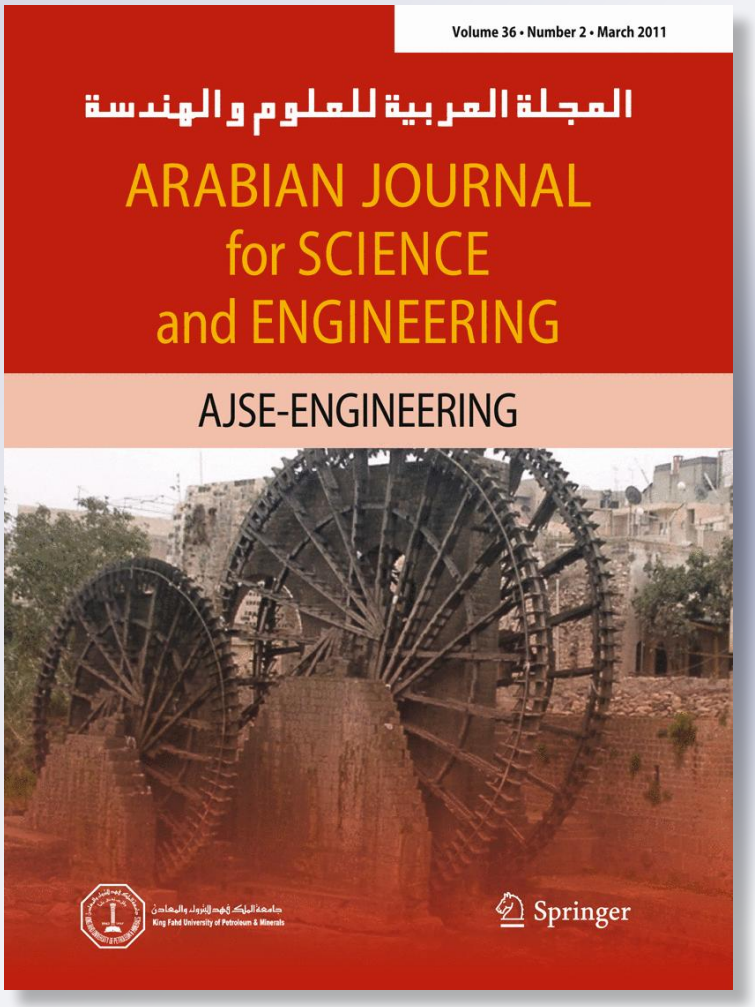
Exploring Asynchronous MMC-Based Parallel SA Schemes for Multiobjective Cell Placement on a Cluster of Workstations

Arabian Journal for Science and Engineering

ISSN 1319-8025
Volume 36
Number 2

Arab J Sci Eng (2011)
36:259-278
DOI 10.1007/
s13369-010-0024-6



Springer

Arab J Sci Eng (2011) 36:259–278
DOI 10.1007/s13369-010-0024-6

RESEARCH ARTICLE – COMPUTER ENGINEERING AND COMPUTER SCIENCE

Sadiq M. Sait · Ali M. Zaidi · Mustafa I. Ali · Khawar S. Khan · Sanaullah Syed

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

Received: 12 June 2009 / Accepted: 21 December 2009 / Published online: 15 January 2011

Abstract Combinatorial optimization problems are generally NP hard problems that require large run-times when solved using iterative heuristics. Parallelization using distributed or shared memory computing clusters thus becomes a natural choice to speed up the execution times of such problems. In this paper, several parallel schemes based on an asynchronous multiple-Markov-chain (AMMC) model are explored to parallelize simulated annealing (SA), used for solving a multiobjective VLSI cell placement problem. The different parallel schemes are investigated based on the speedups and solution qualities achieved on an inexpensive cluster of workstations. The 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. The goal is to develop effective AMMC-based parallel SA schemes to achieve near linear speedups while maintaining or achieving higher solution qualities in less time and to analyze these parallel schemes against the common critical performance factors.

Keywords Asynchronous MMC · Parallel SA schemes · Multiobjective cell placement · Cluster-of-workstations

S. M. Sait (✉) · A. M. Zaidi · M. I. Ali · K. S. Khan · S. Syed
College of Computer Sciences and Engineering, King Fahd University of Petroleum & Minerals,
Dhahran 31261, Saudi Arabia
E-mail: sadiq@kfupm.edu.sa

A. M. Zaidi
E-mail: alizaidi@kfupm.edu.sa

M. I. Ali
E-mail: miali@kfupm.edu.sa

K. S. Khan
E-mail: khawar@kfupm.edu.sa

S. Syed
E-mail: sanaulla@kfupm.edu.sa

الخلاصة

تُعد مشاكل الاندماج الأمثل بعامة مشاكل صعبة تتطلب زمن تشغيل كبير عند حلها باستخدام الاستدلال التكراري. إن الموازاة باستخدام ذاكرة الحسابات الموزعة أو المشتركة أصبحت خيارا طبيعياً لتسريع زمن التنفيذ لتلك المشاكل.

وفي هذه الورقة ثم استكشاف أنموذج مخططات عدة متوازية على أساس غير متزامن متعدد لسلسلة ماركوف (AMMC) وذلك ليوازي محاكاة الصلب (SA) الذي يستخدم لحل مشكلة وضع الخلية متعددة الأغراض VLSI. وقد تم التحقق من مخططات متوازية مختلفة استناداً إلى التسريع وجودة الحل التي تحققت على كتلة غير مكلفة في محطة العمل. وتتطلب المشكلة أفضل استفادة من الأهداف المتعارضة (طول السلك، وتبديد الطاقة ، وتوقيت الأداء) . وقد استخدم منطق FOZZY لدمج الكلفة لهذه الأهداف. وكان الهدف هو وضع AMMC الفعال على أساس مخططات متوازية SA للحصول على أقرب تسريع خطي في أثناء الصيانة أو الحصول على أعلى جودة حل في أقل زمن ومن ثم تحليل هذه المخططات المتوازية ضد عوامل الأداء الحاسمة.

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 (SA) that are capable of delivering acceptable or near-optimal solutions to these problems with reasonable run times [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 is 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 speedups 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 [1,5]. 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 [6]. 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 [6,7]. 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, we attempt to solve the multiobjective VLSI standard cell placement problem. We experiment with different versions of the asynchronous multiple-Markov-chain parallel SA (or AMMC PSA) approach described in [6], as this scheme has been found to be well suited for solving this problem in a distributed-memory environment [7]. 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.

The contributions of this work are highlighted briefly in [8]. This paper presents the comprehensive explanation behind each contribution as well as the detailed discussion and analysis of results with respect to the common critical performance factors discussed in Sect. 5.

2 SA Parallelization Strategies in Literature

Several studies of parallelization strategies for metaheuristics in general have been reported in literature [3,9]. 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 metaheuristics 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 [7] and preserve all the properties of the serial algorithm. Kravitz and Rutenbar [10] implemented this parallel SA method for cell placement on a shared memory multiprocessor, achieving a speedup of two on four 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 [11].

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 [11,12].

Kravitz and Rutenbar [10] implemented this approach on a shared memory multiprocessor, achieving a speedup of 3.5 on four processors. Banerjee et al. [13] 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 twelve on sixteen processors were reported. Casotto et al. [14] worked on speeding up simulated annealing for the placement of macrocells, and achieved speedups of 6 using eight processors using this approach on a shared memory multiprocessor. Sun and Sechen [15] have shown results achieving near linear speedup on a network of workstations, also using this approach. Chandy and Bannerjee [7] 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 two on four processors, and the latter achieving a maximum speedup of 3.75 on eight 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 Fig. 1a. 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

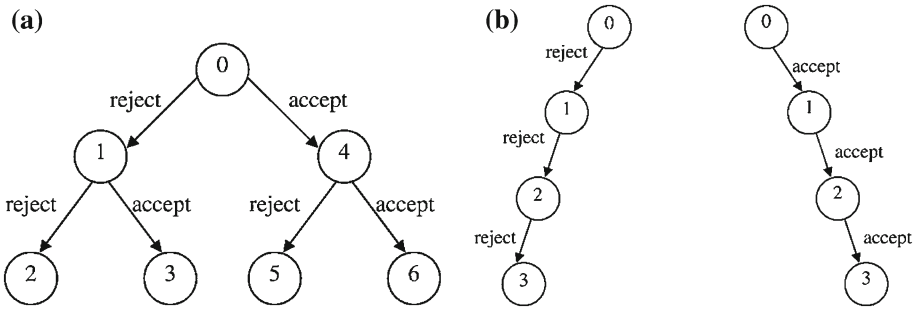


Fig. 1 **a** Possible decision tree for speculative parallel SA. **b** Decision trees at low-temperature and high-temperature regions

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. On the other hand, a very low acceptance rate would imply the creation of only rejection nodes (see Fig. 1b).

Speculative computation seems to be a promising avenue for achieving at least some speedup in the high temperature region. However, the work done by Chandy et al. 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 [7].

2.4 Multiple Markov-Chains

Multiple Markov-chains call for the concurrent execution of separate simulated annealing chains with periodic exchange of solutions [6]. 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 $1/p$ where p is the number of processors. Since the number of moves determines the runtime of the program, a reduction by a factor of $1/p$ will cause a speedup of p. 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 [7].

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 [6]. 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 four processors, and a maximum speedup of 6.26 on eight 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 Sect. 3, a detailed description of our placement optimization problem and cost functions is provided. Next, Sect. 3.2 we present a brief overview of our experimental setup, followed by details of the attempted parallelization strategies and their results, in Sect. 4. This is followed by an analysis of these results in Sect. 5 and finally we conclude in Sect. 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

A Steiner tree approximation, which is fast and fairly accurate in estimating the wire length is adopted [16]. To estimate the length of net using this method, a bounding box, which is the smallest rectangle bounding the net, is found for each net. The average vertical distance Y and horizontal distance X of all cells in the net are computed from the origin, which is the lower left corner of the bounding box of the net. A central point (X, Y) is determined at the computed average distances. If X is greater than Y then the vertical line crossing the central point is considered as the bisecting line. Otherwise, the horizontal line is considered as the bisecting line. A Steiner tree approximation of a net is the length of the bisecting line added to the summation of perpendicular

distances to it from all cells belonging to the net. A Steiner tree approximation is computed for each net and the summation of all Steiner trees is considered as the interconnection length of the proposed solution.

$$X = \frac{\sum_{i=1}^{n} x_i}{n}, \quad Y = \frac{\sum_{i=1}^{n} y_i}{n}, \tag{1}$$

where n is the number of cells contributing to the current net.

$$\text{Steiner tree} = B + \sum_{j=1}^{k} P_j, \tag{2}$$

where B is the length of the bisecting line, k is the number of cells contributing to the net and P_j is the perpendicular distance from cell j to the bisecting line.

$$\text{Interconnection length} = \sum_{l=1}^{m} \text{Steiner tree}_l, \tag{3}$$

where m is the number of nets.

Power Cost

In VLSI circuits with well-designed logic gates, the dynamic power consumption contributes the 90% to the total power consumption [17,18]. Minimizing the dynamic power consumption is among the objectives as mentioned before. Power consumption p_i of a net i in a circuit can be given as:

$$p_i \simeq \frac{1}{2} \cdot C_i \cdot V_{\text{DD}}^2 \cdot f \cdot S_i \cdot \beta, \tag{4}$$

where C_i is total capacitance of net i, V_{DD} is the supply voltage, f is the clock frequency, S_i is the switching probability of net i, and β is a technology dependent constant.

Assuming a fix supply voltage and clock frequency, then power dissipation of a cell depends on its capacitance and its switching probability. Hence, the above equation reduces to the following:

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

The capacitance C_i of cell i is given as:

$$C_i = C_i^{\text{r}} + \sum_{j \in M_i} C_j^{\text{g}}, \tag{6}$$

where C_j^{g} is the input capacitance of gate j and C_i^{r} is the interconnect capacitance at the output node of cell i.

At the placement phase, only the interconnect capacitance C_i^{r} can be manipulated while C_j^{g} comes from the properties of the cell from the library used and is thus independent of placement. Moreover, C_i^{r} depends on wirelength of net i, so Eq. 5 can be written as:

$$p_i \simeq l_i \cdot S_i \tag{7}$$

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

$$\text{Cost}_{\text{power}} = \sum_{i \in M} p_i = \sum_{i \in M} (l_i \cdot S_i) \tag{8}$$

Delay Cost

A digital circuit comprises a collection of paths. A path is a sequence of nets and blocks from a source to a sink. A source can be an input pad or a memory cell output, and a sink can be an output pad or a memory cell input. The longest path (*critical path*) is the dominant factor in deciding the clock frequency of the circuit. A critical path makes a problem in the design if it has a delay that is larger than the largest allowed delay (period) according to the clock frequency. Thus, this cost is determined by the delay along the longest path in a circuit. The delay T_π of a path π consisting of nets $\{v_1, v_2, \ldots, v_k\}$, is expressed as:

$$T_\pi = \sum_{i=1}^{k-1} (\mathrm{CD}_i + \mathrm{ID}_i), \tag{9}$$

where CD_i is the switching delay of the cell driving net v_i and ID_i is the interconnect delay of net v_i. The overall circuit delay is equal to T_{π_c}, where π_c is the longest path in the layout (most critical path). The placement phase affects ID_i because CD_i is technology dependent parameter and is independent of placement. Using the RC delay model, one obtains ID_i:

$$\mathrm{ID}_i = (\mathrm{LF}_i + R_i^{\mathrm{r}}) \times C_i, \tag{10}$$

where LF_i is a load factor of the driving block, that is independent of layout, R_i^{r} is the interconnect resistance of net v_i and C_i is the load capacitance of cell i given in Eq. 6.

The delay cost function can be written as:

$$\mathrm{Cost}_{\mathrm{delay}} = \max\{T_\pi\} \tag{11}$$

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:

$$\mathrm{Width} - w_{\mathrm{avg}} \leq \alpha \times w_{\mathrm{avg}} \tag{12}$$

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 [19] and the membership $\mu(x)$ of a solution x in fuzzy set *GOOD solution* is given as:

$$\mu(x) = \begin{cases} \beta \cdot \min_{j=p,d,l}\{\mu_j(x)\} + (1-\beta) \cdot \frac{1}{3} \sum_{j=p,d,l} \mu_j(x); & \text{if Width} - w_{\mathrm{avg}} \leq \alpha \cdot w_{\mathrm{avg}}, \\ 0; & \text{otherwise.} \end{cases} \tag{13}$$

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 Fig. 2.

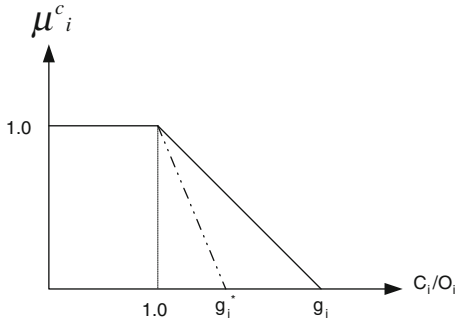


Fig. 2 Membership functions

3.2 Experimental Setup

The experimental setup consists of a dedicated, homogenous cluster of 8 × 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, one finds the performance of a single machine to be 0.3 GFlops. The maximum bandwidth that was achieved using PMB was 91.12 Mbits/s, with an average latency of 68.69 μs 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 Sects. 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 [6,7]. 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 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 Fig. 3 below. 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.

(a)

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. *)

(*β a constant.Gradually increases the time spent in Annealing as the temperature is lowered. *)

(* 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; // Temperature initialized

CurS = S0; // only master has the initial Solution

BestS = CurS; // Initially Current Solution is the Best Solution

CurCost = Cost(CurS); // Calculate cost of Current Solution

BestCost = Cost(BestS); // Calculate cost of Best Solution

Time = 0;

If (my_rank == 0) // i.e. Master process

Broadcast(CurS); // Broadcast Current solution to all slaves

Endif

(b)

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); // All slaves send new best costs to master

Receive_frm_Master(verdict); // Master sends the verdict on if it has the better solution or slave has

If (verdict == 1)

Send_to_Master (BestS); // Send the better solution to Master

Else

Receive_frm_Master(BestS); // Receive the better solution from Master

EndIf

Until (Time $\geq$ Maxtime);

EndIf

If (my_rank == 0) // i.e. Master process

Repeat

Receive_frm_Slave(BestCost); // Waiting for the slave to send best costs

Send_to_Slave(verdict); // Sending verdict on if Master has the better solution or slave has

If (verdict == 1)

Receive_frm_Slave(BestS); // Receive the better solution from Slave

Else

Send_to_Slave (BestS); // Send the better solution to Slave

EndIf

Until (All Slaves are done);

Return(BestS);

EndIf

End. (*Parallel_Simulated_Annealing*)

Fig. 3 **a** Procedure for parallel simulated annealing using asynchronous MMC. **b** Metropolis criterion

Table 1 Results for Strategy 1

Circuit name	# of cells	μ(s) SA	Serial SA 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

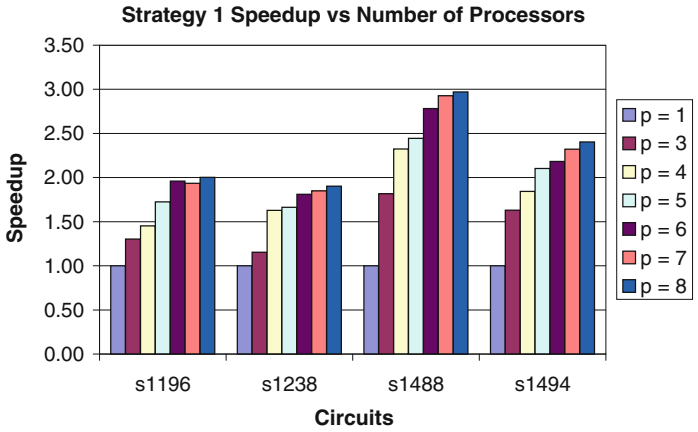


Fig. 4 Speedup versus number of machines for parallel SA AMMC Strategy 1

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.

The above specifications are essentially the same as the Asynchronous MMC scheme described in [7]. We implement four distinct versions of the asynchronous multiple Markov-chain 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 [6], 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 eight processors, we are unable to even achieve a speedup of 3.

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

Table 2 Results for Strategy 2

Circuit name	Number of cells	μ(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

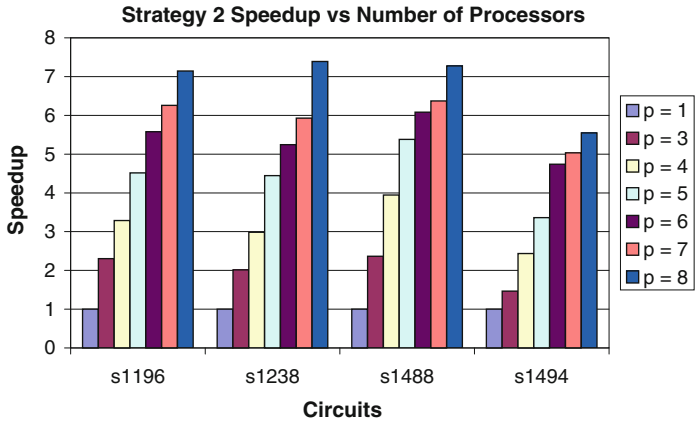


Fig. 5 Speedup versus number of machines for parallel SA AMMC Strategy 2

number of processors where fourth column shows the serial SA time, i.e. time taken by serial SA to achieve this common quality. 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 [7] 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 eight processors. However, our speedup characteristics are slightly better: we achieve an average speedup (over our four benchmark circuits) of 6.84 for eight processors as opposed to their 5.9. These differences in characteristics may be attributed to the following factors: (a) the cost function used are different (wire-lengths of nets as opposed to our multiobjective fuzzy cost function), (b) the benchmarks utilized are different (Physical Design Workshop 91 vs. our use of ISCAS 89), and (c) differences in operating environment (ISC Hypercube vs. our inexpensive cluster of workstations).

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 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. The 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

Table 3 Results for Strategy 3

Circuit name	Number of cells	μ(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

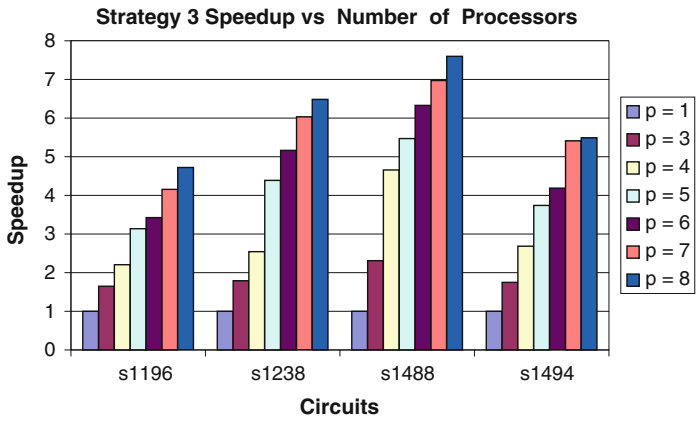


Fig. 6 Speedup versus number of machines for parallel SA AMMC Strategy 3

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 [20], 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 Fig. 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.

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 Fig. 7a, with a close up of the top-left region of this graph shown in Fig. 7b. The quality versus runtime results for similar runs of the Type 3 parallel SA on seven processors are given in Fig. 8, with a closeup of the active region given in Fig. 8b.

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

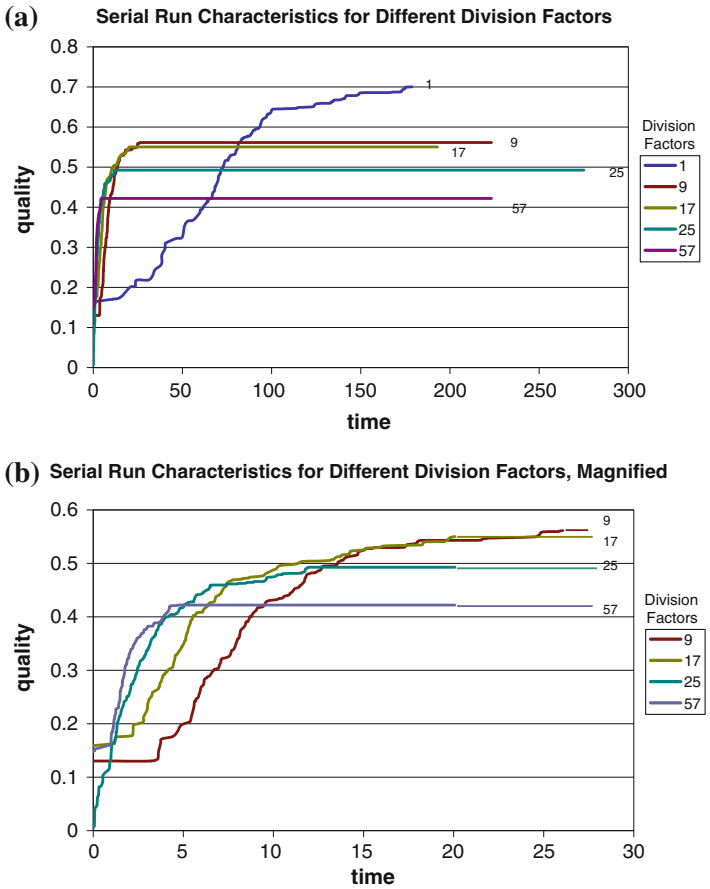


Fig. 7 Quality versus runtime results for Serial SA, with different values for M

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:

1. For the first 100 or so annealing iterations, an average of the quality improvement per Metropolis function call accumulates. 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 the 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 found empirically to dramatically improve algorithm runtimes, without sacrificing final quality achieved.

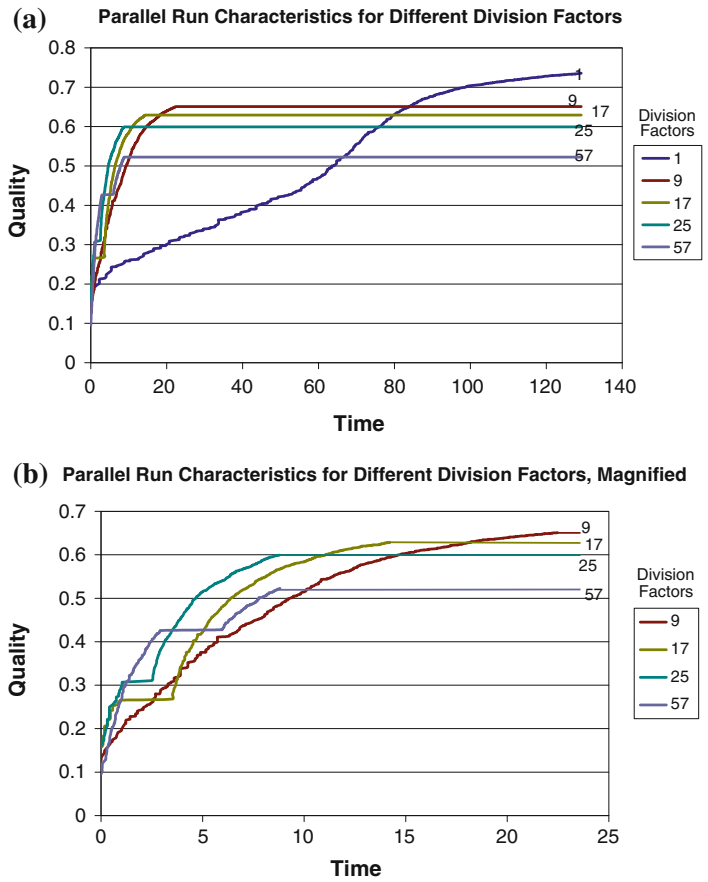


Fig. 8 Quality versus runtime results for AMMC parallel SA (7 processors), with different values for M

Table 4 Results for adaptive Strategy 4 (Strategy 1 qualities)

Circuit name	Number of cells	μ(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

The runtimes 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 runtimes 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 runtimes have improved dramatically over Strategy 1, while the parallel runtimes are largely equivalent to those of Strategy 2.

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 seven processors, 0.764924 for s1238 on eight processors, 0.708843 for s1488 on six processors, and 0.704714 for s1494 on eight 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 (Fig. 9a).

Table 5 Results for adaptive Strategy 4 (Strategy 2 qualities)

Circuit name	Number of cells	μ(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

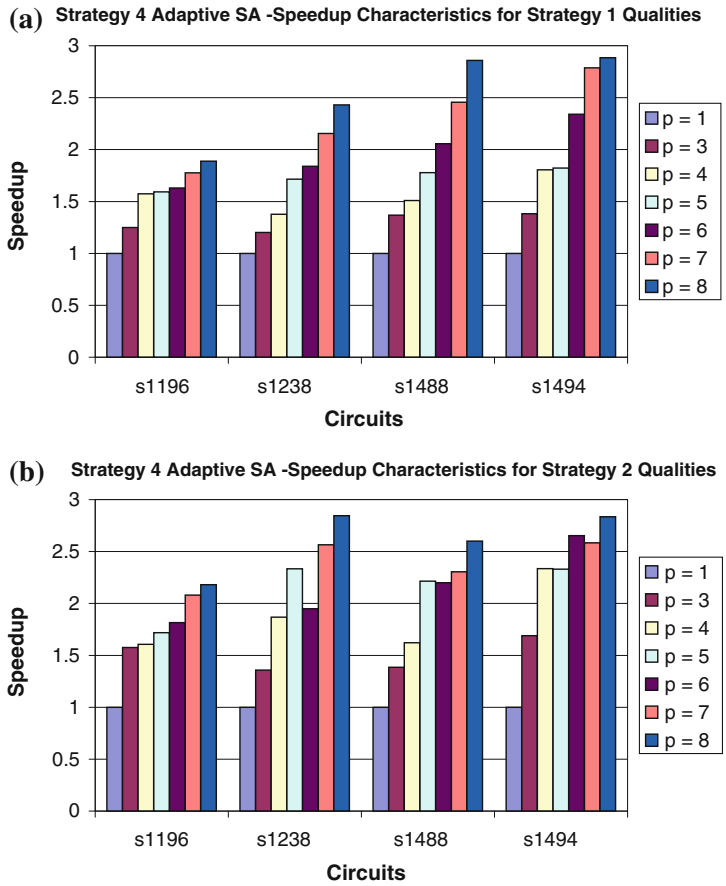


Fig. 9 Speedup characteristics of parallel adaptive simulated annealing (Strategy 4) for solution qualities of **a** Strategy 1, **b** Strategy 2

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

Overall, our results for this strategy have been quite promising for our environment and problem instance. It is possible that it could prove equally useful for other problem types and environments, particularly since this approach is independent of the characteristics of the cost-function, nor does it modify the nature of the parallel algorithm (i.e. does not affect communication schedule, and all modifications are equally applicable to the serial version of SA). Exploration of this aspect, however, falls outside the scope of this paper.

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 multiobjective 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 multiobjective 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 (from an algorithmic point of view) and occur only when necessary, as opposed to the synchronous MMC model that involves barrier synchronization. Thus the impact of communication delays on the runtime performance of these approaches is minimal. This can be verified from Fig. 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. The seven rows in Fig. 10 show the process timeline for seven processors where process '0' is the process indicator for processor '0', i.e. master processor, and likewise. The red color shows the time spent in MPI communication that is sending, waiting and receiving the solutions. The green color indicates the application progress on the processor while the black colored lines connecting all non-zero processes with Process '0' indicate the communication being carried out between processors. The figure also illustrates the asynchronous nature of inter-processor communication since the different processors are communicating with master processor at different times.

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.

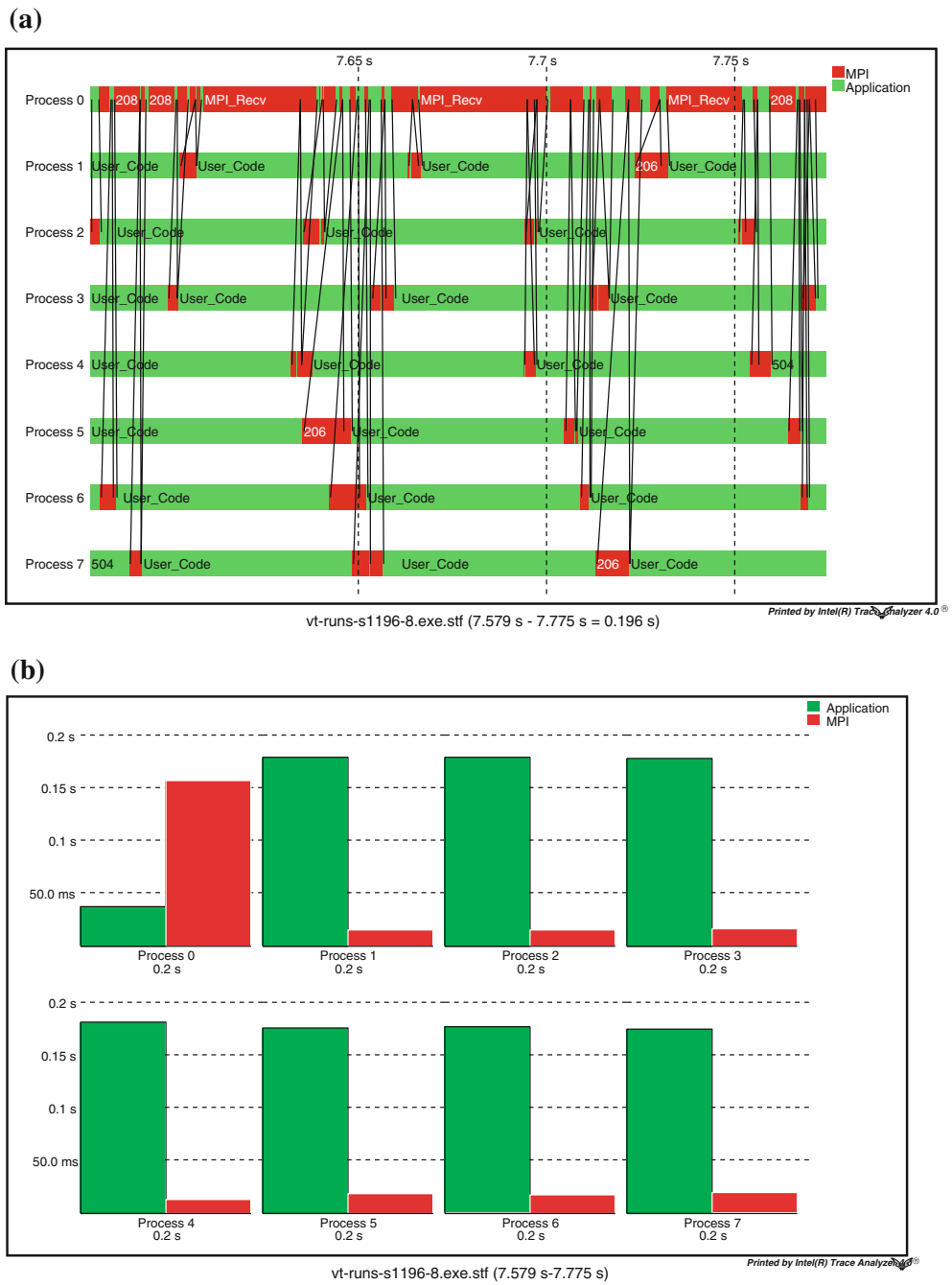


Fig. 10 **a** Communication versus computation traces for all processors for Type-3 parallel SA. **b** Ratio of communication to computation for each processor for Type-3 parallel SA

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 four 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 runtime 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 <3.

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 multiobjective 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 eight 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. We believe that this adaptive approach merits further exploration of its applicability to other problem instances and parallel environments. In addition, we shall also consider other modifications to the cooling schedule of simulated annealing, such as very-fast simulated re-annealing, simulated quenching, and mean-field annealing, etc. [21]. 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.

Acknowledgments 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 Mohammed Faheemuddin in the editing and review of this manuscript.

References

1. Sait SM, Youssef H (1999) Iterative computer algorithms with applications in engineering: solving combinatorial optimization problems. IEEE Computer Society Press, California
2. Banerjee P (1994) Parallel algorithms for VLSI computer-aided design. Prentice-Hall, Englewood Cliffs
3. Cung V-D, Martins SL, Riberio CC, Roucairol C (2001) Strategies for the parallel implementation of metaheuristics. In: Essays and surveys in metaheuristics. Kluwer, Dordrecht, pp 263–308
4. Crainic TG, Toulouse M (2003) Parallel strategies for metaheuristics. In: Glover FW, Kochenberger GA (eds) Handbook of metaheuristics, pp 465–514
5. Witte EE, Chamberlain RD, Franklin MA (1991) Parallel SA using speculative execution. IEEE Trans Parallel Distributed Syst 2(4)
6. Lee S-Y, Lee KG (1996) Synchronous and asynchronous parallel simulated annealing with multiple-Markov-chains. IEEE Trans Parallel Distributed Syst 7(10):993–1008
7. Chandy J, Kim S, Ramkumar B, Parkes S, Bannerjee P (1997) An evaluation of parallel simulated annealing strategies withapplication to standard cell placement. IEEE Trans Comput Aided Des Integrated Circuits Syst 16(4):398–410
8. Sait SM, Zaidi AM, Ali MI (2006) Asynchronous MMC based parallel SA schemes for multiobjective standard cell placement. In: Proceedings of 2006 international symposium in circuits and systems (ISCAS), pp 4615–4618
9. Toulouse M, Crainic TG (2002) State-of-the-art handbook in metaheuristics. In: Parallel strategies for metaheuristics. Kluwer Academic Publishers, Dordrecht
10. Kravitz SA, Rutenbar RA (1987) Placement by simulated annealing on a multiprocessor. IEEE Trans Comput Aided Des 6(4):534–549
11. Jayaraman R, Darema F (1988) Error tolerance in parallel simulated annealing techniques. In: Proceedings of the 1988 IEEE international conference on computer design: VLSI in computers and processors, pp 545–548
12. Durand MD, White SR (2000) Trading accuracy for speed in parallel simulated annealing with simultaneous moves. High Perform Comput Oper Res 26(1):135–150
13. Banerjee P, Jones MH, Sargent JS (1990) Parallel simulated annealing algorithms for standard cell placement on hypercube multiprocessors. IEEE Trans Parallel Distributed Syst 1:91–106
14. Casotto A, Romeo F, Sangiovanni-Vincentelli A (1987) A parallel simulated annealing algorithm for the placement of macro-cells. IEEE Trans Comput Aided Des CAD-6:838–847

15. Sun WJ, Sechen C (1994) A loosely coupled parallel algorithm for standard cell placement. In: Digest of papers, International conference on computer-aided design, pp 137–144
16. Sait SM, Youssef H, Hussain A (1999) Fuzzy simulated evolution algorithm for multiobjective optimization of VLSI placement. In: IEEE congress on evolutionary computation, July 1999, pp 91–97
17. Devadas S, Malik S (1995) A survey of optimization techniques targeting low power VLSI circuits. In: 32nd ACM/IEEE design automation conference
18. Chandrakasan A, Sheng T, Brodersen RW (1992) Low power CMOS digital design. J Solid State Circuits 4(27):473–484
19. Yager RR (1988) On ordered weighted averaging aggregation operators in multicriteria decision making. IEEE Trans Syst Man Cybern 18(1)
20. Konishi K, Taki K, Kimura K (1995) Temperature parallel simulated annealing algorithm and its evaluation. Trans Inf Process Soc Jpn 36(4):797–807
21. Ingber L (1993) Simulated annealing: practice versus theory. J Math Comput Model 18(11):29–57