducing other enhancements to the parallel algorithm. This was done by implementing different cooling schedules on each processor in such a way that some of the processors are searching for new solutions in a greedy manner, while others are still in the high temperature region. We essentially aim to counterbalance the impact of shortened Markov-chains on achievable quality by making intelligent use of the interaction between chains that occurs after every Metropolis loop.

This is different from the Temperature Parallel Simulated Annealing (TPSA) approach described in [18], which maintains all the parallel processes at constant but different temperatures. Whereas in Strategy 3, the values of *alpha* is different on different processors, thus the rate of temperature change is varied across processors. This is because our intended goals are different from those of TPSA. Whereas our primary aim is to achieve serial-equivalent qualities while achieving near-linear runtimes, the aim of TPSA was primarily to enhance the robustness of Parallel SA, and minimize the amount of effort required in parameter setting.

However, we find that even this proposed enhancement of varying *alpha* is insufficient to counteract the impact of divided 'M'. Our results for Strategy 3, shown in Table 3 and Figure 6 show no improvement over the results obtained for Strategy 2 - for some circuits (e.g. s1196), there is even a drop in achievable speedup and quality.

Thus a more insightful and intelligent parallel cooling schedule will be required to achieve the target qualities.

Table 3: Results for Strategy 3

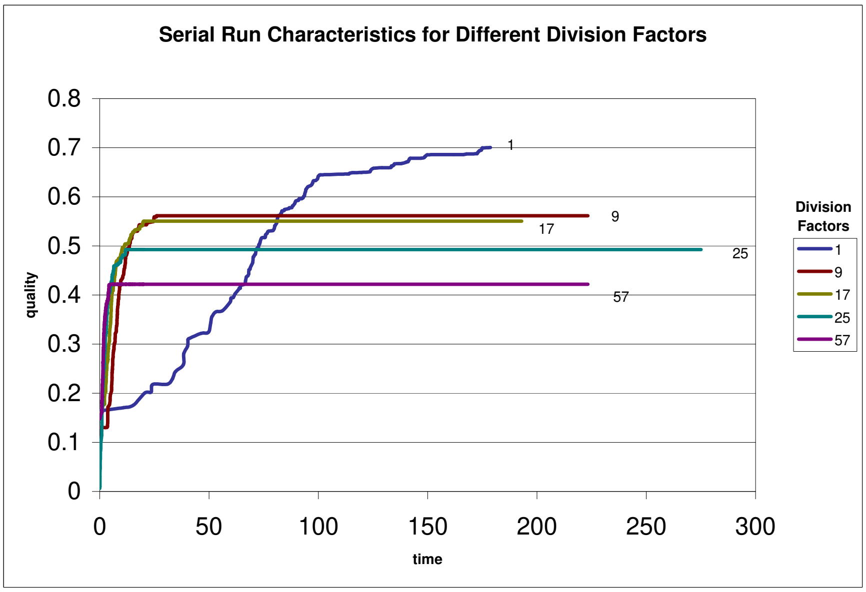
Circuit Name	Number of Cells	$\mu(s)$ SA	Serial SA Time	Time for Parallel SA Strategy 3					
				p=3	p=4	p=5	p=6	p=7	p=8
s1196	561	0.606818	64	38.85	29.03	20.40	18.68	15.41	13.55
s1238	540	0.630573	117	65.36	45.97	26.65	22.65	19.39	18.04
s1488	667	0.582884	101	43.71	21.68	18.46	15.96	14.49	13.29
s1494	661	0.591114	75	42.89	27.95	20.05	17.92	13.86	13.67

4.4 Asynchronous MMC Parallel SA Strategy 4 - Adaptive Cooling Schedule

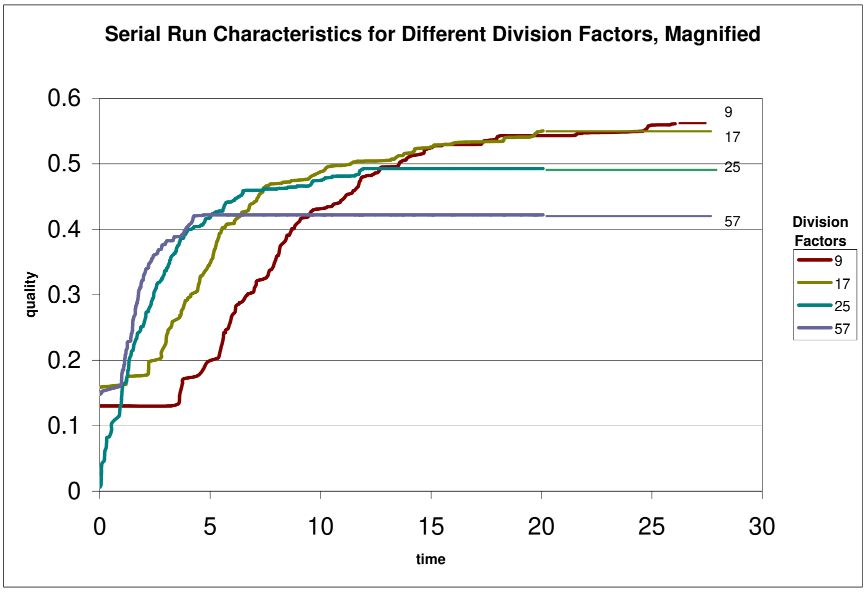
From the results of the previous three strategies, it became evident that for parallel SA, if any progress is to be made towards achieving our goals of near-linear run times with sustained quality, an in depth study of the impact of parameter M on achievable solution quality is required. To this end,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

From these results, we can see that division of M by a larger number increases the rate at which new solutions are found initially, but the system stagnates at a lower final solution quality. 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 increases too fast, runtime is compromised; if M increases too slowly, achievable solution quality is affected. The key to this dilemma of approximating the appropriate value of M comes from an interesting observation made 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.

Based on what we have learned from these experiments, we proposed certain modifications to the cooling schedule of our basic, serial Simulated Annealing algorithm. This adaptive cooling schedule, when implemented for the parallel AMMC scheme, yielded our 4th parallel search SA strategy. A brief description of the adaptive cooling schedule is given below:

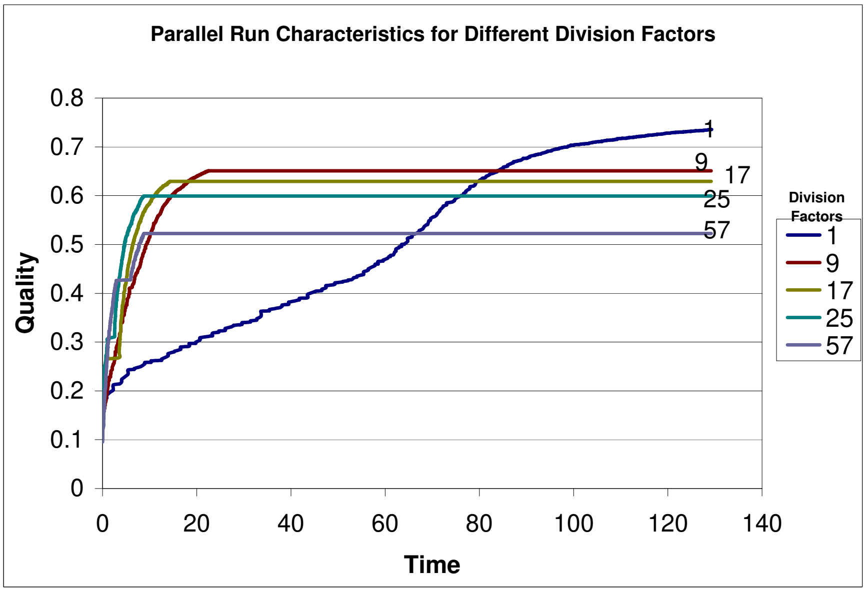


(a)

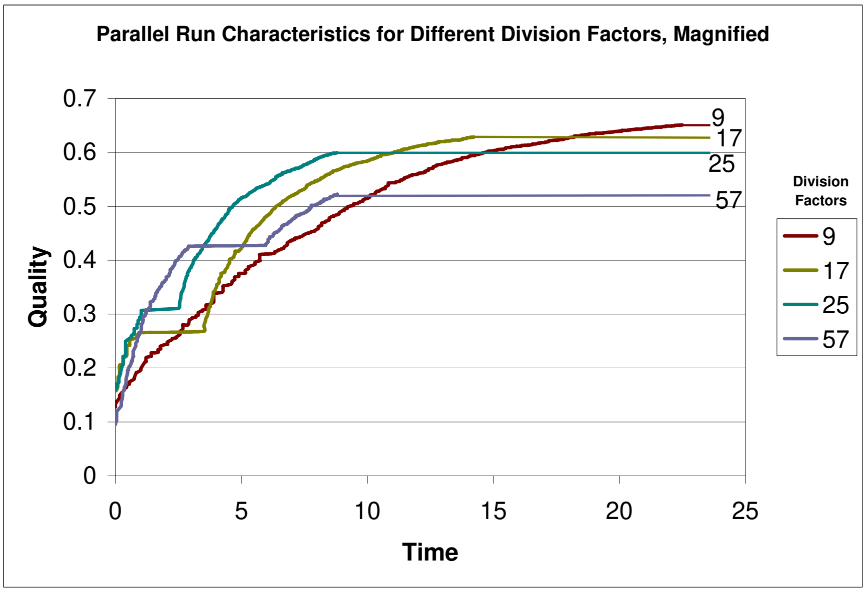


(b)

Figure 7: Quality vs. Runtime results for Serial SA, with different values for M



(a)



(b)

Figure 8: Quality vs. Runtime results for AMMC Parallel SA (7 processors), with different values for M

1. For the first 100 or so Annealing iterations, accumulate an average of the quality improvement per Metropolis function call. This average rate of improvement will serve as a threshold that needs to be maintained per Metropolis Function call.

2. Initially, the value of 'M' is set to a very small value - the value used in the basic algorithm is divided by 25 to provide the initial M in the adaptive version.

3. After the initial average accumulation iterations, adaptivity is initiated. If rate of improvement drops below a certain threshold, increase M incrementally, since not enough time is being spent at each temperature level.

4. If rate of improvement is constantly more than the threshold value, decrease M, since an unnecessary amount of time is being spent at the given quality level.

5. The value of the M parameter is not allowed to exceed twice the value used in the original basic version, until significant stagnation is detected (e.g.: no improvement in solution quality for the past 25 Metropolis calls).

The application of the last condition was empirically found to dramatically improve algorithm run times, without sacrificing final quality achieved.

The run times for Serial and Parallel versions of Simulated Annealing with the adaptive cooling schedule are given in Table 4 for the solution qualities achieved by Strategy 1. Table 5 shows the run times of the adaptive serial and parallel schemes for achieving the quality targets set by Strategy 2. As can be seen, both the serial and parallel run times have improved dramatically over Strategy 1, while the parallel runtimes are largely equivalent to those of Strategy 2.

Table 4: Results for Adaptive Strategy 4 (Strategy 1 Qualities)

Circuit Name	Number of Cells	$\mu(s)$ SA	Serial SA Time	Time for Parallel SA Strategy 4					
				p=3	p=4	p=5	p=6	p=7	p=8
s1196	561	0.675340	75.4	60.31	47.87	47.34	46.25	42.44	39.89
s1238	540	0.699469	115.9	96.45	84.21	67.59	63.05	53.79	47.68
s1488	667	0.650381	106.6	77.84	70.62	59.92	51.80	43.38	37.28
s1494	661	0.647920	139.7	101.1	77.38	76.68	59.68	50.12	48.44

Table 5: Results for Adaptive Strategy 4 (Strategy 2 Qualities)

Circuit Name	Number of Cells	$\mu(s)$ SA	Serial SA Time	Time for Parallel SA Strategy 4					
				p=3	p=4	p=5	p=6	p=7	p=8
s1196	561	0.630367	37.35	23.71	23.24	21.74	20.57	17.95	17.13
s1238	540	0.630573	45.85	33.76	24.52	19.65	23.53	15.03	16.12
s1488	667	0.582884	29.59	21.35	18.26	13.36	13.46	12.84	11.38
s1494	661	0.591114	46.92	27.78	20.09	20.14	17.68	18.16	16.55

Furthermore, for all runs and all circuits on any number of processors, Strategy 4 manages to achieve significantly higher solution qualities than either Strategy 1 or

Strategy 2 before reaching saturation. For instance, Strategy 4 achieved solution qualities of 0.728082 for circuit s1196 on 7 processors, 0.764924 for s1238 on 8 processors, 0.708843 for s1488 on 6 processors, and 0.704714 for s1494 on 8 processors, exhibiting an approximate solution quality improvement of 9% over the basic serial SA, although requiring much longer runtimes than the latter.

Note however, that the speedup characteristics of Strategy 4 are very similar to those of Strategy 1: for the given quality values, speedup never exceeds 3 (Figure 9(a)).

Even for the lower qualities achieved by Strategy 2, the speedup characteristics of Strategy 4 do not improve, as seen in Figure 9(b). In fact it is evident from Tables 2 and 5 that for 6 processors and above, Strategy 2 is often able to achieve its target solution qualities sooner than Strategy 4, particularly with 8 processors.

5 Discussion and Analysis

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 SA implementations with respect to the above factors.

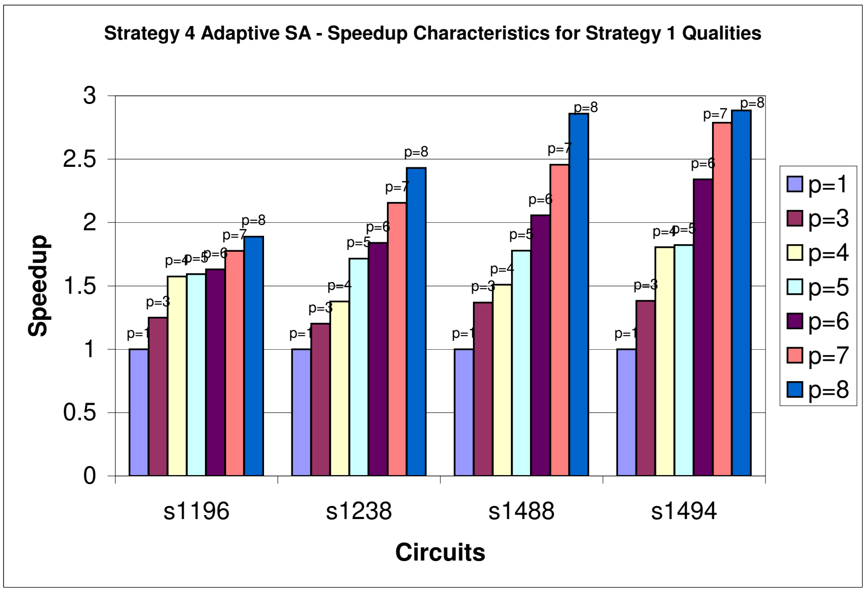
5.1 Cost Computation Function

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. Computing this multi-objective cost function requires the most recent state of the solution to be accurate. As such, partitioning of a single solution over different processes would be infeasible due to interdependencies between cells in the netlist. This is specially true for delay computation which takes place on long paths that can span across row boundaries.

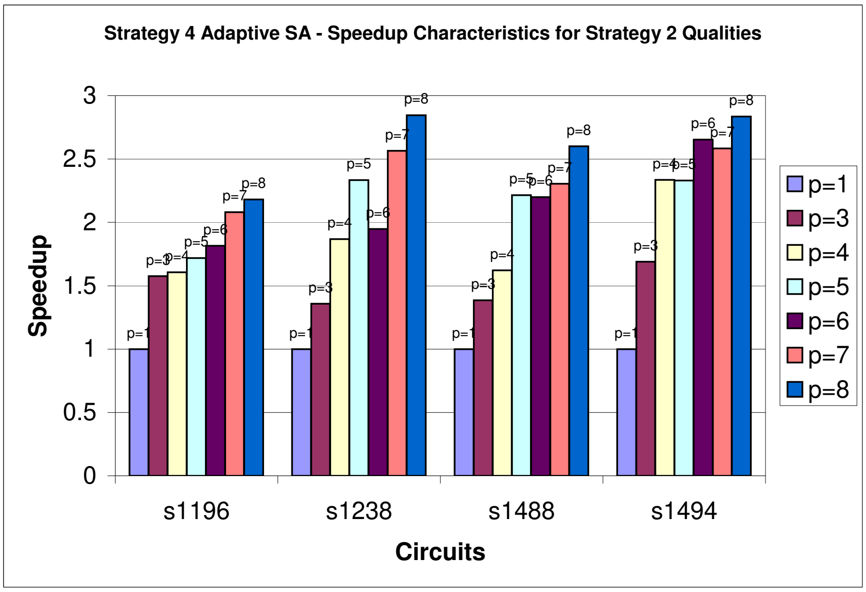
The Type 3 Parallel Search strategies described so far are immune to this issue, since aside from the sparse solution exchanges, each processing element is undertaking an independent but complete search operation. This means that the cost computation functions remain undivided and operate on largely distinct 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.

5.2 Parallelization Environment

In our cluster-of-workstations operating environment, it is essential to minimize the amount of communication in relation to the computation. The periodic, asynchronous communication model used for the Type 3 parallel strategies 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



(a)



(b)

Figure 9: Speedup Characteristics of Parallel Adaptive Simulated Annealing (Strategy 4) for solution qualities of (a) Strategy 1, (b) Strategy 2

can be verified from Figure 10, which shows the ratio of communication time to computation time for our parallel SA Strategy 2, when run on seven processors for circuit s1196.

5.3 Solution Perturbation and Algorithmic Intelligence

The solution perturbation and next-state selection operators are where the intelligence of virtually all stochastic heuristics lies. The solution perturbation operation in SA is inherently sequential and in the chosen parallelization schemes it is left undivided.

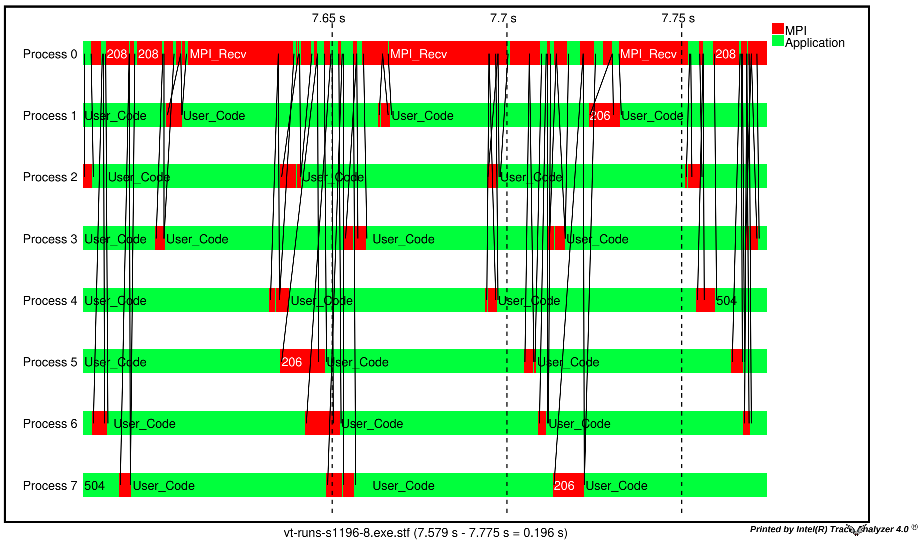
The intelligence of SA lies in its cooling-schedule. In Type 3 parallel SA, each independent parallel search chain periodically starts its search from the best available solution at the time. This, coupled with the ability of SA to escape local minima, allows the parallel search to be focused around 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 Strategies 1 and 4.

As for Strategies 2 and 3, 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 dividing the parameter 'M'.

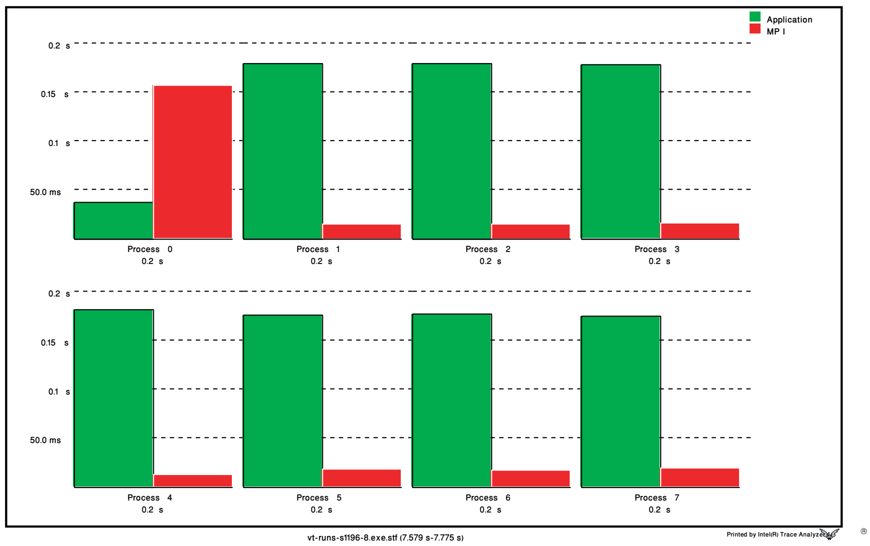
6 Conclusion

In this paper, we have presented 4 distinct implementations of AMMC PSA. Strategy 1 provides significantly better solution qualities than the serial algorithm, but only modest speedup. Strategies 2 and 3 suffer a quality loss of at least 9%, but provide near linear speedups for the achieved qualities. Our best parallel implementation in terms of both solution quality achievable and run time was Strategy 4 - a new implementation of Simulated Annealing utilizing an adaptive cooling schedule.

This cooling schedule was devised 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 earlier parallel strategies. However, compared to the serial version of SA with an Adaptive cooling schedule, the speedup benefits of parallelization appear less significant. They are in fact similar to the runtime characteristics seen between Strategy 1 and the original Serial SA - achieving the same quality solution in slightly lesser time. The speedup with even eight processors remains less than three.



(a)



(b)

Figure 10: (a) Communication versus Computation traces for all processors for Type-3 Parallel SA (b) Ratio of Communication to Computation for each processor for Type-3 Parallel SA.

Our results for the above strategies show that we have been partially successful in achieving our goals. We succeeded in developing viable parallel Simulated Annealing implementations for solving a multi-objective VLSI standard-cell placement on an inexpensive cluster of workstations. We were also able to improve the solution qualities achieved over the serial algorithm in the same amount of time (Strategies 1 and 4). We were, however, unable to achieve near-linear speedups without sacrificing final solution quality (Strategies 2 and 3).

Despite this, it should be noted that the speedup-oriented strategies, particularly Strategy 2 may prove useful in scenarios where speedup is a more urgent requirement than solution quality. It is evident from Tables 2 and 5 that if solution quality may be compromised, the runtime characteristics of Strategy 2 can can compete even with those of Strategy 4 as the number of processors is increased. In fact, for 8 processors (at the lower solution qualities), the former has better runtime results than the latter.

In the future, we aim to explore in greater detail the characteristics of our adaptive cooling schedule, as well as other derivatives of Simulated Annealing such as Very-fast Simulated Re-Annealing, Simulated Quenching, and Mean-Field Annealing etc. [19]. In particular, we aim to focus on the suitability of these approaches for parallelization. It is hoped that a thorough study of these methods will allow us to develop a parallel SA scheme that can provide an improvement on our speedup characteristics without sacrificing final solution quality.

Acknowledgment

The authors thank King Fahd University of Petroleum & Minerals (KFUPM), Dhahran, Saudi Arabia, for support under Project Code COE/CELLPLACE/263. The authors would also like to acknowledge the contributions of Sanaullah Syed and Mohammed Faheemuddin in the editing and review of this manuscript.

References

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

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

[3] V.-D. Cung, S. L. Martins, C. C. Riberio, and C. Roucairol, "Strategies for the Parallel Implementation of metaheuristics," *Essays and Surveys in Metaheuristics*, pp. 263–308, Kluwer 2001.

[4] T. G. Crainic and M. Toulouse, "Parallel strategies for metaheuristics," *Handbook of Metaheuristics, Editors: F. W. Glover and G. A. Kochenberger*, pp. 465–514, 2003.

[5] S. M. Sait and H. Youssef, *Iterative Computer Algorithms and their Application to Engineering*. IEEE Computer Society Press, December 1999.

[6] E. E. Witte, R. D. Chamberlain, and M. A. Franklin, "Parallel SA using speculative execution," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 4, October 1991.

[7] S.-Y. Lee and K. G. Lee, "Synchronous and asynchronous parallel simulated annealing with multiple-markov chains," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 10, pp. 993–1008, October 1996.

[8] J. Chandy, S. Kim, B. Ramkumar, S. Parkes, and P. Bannerjee, "An evaluation of parallel simulated annealing strategies withapplication to standard cell placement," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 4, pp. 398–410, April 1997.

[9] M. Toulouse and T. G. Crainic, *State-of-the-Art Handbook in Metaheuristics*. Kluwer Academic Publishers, 2002, ch. Parallel Strategies for Metaheuristics.

[10] V.-D. Cung, S. Martins, C. Ribeiro, and C. Roucairol, *Essays and Surveys in Metaheuristics*. Kluwer, 2001, ch. Strategies for the parallel implementation of metaheuristics, pp. 263–308.

[11] S. A. Kravitz and R. A. Rutenbar., "Placement by simulated annealing on a multiprocessor," *IEEE Trans. on Computer Aided Design*, vol. 6, no. 4, pp. 534–549, July 1987.

[12] R. Jayaraman and F. Darema, "Error tolerance in parallel simulated annealing techniques," *Proceedings of the 1988 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 545–548, October 1988.

[13] M. D. Durand and S. R. White, "Trading accuracy for speed in parallel simulated annealing with simultaneous moves," *High performance computing in operations research*, vol. 26, no. 1, pp. 135–150, January 2000.

[14] P. Banerjee, M. H. Jones, and J. S. Sargent, "Parallel simulated annealing algorithms for standard cell placement on hypercube multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 91–106, January 1990.

[15] A. Casotto, F. Romeo, and A. Sangiovanni-Vincentelli, "A parallel simulated annealing algorithm for the placement of macro-cells," *IEEE Transactions on Computer-Aided Design*, vol. CAD-6, pp. 838–847, September 1987.

[16] W. J. Sun and C. Sechen, "A loosely coupled parallel algorithm for standard cell placement," *Digest of Papers, International Conference on Computer-Aided Design*, pp. 137–144, November 1994.

[17] R. R. Yager, "On ordered weighted averaging aggregation operators in multicriteria decision making," *IEEE Transaction on Systems, MAN, and Cybernetics*, vol. 18, no. 1, January 1988.

[18] K. Konishi, K. Taki, and K. Kimura, "Temperature parallel simulated annealing algorithm and its evaluation," *Transactions on Information Processing Society of Japan*, vol. 36, no. 4, pp. 797–807, 1995.

[19] L. Ingber, "Simulated annealing: Practice versus theory," *Journal of Mathematical Computation Modelling*, vol. 18, no. 11, pp. 29–57, 1993.