1-1-2012

Parallel Implementation of Non-slicing Floorplans with MPI and OpenMP

Oluvaseun Owojaiye
Ryerson University

PARALLEL IMPLEMENTATION OF NON- SLICING FLOORPLANS WITH MPI AND OPENMP

By

Oluwaseun Owojaiye
BSc. Computer Science, University of Ilorin Nigeria,
August 2012

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Applied Science

in the Program of

Electrical and Computer Engineering

Toronto, Ontario, Canada, 2012

Parallel Implementation of Non-slicing Floorplans with MPI and OpenMP
Master of Applied Science 2012
Oluwaseun Owojaiye
Electrical and Computer Engineering
Ryerson University

ABSTRACT

Advancement in technology has brought considerable improvement to processor design and now manufacturers design multiple processors on a single chip. Supercomputers today consists of cluster of interconnected nodes that collaborate together to solve complex and advanced computation problems. Message Passing Interface and Open Multiprocessing are the popularly used programming models to optimize sequential codes by parallelizing them on the different multiprocessor architecture that exist today. In this thesis, we parallelize the non-slicing floorplan algorithm based on Multilevel Floorplanning/placement of large scale modules using B*tree (MB*tree) with MPI and OpenMP on distributed and shared memory architectures respectively. In VLSI (Very Large Scale Integration) design automation, floorplanning is an initial and vital task performed in the early design stage. Experimental results using MCNC benchmark circuits show that our parallel algorithm produced better results than the corresponding sequential algorithm; we were able to speed up the algorithm up to 4 times, hence reducing computation time and maintaining floorplan solution quality. On the other hand, we compared both parallel versions; and the OpenMP results gave slightly better than the corresponding MPI results.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest appreciation to my supervisor, Dr. Nagi Mekhiel for his guidance and support from the initial to the final phase of my MASc program. I am thankful for his adequate supervision that has given me a solid footing in my research area.

I would like to thank my family members, especially my husband and kids for supporting and encouraging me to pursue this degree.

Lastly, I thank my parents and siblings for their prayers and encouragement throughout my program.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

1. INTRODUCTION

1.1. Motivation

Since the past few years, there has been a significant shift towards parallel computing of commercial applications as the demand for faster computing system has increased considerably. As processor manufacturers tend to develop multi-core processors rather than improve single core processor speeds, parallelization has become increasingly popular [20]. Multiple processors on a parallel computing system work in collaboration to solve advanced computation problems faster and more efficiently [24]. Open Multiprocessing (OpenMP) and Message Passing Interface (MPI) are the two major parallel programming models used on parallel architectures. Both of the paradigms aim to provide an interface for high performance, but their approach is somewhat different. Open MP can be used on a multicore architecture with shared memory and multiple cores exist on a single chip. There exists multicore chips today that have over 64 cores on a single chip and manufacturers like Tilera are increasing this number to 100. This would help to conserve power and also the distance between processors will be highly reduced. MPI can be used on a shared memory as well as distributed memory architecture. In distributed memory architecture, each processor has its own private memory and multiple processors communicate through a communication network. VLSI Floor planning applications can benefit from parallelism because they are time consuming and computationally intensive. The building of electronic components with a very large number of transistors is referred to as VLSI. Defining block shapes and positions with the aim of minimizing dead space and wirelength can be referred to as floor planning

[3]. Dead space can be defined as the amount of unused space contained in between module blocks arranged on a rectangular space. The problem of floor planning is to arrange a set of module blocks on a rectangular area space in an optimal manner, so that none of these blocks extend over another and with minimal amount of free unused space called dead space. Floorplanning is a very critical task in VLSI design and can be computationally intensive. Its purpose is to use the minimal space possible as well as minimal wire for connection. The floorplanning algorithms based on MB*tree [4] has been implemented for two different parallel models: MPI (Message Passing Interface) and OpenMP (Open Multiprocessing) with the aim to speedup computation, as the computational time for the algorithm to run on large circuits could be enormous. Floorplanning in VLSI could be slicing or non-slicing [3, 6, 10, 12]. Our parallel algorithm is based on the work on floorplanning in [4] using B*-trees. To find the optimal orientation, the sequential algorithm is divided into two stages: clustering (packing) and declustering (perturbation) [2, 4]. In this thesis, we deal with non-slicing structures. All the possible orientation is considered and the optimal arrangement that gives the minimal dead space and wirelength is selected. The first phase of the algorithm is clustering which involves packing or grouping of module blocks with the avoidance of overlapping and wirelength computation; in the declustering stage these arrangements are refined to an optimal point using the Simulated Annealing (SA) approach [1, 5, 8, 11, 13]. It is used for finding optimal solution to many-dimensional problems [5, 10]. Wirelength calculation in this algorithm consumes so much computational time because each time the tree is refined, wirelength computation takes place.

1.1.1. Thesis Goal

In this thesis, we explore the parallelization of the MB*tree sequential algorithm. Our approach parallelizes one of the most computationally intensive phase in the algorithm. The main idea of our parallel approach is to use the master-slave approach for parallelization taking advantage of the technology of distributed and shared memory systems. In [4], the algorithm describes the clustering stage where a set of module blocks are grouped into clusters to form one module containing all the blocks and in the same way these clusters are perturbed for optimal solution. Our parallel approach uses the master node to carry out the perturbation task and in the packing stage both the master and the slave nodes work in collaboration to execute the wirelength computation after area update has been completed. In the perturbation phase, simulated annealing is employed to refine the tree constructed during the packing phase. The master processor/thread distributes the total modules by the number of processor modules to each slave node. Each slave process accepts the modules distributed by the master node and then computes the wirelength of its own share of the modules. Each processor forwards its wirelength result to the master; the master then sums the results up, finds the aggregate and computes the final wirelength. Our parallel approach was implemented using two different parallel models: MPI and OpenMP. Results of both models were compared with each other and also with the sequential version and results obtained show decreased runtime as the number of nodes increases and speedup was achieved. The Microelectronics Center of North Carolina (MCNC) benchmark circuit [4, 37] was used for testing. The MCNC benchmark suite is popularly used in academic research. It is a collection of standardized libraries of circuit designs ranging from simple to advanced circuits collected from industries and academia. We obtained the net list of these circuits which contains the module information that make up the circuits and the interconnections between the modules.

1.1.2. Organization of Thesis

Chapter 1 contains the introduction.

Chapter 2 presents some background theories and related work for this thesis. A short introduction to Parallel programming with OpenMP and MPI is given, along with some parallel programming models and performance modeling information. A brief description of multiprocessors and computer architectures are also presented.

Chapter 3 describes the sequential MB*tree Algorithm Floorplanning algorithm.

Chapter 4 describes the Parallelization of MB*Tree using MPI and OpenMP models. This chapter describes the methods employed to achieve parallelism.

Chapter 5 describes the experimental methodology used in this thesis. The installation and configuration of MPI and OpenMP is described. The computer and network information used in this thesis is also presented.

Chapter 6 presents the detailed analysis of performance results from our tests for both the Parallel algorithm; comparison of the sequential and parallel versions is discussed. Execution time and overhead incurred is shown in various charts and discussed. The speed achieved for each test is also presented.

Chapter 7 contains the conclusion and future work in this thesis area.

CHAPTER 2

2. BACKGROUND

In this chapter, an overview of the theory and technology used in this project is presented.

2.1. Why Parallelization

In the past few years, there has been a significant shift towards parallel computing of commercial applications as the demand for computing power has increased considerably. As processor manufacturers tend to develop multi-core processors rather than improve single core processor speeds, parallelization has become increasingly popular [20]. So many large applications with serial programs take an enormous amount of execution time. Running computationally intensive applications sequentially is very time consuming and today's supercomputers have the capability to use multiple cores or multicomputer to run programs in order to improve performance, efficiency and considerably reduce runtime. Running a program on a single core can be optimized by parallelization. The objective is to reduce the execution time. Many applications have gained from parallelization because multiple processes are executing the code. So the goal is to use multiple cores or computers to reduce the execution time. Performance is one of the major factors to be considered in parallel processing.

2.2. Parallelization explained

What is parallelization?

Parallel computing can simply be defined as a computing approach in which multiple cores or processors are used for execution of a single program [20, 24]. In parallel processing, tasks of a sequential program is identified and split amongst multiple processes to execute independently and simultaneously. The work is distributed over multiple computers or cores depending on the architecture. There are various programming models that exists today for parallel processing, they include: pthreads, Message Passing interface, OpenMP e.t.c. These models make use of library of functions for parallel processing [19, 20, 22, 24]. MPI is designed for distributed memory and is probably the best known paradigm in parallel computing. In MPI, communication between processes is well defined and it also provides a good number of functions that opens up for high performance and improvement of the system [20]

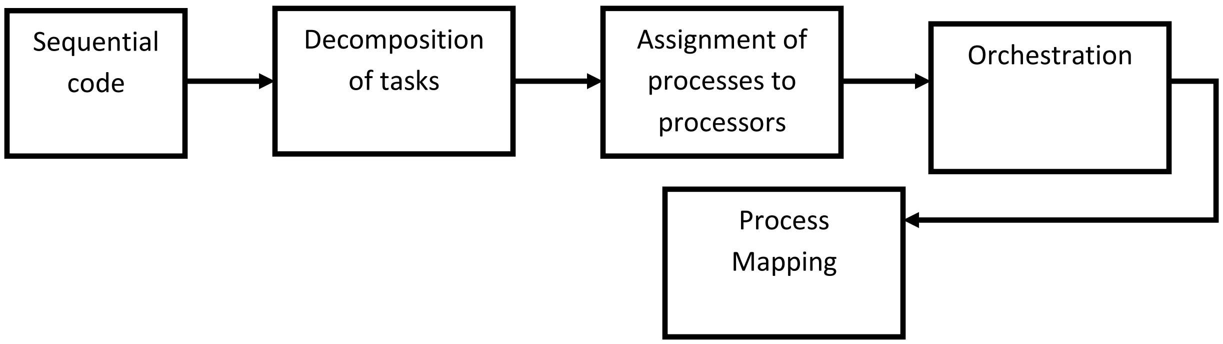


Figure 1: Parallelization tasks

Figure 1 above shows the steps involved in parallelization of a serial application. The following steps are involved in converting a serial program to parallel program:

1. *Task Decomposition*: This involves the breaking down a problem into chunks of tasks to be assigned to multiple processors.
2. *Task Assignment*: Here the decomposed tasks are assigned to the processes that would perform the computation simultaneously.

3. *Orchestration*: This involves the arrangement, control and coordination of processes for necessary data access, communication and synchronization among processes [24].

4. *Mapping*: Mapping involves binding of processes to processors [24].

Benefits of Parallelism: Major aim and benefit of parallelism is to achieve speedup by reducing the total execution time of a program. Also with parallelization, a program can make use of the memory of multiple separate computers at once [23].

Challenges of Parallelism: Interprocessor communication is one of the major challenges in parallel programming. Each processor or core might have to communicate, depending on the algorithm and this brings about overheads that increases the total execution time for some applications and reduces performance and efficiency.

2.3. High Performance Computing (HPC)

2.3.1. Multiprocessors

While some computational problems can be solved on a machine with one processor or core, so many other complex problems require computers comprising of more than one processor, this types of computers are referred to as parallel computers. Due to the high demand for higher performance and more efficient computing machines, processor manufacturers are focusing more on multicore processors. Cluster computers and supercomputers are used today to solve complex computational problems. Different kinds of architectures have been introduced to overcome the limitation of single processor machines; these include vector processors, interconnected workstations, shared memory and distributed memory processors [34].

The processors in a parallel machine collaborate to solve complex and advanced computation problems which save time. These are the most commonly used models in today's world. Many grand challenge applications have benefited from parallel computing, they include: Computational Fluid Dynamics, Climate modeling, Computational Biology, Human Genome problem, e.t.c. Parallel machines can either be shared memory or distributed memory machines.

2.3.1.1. **The Shared Memory Architecture**

In a multiprocessor system, shared memory architecture contains only a single address space with two or more processors accessing the shared memory through a bus. One of the benefits of this model is that communication is done using one shared memory and there is quick access to data [34]. One of the limitations of shared memory model is that there is memory access contention because only one address space exists and this might lead to increased memory access time and poor performance. Access contention can also lead to bottlenecks and generally shared memory computers do not scale well [20, 34]. Another drawback of shared memory architecture is cache coherence; changes of data in cache need to be updated on all other processors that require this information for data consistency; and this has to be done by the system. Figure 2 shows shared memory architecture. OpenMP programming model can be used on shared memory architecture [20, 24]. To solve the problem of access contention of shared memory architecture in Figure 2, each processor can have its own local memory as shown in Figure 3, to store unshared codes and this reduces the access contention time.

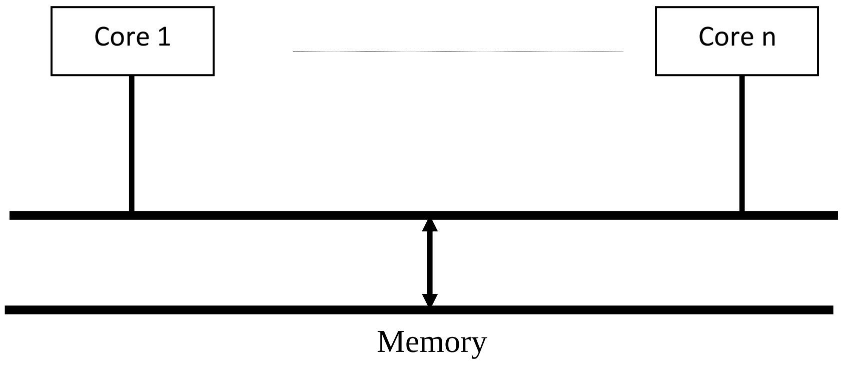


Figure 2: Shared memory architecture

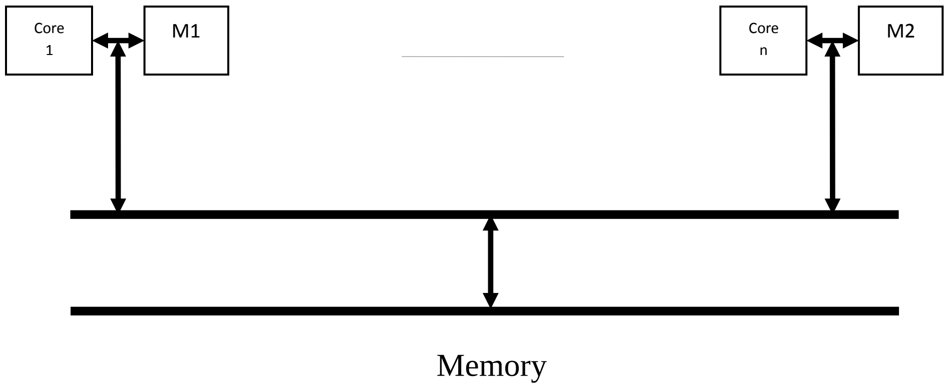


Figure 3: Shared memory architecture with each processor having its own local memory

2.3.1.2. Distributed Memory Architecture

In distributed memory architecture, communication exists between multiple computers or nodes on the network. Each node consists of its own private memory and does not have access to the memory of other nodes in the network but can communicate with them. In this model, data exchange is done by the various nodes in the network. Interprocessor communication is done by message passing via the network. One of the benefits of this model is that it allows for scalability e.g. a network of interconnected work station is distributed memory architecture and therefore it can allow for more machines to be connected to the network and this has led to an increasingly growing use of computer clusters to implement parallel applications [34].

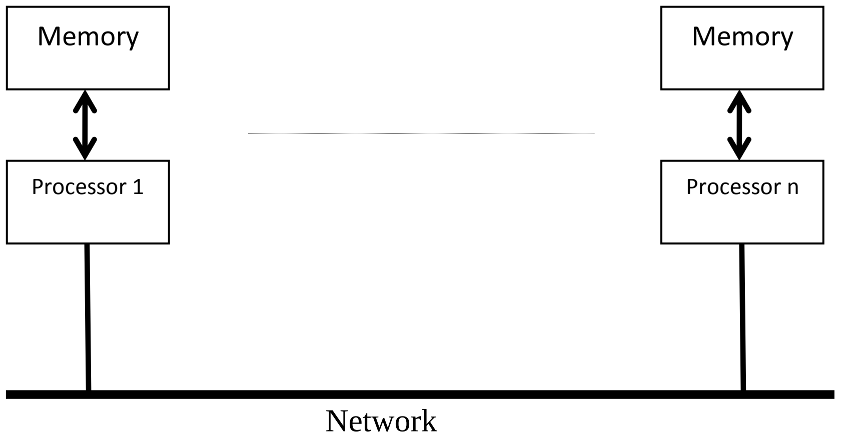


Figure 4: Distributed Memory Architecture

2.2.2. Parallel Programming Models

There exist many programming tools to implement parallel programs. Some of the most popularly used tools are OpenMP and MPI, Parallel Virtual Machines (PVM) is also a commonly used message passing model for distributed parallel programming.

2.2.2.1. OpenMP Overview

OpenMP is a multithreading process [19, 20, 21] and it is directive based. In this approach, directives are used in parts of the code that needs to be parallelized; this is defined by the programmer. These directives are used to parallelize sections of the code defined by the user, task partitioning and synchronization.

“#pragma omp parallel for” is used in C or C++ to parallelize a for loop. Tasks are divided into chunks using the pragma some of the basic OpenMP routines are:

- # pragma omp parallel for - indicates a loop that can be executed in parallel. The for loop’s iterations can be divided among the number of threads and the distribution method is dependent upon the scheduling strategies [25].
- omp_get_num_threads () - It returns the number of threads in the parallel region.
- omp_set_num_threads() - is used to change the number of threads during execution
- omp_num_threads () – is used to define the number of threads to be used in execution.

  OpenMP code is an extension of the sequential code i.e. the OpenMP directives used between the codes provides a parallelization method which is portable for implementation.

2.2.2.1.1. Implementation of OpenMP

Fork-Join approach is employed in executing a sequential code with OpenMP [20, 26]. When a directive is added into an OpenMP code at runtime, the code initially starts with a sequential region execution and then when the OpenMP directives are encountered, additional threads are spawned and the work is partitioned and assigned to the threads. In the parallel region, each thread carries out its own share of the task concurrently with other threads. # pragma omp parallel directive is included in an OpenMP code to define parallel regions in C or C++ programming language.

In the diagram below, the master thread assigns workload to itself just like other worker threads it creates and is involved in computation and execution of data and tasks respectively. The omp_barrier directive is added to the end of the parallel region to allow all other threads to wait until all threads have completed execution of their tasks before synchronization and then a join is created by the master thread to synchronize all other threads.

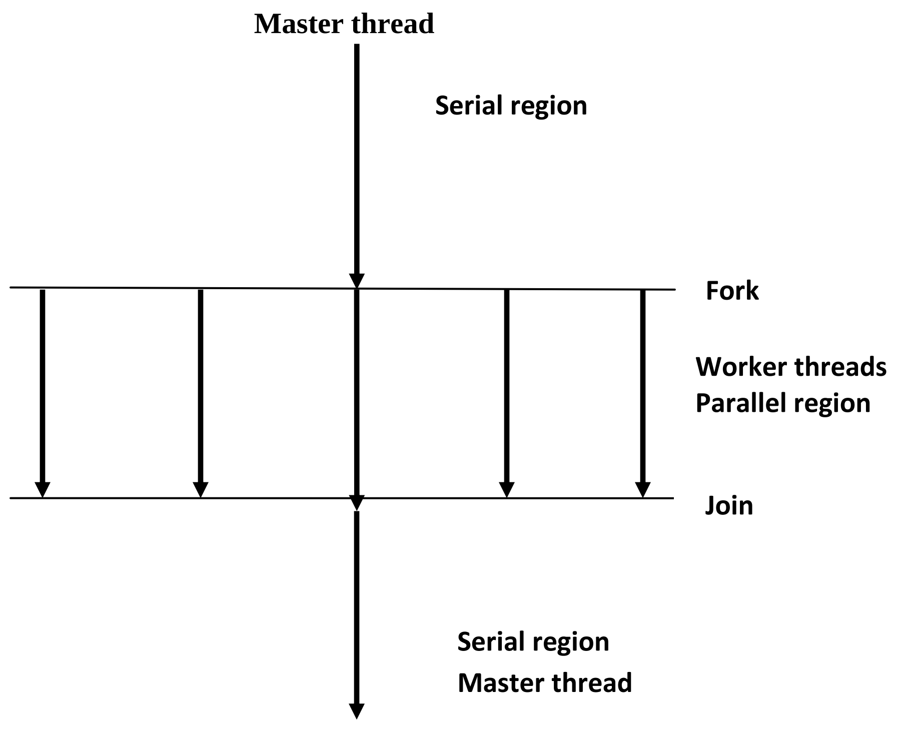


Figure 5: OpenMP Fork and Join approach

2.2.2.1.2. OpenMP Memory Models

- Private Memory: in private memory, each worker thread can only access its own private memory. It makes changes to its own memory and can only access and modifies it.
- Shared Memory: In shared memory, each thread has access to data in the memory; and each thread has the right to make modifications to the shared variable. The threads use a mechanism to read and write to the shared memory.

2.2.2.2. Parallel Virtual Machine (PVM)

PVM is based on message passing that can be used in distributed environment as well as multiprocessor environment. In PVM, a group of computers appear as one single virtual machine; these computers are able to communicate with each other using an interface. PVM works well with data parallel approach in which the same instruction is executed on each share of the data on each processor. However, PVM was not standardized and was not as popular as the Message Passing Interface [20].

2.2.2.3. Message Passing Interface (MPI)

MPI is one of the most popular models for programming a cluster of computers. MPI provides an Application Programming Interface (API) through the set of functions that also handle input/output operations. It uses a distributed memory model and can only access data from its own local memory. MPI standard consists of library of specifications used for point to point communication. The objective of MPI is to establish a communication or message passing standard for message passing coding. Message Passing Interface is known to be portable, flexible and efficient for writing MPI codes on high performance computers [20]. MPI can be used on any distributed memory or shared memory. In MPI, # include “mpi.h” is required for all coding. It is used to make a library call.

The following are the basic routines used in MPI programs.

- MPI_Init: this is used to initialize the environment for code execution. This function is required in every program and should be used before all other MPI functions.

- MPI_Send: This function is used for sending messages. Processors have access only to its own local memory, and might require data from other processes and MPI_Send is used to request data from other processors.
- MPI_Recv: This function is used to receive messages. A processor needs to execute this function to receive messages from another processor. The message is tagged with the MPI_send message to be able to distinguish message pairs.
- MPI_Comm_size: This function determines the number of processors associated with a communicator.
- MPI_Finalize: this function stops the code execution in the MPI environment.
- MPI_Comm_rank: This function determines the identifier of a process.

Other MPI functions include but are not limited to:

- MPI_Isend: is used for synchronous communication for sending messages.
- MPI_IRecv: is used for receiving messages in asynchronous communication.
- MPI_Comm_Create: This function creates a subset of global communicators and then restricts communication to this subset only by specifying the appropriate communicator in the call [20, 22].
- MPI_Bcast: is used to broadcast one message to all processors.
- MPI_Gather: is used to collect messages from all processors.

2.2.2.3.1. Implementation of MPI

To launch MPI, all processes must be started individually. Exchange of data will be done by transferring data across the network from one processor to another. The number of nodes of the cluster to use and how many processes are required on each processor is defined by the programmer during execution. The first function to be executed before any other function is the MPI_Init, MPI_finalize is called at about the time when the program is about to finish. Any other MPI function added after MPI_finalize is ignored and cannot be executed. For this thesis, we executed the MB*tree algorithm using the master-slave approach illustrated below. Figure 5 below shows the Master slave model in which the master node is responsible for coordination, control, partitioning and assignment of tasks and generating the final result from the collective results of the slave nodes.

2.2.2.3.2. MPICH2

MB*tree algorithm in this thesis was implemented with MPI standards using MPICH. MPICH [29] is an open source project used for MPI Implementation. It is one of the most popularly used projects for executing MPI. Majorly developed at the Argonne National Laboratory, it produces portable implementation of MPI standards [28, 29] with programming languages C, FORTRAN and C++. MPICH can be used on distributed environment, on a network of interconnected workstations, single processor or multiprocessor computers

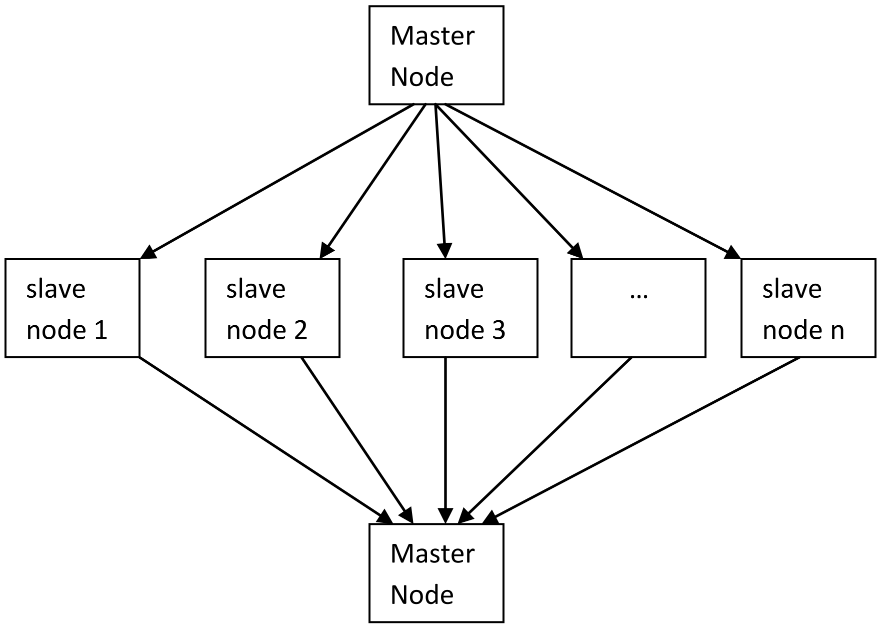


Figure 6: MPI Master/Slave Approach

2.4. Performance Modeling

Performance in parallel processing is harder to evaluate than in sequential processing because a number of processors are working independently on a single program with both sequential and parallel sections. Also synchronization and message passing are not easily measurable; however, they can be measured by inserting some codes into the algorithm. Generally, an algorithm that has a serial section will limit the speedup of an application executed on parallel architectures.

2.4.1. Amdahl's Law

One of the general laws used in parallel computing for evaluating performance and efficiency of applications is Amdahl's law [20, 22]. Amdahl's law is used as a performance prediction model for parallel applications. Amdahl's law can be used to determine the speedup achieved for a parallel application compared to its serial version. The number of threads, or processors used, determines the speedup that can be achieved [20], and also the fraction of the code that can be parallelized is an important factor that determines speedup. It is applicable in codes that have a serial portion and a parallel portion.

2.4.2. Speedup

The major benefit of parallelism is speedup. The speedup of parallel processors is governed by Amdahl's law. This is used to evaluate the time it takes to run a code in serial and in parallel with more than one processor with the assumption that the size of the problem is the same. Amdahl's law is more focused on the speedup that can be achieved from an improvement to a computation that affects a proportion of that computation where the improvement has a speedup of S.

Speedup can be defined as $S = \frac{T(1)}{T(N)}$ eqn (1)

Where T(1) is the time it takes to execute the whole application on one processor and T(N) is the time it takes to execute the application on N processors where N>1

Speedup above 100% is known as super-linear speedup. To further explore Amdahl's

Speedup $S = \frac{N}{(N*P) + (1 - P)}$ eqn (2)

where N = Number of processors and P is the % of the algorithm that is serial.

Equation 1 can be translated into $S(N) = \frac{T(1)}{T(N)} = \frac{Ts + Tp}{Ts + Tp/N}$ eqn(3)

Ts = original single processor serial time

Tp = Parallel time

Equation 3 is known as Amdahl's Law [20]

To see considerable speedup in a parallelized code, the parallel portion of the code must be much greater than the serial portion. Perfect speedup is hard to obtain as the number of threads or processors increase, the parallel overhead begin to saturate and speedup decreases. In general, speedup cannot be greater than the number of processors N. However, when speedup is obtained from a parallel application, then some of the factors considered to enhance speedup are:

More work: Serial algorithms tend to do more work than the corresponding parallel version due to enhanced features in parallel architectures.

Cache: In parallel applications, there is a larger cache available to the algorithm because data has been decomposed into smaller chunks hence enhancing speedup.

2.4.3. Overhead

One of the factors that affect the performance of parallel applications is overhead. There are various sources of overhead in parallel applications and they include but not limited to idle time, synchronization and interprocessor communication, task partitioning and assignment, thread and process creation time. Overhead can therefore be defined as :

$T_o = T_{ex} – (T_s+T_p)$ eqn 4

where Tex = Total execution time

T_s = Time to run the sequential portion of the code

T_p = Time to run the parallel portion of the code on n processors

Overheads could be difficult to measure so we can use equation (4) to compute the overhead incurred and the T_o obtained in equation (4) includes the communication time, idle time of processors, synchronization time, process creation time, and task division. In theory running an application over multiple threads or processors should improve performance when compared to running it on a single thread or processor. But there are other factors that could limit the performance of a parallel application [19, 20].

Parallel applications cause additional work to be done in terms of dividing tasks among processors or threads, creation and synchronization of threads and many more. Most threads will need to exchange data and a cost is associated to it called communication overhead. Also imbalanced load might be another factor that impacts performance. Tasks must be evenly distributed among processors or threads to avoid or reduce the overhead caused by imbalanced load. For an application to benefit from parallelism, parallel overheads have to be considered and minimized considerably [20].

2.4.4. Efficiency

Efficiency can be defined as the time spent in doing useful work [19, 20]. Efficiency of parallel application is measured by comparing work done by sequential algorithm to its parallel counterpart. Theoretically,

Efficiency = speedup / (number of processors = 1) eqn 5

where speedup = number of processors, but as most parallel algorithm comes with overhead, efficiency goes down as it increases number of processors.

CHAPTER 3

3. VLSI FLOORPLANNING WITH MB*TREE

3.1. What is Floorplanning?

In VLSI design, Floorplanning is a very crucial task, performed at the very early and initial stage of design automation. Floorplanning aims at arranging a set of module blocks in an optimized manner thereby reducing the dead space, area space and interconnection cost to a minimum. Floorplanning is known to be an NP-hard problem. There is an optimal way to arrange these module blocks and floorplanning algorithms are employed to find feasible floorplan solutions. They are optimization algorithms that solve complex problems. These algorithms are computationally intensive and time-consuming. Some Floorplanning algorithm could take several days to find a feasible floorplan solution especially for large industrial circuits. The MB*tree (Multilevel placement of large scale modules using B*trees) algorithm has been studied and it is an algorithm that scales well as the circuit size increases but also runtime increases with circuit size [4, 39].

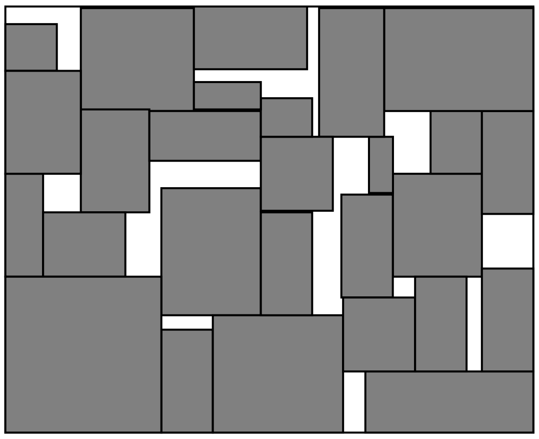


Figure 7: Arrangement of a set of module blocks on a rectangular space

Figure 7 above illustrates floorplan with a set of block modules arranged on a rectangular space in an optimal manner. The white spaces in the figure are the dead spaces which are the unused portion of the space and floorplan aims at minimizing the dead space as much as possible. Floorplans can either be slicing or non-slicing structures. In this project we have dealt with non-slicing structure. In **slicing** floorplan structure, the rectangular blocks can be further divided by either vertical or horizontal lines. For the **non-slicing** structure, the rectangle blocks cannot be recursively divided into smaller rectangles [2, 4, 7, 12, 27].

3.2. The Sequential MB*tree Algorithm

The MB*tree Algorithm [4] is based on the B*tree floorplanning algorithm and works in three phases: packing phase, perturbation phase and simulated annealing which will be discussed in details. Given a set of module blocks to be arranged in an optimal manner in order to reduce cost, minimize wirelength and dead space this algorithm will be described using the following sections.

3.2.1. The Packing phase

A primitive module is a single module block and a cluster module is a group of module blocks. Given a set of module blocks to be arranged on a rectangular space, MB* tree uses the packing algorithm to cluster the given set of modules based on connectivity and area space factors. Packing is the process whereby a given set of module blocks are placed into different groups and then the group set are clustered continuously until they make up one single cluster. The attribute for grouping is the dead space and connectivity density. The dead space is defined as the unused spaces left in between modules that are arranged on a rectangular space. The larger the cluster of modules, the higher the connectivity. The algorithm below describes how modules are clustered together into one. For example, if we need to cluster two module blocks m_1 and m_2 assuming m_1 is on the left of m_2, placed horizontally, then m_2 will be the left child of m_1 for the corresponding B*tree[2,4].

The relation for each pair of modules in a cluster is established and recorded in the corresponding B*-sub tree during clustering. It will be used for determining how to expand a node into a corresponding B*-sub tree during declustering. At the end of clustering an expandable one node tree is formed and the next algorithm is applied to the tree. An initial floorplan is obtained from the clustering of the modules into one single cluster and a corresponding tree is constructed which will be the initial input for refinement in the perturbation stage. The wirelength is then computed for the current solution. The clustering algorithm is illustrated below:

```
1.  Initialize a stack
2.  Initialize the root node index int p = nodes_root
3.  call subplace _module and pass it our root node    index  subplace_module
    (p,NIL);
4.  Initialize the root node on each node
5.  On the root, stack the node to the right side of the root node; directly above it.
    if (n.right != NIL)      S.push(n.right)
6.  Stack the node to the left side of the root node; pushed directly above the node
    at its right side. if(n.left  != NIL)      S.push(n.left);
7.  In order, traverse the stack, till the stack is empty
8.  Inside this loop check if the current popped node from the stack is at the right
    side of its parent if so, call the subplace_module function to pack the nodes
    along with its neighboring nodes and then push the sub cluster on to the stack
    further
9.  Compute Width, Height
10. Record this module information to the corresponding module. Calculate the
    floorplan area and update it each time.
11. Calculate the wirelength
```

Figure 8: Description of the Packing Algorithm

The subplace_module function deals with the geometry of the modules, like rotation and swapping; it is responsible for rotation and swapping of modules for optimal arrangement [4, 39]. It transforms the modules during the packing phase when they are being clustered together.

3.2.2. The Perturbation phase

In the perturbation stage, Simulated Annealing explained below is used for refining the floorplan solution obtained previously. Area utilization (dead space) and the wirelength among modules are the declustering metrics [2, 4]. **Dead space**: These are the unused spaces on a rectangular space with arranged modules. **Wire length**: This is the wirelength of a net is measured by half the bounding the length of the center-to-center interconnection between the modules in a net [4]. To find the total wirelength of a floorplan an aggregate of interconnecting wirelength is computed for the total wirelength value. The declustering stage iteratively expands a node into a sub tree according to the B*-tree constructed at the packing phase and then Simulated Annealing is used for refinement [4]. To find a neighboring solution, we perturb a tree to get another tree by simulated annealing.

The next section gives a brief description of the Simulated Annealing algorithm.

3.2.3. Simulated Annealing

Simulated annealing is an iterative and probabilistic optimization algorithm popularly used for floor plan optimization [2, 4,]. Simulated Annealing works basically on three rules. 1st Rule: Rotate a module; 2nd Rule: Move a module to another place; Rule 3: Swap two modules [4]. These three operations are applied to the initial tree formed for each node of the tree and based on these operations a new tree is formed which implies a new floorplan solution. SA is applied to subsequent solutions formed until a predefined value is reached and the tree is converted to the corresponding floorplan. Simulated Annealing algorithm uses the concept of thermodynamics for the growing of crystals [7, 8, 12]. In the annealing process, with minimal global energy expended, materials in their solid state can form an almost perfect crystal. The solid materials go through a state of melting and cooling, the temperature is raised to a high point and as the temperature decreases steadily, based on predefined value, the material cools down as atoms form a perfect crystal. This concept is simulated for optimization in other areas. The cost function in MB*tree is the energy that needs to be minimized [4, 5, 6] and also the modules are the atoms. Simulated Annealing in floor planning begins with initiating a floor planning placement solution in which the modules are clustered randomly which occurs in the packing phase and the temperature is set to a high level, then a new solution is generated by refinement in the perturbation phase using the operations in Simulated Annealing[2, 4]. The packing and perturbation continues and the temperature decreases gradually, solutions are accepted or rejected based on some predefined rules. The solution obtained is dependent on how high the initial temperature was set and the conditions or criteria for cooling. The slower it cools the higher the probability of obtaining better results. Dynamic and static cooling method can be used. The solution which achieves the minimum of the objective function over the iterations is the best one.

3.3. RELATED WORK

The parallel algorithm in our work is based on a sequential MB*tree for floorplanning proposed in [4]. This approach uses two basic methods for Floorplanning optimization: clustering and declustering and it is based on B*tree representation for non-slicing floorplan proposed in [2]. There has been a number of algorithms proposed for non-slicing floorplans like O-tree[14], Corner Block List[11], B*tree[2], MB*tree[4], Sequence pair[15] and each of these algorithms provide different solutions to floorplan representation with each having its own advantages and disadvantages as stated in [2]. Researchers have also proposed parallel implementations for a number of these algorithms. In [7], Tang et al proposed a parallel genetic algorithm for floorplan area optimization using O-tree representation. This approach adopts Island model with an asynchronous migration method and uses different chromosome representation [7]. Also in [1], Liang et al proposed a parallel VLSI algorithm using corner block list representation which gave considerable speedup. In this method, multiple markov was used in parallelization. Our parallel MB*tree with its 2-stage technique uses a master-slave approach for parallelization, and compared to other implementation, MB*tree scales better as the circuit size is increased [4].

CHAPTER 4

4. THE PARALLEL ALGORITHM

4.1. MB*Tree parallelized for Message Passing Interface (MPI)

To accelerate the sequential algorithm, a parallel approach is incorporated to implement the algorithm with MPI and OpenMP models. The objective of our parallel algorithm is to reduce the computation time, and not compromising the quality of floorplan solution. The wirelength calculation is a very time consuming task of the packing phase. It takes a considerable part of the computation time because each update or change in the B*tree at the perturb phase would require wirelength computation in the packing phase; in order to accelerate this, we have proposed a parallelized approach which splits the cluster modules onto slave processors/threads by the master to compute the wirelength without diminishing the quality of the floorplan. The packing phase of the algorithm is where the wirelength computation occurs. The wirelength calculation occurs at the end of the packing phase [21]

4.1.1. MB*Tree on MIMD Architecture

The MIMD (Multiple Input multiple Data) model was used to implement MB*tree algorithm on Message passing Interface (MPI). The MIMD model was employed to achieve parallelism of MB*tree. We implement MB*tree on MIMD architecture in which each processor node has its own local memory and does not have information of other nodes' memory. We used an interconnection of workstations and employed one processor as the master that controls communication, control and task assignment and the other processor are the slaves that carry out the tasks assigned to them by the master. Each node communicates with the master node in the master-slave approach we employed to parallelize MB*tree. They are multicomputer with local memory. The sequential algorithm in [4] was implemented using a SISD (Single instruction Single data) in which the application is executed on one single core like a conventional computer.

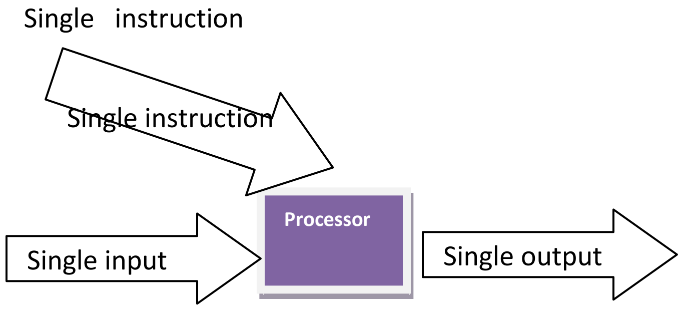


Figure 9: SISD architecture

MIMD is a popular and well accepted architecture for multiprocessors. Here each processor operates separately on its set of data simultaneously. Each processor accesses its own program memory i.e. Non–Uniform Memory Access (NUMA). The architecture of the multiple computer used to run parallel MB*tree is based on MIMD which runs multiple tasks on multiple machines.

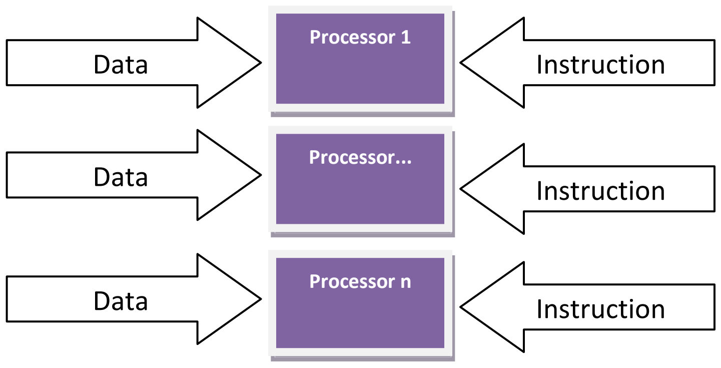


Figure 10: MIMD Architecture for MB*Tree Parallelization

Distributed memory MIMD with MPI and shared memory MIMD with OpenMP have been used in this project to parallelize MB*tree and this will be discussed in details in the following sections.

Figure 9 shows a MIMD architecture, in the distributed MIMD, data is distributed to multiple processors with private memory. Each computing node in this architecture has a processor, memory and communicator.

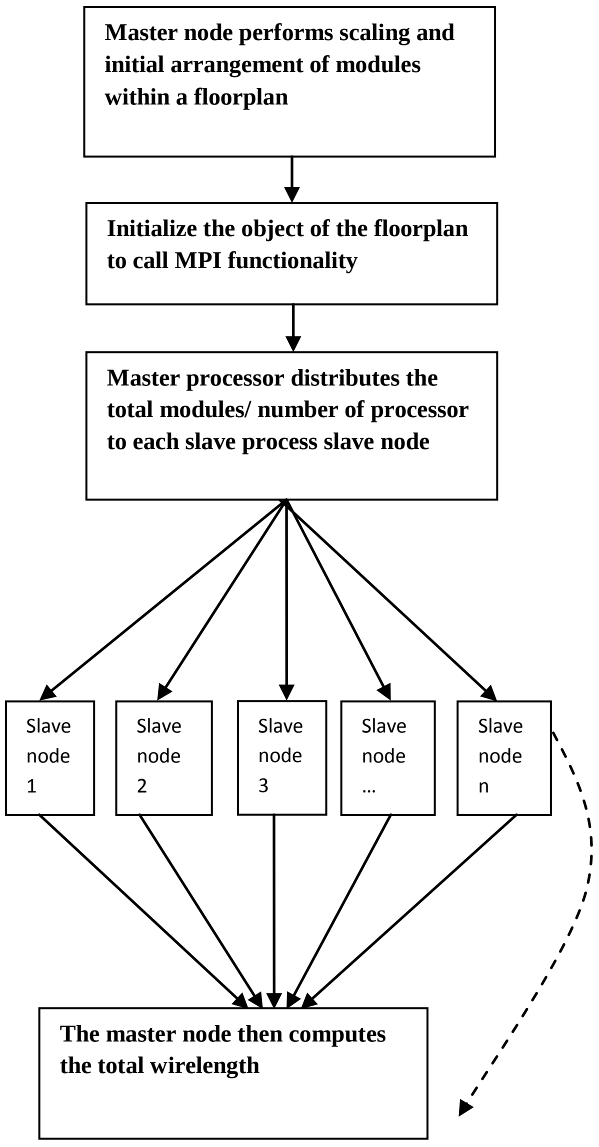
4.1.2. Parallelization Strategy

We employ a master-slave model. All slave nodes will send results of their local computation of their assigned tasks to the master. The master computes the aggregate wirelength result and uses this result for the next refinement of the floorplan, and this is distributed among the slaves for computation, until the best floorplan solution is achieved.

4.1.3. Implementing MB*tree with MPI

The MPI approach creates processes and our method is based on master/slave approach. The master node controls the communication, coordinates and assigns tasks to slave processors. It reads the floorplan solution data, divides the modules by the number of processors and sends them to the slave processors using MPI_Send() and the data is received by the slaves using MPI_Recv( ).Each of the slaves carry out computation of wirelength, the master gathers the results and computes the aggregate wirelength for the entire floorplan. Figure 10 shows the flowchart for the parallel MPI implementation of MB*tree on distributed processors. When the packing function is called, and the area has been updated, the cluster modules are sent by master to the slaves. Because the wirelength computation has to be done each time there is an update to the floor plan based on the perturbation phase, the wirelength calculation has to be calculated for each new update, hence we have parallelized this to achieve speedup. As shown in figure 10, at the initial stage, the master node distributes the cluster modules to the slave nodes (number of clusters/n) where n is the number of processors.

In our parallel algorithm, each of the slave nodes get their share of the cluster modules and then the relative position of these modules with respect to each other, their distance relative to the master module cluster to compute its own module distance from each other and the number of pins on each module is obtained, these parameters are used for the wirelength computation on each slave processors and hence the wirelength is calculated taking into account the number of pins on each module. In order to get their relative distances from each other, their position must be known and in order to get each module’s position, they will reference the module information which contains each module's distance from the master module and the master processor performs final connections among the clusters returned by all processors after each processor has forwarded its wirelength results to the master.


Master node performs scaling and initial arrangement of modules within a floorplan
Initialize the object of the floorplan to call MPI functionality
Master processor distributes the total modules/ number of processor to each slave process slave node
Slave node 1
Slave node 2
Slave node 3
Slave node …
Slave node n
The master node then computes the total wirelength

- **Each slave processor tracks the location of its modules**
- **Each slave processor calculates the pin that falls within the range and then calculates its wirelength**
- **Each slave processor forwards its wirelength result to the master node**
- **While the slave nodes are doing their computation locally, the master node is computing wirelength for wires that extend between processors.**

Figure 11: Parallel MB* tree algorithm using the master-slave approach

4.1.4. Task decomposition and assignment

In parallel programming, decomposition also known as partitioning is one of the first steps in designing a parallel algorithm after the algorithm is studied and understood by a programmer. This involves breaking the problem space of the serial algorithm into multiple tasks to be assigned and run concurrently on multiple processors. The speed-up and performance of a parallel code is highly determined by the level of decomposition of the problem space into concurrent tasks. There are two approaches to decomposition: domain or data and functional or task decomposition [22]. In data decomposition, the problem space is broken down by data and each chunk of data set is processed concurrently by multiple processors. If the dependency among the data is high, then this approach might be difficult to implement. Task decomposition is easy to use if there are no dependencies. In data decomposition, the problem space is partitioned based on tasks and not data, hence a programmer has to analyze all the tasks performed in the algorithm to check which ones can be parallelized. We describe two decomposition methods used our parallel algorithm in the following sections.

4.1.4.1. Task decomposition and assignment by module blocks

The binary tree structure is used in the representation of modules in a floor plan for the serial algorithm. The tree is formed at the initial stage of the algorithm during clustering, the decomposition of data is based on the B*tree, which is illustrated in Figure 12. In our parallel algorithm the modules are partitioned based on the tree structure. One or more binary nodes are mapped onto each processor depending on the number of nodes and number of processors. The splitting of nodes begins from the bottom of the tree to the top. Since a binary tree has a maximum of two leaf nodes for each non-leaf node, the group of modules is divided by the number of processors and each processor has equal number of modules. In the case where the number of processors is not divisible by the number of nodes, then the remainder is given to the master node and in this case the master node has more number of modules than the other processors.

4.1.4.2. Pseudo code for data decomposition

```
1.  Start node grouping from the leaf node i.e. bottom of the tree
2.  If the binary tree is balanced, group each leaf node with the same root in one cluster; the last two levels of the tree is grouped
3.  One group will contain 3 nodes or modules, that can form a binary tree with 2 leaf nodes
4.  Move bottom up to the next 2 levels and group other leaf nodes with their root.
5.  Leaf nodes on the same level with different roots cannot be grouped together.
6.  If a node is unbalanced for grouping, then select a node one step above it into the same group
7.  If a node is divisible by 3, all nodes should have equal number of modules, otherwise the remainder ungrouped node is assigned to a group
8.  If a tree is unbalanced at the leaf node level with just one leaf node at the bottom, begin grouping at the next level, if there are remainder nodes at the top of the tree, put the leaf node in the group
9.  Group in threes until no node is remaining and all nodes are formed into groups.
10. Assign each group to a processor.
```

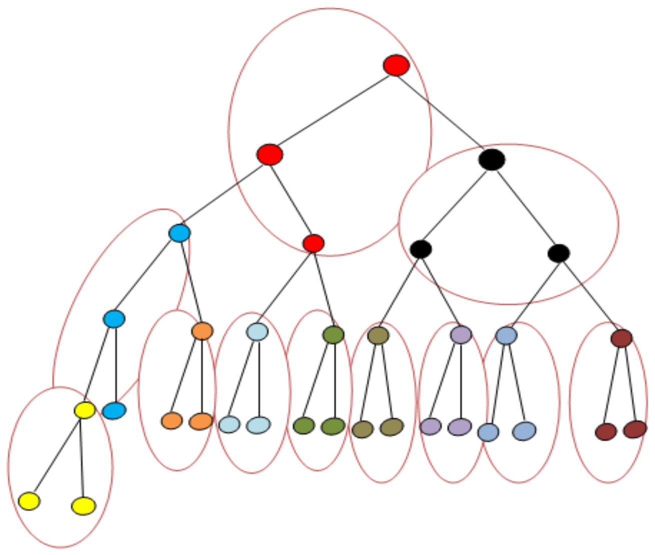


Figure 12: Task partitioning for a circuit with 33 modules

Figure 11 shows a 33 module circuit partitioned into 11 groups with 3 modules represented by nodes in each module. As described in the pseudo code, the partitioning begins from bottom to top of the tree. Each of these groups is now mapped to multiple processors. In this approach, the connection between different clusters is minimized and each of the partitions has an equal computational load. In task assignment, a minimum of one cluster is assigned to a node; the master processor assigns numbers to the clusters from bottom to top, and maps each of these groups to the corresponding processor. More than one cluster can be assigned to a processor this will be an ideal situation for performance, because too many partitions leads to increased overhead and too less partitions leads to processor idling which is another form of overhead. So for the circuit in Figure 12 with eleven clusters after partitioning can be mapped to one processor each and each processor will have the same number of modules. If for example five nodes are used to execute our parallel algorithm, 2 clusters will be assigned to each processor remaining 1 cluster, which will be assigned to the master processor. But in ideal situations, it is better to use the number of processors that is divisible by the number of clusters to balance the computation load. It is a very vital task to appropriately partition the data so as to map the computational load onto the processors [32].

To assign tasks to modules, an one – to – all personalized communication [35, 36] is used, where the master nodes sends a unique message to each slave processor i.e. each slave processor gets the same number of modules but different data set to process locally.

4.1.4.3. Handling wires that extend across processors

One of the major factors of module clustering is their connectivity density i.e. module blocks with higher connectivity are placed into one group and they are positioned close to each other during floorplanning to reduce the wire length. During data decomposition, these module blocks are placed on the same processor. However, wires may extend across modules and a module may require data from another processor to compute its wirelength. This issue is addressed by the master processor. The master processor has the entire net list of the circuits used in the experiment. The net list consists of the connectivity of module blocks, the x and y coordinates of each module block in the circuit and the net. The master processor after sending data to each of the processors will assign the wirelength computation of wires that extend across processors to itself. The computation on the master processor occurs concurrently with the slave processors. This data decomposition method worked well for small scale circuit and the master node was able to handle wires that run across modules in different processors.

4.1.5. Load balancing

Static load balancing was employed to implement MB*tree parallel algorithm. This is a commonly used load balancing method used if a fixed number of processor is assigned to carry out computation task; the tasks assigned to each processor are evenly distributed and the processors are homogenous. Our parallel algorithm, distributes tasks evenly among processors to avoid idle time among processors while others are still running their assigned tasks. In static load balancing, the master node is responsible for task coordination and it assigns tasks to the slaves at runtime. In our parallel algorithm, the cluster modules are forwarded to all processors involved using the one – to- all communication type [35]. In this type of communication the master node has unique data for all the slave processors and transmits specific data to each of the slaves.

4.1.6. Parallel Wirelength Computation

For the wirelength calculation, the half perimeter bounding box method is used [2, 4]. Each module has information of its distance to the root node, contained in the module information data structure, which is their relative x and y coordinate values. Each processor or thread calculates the distances between its modules which is the wirelength within the cluster modules and finds the coordinate of the module pins to get the highest coordinate values and this will be the upper limit, similarly it will find the lower limits coordinates; the lower limits is subtracted from the upper limits to find the bounding box and then half the bounding box can be computed to find the wirelength. The half perimeter bounding box method is the half-perimeter of the minimum bounding box for which all pins are enclosed. The module lying on the lower limits of the box comparatively will be the module whose distance from the other consecutive modules will be calculated; then the distance will be calculated from the node that lies at the upper limit consecutively. However, if the positions of the pins are not given, then the computation is done using a position lying in the center of each module.

This distance is computed by every processor/thread on the modules assigned to them. The local wirelength values are sent to the master node, it then sums up all the wirelength results, scales them accordingly to yield the final wirelength for the floor plan. Figure 12 below shows the wirelength computation of a floorplan, in our MB*tree parallel algorithm, the modules are grouped into clusters and then wirelength computation takes place concurrently because the tasks have been divided among multiple processors, when each of the processors have their local results, they send their local it to the master node to compute the aggregate wirelength. Our algorithm uses the half perimeter estimation as shown in figure 12 which is a popular method for wirelength estimation. Alternatively center to center estimation can also be used in which the wirelength is estimated from the center point of each module.

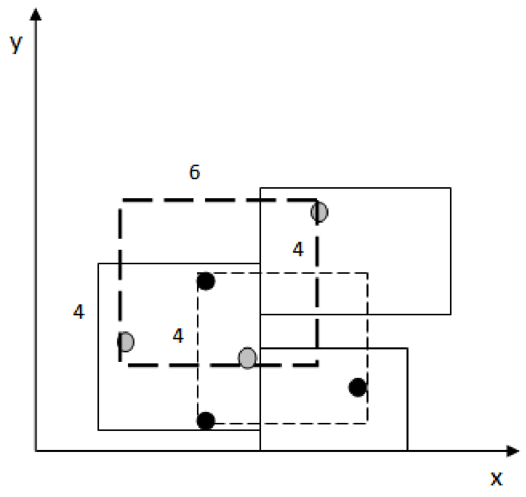


Figure 13: Half-perimeter wirelength estimation method

In Figure 13 above, half the perimeter of the bounding rectangle that encloses all the pins of the net to be connected is used to estimate wirelength. The figure illustrates 3 module blocks with net pins that extend to all the 3 modules. The grey dots indicate pins in one net and the thick dotted bounding box is the minimal rectangle that covers the net pins. To use the HPWL to estimate the wirelength, we find the length and breadth values of the red bounding rectangles and half the perimeter is the estimated wirelength.

Thick dotted bounding box wirelength estimate using HPWL = (6+4)/2 = 5

Another net in figure 12 is illustrated with the black dots with 3 net pins bounded in their minimal bounding rectangle, the HPWL is calculated below:

Thin dotted bounding box wirelength estimate using HPWL = (4+4)/2 = 4

There could be more than 2 nets across multiple nets and this method is used to estimate the wirelength across each of them and the aggregate is computed finally by the master node in case of parallelism.

4.2. MB*Tree parallelized with Open Multiprocessing (OpenMP)

In the multithreading process with OpenMP, we implemented on a quad core computer using the fork and join approach. The master thread creates a thread and the worker threads are assigned data from the master thread, each then computes its own part of the wirelength calculation simultaneously. The node modules are split among the workers on each core. The OpenMP directive is added to the wirelength calculation section. #pragma omp parallel is used before the region of the code that will be parallelized. The master thread encounters the pragma directive and spawns a predefined number of worker threads and then divides tasks among the existing threads. To minimize the overhead caused by multiple creations of fork and joins we have restricted the OpenMP directives to the wirelength calculation section of the code only. At the initial part of the code, the worker threads are idle as the master thread carries out the sequential part of the code and then it finds an initial floor plan solution, creates forks and splits the modules among worker threads to begin wirelength computation. At the end, the threads involved with computation waits for each other to complete tasks, pragma omp barrier is added in order to achieve this. The master thread creates a join and gathers the results of each thread to compute the final wirelength result; omp_set_num_threads is used to define the number of threads to be created by the master thread. Figure 14 shows a description of the OpenMP implementation with fork and joins.

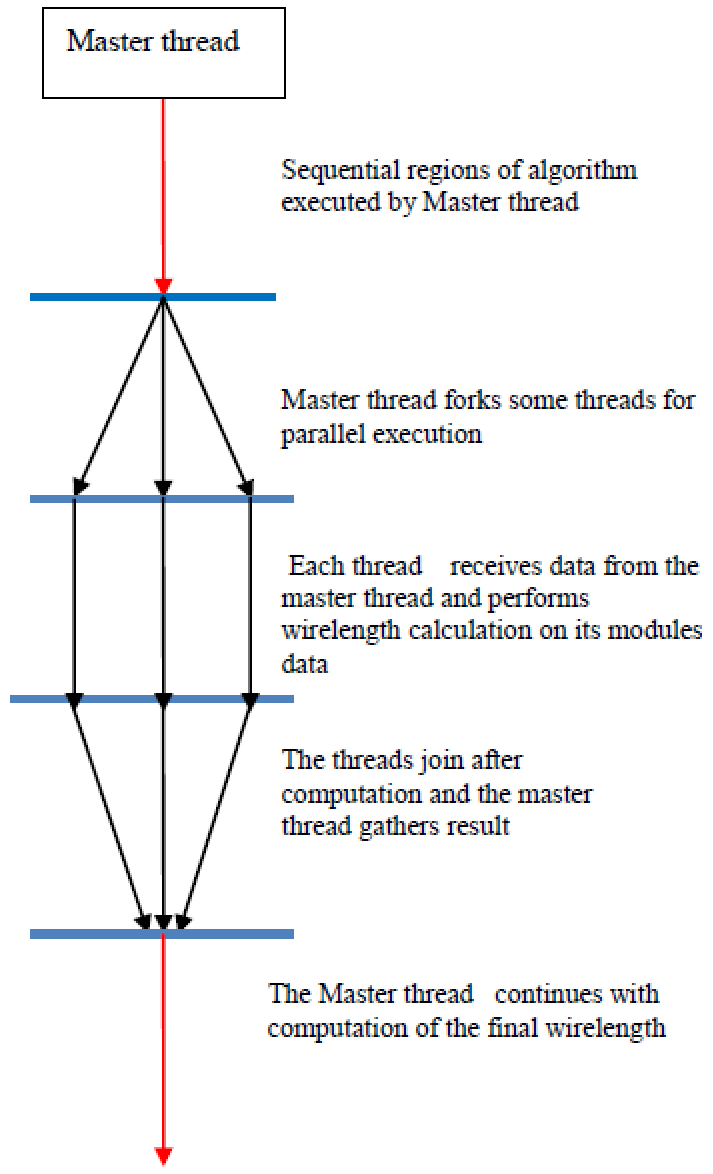

Master thread
Sequential regions of algorithm executed by Master thread
Master thread forks some threads for parallel execution
Each thread receives data from the master thread and performs wirelength calculation on its modules data
The threads join after computation and the master thread gathers result
The Master thread continues with computation of the final wirelength

Figure 14: showing the description of the OpenMP implementation

4.2.1. Execution Mechanism for Parallel Section and barrier synchronization

The MB*tree algorithm has some serial section that is executed solely by the master thread. It also contains some sections to be parallelized and those will be referred to as the parallel section.

The master thread executes the serial section with 1 thread and at the point where the master reaches the parallel section in this case the packing phase, the master sends out data to each thread to start computation. The worker threads on receiving the request from the master, executes the parallel section in the packing stage. The master and workers execute an identical executable object. However, as soon as the program is executed, the workers immediately go on standby, and wait while the master completes executing a serial section. When the master has finished executing the serial section, it requests the workers to start executing the parallel-execution section by utilizing barrier synchronization. Barrier synchronization is a mechanism that synchronizes more than two computing nodes at the same time. Each computing node temporary suspends processing at the provided point of the program and waits until all computing nodes arrives at the same point of the program. Barriers are used in sections of the parallel algorithm to enforce a wait on processors that finish up task faster before proceeding to the next computation phase. All the other processes must reach the barrier before the computation can proceed.

Computing is accomplished in parallel with multiple threads. After the parallel-execution section has been executed, barrier synchronization is again conducted, and the workers again return to standby; the master then executes the serial execution section (see Figure 13). A request for execution of the parallel execution section is informed from the master to the workers by using the barrier synchronization directive.

CHAPTER 5

5. EXPERIMENTAL METHODOLOGY

This chapter details the methodology and tools used to implement the parallelization of MB*tree Floorplanning algorithm. The parallel MB*tree algorithm was implemented using the C++ programming language with MPI and OpenMP parallel programming models.

For distributed architecture, we used the master-slave approach where one processor is named the master and performs coordination, task partitioning and task assignment and control. MPI was employed as the programming model. The main MPI functions used in our implementation are listed on the table below:

MPI_INIT	Initialize the MPI environment
MPI_COMM_SIZE	Returns the number of processes
MPI_COMM_RANK	Returns the rank of the process
MPI_SEND	Sends message
MPI_RECV	Receives message
MPI_FINALIZE	Terminate MPI process

Table 1: Main MPI functions used to parallelize MB*Tree Algorithm

MPICH2 version 1.4 was installed and used to implement MB*tree in parallel .MPICH2 is a high performance and widely portable implementation of the MPI standard [29]. It provides MPI implementation that supports computation on homogeneous computer systems [29]. MPICH1 provides support for heterogeneous platforms.

5.1. Installing and Building MPICH2

MPI was built and installed on Ryerson University Electrical Engineering (EE) network to implement parallel MB*tree algorithm. This section briefly describes the installation and configuration of MPI standards on Ryerson EE network. In the Department of Electrical and Computer Engineering, all the workstations located in the labs, are networked together with home directories and applications served from a Central File server. All workstations have access to the Internet through a gateway. The network consists primarily of x86 workstations running Linux (Fedora Core 5) and Sun SPARC workstations running Solaris (5.9); some Sun workstations have Sun PC cards running Windows 98. A list of some workstations used as servers in this project is as follows:

1. brampton.ee.ryerson.ca
2. acton.ee.ryerson.ca
3. danforth.ee.ryerson.ca
4. eglinton.ee.ryerson.ca
5. whitby.ee.ryerson.ca
6. clarkson.ee.ryerson.ca
7. dixie.ee.ryerson.ca
8. kipling.ee.ryerson.ca
9. guelph.ee.ryerson.ca
10. bloor.ee.ryerson.ca
11. oshawa.ee.ryerson.ca
12. york.ee.ryerson.ca
13. ajax.ee.ryerson.ca
14. malton.ee.ryerson.ca
15. sequence.ee.ryerson.ca
16. vibrato.ee.ryerson.ca
17. vivace.ee.ryerson.ca
18. requiem.ee.ryerson.ca
19. gigue.ee.ryerson.ca
20. forte.ee.ryerson.ca
21. concerto.ee.ryerson.ca
22. minuet.ee.ryerson.ca
23. theex.ee.ryerson.ca
24. keele.ee.ryerson.ca
25. appleby.ee.ryerson.ca

26. finch.ee.ryerson.ca
27. sequence.ee.ryerson.ca
28. sorano.ee.ryerson.ca
29. alto.ee.ryerson.ca
30. concerto.ee.ryerson.ca

Table 2: List of Computer nodes running MPI

These computers can be found in ENG406, ENG411 and ENG412.

MPICH2 uses a default set of configuration options during installation; this creates the socket communication device and the MPD process manager, for languages C, C++, Fortran-77, and Fortran-90 with compilers chosen automatically from the user's environment, without tracing and debugging options. It uses the VPATH feature of make, so that the configuration process can take place on a local disk to improve speed[29]. The mpich2.tar.gz in [29] is unpacked and installed into "home/you/" into our master server; brampton. This directory is shared by all computers in the network. MPICH2 is configured based on the commands in [29]. The MPICH2 commands are also installed and the bin subdirectory of the installation directory is added to the installation location path using "setenv" command based on instructions in [29]. The following commands are used to check the bin subdirectory:

- which mpd
- which mpicc
- which mpiexec
- which mpirun.

On executing these commands, a path is displayed in the command line that shows that each of these installation files exists in the bin directory as this is crucial to the smooth running of MPI in the execution environment.

At this stage the directory is on all computers on the network since it is a Network File System (NFS). MPICH2 uses a process manager to start MPI jobs on multiple machines. The process manager is known as MPD, it forms a daemon ring on the set of computers where MPI programs run[29]. The "mpd &" is used to start up MPD on one computer, "mpdtrace" command displays what computer is started by the daemon. The "mpdboot –n <number to start> -f mpd.hosts" command is run on one of the machines to start MPI on a number of machines depending on the value of n. The value of n cannot be greater than the number of machines listed in the mpdhosts file. The mpdhosts file consists of a list of the computer names on the EE network that will be used to run the parallel MB*Tree algorithm. When the daemon is started, MPI should be running on the machines listed and to check what machines are running in the MPI ring, the "mpdtrace –l" command is used and this displays the list of machines running MPI. All the computers used in executing the code are running on Linux platforms. I used up to 22 interconnected machines to execute the parallel program for MPI.

Computers in ENG 412 have the same specification as shown below:

Processor	- Intel (R) Xeon (R) CPU
Proc. Speed	- 2.00 GHz
Cache	- 6144KB ~ 6MB
Network speed	- 100Mbps
CPU MHz	- 1995.042

ENG 406 and ENG 411 machines have 2 varying specifications

Processor	- Intel (R) Xeon (R) CPU
Proc. Speed	- 2.00 GHz
Cache	- 6144KB ~ 6MB
Network speed	- 100Mbps
CPU MHz	- 1995.042

Processor	- Intel (R) Core ™2 Quad CPU
Proc. Speed	- 2.40 GHz

Cache- 4096KB ~4MB
Network speed- 100Mbps
CPU MHz - 2394.201

In the OpenMP model, one main thread and other worker threads are created. The communication overhead is lower than in MPI because communication between the master and worker threads is within a node eliminating network communication overhead as in MPI. The MB*tree algorithm was compiled and run using a single quad core: Brampton. For the shared memory model programming, OpenMP was employed as the Application Programming Interface (API). With the operating system on each of the machines listed above, OpenMP is already incorporated in gcc; hence, no installation is required. The gcc version 4.2.1 already installed on all machines, supports "-fopenmp" option to compile OpenMP programs. For the OpenMP version, Brampton server in ENG 412 with Intel Xeon processor , 4 cores and Linux operating system with gcc compiler. Due to resource limitation i.e. only a maximum of 4 cores exists on one node, we were unable to run the parallel MB*tree algorithm on a larger number of cores on the same node.

The data set used for running the parallel code is the Microelectronics Center of North Carolina (MCNC) benchmark circuits [4, 37] were used. The ami33 and ami49 were used. These circuits were duplicated up to 4 times to obtain a larger circuit to validate the benefits of parallelism on large circuits that could take several hours to obtain a quality floorplan solution.

Barrier synchronization is used in both MPI and OpenMP versions of the parallel algorithm. For the OpenMP version, "#pragma omp barrier for" directive is used to impose barriers i.e. this instructs all the threads to stop at the point where it is added to the code and wait until all the other threads complete their computation and reach the barrier. For MPI, int MPI Barrier (MPI Comm comm.) was used to instruct all processes to wait until all other processes reach the barrier. The master thread or processor can now gather all results for the final computation.

5.2. Challenges faced in implementing parallel MB*Tree algorithm.

One of main challenges faced parallelizing MB*tree is data partitioning, as the data partitioning scheme plays an important role in the performance of the parallel algorithm. A new partitioning algorithm was devised to decompose the cluster module blocks into sub-trees and assigned evenly to the processors or cores avoiding unbalanced load. In the OpenMP implementation, due to limited number of cores on the machines, we were unable to explore the capability and performance of OpenMP on more than 4 cores.

CHAPTER 6

6. RESULTS AND DISCUSSION

This chapter contains the performance results of the parallel MB*Tree Algorithm and discussion of these results. We optimized the Floorplanning algorithm for Multilevel Floorplanning/placement for large scale modules using B*tree (MB*tree) on MPI and OpenMP parallel models using C++ programming language and the experiment was executed on an Intel Pentium/Celeron processor (2.4GHz) running GNU Linux. For the MPI test we used up to 22 interconnected workstations. The MCNC benchmarks ami33 and ami49 which contains 33 and 49 modules respectively were used for testing. Duplicates of these circuits (two and four times) were also used for testing in order to study the performance of parallelism on larger circuits.

6.1. Runtime and overhead results of ami33 circuit with module based partitioning (sequential vs. parallel) - MPI

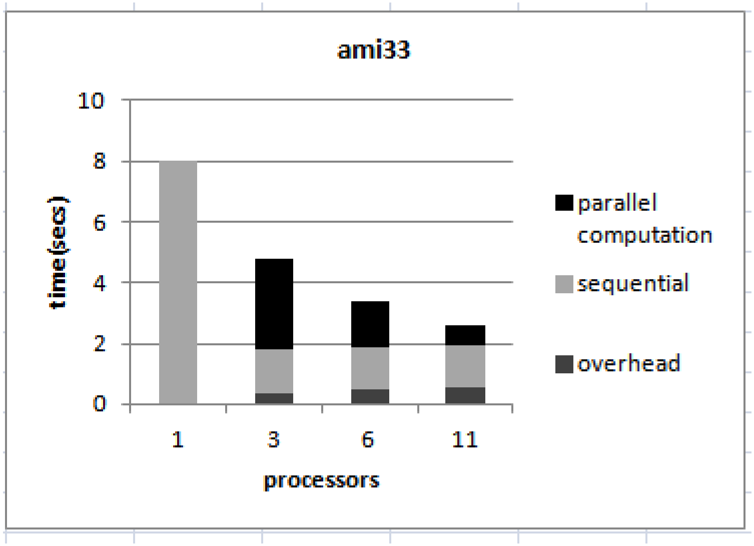


Figure 15: Bar graph showing the execution time for ami33 circuit with MPI

Figure 15 above shows the execution time for ami33 circuit in terms of time spent in executing the sequential part of the code, time spent on parallel computation and also the overhead incurred during communication and computation. The algorithm was run on 1, 3, 6 and 11 processors. The one node bar represents a purely sequential code, in which the overhead is very negligible. The sequential time is measured on one node; this is obtained by using the time function to measure the execution time of the code from start to finish. The parallel computation time is the time it takes to run the parallel part of the code which is majorly the clustering section of the code where wirelength calculation is done on multiple computers. The overhead is then computed using equation 4. To = Tex – (Ts+Tp). Where Tex is the total execution time, Ts is the sequential time and Tp is the parallel computation time. The overhead time covers the time it takes for communication between processes and the time it takes to assign and synchronize data between processes. A speedup of 2.8 was obtained on 11 nodes although the overhead incurred increased as the number of processors is increased, the total execution time was greatly reduced. The overhead incurred includes the time it takes the master node to compute the wirelength for the wires that extend across processors.

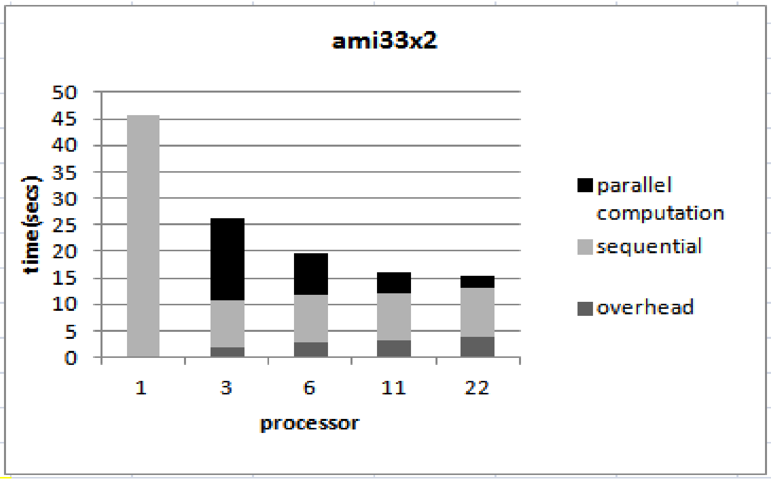


Figure 16: Bar graph showing the execution time for ami33x2 circuit with MPI

In figure 16, we duplicated ami33 twice the circuit is twice as big as ami33. A speedup of 3 is obtained from up to 22 nodes. The number of processors is divisible by the number of nodes, on 22 nodes, each processor would have one cluster module containing 3 blocks, on 11 processors, and each cluster module will have 2 clusters of 6 blocks in total. The overhead in the ami33x2 circuit is higher than the overhead in ami33 because there is more modules and nets, hence more computation work. In figure 17 below, ami33 was duplicated 4 times and this produces a larger circuit, better speedup was achieved for ami33x4 with 3.4 on 22 processors. This is a good indication that the parallel algorithm will scale better as the circuit size increases.

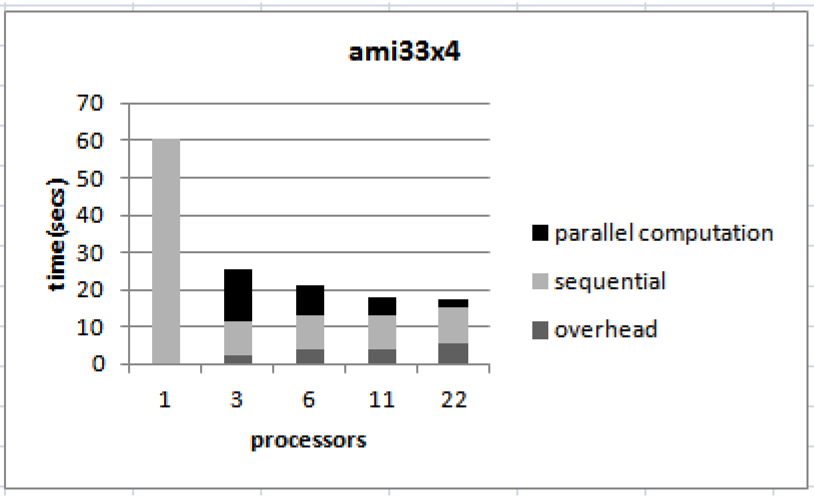


Figure 17: Bar graph showing the execution time for ami33x4 circuit with MPI

6.2. Runtime and overhead results of ami49 circuits with module based partitioning (sequential vs. parallel) – MPI

Figure 18 below shows the result of ami49 on 1,2,4,7 and 14 processors. On 14 processors the overhead is the highest due to the fine granularity nature of the data and the number of processor involved. Although the runtime was reduced but the overhead incurred is higher. On 7 processors the data is coarser grained than on 14, and this reduces the number of wires that extend across multiple nodes or processors. The ami49 circuit is a larger circuit than ami33 and the speedup achieved with this circuit is better.

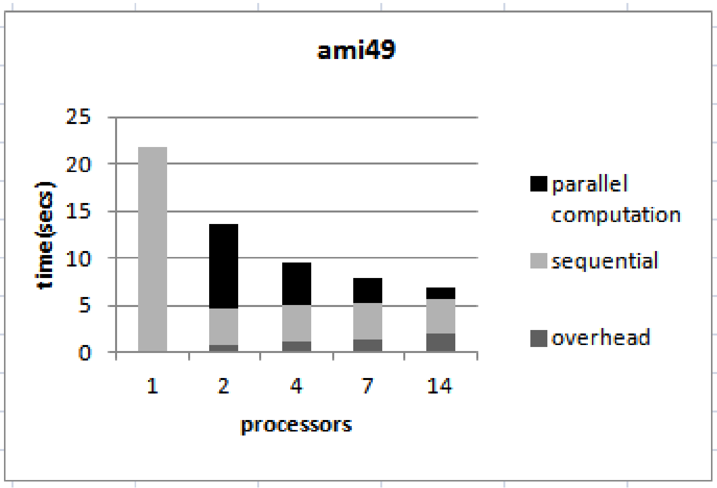


Figure 18: Bar graph showing the execution time for ami49 circuit with MPI

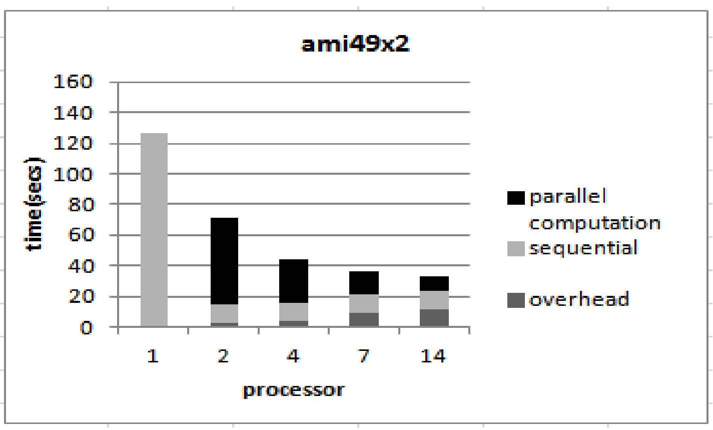


Figure 19: Bar graph showing the execution time for ami49 circuit with MPI

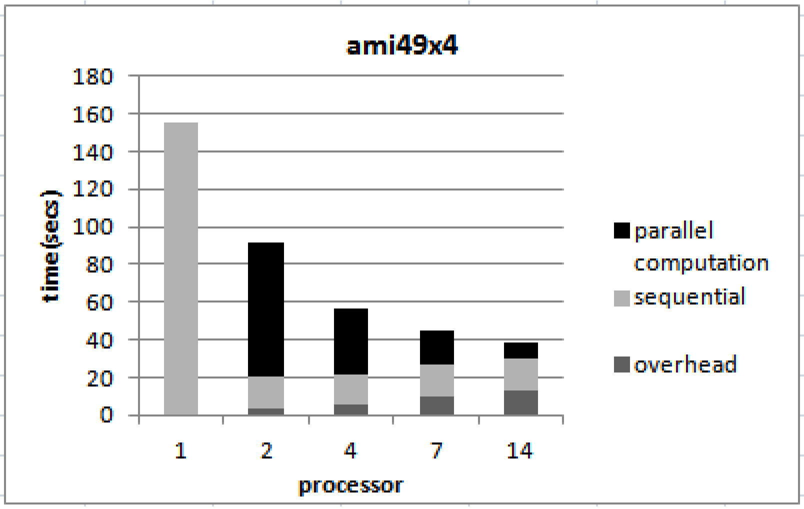


Figure 20: Bar graph showing the execution time for ami49 circuit with MPI

A speedup of 3.2 was obtained for ami49, 3.9 for ami49x2 and 4.01 for ami49x4. Generally we observed that in order to obtain better performance, the granularity of the data should not be purely fine, because the finer the granularity of data, the higher the overhead cost, as many wires will extend across processors which is incurs high overhead cost.

6.3. Effects of Granularity on the execution time

Experiments were performed using fine grained granularity and coarse grained granularity. The extent to which the data is decomposed is the known as granularity. In the parallel MB*tree algorithm, although data is broken down into very fine parts, the number of parts assigned to each processor determines how coarse grained or fine grained the data can be. For example ami33 circuit executed on 11 processors indicates that granularity is very fine i.e. a cluster of 3 modules is mapped onto each of the 11 nodes. This greatly affected the performance as the overhead increased and also too many wires will extend across multiple nodes there by increasing the amount of sequential work to be done by the master processor.

6.4. Runtime and overhead results of ami33 and ami49 circuits with (sequential vs. parallel) – OpenMP

For the OpenMP test we used a quad core workstation. Figures 18 and 19 below show results for OpenMP. The parallel algorithm was tested using, ami33x4 and ami49x4 test circuit.

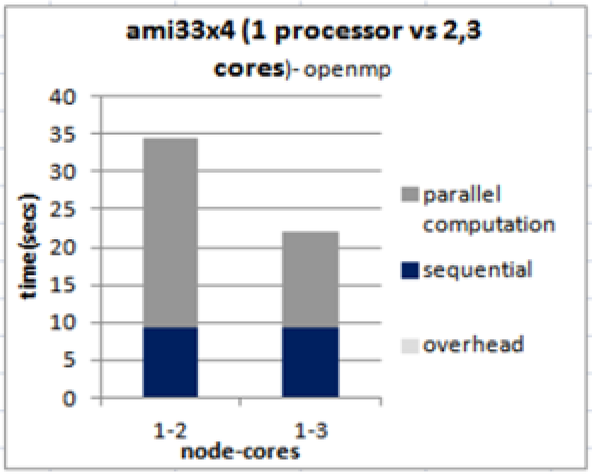


Figure 21: Bar graph showing the execution time for ami33x4 circuit with OpenMP

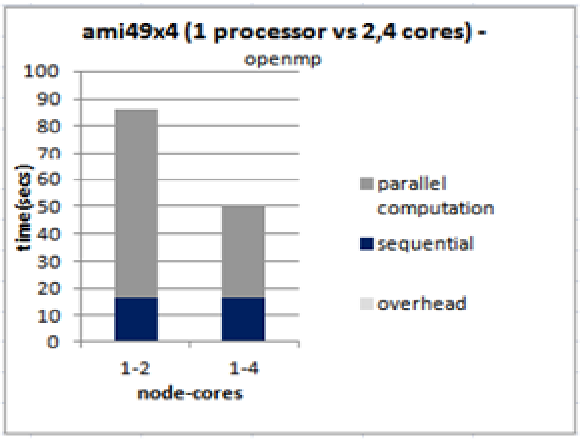
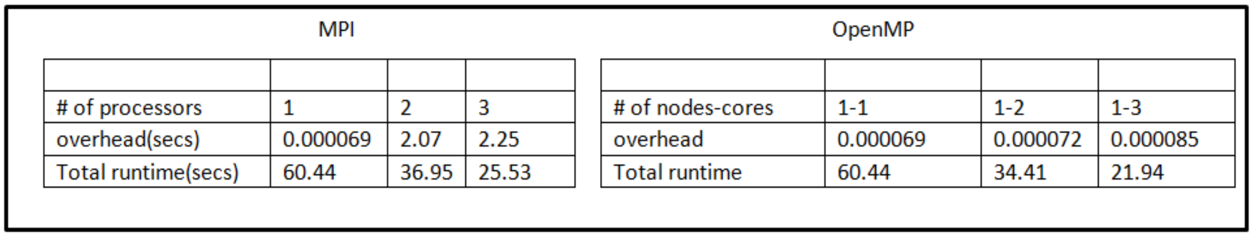


Figure 22: Bar graph showing the execution time for ami33x4 circuit with OpenMP

The overhead in OpenMP is considerably lower than the corresponding MPI results, for example the MPI result for ami33x4 in Table 3 on 4 processors compared to OpenMP result for 1 node to 4 cores shows that OpenMP produced better results in terms of execution time and reduced overhead. Similarly for ami49x4 in Table 4, result shows that OpenMP produced better results than MPI (comparing 3 nodes and 1node/three cores) due to the overhead incurred in communication across nodes.



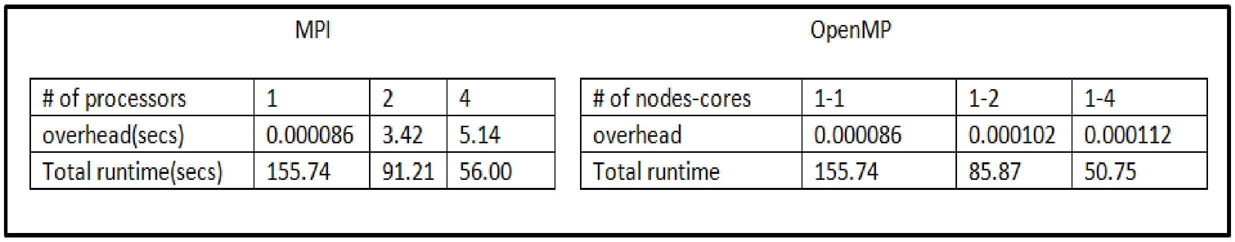
MPI

# of processors	1	2	3
overhead(secs)	0.000069	2.07	2.25
Total runtime(secs)	60.44	36.95	25.53

OpenMP

# of nodes-cores	1-1	1-2	1-3
overhead	0.000069	0.000072	0.000085
Total runtime	60.44	34.41	21.94

Table 3: Table comparing overhead and execution time of ami33x4 on MPI and OpenMP



MPI

# of processors	1	2	4
overhead(secs)	0.000086	3.42	5.14
Total runtime(secs)	155.74	91.21	56.00

OpenMP

# of nodes-cores	1-1	1-2	1-4
overhead	0.000086	0.000102	0.000112
Total runtime	155.74	85.87	50.75

Table 4: Table comparing overhead and execution time of ami49x4 on MPI and OpenMP

Overall, the results obtained from the MPI and OpenMP implementation shows that it gives better runtime and speedup than the sequential implementation. Comparing the MPI and OpenMP results, although we were only able to implement OpenMP on a single quad core node, the runtime is better than the corresponding results for MPI. Due to the resource limitation, we were only able to use a maximum of 4 cores for OpenMP implementation and with the trend of the result and speedup; it is likely to obtain more speedup as the core increases because overhead is less compared to MPI. The overhead is shown in all the results, the average overhead is obtained by taking the total time and deducting the parallel computation time and sequential time, the remaining value could be taken as the overhead.

Figures 23 and 24 shows the overhead trend as the number of processors and the circuit size increases

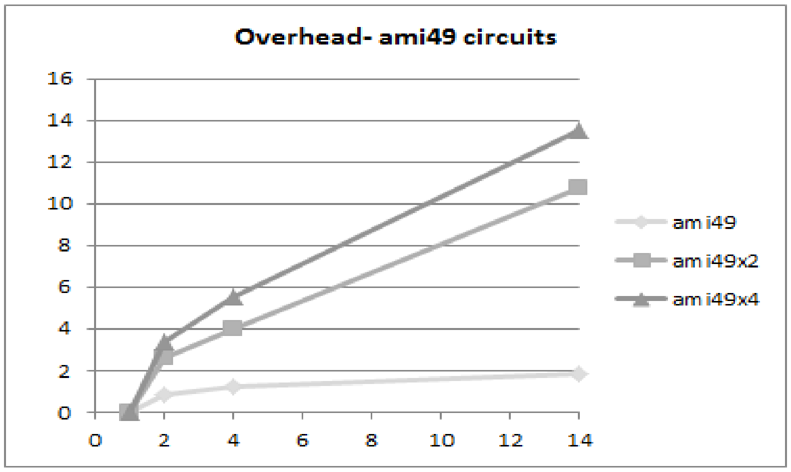


Figure 23: Graph showing overhead time for 3 ami49 circuits on MPI

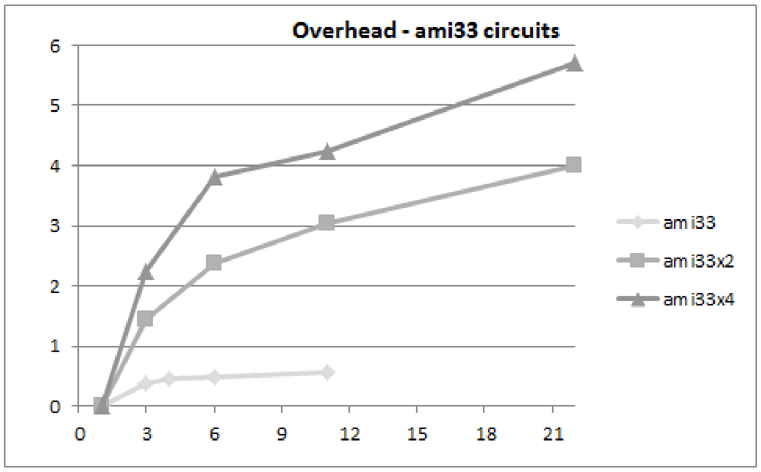
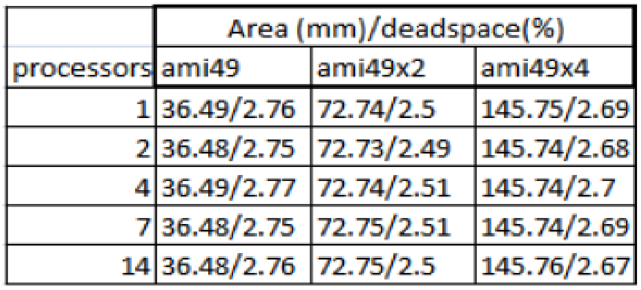


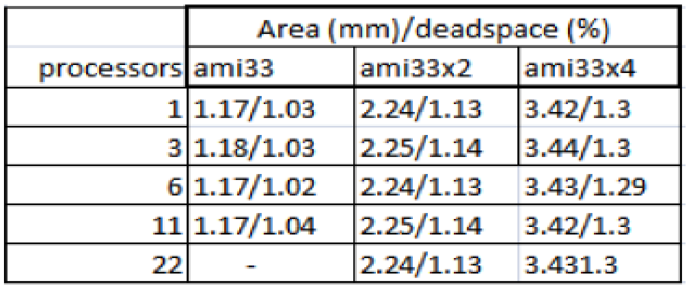
Figure 24: Graph showing overhead time for 3 ami33 circuits on MPI

For the larger circuits there is more increase in the overhead due to the circuit size and the increasing number of processors. Furthermore, the area and wirelength of the parallelized floorplans was not compromised. Tables 4 and 5 below shows the result obtained for the area and dead space for the parallelized algorithm, which is no different from its sequential version as we were able to maintain the floorplan quality while optimizing the algorithm. The quality of the floorplan was maintained while reducing computation time. The MB* tree algorithm is very scalable [4] and hence very large VLSI circuits will achieve more speedup with the parallel approach.



	Area (mm)/deadspace(%)		
processors	ami49	ami49x2	ami49x4
1	36.49/2.76	72.74/2.5	145.75/2.69
2	36.48/2.75	72.73/2.49	145.74/2.68
4	36.49/2.77	72.74/2.51	145.74/2.7
7	36.48/2.75	72.75/2.51	145.74/2.69
14	36.48/2.76	72.75/2.5	145.76/2.67

Table 5: Area and dead space obtained for ami49 and its duplicate circuits



	Area (mm)/deadspace (%)		
processors	ami33	ami33x2	ami33x4
1	1.17/1.03	2.24/1.13	3.42/1.3
3	1.18/1.03	2.25/1.14	3.44/1.3
6	1.17/1.02	2.24/1.13	3.43/1.29
11	1.17/1.04	2.25/1.14	3.42/1.3
22	-	2.24/1.13	3.431.3

Table 6: Area and dead space obtained for ami33 and its duplicate circuits

CHAPTER 6

7. CONCLUSION AND FUTURE WORK

In this thesis we were able to optimize the MB*Tree Floorplanning Algorithm by parallelization using OpenMP/MPI and evaluated the performance of different results produced by these two programming paradigms using the MCNC benchmark circuits for the experiments.

7.1. Contribution and Conclusion

In this thesis, we have implemented a parallel VLSI Floorplanning algorithm; MB* tree using two popular parallel models; Message Passing Interface (MPI) and Open Multiprocessing (OpenMP). We optimized the MB*tree algorithm by parallelization to reduce execution time especially with large circuits. Experimental results show that we were able to achieve good speedup with our parallel approach with the larger circuits giving greater speedup. The parallelism gives good results in terms of reducing the runtime and achieving good speedup which is better compared to the sequential version. Furthermore, the OpenMP version gave the best results in terms of total runtime, speedup and overhead compared to the MPI implementation. The reason for achieving better result with OpenMP is largely due to the negligible amount of overhead incurred. Since all cores reside on a single node, no data exchange across the network and communication among threads incurs less overhead compared to the MPI version. In OpenMP, if a wire extends across more than one processor, the cores can read data information about these wires and use that in computation of its wirelength hence reducing the amount of work done sequentially by the master thread unlike in MPI. The data partitioning by net method works well on the shared memory architecture in that we can divide the total number of nets by the number of cores without any issues with inter-node communication because all modules and net information are transparent to the cores. However, there is a limit to the amount of resources we have, to test the OpenMP which is a single quad core node. With a larger number of cores on a single node, our parallel algorithm will scale better. The floorplan quality was not compromised and we were able to achieve reasonable speedup in the floor plan computation and design as shown in our experimental results.

7.2. Future Work

In this work, MPI and OpenMP have been used to optimize the MB*Tree algorithm through parallelization. A future work is to use the Hybrid programming model i.e. Hybrid MPI/ OpenMP to further optimize the code and explore the possibilities of maximum performance. The hybrid programming models can utilize both shared memory and distributed memory of high-end computing systems. MPI will be used globally to exchange modules and net connectivity information and OpenMP will be used to parallelize the packing phase of the algorithm precisely the wirelength computation.

REFERENCES

[1] L. Huang, Y. Cai, X. Hong, “A Parallel VLSI Floorplanning Algorithm Using Corner Block List Topological Representation,” Proc. ICCCAS, vol. 2, pp 1208-1212, June 2004.

[2] Y-C. Chang, Y.-W. Chang, G.-M. Wu and S.-W. Wu, “B*-Trees: A new representation for non-slicing floorplans,” Proc. DAC, pp. 458-463, 2000.

[3] W.-S. Yuen and E. Young, "Slicing floorplan with clustering constraint," Proc.IEEE TCAD, vol. 22, no. 5, pp. 652--658, May 2003.

[4] H.-C. Lee, Y.-W. Chang, J.-M. Hsu, and H. H. Yang, “Multilevel Floorplanning/placement for large-scale modules using B*-trees,” Proceedings of ACM/IEEE Design Automation Conference, pp. 812-817, 2003.

[5] T.-C. Chen and Y.-W. Chang, "Modern Floorplanning Based on B*-trees and Fast Simulated Annealing," in IEEE Trans. on Computer-Aided Design of Integrated (TCAD) , vol. 25, no. 4, pp. 637--650, Apr. 2006.

[6] F. Balasa, “Modeling Non-Slicing Floorplans with Binary Trees,” Proc. of the 2000 IEEE/ACM International Conference on Computer-aided design pp 13-16 November 2000 San Jose California.

[7] M. Tang, R. Lau, “A Parallel Genetic Algorithm for Floorplan Area Optimization,” Seventh International Conference on Intelligent Systems Design and Applications, pp 801-806 October 2007.

[8] M. Chrzanowska-Jeske, B. Wang, and G. Greenwood, "Floorplanning with performance-based clustering," Proc. ISCAS, vol. 4, pp. 724--727, 2003.

[9] C.Chen, I. G. Tollis "Parallel algorithms for slicing floorplan designs," IEEE Second Symposium on Parallel and Distributed Processing pp.279-282, 1990.

[10] X. Hong, G. Huang, and T. Yoshimura, “Corner Block List: An Effective and Efficient Topological Representation of Non-Slicing Floorplan,” Proc. ICCAD, pp. 8-12, 2000.

[11] M. Li, Y.Zhou, Y. Li, "A Non-Slicing Area Prejudged Algorithm for Floorplanning without Simulated Annealing,"Proc. International Conference on Computer and Automation Engineering (ICCAE), vol 1 pp. 643-647 Feb. 2010.

[12] X. Tang and D. Wong," FAST-SP: A Fast Algorithm for Block Placement Based on Sequence Pair," Proc ASP-DAC, pp. 521 -526, 2001.

[13] P.-N. Guo, C.-K. Cheng, and T. Yoshimu, "An 0-Tree Representation of Non-Slicing Floorplan and Its Applications,"Proc. DAC, pp, 268 -273, 1999.

[14] H. Murata, Ernest S. Kuh, "Sequence Pair Based Placement Method for Hard/Soft/Pre-placed Modules," Proc. ISPD, pp. 167-172.1998.

[15] H.Foo, J. Song, W. Zhuang, H. Esbensen, E. Kuh, "Implementation of a parallel genetic algorithm for floorplan optimization on IBM SP2," High Performance Computing on the Information Superhighway, pp 456-459, April 1997.

[16] J-M. Hsu, Y-W Chang, "A reusable Methodology for non-slicing Floorplanning". Proc. Circuits and Systems, vol 1 pp 165-168, 2004.

[17] T. Yamanouchi, K. Tamakashi, T. Kambe, "Hybrid Floorplanning based on partial clustering and module restructuring," Proc. ICCAD, pp 478-483 Nov 1996.

[18] I. Shanavas, R. Gnanamurthy, "Wirelength Minimization in Partitioning and Floorplanning Using Evolutionary Algorithms," Proc. of VLSI Design 2011.

[19] P. S Pacheco, (1997) Parallel Programming with MPI, Morgan Kaufmann Pub. Inc.

[20] An Oracle White Paper on Parallel Programming with Oracle Developer Tools May 2010.

[21] O. Owojaiye, N. Mekhiel, "Parallel Implementation of Non-Slicing Floorplans with MPI and OpenMP". Proc. CATA, March 2012.

[22] https://computing.llnl.gov/tutorials/parallel_comp.

[23] N. Lasgard "Parallelizing Particle-In-Cell Codes with OpenMP and MPI" Norwegian University of Science and Technology - A technical report.

[24] David Culler, Jaswinder Pal Singh and Anoop Gupta, "Parallel Computer Architecture, A Hardware/Software Approach", Morgan Kaufmann Publishers, 1999.

[25] Bernd Mohr, "Introduction to Parallel Computing", NIC Series, Vol. 31, ISBN 3-00-017350-1, pp. 491-505, 2006.

[26] http://www.openmp.org

[27] R. Ladani, "A Recursive Approach to Floorplanning" - Thesis Report 2005.

[28] MPI Forum. "A Message-Passing Interface Standard", 1995. (http://www.mpiforum.org/docs/docs.html)

[29] http://www.mcs.anl.gov/research/projects/mpich2/.

[30] Liu, W., Hong, X., and Adamson, K., "Parallel implementation of evidence combination in qualitative Markov tress". Technical Report, University of Ulster.

[31] P. Salembier, L. Garrido, "Binary Partition Tree as an Efficient Representation for Image Processing, Segmentation, and Information Retrieval" Proc. IEEE trans. on Image Processing, vol 9 pp. 561-576, Apr 2000.

[32] W. Xue, S. Qi, "Multilevel Task Partition Algorithm for Parallel Simulation of Power System Dynamics" Proc. of the 7th international conference on Computational Science, ICCS '07 pp. 529-537, 2007.

[33] http://static.msi.umn.edu/tutorial/scicomp/general/intro_parallel_prog.

[34] S.L. Martins, C.C. Ribeiro, and N.R. Rodriguez, "Parallel computing environments", Handbook of Combinatorial Optimization, Oxford, to appear.

[35] D.Barth, P. Fraigniaud, "Approximation algorithms for structured communication problems". Proceedings of the 9th ACM Symposium on Parallel Algorithms and Architectures, 1997, 180-188.

[36] J. Duato, S. Yalmanchili, L.M. Ni, "Interconnection networks; An engineering approach", IEEE Computer Society Press, 1997.

[37] http://www.cse.wustl.edu/~jain/cse567-08/ftp/fpga/#MCNC_Benchmark_suite.

[38] http://vlsicad.cs.binghamton.edu/benchmarks.html.

[39] H.-C. Lee, Y.-W. Chang, J.-M. Hsu, and H. H. Yang, "MB*Tree: A Multilevel Floorplanner for Large-Scale Building-Module Design "IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 26, No. 8, August 2007.

[40] A. Kennings and I. L. Markov. Analytical minimization of half-perimeter wirelength. In Proc. Asia South Paci_c Design Automation Conf., pages 179.184, 2000.