# PARALLELIZATION OF ITERATIVE HEURISTIC FOR PERFORMANCE-DRIVEN LOW POWER VLSI STANDARD CELL PLACEMENT

by

**SYED SANAULLAH**

A Thesis Presented to the
DEANSHIP OF GRADUATE STUDIES

In Partial Fulfillment of the Requirements
for the Degree of

## MASTER OF SCIENCE

IN

## COMPUTER ENGINEERING

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

Dhahran, Saudi Arabia

November 2003

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by

Syed Sanaullah

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. Abdul Waheed (Co – Chairman)

Dr. Saleh Al – Sharaeh (Member)

Dr. Sadiq M. Sait
Department Chairman

Dr. Osama A. Jannadi
Dean of Graduate Studies

Date

*Dedicated to*

*My Parents*

# 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 and co-chairman, Dr. Abdul Waheed for their guidance and patience throughout this thesis. Their continuous support, advise and encouragement can never be forgotten. I would also like to express my appreciation for Dr. Saleh Al-Sharaeh for his constructive comments. 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 also thank my beloved parents, and all my family members for their moral support throughout my academic career. I would also like to thank the research assistant community in KFUPM, especially Faheem, Yau, Amisu, PEL associates and others for their support.

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: Syed Sanaullah


Title: Parallelization of Iterative Heuristic for Performance-Driven Low-Power VLSI Standard Cell Placement


Major Field: COMPUTER ENGINEERING

Date of Degree: October 2003



*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 these heuristics is one promising alternate. 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 Tabu Search, 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 (ethernet), by using Message Passing Interface (MPI) communication libraries. Circuits from ISCAS-89 are used as benchmarks. Results for parallel Tabu Search are compared with its sequential counterpart as a reference point for both, the quality of solution as well as the execution time.*



**Keywords:** *VLSI physical design, Multi-objective Optimization, Low-Power, Performance, Placement, Parallelization, Tabu Search, Cluster Computing*


MASTER OF SCIENCE DEGREE

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

October 2003

# Chapter 1

# Introduction

With advances in VLSI (Very Large Scale Integration) technology, it is possible to achieve fabrication on the nano-technology scale with component densities of millions of transistors per circuit. Due to its complexity the VLSI design process is divided into several intermediate levels of abstraction. More details about the new design are introduced as the design progresses from higher to lower levels. Typical levels of abstraction, together with their corresponding design steps, are illustrated in Figure 1.1.

Until recently, physical design optimizing the wirelength (area) and performance (timing) was of prime importance. Results of several research efforts to optimize area and performance are available in literature and summarized in several papers and books [1, 2, 3, 4, 5, 6]. Nowadays, with several new power-constrained applications ranging from mobile phones to laptop computers, power dissipation has emerged as

another important objective of VLSI circuit design [7]. The problem of optimizing the three above mentioned objectives in VLSI circuit design can be addressed at any of mentioned design steps. In this work, the concern is with optimization at the physical level, in particular the cell placement phase.

In VLSI circuit design, placement is the process of arranging circuit blocks on a layout. In *standard cell* design, where circuit blocks are of fixed height and variable widths, placement consists of determining optimum positions of all blocks on the layout to satisfy a number of objectives.

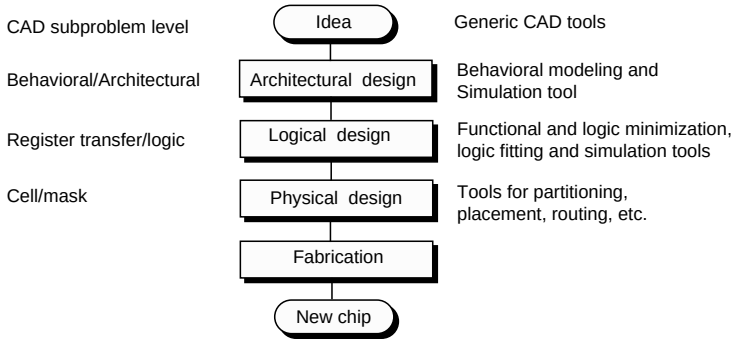


Figure 1.1: Levels of abstraction and corresponding design steps.

Placement is an NP-hard problem, whether the complexity is low or high, irrespective of single or multiple objectives. Conventional constructive techniques have often proved inadequate in achieving multi-objective optimization for placement. Iterative heuristics on the other hand have been a tremendous success in reaching acceptable solutions for such problems.

## 1.1 Iterative Non-deterministic Heuristics

The primary advantage of iterative heuristics over conventional constructive algorithms is their probabilistic ability to escape from local optima. Among the earliest of these heuristics, Simulated Annealing (SA) was proposed in the early eighties [8] and was considered a phenomenal success when applied to combinatorial optimization problems from a variety of disciplines. SA is often credited for sparking the research interest in iterative algorithms and is considered a comparison benchmark for new heuristics in its class. It has become customary that every newly reported randomized search heuristic has to prove itself by performing better than SA on a number of test cases. There are also other heuristics that are used in solving combinatorial optimization problems like Genetic algorithms, GRASP (Greedy Randomized Adaptive Search Procedures), Ant Colonies, but in this work, only Tabu Search is considered.

## 1.2 Parallel CAD

### 1.2.1 Motivation

Despite 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 ac-

celeration 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 [9]:

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 one viable alternate 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 [10], heuristics used to solve them may give non-optimal solutions. The solutions obtained are a 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 [11].

### 1.2.2 Challenges

Albeit it is relatively easy to build parallel machines whose peak performance is in few hundred GigaFlops (GFlops) or even TeraFlops (TFlops), it is quite difficult to harness this computational power. The challenge lies in design of good and efficient parallel algorithm, that can use this hardware resources to get maximum performance.

The use of parallel algorithms for VLSI CAD is thus important as these represent a rich class of important, non-numerical, unstructured applications, that are difficult to parallelize. Researchers exposed to parallel algorithms, typically use structured numerical problems, e.g., in computational chemistry, but have less idea about the intricacies of solving a large non-numerical, unstructured problem. In addition to this, there is a growing need by most of the CAD tools that are used to automate VLSI physical design.

## 1.3 Parallel Processing Models

In this section some of the available parallel processing models and architectures are discussed. There exist two well-known parallel architectures for implementing parallel processing models. General purpose multiprocessors in which each processor can access a common shared address space are classified as *shared memory multiprocessors*. As compared to this model, in a *distributed memory multiprocessor* each processor has its own local memory.

The shared memory model, as the name implies, allows all processors to access data from a shared memory without any explicit declarations. A high-speed bus serves as the interconnection medium between these processors. In distributed memory model, data is shared between the memories of individual processors through explicit message passing. Here the latter is discussed since it is most suitable to the proposed work.

The distributed memory models have evolved into cluster computing systems, using complete stand-alone workstations connected through dedicated low-latency networks. Although such clusters comprising general-purpose workstations provide an affordable and cost-effective means of generating enormous computing power, the challenge lies in achieving their effective utilization. In these parallel computing systems, communication between nodes (workstations) places a significant overhead on the actual runtime due to the network latency. Hence, process communica-

tion among the nodes has to be minimized. Effective partitioning of the computational load among the workstations is another important aspect, to minimize overlap between individual work-domains. Hence, harnessing the computational power of cluster systems through efficient parallel programming is inarguably, a significant challenge.

**Parallel Programming environments:** In order to exploit concurrency, Parallel algorithms have to be designed by taking into consideration the underlying architecture, which effects the degree of parallelism or granularity of the application it can run. The reason for this is that the best parallel strategy cannot be achieved by just adopting a sequential algorithm [12].

Basically, there are three strategies for developing parallel programs. [11]

1. **Parallel Programming Languages:** These are sequential languages augmented by a set of special system calls and provide low-level primitives for message passing, process synchronization, etc. Examples of such languages are HPF (High Performance Fortran) and OpenMP. Such programming languages are well suited to fine-grain synchronous applications.

2. **Communication Libraries:** Communication libraries provide routines that can be used with regular programming languages like C, C++, and Fortran. These communication construct/routines can be augmented within the code to send and receive messages among different processors. PVM (Parallel Virtual

Machines) and MPI (Message Passing Interface) are examples of this type. This programming mode is particularly suited to coarse-grain applications running on clusters of PCs/workstations.

3. **Lightweight Processes (Threads):** With the increase in the use of SMP cluster, threads such as POSIX (Portable Operating System Interface) threads or Java threads are also used. Although, threads are an operating system concept, they are extensively used in parallel programming due to their support for concurrency programming. Their use is particularly suited to medium and coarse-grained parallel applications.

In this thesis, parallel algorithms for Tabu Search applied to multi-objective VLSI cell placement are presented. Parallel implementations are achieved over a dedicated computing cluster with individual nodes connected via an ethernet connection. The performance of these parallel algorithms in terms of run-time and search efficiency are compared with their respective sequential counterparts running on a single processor.

The overall work objective is to develop Parallel algorithm for Tabu Search, on message passing architectures to achieve, either of the two:

1. that can run in lesser time, achieving similar quality of solution, thereby achieving scalability with the increase in the number of processors.

2. that can run for similar amount of time and get better quality of solution.

## 1.4 Thesis Organization

This thesis is organized as follows: In Chapter 2, some theoretical aspects of parallelizing heuristics are discussed along with a review of related literature. Problem formulation is dealt with in Chapter 3, where we discussed the parallel environment, hardware, software, and the paradigm. Various approaches to the solve the problem are reported in Chapter 4 followed by the comparison with other heuristics in Chapter 5, with Conclusion and Future work are discussed in Chapter 6.

# Chapter 2

# Literature Review

This chapter gives a short introduction to Tabu Search heuristic and literature review of the efforts that were done to parallelize it for various optimization problems.

## 2.1 Tabu Search (TS)

Several heuristics are used in solving combinatorial optimization problems. Tabu Search is one such heuristic. The word tabu is a variant of 'taboo', which means excluded or forbidden from use. In context of the heuristic, it maintains a list of moves which are forbidden, in order to prevent cycling.

Conceptually, Tabu Search is elegant. It is a generalization of local search. The heuristic is based on systematic exploration of memory functions. It is an aggressive search technique where for a given solution, a large number of neighbors are gener-

$\Omega$ : Set of feasible solutions.
$S$ : Current solution.
$S^*$ : Best admissible solution.
$Cost$ : Objective function.
$\aleph(S)$ : Neighborhood of $S \in \Omega$.
$\mathbf{V}^*$ : Sample of neighborhood solutions.
**T** : Tabu list.
**AL** : Aspiration Level.

**Begin**
Start with an initial feasible solution $S \in \Omega$.
Initialize tabu lists and aspiration level.
**For** fixed number of iterations **Do**
    Generate neighbor solutions $\mathbf{V}^* \subset \aleph(S)$.
    Find best $S^* \in \mathbf{V}^*$.
    **If** move $S$ to $S^*$ is not in **T** **Then**
        Accept move and update best solution.
        Update tabu list and aspiration level.
        Increment iteration number.
    **Else**
        **If** $Cost(S^*) <$ **AL Then**
            Accept move and update best solution.
            Update tabu list and aspiration level.
            Increment iteration number.
        **EndIf**
    **EndIf**
**EndFor**
**End**.

Figure 2.1: Algorithmic description of short-term Tabu Search (TS) [1]

ated, from which the best is chosen. To determine which of the generated solutions is the best, an *evaluator* that is based on the objectives being optimized and the historical information that has been accumulated, is used [13, 14, 15]. The trace of the current solution is controlled by a recent move history to avoid cycling in the solution space. This is done with the help of a list called the Tabu list, which stores the most recently visited moves. Moves in this list are tabu. However, the tabu status is overridden when an aspiration criteria is satisfied. Use of intensification/diversification in Tabu Search considerably helps in obtaining superior quality solutions. This is accomplished with the help of additional memory structures that keep record of information such as frequencies of moves, elite solutions, etc. An algorithmic description of short-term Tabu Search is given in Figure 2.1. [1, 2].

## 2.2 Parallelization of Tabu Search (TS)

A generic intuitive strategy for achieving parallelization is to partition the data into small subsets distributed among the processors [16, 17]. Each processor is responsible for a data subset and implements a sequential version of the concerned heuristic over this data subset.

However, for combinatorial optimizations, three types of parallelization strategies seem to be appropriate as were reported by Toulouse et. al [18]. They are:

1. The operations within an iteration of the solution method are parallelized.

2. The search space (problem domain) is decomposed.

3. Multi-search threads with various degrees of synchronization.

Crainic et. al classified the parallel Tabu Search heuristic based on a taxonomy along three dimensions [19]. Based on this taxonomy, they also classified the earlier work done related to parallel Tabu Search, which is presented in the Section 2.3.

- The first dimension is **Control Cardinality**, where the algorithm is either *1-control* (one processor executes the search and distributes tasks to other processors) or *p-control* (each processor is responsible for its search and communication with other processors).

- The second dimension is **Control and Communication Type**, where the algorithm can follow a *rigid synchronization (RS)*, *knowledge synchronization (KS)*, *Collegial (C)*, or a *Knowledge Collegial (KC)* strategy. RS and KS belong to synchronous operation mode where the process is forced to establish communication and exchange information at specific points explicitly defined, where as C and KC are asynchronous operation modes where contents of communication are analyzed, concerning the global characteristics of good solutions and search strategy.

- The third dimension is **Search Differentiation** where the algorithm can be *SPSS (single point single strategy)*, *SPDS (single point different strate-*

*gies)*, *MPSS (multiple point single strategy)*, or *MPDS (multiple point different strategies)*.

The second dimension, i.e., Control and Communication type is basically divided into two types: Synchronous and Asynchronous.

## Synchronous Parallel Tabu Search

In synchronous version, a master process runs on one machine and the slaves processes run on different machines. All slave processes start with an initial solution, either sent by the master process, or generated by the slave itself, depending on the third dimension of the taxonomy. If it is *1-control*, then the master sends the initial solution. Otherwise, in *p-control* the slaves generate their own initial solutions. After searching its part of the current neighborhood, each slave process reports its best solution back to the master. The master process selects the best among the received best solution subject to tabu conditions. The slave processes maintain their own tabu list and apply their own aspiration criteria, depending on the first dimension of the taxonomy, i.e. if it is *1-control*, only the master process maintains a Tabu list and aspiration criteria. If the stopping criteria (number of iterations) are met then the search stops; otherwise the master broadcasts the selected solution back to the slaves and the search continues. The main idea behind this is the communication between the master and slave processors at fixed points, that enforces it to

synchronize.

The main steps of the synchronous parallel Tabu Search are outlined in Figure 2.2.

**Algorithm** ParallelTabuSearch;
**Begin**
1. Generate initial solution on all processors
   (In case of 1-control, the master does it & broadcasts)
2. **For** Number of Iterations
3. Each slave process explores its own neighborhood and
   sends its best solution to the master;
4. The master selects the best solution from the solutions
   received from the slaves;
5. The master maintains the Tabu list & applies aspiration criteria
   (In case of 1-control, only the master maintains it)
6. The master broadcasts the best solution so far to all the slaves
7. **EndFor**

**End**.

Figure 2.2: Skeletal outline of synchronous parallel Tabu Search.

## Asynchronous Parallel Tabu Search

In this approach, each processor explores a subset of the neighborhood of its current solution. Each processor is competing with its neighbors (its adjacent processors) in finding a superior solution. When the stopping criteria are met, every processor reports its best solution. There is no fixed point at which all the processors have to synchronize or communicate to exchange information.

## 2.3 Literature Review of Parallel Tabu Search

The first reported studies on the parallelization of Tabu Search were published in the early nineties. The following paragraphs summarize the breadth of applications, where Parallel Tabu Search has been applied.

Malek et. al [20] compare the performance of serial and parallel implementations of simulated annealing and Tabu Search for Travelling Salesman Problem (TSP). The reported experiments were performed on a 10 processor Sequent Balance 8000 computer. The authors report that the parallel version of Tabu Search outperforms not only its sequential counterpart, but produces comparable or better results than the serial and parallel version of simulated annealing. Their strategy was synchronous with 1-control, KS, SPDS.

In order to solve the flow shop sequencing problem, Taillard [21] used a mechanism for parallel implementation of Tabu Search algorithm using search space decomposition strategy. It is a *1-control, RS, SPSS algorithm.*

Battiti et. al [22] used Tabu Search to solve the Quadratic Assignment Problem (QAP) with hashing procedures. The scheme used is the p-control, RS, MPSS. The authors report that the parallelization strategy is efficient and the average success time decreases with increase in the number of processors.

Taillard [23] used parallel Tabu Search for vehicle routing problem. The parallelization strategy is based on domain decomposition strategy, which is p-control,

KS, MPSS. It was implemented on a Silicon Graphics 4D/35 workstation with 4 processors.

Another effort by Taillard [23] to apply parallel Tabu Search to quadratic assignment problem follows a 1-control, RS, SPSS strategy. A ring of 10 transputers were used, but no implementation details were given. Load balancing through partition of the neighborhood is acknowledged as critical.

Chakrapani et al. [24] also used parallel Tabu Search to solve QAP, which is 1-control, RS, SPSS. The search is performed sequentially, while the move evaluation is done in parallel. The implementation is specifically designed for Connection Machines CM-2: a massively parallel SIMD machine. The authors report that the best known solutions were obtained which required significantly lesser number of iterations. Furthermore, they were also able to determine good suboptimal solutions to bigger problems in reasonable time. Same authors applied a similar strategy to the problem, approximated by a very large quadratic assignment problem with sparse flow matrix. Very good results were reported on a 8192 processor hypercube configuration of a CM-2 Connection Machine.

Another effort to parallelize Tabu Search for travelling salesman problem by Fiechter [25], used the p-control, KS, MPSS strategy. Intensification and diversification steps were implemented in the synchronous version. The algorithm was implemented on a network of transputers arranged in a ring structure. The authors report near-optimal solutions to large problems and almost linear speed-ups.

Another parallelization of Tabu Search for vehicle routing problem by Garica et. al [26] also uses search space decomposition strategy. It is also 1-control, RS, SPSS algorithm. The authors report a noticeable improvement in solution quality over one of the best constructive algorithms for vehicle routing problem, with substantial reduction in runtime.

In order to improve parallel Tabu Search using evolutionary principles, the algorithm presented by Falco et. al [27] used multi-search thread strategy for the travelling salesman and quadratic assignment problems. It is a p-control, C, MPSS algorithm. The results reported indicate a marked improvement in solution quality as well as convergence speedup.

Another parallel Tabu Search algorithm for the 0-1 multidimensional knapsack problem was put forth by Nair [28], which used multi-search threads strategy. It is a p-control, RS, MPDS algorithm.

Taillard [29] also applied parallel Tabu Search for job sequencing problem, which is p-control, RS, MPSS strategy. Several parallelization ideas focusing on speeding up computations related to neighborhood evaluation, didn't yield good results, either because the communication overtook computation, or the available computing platform (a ring of transputers, and a 2-processor Cray) were not suitable.

Crainic et. al [19], the authors who put forth the taxonomy of classification of Tabu Search, present several of the strategies for both: synchronous and asynchronous, for multi-commodity location-allocation problem with balancing require-

ments. It was implemented on a heterogenous network of 16 SUNSparc workstations. The results show that the average gap improved in most of the cases, when the number of processors increased.

Mori and Hayashim [30], used parallel Tabu Search algorithm for voltage and reactive power control in power systems. Of the two schemes, one of them used the domain decomposition strategy, while the other scheme followed a multi-search threads strategy. The first one is 1-control, RS, SPSS and the second one is p-control, RS, MPDS algorithm.

A recent work by Yamani et.al [31], parallelized Tabu Search for VLSI cell placement on heterogenous cluster of workstations, using PVM. The algorithm was parallelized on two levels simultaneously. The higher parallelization level can be classified as p-control while the lower level was 1-control. The synchronization strategy was RS and MPSS search differentiation strategy was used for both the levels. The authors report obtaining proportional speed-up in most of the cases.

This brief literature review can be summarized as below:

- The taxonomy proposed classify all parallelization strategies presented earlier. Also, it is independent of the particular problem class or computing platform.
- Synchronous version of parallel Tabu Search seems to be used more often than its counterpart, and the results show that they outperform serial ones.
- The use of parallelism improves the performance of Tabu Search procedures

in most of the cases. Thus, it is definitely worth the effort to explore alternate parallelization strategies.

# Chapter 3

# Problem Description and Cost Functions

In this chapter, the cell placement problem is 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 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 [32, 33]. 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 $\beta$ 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 $\pi$ 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 $\pi$ is given by

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

where $LRAT_\pi$ is the latest required arrival time and $T_\pi$ is the path delay [34, 35]. If $T_\pi$ is greater than $LRAT_\pi$, then the path $\pi$ 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 [34] 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, 36]. A Steiner tree approximation described below, which is fast and fairly accurate in estimating the wire length will be adopted in this work [37]. 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 [38].

## 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 $\alpha$ 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 is either belong 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 [39]. 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 [40]. A linguistic variable is a variable whose values are words or sentences in natural or artificial language [41]. 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 [42].

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.

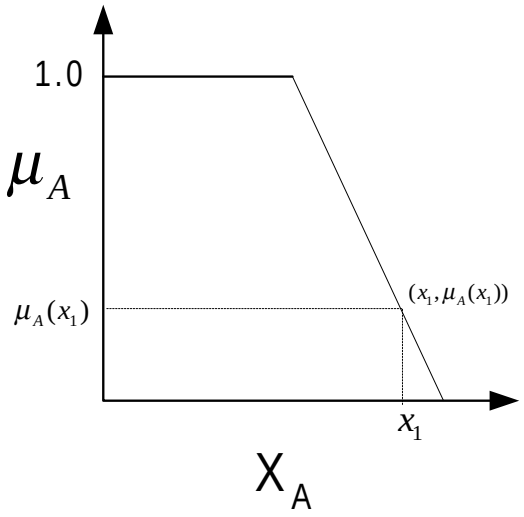


Figure 3.1: Membership function of a fuzzy set A.

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 [40].

### 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) [39]. 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 [43, 44]. This operator allows easy adjustment of the degree of "anding" and "oring" embedded in the aggregation. According to [43, 44], "orlike" and "andlike" OWA for two fuzzy sets $A$ and $B$ are

implemented as given in Equations 3.11 and 3.12 respectively.

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

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

$\beta$ 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 $\Gamma$ 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.

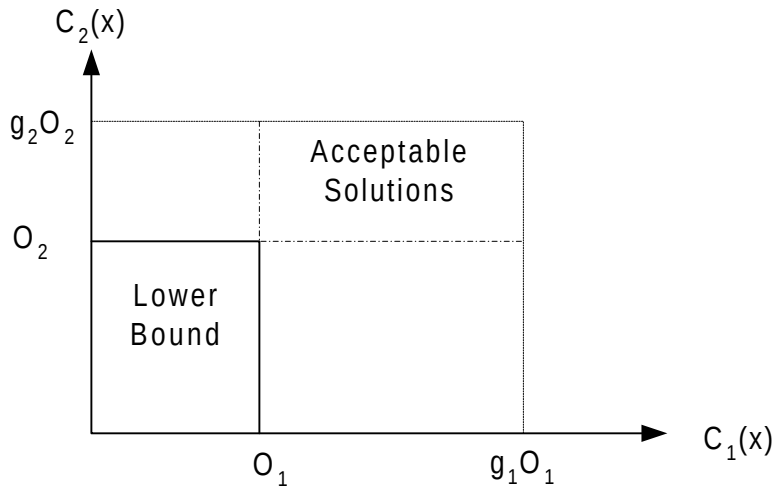


Figure 3.2: Range of acceptable solution set.

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 [38].

**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.

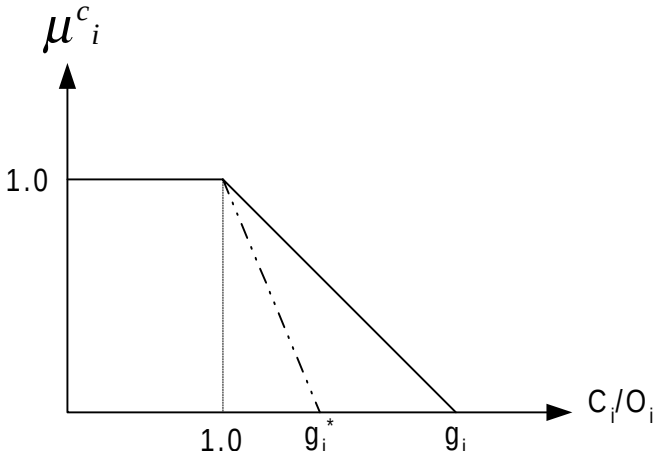


Figure 3.3: Membership functions within acceptable range.

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 $v_i$, $\pi_c$ is the most critical path with respect to optimal interconnect delays, $k$ is the number of nets in $\pi_c$ and $Width_i$ is the width of the individual cell driving net $v_i$.

Using the Ordered Weighted Averaging (OWA) operator [43, 45, 44], 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. $\beta$ 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. Tabu Search.

## 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.

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

**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.

***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.

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

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 |

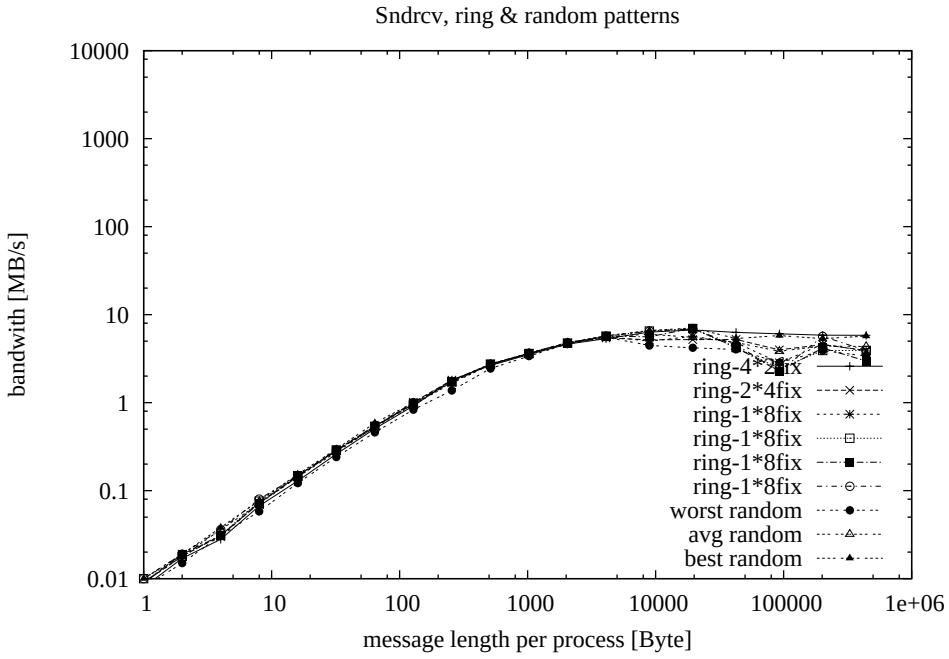


Figure 3.4: Bandwidth versus Message length per process

**Cluster Performance:** In terms of GFlops, the maximum performance of the cluster, with NAS Parallel Benchmarks was found out to be 1.6 GFlops, (using NAS's LU, Class A, for 8 processors). Using this same benchmark for a single processor, the individual performance of one machine was found out to be 0.3 GFlops.

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 $\mu$sec per message.

As for the bandwidth versus message-size, the message size should be in the range of 10KB to utilize the maximum bandwidth. Message size considerable 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

In this chapter, an approach to parallelization of Tabu Search for VLSI Standard-cell placement is discussed. First the sequential implementation of Tabu Search is analyzed.

## 4.1 Analysis of Sequential Implementation

The analysis of the sequential (serial) implementation of an application is an important step before developing its parallel version for the following reasons:

- Resources: The parallel version will definitely have more resources, in terms of number of processors, memory, hard-disk, I/Os than the serial one. Thus, the comparison of an un-optimized serial code with its parallel counterparts will not show correct performance gain.

- Allocation: The identification of bottlenecks and/or hot-spots in the serial code allows tuning of the application, which can help in allocating more resources (if possible, e.g., increasing physical memory) for any task.

- Insight: Profiling of the sequential code gives an insight about the functionality of the code e.g., number of times a function is called, part of the code that took most of the time, etc. This insight gives an idea as to which part of the code is amenable to parallelization.

There are three major steps involved in analyzing the sequential code. They are as follows:

**Profiling:** Profiling gives an idea of the sequential code performance.

**System Performance:** It shows the usage of system resources when the code is executed.

**Tuning:** Based on the code and system performance, recommendations are provided either to fine tune the code into an optimized sequential code or identify parts of code that can be parallelized.

Each of the above steps is explained below.

### 4.1.1 Profiling

Profiling shows where the program spent its time and the callee and the called functions while it was executing. This information can show which chunks of the program are slower than expected, and might be candidates for rewriting to make the program execute faster. It can also tell which functions are being called more or less often than expected. This may also help in spotting bugs that had otherwise been unnoticed.

Since the profiler uses information collected during the actual execution of the program, it can be used on programs that are too large or too complex to analyze by reading the source code. However, the profile data collected depends on how the code was executed. For example, if some features of the program are not used, while it is being profiled, the profile information will not be generated for those features.

Although there are several tools used for profiling the code, the GNU profiler (gprof), a built-in tool for profiling for C compiler is used.

Some of the most common steps in profiling a code include:

- The program must be compiled and linked with the option 'profiling enabled'.
- The program must be executed to generate the profile data file.
- Gprof must be executed to analyze the profile data.

Once the profiling steps are completed, the output from the profiling results are available for analysis. The output can be either a *flat profile* of the code, that shows

how much time was spent in each function or a *call graph* showing the callee and called functions.

Before going into details of the profile, the flowchart of the sequential code will give an idea of the program being executed. This is shown in Figure 4.1

As shown in the figure, in the initialization phase of the algorithm, an initial solution is generated randomly and its cost is calculated (This could have also been generated by enumerative techniques). For the circuit, best cost of its wirelength, delay, power is estimated. Combining these best costs using OWA operators (explained in 3) cost is considered to be the best solution that might be obtained, which is done by placing all the cells adjacent to each other and calculating the cost of each objective. Now, based on this best cost, the fitness of the randomly generated solution is obtained. The calculation of the fitness is based on the fuzzified ordered-weighted average method, and the value obtained is in the range [0,1].

After the initialization step, the rest of the steps are executed for a (parametric) fixed number of iterations. In each iteration, the $N$ neighboring solutions are generated from the current solution. For each neighbor of the current solution, a random different *move* [swapping any two cells] is made to perturb the solution. The cost of each of the new solution is calculated. The one with the best cost (and fitness) is selected to be current solution for the next iteration, even though its cost might be worse that the current solution or the best solution obtained so far. This is the hill-climbing feature of Tabu search, which allows it to explore unvisited regions in

Figure 4.1: Flowchart for Sequential Tabu Search for VLSI Cell Placement

the search space. The move that was made to obtain the new solution is checked if it was 'Tabu' by checking the tabu-list (short-term memory). If it is tabu, then it is checked against aspiration criteria. If the aspiration criteria is satisfied, it is accepted as the new solution, which is the current solution for the next iteration.

**Flat Profile:** The flat profile obtained is tabulated in Table 4.1. All the runs were executed for about 20% of the total runtime to obtain the profile with varying circuit (data) sizes.

Table 4.1: Flat profile of sequential code with varying data sizes.

| Circuit | Runtime (sec) | Wirelength (%) | Delay (%) | Power (%) | TS (%) | TS-Map (%) | Others (%) |
|---|---|---|---|---|---|---|---|
| s641 | 35 | 34.78 | 42.52 | 8.61 | 7.62 | 6.17 | + |
| s1494 | 33 | 62.12 | 10.49 | 11.2 | 8.85 | 7.30 | + |
| s9234 | 1272 | 49.86 | 3.49 | 9.22 | 10.13 | 26.50 | + |
| s15850 | 2838 | 50.75 | 2.75 | 8.85 | 9.63 | 26.92 | + |

In the table, the first and second column shows 'Circuit' used and its 'Runtime'. The 'Wirelength', 'Delay' and 'Power' columns, show the percentages of time spent in computing their respective costs. 'TS' is the time taken by the heuristic itself and 'TS-Map' is the time spent to map a given solution into the layout. This is proportional to the size of the circuit. The last column shows the time spent by other functions, which accounts for less that 2% of the total time, hence negligible.

As can be observed from Table 4.1, the calculation of wirelength took most of the time and is very much independent of the data size.

However, the delay calculation for the smaller circuits was found out to be more than the larger circuits. The reason for this is because delay is calculated from the number of paths, whose higher limit was set to 500. In smaller circuits, most of the critical paths were taken into consideration, while in larger circuits, a maximum of 500 paths were taken. Hence, less time is spent in delay calculation for larger circuits.

The time spent on calculation of power didn't fluctuate and accounted for less than 10% of the total time. The same can be said about the TS (Tabu Search heuristic, itself). The TS-Map increased as the data size increased.

**Call Graph:** The call graph shows the callee and the called functions. The call graph was generated by "GProf" as well as with, "VTune Performance Analyzer" by Intel. The latter was used, as it gives a graphical view as shown in Figure 4.2.

The call graph consists of nodes and edges. The names of the nodes are the function names. The thick edge shows the critical path, either in terms of the most time-consuming path or the call-sequence originating from the root or both.

As it is seen clearly from the call graph, most of the time is spent in cost calculation in every iteration. This is in confirmation with the flat profile obtained earlier.

Figure 4.2: Call Graph of the Sequential Tabu Search.

### 4.1.2 System Performance

In order to obtain the system performance, operating systems' built-in tools were used. They are:

- Vmstat (Virtual Memory Statistics): vmstat reports information about processes, memory, paging, block IO, traps, and cpu activity. The first report produced gives averages since the last reboot. Additional reports give information on a sampling period of length delay. The process and memory reports are instantaneous in either case.

- Top (displays top CPU processes): Top provides an ongoing look at processor activity in real time. It displays a listing of the most CPU-intensive tasks on the system, and can provide an interactive interface for manipulating processes. It can sort the tasks by CPU usage, memory usage and runtime. Most features can either be selected by an interactive command or by specifying the feature in the personal or system-wide configuration file.

- SAR (System Activity Report): The sar command writes to standard output the contents of selected cumulative activity counters in the operating system. The accounting system, based on the values in the count and interval parameters, writes information the specified number of times spaced at the specified intervals in seconds. If the interval parameter is set to zero, the sar command displays the average statistics for the time since the system was started. The

default value for the count parameter is 1. If its value is set to zero, then reports are generated continuously. The collected data can also be saved in the file, in addition to being displayed onto the screen.

The system performance for the sample runs that were executed and tabulated in Table 4.1 can be summarized as follows:

- CPU usage (user): 98.8 %
- CPU usage (system): 1.2%
- CPU usage (idle): 0.0%
- CPU usage (nice): 0.0%
- Average load on the system: 95%
- Number of interrupts/sec (including clock): 500 (otherwise normally: $< 120$)
- Number of context switches: 480 (otherwise normally: $< 150$)
- Memory Usage: 3.1 % (of the total: 256MB)

### 4.1.3 Tuning

This section describes a general tuning methodology. This general tuning methodology presents a high-level overview, beginning at the system level down to the

microarchitecture level. Tuning not only addresses the application, but improves the cost-benefit ratio of the system as well.

The first level of performance tuning is system-level tuning. The goal of system-level tuning is to optimize system resource utilization and speed up an application by improving the way it interacts with the system. System-level tuning usually gives the greatest speed-up with the least amount of effort (up to 3 times in most cases), making it a good place to start. It is especially relevant for I/O-intensive applications such as applications that are disk-intensive or network-intensive. However, as it is observed above, the application is very processor-intensive (and not I/O-intensive), the system-level tuning can be skipped to application-level tuning.

The second level of performance tuning is application-level tuning. The goal of application-level tuning is to speed up the application by improving the application's algorithms, threading implementation, and/or use of APIs or primitives. The tuning can be done by either sampling and/or call graph. The speedup obtained by this tuning could be twice as fast as that of the previous one.

Based on the results of the call graph, an attempt was made to tune the application by making algorithmic changes. The number of critical paths were reduce from an upper limit of 500 to 10. However, this resulted in shifting of cost calculation from delay to wirelength. Therefore, this option didn't contribute much to the tuning. Other alternatives, like threading implementations were not tried as there were no memory bottlenecks.

Since there are no significant application-level tuning, the next level is considered. The third level of performance tuning is microarchitecture-level tuning. The goal of microarchitecture-level tuning is to speed up the application's performance by improving how the application runs on the processor. This type of tuning is especially relevant for processor-intensive applications. The general methodology for microarchitecture-level tuning is as follows:

1. Find the most time-consuming code regions that have a high impact on application performance.

2. Analyze the execution of those code regions on the machine architecture.

3. Identify microarchitecture level performance problems.

4. Determine how to avoid them to improve performance.

Steps 1 and 2 were accomplished with the help of flat profile and call graph. For Steps 3 and 4, remote data collection (RDC) methodology was used. RDC supports 'Sampling', a non-intrusive, instruction-address method to collect, analyze, and display system-wide software performance data. It is an event-based sampling method that uses processor counters.

The following discussion summarizes the results obtained by event-based sampling.

**Problem: High First-level cache load misses:** The results show that there

are $\approx 85\%$ first level cache load misses. First level cache load misses cause a significant negative impact on performance . The reason is that the working data set is too large to fit into the first level cache.

In order to improve this situation, it is better to avoid them. It requires an understanding of the first level cache layout and architecture, the way in which the code has been designed with the data structures and access routines, and their arrangement in the compiled code. Optimization techniques such as blocking, loop interchange, loop skewing and packing can help avoid them. Other possible problems include branch misprediction, second level-cache load misses, trace cache misses. However, These methods are best implemented by the compiler itself.

The code optimization was done with built-in compiler optimization options (such as O1, O2, O3) as the above suggested methods are better handled by the compiler. With 'O1', the compiler tries to reduce the the code size and the execution time. With 'O2', the code is optimized even more. Nearly all supported optimizations that do not involve a space-speed tradeoff are performed. Loop unrolling and function inlining are not done. As compared to 'O1', this option increases both compilation time and the performance of the generated code. 'O3' option optimize yet more. This turns on everything 'O2' does, along with also turning on inline-functions.

The results for a single processor are tabulated in Table 4.2. The results show that runtime decreased by almost 50% in all the test cases. That is the speedup

was twice the non-optimized version. Profiling for the optimized code was not done as 'gprof' doesn't give correct function profiling information, if the code is compiled using optimization options, i.e., O3, as this option swaps assembly code among different procedures, for optimized timing performance.

Table 4.2: Runtimes for different ISCAS'89 circuits with and without compiler optimizations

| Circuit | Runtime (sec) (Without 'O3') | Runtime (sec) (With 'O3') |
|---|---|---|
| s298 | 115 | 56 |
| s386 | 210 | 104 |
| s641 | 3750 | 1865 |
| s832 | 319 | 160 |
| s953 | 798 | 391 |
| s1196 | 1531 | 752 |
| s1238 | 1515 | 755 |
| s1488 | 1086 | 538 |
| s1494 | 1094 | 541 |
| c3540 | 7721 | 3843 |
| s9234 | 21512 | 10717 |
| s15850 | 41023 | 20192 |

To summarize the tuning step, analysis was done as to how to proceed to tune the application, but at the end, it was found that the tuning did help to reduce the runtime, but the application still remained a processor intensive task.

### 4.1.4 Conclusion

Tabu Search is a CPU intensive task. This can be seen from profiling and the system performance, when the code was executed.

In sequential version, the major time spent is in cost recalculation of the new solution. Therefore, one possible option was to parallelize the cost computation in different iterations, i.e., cost of wirelength, delay, power to be calculated separately on each slave processor and send to the master processor for calculating the fitness. However, this solution would not have been feasible for two reasons:

1. There would be lot of communication overhead, as the cost is calculated in every iteration, if different costs are calculated on different processors. Also, very fine-grain synchronization results in communication taking over computation, especially in case of the small data sizes (small and medium-sized circuits). In case of large data sizes, the waiting time to synchronize increases the overall runtime.

2. In Tabu Search, the next solution is calculated based on the fitness of the previous one. Unless the fitness of the current solution is known, the new solution and its fitness cannot be calculated. i.e., there is data dependency.

Thus, just cost calculation parallelization is not considered. Hence, algorithmic parallelization approaches are taken into account.

## 4.2 Parallel Algorithm Design Steps

In the previous section, an analysis of the sequential code was presented. A successful attempt was made by optimizing the code based on its profiling results (see Ta-

ble 4.2). However, even though the runtime decreased, the task remained 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.

Parallel algorithms have four desirable attributes, i.e. Concurrency, Scalability, Locality, and Modularity. Concurrency and Scalability are managed by the communication libraries. Locality (high ratio of local memory accesses to remote memory accesses) and modularity (decomposition of complex entities into simpler components) are an important aspect of software engineering and depends on the algorithm.

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 [12].

### 4.2.1 Design Methodology

There are four distinct stages in designing parallel algorithms, which are: Partitioning, Communication, Agglomeration, and Mapping. Concurrency and scalability are addressed in the first two stages, while performance related issues are considered in the later stages. These steps are discussed in the subsequent paragraphs with reference to the VLSI Cell placement problem.

**1. Partitioning**

The first steps towards designing a parallel algorithm is to identify the subtasks that can be executed in parallel. The decomposition can either be domain-wise (data) or function-wise (computation). Tabu Search algorithm does not have any structured data structure, to which domain decomposition techniques can be applied. Therefore, fine-grain functional decomposition in which each processor will explore different subsets of neighborhoods with different tasks is to be used. Also, the issue of managing the global best solution, which must be available for access by all processors after every iteration needs to be addressed.

The algorithm has several functions, that can be divided, which can be executed in parallel. These are:

- Ts-map (Mapping the solution to the layout)
- Cost-wire (Calculating the cost of the wirelength)
- Cost-power (Calculating the cost of the power)

- Cost-delay (Calculating the cost of the delay)
- Fuzzy-cost (Calculating the Fuzzy-cost using the above costs)
- Check Tabu List and Aspiration Criteria
- Maintain a Global best cost

At this stage, it is not possible to tell whether this strategy is the best if all these functions are executed in different processors, once the communication strategy has been established. An probable deficiency with this design is that the number of tasks do not scale with the problem size, but the individual task size increases.

**2. Communication**

Although the tasks generated in the partition phase are intended to execute concurrently, generally they cannot, as computation performed in a task will require data associated with other tasks. With functional decomposition of tasks, it is relatively easier to identify communication requirements, as the communication corresponds to the flow of data between the tasks.

In the placement algorithm, the partitioned tasks should have communication to:

- find the coordinates of the cells before making a move,
- pass individual costs (wirelength, power, delay) to the fuzzy cost calculation,
- check Tabu List and Aspiration Criteria after getting the best cost, and

- communication to update the overall best cost found so far,

One possible approach to handle is the use of the master-slave model, in which master is responsible for maintaining tasks, that can be easily handled in a centralized manner. This approach is simple and might be efficient if the communication is inexpensive, computation is high in each iteration, and number of processors are not too many. Nevertheless, the centralized approach is essentially difficult to scale, if the number of processors increase. This is because, the master processor takes some time to process a request, hence, the number of requests it can handle efficiently is bounded.

To summarize the communication requirements as a master-slave process, the communication is:

- *Global:* Each task participates in communication with the master process,
- *Unstructured:* It is not a regular structure; such as a tree or graph,
- *Static:* Identity of communication partners do not change over time, and
- *Synchronous:* Processes synchronize at predefined points.

### 3. Agglomeration

In the agglomeration step of the design process, practical aspects for executing the algorithm on the the cluster of workstations is considered. In particular,

the tasks that can be combined (agglomerated), so as to provide a smaller number of tasks, each of greater size, with lesser inter-process communication

The main idea behind this is to reduce the communication overhead by grouping tasks that communicate very frequently. This is because, the communication does not only depend on the message size to be sent, but also depends on the fixed startup cost that is incurred. In blocking synchronous mode of communication, a processor has to stop computation, to send/receive messages. Speed-up can be thus, achieved by reducing communication overhead, as the overall runtime decreases.

One more additional concern in this stage is the software engineering costs that must be considered when parallelizing existing sequential code. This is very important especially in our case, as the sequential code is available and efforts to parallelize is to go through all the design checklists, but keeping in mind that extensive code changes have to be avoided.

**4. Mapping:**

This is the final stage of designing parallel algorithm, where each task is mapped to a processor. The mapping problem does not arise on a single processor or on a shared memory processor, that has automatic task-scheduling.

Based on the literature survey presented in Chapter 2, the strategy chosen

is the master/slave strategy, in which, the master processor distributes equal number of tasks to the slave processors. This strategy is effective for moderate number of processors. The design methodology studied above, will be used with the master/slave strategy, as explained in the subsequent paragraphs.

## 4.3 Parallel Models for Tabu Search

### 4.3.1 First Parallel Model - Algorithm A

The first parallel model is shown in Figure 4.3. This model is developed after applying the first two techniques, that is Partitioning and Communication. Basically, it is a Master-Slave strategy, in which the master is executed on a single machine and the slaves are executed on multiple machines. The master generates the initial solution, and sends it to the slaves (1-control). Each slaves, generates $N$ neighboring solutions and calculates the cost of each neighbor and selects the best neighbor. The best solution with its fitness is sent to the master by all the slaves and the master selects the best among these, to all the slaves as the initial solution in the next iteration (SPSS). Synchronization occurs after every iteration (RS). Thus, this can be classified as 1-control, RS and SPSS.

The profiling of this parallel code is done using VampirTrace to generate the

Figure 4.3: Flowchart for Parallel Tabu Search (1-RS-SPSS) Algorithm-A.

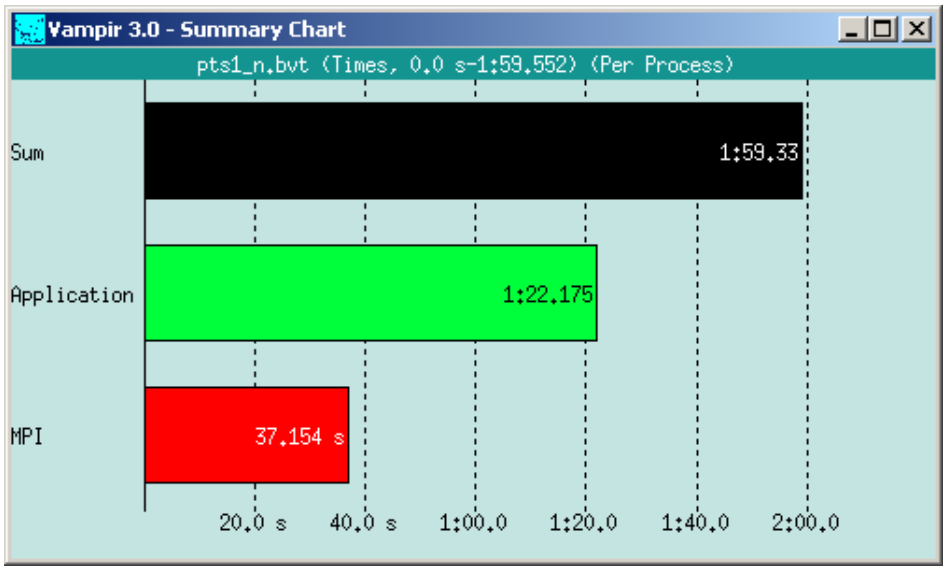


Figure 4.4: Summary of time spent in Application versus MPI routines: Algorithm - A.

trace file and Vampir to visualize the trace. The profiling was done on a circuit for some small amount of time. Figure 4.4 shows the summary of time spent in application code as well as MPI routine. The profiling was done for a sampled amount of time on the cluster of 8 workstations. The topmost horizontal bar in the figure shows the total runtime ($\approx$ 2 minutes). The second and the third bar shows the time taken by the 'application' and 'MPI routines', respectively. As it is seen, almost 19% of the time is spent in MPI routines, while the remaining time was spent for executing the application.

Figure 4.5 shows the timeline of the communication pattern. The horizontal bars represent the processors from 0 to 7. The vertical lines between the processors is the communication (send or receive) among them. The darker bar

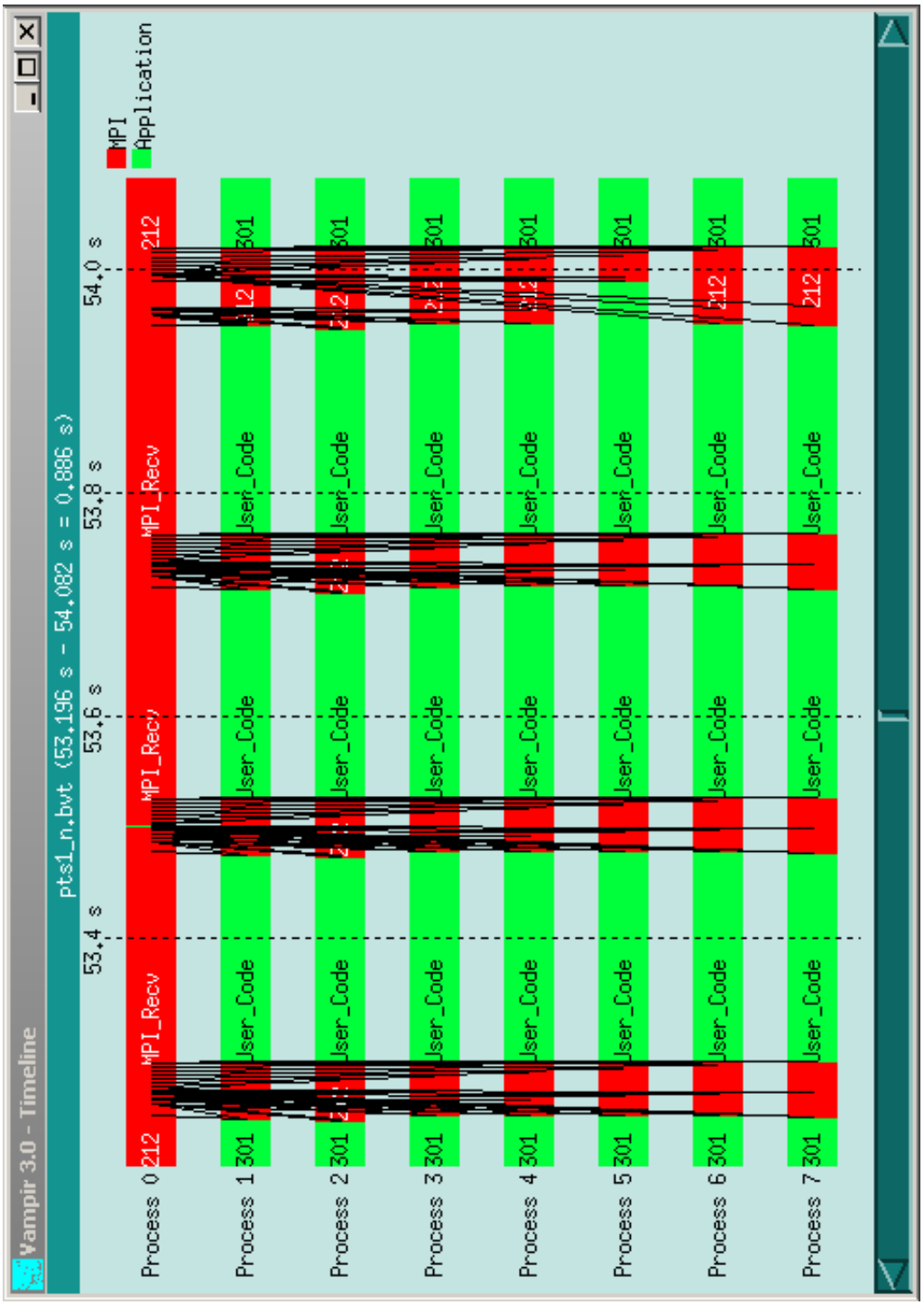


Figure 4.5: Timeline of communication pattern: Algorithm - A.

(red colored) indicates the MPI routine activity, while the lighter bar (green) shows the time spent in the executing the user application. As it can be seen from the figure, the communication is very frequent, and the master is idle for quite some time before it receives any messages from the slave processors.

Figure 4.6 shows the load on each of the 8 processors in form of a pie-chart depicting the percentages of time spent on application and MPI routines. The master process is busy with the MPI routines, while the others spend less than 25% of the total time. The load is almost equally balanced on all the slave processors.

Finally, the Figure 4.7 shows the average length of the messages sent among the processors. Processes from 0-7 on the left are senders while the ones on the top are the receivers. The first empty box shows that there is no message sent from Process 0 to itself. The box beside it with '20.235' is the size of the message sent from process '0' to process '1', and so on. Thus, the communication is between the master and the slave processes, but not among the slave processors. The length of the message send/received is greater than the minimum message size required to use the network bandwidth effectively.

The results of the parallel tabu search (Algorithm A) for VLSI cell placement is presented in Chapter 5.

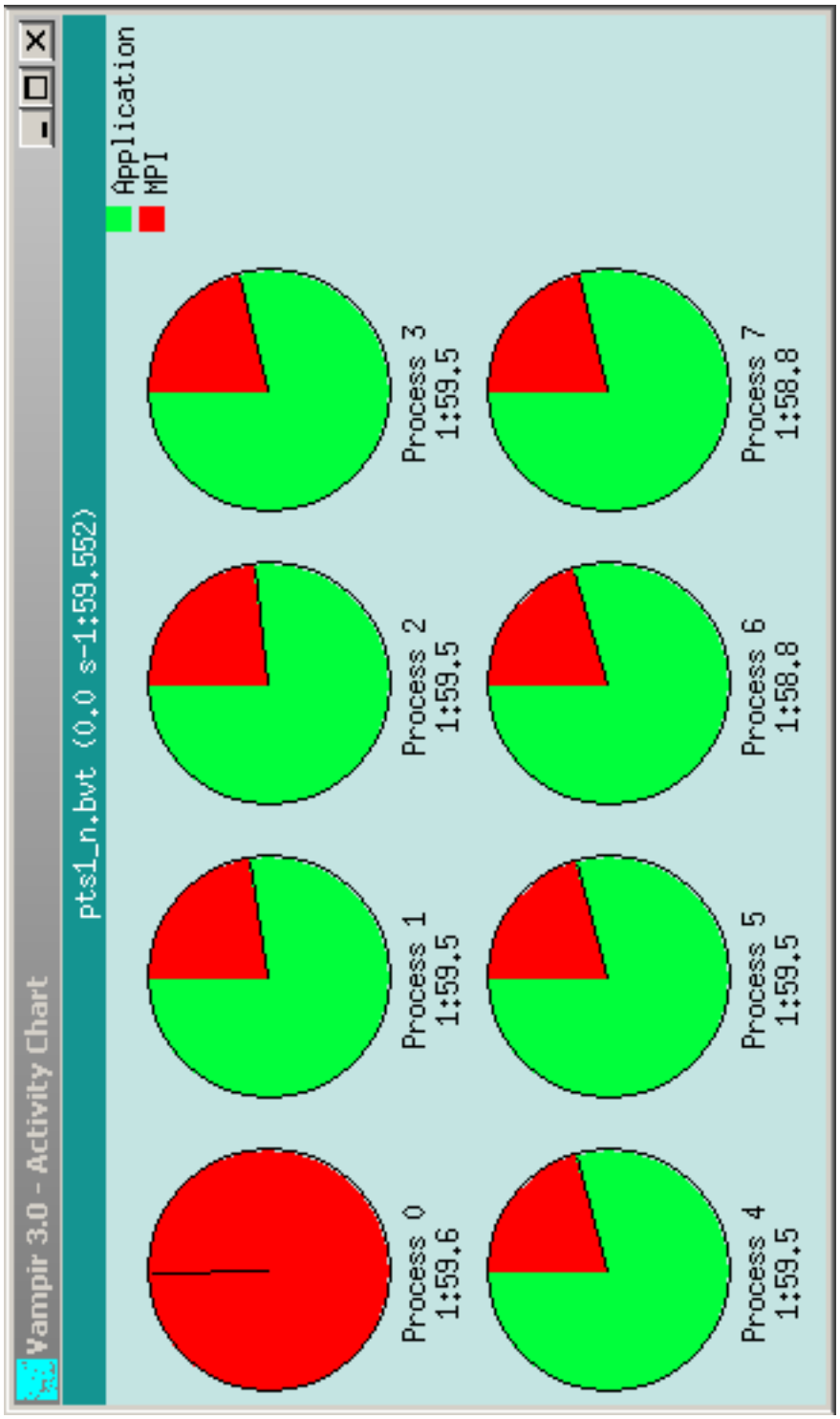


Figure 4.6: Load on each processor (Application versus MPI): Algorithm - A.

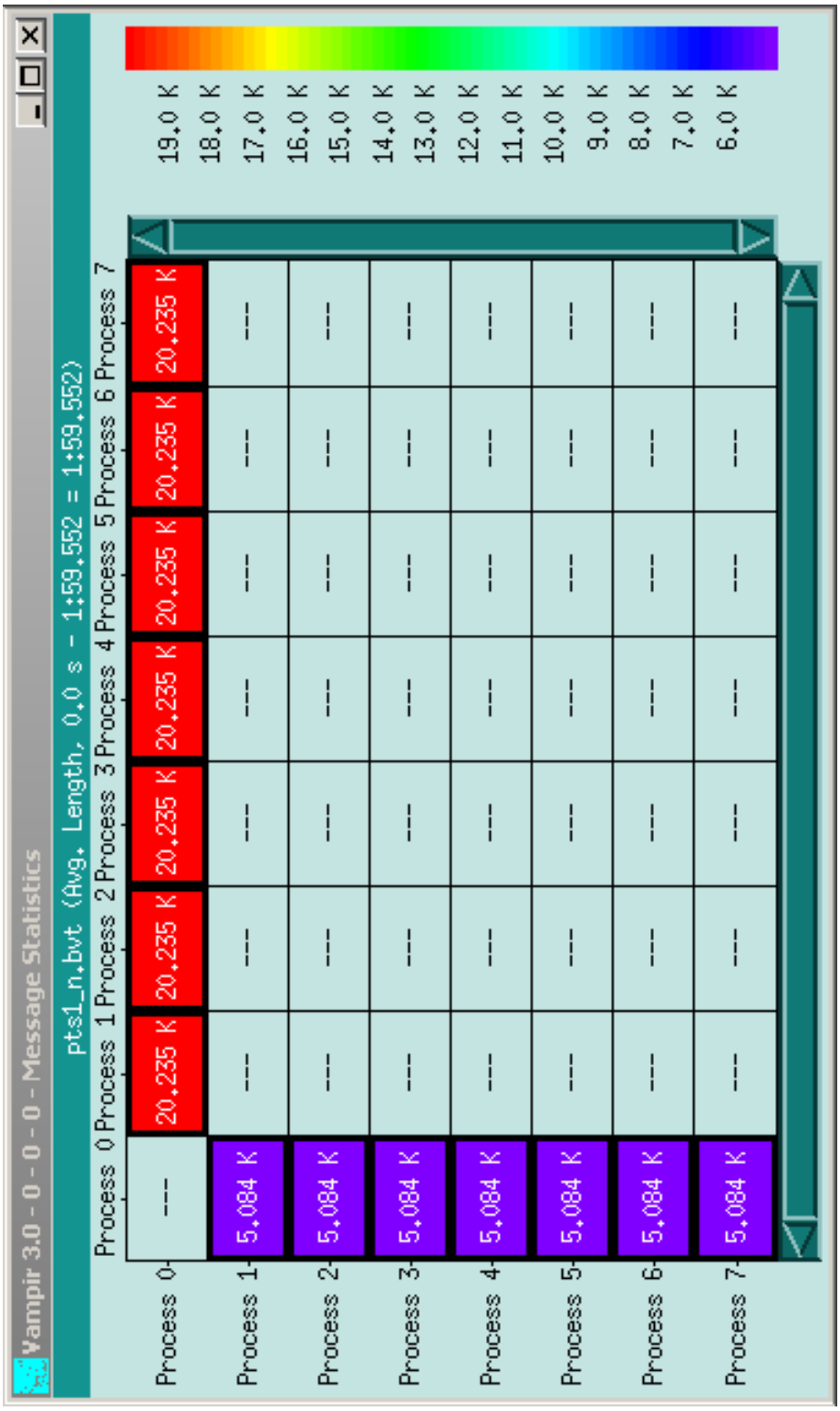


Figure 4.7: Summary of Average Message length Statistics: Algorithm - A.

### 4.3.2 Second Parallel Model - Algorithm B

As observed in the first parallel model, communication between master and the slaves is very frequent, due to synchronization after each iteration. Also, the communication is blocking, so, the processor has to stop computation to communicate.

Therefore, the second model is developed by applying the third and fourth steps of the design methodology, i.e., Agglomeration and Mapping. The basic idea is to reduce communication costs, by grouping tasks that communicate very frequently, thereby making it more course-grain.

In this second model, the control is vested among all the processors (p-control), such that all the slaves start from their own initial solution (MPSS). But they have to synchronize at fixed points (RS). Also, separate Tabu lists are maintained and Aspiration Criteria is applied. Instead of $NUM$ iterations, the total number of iterations are $NUM/r$, and for each $NUM/r$ iteration, there are $r$ local iterations in each slave. Figure 4.8 shows the flowchart of the algorithm. Therefore, this can be classified as p-control, RS, and MPSS.

Similar profiling is done to visualize the tracefile. The profiling was done on the same circuit for the same small amount of time as in the previous case.

Figure 4.9 shows the summary of time spent in application code as well as MPI routine. As it is seen, the time taken by MPI routines reduced from 19%

Figure 4.8: Flowchart for Parallel Tabu Search (p-RS-MPSS): Algorithm - B.

to approximately 8% (16.43 seconds out of 2 minutes) of the total time spent.

Figure 4.10 shows the timeline of the communication pattern. This shows that the communication is less frequent than the previous one.

Figure 4.11 shows the load on each of the processors. The load is almost equally balanced on all the slaves, with reduced load of MPI routines.

Finally the figure 4.12 shows the average length of the messages sent. Here, the average for sending to the slave is lesser than the previous one.

The results of Algorithm B of parallel tabu search algorithm for VLSI cell placement is presented in chapter 5.

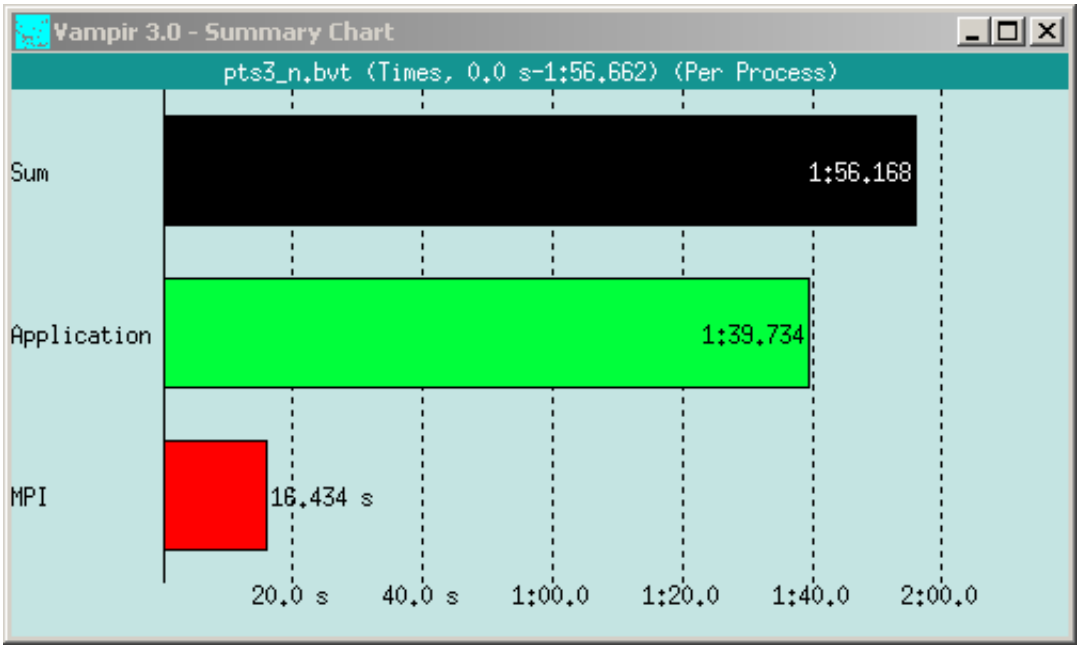


Figure 4.9: Summary of time spent in Application vs. MPI routines: Algorithm - B.

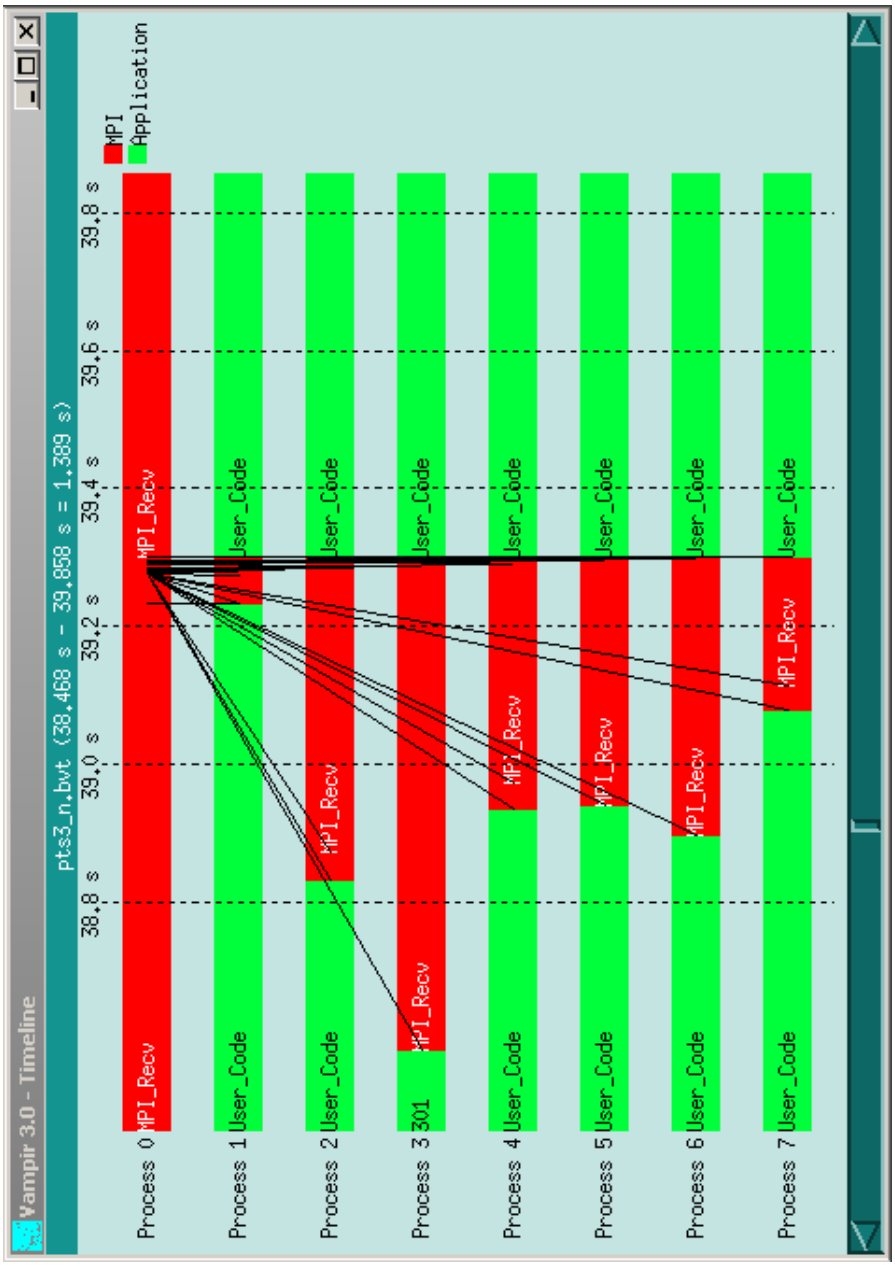


Figure 4.10: Timeline of Communication pattern: Algorithm - B.

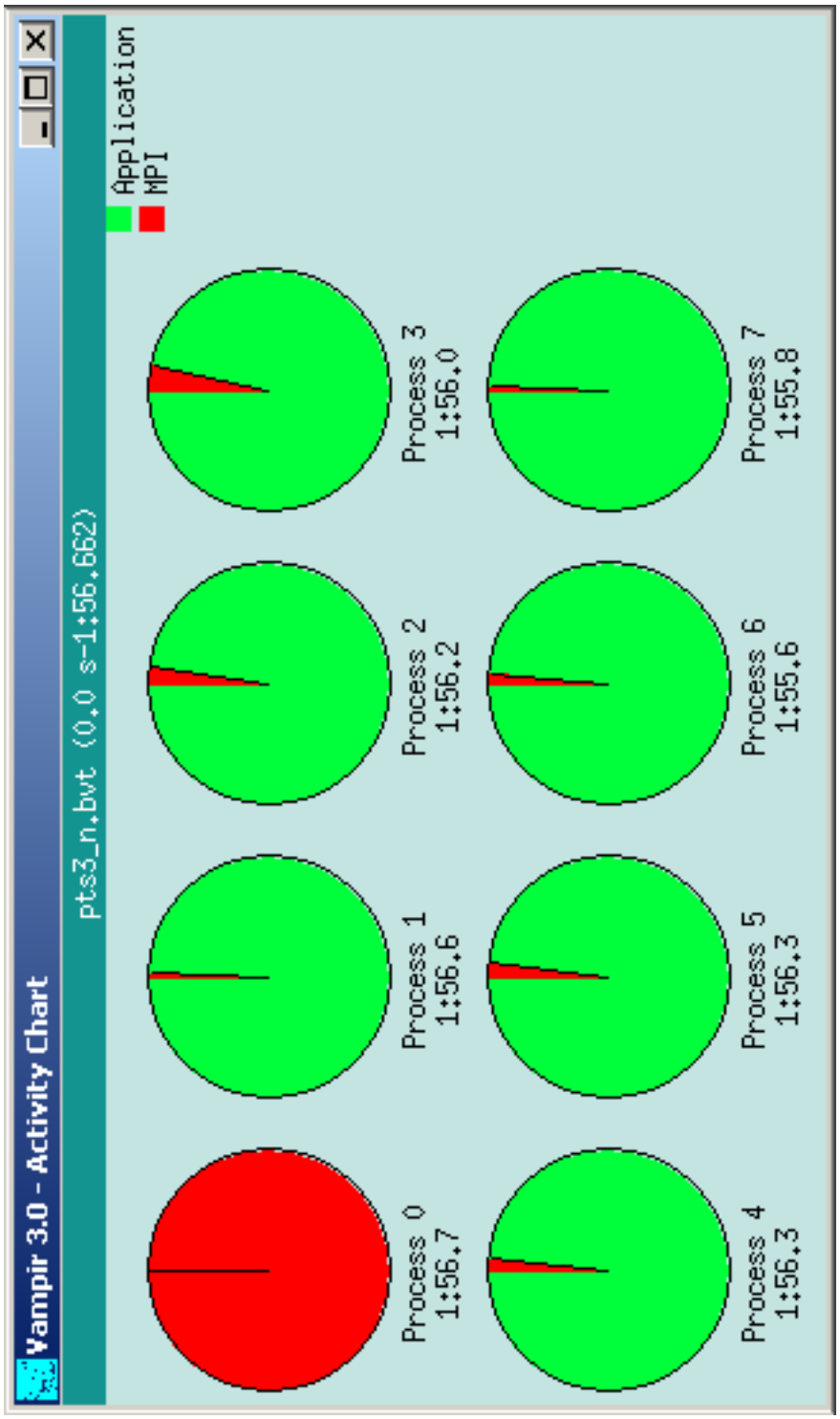


Figure 4.11: Load on each of the processors (Application vs. MPI): Algorithm - B.

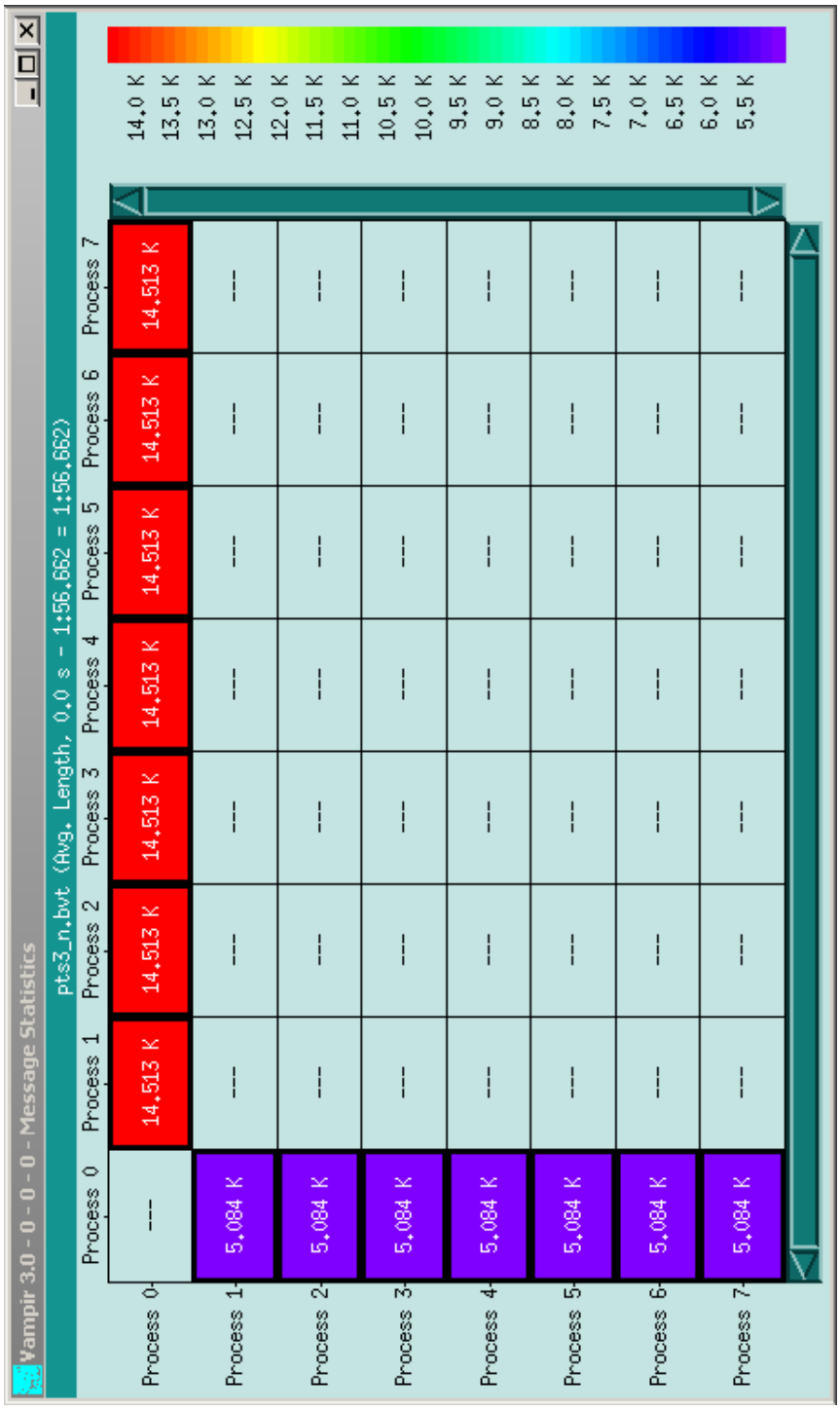


Figure 4.12: Summary of Average Message length Statistics: Algorithm - B.

# Chapter 5

# Experiments and Results

This chapter presents the experimental results of Tabu Search for VLSI Cell placement. Results obtained from sequential version are compared with parallel versions in the terms of the quality of solution as well as the execution time.

Speed-up in runtime is a ratio of

Regarding the speed-up obtained on multiple processors, it can be formulated using Amdahl's law.

## 5.1 Sequential TS Algorithm

The results of various circuits using a sequential Tabu Search on a single processor (a single machine from the cluster) are tabulated in Table 5.1. The same algorithm was executed in a PC environment on a single machine in [46]. 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 'WL', 'P' and 'D' are the wire length, power and delay costs respectively, whereas the aggregate fuzzy cost is denoted by ($\mu$) and 'T' is the execution time in seconds.

Table 5.1: Results for ISCAS'89 circuits for one processor

| Circuit | Gates | Paths | WL | P | D | $\mu$ | T |
|---|---|---|---|---|---|---|---|
| s298 | 136 | 150 | 4545 | 863 | 127 | 0.726 | 56 |
| s386 | 172 | 205 | 6520 | 1582 | 188 | 0.683 | 104 |
| s641 | 433 | 687 | 12176 | 2834 | 659 | 0.799 | 1865 |
| s832 | 310 | 240 | 17878 | 4016 | 355 | 0.651 | 160 |
| s953 | 440 | 583 | 25771 | 4041 | 202 | 0.670 | 391 |
| s1196 | 561 | 600 | 35690 | 10777 | 316 | 0.676 | 752 |
| s1238 | 540 | 661 | 38913 | 11473 | 358 | 0.639 | 755 |
| s1488 | 667 | 557 | 54786 | 13670 | 651 | 0.620 | 538 |
| s1494 | 661 | 558 | 52209 | 12880 | 565 | 0.631 | 541 |
| c3540 | 1753 | 668 | 168831 | 59724 | 695 | 0.693 | 3843 |
| s9234 | 5844 | 512 | 938313 | 170119 | 969 | 0.689 | 10717 |
| s15850 | 10470 | 512 | 2921966 | 228512 | 1831 | 0.683 | 20192 |

## 5.2 Algorithm - A: 1-control, RS, SPSS

Results for the $1^{st}$ parallel model is are tabulated. Table 5.2 gives the comparison with the sequential version in terms of best quality of solution (fitness) obtained, while the Table 5.3 illustrates comparison of their best runtimes for 5000 iterations.

Table 5.2: Fitness results of Algorithm - A on P processors

| | Fitness ($\mu$) | | | | |
|---|---|---|---|---|---|
| Circuit | P=1 | P=2 | P=4 | P=6 | P=8 |
| s298 | 0.726 | 0.747 | 0.756 | 0.804 | 0.793 |
| s386 | 0.683 | 0.686 | 0.709 | 0.672 | 0.654 |
| s641 | 0.799 | 0.773 | 0.744 | 0.785 | 0.798 |
| s832 | 0.651 | 0.633 | 0.646 | 0.649 | 0.634 |
| s953 | 0.671 | 0.650 | 0.691 | 0.685 | 0.675 |
| s1196 | 0.676 | 0.621 | 0.645 | 0.661 | 0.643 |
| s1238 | 0.639 | 0.621 | 0.619 | 0.639 | 0.620 |
| s1488 | 0.621 | 0.615 | 0.620 | 0.624 | 0.617 |
| s1494 | 0.631 | 0.617 | 0.608 | 0.617 | 0.607 |
| c3540 | 0.693 | 0.661 | 0.663 | 0.659 | 0.662 |
| s9234 | 0.707 | 0.625 | 0.625 | 0.627 | 0.628 |
| s15850 | 0.683 | 0.611 | 0.617 | 0.618 | 0.621 |

**Observations**

- The number of iterations (stopping criteria) for the $1^{st}$ parallel model were half of that of sequential one. As seen from the tables, the solution of the quality for the smaller circuits increased, but at the cost of execution time. As for medium sized circuits, the runtime was comparable to that of the sequential one, and the fitness was also in the similar range. However,

Table 5.3: Runtime results of Algorithm - A on P processors

| | Runtime (sec) | | | | |
|---|---|---|---|---|---|
| Circuit | P=1 | P=2 | P=4 | P=6 | P=8 |
| s298 | 56 | 73 | 151 | 225 | 301 |
| s386 | 104 | 91 | 166 | 243 | 320 |
| s641 | 1865 | 957 | 998 | 1096 | 1194 |
| s832 | 160 | 119 | 192 | 268 | 343 |
| s953 | 391 | 234 | 307 | 386 | 455 |
| s1196 | 752 | 425 | 493 | 566 | 644 |
| s1238 | 755 | 425 | 493 | 566 | 644 |
| s1488 | 538 | 304 | 384 | 466 | 543 |
| s1494 | 541 | 309 | 382 | 460 | 536 |
| c3540 | 3843 | 1941 | 1956 | 1989 | 2003 |
| s9234 | 10717 | 5464 | 5539 | 5612 | 5639 |
| s15850 | 20192 | 10093 | 10178 | 10263 | 10352 |

for larger circuits, since the computation was intense than the smaller circuits, the computation took over communication, and as a result, the runtime was almost half of the sequential one, but the compromise was on its quality.

– With the increase in the number of processors, the execution times and the quality of solutions obtained are very close to each other for almost all the circuits. This is because all the slave processors were generating $N$ neighbor solutions.

– The communication among the processors is blocking, i.e., the processor has to stop computation in order to communicate.

– It can thus be observed that this model of parallel Tabu Search has lot of

communication overhead. The communication among the processes has its own cost, and its frequency of occurrence in every iteration accounts for increase in the overall runtime (for smaller circuits) or decrease in the solution quality (for medium and large circuits).

## 5.3 Algorithm - B: p-control, RS, MPSS

The second parallel model is an improvement over the first model. Not only is the communication reduced by agglomeration, but in order to improve the quality of the solution faster, each slave generates its initial solution and iterates for $r$ local iterations, before synchronizing with the master process.

The results of this model are presented in two tables. While the first table (Table 5.4) gives the comparison of quality, the second table, Table 5.5 compares the runtimes with the sequential version.

**Observations:**

– As seen from the fitness table for the second model, (Table 5.4), the quality of solution increased as the number of processors increased in almost all the cases. The main difference from the earlier model was that, the communication overhead was greatly reduced, which gave more time for computation.

Table 5.4: Fitness results of Algorithm - B on P processors

| | Fitness ($\mu$) | | | | |
|---|---|---|---|---|---|
| Circuit | P=1 | P=2 | P=4 | P=6 | P=8 |
| s298 | 0.726 | 0.758 | 0.762 | 0.760 | 0.775 |
| s386 | 0.683 | 0.698 | 0.689 | 0.701 | 0.699 |
| s641 | 0.799 | 0.793 | 0.809 | 0.801 | 0.809 |
| s832 | 0.651 | 0.653 | 0.647 | 0.663 | 0.647 |
| s953 | 0.671 | 0.679 | 0.705 | 0.710 | 0.695 |
| s1196 | 0.676 | 0.656 | 0.671 | 0.668 | 0.684 |
| s1238 | 0.639 | 0.641 | 0.681 | 0.661 | 0.663 |
| s1488 | 0.621 | 0.631 | 0.633 | 0.632 | 0.641 |
| s1494 | 0.631 | 0.613 | 0.639 | 0.638 | 0.645 |
| c3540 | 0.693 | 0.702 | 0.706 | 0.695 | 0.709 |
| s9234 | 0.707 | 0.705 | 0.707 | 0.706 | 0.708 |
| s15850 | 0.683 | 0.689 | 0.688 | 0.698 | 0.692 |

Table 5.5: Runtime results of Algorithm - B on P processors

| | Runtime (sec) | | | | |
|---|---|---|---|---|---|
| Circuit | P=1 | P=2 | P=4 | P=6 | P=8 |
| s298 | 56 | 67 | 69 | 70 | 71 |
| s386 | 104 | 105 | 99 | 107 | 110 |
| s641 | 1865 | 1862 | 1841 | 1876 | 1900 |
| s832 | 160 | 159 | 159 | 162 | 163 |
| s953 | 391 | 389 | 384 | 392 | 392 |
| s1196 | 752 | 762 | 779 | 775 | 769 |
| s1238 | 755 | 750 | 745 | 751 | 752 |
| s1488 | 538 | 539 | 547 | 540 | 542 |
| s1494 | 541 | 548 | 540 | 542 | 539 |
| c3540 | 3843 | 3973 | 3966 | 3953 | 3942 |
| s9234 | 10717 | 10571 | 10652 | 10705 | 10840 |
| s15850 | 20192 | 20270 | 20332 | 20398 | 20469 |

- The runtime in almost all the cases were either similar or a little more than the sequential one. The reason for this is that the neighborhood size $N$ remained fixed on each processor. Therefore, larger search space was traversed on different slave processors, plus some extra time required to communicate.
- It can thus be observed that, with more resources, better quality of solutions was obtained in similar runtime as of the sequential one. However, effort had to be done to reduce the communication, but decreasing its frequency.

## 5.4 Modified Algorithm - B

In the previous two models, one of the aforementioned targets was obtained: namely, getting better solution in similar amount of time. This is because of more resources available, henceforth larger search space was traversed. Communication was a bottleneck in the first model, which was reduced in the second model.

In order to achieve our second objective, that is, similar quality of solution in lesser time, the second model is modified. The modification is as follows:

Each slave process has to generate '$N/(p-1)$' neighbors instead of $N$ neighbors (where, 'p' is the number of processors). As the number of slaves processors

increases, each processor has lesser number of neighboring solutions to work with.

However, there is a trade-off in this case. As the neighborhood size is divided among the slave processors, the number of neighbor solutions generated at each slave is reduced with the increasing number of processors, and the quality of the solution decreases. The reason for this degradation is that each slave has a lower number of possible moves to make and hence the search pool is reduced. It is observed that the percentage of quality degradation of almost 15% (worst case) as compared to that obtained by sequential Tabu Search strategy.

The results obtained for the solution quality from this model are tabulated in Table 5.6.

Table 5.7 shows the execution times as well as the corresponding speed-ups for 2, 4, and 6 processors. As can be seen, there is almost linear speed-up in most of the cases with minor variations. A significant observation is that in case of larger circuits, excellent results are obtained in terms of scalability and speed-up. For instance, in case of s15850 having 10470 gates, the execution time is reduced from 20,192 seconds down to 3,441 seconds when using 6 processors resulting in speed-up of 6.03.

Table 5.6: Fitness results of modified Algorithm - B on P processors and degradation in quality with respect to that obtained on P=1

| | Fitness ($\mu$) | | | |
|---|---|---|---|---|
| Circuit | P=1 | P=2 | P=4 | P=6 |
| s298 | 0.726 | 0.788 (-8.5%) | 0.744 (-2.4%) | 0.660 (11.3%) |
| s386 | 0.683 | 0.742 (-8.6%) | 0.642 (6%) | 0.584 (14.5%) |
| s641 | 0.799 | 0.739 (7.5%) | 0.697 (12.7%) | 0.676 (15.3%) |
| s832 | 0.651 | 0.648 (0.4%) | 0.577 (11.4%) | 0.559 (14.1%) |
| s953 | 0.671 | 0.661 (1.5%) | 0.596 (11.2%) | 0.571 (14.9%) |
| s1196 | 0.676 | 0.639 (5.4%) | 0.601 (11.1%) | 0.579 (14.3%) |
| s1238 | 0.639 | 0.625 (2.2%) | 0.583 (8.7%) | 0.541 (15.3%) |
| s1488 | 0.621 | 0.584 (5.9%) | 0.549 (11.5%) | 0.527 (15.1%) |
| s1494 | 0.631 | 0.620 (1.7%) | 0.583 (7.6%) | 0.541 (14.2%) |
| c3540 | 0.693 | 0.685 (1.2%) | 0.636 (8.2%) | 0.584 (15.7%) |
| s9234 | 0.707 | 0.679 (3.9%) | 0.632 (10.6%) | 0.607 (14.1%) |
| s15850 | 0.683 | 0.667 (2.3%) | 0.631 (7.6%) | 0.608 (11.1%) |

Table 5.7: Runtimes and (speed-ups) of modified Algorithm - B on P processors.

| | Runtime | | | |
|---|---|---|---|---|
| Circuit | P=1 | P=2 | P=4 | P=6 |
| s298 | 56 | 34 (1.65) | 19 (2.95) | 15 (3.73) |
| s386 | 104 | 56 (1.86) | 29 (3.58) | 21 (5.47) |
| s641 | 1865 | 962 (1.94) | 503 (3.71) | 308 (6.05) |
| s832 | 160 | 87 (1.83) | 44 (3.64) | 26 (6.15)) |
| s953 | 391 | 201 (1.95) | 104 (3.76) | 62 (6.30)) |
| s1196 | 752 | 401 (1.87) | 200 (3.76) | 130 (5.78) |
| s1238 | 755 | 390 (1.93) | 199 (3.79) | 123 (6.14) |
| s1488 | 538 | 283 (1.90) | 144 (3.73) | 93 (5.78) |
| s1494 | 541 | 287 (1.88) | 144 (3.75) | 93 (5.81) |
| c3540 | 3843 | 2020 (1.90) | 986 (3.89) | 621 (6.18) |
| s9234 | 10717 | 5547 (1.93) | 2759 (3.88) | 1777 (6.03) |
| s15850 | 20192 | 10621 (1.90) | 5294 (3.81) | 3441 (6.07) |

# Chapter 6

# Conclusion & Future Work

## 6.1 Conclusion

In this thesis, a methodology was followed to design a parallel algorithms for Tabu Search for VLSI Cell placement. The taxonomy that was proposed by Crainic et. al [19], is sufficiently comprehensive to account for parallelization strategies applied to different problems.

Two different algorithms for Parallel Tabu Search for VLSI Cell Placement were presented: Algorithm - A that can be classified as 1-RS-SPSS and Algorithm - B, which is P-RS-MPSS. The third one was modified version of Algorithm - B.

The use of parallelism improves the performance of Tabu Search for VLSI

Cell placement. However, certain crucial parameters to be considered are time spent in communication, stopping criteria, neighborhood size (TS parameter), number of iteration, and synchronization frequency.

## 6.2 Future Work

For the future work, other parallel strategies can be implemented, especially, the asynchronous ones, to improve communication costs over synchronous strategies. Variations in asynchronous version to fine tune various parameters has also to be taken into considerations. Intensification and diversification strategies can be implemented to improve the quality of solution. Comparison with other iterative heuristics for performance and quality of solution can be done for comparison purposes. These experiments can be be executed on a larger cluster with more machines, as there is a tendency to improve more.

# Appendix A

# MPI (Message Passing Interface)

Message Passing Interface (MPI), the de facto message-passing standard used in industry as well as academia.

In the message-passing library approach to parallel programming, a collection of processes executes programs written in a standard sequential language augmented with calls to a library of functions for sending and receiving messages.

In the MPI programming model, a computation comprises one or more processes that communicate by calling library routines to send and receive messages to other processes. In most MPI implementations, a fixed set of processes is created at program initialization, and one process is created per processor.

However, these processes may execute different programs. Hence, the MPI programming model is sometimes referred to as multiple program multiple data (MPMD) to distinguish it from the SPMD model in which every processor executes the same program.

Processes can use point-to-point communication operations to send a message from one named process to another; these operations can be used to implement local and unstructured communications. A group of processes can call collective communication operations to perform commonly used global operations such as summation and broadcast. MPI's ability to probe for messages supports asynchronous communication. Probably MPI's most important feature from a software engineering viewpoint is its support for modular programming. A mechanism called a communicator allows the MPI programmer to define modules that encapsulate internal communication structures.

To summarize, the principal features of the message-passing programming model as realized in MPI are as follows [12].

- A computation consists of a (typically fixed) set of heavyweight processes, each with a unique identifier (integers 0..P–1).
- Processes interact by exchanging typed messages, by engaging in collective communication operations, or by probing for pending messages.
- Modularity is supported via communicators, which allow subprograms to

encapsulate communication operations and to be combined in sequential and parallel compositions.

- Algorithms that do create tasks dynamically or place multiple tasks on a processor can require substantial refinement before they can be implemented in MPI.
- Determinism is not guaranteed but can be achieved with careful programming.

# Bibliography

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

[2] Sadiq M. Sait and Habib Youssef. *VLSI Physical Design Automation: Theory and Practice.* Lecture Notes Series on Computing - Vol. 6. World Scientific, Singapore, 2001.

[3] K. Shahookar and P. Mazumder. VLSI cell placement techniques. *ACM Computing Surveys*, 23(2):143–220, June 1991.

[4] Hirendun Vaishnav and Massoud Pedram. PCUBE: A performance driven placement algorithm for low power designs. *IEEE, Design Automation Conference, with Euro-VHDL*, 1993.

[5] A. Srinivasan, K. Chaudhary, and E. S. Kuh. Ritual: A performance-driven placement algorithm. *IEEE Transactions on Circuits and Systems - II*, 39(11):825–840, November 1992.

[6] S. Sutanthavibul, E. Shragowitz, and Rung-Bin Lin. An adaptive timing-driven placement for high performance VLSI's. *IEEE Transactions on Computer Aided Design*, 12(10):1488–1498, October 1993.

[7] Massoud Pedram. Logical-Physical Co-design for Deep Submicron Circuits: Challenges and Solutions. *Proceedings of Design Automation Conference, the ASP-DAC'98. Asia and South Pacific*, pages 137–142, February 1998.

[8] S. Kirkpatrick, Jr. C. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science*, pages 498–516, May 1983.

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

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

[11] Van-Dat Cung, Simone L. Martins, Celso C. Riberio, and Catherine Roucairol. Strategies for the Parallel Implementation of metaheuristics. *Essays and Surveys in Metaheuristics*, pages 263–308, Kluwer 2001.

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

[13] F. Glover, E. Taillard, and D. de Werra. A user's guide to tabu search. *Annals of Operations Research*, 41:3–28, 1993.

[14] R. Hübscher and F. Glover. Applying tabu search with influential diversification to multiprocessor scheduling. *Computers & Operations Research*, 21(8):877–884, 1994.

[15] F. Glover. Tabu search: A tutorial. *Technical Report, University of Colorado, Boulder*, February 1990.

[16] A. Casotto, F. Romeo, and A. L. Sangiovanni-Vincentelli. A parallel simulated annealing algorithm for the placement of macro-cells. *IEEE Transactions on Computer-Aided Design*, CAD-6(5):838–847, September 1987.

[17] P. Banerjee and M. Jones. A parallel simulated annealing algorithm for standard-cell placement on a hypercube computer. *Proceedings of International Conference on Computer-Aided Design, ICCAD-86*, 1986.

[18] M. Toulouse, T.G. Crainic, and M. Gendreau. Issues in Designing Parallel and Distributed search Algorithms for Discrete Optimization Problems. *Publication CRT-96-36, Centre de recherche sur les transports, Université de Montréal, Montréal, Canada*, 1996.

[19] T.G. Crainic, M. Toulouse, and M. Gendreau. Towards a taxonomy of parallel tabu search heuristics. *INFORMS Journal of Computing*, 9(1):61–72, 1997.

[20] M. Malek, M. Guruswamy, M. Pandya, and H. Owens. Serial and parallel simulated annealing and tabu search algorithm for the travelling salesman problem. *Annals of Operation Research*, 21:59–84, 1989.

[21] E. Taillard. Some efficient heuristic methods for the flow shop sequencing problem. *European Journal of Operational Research*, 417:65–74, 1990.

[22] R. Battiti and G. Tecchiolli. Parallel biased search for combinatorial optimization: Genetic algorithms and tabu. *Microprocessors and Microsystems*, 16(351-367), 1992.

[23] E. Taillard. Parallel iterative search methods for the vehicle routing problem. *Networks*, 23:661–673, 1993.

[24] J. Chakrapani and Skorin-Kapov. Massively parallel tabu search for quadratic assignment problem. *Annals of Operation Research*, 41(327-341), 1993.

[25] C.N. Fiechter. A parallel tabu search algorithm for large travelling salesman problems. *Discrete Applied Mathematics*, 51(243-267), 1994.

[26] Bruno-Laurent Garica, Jean-Yves Potvin, and Jean-Marc Rousseau. A parallel implementation of the tabu search heuristic for vehicle routing problems with time window constraints. *Computers & Operations Research*, 21(9):1025–1033, November 1994.

[27] I. De Falco, R. Del Balio, E. Tarantino, and R. Vaccaro. Improving search by incorporating evolution principles in parallel tabu search. In *Proc. of the first IEEE Conference on Evolutionary Computation- ICEC'94*, pages 823–828, June 1994.

[28] S. Nair and A. Freville. A parallel tabu search algorithm for the 0-1 multidimentional knapsack problem. *11th International Parallel Processing Symposium*, April 1997.

[29] Taillard. Parallel tabu search techniques for the job sequencing problem. *ORSA: Journal on Computing*, 6(2)(108-117), 1994.

[30] H. Mori and T. Hayashim. New parallel tabu search for voltage and reactive power control in power systems. *In Proc. of the 1998 IEEE International Symposium on Circuits and Systems - ISCAS'98*, pages 431–434, May 1998.

[31] Ahmad Al-Yamani, Sadiq M. Sait, Habib Youssef, and Hassan Barada. Parallelizing tabu search on a cluster of heterogenous workstations. *Journal of Metaheuristics*, 8:277–304, May 2002.

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

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

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

[35] 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.

[36] 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.

[37] 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.

[38] Mahmood R. Minhas. Iterative algorithms for timing and low-power driven vlsi standard cell placement. Masters thesis, KFUPM, Dhahran, Saudi Arabia, July 2001.

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

[40] 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.

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

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

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

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

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

[46] Sadiq M. Sait, Mahmood R. Minhas, and Junaid A. Khan. Performance and low-power driven VLSI standard cell placement using tabu search. *Proceedings of the 2002 Congress on Evolutionary Computation, 2002. CEC '02.*, 1:372–377, May 2002, Honolulu, HI, USA.

# Vitae

- Syed Sanaullah
- Received B.S. degree in Computer Engineering from Marmara University, Istanbul, Turkey in 1999
- Completed M.S. degree requirements at KFUPM, Saudi Arabia in 2003