Parallelization of Stochastic Evolution for Cell Placement

MS Thesis Proposal

Khawar Saeed Khan

Student ID: 220514

Computer Engineering Department

College of Computer Sciences & Engineering

King Fahd University of Petroleum & Minerals

Contents

List of Figures

Abstract

VLSI physical design and the problems related to it such as placement, channel routing, etc, carry inherent complexities that are best dealt with iterative heuristics. However the major drawback of these iterative heuristics has been the large runtime involved in reaching acceptable solutions especially when optimizing for multiple objectives. Among the acceleration techniques proposed, parallelization is one promising method. Distributed memory multiprocessor systems and shared memory multiprocessor systems have gained considerable attention in recent years of research. This idea of parallel computing has attracted both the researchers and manufacturers who are targeting to reduce the time to market. Our objective is to exploit the benefits of parallel computing for a time consuming placement problem in VLSI. Finding the best solution for the placement of n modules is a hard problem. Thus the enumerative search techniques, specially those which employ the brute force, are unaccepted for the circuits in which n (number of modules) is large. Constructive and Iterative heuristics play the key role in this scenario and hence are frequently used. We will use Stochastic Evolution for finding the optimal solution to the above mentioned placement problem where the major task in our objective will be the parallelization of Stochastic Evolution using different parallelization techniques and the comparison between these different parallelized versions based on the results achieved. The parallelization will be carried out using MPI (Message Passing Interface) on a distributed memory multiprocessor system and conclusion will be based on the results achieved that are expected to show speedup nearly equal to linear speedup when run over increasing number of processors.

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. 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 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.

Literature survey of current researches being carried out in all VLSI fields reveals that until now the objectives that have emerged as the prime objectives in optimizing VLSI circuit designs are wirelength (area), performance (timing) and power dissipation. Wirelength and performance being the target objectives from quite past whereas power dissipation emerged quite recently as a critical objective of VLSI circuit design specially in power-constrainted applications such as mobile phones and laptop computers. The problem of optimizing these three objectives in VLSI circuit design can be addressed at any of design steps. In this work, we are concerned with optimization at the physical level, in particular the cell placement phase.

1.1 Low power VLSI standard cell placement

The VLSI cell placement problem involves placing a set of cells on a VLSI layout, given a netlist which provides the connectivity between each cell and a library containing layout information for each type of cell. This layout information includes the width and height of the cell, the location of each pin, the presence of equivalent pins, and the possible presence of feed through paths within the cell. The primary goal of cell placement is to determine the

best location of each cell so as to minimize the total area of the layout and the length of the nets connecting the cells together. With standard cell design, the layout is organized into equal height rows, and the desired placement should have equal length rows. Standard cell placement is a crucial step in the layout of VLSI circuits which has a major impact on the final speed and cost of the chip. This problem has been studied for several years and the best solutions have been obtained by iterative improvement algorithms such as simulated annealing, genetic algorithm and stochastic evolution which have some mechanism for escaping from local minima.

1.2 Iterative Heuristics

In combinatorial optimization the complexity of any algorithm that searches for better solutions increases with the increase in modules present in a given workspace where this complexity may also increase linearly. Considering the time limits it becomes impossible to apply a brute force technique which evaluates all possible combinations in a solution space with large number of modules. Moreover, the search space in these combinatorial problems consists of many frequent high or low jumps and deep craters due to which a searching criteria that only accepts good solutions (greedy algorithms) once trapped in a crater or a local optima will never come out of it. These greedy algorithms may search efficiently in scenarios where there is a parabolic search space with only one minima but for a combinatorial optimization problem they will never work. Another technique to search in the environments of combinatorial problems is a random walk. Studies have shown that given the limited CPU time algorithms that apply random walk performs worst.

1.2.1 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 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 seach 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 toward '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 is heavily dependent on how in any problem the cost function and mechanism to traverse the search space is developed. Also, the configuration and tuning of the control parameters in these heuristics have a major and critical impact on performace. However, as described earlier, since, these heuristics are sequential in nature thus the platform on which these heuristics or algorithms are executed also have an impact on runtimes of these heuristics. These runtimes becomes crictical when we talk about combinatorial problems with very large number of modules since any reduction in time makes the algorithm a prior algorithm for that specific problem.

1.2.2 Iterative Heuristics parallelization

1.2.2.1 Introduction Given the combinatorial problems with large number of modules, requiring large runtimes of the applied iterative heuristics, it becomes important to reduce these runtimes either by carrying out configuration tuning of the control parameters involved or by any other effective methods. Two (2) methods to reduce these large run times have been mentioned in [1].

1. Hardware acceleration, and
2. Software accelration.

Hardware acceleration requires the computationally intensive part of an algorithm to be mapped on hardware. This increases the cost in terms of resources and the speed up achieved in the end does not encourage one to opt for hardware acceleration, thus, it is not a cost effective strategy. On the other hand, software accelration can be carried out using multiprocessor environment using either MISD or MIMD machine.

1.2.2.2 Some Parallelized Heuristics Almost all of the above mentioned iterative heuristics have already been parallelized and their results as compared to their sequential counter parts have encouraged researchers to ponder in this direction. It is worth mentioning that none of the research has been carried out on the parallelization of Stochastic Evolution. Following are some details on parallel versions of some iterative heuristics:

- **Parallel Simulated Annealing**

  Applicability of three different parallel simulated annealing (SA) strategies to the problem of standard cell placement was investigated in [2]. 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 [2]:

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 [3]. 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 [4]. 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.

2 Parallelization

2.1 Introduction

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 runtimes with parallel processing but also handle larger problem sizes, obtain better quality results, etc. These potential advantages are enumerated and detailed below [5]:

1. **Faster Runtimes:** 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 [6], 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.

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,
   - 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.

For this particular problem we have selected MPI for parallelizing our code for standard cell placement. We have discussed MPI in more details in the following section.

2.2 MPI

2.2.1 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. [7]

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

2.2.2 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

$$((1\text{-}p) + p/n) \text{ x (serial running time)}$$

This suggests the importance of carefully idenifying 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 bandwith is one way to collectively assess the many factors that influence interprocess communication.

In any paralleization the major objective is always to acheive 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 send.

2.2.3 Parallelizing I/O Operations

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

2.2.3.1 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.

2.2.3.2 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. The appropriate MPI library routine for this approach is MPI_GATHER, and

3. Each process writes its data sequentially to a shared file. Use routine MPI_BARRIER to synchronize the processes and avoid data corruption.

2.2.4 Parallelizing Loops

Parallelizing loops among multiple processors is a critical task in parallelization. Three (3) possible solutions for distributing loops that are not nested among multiple processors are as follows:

1. Block Distribution

   The block distribution approach divides the loop iterations into p equal-sized parts, where p is the number of parallel processes. You can either make the parts as evenly sized as possible, so that all processes get the same number of interations. This allows the use of routine MPI_REDUCE to manage results afterward. Use a specified part size to create the iteration blocks, and leave a remainder (possibly but usually not zero) for one process. The PESSL library uses this method.

2. Cyclic Distribution

   Cyclic distribution assigns loop iterations to parallel processes one iteration at a time, round robin. In some situations (e.g., LU factorization) this can balance the workload better than block distribution, but it can also cause frequent cache misses.

3. Block-Cycle Distribution

   Block-cyclic distribution assigns loop iterations to parallel processes by first dividing them into equal-sized blocks and then assigning the blocks to processes round robin, cyclicly. The goal is to reduce cache misses yet still get the workload balance of cyclic distribution.

For parallelizing nested loops we can parallelize them in a way that minimizes the communication between processes as well as the frequency of cache misses if we consider:

- Storage order for multidimensional arrays. Fortran stores such arrays in column-major order, but C stores them in row-major order,
- Dependence of each element on its neighboring elements in the same row, and
- Possible dependence of an element on its neighbors in more than just one dimension.

2.2.5 Message Passing

Message passing is the solution to the distributed memory problem of processes that do not have data required for computational iterations where the data has been distributed among multiprocessors. Two basic cases occur:

1. No order (loop-carried) dependencies exist among the iterations.
2. Order (loop-carried) dependencies do exist among the iterations that are distributed.

2.2.5.1 No Order Dependencies When there exist no dependencies among the iterations we can carry out the following steps to pass messages among processors:

1. Broadcasting Single Element,
2. 1-D Finite Difference Method,
3. Bulk Data Transmissions, and
4. Reduction Operations.

2.2.5.2 Order Dependencies When there exist dependencies among iterations, i.e., the loops are nested, there are two (2) ways of parallelization:

1. Pipeline Method

   The pipeline method is the way to parallelize a loop that has a flow dependence, so that each iteration has to be executed strictly in order. The Incomplete Cholesky Conjugate Gradient Method is an example of such a situation. In this method, there is no danger of deadlock.

2. Twisted Decomposition

   This is the way to parallelize when one loop is flow dependent on one dimension of a matrix, and a second loop is simultaneously flow dependent on the second dimension of the matrix. Here, unlike with the pipeline method, there is a danger of deadlock.

2.2.6 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. [7]

3 Stochastic Evolution

3.1 Introduction

In biological processes, most species adapt themselves better to the existing environment as they evolve from one generation to the next one. The evolution process hopefully eliminates some of the bad characteristics of the old generation resulting in a better new generation. This concept has been exploited in iterative improvement techniques for some specific combinatorial optimization problems. [8]

Stochastic Evolution is a powerful general and randomized iterative heuristic for solving combinatorial optimization problems. The first paper describing Stochastic Evolution appeared in 1989. The paper was authored by Youssef Saab and Vasant Rao. [1]

Stochastic Evolution is an instance of the class of general iterative heuristics discussed in [9]. It is stochastic because the decision to accept a move is a probabilistic decision. Good moves, i.e., moves which improve the cost

function are accepted with probability one, and bad moves may also get accepted with a non-zero probability. Thus feature gives Stochastic Evolution hill-climbing propoerty. The word evolution is used in reference to the alleged evolution processes of biological species. Like simulated annealing and simulated evolution stochastic evolution is conceptually simple and elegant. Actually stochastic evolution is somehow inspired in part by both simulated annealing and simulated evolution. [1]

3.2 Sequential StocE

3.2.1 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). [8]

3.2.2 StocE Algorithm

Stochastic Evolution algorithm, shown in Figure 1, seeks to find the global minimum in a given search space. During this search the algorithm when stucked into a local minimum 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 another parameter p is defined equal to p_0. This main

```
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 1: The Stochastic Evolution algorithm.

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 funtion *PERTURB*, shown in Figure 2, is called.

In *PERTURB* funtion, 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

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: The PERTURB function.

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 numer 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 compound 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 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

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 3: The UPDATE procedure.

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 [8] that values ranging from $10 - 20$ are recommended.

3.2.3 StocE Applications

Since the development of Stochastic Evolution algorithm, it has been employed to solve several problems. Its implementation depends on the problem type it is being applied on. Following general steps are always required to implement StocE on any problem:

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 criteria.

StocE has also been used to solve the following hard combinatorial optimization problems [1]:

- Network bisection problem,
- Vertex cover problem,
- Hamiltonian circuit problem,
- Traveling salesman problem, and
- StocE based technology mapping of FPGAs.

Literature survey reveals the application of StocE in solving variety of problems. Some recent applications and their brief details are listed below:

1. **Graph covering problem**

   Stochastic Evolution algorithm is applied to solve the graph covering problem in which 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 the VLSI synthesis area. Experimental results with some selected benchmark circuits show that *StocE* algorithm produces better results than traditional tree mapping algorithm within a reasonable range of run time. Though efforts have been made to reduce the run time, the *StocE* algorithm still takes far more time compared with SIS. Experiences from this work show that *StocE* can be a good altenative for constrained optimization problems like graph covering problems. [10]

2. **StocE non-linear model of neuronal activity with random amplitude**

   A new *stochastic nonlinear evolution* model is proposed with the stochastic amplitude in neuronal activities to obtain the average number density, which is used to describe the neurocommunication among population of neurons. The average number density is a function of the amplitude, phase and time. The number density of the diffusion process of neurocommunication is given for the active states of populations of coupled oscillators under perturbation by both periodic stimulation and random noise. Particularly, the evolution model presented in this problem can be used to describe the *stochastic evolution* process of the amplitudes in activities of multiple interactive neurons. [11]

3. **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 (CMU-DA), 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 this approach aIlocation of registers to multiport memories proceeds in a way as to minimize the interconnection between 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 readwrite ports per bank would best suit the application. The algorithm is implemented and tested in on standard benchmarks. The approach yields good results. [12]

4. **Sceduling-based CAD methodology for low-power ASICs' design space exploration**

   This problem describes a novel approach to scheduling with multiple supply voltages in the high-level synthesis of ASICs. In a significant shift from the existing scheduling algorithms for multiple voltages, the proposed approach considers, identifies, and exploits the maximal parallelism available in an initial schedule, and applies a modified *stochastic evolution* mechanism to iteratively improve, or re-schedule, the previously obtained best schedule to reduce the maximal power consumption of function-units. [13]

3.2.4 Implementation Details

In this particular project, StocE algorithm is employed to solve performance driven low-power VLSI standard cell placement problem. It is a multi-objective problem where the objective is to reduce power, width and delay of the over all solution. Besides being objective, width is also a constraint in this problem. Thus, a code is developed in C language that reads the bench files and the other necessary files to have complete data structures. The complete problem statement and cost functions used in the problem are described in detail in Section-4. This section contains the actual implementation details of StocE algorithm in the problem mentioned above.

In the current implementation, initially when StocE funtion is called, an initial solution, a range variable p_0 and a termination parameter R, which is

equal to 10, are passed as arguments. The range variable p_0 is calculated by calculating the standard deviation in cost when each new solution is generated, i.e., the standard deviation of the cost for new solution from the average cost of a solution. In future, this p_0 calculation criteria may be changed with any alternative criteria, yet to be investigated, that may prove more beneficial. Once the code enters into StocE function then as per general StocE algorithm description, the cost for the initial solution is calculated and is saved as best cost and current cost. A sorted array is generated which decrementally sorts the cells in the initial solution according to their connectivity. This sorted array helps to bias the *PERTURB* funtion by moving the most heavily connected cells first. The code then enters into the main loop where the current cost is saved as previous cost. *PERTURB* funtion is called afterwards for the compound move where the first cell is always selected according to the ordering of sorted array and second cell is randomly selected. The two selected cells are then swapped and thus the solution goes into a new state. The cost of this new solution is calculated and gain 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. Here, we are dealing with objective value maximizing thus if the gain is greater than the random number generated the solution is accepted. Since, we are dealing with objective function maximization this criteria of accepting solutions intrinsically make sure that if the gain calculated is positive, i.e., new cost is greater than previous cost, the solution is always accepted. Whereas, if the gain calculated is negative, i.e., the new cost is less than the old cost, then the solution is only accepted when the gain is greater than the negative random number generated since the range variable defines the negative range. After the compound move of *PERTURB* funtion, the cost of new solution is calculated and is saved as current cost. *UPDATE* procedure is called afterwards where as discussed earlier if the cost of current solution and cost of previous solution are found equal then the range variable p_0 is incremented. In our case, considering the very small values of gain and rage variable p_0, we multiply p_0 by 10. Else, if the two (2) costs are not equal then the range variable p_0 is initialized with its first calculated value. After returning to the main loop from *UPDATE* procedure current cost is compared with the best cost. If current cost is found smaller than the best cost, the current cost is saved as best cost and the algorithm awards itself by decrementing ρ by R. Else, if current cost is found greater than the best cost ρ is incremented by one (1). The code or algorithm keeps searching for better solutions until ρ becomes equal to R.

3.3 Parallel StocE

As discussed already, StocE has iterative and blind nature. Due to these characterstics of StocE it may require large run times if employed on some big problems having thousands of moving elements and CPU intensive cost functions. Thus, parallelizing StocE using software acceleration technique, discussed in section 1.3.2, it is expected to get the same results in much less time. Thus, the whole objective of parallelizing StocE is to get the same or better results in much less time when compared with sequential implementation results.

3.3.1 General Description

Compared to simulated annealing, StocE has an intrinsic and highly sequential nature which makes it a difficult candidate for parallelization. A possible parallelization approach is a master-slave configuration where an initial solution is assigned to all slave processors by the master. If enough information is available about search space of the problem one can divide the search space among processors such that each slave has a non-overlapping search region assigned to it. In some cases processors may search the search space in parallel with a minimal overlap among them. But most of the times, this assumption is not true since very less is known about search space of the given problem. Possible strategies for parallelization of StocE are described in [1] where it is stated that parallelization can be carried out using the following two (2) approaches:

1. Move acceleration, and
2. Parallel moves.

In *move acceleration* each simple move can be performed in parallel and since each move has some trial relocations thus these trial relocations must be divided among processors. Master processor will remain in-charge of accepting or rejecting any simple move based on better solution cost criteria. Thus, parallelizing simple moves will make the execution faster. This communication intensive strategy of *move acceleration* can be improved by using *parallel moves* strategy where in *parallel moves* strategy several moves are performed in parallel.

Consider the situation where master orders p simple moves to p processors. In this case, each slave will report to master after completing its move and all the processors are forcced to synchronize after each trial. Master accepts

the best move from p simple moves and again orders for p simple moves. Thus, slaves are responsible for running ***PERTURB*** function where master remains responsible for updating ρ and R. This was an example of *move accelration* In a *parallel move* strategy, implementation would be such that the movable elements will be divided among available processors and each processor is in charge of trial relocations for each movable element assigned to it. Synchronization is forced only when any slave processor comes with a better solution cost, thus, communication will be less in this particular strategy. A possible parallel StocE algorithm is shown in Figure 4.

3.3.2 Implementation Details

As the current and initial design for parallelizing StocE algorithm. We have implemented parallelization using asynchronous multiple markov process where each slave processor is executing ***PERTURB*** and also is in-charge of updating ρ and R. Each slave is forced to communicate with master once it finds a better solution than the solution it already has where it sends its solution to master and master sends its solution to slave. The best solution among the two (2) is kept by both and the slave starts searching for a better solution again. In this way, a good solution found by any slave is propagated to other slave processors as well. Slave processors keep searching till they meet their termination criteria. Once all the slaves terminate, master has the best solution with it. This strategy has produced some interesting results which has motivated us to investigate further in this direction.

4 Problem Formulation and Cost Functions

4.1 Problem Statement

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 we 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, we deal with standard cell design, where all the circuit cells are constrained to have the same height,

AlgorithmParallel_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 = $(\bigvee_{i=1}^{p} A_i = True)$ */

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**

$BestS$= $NewS$;

$\rho = \rho - R$

Else $\rho = \rho + 1$;

EndIf

EndIf

Until $\rho > R$;

Return ($BestS$)

End. /*$Parallel\_StocE$*/

Figure 4: General parallel stochastic evolution algorithm where synchronization is forced after each trial.

while the width of the cell is variable and depends upon its complexity [1].

4.2 Cost Function for Power

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

$$p_{total} = \sum_{i \in V} p_i (C_i \times V_{dd}^2 \times f_{clk}) \times \beta \tag{1}$$

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.

$$C_i = \sum_{j \in F_i} C_j^g + C_{ij}^r \tag{2}$$

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.

4.3 Cost Function for Delay

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

$$T_\pi = \sum_{i=1}^{k-1} (CD_{vi} + ID_{vi}) \tag{3}$$

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.,

$$ID_{vi} = LF_{vi} \times C_{vi} \tag{4}$$

$SLACK_\pi$ of path π is given by

$$SLACK_\pi = LRAT_\pi - T_\pi \tag{5}$$

where $LRAT_\pi$ is the latest required arrival time and T_π is the path delay [14, 15]. 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, we shall use the approach reported in [14] to predict the K-most critical paths. The placement program will seek to satisfy the delay constraints imposed by these paths.

4.4 Cost Function for Wirelength

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, 16]. A Steiner tree approximation described below, which is fast and fairly accurate in estimating the wire length will be adopted in this work [17]. 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.

$$X = \frac{\sum_{i=1}^{n} x_i}{n} \qquad Y = \frac{\sum_{i=1}^{n} y_i}{n} \tag{6}$$

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

$$Steiner\ Tree = B + \sum_{j=1}^{k} P_j \tag{7}$$

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.

$$Interconnection\ Length = \sum_{l=1}^{m} Steiner\ Tree_l \tag{8}$$

where m is the number of nets.

In standard cell placement, cells (or blocks) of fixed heights are placed in rows. It is the width of these rows that varies with the proposed solution according to the type and number of cells placed in the row. An approximation would be to treat cells as points, but in order to estimate lengths of interconnects more accurately, widths of cells are taken into account. Heights of routing channels are estimated using the vertical constraint graphs constructed during the channel routing phase. With this information, a fairly accurate estimate of the overall area, delay, and power dissipation can be obtained.

5 Objectives

The objectives of this research project are stated as follows:

1. Since, Stochastic Evolution algorithm has never been parallelized before for any problem thus main objective of this research is to parallelize StocE algorithm using MPI to get good solutions in minimum time,
2. Investigation of applied strategies for different iterative heuristics,
3. Test and compare the sequential and parallel versions of StocE on big test cases like s15850, s35932, s38417 and s38584,
4. Translation of bigger test cases for single objective placement problem,
5. Compare the results of StocE parallelization with already parallelized heuristics like Simulated Annealing, Tabu Search, Simulated Evolution and Genetic Algorithm, and
6. Analyze the effect of medium dependency by executing the same sequential and parallel StocE algorithm on some physical media other than gigabit ethernet.

6 Tasks Outline

The tentative designed tasks' outline is stated as follows:

1. Study and use the cluster of workstation environment,
2. Design and implement using MPI different parallelization strategies for StocE algorithm,
3. Analyze the bottlenecks in parallel strategies in terms of CPU time and I/O requirements,
4. Compile results from sequential and parallel versions of StocE algorithm in terms of achieved cost solutions, time to best solutions, etc,
5. Test the implementation on different test cases ranging from s386 to s38584,
6. Compare the results from all employed parallelization strategies, report the most effective strategy and analyze details in terms of CPU time, I/O requirements, communication delays, etc,
7. Compare the results achieved through the best parallel strategy of StocE with the documented results of other parallelized iterative heuristics.

References

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

[2] Balkrishna Ramkumar Steven Parkes John A. Chandy, Sungho Kim 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.

[3] 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.

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

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

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

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

[8] 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.

[9] S. Nahar, S. Sahni, and E. Shragowitz. Experiments with Simulated Annealing. *Proceedings of 22nd Design Automation Conference*, pages 748–752, 1985.

[10] 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.

[11] 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.

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

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.