PARALLELIZATION OF STOCHASTIC EVOLUTION

BY

KHAWAR SAEED KHAN

A Thesis Presented to the
DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER ENGINEERING

May 2006

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS

DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **Khawar Saeed Khan** under the direction of his thesis advisor and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE In Computer Engineering.**

THESIS COMMITTEE

Dr. Sadiq M. Sait (Chairman)

Dr. Mohammed Al – Suwaiyel (Member)

Dr. Aiman El – Maleh (Member)

Department Chairman
Dr. Adnan A. Gutub

Dean of Graduate Studies
Dr. Mohammad A. Al-Ohali

25/03/07
Date

Dedicated to

My Beloved Parents, Sisters and Brother

ACKNOWLEDGEMENTS

All sincere praises and thanks are due to Allah (SWT), for His limitless blessings on us. May Allah bestow his peace and blessings be upon his Prophet Mohammad (P.B.U.H.) and his family. Acknowledgements are due to King Fahd University of Petroleum & Minerals for providing the computing resources for this research.

I would like to express my profound gratitude and appreciation to my thesis committee chairman Dr. Sadiq M. Sait for his indomitable support and advice. Without his sheer guidance and suggestions, this work would have become a daunting task. His extensive knowledge and experience made it possible to shape the thesis. My unrestrained recognition also goes to the committee members, Dr. Aiman El-Maleh and Dr. Mohammad H. Al-Suwaiyel for their interest, invaluable cooperation and support. Their continuous support, advise and encouragement can never be forgotten. Also, I would like to express my deepest thanks to every instructor who contributed in building my knowledge and experience. Thanks are also due to faculty and staff members of Computer Engineering Department for their cooperation.

I would like to render my profound acknowledgements to my friends Faheem, Mustafa, Ali and Sanaulla for the countless hours that they spent on discussing and solving problems that I faced and also to Mr. Hamid Kadry for helping me in thesis abstract translation. Acknowledgements are due to my friends and colleagues who made my stay at the university a cherished and an unforgettable era. In particular I

would like to express my gratefulness to Moin BHAI, Saad Bhai, Ahmer Bhai, Kashif Khan, Aiman Rashid, Syed Adnan Shahab, Mudassir Masood, Imran Naseem, Imran Azam, Zeeshan, Saad Mansoor, Adnan Yusuf, Sheikh Faisal and Mazhar for their ever helpful natures.

Finally, I also thank my beloved parents, and all my family members for their moral support throughout my academic career. They have always helped me spur my spirits and encourage me throughout my life with their inspiriting and hortatory words. Their celebrations of my minutest successes elate me to the pinnacle of confidence. Their prayers, patience, guidance and support lead to the successful accomplishment of this thesis.

Finally, thanks to everybody who contributed to this achievement in a direct or an indirect way.

Contents

List of Tables

List of Figures

THESIS ABSTRACT

Name: Khawar Saeed Khan

Title: Parallelization of Stochastic Evolution

Major Field: COMPUTER ENGINEERING

Date of Degree: May 2006

The complexity involved in VLSI design and its sub-problems has always made them ideal application areas for non-deterministic iterative heuristics. However, the major drawback has been the large runtime involved in reaching acceptable solutions especially in the case of multi-objective optimization problems. Among the acceleration techniques proposed, parallelization of iterative heuristics is a promising one. The motivation for Parallel CAD include faster runtimes, handling of larger problem sizes, and exploration of larger search space. In this work, the development of parallel algorithms for Stochastic Evolution, applied on multi-objective VLSI cell-placement problem is presented. In VLSI circuit design, placement is the process of arranging circuit blocks on a layout. In standard cell design, placement consists of determining optimum positions of all blocks on the layout to satisfy the constraint and improve a number of objectives. The placement objectives in our work are to reduce power dissipation and wire-length while improving performance (timing). The parallelization is achieved on a cluster of workstations interconnected by a low-latency network, by using MPI communication libraries. Circuits from ISCAS-89 are used as benchmarks. Results for parallel Stochastic Evolution are compared with its sequential counterpart as well as with the results achieved by parallel versions of Simulated Annealing as a reference point for both, the quality of solution as well as the execution time. After parallelization, linear and super linear speed-ups were obtained, with no degradation in quality of the solution.

Keywords: *Parallelization, Multi-Objective Optimization, Stochastic Evolution, Cluster Computing, VLSI Physical Design, Low-Power, Performance, Placement*

MASTER OF SCIENCE DEGREE

King Fahd University of Petroleum & Minerals, Dhahran, Saudi Arabia

May 2006

ملخص الرسالة

الاسم : خاور سعيد خان

عنوان الرسالة : تطوير خوارزم للتوازي العشوائي
المتكاملة فائقة السعة ذات القدرة المنخفضة

التخصـــــص : هندسة الحاسب الآلي

تاريخ التخـرج : مايو 2006م .

إن درجة التعقيد في تصميم الدوائر المتكاملة فائقة السعة وما ينتج عنها من مشاكل فرعية جعلت منها مجالاً خصباً للتطبيقات المتعلقة بالاكتشاف الذاتي التكراري غير المحدد ، إلا أن العيب الرئيسي في هذه الطريقة يتمثل في وقت التنفيذ المتزايد المرتبط بالوصول إلى حلول مقبولة وخاصة في حالة مسائل التحسين متعدد الأهداف ، ومن بين طرق التسريع المقترحة خوارزم التوازي للاكتشاف الذاتي التكراري والتي تبدو كطريقة واعدة .

أن الدافع لاستخدام خوارزم التوازي للتصميم بمساندة الحاسب يشمل سرعة التنفيذ ، تناول مسائل ذات أحجام أكبر ، إضافة إلى استكشاف فضاء أوسع للبحث . وفي هذا البحث ، فإن تطوير خوارزم التوازي العشوائي وتطبيقه على مشكلة تحديد موقع الخلية المعيارية للدوائر المتكاملة فائقة السعة هي عملية ترتيب لمكونات الدائرة على المخطط . وفي التصميم القياسي للخلية ، فإن ذلك يشمل التحديد الأمثل لأماكن كافة المكونات على المخطط وفقاً للقيود وتحسيناً لعدد من الأهداف . إن أهداف تحديد الموقع للخلية في هذا البحث تشمل خفض استهلاك القدرة وأطوال المواصلات مع تحسين معدل الأداء (وقت التنفيذ) . ويتحقق خوارزم التوازي باستخدام حزمة من محطات العمل للحاسبات المرتبطة بشبكة قليلة الانتظار ، والاستعانة "وصلة بينية لمرور الرسائل" كبرامج اتصال فيما بينها . كما تم استخدام دوائر قياسية لمعيار المقارنة للأداء وفق المواصفات القياسية "للندوة الدولية للدوائر والنظم – 89 لمعهد مهندسي الكهرباء والإلكترونيات" .

وتم مقارنة نتائج تطوير خوارزم التوازي العشوائي مع نظيره باستخدام معالج متسلسل وكذلك مع النتائج التي تم تحقيقها مع النسخ المتوازية بطريقة التلدن التماثلي كنقطة مرجعية لجودة الحل ووقت التنفيذ . وبعد استخدام خوارزم التوازي فقد تم الحصول على المعدل الخطي وفائق الخطية للزيادة في معدلات التسارع للحل دون المساس بجودته .

مفاتيح البحث : خوارزم التوازي . التحسين متعدد الأهداف . التطوير العشوائي . حزم الحاسبات . التصميم الفيزيائي للدوائر المتكاملة فائقة السعة . القدرة المنخفضة . معدلات الأداء . تحديد الموقع .

درجة الماجستير في العلوم

جامعة الملك فهد للبترول والمعادن . الظهران . المملكة العربية السعودية

مايو 2006م

Chapter 1

INTRODUCTION

The process of mapping structural representation of a circuit into its layout representation is termed as **physical design of VLSI circuits**. In structural representation a system is defined in terms of logic components and their interconnects whereas layout representation describes the circuit in terms of a set of geometric object which specify the dimensions and locations of transistors and wires on a silicon surface. As the module count on a chip grows, the quality and speed of automatic layout algorithms need to be re-evaluated. Figure 1.1 shows the abstraction of design steps involved in chip manufacturing.

One of the most critical problems encountered in the design of VLSI circuits is how to assign locations to circuit modules and to route the connections among them such that the ensuing area is minimized. Due to the complexity of this problem, it is partitioned into two consecutive stages. The first deals with assigning locations to

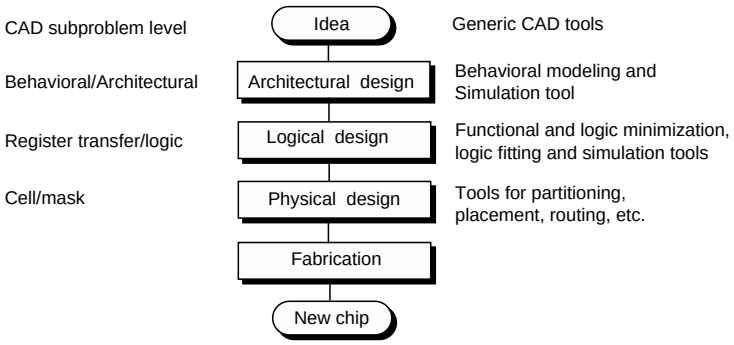

Figure 1.1: Levels of abstraction and corresponding design steps.

individual modules and is commonly referred to as the **Placement problem**. The second involves routing of the connections among the already positioned modules. The quality of the routing obtained at the second stage depends critically on the placement output of the first stage. Hence, the goal of a good placement techniques is to position the cells such that the ensuing area is minimized, while the wire lengths are subject to critical length constraints. The optimization of this placement problem is also referred to as combinatorial optimization problem.

Though certain metaheuristics provide different strategies for finding the approximate solutions to the combinatorial optimization problem but the complexity of this problem still makes it a very time consuming problem, where targeting a better quality costs nothing but extra time. Thus, the computation time associated with the exploration of solution space itself becomes a critical factor when a circuit with large number of modules is optimized. Multi-objective optimization furthers the complexity of the problem by increasing the computation time. To combat the

computation time issue, certain fast metaheuristics have been proposed that reduces the computation time but still these fast metaheuristics do not cope with the speed of increase in module count.

1.1 Iterative Non-deterministic Heuristics

Brute force techniques are never a solution in cases where problem instances have large module counts. The solution space in these combinatorial problems are always non-deterministic. Thus, a searching criteria that accepts good solutions only (greedy algorithms) when reaches a local optima have no chance to escape or proceed. Certain techniques apply random walk to perform well in these environments but studies have shown that given the limited CPU time algorithms that apply random walk performs worst [1]. Iterative heuristics have played an important role in this case and have produced some better results.

Sequential Iterative Heuristics

Following are the five dominant algorithms that are instance of *general iterative non-deterministic algorithms*,

1. Genetic Algorithm (GA)
2. Tabu Search (TS)
3. Simulated Annealing (SA)

4. Simulated Evolution (SimE) and

5. Stochastic Evolution (StocE)

We have categorized the above mentioned five (5) heuristics as 'sequential iterative heuristics' because these heuristics were developed and thus meant to be run on a single processor environment. These heuristics have certain common properties which are as follows [1]:

1. They are blind, in that they do not know when they reached the optimal solution. Therefore, they must be told when to stop

2. They are approximation algorithms, that is, they do not guarantee finding an optimal solution

3. They have 'hill climbing' property, that is, they occasionally accept uphill (bad) moves

4. They are easy to implement. All that is required is to have suitable solution representation, a cost function, and a mechanism to traverse the search space

5. They are all 'general'. Practically they can be applied to solve any combinatorial optimization problem

6. They all strive to exploit domain specific heuristic knowledge to bias the search towards 'good' solution subspace. The quality of subspace searched depends to a large extent on the amount of heuristic knowledge used

7. Although they asymptotically converge to an optimal solution, the rate of convergence is heavily dependent on the adequate choice of several parameters

The performance of these heuristics depends heavily on the cost function and search space traversing mechanism. The control parameters configuration and tuning does also have a major and critical impact on the performance. However, as mentioned earlier, the run-times of these sequential heuristics depends critically on their execution platform which impacts the run-times of these heuristics. These run-times becomes critical when we focus on combinatorial problems with very large number of modules where choice of a heuristic for a particular problem instance will be based on their runtime values. Any reduction in run times makes the specific heuristic a prior heuristic when compared to others for that specific problem.

With the proliferation of parallel computers, powerful workstations, and fast communication networks, parallel implementations of meta-heuristics appear quite naturally as an alternative to modifications in algorithm itself for speedup. Several parallel strategies have already been proposed for different meta-heuristics and applied to different problems. Moreover, parallel implementations do allow solving large problem instances and finding improved solutions in lesser times, with respect to their sequential counterparts. This advantage of parallel implementation comes from the possible partitioning of the search space and more possibilities for search intensification and diversification.

1.2 Parallel CAD

1.2.1 Motivation

Despite the advances in VLSI technology, there are still a few challenges that pose an obstacle in its rapid development. One of them is the large run-time required for iterative heuristics which play a crucial role in VLSI design. Of the various acceleration strategies attempted, parallel computing has always exhibited the most potential. Not only is it possible to achieve shorter run-times with parallel processing but also handle larger problem sizes, obtain better quality results, etc. These potential advantages are enumerated and detailed below [2]:

1. **Faster Run-times:** Most of the VLSI design problems are computationally intensive and take a large amount of time ranging from several hours to days. Moreover, future design tools will require even more computational capabilities. Given such increased requirements for speed and accuracy, parallel processing is the only way to accelerate the design tasks.

2. **Larger Problem Sizes:** Sometimes, due to time or memory limitations, these design tools cannot handle larger problem sizes. This can be overcome by using parallel processing, as both computational speed and memory size are enhanced by using parallel architectures.

3. **Better Quality:** As most of the VLSI design problems are NP complete [3], heuristics used to solve them may give non-optimal solutions. The solutions obtained are function of the fraction of the search space traversed. With the use of parallel search techniques, better quality results can be obtained. This is possible as a larger search space can be traversed in the same time constraint.

4. **Cost-effective Technology:** With the proliferation of parallel computers, powerful workstations, and fast communication networks, parallel implementation of iterative heuristics, seem to be a natural alternate to speedup the search for approximate solutions.

Combinatorial optimization problems have historically been the target application areas for non-deterministic heuristics. As already mentioned, these domains, with vast multi-modal search spaces can not reliably be navigated by deterministic algorithms, and hence rely on conceptually simple, yet robust stochastic methods. Among the more popular of these are Simulated Annealing [4], Genetic Algorithm [1], Tabu Search [5], and to a lesser extent, Simulated Evolution [6] and Stochastic Evolution [7]. However, given the invariably increasing problem complexities and conflicting optimization objectives, the run-times of with these algorithms for achieving optimal solutions can be prohibitively high. Though there are several domain and application-specific acceleration strategies, one generic and effective approach is to explore parallelization methods.

By leveraging the constantly decreasing 'Cost to Computing Power' ratio through off-the-shelf computer clusters, and given efficient parallelization strategies, it is possible to achieve several magnitudes of speedup. The latter part of the above equation is of special significance, as determining an appropriate parallel approach can be a non-trivial exercise. The factors to be considered are the nature of the problem domain, the solution landscape, the heuristic itself and slightly less obvious, the kind of computing environment, whether distributed or shared-memory [8].

Parallelization of heuristics has been a well-research topic, with extensive efforts on Simulated Annealing [4] [9] [10], Genetic Algorithms [11] [12] [13], Tabu Search (TS) [5] [14] [15] among others. Sequential StocE reported in literature [7], demonstrated improvements in runtime and solution quality over Simulated Evolution and Genetic Algorithms [16]. However, parallelization of StocE hasn't received much attention, very likely because of its highly sequential nature.

1.3 Parallel Processing Models

Parallelization strategies can be implemented in many different ways depending on the problem to be parallelized. But in the end all software parallelization strategies can majorly be categorized in two (2) ways depending on the their basic infrastructure. These two (2) categories are as follows:

1. Shared Memory Parallelization

A single computer system utilizing multiple CPUs that share access to a common set of memory addresses is defined as a Shared Memory Parallel system. This parallelization assumes the following:

- All processors can access all the memory in the parallel system, i.e., there is only one address space accessible by all
- The time to access the memory may not be equal for all processors
- Parallelizing on a shared memory procedure does not reduce CPU time - it reduces wallclock time
- Parallel execution is achieved by generating threads which execute in parallel and
- Number of threads is independent of the number of processors

This parallelization is more a fine-grained approach to parallel computing that involves creating independent 'threads' of execution within one process rather than passing messages among many separate processes. This alternative may be more efficient but is much more complex to program. OpenMP is the most commonly used and accepted tool or language for parallelizing a code for SMP.

2. Distributed Memory Parallelization

Multiple single-CPU computers connected over a high speed network to process a single program is known as a Distributed Memory Processing (DMP)

system. This approach has proven to be very successful at solving extremely large problems and is popular within the University research and high energy physics communities. The typical hardware configuration is a group of commodity (often Intel-based) PCs with lots of memory connected via high speed ethernet. This configuration, dubbed a 'Beowulf' class system, passes instructions and data between systems via Message Passing Interface (MPI) libraries, a portable, easy to use system of exchange.

1.4 Message Passing Interface(MPI)

1.4.1 Introdution

In April of 1992, a group of parallel computing vendors, computer science researchers, and application scientists met at a one-day workshop and agreed to cooperate on the development of a community standard for the message-passing model of parallel computing. The MPI Forum that eventually emerged from that workshop became a model of how a broad community could work together to improve an important component of the high performance computing environment. The Message Passing Interface (MPI) definition that resulted from this effort has been widely adopted and implemented, and is now virtually synonymous with the message-passing model itself. MPI not only standardized existing practice in the service of making applications portable in the rapidly changing world of parallel

computing, but also consolidated research advances into novel features that extended existing practice and have proven useful in developing a new generation of applications. [17]

In short, the standard message-passing interface (MPI) library is a way to share data among parallel processes running on distributed-memory parallel computers.

Parallelization Effects

MPI works on the basic parallelization principle defined by Amdahl's law, which states "If p is the fraction of your program that can be parallelized (and 1-p is the fraction that cannot), and if you run it on n processors, then the ideal parallel running time will be

$$((1-p)+\frac{p}{n}) \times (serial\_running\_time)''$$

This suggests the importance of carefully identifying the fraction of the code that can be parallelized, since it sets a limit on improvements in how fast the parallelized program will run. The effectiveness of parallelization also depends on how well the program's many processes communicate with each other. Effective bandwidth is one way to collectively assess the many factors that influence interprocess communication.

In any parallelization the major objective is always to achieve the speed-up.

Speed-up thus remains the evaluating parameter for any parallelized program. To improve speed-up in any parallelization strategy, following are the key factors:

1. Decrease the amount of data sent between processes and
2. Decrease the number of times data is sent

Parallelizing I/O Operations

There are two (2) cases in parallelizing I/O operations. These two (2) cases are describes below:

Input Cases: For a massively parallel program, there are three ways to handle data input among multiple processes:

1. All processes read the same input file from a shared file system (if there is one)
2. All processes have a local copy of the input file before computation starts and
3. One process reads the input file and distributes it to the others using appropriate MPI library routines

Output Cases: For a massively parallel program, there are three ways to handle data output from among the many processes:

1. All processes write to a standard output

2. One process gathers all the data and writes it to a local file and

3. Each process writes its data sequentially to a shared file

Future of MPI

MPI was deliberately designed to grant considerable flexibility to implementors, and thus provides a useful framework for implementation research. Successful implementation techniques within the MPI standard can be utilized immediately by applications already using MPI, thus providing an unusually fast path from research results to their application. At Argonne National Laboratory MPICH, a portable, high performance implementation of MPI, has been developed and distributed from the very beginning of the MPI effort. Now MPICH-2, a completely new version of MPICH is being released. This hopefully will stimulate both, further research and a new generation of complete MPI-2 implementations, along with some early performance results. A speculative look at the future of MPI, including its role in other programming approaches, fault tolerance, and its applicability to advanced architectures is also expected shortly [17].

1.5 Thesis Organization

This thesis is organized as follows: In Chapter 2, theoretical aspects of parallelizing heuristics are discussed along with a review of related literature for Stochastic

Evolution and the survey of some parallel models developed for other heuristics. Problem formulation is dealt with in Chapter 3, where parallel environment, hardware, software, and paradigm are discussed. Analysis of the sequential Stochastic Evolution is done in Chapter 4 followed by design methodology of developing parallel algorithms. This chapter also reports the parallel models of Stochastic Evolution developed during the course of this work. Experimental results of the research are presented in Chapter 5. Chapter 6 shows the comparison of achieved results with reported results for other heuristics. Conclusion and Future work are discussed in Chapter 7.

Chapter 2

LITERATURE REVIEW

Literature survey of parallel Stochastic Evolution reveals the absence of any research efforts in this direction where extensive literature and research is found on parallelization of similar iterative heuristics. This lack of research at a time presented a challenging task of implementing any parallelization scheme for Stochastic Evolution as well as a vast room for experimentation and evaluation. Thus for parallelizing Stochastic Evolution it was natural to do a comprehensive literature survey of parallel strategies that performed well on similar evolutionary heuristics. Though it was never guaranteed that parallel strategies performing well for other similar heuristics would also perform better for Stochastic Evolution but implementation and analysis was the only combination that was required to achieve a parallel strategy that delivers for Stochastic Evolution.

This chapter presents the literature survey of parallel strategies implemented for

other evolutionary iterative heuristics which led to the inspiration of parallelizing stochastic evolution.

2.1 Stochastic Evolution Applications

Although, not extensively implemented when compared to other heuristics like Simulated Annealing, StocE still has been used in optimizing several problem instances. The implementation always remains problem specific but the following general steps are required to implement StocE for any problem instance:

1. Solution space definition
2. Suitable state representation
3. Identification of the notions of cost and perturbations
4. Initial value for control parameter p and method to update it and
5. Value for stopping criterion

StocE applications cover diverse problems, many of which have been reported in literature such as the Network Bisection problem, the Travelling Salesman problem, Hamiltonian Circuit problem, as well as in a recent study on the evolution of neural information processing [18]. Some of these are discussed below:

1. **Graph covering problem**

   Stochastic Evolution algorithm is applied to solve the graph covering problem where a set of patterns that fully covers a subject graph with a minimal cost is sought. This problem is a typical constrained combinatorial optimization problem and is proven to be NP-complete. Many branch-and-bound algorithms with different heuristics have been proposed. But most of them cannot handle practical sized problems like the technology mapping problem from VLSI synthesis. Experimental results with some selected benchmark circuits show that *StocE* produces better results than the traditional tree mapping algorithm within a reasonable range of runtime. Experiences from this work show that *StocE* can be a good alternative for constrained optimization problems like graph covering [19].

2. **StocE based register allocation using multiport memories**

   In data path synthesis, intermediate outputs of functional blocks are stored in registers. Allocation of physical resources (register files) to registers is done by the designer. In some High Level Synthesis systems, memory ports are allocated to registers with disjoint access times. In this problem, a *Stochastic Evolution* based approach to Register Allocation using multiport memories is used. In the StocE based approach for allocation of registers to multiport memories, the procedure works towards minimizing the interconnection be-

tween memory ports and the functional units, while placing constraints on access time requirements of registers. This approach could be used in design space exploration to determine how many read-write ports per bank would best suit the application. The algorithm was implemented and tested on standard benchmarks yielded good results [20].

3. **Scheduling-based CAD methodology for low-power ASIC design-space exploration**

   This problem described a novel approach to scheduling with multiple supply voltages in the high-level synthesis of ASICs. In a significant shift from existing scheduling algorithms for multiple voltages, the proposed approach exploits the maximal parallelism available in an initial schedule, and applies a modified *Stochastic Evolution* to iteratively improve, or re-schedule, the previously obtained best schedule to reduce the maximal power consumption of function-units [21].

2.2 Stochastic Evolution (StocE)

Definition

The state model: Given a finite set M of movable elements and a finite set L of locutions, a state is defined as a function S : M + L satisfying certain state-constraints.

Also, each state S has an associated cost given by COST(S) [7].

StocE Algorithm

Stochastic Evolution algorithm, shown in Figure 2.1, seeks to find the global minimum in a given search space. During the search if the algorithm gets locked into a local minimum, it comes out of it by accepting bad solutions. This acceptance of good and bad solutions is probabilistic where the good moves are always accepted with probability one (1) and bad moves may also be accepted or rejected based on certain probability. This probabilistic decision of accepting or rejecting the bad moves is what makes this algorithm stochastic. The algorithm as discussed earlier is an iterative algorithm that searches for the solutions within the constraints while minimizing or maximizing the objective function as desired. The algorithm is blind in a sense that it needs to be told when to stop.

Algorithm requires the following as inputs:

1. An initial solution
2. A range variable p_0 and
3. A Termination parameter R

At start, the algorithm saves the initial solution as best solution and current solution. The cost for the initial solution is calculated and again this cost is saved as best cost and current cost. A parameter ρ, initially equal to zero, is defined and

```
AlgorithmStocE(S_0, p_0, R);
Begin
    BestS= S = S_0;
    BestCost= CurCost= Cost(S);
    p = p_0;
    ρ = 0;
    Repeat
        PrevCost= CurCost;
        S = PERTURB(S, p);
        /* perform a search in the neighborhood of s */
        CurCost= Cost(S);
        UPDATE(p, PrevCost, CurCost);
        /* update p if needed */
        If (CurCost< BestCost) Then
            BestS=S;
            BestCost= CurCost;
            ρ = ρ − R;
            /* Reward the search with R more generations */
        Else
            ρ = ρ + 1;
        EndIf
    Until ρ > R
    Return (BestS);
End
```

Figure 2.1: The Stochastic Evolution algorithm.

another parameter p is defined equal to p_0. This main loop runs till the value of ρ is less than the termination parameter R. The algorithm then enters into a its main loop where current cost of solution is saved as previous cost and then a function ***PERTURB***, shown in Figure 2.2, is invoked.

In ***PERTURB*** function, the algorithm enters into a second loop that for each main iteration runs for total number of movable elements in the given problem. This is what is termed as a compound move of stochastic evolution. ***MOVE*** function is called inside the loop which makes a simple move by moving one movable element to a new location. This movement changes the whole state of the solution thus cost of this new solution is calculated again. Gain is calculated by subtracting the new cost from the previous cost. If the gain calculated is positive, i.e., the new solution is better than the previous solution if cost minimization is our objective, then the new solution is accepted. But if the gain calculated is negative, i.e., the new solution is worse than the old solution, then a negative random number is generated between zero (0) and the range variable p, where range variable is also negative. If this negative gain is greater than the random number generated the solution is accepted else the solution is rejected. At the end of each simple move ***MAKE_STATE*** routine is called that makes sure the solution accepted does not violate any constraints. If any constraint is violated, then the algorithm takes few steps back and accept the solution within the constraints.

The algorithm enters into the main loop again after the completion of a com-

pound move by the ***PERTURB*** function. In the main loop, the cost of the accepted solution is calculated and is saved as current cost then ***UPDATE*** procedure is called where the previous and current costs are compared. If found equal the range variable p is incremented by p_{incr} and if the two (2) values not found equal than p is re-initialized by its initial value p_0 again. ***UPDATE*** procedure is shown in Figure 2.3.

After returning from ***UPDATE*** procedure, the algorithm compares the current cost and the best cost. If the current cost is found better than the best cost, the solution returned by the ***PERTURB*** function, i.e., the current solution, is saved as the best solution and its cost, i.e., the current cost, is saved as the best cost. Also, the algorithm awards itself on finding a good solution by decrementing the value of ρ by R else ρ is incremented by one (1) in each iteration and the algorithm continues to search for better solutions till ρ becomes equal to R.

It is clear that the control parameters like p_0, p_{incr} and R must be chosen carefully since they effect the behavior of algorithm and thus will effect the results. p_0 and p_{incr} are problem specific parameters whereas for R, it is shown in [7] that values ranging from $10 - 20$ are recommended.

FUNCTION PERTURB(S, p);
Begin
  ForEach $(m \in M)$ **Do**
  /* according to some apriori ordering */
    $S' = MOVE(S, m)$;
    $Gain(m) = Cost(S) - Cost(S')$;
    If $(Gain(m) > RANDINT(-p, 0))$ **Then**
      $S = S'$
    EndIf
  EndFor;
  S =MAKE_STATE(S);
  /* make sure S satisfies constraints */
  Return (S)
End

Figure 2.2: The PERTURB function.

PROCEDURE UPDATE(p, $PrevCost$, $CurCost$);
Begin
  If ($PrevCost$=$CurCost$) **Then**
  /* possibility of a local minimum */
    $p = p + p_{\text{incr}}$;
    /* increment p to allow larger uphill moves */
  Else
    $p = p_0$; /* re-initialize p */
  EndIf;
End

Figure 2.3: The UPDATE procedure.

2.3 Literature Review of Parallel Models for Stochastic Evolution

As already discussed, the absence of any work that addresses the parallelization of Stochastic Evolution presents a wealth of opportunity for experimentation. At the same time, this lack of related literature makes parallel StocE a challenging task. The problems that one must address are further highlighted given StocE's sequential nature. This section discusses tentative techniques that have been pointed to, specifically in the book by Sait and Youssef [1] and also draws from our experiences with Simulated Annealing and Simulated Evolution.

Because of the highly sequential nature of the StocE algorithm, the obvious parallelization approach would be to assign each processor a particular initial solution, and let each run StocE on it. This simple approach would be very good if the search subspaces of the various processors do not overlap (or have minimal overlap). In this case all processors would be concurrently searching distinct parts of the solution space. However, this would require that one has enough knowledge about the search space in order to partition it among the individual processors. In some instances, this can be a very unrealistic assumption, because very little will be known about the search space. For many problems, the subspace corresponding to the neighborhood of a particular solution is controlled by the algorithm designer (the state model, the parameter p, and the move operation). In many cases, it may be possible to tune

the algorithm for a particular problem instance so that the state space is searched in parallel (with minimal overlap) by several processors.

The logical flow in StocE is highly sequential, similar to that seen in Simulated Annealing. As such, we discuss plausible parallelization schemes within the same framework adopted for annealing. StocE parallel strategies can be classified under (1) *move acceleration*,(2) *parallel moves*,(3) *Markov Chains* and (4) *Row-Division*.

2.3.1 Move Acceleration

In move acceleration, a move is performed faster by distributing the various trial relocations on several processors working in parallel. The speed-up that can be achieved by this strategy depends to a large extent on the problem instance. Each simple move usually consists of several trial relocations of a particular movable element. The trial relocations can be performed in parallel without affecting the correctness of the algorithm. For problem instances where the window of trial relocations is large, sizable speed-up can be achieved.

2.3.2 Parallel Moves

In the *Parallel Moves* approach, several moves are performed in parallel, each executing on a single processor. A Master processor is often allocated which arbitrates the concurrent execution of p simple moves, where p is the number of processors. Figure 2.4 is a general description of a possible parallel stochastic evolution algo-

rithm following this parallelization strategy. Here the Master evaluates the outcome from all trials, and in case of no success, orders the parallel evaluation of p new trials; otherwise, it selects the best new current solution among the accepted solutions, and updates the state of all processors. This process repeats until the termination criteria is reached. At the end of the parallel execution of *PERTURB*, the master processor may be required to run the *MAKE_STATE* procedure to ensure a valid new state. The master is also in charge of updating the parameter p and the counter ρ.

The parallel algorithm given in Figure 2.4 assumes synchronous communication where the processors are forced to synchronize after each trial. A variation of this algorithm is an asynchronous model, wherein the various processors proceed asynchronously with their trials until at least one of them accepts a simple move. In this approach, the movable elements are distributed equally among the available processors, where each will be in charge of the trial relocations of its associated movable elements. Synchronization is forced only when one of the processors performs a successful trial. In this new variation communication will be less, and only when required. Furthermore, it is more efficient since no processor is forced to remain idle waiting for other processors with more elaborate trials to finish.

Both variations of this parallel algorithm can be implemented to run on a multicomputer or a multiprocessor machine. It is assumed that each processor must be able to set a common variable to *True* whenever it accepts a simple move; then the

```
Algorithm Parallel_StocE;
        /* S_0 is the initial solution */
Begin
        Initialize parameters;
        BestS=S_0; CurS=S_0; p = p_0;
            Repeat
                Communicate CurS and a movable element m_i to each processor i;
                ParFor each processor i
                    NewS^i=MOVE(CurS, m_i);
                    If Gain(CurS, NewS^i) > RANDOM(-p, 0)
                        THEN A_i = TRUE;
                EndParFor
                If Success Then
                    /* Success = (⋁_{i=1}^{p} A_i = True) */
                    Select(NewS); /* NewS is best solution among all NewS^i's */
                    If Cost(NewS) = Cost(CurS) Then p = p - 1;
                    Else p = p_0;
                    EndIf
                    If Cost(NewS) < Cost(BestS) Then
                        BestS= NewS;
                        ρ = ρ - R
                        Else ρ = ρ + 1;
                    EndIf
                EndIf
            Until ρ > R;
            Return (BestS)
End. /*Parallel_StocE*/
```

Figure 2.4: General parallel stochastic evolution algorithm where synchronization is forced after each trial.

solution accepted by the processor is communicated to a master processor which will force all other processors to halt and properly update the current solution.

Another possible approach would be to distribute the solution among multiple processors and allow them to proceed concurrently with their search as well as accept moves, with no interaction whatsoever. Algorithms following this strategy are known as *error algorithms*. The word error is used to highlight the fact that the processors have incorrect knowledge about the state of the parallel search. Studies on such parallelization schemes in the case of simulated annealing have indicated, that by limiting the number of concurrent moves or by ensuring that the moves are always *non-interacting*, error is minimized and convergence is maintained [22, 23]. However, it is not always clear how one can go about restricting the moves to be of a particular type.

Another approach would be to use the notion of a *serializable move set* introduced by Kravitz and Rutenbar [24] for the case of simulated annealing. The idea consists of restricting the set of concurrent moves to be *serializable*, i.e., a set of moves that would produce the same reject/accept decisions whether executed in parallel or in some serial order. For example, any set of rejected moves is a serializable set. Also moves that are completely non-interacting are serializable too. However, in general, the identification of the largest possible serializable subset of moves (to maximize speedup) is a very difficult problem. Kravitz and Rutenbar suggested instead a subclass of serializable move-sets that are easy to identify and referred to it as the

simplest serializable set. It is formed by taking a number of rejected moves and appending an accepted move. The expected size of the serializable move-set is a good estimation of the speedup, since it is a measure of the average number of trials that are evaluated concurrently. The problem with this approach is that the size of this set is controlled by the parameter p, which gets updated in an unpredictable way. This is unlike the case of simulated annealing where the size of this set is controlled by the value of the temperature which steadily decreases. For large values of p, the size of this set is very small (close to 1 always) leading to unacceptably low speedup (near 1). The reason is that, a large p is equivalent to the hot regime in simulated annealing, where almost all moves are accepted forcing the processors to communicate almost after each move. For small p, most simple moves are rejected thus allowing the various processors to run in parallel for most of the time. Because of the way this control parameter is updated, it is difficult to predict the speed up that may result with such a parallelization strategy.

The above two strategies can also be carried at the level of a compound move (a call to the function *PERTURB*), leading to (1) *perturbation acceleration*, and (2) *parallel perturbations*. In the former, a compound move is performed faster by distributing the various simple moves on several processors working in parallel. This is similar to the *parallel moves* approach discussed earlier. For the *parallel perturbations* approach, several perturbations are conducted in parallel. To ensure that the various processors do not search the same subspaces, one can instruct

the processors to run sequential StocE with different initial solutions. Once all the processors have converged, the best solution among all processors is selected. Then each processor reruns sequential stochastic evolution on a mutated version of the current best solution. Obviously, each processor must be assigned a different mutated solution. These steps are repeated until no significant improvement is obtained in k (for example $k = 2$) consecutive iterations.

2.3.3 Asynchronous Multiple Markov Chains (AMMC)

In this approach a managing node or server maintains the best cost and placement. At periodic intervals, processors query the server and if their current placement is better than that of the server, they export their solution to it; otherwise they import the server's placement. This removes the need for expensive synchronization across all processors. The managing node can either be involved in sharing computing load with its own annealing process or can be restricted to serving queries. For a very small number of processors, the server may also be involved with annealing, but in a scalable design, the server is better off servicing queries only.

The above defined approach does not divide the workload.

2.3.4 Row-Division

In this parallel strategy each processor is assigned two sets of non-overlapping rows which then uses these alternatively in each iteration. The advantage of this strategy

over AMMC or others is the effective workload division among the processors. The distribution keeps the communication cost low and increases the computation task of each processor.

These different strategies for StocE parallelization can be categorized within three types:

- Type 1: Parallelization may be obtained by concurrent execution of the operations or the concurrent evaluation of several moves making up an iteration of a search method.

- Type 2: Parallelism comes from the decomposition of decision variables into disjoint subsets. StocE is applied to each subset and the variables outside the subset are considered fixed. These strategies are generally implemented in master-slave framework.

- Type 3: This consists of several concurrent searches in the solution space. Each concurrent thread may or may not execute the same StocE parameters. They may start from the same or different initial solutions and may communicate during the search or only at the end to identify the best overall solution.

Besides the above explained parallel schemes, some parallel models of heuristics listed in Chapter 1 were also surveyed and are reported as follows:

- **Parallel Simulated Annealing**

  Applicability of three different parallel simulated annealing (SA) strategies to the problem of standard cell placement was investigated in [25]. The first strategy, parallel moves, based on the use of priorities and a dynamic message sizing, was used to deliver good consistent speedups with little degradation in the wire length. Multiple Markov chains appears to be promising as a means to achieving moderate speedup without losing quality, and in fact in some cases improving quality. Speculative computation, however, is shown to be inadequate as a means of parallelization of cell placement. A combination of the parallel moves approach with intermediate exchanges as in multiple Markov chains may offer benefits in terms of reducing the error present in the parallel moves approach alone.

  Following are the four (4) strategies for parallelizing simulated annealing [25]:

  1. Move Acceleration:

     In this approach, each individual move is evaluated faster by breaking up the task of evaluating a move 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 individual subtasks to different processors. Such approaches to parallelization are restricted to shared memory architectures and have limited

scope for large scale parallelism.

2. Parallel Moves:

   In this method, each processor generates and evaluates moves independently as if the other processors are not making any moves. One problem with this approach is that the cost function calculations may be incorrect due to the moves made by the other processors. This can be handled by either evaluating only moves that do not interact, or by handling interacting moves with some error tolerance procedure.

3. Multiple Markov Chains:

   Multiple Markov chains calls for the concurrent execution of separate simulated annealing chains with periodic exchange of solutions. This algorithm is particularly promising since it has the potential to use parallelism to increase the quality of the solution.

4. Speculative Computation:

   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 it does have the advantage of retaining the exact execution profile of the sequential algorithm, and thus the convergence characteristics are maintained.

- **Parallel Tabu Search**

  A parallel tabu search (TS) strategy for accelerating the solution to a constrained multiobjective VLSI cell placement problem is proposed in [26]. The proposed strategy belongs to p-control, RS, MPSS class and was implemented on a dedicated cluster of workstations where a distributed parallel Genetic Algorithm (GA) was also implemented for the comparison purposes. Experimental results on ISCAS-85/89 benchmarks exhibited that the proposed parallel TS shows an excellent trend in terms of speedup and requires far lesser run times as compared to serial TS for obtaining the same quality of placement solutions.

- **Parallel Genetic Algorithm**

  Different Genetic Algorithm (GA) parallelization strategies for multiobjective optimization problems are proposed in [27]. One approach described is a variation of the canonical Master-Slave parallel GA, with both fitness and crossover distributed among processors where selection is only implemented by the Master. Performance gains in terms of reduced run-time were seen only for larger circuits. On the other hand, another approach called Multi-Deme approach reported consistent performance gains independent of problem complexity and size of the search space.

Thus, as far as the parallelization of meta-heuristics is concerned, we found initial

efforts concentrated in the parallelization of Simulated Annealing (SA) [4] [9] [10] where the same approach of distributed parallel computing for cell placement have also been reported using Genetic Algorithm (GA) [11] [12] [13] and Tabu Search (TS) [5] [14] [15]. Moreover, parallelization of SA with respect to cell placement problem has also been an actively researched problem [28] [29] [30] [31] [32] with the same efforts of parallelization for Tabu Search (TS) [33] [34], Genetic Algorithm (GA) and Simulated Evolution (SimE) [35]. Considering cell placement problem, some of the above mentioned heuristics like SA, GA and SimE generated interesting time reduction trends when parallelized and compared to their sequential counterparts [35]. Much of research on parallelization effects has already been carried out on these heuristics but none is found for Stochastic Evolution.

Chapter 3

PROBLEM DESCRIPTION AND COST FUNCTIONS

In this chapter, the parallelization and cell placement problems are explained. Cost functions for the three objectives viz. power, delay and wire length are also formulated. Finally the experimental setup for the research conducted is described.

3.1 Problem Statement

The parallelization problem can be stated as follows: Given a combinatorial problem with an iterative heuristic applied on it to maximize the objective, a parallelization strategy using distributed computation has to be designed which should reduce the runtimes linearly, i.e., runtimes keep reducing with increase in number of processors.

The cell placement problem can be stated as follows: Given a collection of cells or modules with ports (inputs, outputs, power and ground pins) on the boundaries, the dimensions of these cells (height, width, etc), and a collection of nets (which are sets of ports that are to be wired together), the process of *placement* consists of finding suitable physical locations for each cell on the entire layout. By suitable it mean those locations that minimize given objective functions, subject to certain constraints imposed by the designer, the implementation process, or layout strategy and the design style. The cells may be standard-cells, macro blocks, etc. In this work, standard cell design is used, where all the circuit cells are constrained to have the same height, while the width of the cell is variable and depends upon its complexity [1].

3.2 Cost Functions Estimations

This section discusses the modelling of the cost functions used for estimating the values of three objectives as well as the constraint.

Power Estimation

In VLSI circuits with well designed logic gates, the dynamic power consumption contributes the 90% to the total power consumption [36, 37]. Hence, most of the reported work is focused on minimizing the dynamic power consumption. Also,

in the case of standard-cell placement, the cells are obtained from the technology library.

In standard CMOS technology, power dissipation is a function of the clocking frequency, supply voltages and the capacitances in the circuit,

$$p_{total} = \sum_{i \in V} p_i (C_i \times V_{dd}^2 \times f_{clk}) \times \beta \tag{3.1}$$

where p_i is the switching probability of gate i over a clock cycle, C_i represents the capacitive load of gate i, f_{clk} is the clock frequency, V_{dd} is the supply voltage, and β is a technology dependent constant. Assuming that the clocking frequency and power voltages are fixed, the total power dissipation of the circuit is a function of the total capacitance and the switching probabilities of the various gates in the logic circuit. The capacitive load of a gate comprises the input gates capacitances of cells and those of interconnects,

$$C_i = \sum_{j \in F_i} C_j^g + C_{ij}^r \tag{3.2}$$

where C_j^g is the capacitance for gate j, C_{ij}^r represents the interconnect capacitance between gates i and j, and $F_i = \{j | (i,j) \in E\}$. Two other terms contribute to power dissipation, the short-circuit current and the leakage current. These are not considered at this level of design.

Delay Estimation

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.

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

$$T_\pi = \sum_{i=1}^{k-1} (CD_{vi} + ID_{vi}) \tag{3.3}$$

where CD_{vi} is the switching delay of the driving cell and ID_{vi} is the interconnection delay that is given by the product of the load factor of the driving cell and the capacitance of the interconnection net, i.e.,

$$ID_{vi} = LF_{vi} \times C_{vi} \tag{3.4}$$

$SLACK_\pi$ of path π is given by

$$SLACK_\pi = LRAT_\pi - T_\pi \tag{3.5}$$

where $LRAT_\pi$ is the latest required arrival time and T_π is the path delay [38, 39]. If T_π is greater than $LRAT_\pi$, then the path π will have a negative $SLACK$ which is an indicator of a **long path** problem. Upper bounds can be applied to nets belonging to the critical path as constraints not to allow them to exceed a certain limit beyond which the $SLACK$ will be negative.

In this work, the approach reported in [38] to predict the K-most critical paths is used. The placement program will seek to satisfy the delay constraints imposed by these paths.

Wirelength Estimation

Different models have been proposed for the estimation of length of a given *net*. Semi-perimeter of bounding box, minimum Steiner tree, minimum spanning tree, etc., are among those models [1, 40]. A Steiner tree approximation described below, which is fast and fairly accurate in estimating the wire length will be adopted in this work [41]. 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. 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. 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.

$$X = \frac{\sum_{i=1}^{n} x_i}{n} \qquad Y = \frac{\sum_{i=1}^{n} y_i}{n} \tag{3.6}$$

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

$$Steiner\ Tree = B + \sum_{j=1}^{k} P_j \tag{3.7}$$

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.

$$Interconnection\ Length = \sum_{l=1}^{m} Steiner\ Tree_l \tag{3.8}$$

where m is the number of nets [42].

Layout Width Estimation

In standard-cell design, cells have fixed height and variable widths. Cells are placed in rows separated by routing channels. The overall area of the layout is represented by the rectangle that bounds all the rows and routing channels. In this work, the

channels heights are initially estimated using an area efficient placement tool and then assumed to be fixed. This leaves only the width of the layout that can effect the layout area. Since the available area for the placement is normally predefined, therefore the width of the layout is used as a constraint. The upper limit on the layout width is defined as,

$$Width_{max} = (1 + \alpha) \times Width_{opt} \tag{3.9}$$

where $Width_{max}$ is the maximum allowable width of the layout, $Width_{opt}$ is the minimum possible layout width obtained by adding the widths of all cells and dividing by number of rows in the layout, and α denotes the maximum allowed fractional increase in the layout width as compared to the optimal width.

3.3 Fuzzy Logic

Fuzzy Logic is a mathematical tool invented to express human reasoning. In classical (crisp) reasoning a proposition is either true or false whereas in fuzzy system a proposition can be true or false with some degree.

A classical (crisp) set is normally defined as collection of elements or objects $x \in X$. Each single x element either belongs to the set X (true statement), or not belong to the set (false statement). Whereas a fuzzy set can be defined as follows.

$$A = \{(x, \mu_A(x)) | x \in X\}$$

$\mu_A(x)$ is called the membership function or grade of membership(or degree of truth) of x in A that maps X to the membership space M. The range of the membership function is a subset of the non-negative real numbers whose supremum is finite [43]. Elements with zero degree of membership are normally not listed.

Like crisp sets, set operations such as union, intersection, and complementation etc., are also defined on fuzzy sets. There are many operators for fuzzy union and fuzzy intersection. For fuzzy union, the operators are known as **s-norm** operators (denoted as $\oplus$). While fuzzy intersection operators are known as **t-norm** (denoted as *).

3.3.1 Fuzzy Reasoning

Fuzzy reasoning is a mathematical discipline to express human reasoning in vigorous mathematical notation. Unlike classical reasoning in which propositions are wither true or false, fuzzy logic establishes approximate truth value of propositions based on linguistic variables and inference rules [44]. A linguistic variable is a variable whose values are words or sentences in natural or artificial language [45]. For example, wirelength is a linguistic variable is its values are linguistic rather than numerical, i.e., very short, short, medium, long, very long and very long etc., rather than $20\mu m$, $25\mu m$, $35\mu m$, $45\mu m$, $55\mu m$ and $80\mu m$. The linguistic variables can be composed to form propositions using connectors like AND, OR and NOT. Formally, a linguistic variable comprises five elements [46].

1. The variable name.

2. The primary term set.

3. The Universe of discourse U.

4. A set of syntactical rules that allows composition of the primary terms and hedges to generate the term set.

5. A set of semantic rules that assigns each element in the term set a linguistic meaning.

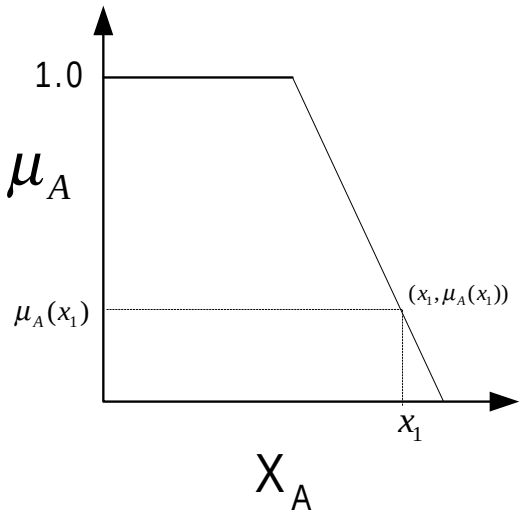

Figure 3.1: Membership function of a fuzzy set A.

For example wirelength can be used as linguistic variable for VLSI placement problem. According to the syntactical rule, the set of linguistic values of wirelength may be defined as very short, short, medium, long, very long and very long wirelength. The universe of discourse for linguistic variable is positive range of wirelength

of a design, eg., $[25\mu m,\ 80\mu m]$. The set of semantic rules define fuzzy sets for each linguistic value. A linguistic value is characterized by its corresponding fuzzy set. The membership in fuzzy set is controlled by membership functions like Figure 3.1. It shows the designer knowledge of problem [44].

3.3.2 Fuzzy Operators

There are two basic types of fuzzy operators. The operators for the intersection, interpreted as the logical "and", and the operators for the union, interpreted as the logical "or" of fuzzy sets. The intersection operators are known as triangular norms (t-norms), and union operator as triangular co-norms (t-co-norms or s-norms) [43]. Some examples of s-norm operators are given below, (were A and B are the fuzzy sets of universe of discourse X).

1. Maximum. $[\mu_{A \bigcup B}(x) = max\{\mu_A(x), \mu_B(x)\}]$.

2. Algebric sum. $[\mu_{A \bigcup B}(x) = \mu_A(x) + \mu_B(x) - \mu_A(x)\mu_B(x)]$.

3. Bounded sum. $[\mu_{A \bigcup B}(x) = min(1, \mu_A(x) + \mu_B(x))]$.

4. Drastic sum. $[\mu_{A \bigcup B}(x) = \mu_A(x)$ if $\mu_B(x) = 0$, $\mu_B(x)$ if $\mu_A(x) = 0$, 1 if $\mu_A(x), \mu_B(x) > 0]$.

An s-norm operator satisfies commutativity, monotonicity, associativity and $\mu_{A \bigcup 0}(x) = \mu_A(x)$ properties. Following are some examples of t-norm operators.

1. Minimum. $[\mu_{A\bigcap B}(x) = min\{\mu_A(x), \mu_B(x)\}]$.

2. Algebraic product. $[\mu_{A\bigcap B}(x) = \mu_A(x)\mu_B(x)]$.

3. Bounded product. $[\mu_{A\bigcap B}(x) = max(0, \mu_A(x) + \mu_B(x) - 1)]$.

4. Drastic product. $[\mu_{A\bigcap B}(x) = \mu_A(x)$ if $\mu_B(x) = 1, \ \mu_B(x)$ if $\mu_A(x) = 1, \ 0$ if $\mu_A(x), \mu_B(x) < 1]$.

Like s-norm, t-norms also satisfy commutativity, monotonicity, associativity and $\mu_{A\bigcap 1}(x) = \mu_A(x)$. Also, the fuzzy complementation operator is defined as follows.

$$\bar{\mu}_B(x) = 1 - \mu_B(x) \tag{3.10}$$

3.3.3 Ordered Weighted Averaging (OWA) Operator

Generally, the formulation of multi criterion decision functions neither desires the pure "anding" of **t-norm** nor the pure "oring" of **s-norm**. The reason for this is the complete lack of compensation of **t-norm** for any partial fulfillment and complete submission of **s-norm** to fulfillment of any criteria. Also the indifference to the individual criteria of each of these two forms of operators led to the development of Ordered Weighted Averaging (OWA) operators [47, 48]. This operator allows easy adjustment of the degree of "anding" and "oring" embedded in the aggregation. According to [47, 48], "orlike" and "andlike" OWA for two fuzzy sets A and B are

implemented as given in Equations 3.11 and 3.12 respectively.

$$\mu_{A\cup B}(x) = \beta \times \max(\mu_A, \mu_B) + (1-\beta) \times \frac{1}{2}(\mu_A + \mu_B) \tag{3.11}$$

$$\mu_{A\cap B}(x) = \beta \times \min(\mu_A, \mu_B) + (1-\beta) \times \frac{1}{2}(\mu_A + \mu_B) \tag{3.12}$$

β is a constant parameter in the range [0,1]. It represents the degree to which OWA operator resembles a pure "or" or pure "and" respectively.

To solve an MOP using fuzzy logic, the problem is first defined in linguistic terms then the membership of different fuzzy sets is combined using t-norm or s-norm operator (depends upon problem). Then the resulting membership is used in minimization or maximization problem.

3.4 Fuzzy Cost Function for VLSI Standard-Cell Placement Problem

In this method, it is assumed that there are Γ Pareto-optimal solutions. Also a p-valued cost vector $C(x) = (C_1(x), C_1(x), ..., C_p(x))$, where $x \in \Gamma$ is given. There is a vector $O = (O_1, O_2, ..., O_p)$ that gives the lower bounds on the cost for each objective such that $O_j \leq C_j(x)\ \forall j$, and $\forall x \in \Gamma$. These lower bounds are normally not reachable in practice. There is another user defined goal vector $G = (g_1, g_2, ..., g_p)$ that represents the relative acceptance limits for each objective. It means that x is

an acceptable solution if $C_j(x) \leq g_j \times O_j, \ \ \forall j$ where $g_j \geq 1.0$. For two dimension problem, Figure 3.2 shows the region of acceptable solution.

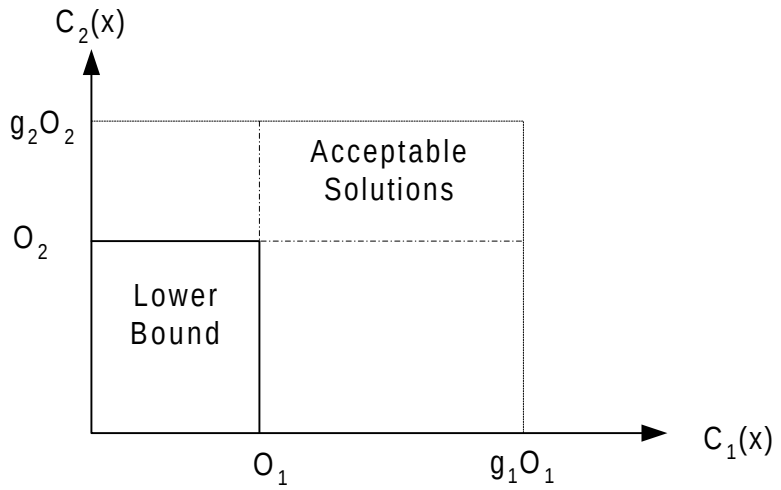

Figure 3.2: Range of acceptable solution set.

In order to solve multiobjective placement problem, three linguistic variables wirelength, power dissipation, and delay are defined. The following fuzzy rule is used to combine the conflicting objectives [42].

Rule R1:

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

The above mentioned linguistic variables are mapped to the membership values

in fuzzy sets *small wirelength, low power dissipation*, and *short delay* respectively. These membership values are computed using the fuzzy membership functions shown in Figure 3.3.

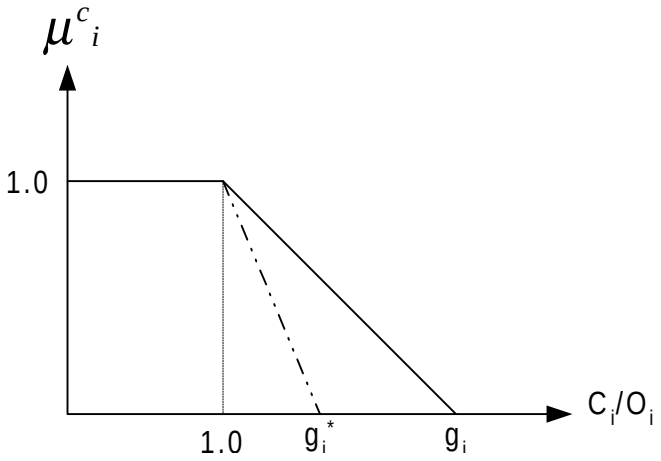

Figure 3.3: Membership functions within acceptable range.

As layout width is a constraint, therefore if a solution violates this constraint, it is not a valid solution and is hence discarded. However, for the objectives, by increasing and decreasing the value of g_i, its preference can be varied in combined membership function. The lower bounds O_j (shown in Figure 3.3) for different objectives are computed as given in Equations 3.13-3.16.

$$O_l = \sum_{i=1}^{n} l_i^* \qquad \forall v_i \in \{v_1, v_2, ..., v_n\} \tag{3.13}$$

$$O_p = \sum_{i=1}^{n} S_i l_i^* \qquad \forall v_i \in \{v_1, v_2, ..., v_n\} \tag{3.14}$$

$$O_d = \sum_{j=1}^{k} CD_j + ID_j^* \qquad \forall v_j \in \{v_1, v_2, ..., v_k\} \; in \; path \; \pi_c \tag{3.15}$$

$$O_{width} = \frac{\sum_{i=1}^{n} Width_i}{\#\; of \; rows \; in \; layout} \tag{3.16}$$

where O_j for $j \in \{l, p, d, width\}$ are the lower bounds on the costs for wirelength, power dissipation, delay and layout width respectively, n is the number of nets in layout, l_i^* is the lower bound on wirelength of net v_i, CD_i is the switching delay of the cell i driving net v_i, ID_i^* is the lower bound on interconnect delay of net v_i calculated with the help of l_i^*, S_i is the switching probability of net v_i, π_c is the most critical path with respect to optimal interconnect delays, k is the number of nets in π_c and $Width_i$ is the width of the individual cell driving net v_i.

Using the Ordered Weighted Averaging (OWA) operator [47, 49, 48], rule **R1** is interpreted as follows:

$$\mu(x) = \beta \times min(\mu_p(x), \mu_d(x), \mu_l(x)) + (1 - \beta) \times \frac{1}{3} \sum_{j=p,d,l} \mu_j(x) \tag{3.17}$$

where $\mu(x)$ is the membership of solution x in fuzzy set of acceptable solutions,

whereas $\mu_j(x)$ for $j = p, d, l$, are the membership values in the fuzzy sets *within acceptable power, within acceptable delay, and within acceptable wirelength* respectively. β is the constant in the range $[0, 1]$.

In this thesis, $\mu(x)$ is used as the aggregating function. The solution that results in maximum value of $\mu(x)$ is reported as the best solution found by the search heuristic i.e. Stochastic Evolution.

3.5 Experimental Setup

In this section, the experimental setup for conducting the research is described. It includes hardware, software, parallel programming paradigm, and other related issues.

Hardware & Software: The hardware part of the experimental setup consists of the following:

- A dedicated cluster of 8 machines, x86 architecture, Pentium-4 of 2 GHz clock speed, 256 MB of memory per processor.
- Ethernet connection (100 Mbit/sec)

The software part consists of the following:

- The operating system used in Redhat Linux 7.2 (kernel 2.4.7-10).

- MPICH [1], version 1.2.5, a portable implementation of MPI standard 1.1 is used.

Programming Paradigm: MPI is a library specification for message-passing, proposed as a standard by a broadly based committee of vendors, implementors, and users. MPI was designed for high performance on both massively parallel machines and on workstation clusters. MPICH 1.2.5 (a specific implementation of MPI 1.1 Standard) is used.

Tools: Various tools for different purposes were used. They include:

Debugging: GDB (Gnu DeBugger) and Totalview from Etnus cop. was used to debug programs.

Performance(System): Built-in UNIX/Linux tools, such as vmstat, top, sar were used.

Performance(Network): Tools available in public domain, such as Netpipe, PMB (Pallas MPI Benchmarks) and NPB (NAS Parallel Benckmarks)were used to obtain performance of the cluster and the network.

Profiling: GProf (Gnu Profiler) was used to profile sequential programs, Intel's VTune Performance Analyzer was used for Remote Data Collection for Sampling and Call Graph generation. VampirTrace was used to generate trace for parallel programs.

[1] http://www-unix.mcs.anl.gov/mpi/mpich/

Visualizations: Accompanied with MPICH, upshot was used for visualizing traces generated by MPI routines. Vampir was used to visualize more details, that were generated with VampirTrace.

Benchmarks for Placement: In this work, ISCAS-89 [2] benchmarks circuits are used. These contain a set of circuits with various sizes, in terms of number of gates and paths.

Table 3.1: Different benchmark circuits.

Circuit	No. of gates	No. of paths
s298	136	150
s386	172	205
s641	433	687
s832	310	240
s953	440	583
s1196	561	600
s1238	540	661
s1488	667	557
s1494	661	558
c3540	1753	668
s9234	5844	512
s15850	10470	512

Cluster Performance:

The cluster is connected with 100 MBit/sec Ethernet. The maximum bandwidth that was achieved using PMB was 91.12 Mbits/sec, with a latency of 68.69 μsec per message.

As for the bandwidth versus message-size, the message size should be in the range

[2] http://www.cbl.ncsu.edu/CBL_Docs/Bench.html

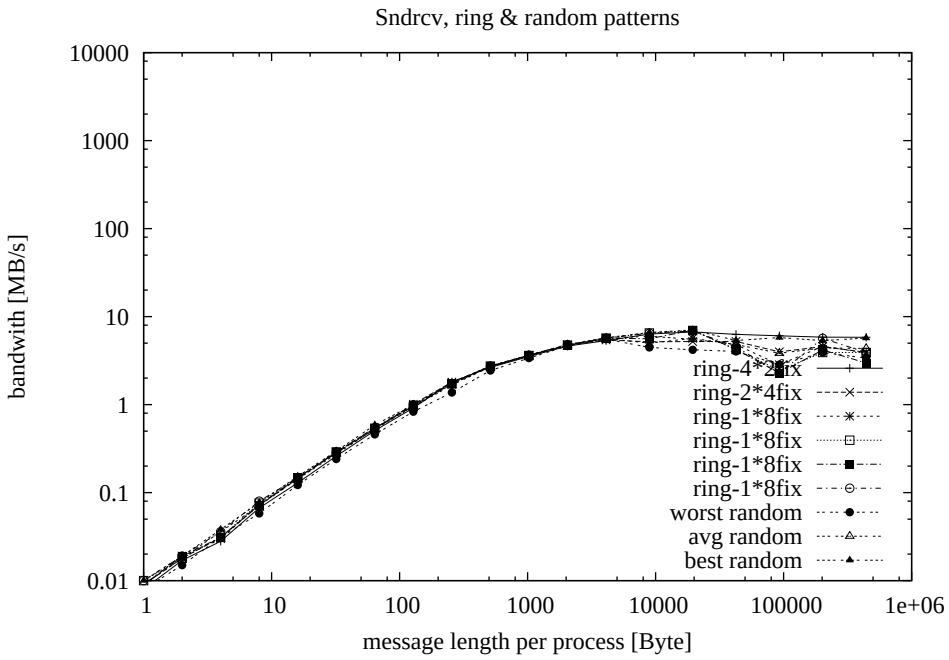

Figure 3.4: Bandwidth versus Message length per process.

of 10KB to utilize the maximum bandwidth. Message size that is considerably less than this will waste the bandwidth, while very large size will take longer to send. This can be seen in the Figure 3.4. In this figure, the bandwidth is measured using the NetPipe benchmark for finding the latency by sending and receiving messages of different sizes from one processor to other in ring and random patterns. It can be seen that irrespective of different test patterns, the behavior remains very similar, i.e., in order to utilize the maximum bandwidth, the message size must be greater than 1KB.

Chapter 4

PARALLELIZATION APPROACH

This chapter presents an overall approach and steps that have been taken in StocE parallelization. Before covering any aspects of parallelization, the implementation details and analysis of sequential StocE must be clear. The details of actual implementation of StocE over VLSI placement problem is thus described below followed by the analysis of this sequential implementation.

4.1 Sequential StocE Implementation

In this particular research, StocE algorithm is employed to solve performance driven low-power VLSI standard cell placement problem. It is a multi-objective constraint

optimization problem where width being a constraint the objective is to reduce wire-length and power. This section describes the the implementation details of StocE algorithm on placement problem.

In the implementation, an initial solution, a range variable p_0 and a termination parameter R, equal to 10, are passed as arguments to the main StocE function. The range variable p_0 is calculated based on the standard deviations in cost of initial solution and newly generated solutions. This range variable thus serves as the basic criterion in accepting or rejecting bad solutions. In the StocE function, the cost of initial solution is calculated and saved as the current cost as well as the best cost. A sorted array of cells is generated based on their connectivity. This array biases the search process by forcing ***PERTURB*** function to move the most connected cells first. The code then enters into the main loop where the current cost is saved as previous cost. ***PERTURB*** function is then called to make a compound move. The compound move is made by selecting two cells, one cell from the sorted array while the other cell in chosen randomly, and swapping them. The perturb process continues until whole the solution is perturbed. On each swap the solution enters a new state and is considered a new solution. The cost of this new solution is calculated and a gain function is calculated by subtracting the cost of previous solution from the cost of new solution. A random number is generated between zero (0) and the range variable p_0. Based on our objective function if the gain achieved is greater than the random number generated the solution is accepted else it is rejected. This

criteria of accepting solutions make sure that if the gain calculated is positive, i.e., the cost of new solution is greater than the cost of previous solution, the solution will always be accepted. Whereas, if the gain calculated is negative, the solution will only be accepted probabilistically, i.e., if the gain is greater than the negative random number generated based on the range variable. After the compound move of ***PERTURB*** function, the cost of new solution is calculated and is saved as current cost. ***UPDATE*** procedure is called afterwards where the costs of current solution and previous solution are compared. The range variable p_0 is incremented if two costs are found equal. Else, if the two costs are not equal the range variable p_0 is initialized with its initial value. On returning to the main loop from ***UPDATE*** procedure current cost is compared with the best cost. If current cost is found greater than the best cost, the current cost is saved as best cost and the algorithm awards itself with more search iterations by decrementing ρ by R. Else, ρ is incremented by one (1) and the algorithm keeps searching for better solutions until ρ becomes equal to R.

4.2 Analysis of Sequential Implementation

Prior to formulating parallelization strategies, it is critical to profile the targeted sequential application to gain insights into performance bottlenecks and time intensive routines. By factoring in these observations with the type of parallel computing en-

vironment - whether distributed or shared-memory, effective and relevant algorithms can be engineered.

The sequential StocE was profiled using the 'gprof' utility [50], and the percentage of time taken by problem-specific cost computations versus other functions is documented in Table 4.1, where the first column specifies the name of ISCAS89 benchmark circuit to which the sequential algorithm is applied, second column shows the problem size in terms of total number of cells present in the circuit, third and fourth columns give the runtime percentages of algorithm spent in calculating the cost functions and the other functions respectively. The profiling results clearly demonstrate that more than 90% of time is spent in the cost functions calculating wirelength, power and delay, thereby highlighting the computational complexity involved.

Table 4.1: Execution-Time Profile for sequential Stochastic Evolution.

Circuit Name	Number of Cells	Wire Length, Power and Delay calculation Time%	Other Functions %
s1494	661	92.66	7.34
s3330	1961	92.86	7.14
s5378	2993	93.43	6.57
s9234	5844	92.87	7.13
s15850	10383	89.64	10.36

Given the above profile, an intuitive approach would be to parallelize or divide the cost functions, thereby achieving workload division. This strategy, which can be categorized under Type I as discussed earlier may prove well in shared memory

architectures where communication overheads are low. However, in distributed parallel environments, where node-to-node communication involves high cost, it would be ineffective.

Given this sequential nature of StocE, a Type-II domain decomposition approach involving partitioning the data set, or a Type-III parallel search strategy may deliver favorable results in speedup. With this profiling analysis, parallelizing StocE in a distributed computing environment should address the following principles:

1. Avoid any low-level or fine grained parallelization
2. Workload division or data-set partitioning should not alter the algorithm's sequential flow
3. Any errors introduced to data partitioning or parallel search threads should be continuously remedied and
4. Communication overhead should be minimal

4.3 Parallel Algorithm Design Steps

In the previous section, an analysis of the sequential code was presented. No matter how finely tuned the sequential algorithm is, the task still remain processor-intensive.

As mentioned earlier, parallel algorithm executed in a parallel environment provides an opportunity to increase the performance of sequential code. The challenge

however, is to design the parallel algorithm in such a way, that it makes the best use of the parallel environment.

Designing parallel algorithms is not straight-forward. It is not just running the sequential code on more processors. It requires an understanding of parallel environment, knowledge of the algorithm and the modules that can be executed in parallel. In addition to this, there are several issues, such as communication strategies, load-balancing, etc. Therefore, a design methodology has to be followed to develop parallel algorithms, as it maximizes the consideration of the available options, and reduces the cost of backtracking from bad choices, made in earlier stages. It also helps in identifying design flaws that compromise the desirable attributes of parallel algorithm [51].

4.4 Parallel Models for Stochastic Evolution

4.4.1 Asynchronous Multiple Markov Chain (AMMC)

This parallelization approach was reported for Simulated Annealing (SA) and showed good results in terms of runtimes trend [52]. A similar approach to StocE is implemented, thereby utilizing the advantages that AMMC offers in terms of relaxing synchronization requirements among individual processors. Since Stochastic Evolution is strictly sequential in nature, the asynchronous feature relaxes demanding communication requirements, and can be intuitively considered to perform well.

Moreover, StocE follows a search path based on randomization seeds and determined by the acceptance/rejection of moves, and hence each of these paths can be viewed as as separate Markov chain. With such a parallel AMMC approach, there is no work division among processors as each runs the whole StocE algorithm over the solution it posses. However, since each processor is exploring the solution space in different regions (by using different random seeds), a larger number of them increases the chances of reaching good solutions. Moreover, the search process is biased by propagating the best solution among all processors. Thus, whenever one of them reaches a solution compared to what all other processors have reached, the same is communicated to all, thereby intensifying exploration around that area in the search space.

This AMMC approach used a master-slave architecture, the working of which are shown in Figure 4.1 and Figure 4.2.

Here, a slave processor always sends the solution cost to the master node, whenever it reached a solution better than the one it already posseses. The master then compares the received cost with what it already has. If it is better, the slave is instructed to send the entire solution; otherwise, the master sends its solution to the slave. There were two options for the master processor - either it is limited only to servicing slave requests for cost comparison, or can also be a processing node itself, using StocE to follow its own search path. We follow the first approach to avoid any queuing delays for slave processors.

```
Algorithm_Parallel_StocE_AMMC_Master_Process
Notation
 (* CurS is the current solution. *)
 (* Cost(S) returns the cost of solution. *)
 (* BestS is the best solution. *)
Begin
 Read_User_Input_Parameters( )
 Read_Input_Files
 Construct_Initial_Placement
 CurS = S0; // only master has the initial Solution
 BestS = CurS;
 CurCost = Cost(CurS);
 BestCost = Cost(BestS);
    Broadcast(CurS);
 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. (*Master_Process*)
```

Figure 4.1: Master Process for Parallel AMMC StocE Algorithm.

4.4.2 Fixed Pattern Row-Division

The real essence of this strategy lies in work division among the candidate processors. In contrast to Asynchronous Multiple Markov Chain strategy, where the work load remains the same on each processor this strategy proves more promising since it ensures the reduction in effective load on each working node. Moreover, the work allocation is based on a fair distribution of rows among processors. Each is

```
Algorithm_Parallel_StocE_AMMC_Slave_Process
Notation
 (* CurS is the current solution. *)
 (* Cost(S) returns the cost of solution. *)
 (* BestS is the best solution. *)
Begin
 Read_User_Input_Parameters( )
 Read_Input_Files
     Receive_Initial_Sol(CurS);
 CurS = S0;
 BestS = CurS;
 CurCost = Cost(CurS);
 BestCost = Cost(BestS);
 Repeat
       S = PERTURB(S,p); /* perform a search in the neighborhood of s */
       CurCost= Cost(S);
       UPDATE(p, PrevCost, CurCost); /* update p if needed */
       If (CurCost< BestCost) Then
          BestS=S;
          BestCost= CurCost;
          ρ = ρ − R; /* Reward the search with R more generations */
       Else
          ρ = ρ + 1;
       EndIf
       Send_to_Master(BestCost);
       Receive_frm_Master(verdict);
       If (verdict == 1)
          Send_to_Master (BestS);
       Else
          Receive_frm_Master(BestS);
       EndIf
    Until  ρ > R
    Return (BestS);
 End. (*Slave_Process*)
```

Figure 4.2: Slave Process for Parallel AMMC StocE Algorithm.

assigned with two sets of rows and is responsible for swapping cells among them, while alternating between these sets every iteration.

The pattern for each processor alternates in each iteration to ensure the movement of a cell to any place in the solution in at most two steps. Moreover, this scheme does not mandate much communication overhead, since the processors do not need to exchange information or synchronize during iterations. Using row division strategy, it is expected that quality of results will increase for large problem instances with increasing number of rows.

To understand further the row division strategy lets consider a loosely coupled distributed environment with three participating processors.

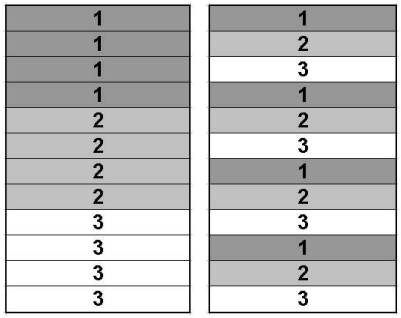

Figure 4.3: Row-Division.

Figure 4.3 shows the allocation of rows to the three processors. The left pattern shows the distribution in odd-numbered iterations where the right one shows the distribution in even-numbered iterations. The pattern for each processor alternates in each iteration to ensure the movement of a cell to any place in the solution in at most two (2) steps, as long as the number of rows in the solution is sufficiently

large. Moreover, this scheme do not require much communication time since the processors do not need to communicate during iterations. Thus, the total amount of real time needed is the sum of real times required by the slowest processor during each iteration. As mentioned earlier, since the quality of final result mainly depends on the parallel strategy implemented, using row division strategy it is expected that quality of results will increase for solutions with large number of rows.

The implemented strategy was termed as Pure Fixed Pattern Row-Division. A hybrid mechanism combining Row-Division and AMMC was also implemented but did not give good trends in terms of runtime savings. The advantage of the former approach is that all the processors work on a single solution where each focusses on a different set of rows in a single iteration. We used a master-slave parallel architecture, which was tweaked, so that the master does not assign and send rows to all the slaves every iteration; instead all processors including the master calculate determine their allocated rows by themselves. This significantly saves on communication overhead. The algorithm is shown in Figure 4.4 for the master node, while Figure 4.5 shows the parallel algorithm followed by slaves.

As is clear from from the algorithms, following the pre-computation tasks, such as reading circuit data, generating an initial solution and its evaluation, ***PERTURB*** function is called. Each processor has the same initial solution and starts working on its own set of rows, such that all rows being worked on in a single iteration are non-overlapping. Since all the slaves were working on a single solution, the resulting

Algorithm_Parallel_StocE_Master_Process
Notation
(* *CurS* is the current solution. *)
(* Φ_s is the partition selected to work upon. *)
Begin
Read_User_Input_Parameters( )
Read_Input_Files
Construct_Initial_Placement
Calculate_Rows
Repeat
ParFor
Slave_ Process(CurS)
(* Broadcast Cur Placement. *)
EndParFor
$S = PERTURB(S, p)$; /* perform a search in the restricted neighborhood of s */
(* For each slave process. *)
ParFor
Receive_Partial_Solutions
EndParFor
Make_Complete_Solution
$CurCost = Cost(S)$;
$UPDATE(p, PrevCost, CurCost)$;
/* update p if needed */
If ($CurCost < BestCost$) **Then**
$BestS = S$;
$BestCost = CurCost$;
$\rho = \rho - R$;
/* Reward the search with R more generations */
Else
$\rho = \rho + 1$;
EndIf
Until $\rho > R$
Return (Best_Solution)
End. (*Master_Process*)

Figure 4.4: Master Process for Fixed Row-Division Parallel StocE Algorithm.

Algorithm_Parallel_StocE_Slave_Process($CurS, \Phi_s$)
Notation
(* $CurS$ is the current solution. *)
(* $\Phi_s$is the partition calculated by the slave s to work upon. *)
(* m_i is module i in Φ_s. *)
Begin
Read_User_Input_Parameters( )
Read_Input_Files
Construct_Initial_Placement
Calculate_Rows
Repeat
Receive Placement
$S = PERTURB(S, p)$;
/* perform a search in the restricted neighborhood of s */
Send_Partial_Solution
Until Fitness_Value_not_achieved
End. (*Slave_Pocess*)

Figure 4.5: Slave process for Fixed Row-Division Parallel StocE Algorithm.

placements achieved are termed as partial solutions. After completing ***PERTURB***, all the slaves send their partial solutions to the master where it also includes its own partial partial placement and aggregates all into a single solution again. The Master evaluates this new solution and based on the cost of new solution, it either increments the rho parameter by one or decrements it by R. This new solution is then again broadcast to all the slaves where the slaves again call ***PERTURB*** function but now work with a different set of rows. This process continues till the target fitness value is achieved. Master keeps track of the time when each higher quality solution is received.

4.4.3 Random Row-Division

This parallel strategy is the variation of Fixed Pattern Row Division described above. In contrast to Fixed Pattern Row Division where all the processors have two fixed sets of non-overlapping rows for alternative iterations, here the master processor generates the non-overlapping rows randomly and broadcast it to the slaves in each iteration. The apparent advantage of this scheme over the previous is the randomness in rows distribution which makes sure that unlike the previous strategy none of the rows remains with any specific processor throughout the search process. The algorithm is shown in Figure 4.6 for the master node, while Figure 4.7 shows the parallel algorithm followed by slaves.

As can be seen from the figure above, in each iteration the slaves receive from master the randomly generated rows and then similar to the Fixed Row Division strategy ***PERTURB*** is called to search for better solutions.

Algorithm_Parallel_StocE_Master_Process
Notation
(* *CurS* is the current solution. *)
(* S is the solution to perturb. *)
(* p is the control variable. *)
Begin
Read_User_Input_Parameters( )
Read_Input_Files
Construct_Initial_Placement
Repeat
ParFor
Slave_ Process(CurS)
(* Broadcast Cur Placement. *)
(* Broadcast Randomly Generated Rows. *)
EndParFor
$S = PERTURB(S, p)$; /* perform a search in the restricted neighborhood of s */
(* For each slave process. *)
ParFor
Receive_Partial_Solutions
EndParFor
Make_Complete_Solution
$CurCost = Cost(S)$;
$UPDATE(p, PrevCost, CurCost)$;
/* update p if needed */
If ($CurCost < BestCost$) **Then**
$BestS = S$;
$BestCost = CurCost$;
$\rho = \rho - R$;
/* Reward the search with R more generations */
Else
$\rho = \rho + 1$;
EndIf
Until $\rho > R$
Return (Best_Solution)
End. (*Master_Process*)

Figure 4.6: Master Process for Random Row-Division Parallel StocE Algorithm.

Algorithm_Parallel_StocE_Slave_Process($CurS, \Phi_s$)
Notation
(* S is the current solution. *)
(* p is the control parameter. *)
Begin
Read_User_Input_Parameters( )
Read_Input_Files
Construct_Initial_Placement
Repeat
Receive Placement
Receive Randomly Generated Rows
$S = PERTURB(S, p)$;
/* perform a search in the restricted neighborhood of s */
Send_Partial_Solution
Until Master_Terminates
End. (*Slave_Pocess*)

Figure 4.7: Slave process for Random Row-Division Parallel StocE Algorithm.

Chapter 5

EXPERIMENTS AND RESULTS

This chapter presents the experimental results of implementing parallel strategies to Stochastic Evolution as described in chapter 4. Results obtained from sequential version are compared with the parallel implementation results. The results are organized in tables to give a clear picture of speedup trends where speedup graphs are also shown.

5.1 Performance Evaluation

A basic method to evaluate the performance of a parallel algorithm is to reduce the runtime, when the application is executed on more than one processor. Speed-up is a ratio of runtime of a single processor over multiple processors. In an ideal case, the speed-up of an application is the number of processors (P) available. However,

in most cases, it is not possible to obtain P speed-up, either due to communication overhead or the presence of a non-parallelizable part in the application. Amdahl's law can be used to formulate this as following:

$$S_P = \frac{T_1}{T_P} \tag{5.1}$$

$$T_P = \frac{T_1 + Q}{P} \tag{5.2}$$

where,

P = Number of Processors

S_P = Speedup on P Processors

T_1 = Time on 1 Processor

T_P = Time on P Processor

Q = Communication Overhead and/or non-parallelization part

Using Equations 5.1 and 5.2, the speed obtained in terms of time on one processor is,

$$S_P = \frac{PT_1}{T_1 + Q} \tag{5.3}$$

In an ideal case, where $Q = 0$, gives,

$$S_P = \frac{PT_1}{T_1} = P \tag{5.4}$$

5.2 Algorithm - A: Parallel Asynchronous Multiple Markov Chain

The theory and implementation of Parallel Asynchronous Multiple Markov Chain is discussed in detail in Chapter 4. This section presents the results achieved when our parallel AMMC StocE strategy was tested on different ranges of ISCAS89 benchmark circuits.

Our results are obtained after profiling the sequential code and optimizing it for minimum runtime on a single processor of one of the machines of the cluster. Here 'Circuit Name' describes the benchmark circuit name, 'Number of Cells' shows the total number of gates present in the given benchmark circuit, and 'mu(s)' represents the targeted fitness value or fuzzy cost. 'Time for Sequential StocE' shows the time required by a single processor to reach the particular fitness member indicated by 'mu(s)' whereas 'p' under 'Time for Parallel StocE' shows the number of processors used in computation where time is in seconds.

The run-times achieved with parallel AMMC strategy shown in Table 5.1, shows how poorly the algorithm performs after the initial workload distribution with two processors. The number of processors is limited to six, as the poor performance

Table 5.1: Results for Parallel AMMC StocE.

Circuit Name	Number of Cells	$\mu(s)$ StocE	Time for Sequential StocE	Time for Parallel StocE				
				p=2	p=3	p=4	p=5	p=6
s1494	661	0.6	94	32.78	32.72	32.73	32.79	34.2
s3330	1961	0.6	186	96.92	95.87	89.07	92.66	95.17
s5378	2993	0.6	479.93	268.98	270.78	265.89	270.59	268.63
s9234	5844	0.6	1143	799.36	802.63	800.83	799.42	799.15
s15850	10383	0.6	2103	1908	1903	1908	1905	1908

trend doesn't warrant extra resources. The negligible speedup characteristics, if any, are shown in Figure 5.1.

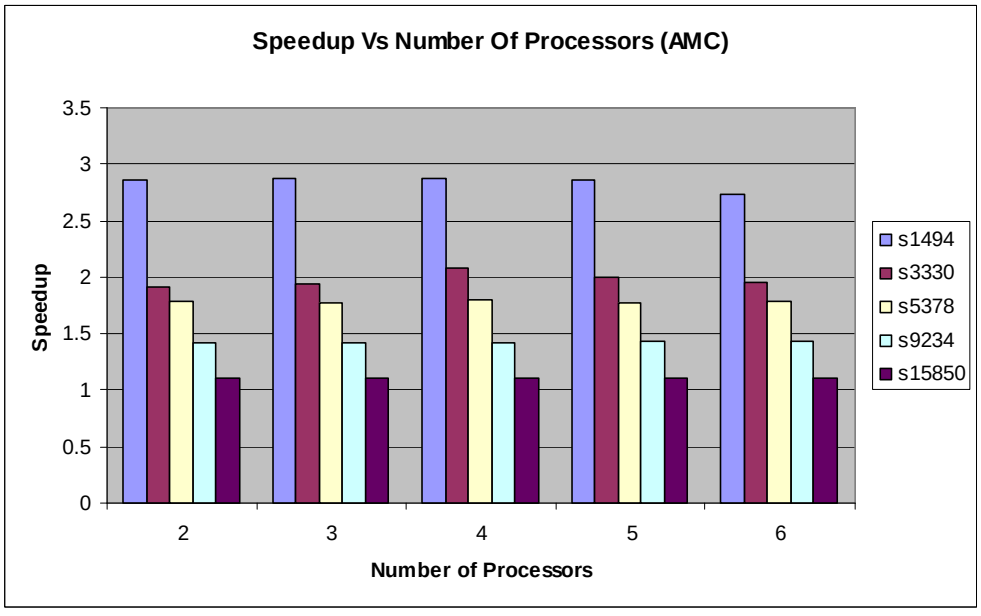

Figure 5.1: Speedup characteristics for the Asynchronous Multiple Markov Chain (AMMC) strategy.

The speedup graph clearly shows that parallel AMMC did not give good results when applied on StocE. Whereas in literature the same parallel approach gave better results when implemented on Simulated Annealing. The difference in behavior of strategy behavior when applied on two different heuristics, though not much different in nature, required a comprehensive research into the algorithm's flow. The findings are discussed in Section 5.5.

5.3 Algorithm - B: Fixed Pattern Row Division

Similar to the previous section this section present the results achieved when fixed row division was implemented to parallelize StocE. This strategy gave excellent results specially when applied on circuits with large number of rows.

The results achieved with the Fixed Pattern Row Division strategy are documented in Table 5.2. The number of processors that can be allocated is limited by the number of rows, and hence the size of the circuit. As such, this parallel strategy applied to smaller circuits was tested with a varying subset of processors. The runtime gains for this particular strategy are shown in terms of speedup in Figure 5.2. The details of the table elements have already been explained in Section 5.2.

Table 5.2: Results for Fixed Pattern Row-Division Parallel StocE.

Circuit Name	$\mu(s)$	Time Serial	Runtime for Parallel StocE				
			p=2	p=3	p=4	p=5	p=6
s1494	0.6	60	49	55	112	-	-
s3330	0.7	1087	355	214	190	186	170
s5378	0.65	1047	495	365	311	305	293
s9234	0.65	2140	1261	917	704	616	615
s15850	0.65	3538	2876	1841	1543	1423	1167
s35932	0.65	12067	6709	4600	3318	3144	2314
s38417	0.65	14437	8559	5698	4649	3982	3340

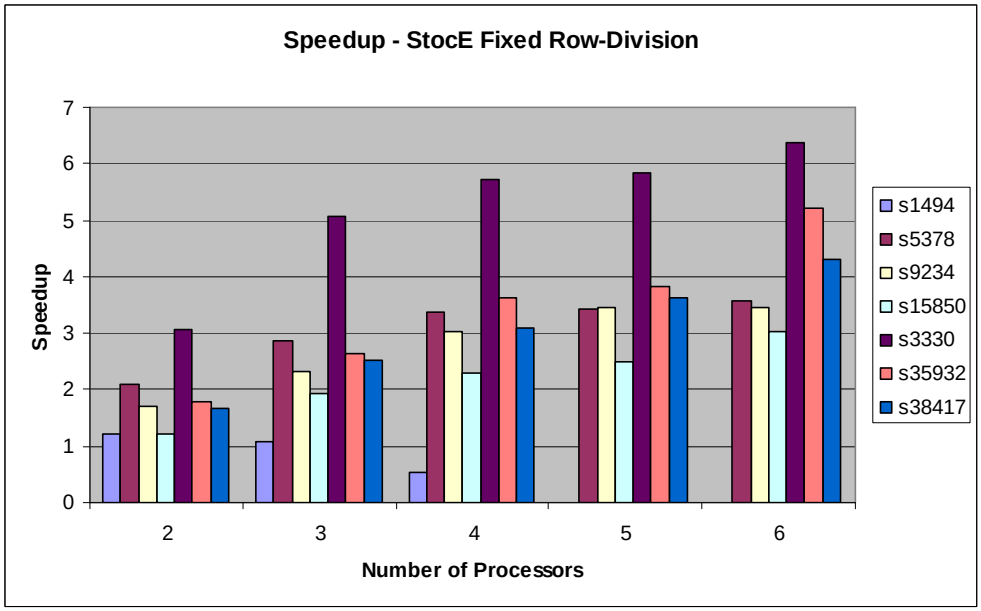

Figure 5.2: Speedup trend for Parallel StocE with Fixed Row-Division.

As is clear from the tables and graphs that Fixed pattern row division gave excellent results in terms of speedup performance. The analysis of row division strategy gave the insight of algorithms flow and justifies the algorithm's good behavior when implemented on StocE. The discussion is presented in the Section 5.5.

5.4 Modified Algorithm - B: Random row division

This section presents the results of random row division. The results achieved with the Fixed Pattern Row Division strategy are documented in Table 5.3. The number of processors that can be allocated is again limited by the number of rows, and hence the size of the circuit. As such, this parallel strategy applied to smaller circuits was tested with a varying subset of processors. The runtime gains for this particular strategy are shown in terms of speedup in Figure 5.3. The details of the table elements have already been explained in Section 5.2.

Table 5.3: Results for Parallel Random Row-Division StocE.

Circuit Name	Number of Cells	$\mu(s)$ StocE	Time for Sequential StocE	Time for Parallel StocE					
				p=2	p=3	p=4	p=5	p=6	p=7
s1494	661	0.6	60	21.4	18.2	11.75	11.62	15.38	-
s3330	1961	0.7	1087	166	125	89	77	63	66
s5378	2993	0.65	1047	193	135	97.86	94	86	69
s9234	5844	0.65	2140	660	447	334	265	250	214
s15850	10383	0.65	3538	1503	1022	843	760	635	500

As shown in results fixed pattern row division thus provided best results in terms

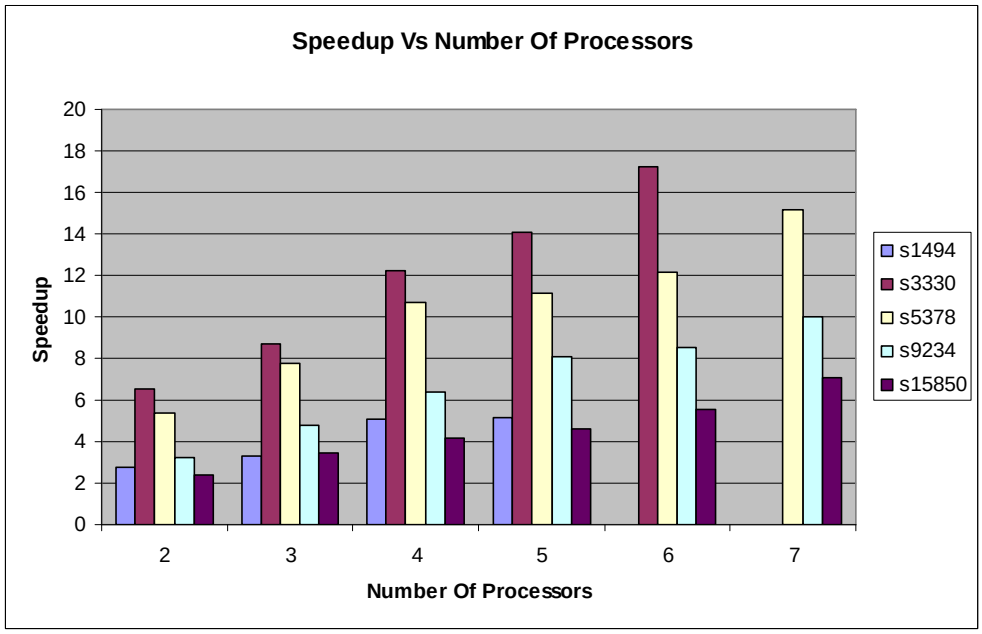

Figure 5.3: Speedup trend for Parallel StocE with Random Row-Division.

of speedup performance. The discussion is presented in the Section 5.5.

5.5 Discussion and Analysis

This section, discusses and analyzes the results seen for the different parallel strategies described in Chapter 4 against two dominant factors - 1) the solution perturbation operation, and 2) characteristics of the parallel environment.

5.5.1 Solution Perturbation and Algorithmic Intelligence

The intelligence and effectiveness of any optimization algorithm is determined by how it navigates the search space with a strong bias towards higher quality solutions. The algorithmic intelligence of StocE lies in its perturbation function, in its method

of calculating and updating its control parameter p, and in selecting an appropriate termination criteria represented by R. Though the AMMC approach worked well with Annealing, its results with StocE are very limited, showing runtime gains for up to two processors, beyond which, speedup is non-existent.

Annealing can be modeled as a time-inhomogenous algorithm, given the way it moves from solution state to the other, the acceptance rate decided by a predictable varying value of temperature. An important property is that the next state does not depend on the solution states that have preceded the present solution. StocE, on the other hand is a non-homogenous Markov chain, as the acceptance probability for solutions is dependent on the parameter "p", which varies unpredictably. In such a case, the probability of a new solution determining the next state of the search process depends on the history of earlier moves. Also, unlike annealing, each iteration of StocE involves a complex move wherein all cells and their locations in the circuit layout are processed. Thus, the reason behind this limited performance is that each StocE thread performs a compound move that optimizes the solution to a large extent without cooperation from other processors. Furthermore, the self-rewarding criteria of StocE, triggered on finding good solutions, relaxes the termination criteria. This, in effect, gives each processor enough time to keep improving the solution when in local minima and thus the cooperation from other processors gives no noticeable benefit. As a result, StocE fails to show any benefit beyond two processors.

In the fixed pattern row-division parallelization approach, the workload is ef-

ficiently distributed by dividing the solution among multiple processors, without disturbing the intelligence of the perturbation mechanism. Here, the Master processor remains in charge of updating and controlling parameters. This direct workload distribution was the primary reason behind the favorable speedup trends seen with this strategy.

The random row-division further improved the speedup when compared to Fixed Pattern row-division. The analysis of fixed pattern row division shows that although processors have two distinct sets of rows in alternative iterations but one of these rows always belongs to the one specific processor. This decreases the probability for some cells of those particular row to be moved frequently since this row will never belong to any other processor.

Also, speedup is increased in case of random-row division since the probability of a cell movement to any location in the solution becomes non-zero in the very first iteration unlike the case of fixed row-division where two iterations were needed to achieve this non-zero probability. Thus, reducing the overall runtime.

Based on this assumption when random row division strategy was implemented it confirmed the drawn conclusions by improving the speedups as clear from the Figure 5.4.

5.5.2 Parallelization Environment

In a distributed, clustered workstation environment, where all communication is carried out via message passing, the communication time among processors is a critical concern. This concern thus directs us to the selection of parallel strategies that ensure minimum communication time. This is the reason for not implementing low-level parallelization strategies as they require intensive communication among processors in order to keep errors minimized. This also explains the reason behind the selection of the AMMC and row-division strategies.

In AMMC parallel approach, each processor is working on a complete solution independently and communicates only when it gets a good solution. Additionally, as there is a master node allocated for solution synchronization among all processors, the communication time is significantly reduced. In fixed pattern row-division parallel approach, again the slaves only communicate with master when they complete the computation intensive perturbation part. In contrast to AMMC, here slaves do not have to stay idle as the master waits for the slowest processor to respond, as the workload distribution is straightforwardly fair and equal. Thus all nodes finish their computation at the same time.

Chapter 6

COMPARISON

The primary reason for using parallel algorithms is to speedup the sequential computations. It is therefore quite natural to compare the running time of a parallel algorithm designed for a certain problem to that of the best available sequential algorithm for the same problem. This is usually done by computing a ratio known as the *speedup* [53]. Where speedup is defined as follows: Let t_1 denote the worst case running time of the fastest known sequential algorithm for the problem, and let t_p denote the the worst case running time of the parallel algorithm using p processors. Then the speedup provided by the parallel algorithm is

$$S(1,p) = \frac{t_1}{t_p} \tag{6.1}$$

A good parallel algorithm is one for which this ratio is large. This chapter com-

pares the Stochastic Evolution (StocE) algorithm with Simulated Annealing (SA). The focus is on performance of these algorithms for the fitness values best achieved by StocE, especially between their parallel implementations. The comparison is among the best strategies that were implemented for individual heuristics. Also it would be pertinent to point out here, that the performance of these heuristics is not an across-the-board endorsement of an algorithm over another, but rather the results are respective to the parallel environment and problem instance.

6.1 Simulated Annealing & Stochastic Evolution

Stochastic Evolution (StocE) is admittedly inspired by Simulated Annealing, its inherent simplicity and elegance. StocE was indeed designed to minimize the time likely wasted during the initial runs in the high temperature region, where annealing approximates a random walk strategy. Similar to SA, StocE employs control parameters to decide uphill moves. Parallel Annealing implementation was also the part of the project and similar strategies were also implemented to for SA parallelization. This gave the chance to compare the two algorithms with similar parallel implementations. In the case of Simulated Annealing, Asynchronous Multiple Markov Chain proved to be better when compared to Row-Division model specially in case of bigger circuits. Whereas, in case of Stochastic Evolution, the Markov Chain models failed to produce speedup, and rather the Row-Division method which distributed

the circuit placement among individual processors worked better.

Here, comparison between Asynchronous Multiple Markov Chain model and Fixed Row-Division applied to Adaptive Simulated Annealing against the same two models, Asynchronous Multiple Markov Chain and Fixed Row-Division, with the addition of Random Row-Division applied to Stochastic Evolution is presented. The Row-Division method in StocE assigns the same solution to all processors and instructs them to work on distinct rows of the placement simultaneously. This model, understandably works best on large circuits with extensive number of rows and cells. Thus, as the circuit size increases the difference among the performance of two algorithms become more clear.

Table 6.1 gives the run-times achieved for the AMMC model applied to adaptive simulated annealing. We focus on medium to large circuits, thereby navigating more complex and vast search landscapes. Both Annealing and StocE ran till they reached the target fitness values noted in the second column. Runtime trends have already been tabulated in Chapter 5 for StocE AMMC, StocE Fixed Row-Division and StocE Random Row-Division. Tables below shows the runtime trends for Simulated Annealing AMMC and Simulated Annealing Fixed Row-Division.

It is important to note that the AMMC annealing approach used a single processor dedicated to controlling communications between the working nodes. As such, a minimum of three processors is required to see any advantages of the parallel strategy. However, in case of Row-Division used for SA, the manager node was

Table 6.1: Runtime trends seen for Asynchronous MMC for Simulated Annealing.

Circuit Name	$\mu(s)$	Time Serial	Runtime for Parallel SA				
			p=3	p=4	p=5	p=6	p=7
s1494	0.6	50	33	25	27	16	15
s3330	0.7	695	300	200	207	191	126
s5378	0.65	2489	1798	1107	1008	1112	1033
s9234	0.65	14851	2821	2026	1575	1186	886
s15850	0.65	18940	14890	8432	1975	1682	1590

Table 6.2: Runtime trends for Parallel Simulated Annealing - Fixed Row-Division.

Circuit Name	$\mu(s)$	Time Serial	Runtimes for Parallel SA					
			p=2	p=3	p=4	p=5	p=6	p=7
s1494	0.6	50	28	19	16	16	*	*
s5378	0.65	2489	919	690	507	444	389	378
s9234	0.65	14851	4898	3741	2894	2628	1858	1818
s15850	0.65	18940	10036	7892	5092	4525	3409	3473

also allocated a section of the solution and was involved in the main computation and evaluation. Therefore, Table 6.2 lists parallel performance starting with two processors.

Figures 6.1, 6.2, 6.3, 6.4 and 6.5 shows the comparison amongst the results achieved against a particular benchmark circuit. For each benchmark circuit, all the above mentioned parallel models of Stochastic Evolution and Simulated Annealing were employed. Each figure shows a time reduction curve on increasing number of processors as well as the corresponding speedup achieved. The speedup shown is the ratio of best sequential time among compared algorithms, to the parallel time of the algorithm being compared. The best sequential time is the time taken by sequential StocE. In all the figures, the left graph shows the run-time reduction per increase in processor while the right figure shows the gain in speedup per increase in processor. These curves show the average trends in time reduction and speedup. In both the graphs, X-axis shows the number of processors employed. Whereas, Y-axis in time reduction graph represents the time acquired to achieve a targeted quality while in speedup curves it is the speedup achieved on increasing number of processors.

In all the comparison figures except Figure 6.5, only positive speedups have been shown. Thus, the parallelization points where increasing an extra processor reduced the speedups are not depicted.

Similar to the results in Figure 6.1, Figure 6.2 shows the same trend in results for benchmark circuit s9234, where again the StocE Random Row-Division out-

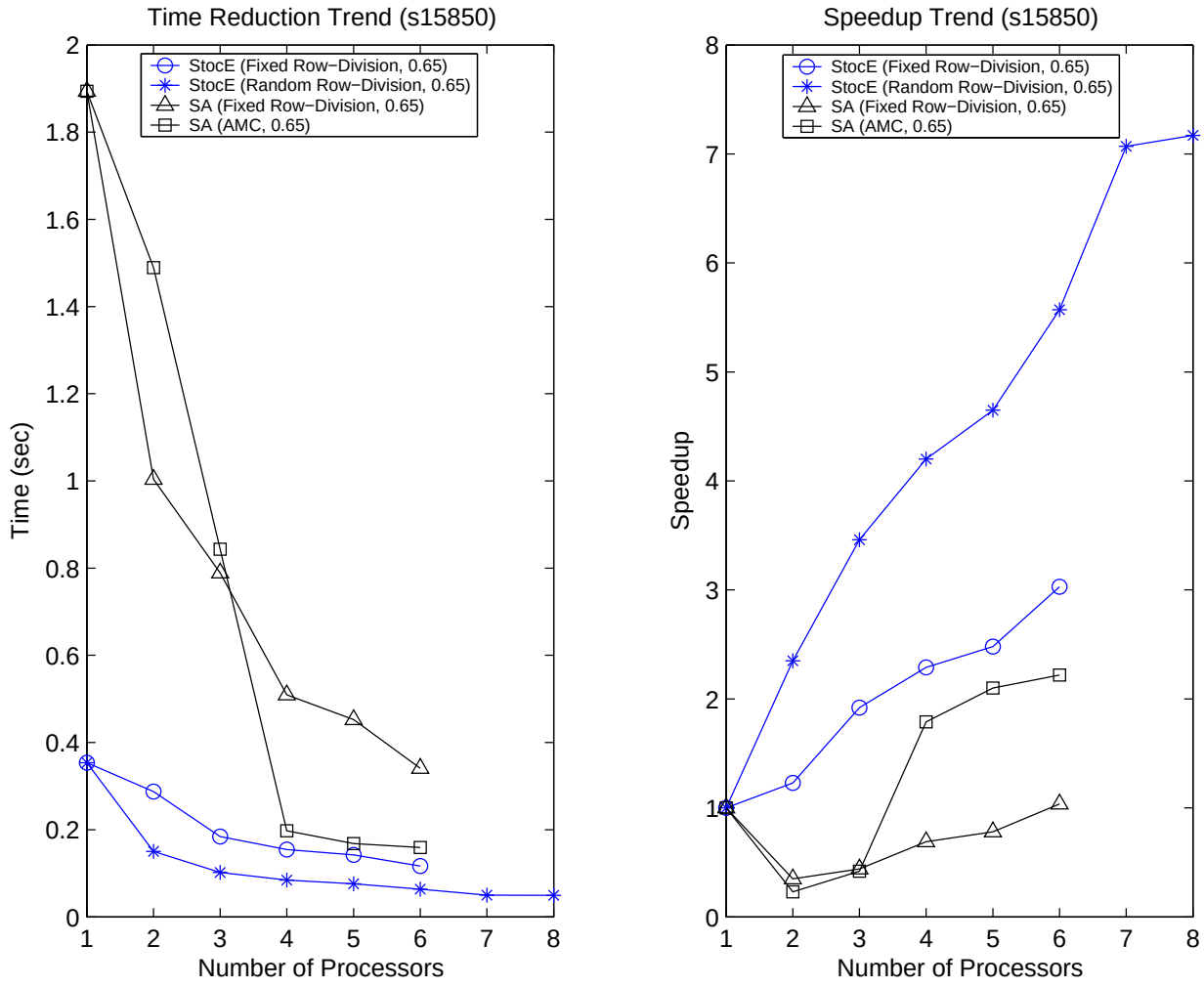

Figure 6.1: StocE Vs SA,s15850(0.65), StocE random row-division outperforms all by achieving the speedup of above 7 with 8 processors. Where StocE fixed row-division and SA AMMC produced the speedups of 3 and 2 respectively with 6 processors. SA fixed row-division was not able to produce any speedup on this circuit.

performs the other parallel algorithms by achieving the targeted fitness quality in 214 seconds with 7 processors. The details of other algorithms' runtimes can be clearly seen in the figure. Similar trends are seen for circuit s5378 in Figure 6.3. For circuit s3330, Figure 6.4, StocE Random Row-Division once again performed the best among all the compared techniques, where StocE Fixed Row-Division and SA AMMC performed almost equally. Results for SA Fixed Row-Division were not present for this benchmark circuit. For the smallest benchmark circuit s1494 with least number of cells, results shown in Figure 6.5, StocE Fixed Row-Division was unable to perform well where Simulated Annealing AMMC and Simulated Annealing Fixed Row-Division performed better. But again in this case, StocE Random Row-Division still out performs the other techniques by achieving the target fitness value in 11.62 seconds with 5 processors.

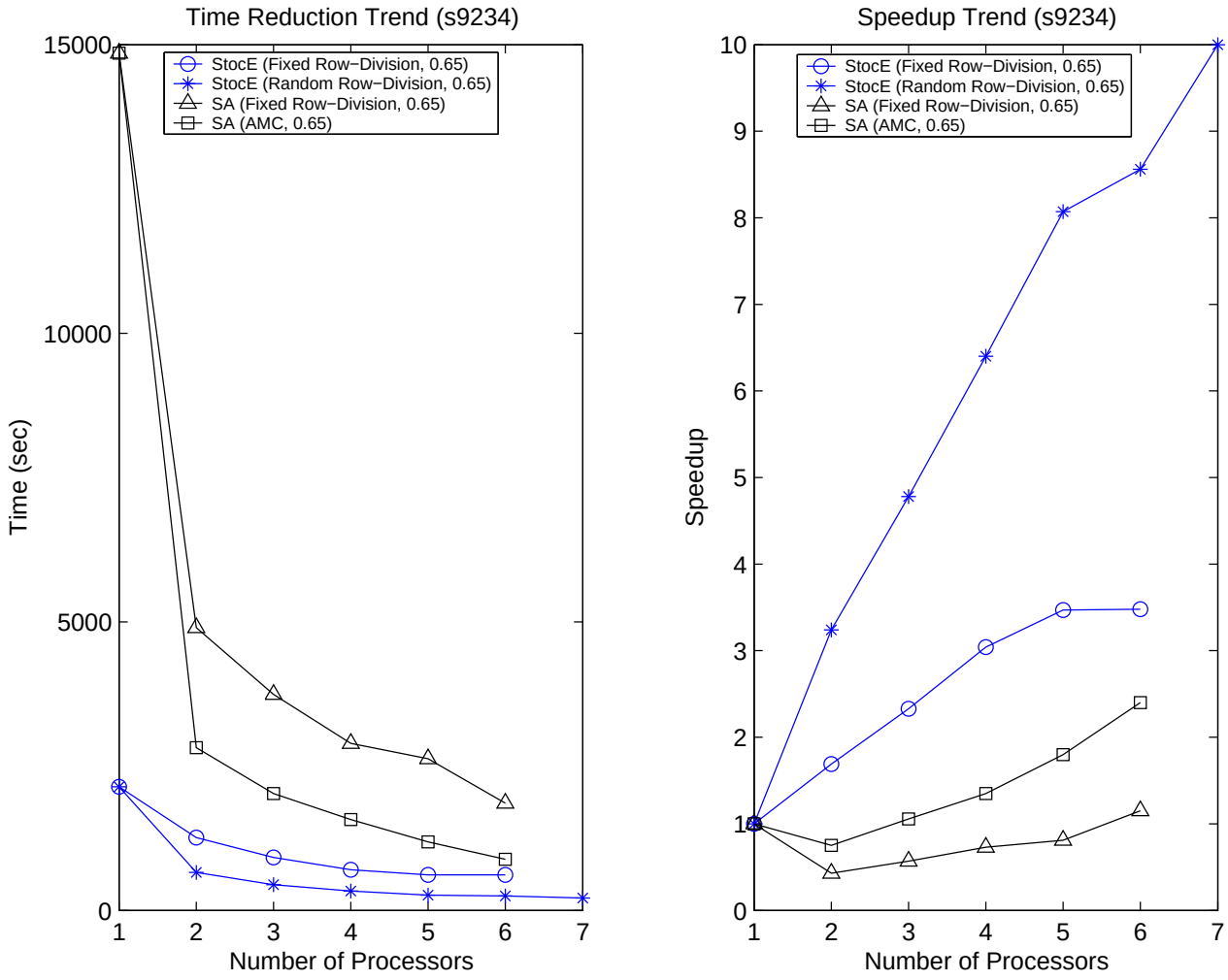

Figure 6.2: StocE Vs SA,s9234(0.65), StocE random row-division outperforms all by achieving the speedup around 10 with 7 processors. Where StocE fixed row-division and SA AMMC produced the speedups up to 3.5 and 2.5 respectively with 6 processors. SA fixed row-division was not able to produce any speedup on this circuit.

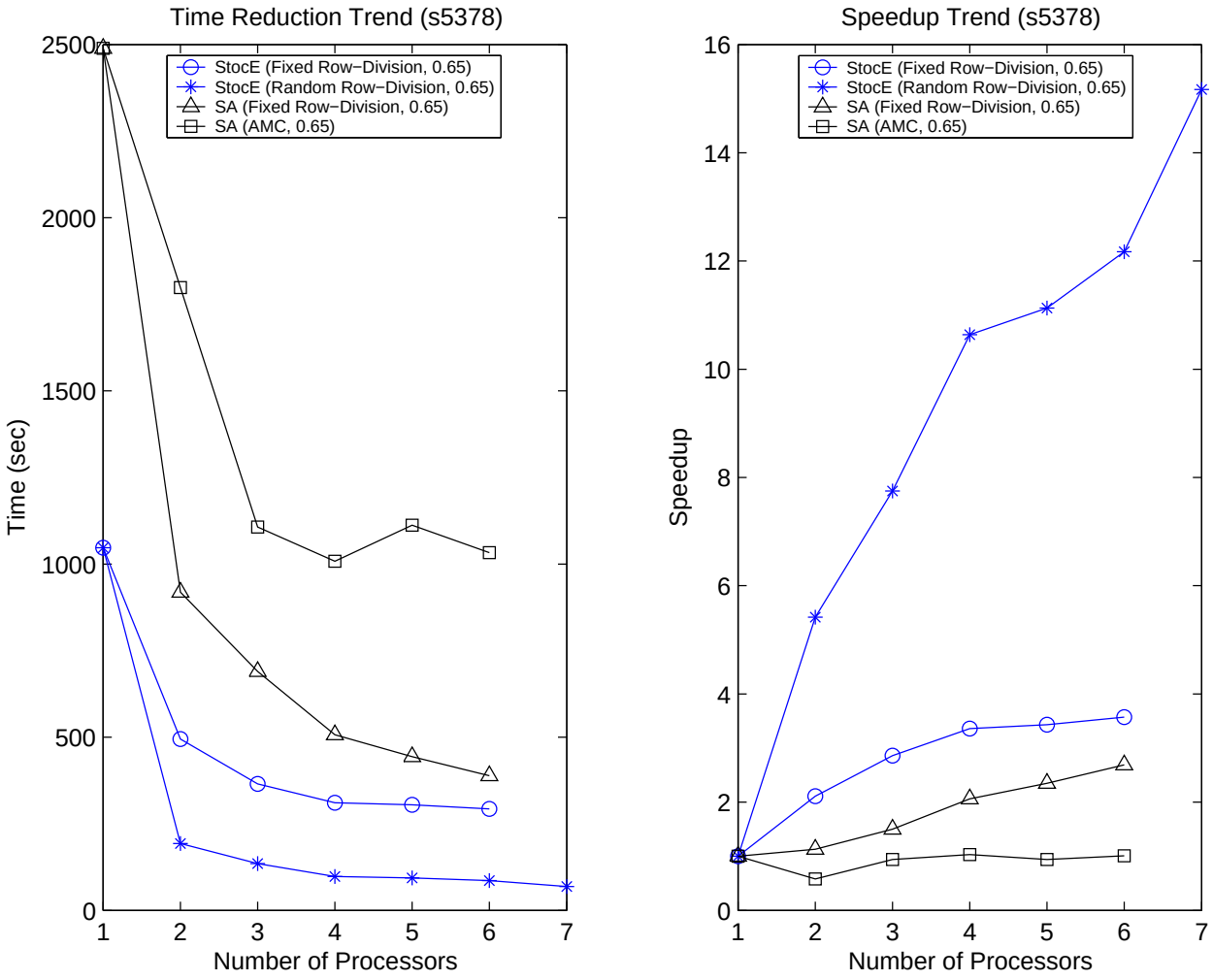

Figure 6.3: StocE Vs SA, s5378(0.65), StocE random row-division outperforms all by achieving the speedup around 15 with 7 processors. Where StocE fixed row-division and SA AMMC produced the speedups up to 3.5 and 2.75 respectively with 6 processors. SA fixed row-division was not able to produce any speedup on this circuit.

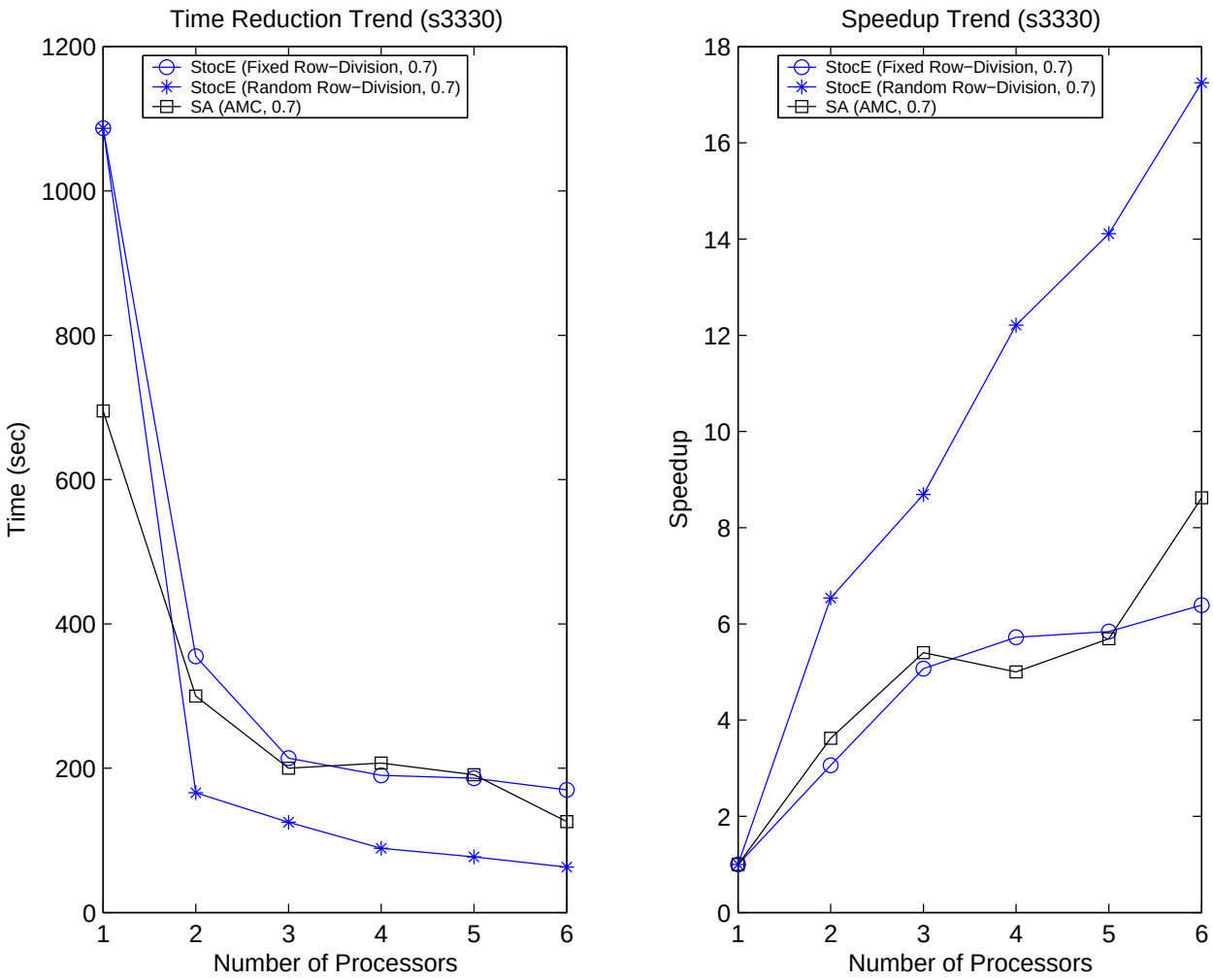

Figure 6.4: StocE Vs SA, s3330(0.7), StocE random row-division outperforms all by achieving the speedup of above 17 with 6 processors. Where StocE fixed row-division and SA AMMC produced the speedups up to 8.5 and 6.25 respectively with 6 processors. Results for SA fixed row-division were not available for this circuit.

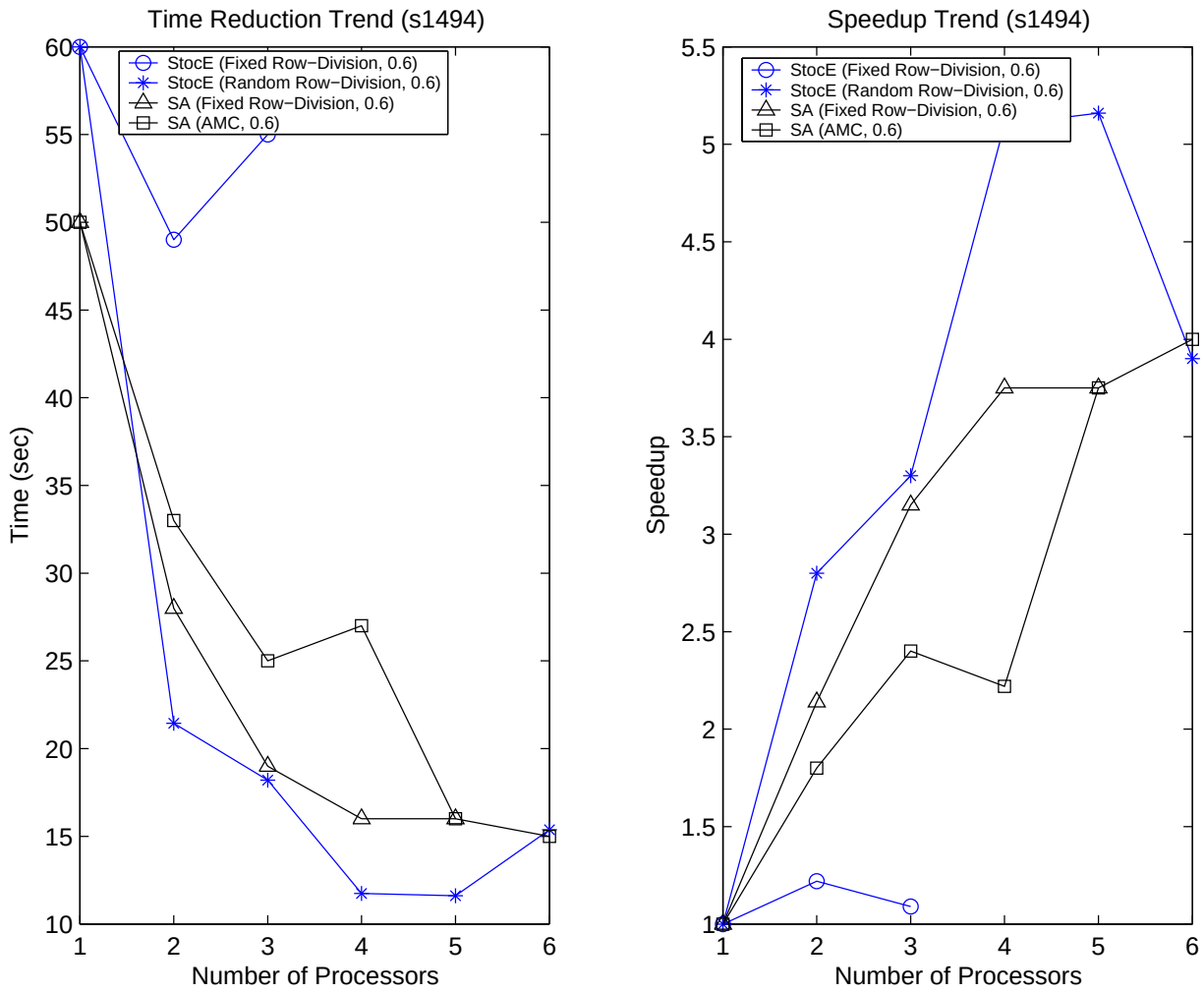

Figure 6.5: StocE Vs SA, s1494(0.6), StocE random row-division outperforms all by achieving the speedup of above 5 with 5 processors. Where SA fixed row-division and SA AMMC produced the speedups of up to 3.75 and 4 with 5 and 6 processors respectively. In case of StocE random row division, increasing an extra processor reduced the speedups from 5.1 to less than 4. StocE fixed row-division was not able to produce any speedup on this circuit.

Chapter 7

CONCLUSION AND FUTURE WORK

7.1 Conclusion

This research work focused on engineering three possible parallelization models for StocE for the VLSI cell placement problem. Low-Level parallelization of StocE appeared as an ineffective approach given the distributed computation environment. A parallel search model using an AMMC approach was designed and implemented for StocE and found to give very limited speedups. Considering domain decomposition, fixed and random row-division strategies were designed and implemented. Both the strategies gave excellent results when compared to the parallel implementation of other heuristics. StocE-AMMC approach reported runtime gains for very

few processors where as the same parallelization scheme was reported to work well with Simulated Annealing. Row-Division method distributed the workload effectively, allowing very good speedups for large circuits with large number of rows. It however does not perform very well for smaller circuits, as the runtime gains achieved by dividing computation, quickly saturate. This Row-based division was further enhanced by modifying the row distribution method which further reduced the run-times. Run-times achieved were compared against the run-times of parallel Simulated Annealing with AMMC and Fixed Row-Division model and were found the lowest among the two heuristics. Hence, the domain decomposition (Type-II) parallel strategies for StocE proves to be better than all the parallel versions of Simulated Annealing.

7.2 Future Work

In ongoing work, we focus on strategies which are variants of the ones reported here, some new strategies as well as hybrid models. Also, combining the characteristics of StocE with Tabu Search's memory components may very well lead to even further runtime reduction and speedup.

Bibliography

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

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

[3] M. Garey and D. Johnson. *Computer and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman, San Francisco, 1979.

[4] E. H. L. Aarts and J. Korst. *Simulated annealing and Boltzmann machines*. Wiley, 1989.

[5] Sadiq M. Sait, M. R. Minhas, and J. A. Khan. Performance and low-power driven VLSI standard cell placement using tabu search. *Proceedings of the 2002 Congress on Evolutionary Computation*, 2002.

[6] R. M. Kling and P. Banerjee. ESP: Placement by simulated evolution. *IEEE Transaction on Computer-Aided Design*, 1989.

[7] Youssef G. Saab and Vasant B. Rao. Stochastic evolution: A fast effective heuristic for some generic layout problems. *27th ACM/IEEE Design Automation Conference*, pages 1–6, 1990.

[8] Van-Dat Cung, Simone L. Martins, Celso C. Ribeiro, and Catherine Roucairol. Strategies for the parallel implementation of metaheuristics. May 2001.

[9] R. Azencott. *Simulated annealing: Parallelization techniques*. Wiley, 1992.

[10] D.R. Greening. Parallel simulated annealing techniques. *Physica*, pages 293–306, 1990.

[11] Erick Cant-Paz. Markov chain models of parallel genetic algorithms. *IEEE Transactions On Evolutionary Computation*, 2000.

[12] Johan Berntsson and Maolin Tang. A convergence model for asynchronous parallel genetic algorithms. *IEEE*, 2003.

[13] Lucas A. Wilson, Michelle D. Moore, Jason P. Picarazzi, and Simon D. San Miquel. Parallel genetic algorithm for search and constrained multi-objective optimization. *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, 2004.

[14] Hiroyuki Mori. Application of parallel tabu search to distribution network expansion planning with distributed generation. *IEEE Bologna PowerTech Conference*, 2003.

[15] Michael Ng. A parallel tabu search heuristic for clustering data sets. *International Conference on Parallel Processing Workshops*, 2003.

[16] Sadiq M. Sait, Habib Youssef, Junaid A. Khan, and Aiman El-Maleh. Fuzzified iterative algorithms for performance driven lowpower VLSI placement. *IEEE Proceedings of the International Conference on Computer Design*, 2001.

[17] Ewing Lusk. MPI in 2002: Has it been ten years already? *Proceedings of the IEEE International Conference on Cluster Computing*, page 1, 2002.

[18] Rubin Wang, Hatsuo Hayashi, and Zhikang Zhang. A stochastic nonlinear evolution model of neuronal activity with random amplitude. *Proceedings of the 9th International Conference on Neural Information Processing (ICONIP'OZ), Vol. 5*, pages 2497–2501, 2003.

[19] Lae-Jeong Park Cheol Dae-Hyun Lee, Hoon Choi and Hoon Park Seung Ho Hwang. A stochastic evolution algorithm for the graph covering problem and its application to the technology mapping. *IEEE Conference*, pages 475–479, 1996.

[20] S. Varadarajan, N. A. Ramakrishna, and M. A. Bayoumi. A stochastic evolution based register allocation using multiport memories. *IEEE Conference*, pages 472–475, 1993.

[21] Ashok Kumar and Magdy Bayoumi. A novel scheduling-based cad methodology for exploring the design space of asics for low power. *IEEE Conference*, pages 115–118, 1998.

[22] K. Ueda, T. Komatsubara, and T. Hosaka. A parallel processing approach for logic module placement. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, 2(1):39–47, January 1983.

[23] A. Iosupovici, C. King, and M. Breuer. A module interchange placement machine. *Proceedings of 20th Design Automation Conference*, pages 171–174, 1983.

[24] S. A. Kravitz and R. A. Rutenbar. Placement by simulated annealing of a multiprocessor. *IEEE Transactions on Computer-Aided Design*, CAD-6(4):534–549, July 1987.

[25] John A. Chandy, Sungho Kim, Balkrishna Ramkumar, Steven Parkes, and Prithviraj Banerjee. An evaluation of parallel simulated annealing strategies with application to standard cell placement. *Proceedings of the 9th International Conference on VLSI Design*, January 1996.

[26] Mahmood R. Minhas and Sadiq M. Sait. A parallel tabu search algorithm for optimizing multiobjective VLSI placement. *Springer-Verlag Berlin Heidelberg*, pages 587–595, 2005.

[27] Sadiq M. Sait, Mohammed Faheemuddin, Mahmood R. Minhas, and Syed Sanaullah. Multiobjective VLSI cell placement using distributed genetic algorithm. *ACM*, June 2005.

[28] P. Banerjee and M. Jones. A parallel simulated annealing algorithm for standard cell placement on a hypercube computer. *Proceedings of the 1986 International Conference on Computer-Aided Design*, pages 34–37, 1986.

[29] P. Banerjee, M. Jones, and J. Sargent. Parallel simulated annealing algorithms for standard cell placement on hypercube multi-processors. *IEEE Transactions on Parallel and Distributed Systems*, pages 91–106, 1990.

[30] J. A. Chandy and P. Banerjee. Parallel simulated annealing strategies for VLSI cell placement. *Proceedings of the 9th International Conference on VLSI Design*, 1996.

[31] J. A. Chandy, S. Kim, B. Ramkumar, S. Parkes, and P. Banerjee. An evaluation of parallel simulated annealing strategies with applications to standard cell placement. *IEEE Transactions on Computer Aided Design*, pages 398–410, 1997.

[32] J. S. Rose, W. M. Snelgrove, and Z. G. Vranesic. Parallel cell placement algorithms with quality equivalent to simulated annealing. *IEEE Transactions on Computer-Aided Design*, pages 387–396, 1998.

[33] Sadiq M. Sait, Habib Youssef, Hassan R. Barada, and Ahmad A1-Yamani. A parallel tabu search algorithm for VLSI standard-cell placement. *IEEE International Symposium on Circuits and Systems*, 2000.

[34] Ahmad, Al-Yamani, Sadiq M. Sait, and Hasan R. Barada. HPTS: Hetrogeneous parallel tabu search for VLSI placement. *IEEE*, 2002.

[35] Sadiq M. Sait, Syed Sanaullah, Ali Mustafa Zaidi, and Mustafa I. Ali. Comparative evaluation of parallelization strategies for evolutionary and stochastic heuristics. *GECCO05*, 2005.

[36] Srinivas Devadas and Sharad Malik. A Survey of Optimization Techniques Targeting Low Power VLSI Circuits. *32nd ACM/IEEE DAC*, 1995.

[37] A. Chandrakasan and T. Sheng and R. W. Brodersen. Low Power CMOS Digital Design. *Journal of Solid State Circuits*, 4(27):473–484, April 1992.

[38] Sadiq M. Sait and Habib Youssef. Timing-influenced general-cell genetic floorplanner. *Microelectronics Journal*, 28(2):151–166, 1997.

[39] Habib Youssef, Sadiq M. Sait, and K. Al-Farra. Timing-influenced force directed floorplanning. In *European Design Automation Conference Euro-DAC 95*, pages 156–161, September 1995.

[40] P. Cheung, C. Yeung, S. Tse, C. Yuen, and W. Ko. A new optimization cost model for VLSI standard cell placement. In *IEEE International Symposium on Circuits and Systems*, pages 1708–1711, June 1997.

[41] Sadiq M. Sait, H. Youssef, and Ali Hussain. Fuzzy simulated evolution algorithm for multiobjective optimization of VLSI placement. In *IEEE Congress on Evolutionary Computation*, pages 91–97, July 1999.

[42] Mahmood R. Minhas. Iterative algorithms for timing and low-power driven VLSI standard cell placement. Masters Thesis, KFUPM, Dhahran, Saudi Arabia, 2001.

[43] H. J. Zimmerman. *Fuzzy Set Theory and Its Applications.* Kluwer Academic Publishers, 3rd edition, 1996.

[44] Sadiq M. Sait, Habib Youssef, and Ali Hussain. Fuzzy Simulated Algorithm for Multiobjective Optimization of VLSI Placement. *IEEE Congress on Evolutionary Computation*, pages 91–97, July 1999.

[45] L. A. Zadeh. Outline of a new approach to the analysis of complex systems and decision processes. *IEEE Trans. Systems Man. Cybern*, SMC-3(1):28–44, 1973.

[46] L. A. Zadeh. The concept of linguistic variable and its application to approximate reasoning. *Information Science*, 8:199–249, 1975.

[47] Ronald R. Yager. Multiple objective decision-making using fuzzy sets. *International Journal of Man-Machine Studies*, pages 9:375–382, 1977.

[48] Ronald R. Yager. Second Order Structures in multi-criteria decision making. *International Journal of Man-Machine Studies*, pages 36:553–570, 1992.

[49] Ronald R. Yager. On Ordered Weighted Averaging Aggregation Operators in Multicriteria Decision Making. *IEEE Transaction on Systems, MAN, and Cybernetics*, 18(1), January 1988.

[50] http://www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html.

[51] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering.* Addison-Wesley, 1995.

[52] Malay Haldar, Anshuman Nayak, Alok Choudhary, and Prith Banerjee. Parallel algorithms for fpga placement. *ACM*, 2000.

[53] Selim G. Akl. *Parallel Computation: Models And Methods.* 1997.

Vita

- Khawar Saeed Khan
- Received B.E. (Bachelor of Engineering) degree in Computer Engineering from National University of Sciences and Technology, Rawalpindi, Pakistan in 2002.
- Joined Computer Engineering Department at KFUPM, Saudi Arabia in 2003.
- Completed M.S. (Master of Science) degree requirements in Computer Engineering at KFUPM, Saudi Arabia in 2006.