# Stony Brook University

# On the Optimization of Grid Systems

A Dissertation Presented

by

**Kai Wang**

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

**Doctor of Philosophy**

in

**Electrical and Computer Engineering**

Stony Brook University

**December 2013**

**Stony Brook University**

The Graduate School

**Kai Wang**

We, the dissertation committee for the above candidate for the
Doctor of Philosophy degree, hereby recommend
acceptance of this dissertation.

**Thomas G. Robertazzi - Dissertation Advisor**
**Professor, Electrical and Computer Engineering**

**Wendy Tang - Chairperson of Defense**
**Associate Professor, Electrical and Computer Engineering**

**Carlos F Gamboa**
**Adjunct Professor, Electrical and Computer Engineering**

**Esther M. Arkin**
**Professor, Applied Mathematics and Statistics**

This dissertation is accepted by the Graduate School.

Charles Taber
Dean of the Graduate School

Abstract of the Dissertation

# On the Optimization of Grid Systems

by

**Kai Wang**

**Doctor of Philosophy**

in

**Electrical and Computer Engineering**

Stony Brook University

**2013**

Grid systems are widely used to transfer power and information in various forms in many engineering and scientific areas such as grid computing systems, electrical grids, control grid and etc. A good handling of task partition, task allocation and load balancing can significantly increase a grid systems' efficiency. In this dissertation, balancing the loads in electrical grid systems and optimizing grid computing systems are analyzed.

Unbalanced loads on feeders increase power system investment and operating costs. Three-phase lateral loads phase swapping is one of the popular methods to balance such systems. We employed a dynamic programming algorithm that makes optimal suggestions to balance the load in electrical

grid systems given an input of previous years' data. The algorithm is compared with exhaustive search, the greedy algorithm and heuristic algorithms and it excels in terms of optimality and running time. Based on this, a more general load balancing algorithm with spatial consideration for electrical grid is developed.

For the grid computing systems, an interesting class of research topics is the optimal task partition and their mapping to different distributed computing machines with communication time that is nonlinear to the size of the transferring files. Grid computing systems are essentially distributed computing systems without workload dependencies on different machines and with internal communications. Thus, Divisible Load Theory (DLT) is a good match to the scheduling problems in grid computing systems. We developed a DLT-based method to optimally partition the computing load into fractions and map them to computing machines with nonlinear communication speed in the size of loads. Furthermore, two novel performance measurements for grid computing systems with multi-level tree networks are examined. One measure is utilization: the fraction of time processors are busy processing computational load. The other is progress: the percentage of load processed so far at a given time. A variety of scheduling policies are considered.

# Contents

# List of Figures

# Acknowledgments

It would not be possible to write this doctoral thesis without the help of so many people. Here I can only mention a small part of them.

First and foremost, I would like give my earnest thanks to my advisor, Prof. Thomas Robertazzi for supporting me these years, both financially and intellectually. He is the most easygoing and one of the smartest people I know. As an advisor, he is a role model that teaches me how to do research with patience, carefulness and hard work. He not only gave me freedom to manage the time and do research without pressure but also instructive suggestions when I faced difficulties.

I am grateful to Prof. Steven Skiena who taught me and gave me so many helpful discussions on phase balancing algorithm design. I also owe my gratitude to Prof. Eugene Feinberg for his guidance and Janos for his advice on the smart grid project.

I would also like to thank the two research groups I have worked with: Roy Fei, Eting Yuan, Muqi Li, Xunyi Zhang and Manasa Mandava in the Smart Grid lab, Zhongwen Ying, Zhe Shen, Li Geng, Zhemin Zhang and Yang Liu in the COSINE lab. We shared research and practical information and we had a great time as close friends.

At last, I would like to thank my amazing parents: Ying Wang and Aiping Liu. They give me unconditional support throughout my life and have cherished with me every achievement I have made.

# Chapter 1

# Introduction

In this thesis, load balancing and scheduling for two types of grids are introduced: electrical grids and computing grids. An electrical grid is a network used to transfer electricity and a computing grid is a collection of computing machines connected for a common computing task.

An electrical grid is comprised of electrical feeders in each part of which there are three phase lines. At any location along a feeder, each of these three phase lines have a certain amount of current and the variation of the amount of these current is defined as the phase unbalance. It is desired to have equal current at every point along all the feeders so that the phases could be fully utilized and current on phase lines could be more easily kept under the line capacity. Traditionally, the electricity companies send workers to adjust their transformers' phase assignments using workers' experience and intuition once a year. This greedy algorithm-like method is hardly able to reach the optimal assignment. As a result, electricity companies are introducing operational

methods for this problem. The goal is to reduce the unbalance on feeders and make the smallest change to the transformers to relieve the workers' burden and minimize the power cut off time for residents. To exhaustively search an optimal solution to the phase balancing problem is not possible because of the large number of customers. As a result, researchers used linear/integer programming method and heuristic algorithms to solve this problem. However, both of these two methods have their disadvantages: the linear/integer programming method is not able to use nonlinear objective functions and heuristic algorithms are not able to guarantee their solutions' optimality. In this thesis, we introduce a dynamic programming algorithm to solve phase balancing problem. It produces optimal solution and it has a reasonable running time.

A grid computing system is a loosely coupled and geographically dispersed distributed system with (usually) non-interactive workloads. As the system receives a computing task, it needs to partition and allocate the partitioned fractions of loads to different computing machines. The scheduling problem is to find the best arrangement to minimize the total running time. Here, we focus on the "divisible loads" which has no dependency and can be partitioned into arbitrary size. Also, we suppose the computing time for a task on a specific machine is known or can be estimated before running it. The divisible load model has a large number of applications including digital image processing, banks, insurance companies and etc. For divisible loads scheduling, Divisible Load Theory (DLT) is a good theory because it is tractable and scalable. The linear divisible load model has been studied

for over a decade with various network topologies. Divisible loads with nonlinear running time has been investigated recently. However, divisible loads with nonlinear communication time has not received that much researchers' attention. In this thesis, we introduce a DLT-based algorithm to solve this problem. Meanwhile, in the field of performance evaluation for grid computing system, several performance measures have been considered such as makespan (finish time), speedup and isoefficiency. In this thesis, we propose two novel performance measures: progress and utilization. These two performance measures are useful for real time data processing, scheduling policy comparative evaluation and resource allocation.

# Chapter 2

# A Dynamic Programming Algorithm for Phase Balancing Problem

## 2.1 Introduction

Over the past 15 years, research has been conducted on three phase feeder balancing. Phase balancing aims to reduce the unbalance of loads on three phases which can bring severe voltage drops in the feeders. The majority of electric power systems utilize, in the electric distribution system, feeders which carry three phases of alternating current/voltage. It is desirable for electric utilities and providers of electric power distribution systems to have approximately equal loads on each phase. This is a problem as even if loads are initially balanced, with time loads increase, decrease, are added or

removed from each phase, causing an unbalance of loads. Even during the same day there may be much variation of load on each phase of a feeder. There are two major phase balancing methods: there is feeder reconfiguration at the system level and there is phase swapping at the feeder level [2]. Phase swapping is not as well studied in the electric power literature as feeder reconfiguration. This chapter is about phase swapping algorithms.

Why does one wish phases to be in balance? Phase unbalance can limit the amount of power transferred on a feeder as on an unbalanced feeder one phase may reach its maximum carrying capacity measured in amperes (i.e. ampacity) while the other two phases are then underutilized and unable to carry their full or even nearly their full amount of current. This is poor utilization of the existing power distribution network and may result in unnecessary feeder expansion and upgrades which raise utility costs. Because one phase may be near its maximum ampacity, phase unbalance can also lead to preventive breaker/relay tripping and shutdown of a feeder whose restoration also involves a cost to the electric utility.

Periodically crews rebalance feeders. This can be done during periods of maintenance or restoration. One suburban Northeast U.S. utility rebalances feeders if the percentage of unbalance exceeds 15%. Generally it takes 10 to 15 minutes to switch a load so the overall job may take an hour plus travel time to the location. Work by a crew of two employees can cost several hundred dollars. However preparatory work such as scheduling can bring the total cost to several thousand dollars for one tap change. Three factors are considered in making a decision to rebalance a feeder: the monetary

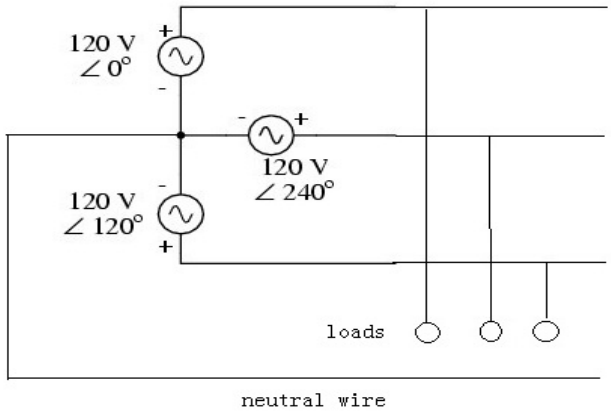


Figure 2.1: Three phase wiring diagram

cost of making the tap change(s), the expected increase in feeder balance (saved energy) and the temporary interruption of power to the customer. Tap change generally fall into two situations: a new customer is to be connected or the phase balance for existing feeders has become significantly unbalanced. Once a feeder is re-balanced it will initially be in balance but drift into unbalance as time goes on.

Even in more limited electric power systems, the same problems may arise. For instance *Gaffney* [3] reports problems with effective phase balancing in electric power systems in the tactical battlefield environment, largely because of insufficient operator training and experience. *David* [4] [5] proposes automatic phase balancing but does not propose an algorithm for this purpose.

The variables in the phase balancing problem are the phases each load is connected to and the goal is to minimize the degree of unbalance on feeders.

Many algorithms have been used to solve phase balancing problem. The original work is *Zhu*, *Chow* and *Zhang*'s mixed-integer programming in 1998 [1], but this algorithm has a drawback that the objective functions can only be linear. In 1999, to expand to nonlinear objective functions, *Zhu*, *Bilbro* and *Chow* introduced simulated annealing [2]. In 2000 and 2004, *Chen* and *Cherng* and *Gandomkar* applied a genetic algorithm to the problem [6] [7]. In 2005, *Lin*, *Chen*, et. al adopted a heuristic greedy algorithm [8]. In 2007, *Huang*, *Chen*, *Lin*, et. al used an immune algorithm to solve the problem [9]. These heuristic algorithms can get near-optimal solution quickly but can not guarantee optimal solutions.

Many combinatorial optimization problems have no known efficient algorithms capable of always producing optimal solutions. For those problems that computer scientists have been shown to be $NP$-complete, there is convincing evidence that no correct, efficient algorithms can exist. An efficient algorithm for any one of the hundreds of known $NP$-complete problems would imply efficient algorithms for all of them, implying that all are equally hard to compute.

The phase balancing problem we describe in this chapter can readily be shown to be equivalent to integer partitioning, a well-known $NP$-complete problem. Thus an efficient algorithm for phase balancing which always produced optimal solutions would imply efficient algorithms for all problems in $NP$, which computer scientists considered extremely unlikely. However heuristic algorithms that produce near optimal solutions with reasonable efficiency are possible, and are often developed for this purpose. [10]

In this chapter, five algorithms are applied to the phase balancing problem and a comparison is made between those five methods: Exhaustive search, Greedy Algorithm, Simulated Annealing, Genetic Algorithm and Dynamic Programming. All the algorithms use the same variables and objective function so that a comparison can be made.

We purposefully did not consider particle swarm optimization (PSO) and differential evolution (DE) algorithms in this chapter. We note that these methods are most appropriate for complex problems with ill-defined search spaces, as opposed to classical combinatorial optimization problems like ours, which is essentially a variant of the knapsack problem.

The key lesson of this chapter is that heuristic techniques such as simulated annealing and genetic algorithms (and DE and PSO) are superseded by the dynamic programming and combinatorial search methods we employ, which give optimal results instead of heuristic ones. Our new dynamic programming algorithm gives optimal results in reasonable time.

## 2.2 Objective function and stopping criteria

There are various kinds of objective functions such as cost functions in [2] and the loss function in [6]. Also, loads could be connected to two or three phases. Here we consider that all the loads are connected to single phase. The load range is set as integers between 1 and 100. Larger loads range can be scaled to this range. In this test, the objective function is the phasing unbalance index ($PUI$) which is used in many phase balancing papers [9] [8]

[12]:

$$PUI = \frac{Max(|I_a - I_{avg}|, |I_b - I_{avg}|, |I_c - I_{avg}|)}{I_{avg}} * 100\% \qquad (2.1)$$

Here, $I_a$, $I_b$ and $I_c$ are the total current on phase $a$, $b$ and $c$. $I_{avg}$ is the mean value of the current on each single phase.

For greedy and heuristic algorithms, the stopping criteria is: when the objective value reaches 1/500 times of the initial value. Other stopping criterias are of course possible.

## 2.3 Exhaustive search and a backtracking algorithm

For $n$ loads, each load is connected to one phase throughout the chapter. Since there are $n$ loads, and each load can be on one of three phases, there are $3^n$ ways to assign the loads to different phases. Under exhaustive search we calculate minimum objective functions for any potential number of tap changes. One then selects the solutions which satisfy the stopping criteria and finds the minimum number of tap changes among them.

Here we present a backtracking algorithm which can obtain an optimal solution as exhaustive search does, but has a smaller computational complexity [10]. Suppose one wants the most balanced solution using at most $t$ tap changes. This can be done with backtracking in $O((2n)^t)$ since each tap change has two phase choices, which is better than the exhaustive search for

small $t$. In this solution, the state space will be a vector of length $t$: The candidates for the $i$th position will be the possible tap changes greater than the last one in terms of load indexes.

The pseudo code appears below. Here $S_k$ is the set of candidate nodes in the decision tree for $k$ tap changes:

Backtrack - DFS(A,k)

if $A = (a_1, a_2, \cdots, a_k)$ is a solution, report it.

else

$k = k + 1$

compute $S_k$

while $S_k \neq \oslash$ do

$a_k =$ an element in $S_k$

$S_k = S_k - a_k$

Backtrack - DFS(A,k)

## 2.4 Greedy Algorithm

### 2.4.1 Greedy Algorithm

A greedy algorithm is any algorithm that finds a local optimal solution at every step. It gives a global optimal solution to many problems, but not all problems. It does not give a globally optimal solution to the phase balancing problem.

### 2.4.2 Greedy Algorithm for Phase Balancing

The steps are:

1. Input the loads and initial phases.

2. Calculate total loads on each phase.

3. Select one load from the phase with largest total load and move it to the phase with smallest total load. The load is selected so it can minimize the difference of the total loads on those two phases.

4. Calculate the objective value and see if it satisfies the stopping criteria. If yes, finish. If not, return to step 3.

### 2.4.3 Results

In figure 2.2, 100 randomly generated loads and phases are used for testing. The figures are the average of 300 runs. The horizontal axis is the number of tap changes the program needs and the vertical axis is the probability they appeared in 300 runs.

## 2.5 Simulated Annealing

### 2.5.1 Simulated Annealing Algorithm

The intuition behind the simulated annealing algorithm comes from the process of molten metals. The system is slowly cooled in order to achieve its lowest energy state. The basic idea of the method is that, in order to avoid being trapped in local minima, the algorithm usually accepts a "move" to a

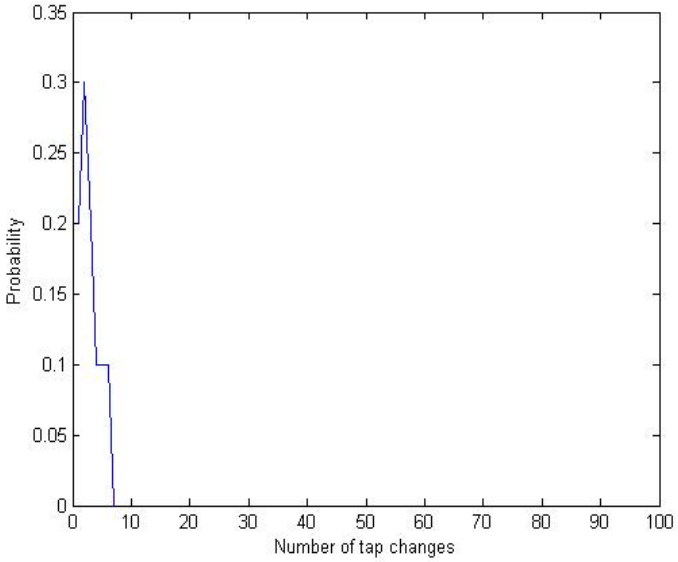


Figure 2.2: Probability VS Number of tap changes for the greedy algorithm

better solution but occasionally accepts a "move" that worsens the objective function with probability of:

$$Prob_{Accept} = e^{-\triangle E/kT} \tag{2.2}$$

Here e is the irrational number (=2.71828...). $\triangle E$ is the change between the objective values for two different solutions, $k$ is a constant relationship between temperature and energy, and $T$ is "temperature".

Simulated annealing is applicable to problems where one solution can be transformed into another by a "move" and there is an objective function available for evaluating the quality of a solution.

### 2.5.2 Simulated Annealing Algorithm for Phase Balancing

The steps are:

1. Input the loads and initial phases. Calculate the objective value.

2. Randomly select one load, move its phase to a randomly selected phase. Calculate the objective value.

3. Calculate the difference between the values in previous two steps. If the difference is negative, accept the "move". If not, accept the move with the probability in equation 2.

4. Repeat step 2 until it meets the stopping criteria.

### 2.5.3 Results

In figure 2.3, 100 randomly generated loads and phases are used for testing. The horizontal axis is time axis, the vertical axes are objective values and numbers of tap changes. It can be seen that beyond a certain number of tap changes there is only a minimal improvement on the objective function (law of "diminishing returns").

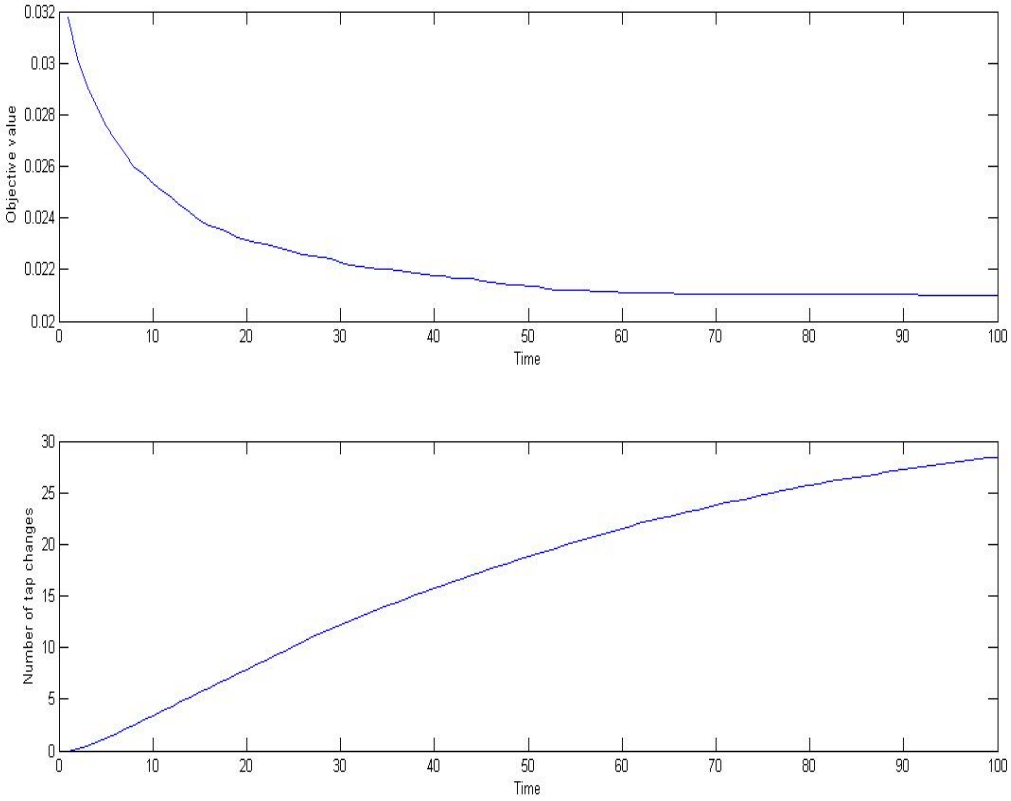


Figure 2.3: Probability VS Number of tap changes for the simulated annealing

## 2.6 Genetic Algorithm

### 2.6.1 Genetic Algorithm

Genetic algorithms belong to a larger family of algorithms known as evolutionary algorithms. They apply concepts from the theory of biological evolution, such as natural selection, reproduction, genetic diversity and prop-

agation, species competition/cooperation, and mutation, to search and optimization problems.

A genetic algorithm starts from the initial population (initial phases of the loads from some random solution) which are represented by a string of binary numbers. In each generation, crossover, mutation and selection are applied to the population in order to converge to the best solution.

### 2.6.2 Coding

In $GA$, a population is a set of solutions (chromosomes) for the objective function. In the population, the variables are encoded by use as binary numbers. For phase balancing problem, 2 bits are considered since one needs to represent 3 phases. "00", "01" and "10" represent phase 1, 2 and 3 respectively. After mutation or crossover, "11" will be changed to "00", "01" and "10" with equal probability if it is generated.

### 2.6.3 Objective function

In the $GA$ process, the value of the objective function mirrors the property of a solution. Better solutions have larger objective values.

### 2.6.4 Crossover

The crossover operator will proceed as follows. A crossover point is selected randomly for each of two solutions. A crossover probability is then invoked

($P_c$ from 0.6 to 0.8) to decide whether to make a swap of bits. An example is shown as follow:

String 1: 110|**10011**

String 2: 101|**01110**

Crossover point

String 1: 101|**01110**

String 2: 101|**10011**

### 2.6.5 Mutation

The mutation operation is implemented by randomly selecting any binary bit with a prespecified probability (about 0.01) and reversing it. The purpose of mutation in $GA$s is preserving and introducing diversity. Mutation should allow the algorithm to avoid local minima by preventing the population of chromosomes from becoming too similar to each other, thus slowing or even stopping evolution.

An example is shown as follow:

Before: 110**1**0011

Mutation point

After: 110**0**0011

### 2.6.6 Selection

The selection operator creates new populations or generations by selecting individuals from the old population. The selection is probabilistic but biased

towards the best solutions as special deterministic rules are used. In the new generations created by the selection operator, there will be more copies of the best individuals and fewer copies of the worst. A common technique for implementing the selection operator is the roulette wheel approach.

In this process, the individuals of each generation are selected for survival into the next generation according to a probability value proportional to the ratio of individual fitness (i.e. value of objective function) over total population fitness; this means that on average the next generation will receive copies of an individual in proportion to the importance of its fitness value.

### 2.6.7 Results

As shown in figure 2.4 and 2.5, 100 random generated loads and phases are used for testing. The figures are the average of 300 runs. In first graph, the horizontal axis is the number of generations for the genetic algorithm and the vertical axis is the corresponding objective value. In second graph, the horizontal axis is the number of tap changes the program needs and the vertical axis is the probability they appeared in 300 runs.

We also simulated an immune algorithm which is similar to genetic algorithm in the use of crossover, mutation and selection. It was found to perform similar to our genetic algorithm.

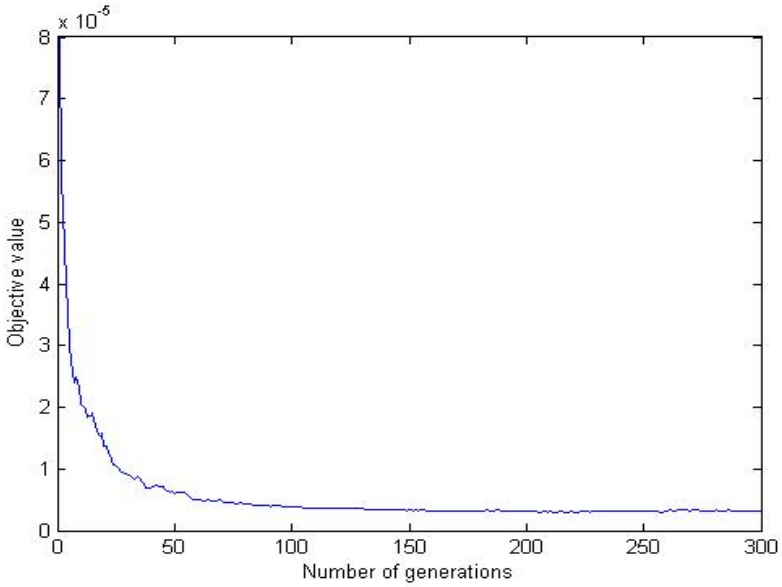


Figure 2.4: Fitness value VS Number of generations for a genetic algorithm

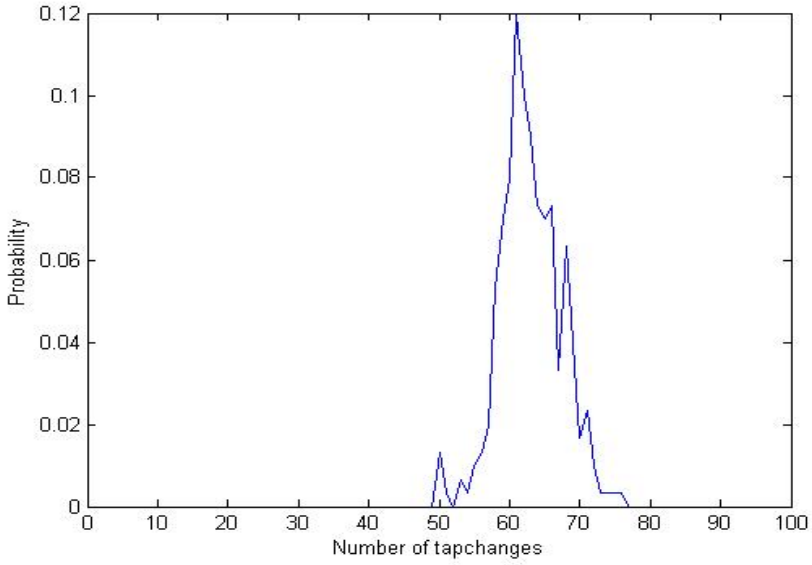


Figure 2.5: Probability VS Number of tap changes for a genetic algorithm

## 2.7 Dynamic Programming

### 2.7.1 A Dynamic Programming Algorithm to Solve the Phase Balancing Problem

An "optimal" algorithm for phase balancing is now presented. The phase balancing problem is $NP$-complete even with two phases and no cost per tap change, because it is equivalent to the integer partition problem and the integer partition problem is $NP$-complete. The hardness of integer partition depends upon large numbers, because it is not strongly $NP$-complete. For the phase balancing problem, the loads range between 1 and several thousand amperes. Assume that there are $n$ loads, where the $i$th load has weight $w_i$ and is currently assigned to feeder $l_i$. We assume the weights of all loads are integers, and the total load $T = \sum_{i=1}^{n} w_i$. As will be seen the algorithm runs faster with smaller $T$. Loads can be scaled to bring this about. The solution produced by the dynamic programming algorithm are optimal but it should be noted that the scaling is a source of approximation.

We present an algorithm which runs in $O(nT^2)$ to find the minimum number of changes to reach a particular quality criteria.

Denote the total load on phase $i$ by $L_i$. Because there are 3 phases, there are about $T^2$ sets of possible values for $L_1, L_2$, and $L_3$. This is as both $L_1$ and $L_2$ are integers between 0 and $T$, and $L_3 = T - L_1 - L_2$, $L_3$ would be specified after one has $L_1$ and $L_2$.

The algorithm will enumerate all possible partitions of $T$ into $L_1, L_2$, and

$L_3$, and in particular for each such partition $P$ find way to move from the current state to $P$ using the fewest number of changes. One can evaluate each of these $O(T^2)$ partitions according to the objective function, eliminate all which are not good enough, and then find the minimum cost good-enough transformation.

Define $C[x, y, i]$ to be the minimum cost (in terms of number of moves) to realize a balance of $L_1 = x, L_2 = y$ and implicitly $L_3 = T - L_1 - L_2$ after reassignments to the first $i$ loads (from 1 to $i$).

We define the following recurrence relation:

$$C[x, y, i] = Min[C[x-l_i, y, i-1]+t(i, 1), C[x, y-l_i, i-1]+t(i, 2), C[x, y, i-1]+t(i, 3)] \tag{2.3}$$

Here $t(i, \phi)$ is the cost of moving the $i_{th}$ load to phase $\phi$. $C[x, y, i]$ is the minimum number of tap changes to move from the initial loads to $[x, y, T_i - x - y]$.

$$T_i = \sum_{j=1}^{i} L_j \tag{2.4}$$

If $i_{th}$ load stays on phase $\phi$

$$t(i, \phi) = 0 \tag{2.5}$$

If $i_{th}$ load leaves phase $\phi$

$$t(i, \phi) = 1 \tag{2.6}$$

Assume the ith load is initially on line 1. Then the optimal solution either leaves load $i$ on line 1 (incurring no cost for the move), or moves it to line 2, or moves it to line 3 (both of which incur a cost of 1 operation). We need similar recurrences for the cases where load i is on line 2 or line 3. The basis of this recurrence is that $C[L_1, L_2, 0] = 0$, $C[x_0, y_0, 0] = \infty$ for all $x_0 \neq L_1$ and $y_0 \neq L_2$ (meaning no other states are achievable with zero moves).

Lastly, one calculates objective values for all $(T_n + 1)^2$ possible $[x, y, n]$ using equation 1 and get the minimum tap changes from $C[x, y, n]$. Thus, one does not need to calculate objective values for other $C[x, y, i]$, $i \in [1, n-1]$ which significantly reduces running time. In other words, the dynamic programming algorithm naturally produces a minimal solution for all number of tap changes desired, efficiently.

### 2.7.2 Results

As in figure 2.6, 100 randomly generated loads and phases are used for testing. The figure is the average of 300 runs. The horizontal axis is the number of tap changes the program recommends and the vertical axis is the probability they appeared in 300 times running.

## 2.8 Comparison

Two factors affect the results: the objective value and the number of tap changes. The objective value represents the unbalance of the loads on feeders. Meanwhile, each tap change costs some amount of money. So the aim is to

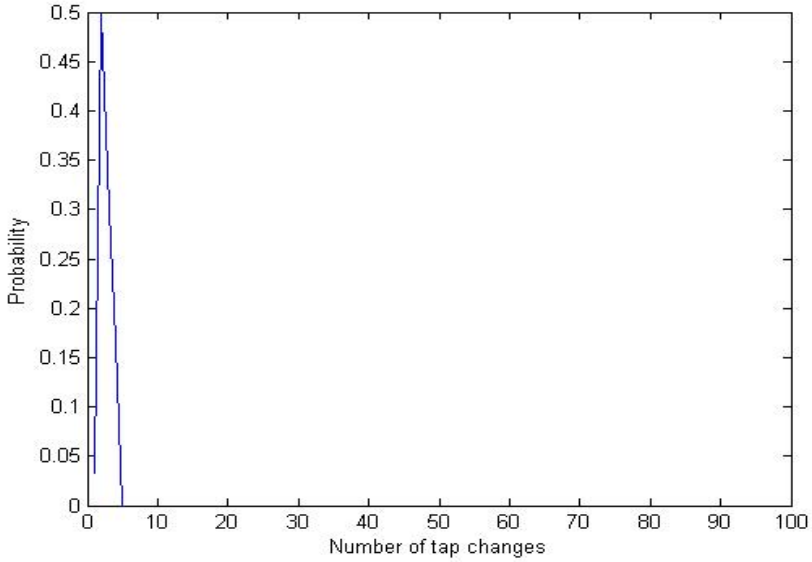


Figure 2.6: Probability VS Number of tap changes for the dynamic programming

get the desired objective value with a number of tap changes which is as small as possible.

Table 2.1 compares the performance of the six approaches in terms of running time and the number of tap changes for a single run of each program written in *Matlab*. Table 2.2 illustrates the performance improvement of each algorithm. The table shows wins ($W$), losses ($L$) and ties ($T$) of the algorithm in the first column compared to the algorithm listed for each column.

From the previous two tables, one can see that $DP$ and $GA$ are probably the better algorithms for the Phase Balancing problem. So to further compare the performance of $DP$ and $GA$, we have two tables. Table 2.3 and table 2.4 are two tables made from 20 examples of $DP$. Every example has

Table 2.1: Performance comparison

| Algorithms | Running Time (ms) |
|---|---|
| Exhaustive Search | 6642.97 |
| Backtracking Algorithm | 886.49 |
| Greedy Algorithm | 5.65 |
| Simulated Annealing | 1.45 |
| Genetic Algorithm | 72.15 |
| Dynamic Programming | 223.62 |

Table 2.2: Performance comparison 2

| Algorithm | DP | GA | Greedy | SA |
|---|---|---|---|---|
| DP | | 10W/40T | 42W/8T | 45W/5T |
| GA | 10L/40T | | 40W/6T/4L | 42W/4T/4L |
| Greedy | 8T/42L | 4W/6T/40L | | 40W/4T/6L |
| SA | 39L/11T | 4W/4L/42T | 6W/4T/40L | |

10 loads and the integer load range is 1 to 10. The columns differ in the number of tap changes. The rows are the different runs. The numbers in the table are the objective values. Table 2.5 is made from 20 examples of the genetic algorithm, every one with the same initial loads and phases as $DP$ has. The vertical axis is the number of runs. The first column holds the best objective values that the $GA$ obtained and the second column includes the corresponding number of tap changes the $GA$ needed for those objective values. From these two tables, one can see that in some cases the best objective values that the two algorithms can get are same and $DP$ needs less tap changes and other times $GA$ loses both in terms of the objective value and number of tap changes.

Another test as in figure 2.7 is a gathering of 30 examples. In each

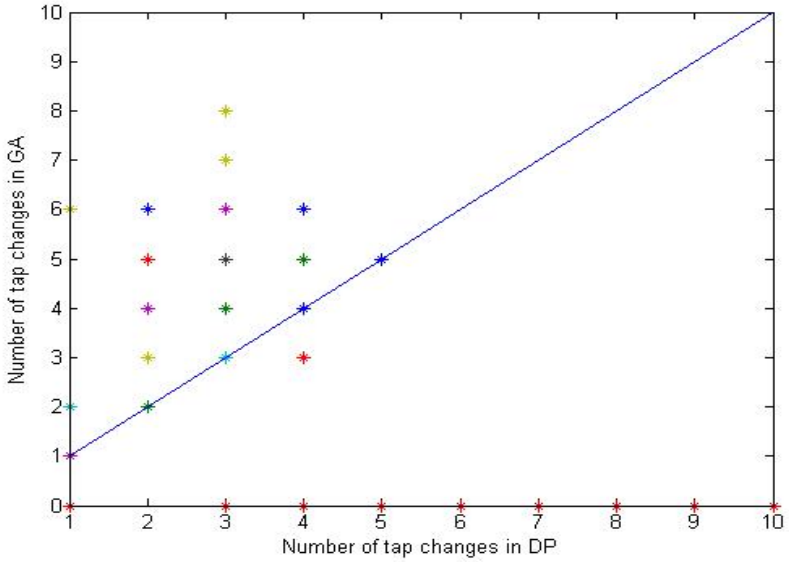


Figure 2.7: Comparision of number of tap changes between DP and GA

example, both the $DP$ and the $GA$ are give same set of loads and initial phases and then we record the result they give in terms of the number of tap changes. Every example has 10 loads and the integer load range is 1 to 10. $DP$ lost very few times but $DP$ gave better objective values in those examples.

Since most of the time $DP$ and $GA$ have the same performance in terms of the unbalance factor, but $GA$ needs a larger number of tap changes, it was desired to investigate the size of the loads those two algorithm transferred, that is, whether $GA$ transfers more loads with relatively smaller size and $DP$ transfers loads with relatively larger size. Figure 2.8 and figure 2.9 show the loads' size VS the frequency that were transferred. From figure 2.8 and figure 2.9, one can see that $DP$ is somewhat biased to larger values of

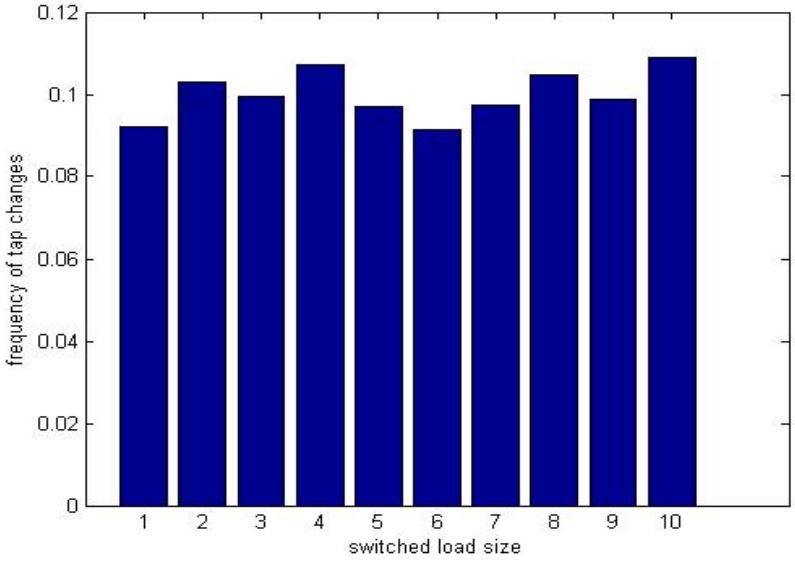


Figure 2.8: Tap change load size VS frequency for genetic algorithm

switched load size and the loads that $GA$ transfers are distributed evenly in load size because $GA$ transfers loads randomly but $DP$ transfers loads with less tap changes (less total load transfer). While most of the algorithms in the chapter were implemented in $Matlab$, running time of a faster $C$ version of the dynamic programming algorithm appears in Table 2.6.

## 2.9 Conclusion

Dynamic programming does the best in terms of performances though it is the slowest of the non-exhaustive algorithms. This is a very promising algorithm because of the algorithm's optimality. It was found that even though the genetic algorithm and dynamic programming produced solutions that were

Table 2.3: Objective function and number of tap changes for dynamic programming for ten runs

| Number of tap changes: | 1 | 2 | 3 |
|---|---|---|---|
| 1st Run | 0.2750 | 0.0500 | 0.0500 |
| 2nd Run | 0.0179 | 0.0179 | 0.0179 |
| 3rd Run | 0.0678 | 0.0169 | 0.0169 |
| 4th Run | 0.0500 | 0.0500 | 0.0500 |
| 5th Run | 0.2558 | 0.0465 | 0.0465 |
| 6th Run | 0.4769 | 0.1077 | 0.0154 |
| 7th Run | 0.1053 | 0 | 0.0526 |
| 8th Run | 0.1321 | 0.0755 | 0.0189 |
| 9th Run | 0.6154 | 0.2000 | 0.0154 |
| 10th Run | 0.1077 | 0.0154 | 0.0154 |

Table 2.4: Objective function and number of tap changes for dynamic programming for ten runs

| Number of tap changes: | 4 | 5 | 6 |
|---|---|---|---|
| 1st Run | 0.1250 | 0.2750 | 0.8000 |
| 2nd Run | 0.0714 | 0.1786 | 0.4464 |
| 3rd Run | 0.0678 | 0.0169 | 0.2203 |
| 4th Run | 0.0500 | 0.0500 | 0.0500 |
| 5th Run | 0.0154 | 0.0615 | 0.2462 |
| 6th Run | 0.0154 | 0.0615 | 0.2462 |
| 7th Run | 0.0526 | 0.2105 | 0.3684 |
| 8th Run | 0.0189 | 0.1887 | 0.4151 |
| 9th Run | 0.0154 | 0.1538 | 0.2462 |
| 10th Run | 0.0615 | 0.1077 | 0.2000 |

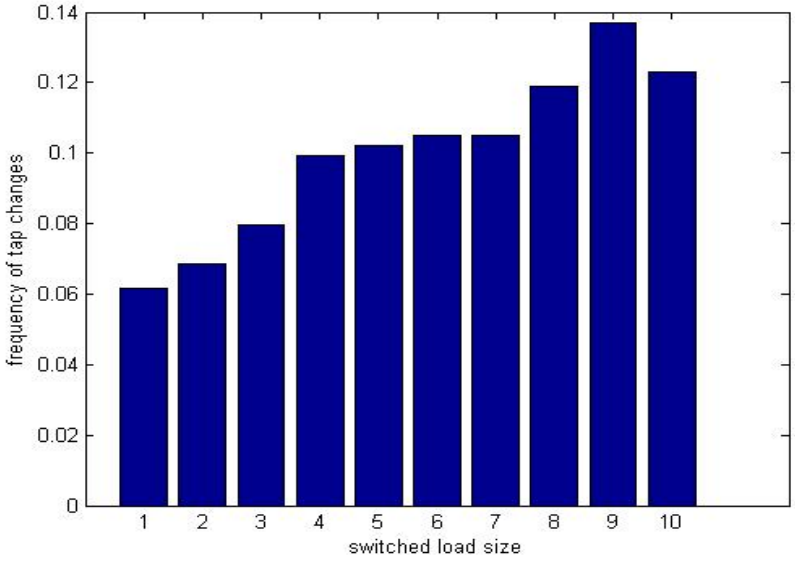


Figure 2.9: Switched load size VS frequency for dynamic programming

almost identical in terms of the unbalance factor to many significant places, the genetic algorithm can require many more tap changes than dynamic programming did (often by more than a factor of two). This suggests that the solution space contains a variety of optimal/near-optimal solutions that differ significantly in the number of tap changes. The genetic algorithm, at least as presently constituted, is not able to discern the best solution as well as the dynamic programming algorithm. The good news is that an optimal algorithm for phase balancing in dynamic programming is available with reasonable complexity ( $O(nT^2)$ ). This is an interesting problem from both the viewpoint of algorithms and an interesting power system application.

# Chapter 3

# A Dynamic Programming Algorithm for Spatial Phase Balancing Problem

## 3.1 Introduction

In the previous chapter, we introduced a dynamic programming algorithm to obtain the optimal solution for phase balancing problem in a reasonable running time. However, this algorithm has a shortcoming in that it only balances the whole feeder, but not every section along the feeder. This may lead to a situation where the three phase current is balanced at the beginning of the feeders, but not balanced at other positions of the feeders.

In this chapter, a dynamic programming algorithm is applied to solve the phase balancing problem along each part of the feeder. The computation

complexity of this algorithm is $O(nT^2)$ ($T$ is the sum of all the loads) in the worst case, and all operations we do are in linear time. A mathematical model, algorithm and objective functions are introduced (section 3.2). An optimal dynamic programming algorithm is discussed in detail (section 3.3). Simulation results appear in section 3.4. The conclusion is in section 3.5.

## 3.2 Problem and Algorithm Formulation

### 3.2.1 Overview

The algorithm we discuss in this chapter is a dynamic programming algorithm. Assume the feeder is linear and the generation input is at the left. In terms of an objective function we seek to minimize a weighted sum of the degree of imbalance of each section along the feeder for a given number of tap changes. Suppose we have $N$ loads on a linear feeder. To do this we create $N$ objective function matrices (one for each section) as well as $N$ cost matrices. For each matrix in both sets the (horizontal) rows correspond to potential load on phase $A$ and the (vertical) columns correspond to the potential load on phase $B$. Thus the $(i, j)$th entry of the $k$th objective function matrix is the objective function value with partial total load $i$ for phase $A$ for the first $k$ sections and with partial total load $j$ for phase B for the first $k$ sections. Implicitly the partial total load on phase $C$ is the total load on the first $k$ sections minus load $i$ and minus load $j$ for the first $k$ sections.

Also, the $(i, j)$th entry of the $k$th cost matrix is the minimum number of

Table 3.1: An example feeder with single phase loads

| Load size: | 2 | 1 | 2 | 1 |
|---|---|---|---|---|
| Phase: | A | B | C | A |

tap changes one has to use to achieve the corresponding objective function in the $(i, j)$th entry of the $k$th objective function matrix. How does one compute the $(i, j)$th entry in the $k$th objective function matrix? One knows the load on phase $A$ is $i$ for the first $k$ sections and the load on phase $B$ is $j$ for the first $k$ sections. Implicitly one then knows the load on phase $C$ is the total load for the first $k$ sections minus $i$ and minus $j$. So the $(i, j)$th entry is the absolute difference between the maximum of the loads on each phase minus the average load per phase for the first $k$ sections and this difference is divided by the average load per phase for the first $k$ sections. On the other hand, the $(i, j)$th entry of the $k$th cost matrix (which is the minimum number of tap changes to achieve the corresponding objective function value) is generated by a recursion that appears below. Note there are different recursions depending on whether loads are connected to one phase or two/three phases.

Once all of the matrices are generated, what is essentially a shortest path algorithm can be run from matrix to matrix where the distances are the objective function entry values. However in generating the possible paths thru the matrices there are some constraints on which entries in the $(k + 1)$st matrix an entry in the $k$th matrix can be connected to. For instance if one is at entry (4,5) with a load of 1 on phase $C$ in the 3rd objective function matrix, a path can connect it to entry (6,5) with a load of 1 on

phase $C$ in the 4th matrix if the 4th load is 2 (single phase load) but a path cannot connect it to (8,5) with a load of one on phase $C$ in the 4th matrix. Thus, unlike the *Dijkstra* shortest path algorithm we generate all possible feasible paths. However these constraints reduce the number of paths to be considered. Actually there is one more set of matrices that is generated as the recursion is run to record the associated paths for future use. This is done in $k$ path matrices, where using similar definitions of $i$ and $j$ as before, the $(i, j)$th entry of the $k$th matrix is a pointer to the position of its parent entry along a path in the $k-1$st objective function matrix.

To obtain a solution, one fixes as an input parameter to the algorithm the maximum number of tap changes allowed. Call it $M$. The last cost matrix is used to determine the set of solutions that meet this constraint. From the remaining solutions we can select the solution with the best objective function value. Once the best solution is selected one can retrieve the phase assignment from the corresponding path matrix.

Alternately we can create a table of the best solution for each specific number of tap changes. To create the $p$th row in the table one can run the steps of the previous paragraph, keeping only the solutions with p tap changes. Note that beyond a certain number of tap changes the objective function value of solutions tend to get worse.

This overview has been phrased in terms of minimizing a weighted sum of the degree of imbalance on each section. However the use of other objective functions using these techniques is certainly possible. A different one is discussed below. Note that the objective function and cost matrices may

be viewed as $N$ sets of two dimensional matrices, or each one as a three dimensional matrix. Another note is that an implementation of this dynamic programming algorithm can do without the objective function matrices since the objective function values can be computed from corresponding indexes of the cost matrices.

Another idea to save memory is to use a different notation to describe the statuses. Let L(i) be the load on each phase after the first i loads on the line, then We can change our state space from:

$C(i, j, k)$ – the cost to achieve a total load of $i$ on phase $a$ and total load of $j$ on phase $b$, after the first $k$ loads, leaving the total load on phase $c$ implicit.

to

$C(da, db, i)$ – the cost to achieve a difference (delta) load on phase $a$ from $L(i)$, a delta load on phase $b$ from $L(i)$ after the first $i$ loads, leaving the delta load on phase $c$ implicit.

The advantage is that this method can reduce the complexity and also we can delete the solutions with very large deltas. So one can save a great deal of memory, which otherwise would limit the scalability of this algorithm.

Figure 3.1 and figure 3.2 are examples of objective function values matrices and cost function matrices of a feeder with four single phase loads.

The dynamic programming algorithm is now outlined in more detail.

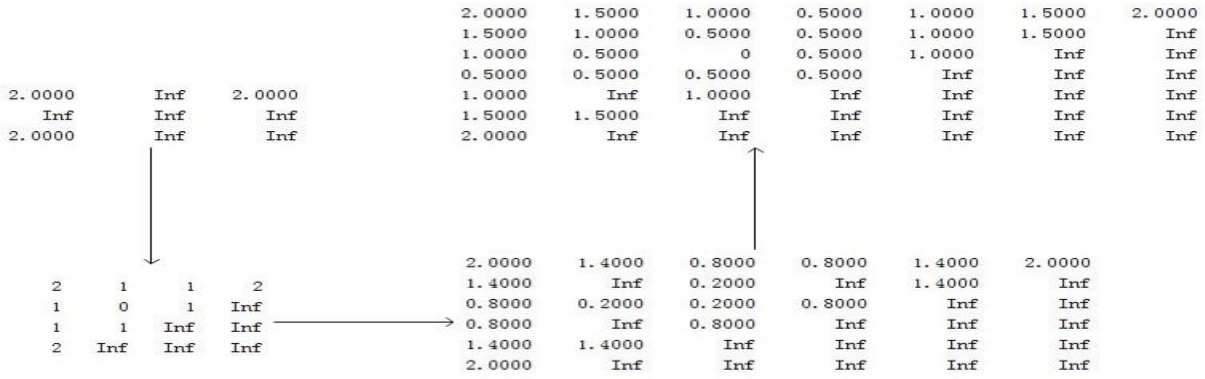

Figure 3.1: Example of objective function values matrices

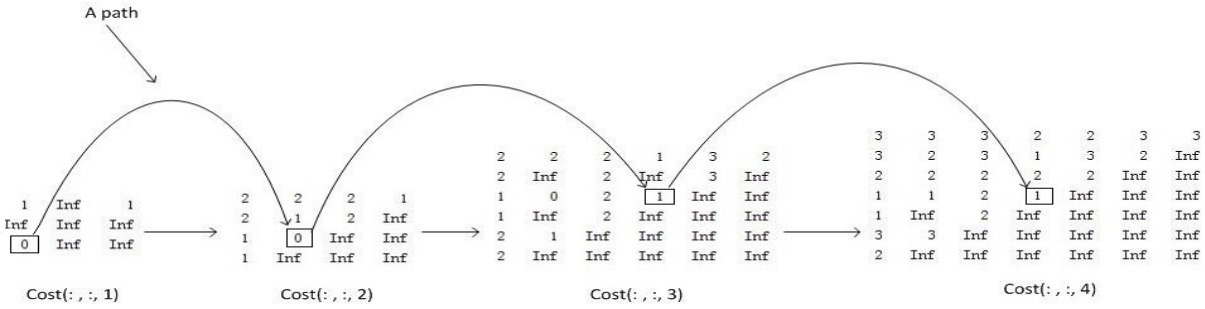


Figure 3.2: Example of cost matrices

### 3.2.2 Objectives

In phase balancing problem, there are three main objectives:

1. To avoid overloading.
2. To balance three phase current along the feeders.
3. Reduce number of phase changes to save labor cost.

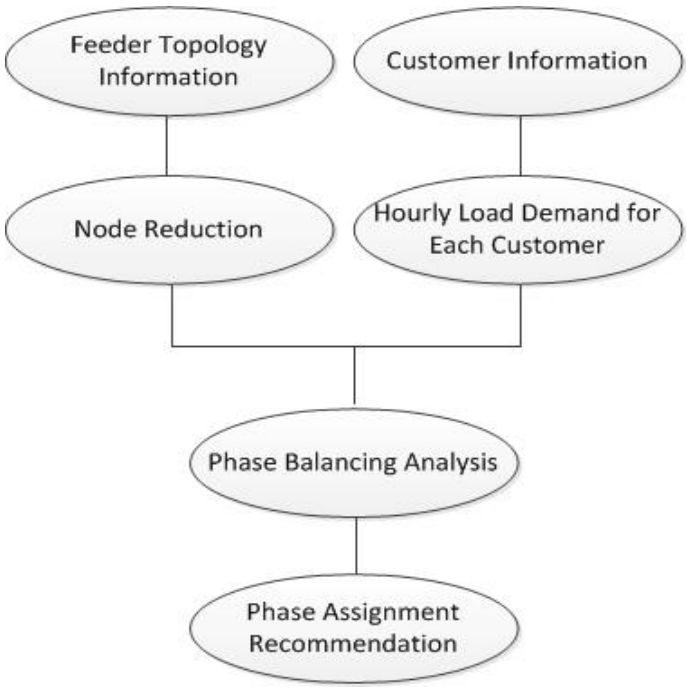


Figure 3.3: Overall structure for phase balancing

### 3.2.3 Overall structure

Figure 3.3 illustrates the overall structure for phase balancing of distribution feeders. One abstracts the node connection and hourly load demand for each node from feeder topology information and customer information. Based on this information, one can do a phase balancing analysis and give a phase assignment recommendation.

### 3.2.4 Sample feeder with connecting branches

Figure 3.4 shows a radial feeder configuration and the loads ($L_{i,1}$, $L_{i,2}$, $L_{i,3}$) at node i. In our model, the feeder is divided into nodes and sections. Here,

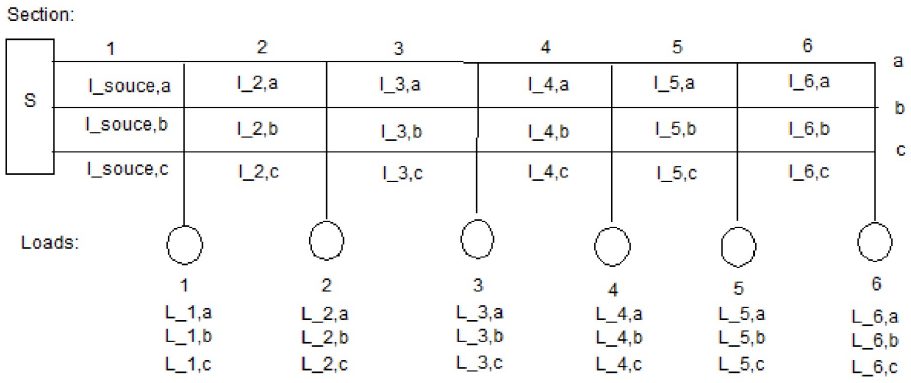


Figure 3.4: Sample feeder model

$I_{i,j}$ denotes the current on phase $j$ of section $i$. $L_{i,j}$ is the current (load) demand of node $i$ on phase $j$. The phase balancing's objective is to find the optimal phase assignment for each load to minimize the unbalanced flows at monitored sections with a certain number of tap changes which is smaller than the given maximum one.

### 3.2.5 Objective function

To balance the three phase flows along the whole feeder, one needs to balance the three phase flow in each section. From Kirchhoff's current law, one knows that the current on phase $\phi$ flowing out of section $j$ equals to the current on phase $\phi$ from the source minus the total current on phase $\phi$ of the first $j-1$ sections.

$I_{j,\phi} = I_{source,\phi} - \sum_{i=1}^{j-1} L_{i,\phi}$ for all $\phi$=a,b,c.

Here, $j$ is the section index and $i$ is the load index. $j \geq 2$.

Then the problem becomes to balance $\sum_{i=1}^{k} L_{i,a}$, $\sum_{i=1}^{k} L_{i,b}$ and $\sum_{i=1}^{k} L_{i,c}$

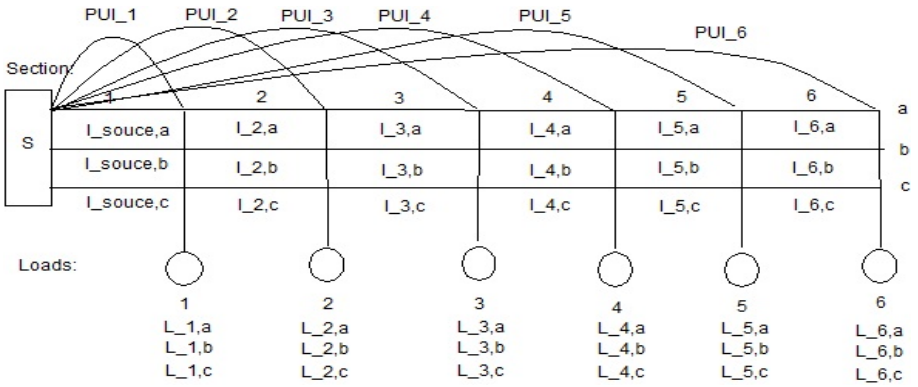


Figure 3.5: Phasing unbalance index

for $k = 1$ to $N$.

There are various kinds of objective functions such as cost functions in [2] and the loss function in [6]. In this chapter, the objective function is the phasing unbalance index ($PUI$) which is used in many phase balancing papers [9] [8] [12]:

$$PUI_i = \frac{Max(\left|I_{a,i} - I_{avg_i}\right|, \left|I_{b,i} - I_{avg_i}\right|, \left|I_{c,i} - I_{avg_i}\right|)}{I_{avg_i}} * 100\% \qquad (3.1)$$

Here, $I_{a,i}$, $I_{b,i}$ and $I_{c,i}$ are the total current loads on phase $a$, $b$ and $c$ of section $i$. $I_{avg_i}$ is the mean value of the current load on each single phase of section $i$. Considering the single phase loads case is a subset of the three phase loads case, we assume that all the loads are connected to three phases. The load range is set as integers between 1 and 100. Larger loads range can be scaled to this range.

Also, to avoid overloading, the current on each phase has to be smaller

than the line capacity.

In this chapter, two approaches are introduced.

The first approach is to limit all the $PUI$'s of each section under a certain threshold:

$$PUI_i <= threshold \qquad \text{for all i = 1 to N} \tag{3.2}$$

Here, the threshold can be set by operator or one can use binary search to find the minimum possible threshold.

The second approach is to minimize the weighted sum of phase unbalance indexes for each section of the feeder:

$$\text{Minimize} \sum_{i=1}^{i=N} w_i * PUI_i \tag{3.3}$$

Subject to:

$$w_i = \sum I_{\phi,i} \tag{3.4}$$

$$I_{\phi,i} \leq C_i \tag{3.5}$$

where

$\phi$ is one of three phases $a$, $b$ and $c$.

$i$ is the index of section from 1 to $N$.

$C_i$ is the phase line capacity of phase $j$ of section $i$.

### 3.2.6 Load type

There are two types of loads on the feeder: one phase loads and two or three phase loads. Loads on one phase feeders can only connect to one of the three phases. Loads on the two and three phase feeders can connect to two or three phases. That is: single phase loads have three tap change possibilities, two and three phase loads have six tap change possibilities.

The tabular below shows valid connection schemes for various types of laterals.

| Original phase | | Valid rephasing schemes | | |
|---|---|---|---|---|
| $3\phi$ | abc | abc | acb | |
| | | bca | bac | |
| | | cab | cba | |
| $2\phi$ | ab | ab* | ba* | a*b |
| | | b*a | *ab | *ba |
| | bc | bc* | cb* | b*c |
| | | c*b | *bc | *cb |
| | ac | ac* | ca* | a*c |
| | | c*a | *ac | *ca |
| $1\phi$ | a | a** | *a* | **a |
| | b | b** | *b* | **b |
| | c | c** | *c* | **c |

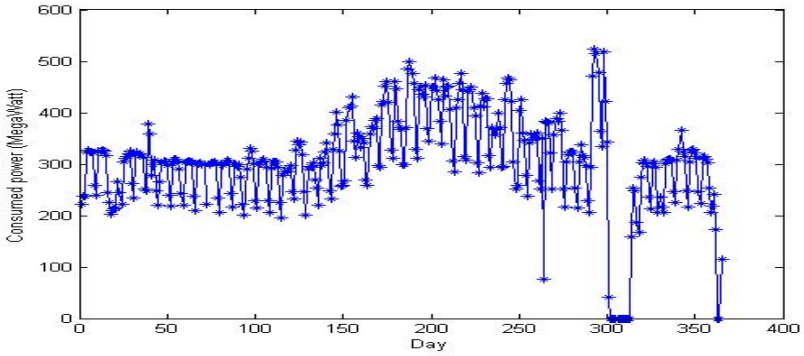


Figure 3.6: A yearly load profile

### 3.2.7 Load pattern

The customer hourly load data can be collected by the AMI (Advanced Metering Infrastructure) meters. However, it would be hard to do the computation if one makes phase balancing recommendation for the next year based on all hourly load data of the last year because of the large amount of data. So the evaluation of the load pattern is considered. One can do phase balancing analysis based on the load in the "peak time" or the "peak day" in the summer. Figure. 3.6 and 3.7 shows the load pattern obtained from one of $LIPA$'s substation's data. From the two load profiles, one can see the peak days of the year are in the summer and autumn and the peak hours of the day are in the afternoon.

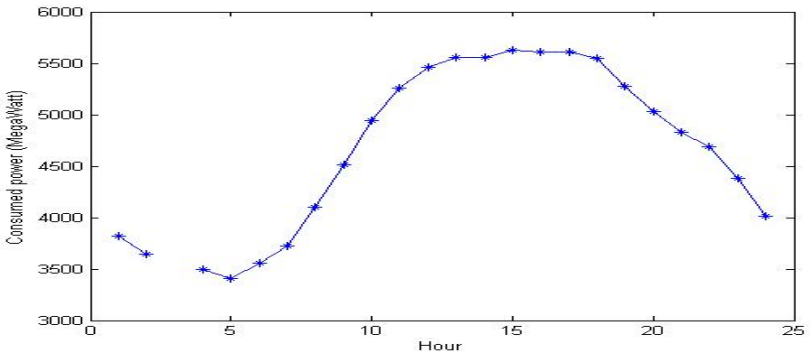


Figure 3.7: A daily load profile

## 3.3 A Dynamic Programming Algorithm to Solve the Phase Balancing Problem

### 3.3.1 Step1: Use recurrence to record number of tap changes

Define $C[x, y, i]$ to be the minimum cost (in terms of number of moves) to realize a balance of $L_1 = x, L_2 = y$ and implicitly $L_3 = T - L_1 - L_2$ after reassignments to the first $i$ loads (from 1 to $i$).

For the cost matrix for single phase loads we define the following recurrence relation:

$$C[x,y,i] = Min[C[x-l_i, y, i-1]+t(i,1), C[x+l_i, y-l_i, i-1]+t(i,2), C[x+l_i, y, i-1]+t(i,3)] \tag{3.6}$$

Here in the cost matrix, $l_i$ is the weight of $i$th load (single phase load),

$t(i, \phi)$ is the cost of moving the $i_{th}$ load to phase $\phi$. $C[x, y, i]$ is the minimum number of tap changes to move from the initial status to $[x, y, T_i - x - y]$.

$$T_i = \sum_{j=1}^{i} L_j \tag{3.7}$$

If $i_{th}$ load stays on phase $\phi$

$$t(i, \phi) = 0 \tag{3.8}$$

If $i_{th}$ load leaves phase $\phi$

$$t(i, \phi) = 1 \tag{3.9}$$

Assume the $i$th load is initially on phase $a$. Then the optimal solution either leaves load $i$ on phase $a$ (incurring no cost for the move), or moves it to phase $b$, or moves it to phase $c$ (both of which incur a cost of 1 operation). We need similar recurrences for the cases where load i is on phase $b$ or phase $c$. The basis of this recurrence is that $C[L_1, L_2, 0] = 0$, $C[x_0, y_0, 0] = \infty$ for all $x_0 \neq L_1$ and $y_0 \neq L_2$ (meaning no other states are achievable with zero moves).

For two and three phase loads, suppose the $i$th load has three single phase loads $l_{i,a}, l_{i,b}, l_{i,c}$ and they are initially on phase a, b and c. We define the following recurrence relation:

$$
\begin{aligned}
C[x,y,i] = Min[&c[x-l_{i,a}, y-l_{i,b}, i-1], c[x-l_{i,a}, y-l_{i,c}, i-1]+1, \\
&c[x-l_{i,b}, y-l_{i,a}, i-1]+1, c[x-l_{i,b}, y-l_{i,c}, i-1]+1, \\
&c[x-l_{i,c}, y-l_{i,a}, i-1]+1, c[x-l_{i,c}, y-l_{i,b}, i-1]+1]
\end{aligned} \tag{3.10}
$$

### 3.3.2 Step 2: Record the "Path"

In last subsection, when calculating the number of tap changes for each $[x, y, i]$, one needs to create a three dimensional path matrix (since $C$ is three dimensional) to record what is the "parent" of a status $[x, y, i]$. That is, to record the parent's position of $[x, y, i]$ as a cell. With all the record of these relationships, one can know the paths.

### 3.3.3 Step 3: Calculate objective values

After using the recurrence to record the path, one calculates objective values for all $(T_i + 1)^2$ possible $[x, y, i]$ using objective function for all $i \in [1, n-1]$. Then, to take the imbalance of each section into consideration, one can calculate the weighted sum objective values for each path.

### 3.3.4 Step 4: Avoid the overload

One needs to delete the solutions that cause overload on the feeders by setting the positions that have indexes larger than the line capacity to infinity. The deletion simply removes the incoming or outgoing edges to these nodes. That

is to make sure that all three phase currents in each section is smaller or equal to the line capacity.

### 3.3.5 Step 5: Make phase assignment recommendation

For the first approach, if we have a threshold of "what is balanced enough", then we can delete any partial solution that is not "balanced enough". If there is a solution remaining, it would be found by any path from an end state to a state that passes through "balanced enough" vertices. If we do not have a threshold but want to find the path with the minimum balance, do a binary search on the possible thresholds. Repeatedly pick a possible threshold in the middle of the range of possible thresholds. Delete all vertices more unbalanced than this. Look for a path in the remaining graph. If we find one, try a smaller threshold. If not, try a larger one.

For the second approach, consider now that one has three matrixes: the number of tap changes (cost) matrix $C[x, y, N]$, the path matrix and objective values matrix $Objv[x, y, N]$. Then one can make a table with $N$ rows and three columns. The first column is the maximum number of tap changes allowed to make. The second column is the corresponding best objective value one can get, this can be obtained by searching all the $x$ and $y$ in $Objv[x, y, N]$ that satisfies $C[x, y, N]$ = maximum number of tap changes allowed. The third is the corresponding phase assignment for each load which can be obtained by retrieving the path. From this table, one can make phase assignment recommendation provided the desired number of tap changes or

objective values.

### 3.3.6 An iterative method to balance tree network feeders

For tree network feeders, one can use the dynamic programming algorithm above to balance each subtree feeder and take every subtree feeders as equivalent nodes in the upper level of the tree. One can balance the whole system using this bottom-up method.

For example, one can use the algorithm to balance three phase current of IEEE sample feeder (figure. 3.8). This feeder contains several branch feeders and they form a tree network. One can divide it into five groups: node 632, 645 and 646 as group $A$, node 633 and node 634 as group $B$, node 692 and node 675 as group $C$, node 611,684 and 652 as group $D$ and node 671 and 680 as group $E$. One can first balance group $A$ and $B$ and take them as one nodes. Then balance group $C$, $D$ and $E$ and take them as the second node. At last, balance those two equivalent nodes.

## 3.4 Simulation

### 3.4.1 Implementing a 20 node feeder

Here, a feeder with 10 randomly generated loads and phases is tested. Table 3.2 and Table 3.3 show the phase assignment for each load before and after phase balancing. Figure 3.9 and 3.10 show the three phase current for each

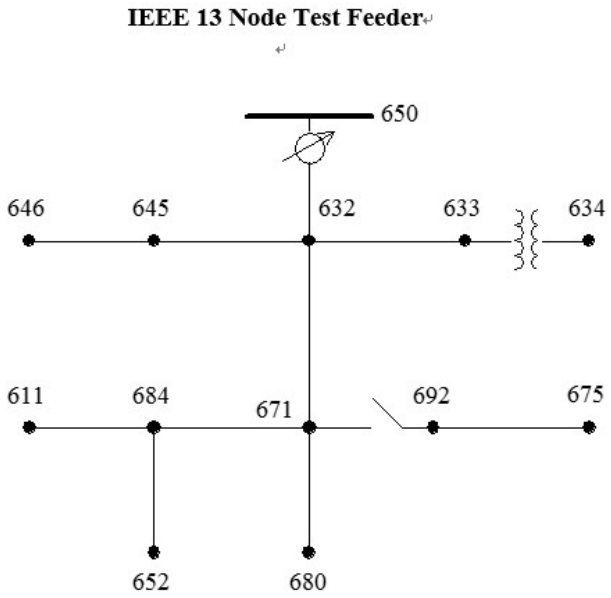


Figure 3.8: IEEE sample feeder with 13 nodes

section before and after phase balancing respectively. Figure 3.11 shows the corresponding objective values. Note that in the objective values at the end of the curves in figure 3.11 are worse because there is less flexibility in making tap changes at that point. In fact, no tap changes were made for the last two loads in the example.

### 3.4.2 Running time and required memory

To illustrate the running time and required memory (without the memory reducing trick mentioned in the overview), randomly generated loads and phases are used for testing. In figure 3.12 and 3.13, the horizontal axis is the number of loads and the vertical axes are running time in $ms$ and allocated

Table 3.2: Feeder before phase balancing

| Load index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| a: | 0 | 5 | 1 | 2 | 7 | 6 | 10 | 3 | 9 | 0 |
| b: | 0 | 2 | 7 | 0 | 0 | 0 | 0 | 6 | 0 | 2 |
| c: | 5 | 0 | 10 | 0 | 0 | 7 | 3 | 0 | 3 | 6 |
| Total on a: | 0 | 5 | 6 | 8 | 15 | 21 | 31 | 34 | 43 | 43 |
| Total on b: | 0 | 2 | 9 | 9 | 9 | 9 | 9 | 15 | 15 | 17 |
| Total on c: | 5 | 5 | 15 | 15 | 15 | 22 | 25 | 25 | 28 | 34 |

Table 3.3: Feeder after phase balancing

| Load index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| a: | 0 | 5 | 1 | 2 | 7 | 6 | 0 | 3 | 9 | 0 |
| b: | 0 | 2 | 10 | 0 | 0 | 0 | 10 | 6 | 0 | 2 |
| c: | 5 | 0 | 7 | 0 | 0 | 7 | 3 | 0 | 3 | 6 |
| Total on a: | 0 | 5 | 6 | 8 | 15 | 21 | 21 | 24 | 33 | 33 |
| Total on b: | 0 | 2 | 12 | 12 | 12 | 12 | 22 | 28 | 28 | 30 |
| Total on c: | 5 | 5 | 12 | 12 | 12 | 19 | 22 | 22 | 25 | 31 |

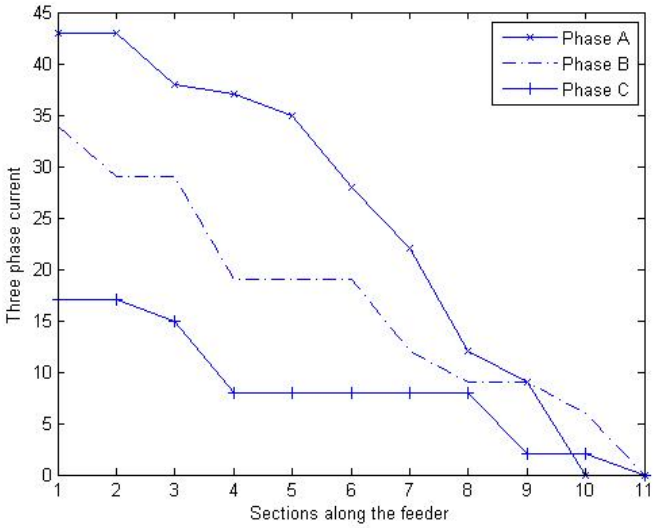


Figure 3.9: Three phase current along the feeder before phase balancing

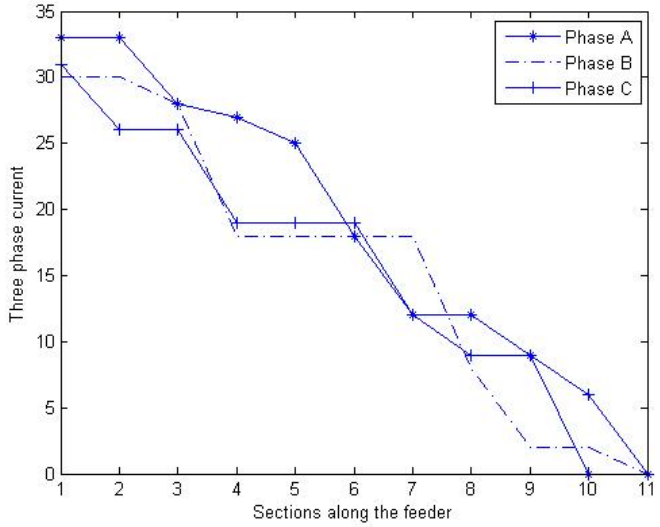


Figure 3.10: Three phase current along the feeder after phase balancing

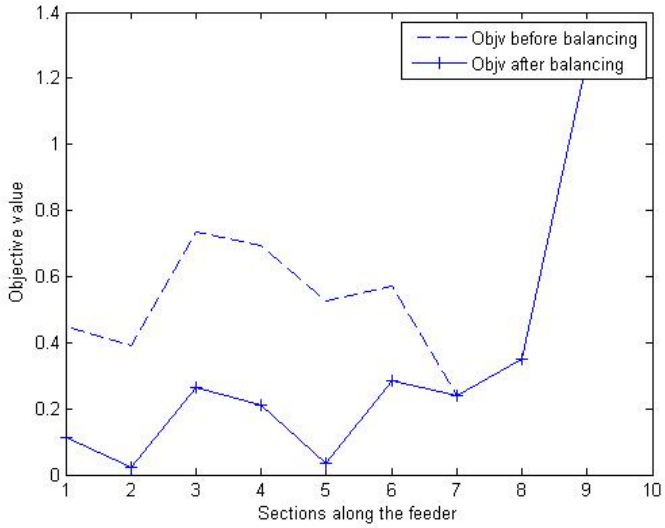


Figure 3.11: Objective values before and after phase balancing

memory in *bytes*.

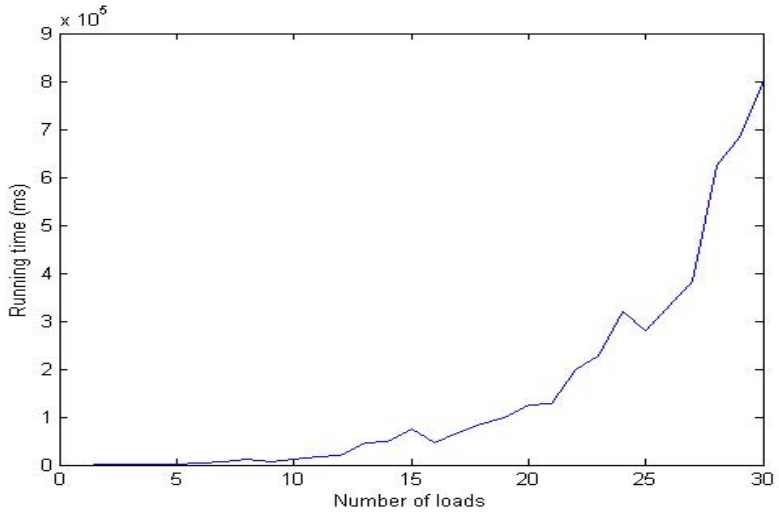


Figure 3.12: Running time VS Number of loads

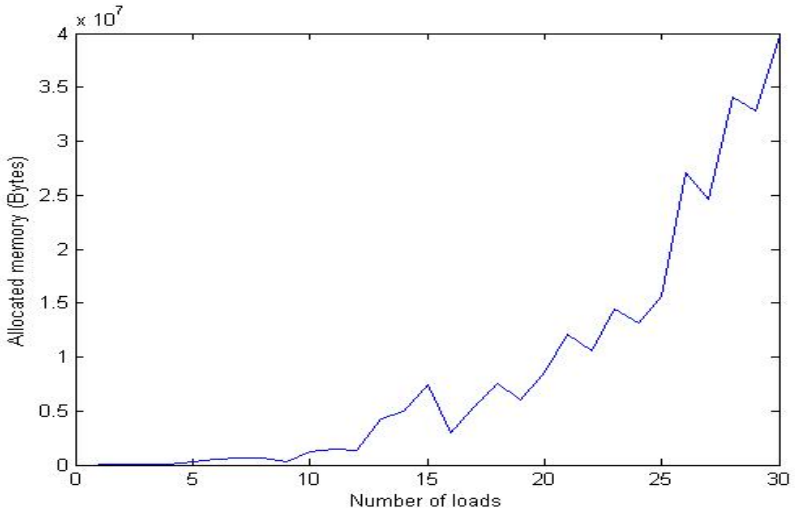


Figure 3.13: Running time VS Allocated memory

## 3.5 Conclusion

Of all the algorithms examined in our earlier work [11], dynamic programming was the most promising in its ability to provide optimal solutions with-

out using an exhaustive search approach. This chapter has examined and discussed dynamic programming in much greater detail and also adapted it to include a consideration of spatially distributed loads. Many variations on the basic approach described here are possible. This includes the use of different objective functions and data structure implementation of the algorithm. Most significantly the use of dynamic programming allows a better quality combinatorial solution at much less the cost of an exhaustive search.

# Chapter 4

# Scheduling Divisible Loads With Nonlinear Communication Time

## 4.1 Introduction

For decades it has been realized that many algorithms of much practical interest have a computational time that is a nonlinear in the size of the algorithms input. This includes algorithms used in aerospace applications such as the fast Fourier transform, matrix operations, line detection using the Hough transform [13] and pattern recognition using 2D Hidden Markov models.

But a related question that has received much less attention is whether the transmission time of data moving over links between processing nodes

can be nonlinear in the size of the data to be transmitted. Normally one would think this is not possible. If one doubles the amount of data to be transmitted one would think it should take twice as much time to transmit as half that amount of data. This intuition is based on the usual linear intuitive model of a channel. Naturally we are ignoring overhead such as packet headers in this consideration.

However there is another way of looking at things: indexing data transmission not by time but by data structural properties. For instance, if one transmits a square matrix and indexes data transmission by matrix (i.e. row/column) size, the transmission time is proportional to a square power of the size of the matrix. Alternately if one transmits a binary tree of data where each node holds x bytes and indexes data transmission by the size of the tree in levels, L, the transmission time is proportional to $2^L - 1$.

In this chapter such nonlinear models of communication time operating either with linear or nonlinear models of computation is investigated. This is done in the context of divisible loads and divisible load scheduling. Divisible loads are perfectly partitionable loads that are a good model for parallel systems processing embarrassingly parallel loads consisting of voluminous amounts of data. That is, we assume that there are no precedence relationships in the processing. Divisible load scheduling techniques are used in this chapter because of their tractability in order to make analytical progress.

Over a hundred journal papers [14] describing the divisible load scheduling have been published since the original work of Cheng and Robertazzi in 1988 [15] and Agrawal and Jagadish [46] that year. The basic problem is

to determine the optimal scheduling of load given the interconnection and processor network topology, scheduling policy, processor and link speeds, and computing and communication intensities. The aim is to finish processing in the shortest time by the optimal scheduling of the load taking into consideration the aspects of both computation and communication. This occurs if processors stop working at the same time. If not, the load can be transferred from busy to idle processors to improve the solution. This linear model is well suited for parallel and distributed computing because of its tractable and realistic characteristics [17].

Over the years, divisible load theory has been applied to various kinds of networks topologies including linear daisy chains, tree and bus networks using a set of recursive equations [15] [18] [19]. Further studies have been made for hypercubes [20] and mesh networks [21]. The idea of equivalent networks [45] was developed for complex networks such as multilevel tree networks. Research has also examined scheduling policy with multi-installment [48], multi-round algorithms [24], independent task scheduling [25], fixed communication charges [26], detailed parameterizations and solution reporting time optimization [27], large matrix vector computations [28] and combinatorial optimization [29]. Recent new applications includes aligning biological sequences [34], aligning multiple protein sequences [35], optimizing parallel image registration [36], divisible MapReduce computations [47], multicore and parallel channel systems [38] and parallel video transcoding [39].

To the best of our knowledge, only nonlinear computation, not nonlinear communication, has been investigated to date using divisible load theory.

The first to so was Drozdowski and Wolniewicz [30] who demonstrated superlinear speedup by defining processing times as a piecewise linear (and thus nonlinear) function of the size of the input load for evaluating the memory hierarchy of a computer. These results were determined using mathematical programming. Later, Hung and Robertazzi [31] obtained analytically optimal load allocation and speedup for simultaneous (i.e. concurrent) load distribution for a quadratic nonlinearity. They also presented an iterative solution for sequential load distribution with a nonlinearity of arbitrary power. Suresh, Run, Kim et.al. [32] [33] used a mild assumption on the communication to computation speed ratio to present scheduling solutions for certain nonlinear divisible load problems including optimal sequencing and arrangement results.

This chapter is organized as follows. In section 4.1, the introduction was made. Section 4.2 explains the notations. In section 4.3, we discuss the optimal scheduling for different distribution policies. In section 4.4, several specific examples are presented. In section 4.5, we make a conclusion.

## 4.2 List of symbols

$\alpha_i$: The load fraction assigned to the $i$th child processor from the root node.

$w_i$: The inverse of the computing speed of the $i$th child processor.

$z_i$: The inverse of the link speed of the link connects $i$th processor to the root node.

$T_{cp}$: Computing intensity constant: The entire load is processed in $w_i T_{cp}$ seconds by the $i$th child processor.

$T_{cm}$: Communication intensity constant: The entire load can be transmitted in $z_i T_{cm}$ seconds to the $i$th child processor from the root node.

## 4.3 Optimal scheduling under different distribution policies

### 4.3.1 Sequential distribution, simultaneous start

We consider single level trees in this chapter as a basic starting point architecture. Note that if link speeds are homogenous, a single level tree under sequential distribution is equivalent to a bus. We also assume communication speed and computation speed are known though they can be estimated [40]. Consider now a single level tree network, sequential distribution (load is distributed from the root to each child in turn), simultaneous start policy (load reception and computation start at the same time). We also assume that the root does processing.

Certainly many nonlinear communication/computation functions are possible. In this chapter, for purposes of demonstration we use a communication integer power nonlinearity $\chi$ ($\chi > 1$). Certainly also polynomial nonlinearities could beconsidered. From figure 4.1, one has the timing equations:

$$\alpha_0 w_0 T_{cp} = \alpha_1 w_1 T_{cp} \tag{4.1}$$

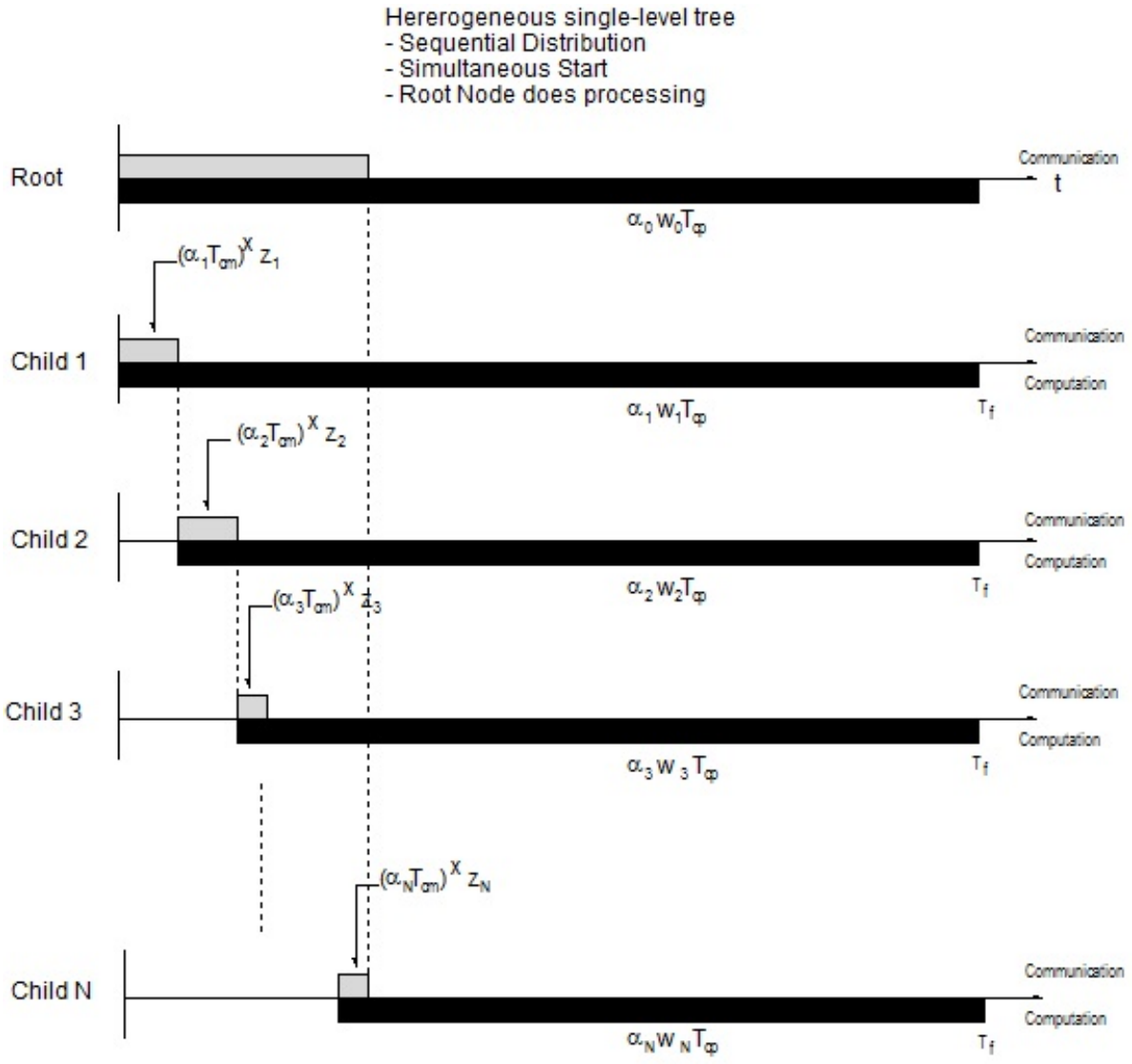


Figure 4.1: Sequential distribution simultaneous start

$$\alpha_1 w_1 T_{cp} = (\alpha_1 T_{cm})^{\chi} z_1 + \alpha_2 w_2 T_{cp} \tag{4.2}$$

$$\alpha_2 w_2 T_{cp} = (\alpha_2 T_{cm})^{\chi} z_2 + \alpha_3 w_3 T_{cp} \tag{4.3}$$

$$\cdots\cdots$$

$$\cdots\cdots$$

$$\cdots\cdots$$

$$\alpha_{N-1} w_{N-1} T_{cp} = (\alpha_{N-1} T_{cm})^{\chi} z_{N-1} + \alpha_N w_N T_{cp} \tag{4.4}$$

We assume in our timing diagram that communication time does not extend beyond computation time (ie. communication is generally faster than computation).

The normalization equation is:

$$\alpha_0 + \alpha_1 + \alpha_2 + \cdots + \alpha_N = 1 \tag{4.5}$$

These equations can be re-written as:

$$f_0 = \alpha_0 w_0 T_{cp} - \alpha_1 w_1 T_{cp} = 0 \tag{4.6}$$

$$f_1 = (\alpha_1 T_{cm})^{\chi} z_1 - \alpha_1 w_1 T_{cp} + \alpha_2 w_2 T_{cp} = 0 \tag{4.7}$$

$$f_2 = (\alpha_2 T_{cm})^{\chi} z_2 - \alpha_2 w_2 T_{cp} + \alpha_3 w_3 T_{cp} = 0 \tag{4.8}$$

.....

.....

.....

$$f_{N-1} = (\alpha_{N-1} T_{cm})^{\chi} z_{N-1} - \alpha_{N-1} w_{N-1} T_{cp} + \alpha_N w_N T_{cp} = 0 \tag{4.9}$$

$$f_N = \alpha_0 + \alpha_1 + \alpha_2 + \cdots + \alpha_N - 1 = 0 \tag{4.10}$$

and

$$\vec{f} = \begin{bmatrix} f_0(\alpha_0, \alpha_1, \alpha_2, \cdots, \alpha_N) \\ f_1(\alpha_0, \alpha_1, \alpha_2, \cdots, \alpha_N) \\ f_2(\alpha_0, \alpha_1, \alpha_2, \cdots, \alpha_N) \\ \vdots \\ f_{N-1}(\alpha_0, \alpha_1, \alpha_2, \cdots, \alpha_N) \\ f_N(\alpha_0, \alpha_1, \alpha_2, \cdots, \alpha_N) \end{bmatrix} = 0 \tag{4.11}$$

One can use the multivariate Newton's method to solve this set of timing equations. For the above equations, the Taylor expansion of $f_i$ in the neighborhood of x:

$$\begin{aligned} f_i(\vec{\alpha} + \delta\vec{\alpha}) = f_i(\vec{\alpha}) + \delta\alpha_0 \frac{\partial f_i}{\partial \alpha_0}(\vec{\alpha}) + \cdots + \delta\alpha_N \frac{\partial f_i}{\partial \alpha_N}(\vec{\alpha}) \\ + O(|\delta\vec{\alpha}|^2) \approx f_i(\vec{\alpha}) + \nabla f_i(\vec{\alpha}) \cdot \delta\vec{\alpha} \end{aligned} \tag{4.12}$$

This can be rewritten as

$$\vec{f}(\vec{\alpha} + \delta\vec{\alpha}) \approx \vec{f}(\vec{\alpha}) + \mathcal{J}_{\vec{f}}(\vec{\alpha})\delta\vec{\alpha} = 0 \tag{4.13}$$

where $\mathcal{J}_{\vec{f}}(\vec{\alpha})$ is the Jacobian of $\vec{f} = (f_1, \cdots, f_N)^T$.

$$\mathcal{J}_{\vec{f}}(\vec{\alpha}) = \begin{bmatrix} \nabla f_0^T(\vec{\alpha}) \\ \nabla f_1^T(\vec{\alpha}) \\ \nabla f_2^T(\vec{\alpha}) \\ \vdots \\ \nabla f_N^T(\vec{\alpha}) \end{bmatrix} = \begin{bmatrix} \frac{\partial f_0}{\partial \alpha_0}(\vec{\alpha}) & \cdots & \frac{\partial f_0}{\partial \alpha_N}(\vec{\alpha}) \\ \frac{\partial f_1}{\partial \alpha_0}(\vec{\alpha}) & \cdots & \frac{\partial f_1}{\partial \alpha_N}(\vec{\alpha}) \\ \frac{\partial f_2}{\partial \alpha_0}(\vec{\alpha}) & \cdots & \frac{\partial f_2}{\partial \alpha_N}(\vec{\alpha}) \\ \vdots & \vdots & \vdots \\ \frac{\partial f_N}{\partial \alpha_0}(\vec{\alpha}) & \cdots & \frac{\partial f_N}{\partial \alpha_N}(\vec{\alpha}) \end{bmatrix} \tag{4.14}$$

One can first make a guess of the solution for the $\alpha$s and then iterate the relation below until it converges to a solution:

$$\vec{\alpha}^{k+1} = \vec{\alpha}^k - \mathcal{J}^{-1}(\vec{\alpha}^k)\vec{f}(\vec{\alpha}^k) \tag{4.15}$$

### 4.3.2 Sequential distribution, staggered start

A sequential distribution, staggered start policy involves the root node distributing loads to its children nodes in a sequential way and the children nodes starting computation after they have received their entire fractions of the loads.

From the timing diagram figure 2, one can write timing equations as

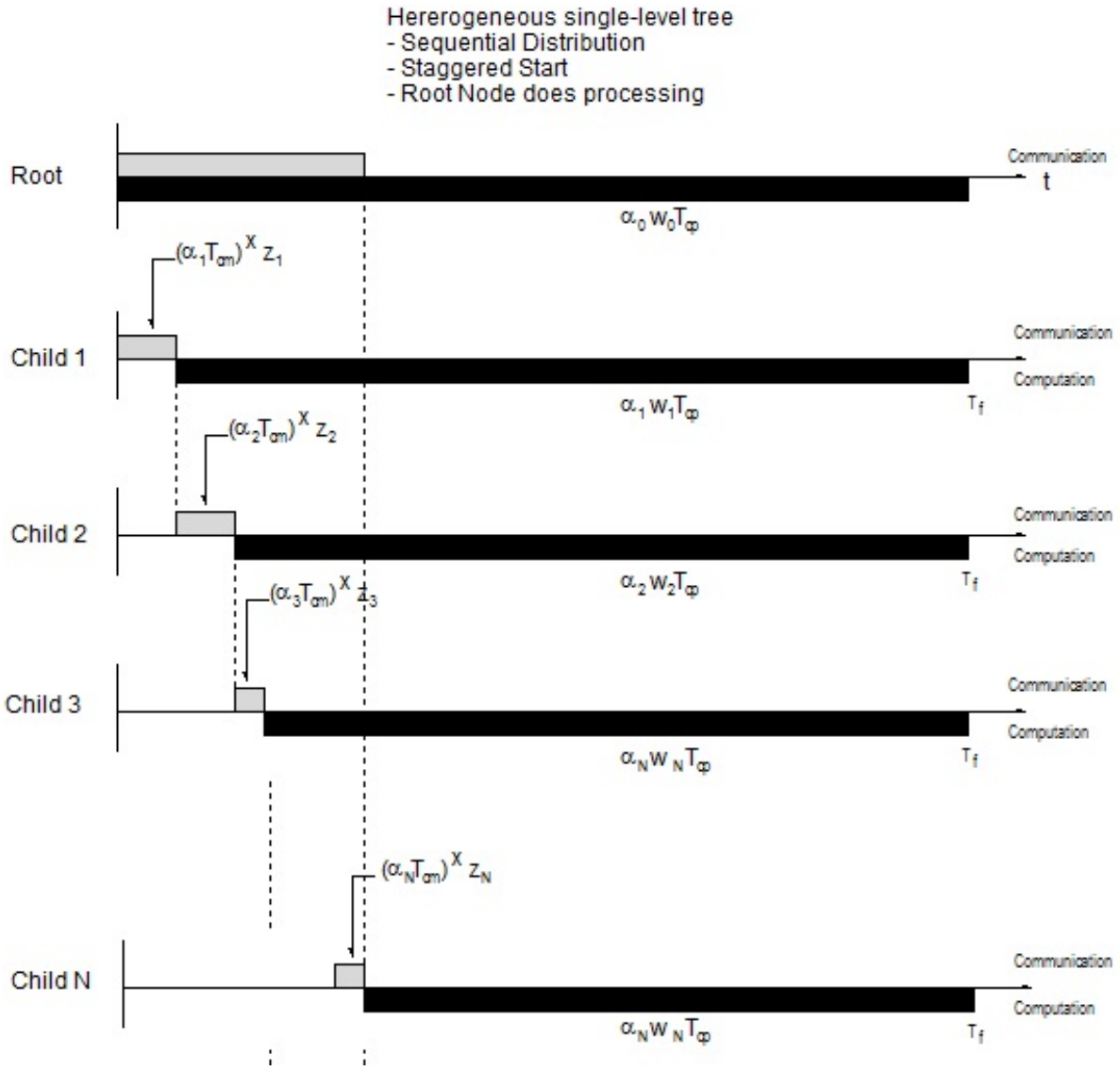


Figure 4.2: Sequential distribution staggered start

below:

$$\alpha_0 w_0 T_{cp} = (\alpha_1 T_{cm})^{\chi} z_1 + \alpha_1 w_1 T_{cp} \tag{4.16}$$

$$\alpha_1 w_1 T_{cp} = (\alpha_2 T_{cm})^{\chi} z_2 + \alpha_2 w_2 T_{cp} \tag{4.17}$$

$$\alpha_2 w_2 T_{cp} = (\alpha_3 T_{cm})^{\chi} z_3 + \alpha_3 w_3 T_{cp} \tag{4.18}$$

$$\cdots\cdots$$

$$\cdots\cdots$$

$$\cdots\cdots$$

$$\alpha_{N-1} w_{N-1} T_{cp} = (\alpha_N T_{cm})^{\chi} z_N + \alpha_N w_N T_{cp} \tag{4.19}$$

Manipulating the recursive equations and normalization equation one can obtain

$$f_0 = \alpha_0 w_0 T_{cp} - (\alpha_1 T_{cm})^{\chi} z_1 - \alpha_1 w_1 T_{cp} = 0 \tag{4.20}$$

$$f_1 = \alpha_1 w_1 T_{cp} - (\alpha_2 T_{cm})^{\chi} z_2 - \alpha_2 w_2 T_{cp} = 0 \tag{4.21}$$

$$f_2 = \alpha_2 w_2 T_{cp} - (\alpha_3 T_{cm})^{\chi} z_3 - \alpha_3 w_3 T_{cp} = 0 \tag{4.22}$$

$$\cdots\cdots$$

$$\cdots\cdots$$

.....

$$f_{N-1} = \alpha_{N-1} w_{N-1} T_{cp} - (\alpha_N T_{cm})^{\chi} z_N - \alpha_N w_N T_{cp} = 0 \tag{4.23}$$

$$f_N = \alpha_0 + \alpha_1 + \alpha_2 + \cdots + \alpha_N - 1 = 0 \tag{4.24}$$

Then one can use the Newton's method described above to solve for the unknown $\alpha$'s.

### 4.3.3 Simultaneous distribution, staggered start

A simultaneous distribution, staggered start policy involves the root distributing loads to its children nodes simultaneously and the children nodes starting computation after they have received the entire fractions of the loads.In figure 4.3:

$$\alpha_0 w_0 T_{cp} = (\alpha_1 T_{cm})^{\chi} z_1 + \alpha_1 w_1 T_{cp} \tag{4.25}$$

$$(\alpha_1 T_{cm})^{\chi} z_1 + \alpha_1 w_1 T_{cp} = (\alpha_2 T_{cm})^{\chi} z_2 + \alpha_2 w_2 T_{cp} \tag{4.26}$$

$$(\alpha_2 T_{cm})^{\chi} z_2 + \alpha_2 w_2 T_{cp} = (\alpha_3 T_{cm})^{\chi} z_3 + \alpha_3 w_3 T_{cp} \tag{4.27}$$

.....

.....

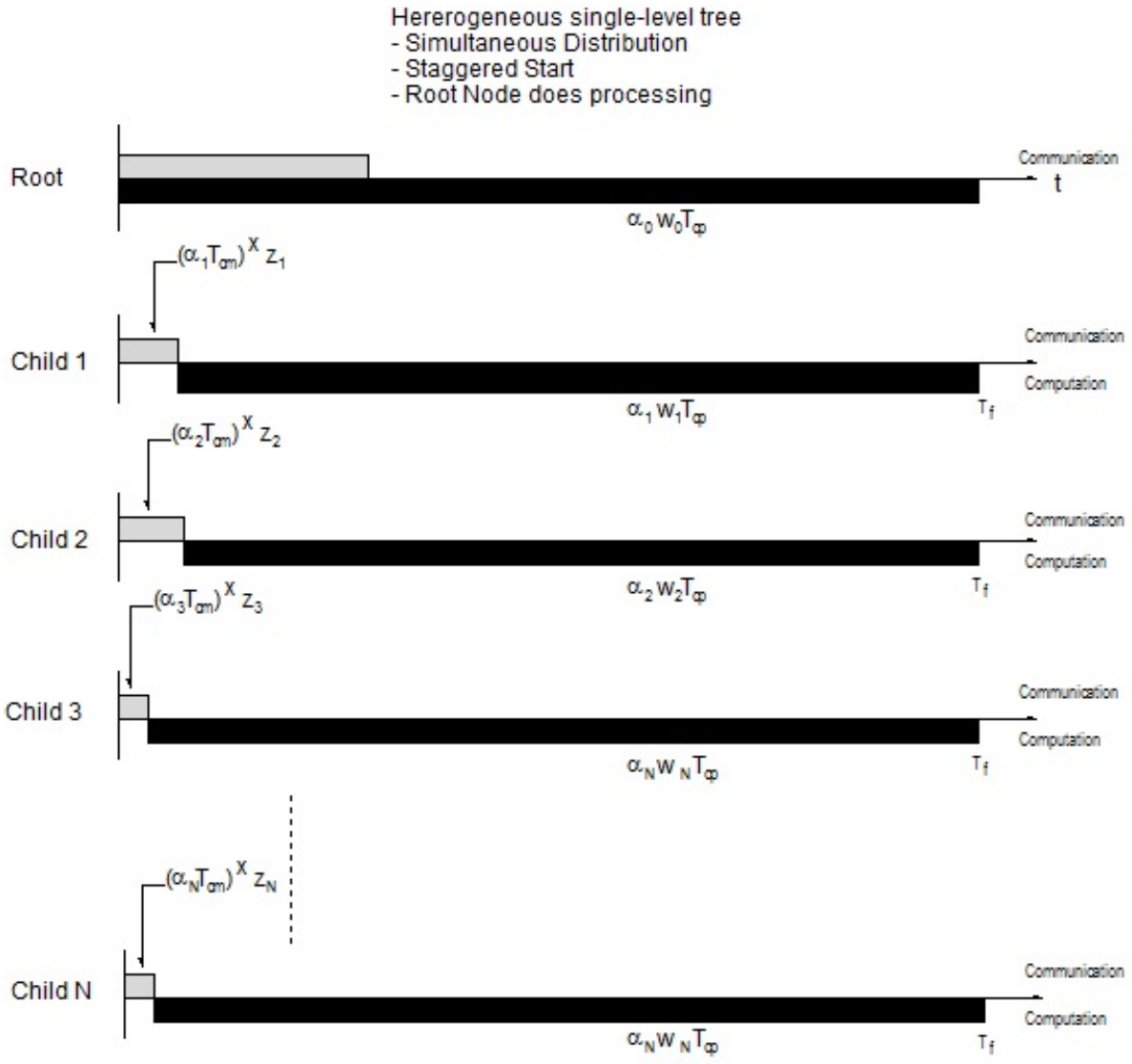


Figure 4.3: Simultaneous distribution staggered start

.....

$$(\alpha_{N-1}T_{cm})^{\chi}z_{N-1} + \alpha_{N-1}w_{N-1}T_{cp} = (\alpha_N T_{cm})^{\chi}z_N + \alpha_N w_N T_{cp} \tag{4.28}$$

Manipulating the recursive equations and normalization equation one can obtain

$$f_0 = \alpha_0 w_0 T_{cp} - (\alpha_1 T_{cm})^{\chi}z_1 - \alpha_1 w_1 T_{cp} = 0 \tag{4.29}$$

$$f_1 = (\alpha_1 T_{cm})^{\chi}z_1 + \alpha_1 w_1 T_{cp} - (\alpha_2 T_{cm})^{\chi}z_2 - \alpha_2 w_2 T_{cp} = 0 \tag{4.30}$$

$$f_2 = (\alpha_2 T_{cm})^{\chi}z_2 + \alpha_2 w_2 T_{cp} - (\alpha_3 T_{cm})^{\chi}z_3 - \alpha_3 w_3 T_{cp} = 0 \tag{4.31}$$

.....

.....

.....

$$\begin{aligned} f_{N-1} = (\alpha_{N-1}T_{cm})^{\chi}z_{N-1} + \alpha_{N-1}w_{N-1}T_{cp} \\ -(\alpha_N T_{cm})^{\chi}z_N - \alpha_N w_N T_{cp} = 0 \end{aligned} \tag{4.32}$$

$$f_N = \alpha_0 + \alpha_1 + \alpha_2 + \cdots + \alpha_N - 1 = 0 \tag{4.33}$$

Then one can use the Newton's method to solve it for the unknown $\alpha$'s.

### 4.3.4 Nonlinear communication, nonlinear computation, sequential distribution, staggered start

Here, both the computation and communication time is nonlinear to the size of the load. We have power $\chi$ for the communication nonlinearity and power $y$ for the computation nonlinearity.

A sequential distribution, staggered start policy involves the root node distributing loads to its children nodes in a sequential way and the children nodes starting computation after they have received the entire fractions of the loads.

We now consider integer computation power nonlinearity $y$ ( $y > 1$) as used as communication nonlinearity

One can have timing equations (figure 4.4):

$$(\alpha_0 T_{cp})^y w_0 = (\alpha_1 T_{cm})^\chi z_1 + (\alpha_1 T_{cp})^y w_1 \tag{4.34}$$

$$(\alpha_1 T_{cp})^y w_1 = (\alpha_2 T_{cm})^\chi z_2 + (\alpha_2 T_{cp})^y w_2 \tag{4.35}$$

$$(\alpha_2 T_{cp})^y w_2 = (\alpha_3 T_{cm})^\chi z_3 + (\alpha_3 T_{cp})^y w_3 \tag{4.36}$$

.....

.....

.....

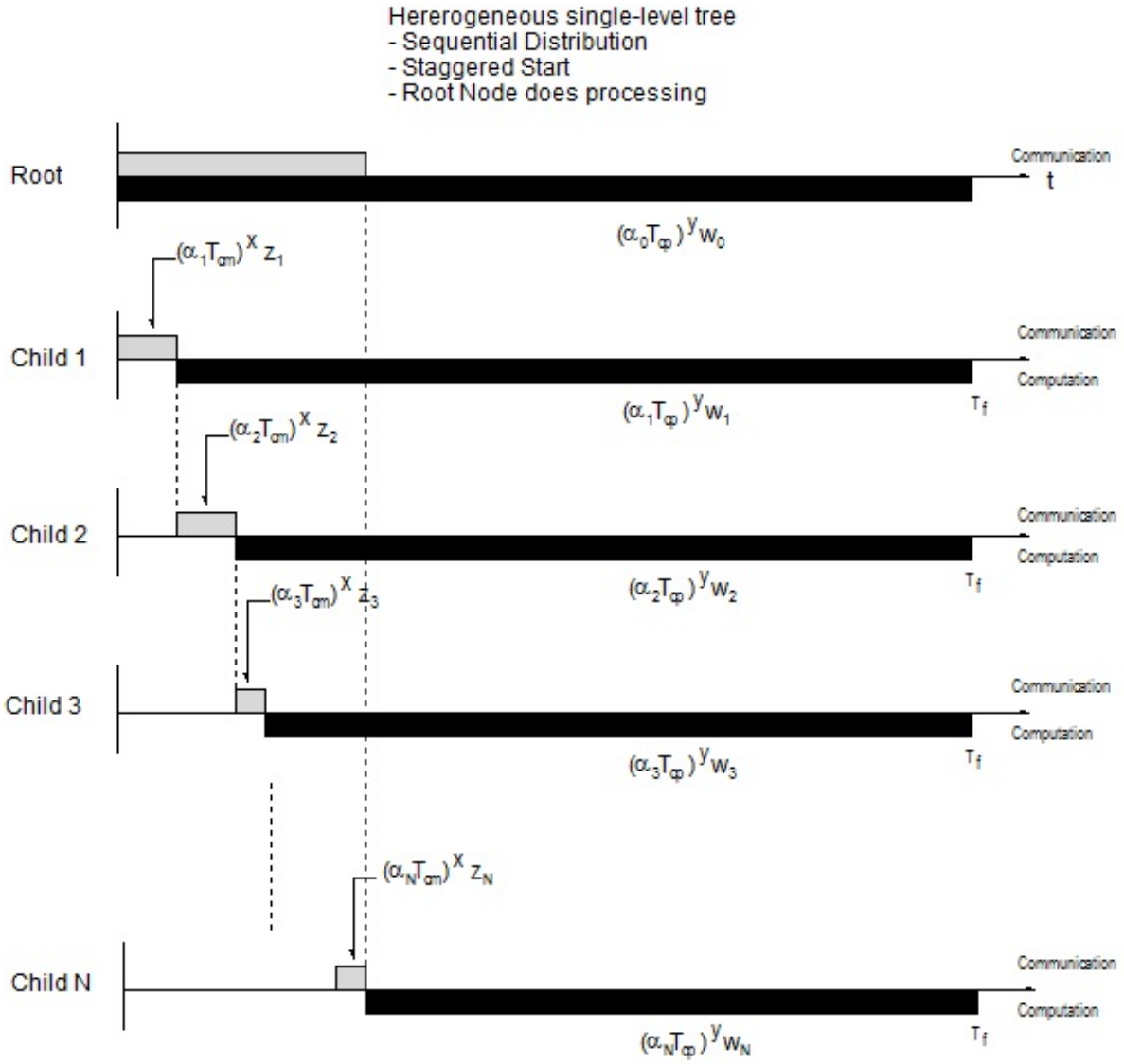


Figure 4.4: Nonlinear communication and nonlinear computation

$$(\alpha_{N-1}T_{cp})^y w_{N-1} = (\alpha_N T_{cm})^\chi z_N + (\alpha_N T_{cp})^y w_N \tag{4.37}$$

$$\alpha_0 + \alpha_1 + \alpha_2 + \cdots + \alpha_N = 1 \tag{4.38}$$

Manipulating the recursive equations and normalization equation one can obtain

$$f_0 = (\alpha_0 T_{cp})^y w_0 - (\alpha_1 T_{cm})^\chi z_1 - (\alpha_1 T_{cp})^y w_1 = 0 \tag{4.39}$$

$$f_1 = (\alpha_1 T_{cp})^y w_1 - (\alpha_2 T_{cm})^\chi z_2 - (\alpha_2 T_{cp})^y w_2 = 0 \tag{4.40}$$

$$f_2 = (\alpha_2 T_{cp})^y w_2 - (\alpha_3 T_{cm})^\chi z_3 - (\alpha_3 T_{cp})^y w_3 = 0 \tag{4.41}$$

$$\cdots\cdots$$

$$\cdots\cdots$$

$$\cdots\cdots$$

$$f_{N-1} = (\alpha_{N-1}T_{cp})^y w_{N-1} - (\alpha_N T_{cm})^\chi z_N - (\alpha_N T_{cp})^y w_N = 0 \tag{4.42}$$

$$f_N = \alpha_0 + \alpha_1 + \alpha_2 + \cdots + \alpha_N - 1 = 0 \tag{4.43}$$

As an example, if the computation problem is the solution of a set of linear equations ($AX = B$), one may have an equation in place of equation

34 for instance:

$$(\alpha_0 T_{cp})^3 w_0 = (\alpha_1 T_{cm})^2 z_1 + (\alpha_1 T_{cp})^3 w_1 \tag{4.44}$$

where computation time is cubic of the size of the matrix (in terms of the number of rows) and square of the communication (in terms of the number of rows).

One can calculate the Jacobian of the $\vec{f}$ and use Newton's method to solve for the unknown $\alpha$'s.

## 4.4 Specific Examples

Here, we take $z$, $T_{cp}$ and $T_{cm}$ all as 1 and leave $w$ as a variable. Note $z$ (inverse communication speed) appears linearity in the equations. Below are two performance measurements we use:

**Makespan**: the time period between when the root processor starts to send loads and the last processor finishes computing.

**Speedup**: the ratio of processing time on one processor to processing time on the entire network.

### 4.4.1 Second order, sequential distribution, staggered start

In this case, the communication time is the square of the size of the load. One has the timing equations:

$$\vec{f} = \begin{bmatrix} \alpha_0 w_0 T_{cp} - (\alpha_1 T_{cm})^2 z_1 - \alpha_1 w_1 T_{cp} \\ \alpha_1 w_1 T_{cp} - (\alpha_2 T_{cm})^2 z_2 - \alpha_2 w_2 T_{cp} \\ \alpha_2 w_2 T_{cp} - (\alpha_3 T_{cm})^2 z_3 - \alpha_3 w_3 T_{cp} \\ \vdots \\ \alpha_{N-1} w_{N-1} T_{cp} - (\alpha_N T_{cm})^2 z_N - \alpha_N w_N T_{cp} \\ \alpha_0 + \alpha_1 + \alpha_2 + \cdots + \alpha_N - 1 \end{bmatrix} = 0 \tag{4.45}$$

Let

$$\epsilon_i = w_i T_{cp} \tag{4.46}$$

and

$$\theta_i = 2\alpha_i T_{cm}^2 z_i \tag{4.47}$$

The Jacobian of $\vec{f}$ is:

$$\mathcal{J}_{\vec{f}}(\vec{\alpha}) = \begin{bmatrix} \frac{\partial f_0}{\partial \alpha_0}(\vec{\alpha}) & \cdots & \frac{\partial f_0}{\partial \alpha_N}(\vec{\alpha}) \\ \frac{\partial f_1}{\partial \alpha_0}(\vec{\alpha}) & \cdots & \frac{\partial f_1}{\partial \alpha_N}(\vec{\alpha}) \\ \vdots & \vdots & \vdots \\ \frac{\partial f_N}{\partial \alpha_0}(\vec{\alpha}) & \cdots & \frac{\partial f_N}{\partial \alpha_N}(\vec{\alpha}) \end{bmatrix}$$

$$= \begin{bmatrix} \epsilon_0 & -\theta_1 - \epsilon_1 & 0 & \cdots & 0 & 0 \\ 0 & \epsilon_1 & -\theta_2 - \epsilon_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & \epsilon_{N-1} & -\theta_N - \epsilon_N \\ 1 & 1 & 1 & \cdots & 1 & 1 \end{bmatrix} \tag{4.48}$$

One can set the initial $\vec{\alpha}$ as all zeros and insert into the right hand side of the iterative function below to get a newer set of the $\vec{\alpha}$ on the left hand side. Substitute the newer $\vec{\alpha}$ into the right hand side again and so on.

$$\vec{\alpha}^{k+1} = \vec{\alpha}^{k} - \mathcal{J}^{-1}(\vec{\alpha}^{k})\vec{f}(\vec{\alpha}^{k}) \tag{4.49}$$

After the $\vec{\alpha}$ is obtained, one can calculate the speedup and the makespan (finish time).

$$Makespan = \alpha_0 w_0 T_{cp} \tag{4.50}$$

$$Speedup = \frac{w_0 T_{cp}}{\alpha_0 w_0 T_{cp}} = \frac{1}{\alpha_0} \tag{4.51}$$

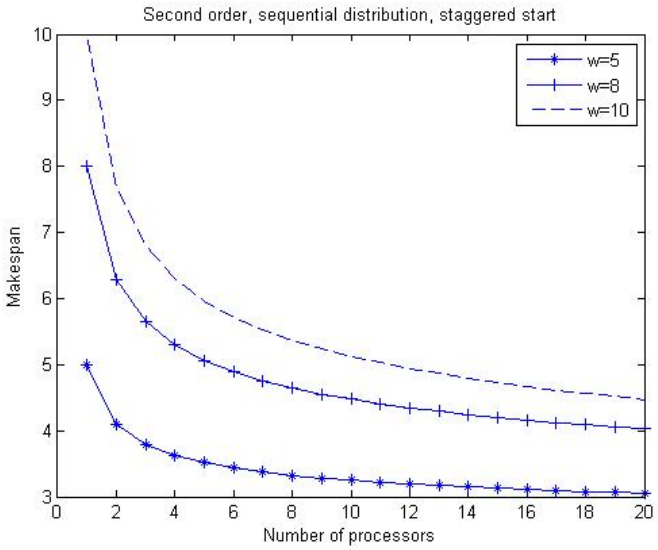


Figure 4.5: Makespan - Sequential distribution staggered start (second order communication)

Figure 4.5 and 4.6 show how the makespan and speedup change as the number of processors increases with sequential distribution, staggered start and this second order nonlinearity.

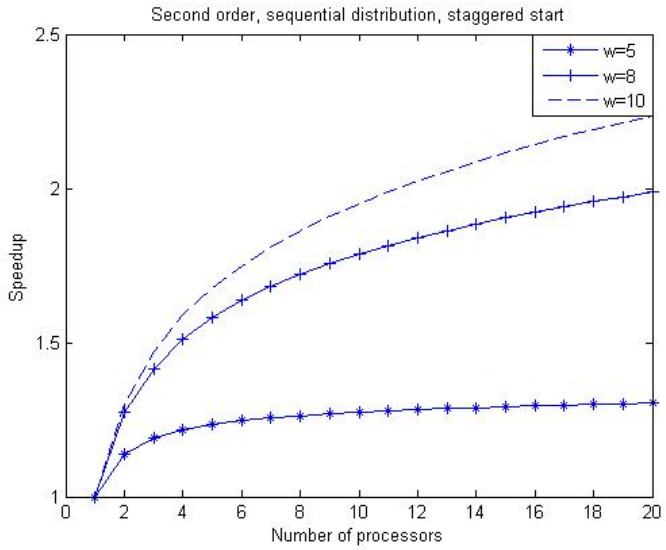


Figure 4.6: Speedup - Sequential distribution staggered start (second order communication)

### 4.4.2 Third order, sequential distribution, staggered start

In this case, the communication time is the cubic of the size of the load. One has the timing equations:

$$\vec{f} = \begin{bmatrix} \alpha_0 w_0 T_{cp} - (\alpha_1 T_{cm})^3 z_1 - \alpha_1 w_1 T_{cp} \\ \alpha_1 w_1 T_{cp} - (\alpha_2 T_{cm})^3 z_2 - \alpha_2 w_2 T_{cp} \\ \alpha_2 w_2 T_{cp} - (\alpha_3 T_{cm})^3 z_3 - \alpha_3 w_3 T_{cp} \\ \vdots \\ \alpha_{N-1} w_{N-1} T_{cp} - (\alpha_N T_{cm})^3 z_N - \alpha_N w_N T_{cp} \\ \alpha_0 + \alpha_1 + \alpha_2 + \cdots + \alpha_N - 1 \end{bmatrix} = 0 \qquad (4.52)$$

Let

$$\epsilon_i = w_i T_{cp} \tag{4.53}$$

and

$$\theta_i = 3\alpha_i^2 T_{cm}^3 z_i \tag{4.54}$$

The Jacobian of $\vec{f}$ is:

$$\begin{aligned} \mathcal{J}_{\vec{f}}(\vec{\alpha}) &= \begin{bmatrix} \frac{\partial f_0}{\partial \alpha_0}(\vec{\alpha}) & \cdots & \frac{\partial f_0}{\partial \alpha_N}(\vec{\alpha}) \\ \frac{\partial f_1}{\partial \alpha_0}(\vec{\alpha}) & \cdots & \frac{\partial f_1}{\partial \alpha_N}(\vec{\alpha}) \\ \vdots & \vdots & \vdots \\ \frac{\partial f_N}{\partial \alpha_0}(\vec{\alpha}) & \cdots & \frac{\partial f_N}{\partial \alpha_N}(\vec{\alpha}) \end{bmatrix} \\ &= \begin{bmatrix} \epsilon_0 & -\theta_1 - \epsilon_1 & 0 & \cdots & 0 & 0 \\ 0 & \epsilon_1 & -\theta_2 - \epsilon_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & \epsilon_{N-1} & -\theta_N - \epsilon_N \\ 1 & 1 & 1 & \cdots & 1 & 1 \end{bmatrix} \end{aligned} \tag{4.55}$$

One can set the initial $\vec{\alpha}$ as all zeros and insert it into the right hand side of the iterative function below to get a newer set of the $\vec{\alpha}$ on the left hand side. Substitute the newer $\vec{\alpha}$ into the right hand side again and so on.

$$\vec{\alpha}^{k+1} = \vec{\alpha}^k - \mathcal{J}^{-1}(\vec{\alpha}^k)\vec{f}(\vec{\alpha}^k) \tag{4.56}$$

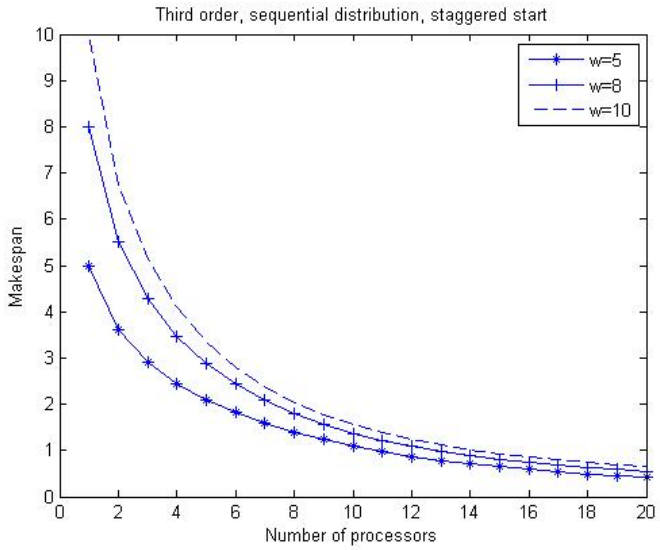


Figure 4.7: Makespan - Sequential distribution staggered start (third order communication)

After the $\vec{\alpha}$ is obtained, one can calculate the speedup and the makespan (finish time).

$$Makespan = \alpha_0 w_0 T_{cp} \tag{4.57}$$

$$Speedup = \frac{w_0 T_{cp}}{\alpha_0 w_0 T_{cp}} = \frac{1}{\alpha_0} \tag{4.58}$$

Figure 4.7 and 4.8 show how the makespan and speedup change as the number of processors increases with sequential distribution, staggered start and third order nonlinearity.

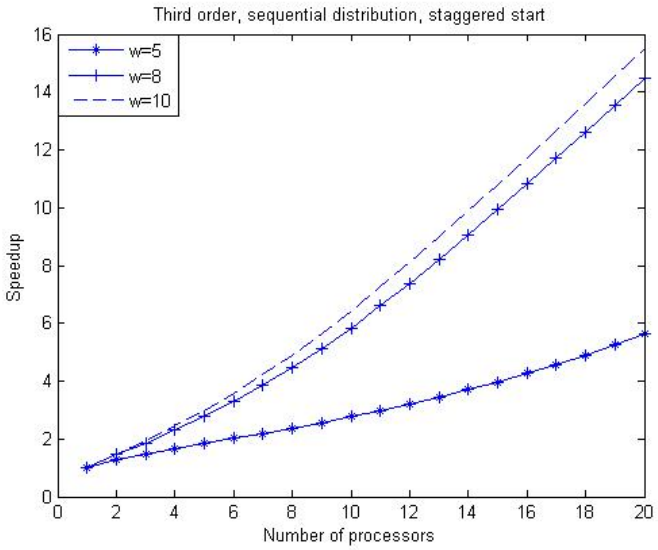


Figure 4.8: Speedup - Sequential distribution staggered start (third order communication)

### 4.4.3 Second order, simultaneous distribution, staggered start

In this case, the communication time is the square of the size of the load. One has the timing equations:

$$\vec{f} = \begin{bmatrix} \alpha_0 w_0 T_{cp} - (\alpha_1 T_{cm})^2 z_1 - \alpha_1 w_1 T_{cp} \\ \alpha_1 w_1 T_{cp} + (\alpha_1 T_{cm})^2 z_1 - (\alpha_2 T_{cm})^2 z_2 - \alpha_2 w_2 T_{cp} \\ \alpha_2 w_2 T_{cp} + (\alpha_2 T_{cm})^2 z_2 - (\alpha_3 T_{cm})^2 z_3 - \alpha_3 w_3 T_{cp} \\ \vdots \\ \alpha_{N-1} w_{N-1} T_{cp} + (\alpha_{N-1} T_{cm})^2 z_{N-1} - (\alpha_N T_{cm})^2 z_N - \alpha_N w_N T_{cp} \\ \alpha_0 + \alpha_1 + \alpha_2 + \cdots + \alpha_N - 1 \end{bmatrix} = 0 \tag{4.59}$$

Let

$$\epsilon_i = w_i T_{cp} \tag{4.60}$$

and

$$\theta_i = 2\alpha_i T_{cm}^2 z_i \tag{4.61}$$

The Jacobian of $\vec{f}$ is:

$$\mathcal{J}_{\vec{f}}(\vec{\alpha}) = \begin{bmatrix} \frac{\partial f_0}{\partial \alpha_0}(\vec{\alpha}) & \cdots & \frac{\partial f_0}{\partial \alpha_N}(\vec{\alpha}) \\ \frac{\partial f_1}{\partial \alpha_0}(\vec{\alpha}) & \cdots & \frac{\partial f_1}{\partial \alpha_N}(\vec{\alpha}) \\ \frac{\partial f_2}{\partial \alpha_0}(\vec{\alpha}) & \cdots & \frac{\partial f_2}{\partial \alpha_N}(\vec{\alpha}) \\ \vdots & \vdots & \vdots \\ \frac{\partial f_N}{\partial \alpha_0}(\vec{\alpha}) & \cdots & \frac{\partial f_N}{\partial \alpha_N}(\vec{\alpha}) \end{bmatrix}$$

$$= \begin{bmatrix} \epsilon_0 & -\epsilon_1 - \theta_1 & 0 & \cdots & 0 & 0 \\ 0 & \epsilon_1 + \theta_1 & -\epsilon_2 - \theta_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & \epsilon_{N-1} + \theta_{N-1} & -\epsilon_N - \theta_N \\ 1 & 1 & 1 & \cdots & 1 & 1 \end{bmatrix} \tag{4.62}$$

One can set the initial $\vec{\alpha}$ as all zeros and insert it into the right hand side of the iterative function below to get a newer set of the $\vec{\alpha}$ on the left hand side. Substitute the newer $\vec{\alpha}$ into the right hand side again and so on.

$$\vec{\alpha}^{k+1} = \vec{\alpha}^k - \mathcal{J}^{-1}(\vec{\alpha}^k)\vec{f}(\vec{\alpha}^k) \tag{4.63}$$

After the $\vec{\alpha}$ is obtained, one can calculate the speedup and the makespan (finish time).

$$Makespan = \alpha_0 w_0 T_{cp} \tag{4.64}$$

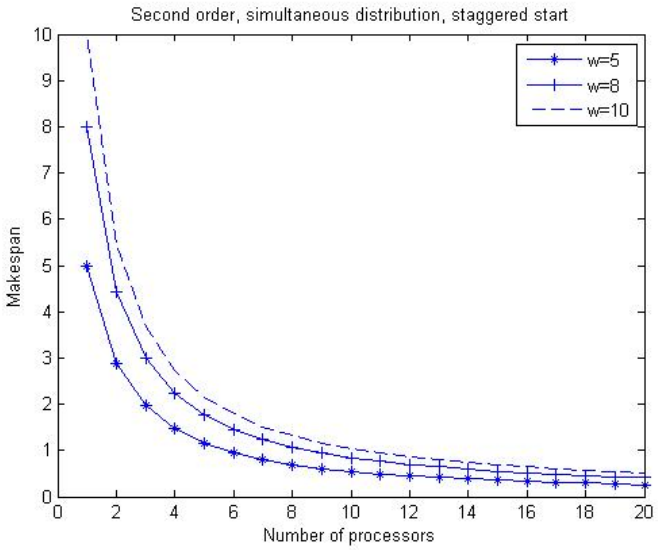


Figure 4.9: Makespan - Simultaneous distribution staggered start (second order communication)

$$Speedup = \frac{w_0 T_{cp}}{\alpha_0 w_0 T_{cp}} = \frac{1}{\alpha_0} \tag{4.65}$$

Figure 4.9 and 4.10 show how the makespan and speedup change as the number of processors increases with simultaneous distribution, staggered start and second order nonlinearity.

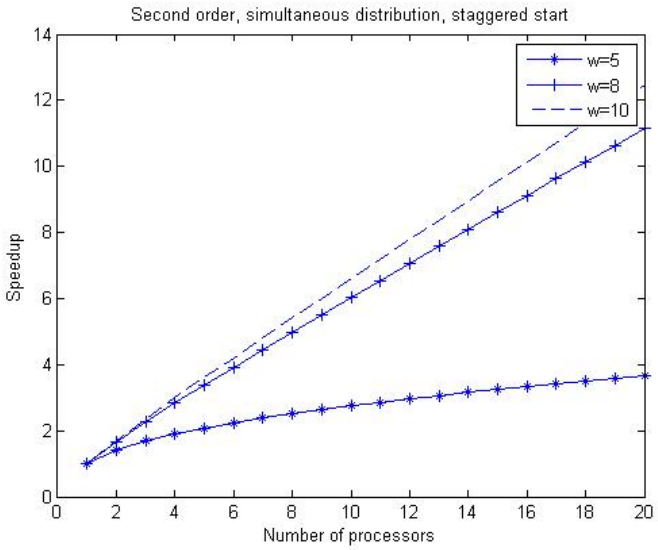


Figure 4.10: Speedup - Simultaneous distribution staggered start (second order communication)

### 4.4.4 Third order, simultaneous distribution, staggered start

In this case, the communication time is the cube of the size of the load. One has the timing equations:

$$\vec{f} = \begin{bmatrix} \alpha_0 w_0 T_{cp} - (\alpha_1 T_{cm})^3 z_1 - \alpha_1 w_1 T_{cp} \\ \alpha_1 w_1 T_{cp} + (\alpha_1 T_{cm})^3 z_1 - (\alpha_2 T_{cm})^3 z_2 - \alpha_2 w_2 T_{cp} \\ \alpha_2 w_2 T_{cp} + (\alpha_2 T_{cm})^3 z_2 - (\alpha_3 T_{cm})^3 z_3 - \alpha_3 w_3 T_{cp} \\ \vdots \\ \alpha_{N-1} w_{N-1} T_{cp} + (\alpha_{N-1} T_{cm})^3 z_{N-1} - (\alpha_N T_{cm})^3 z_N - \alpha_N w_N T_{cp} \\ \alpha_0 + \alpha_1 + \alpha_2 + \cdots + \alpha_N - 1 \end{bmatrix} = 0 \tag{4.66}$$

Let

$$\epsilon_i = w_i T_{cp} \tag{4.67}$$

and

$$\theta_i = 3\alpha_i^2 T_{cm}^3 z_i \tag{4.68}$$

The Jacobian of $\vec{f}$ is:

$$\begin{aligned}
\mathcal{J}_{\vec{f}}(\vec{\alpha}) &= \begin{bmatrix} \frac{\partial f_0}{\partial \alpha_0}(\vec{\alpha}) & \cdots & \frac{\partial f_0}{\partial \alpha_N}(\vec{\alpha}) \\ \frac{\partial f_1}{\partial \alpha_0}(\vec{\alpha}) & \cdots & \frac{\partial f_1}{\partial \alpha_N}(\vec{\alpha}) \\ \frac{\partial f_2}{\partial \alpha_0}(\vec{\alpha}) & \cdots & \frac{\partial f_2}{\partial \alpha_N}(\vec{\alpha}) \\ \vdots & \vdots & \vdots \\ \frac{\partial f_N}{\partial \alpha_0}(\vec{\alpha}) & \cdots & \frac{\partial f_N}{\partial \alpha_N}(\vec{\alpha}) \end{bmatrix} \\
&= \begin{bmatrix} \epsilon_0 & -\epsilon_1 - \theta_i & 0 & \cdots & 0 & 0 \\ 0 & \epsilon_1 + \theta_1 & -\epsilon_2 - \theta_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & & & \\ 0 & 0 & 0 & \cdots & \epsilon_{N-1} + \theta_{N-1} & -\epsilon_N - \theta_N \\ 1 & 1 & 1 & \cdots & 1 & 1 \end{bmatrix}
\end{aligned} \tag{4.69}$$

One can set the initial $\vec{\alpha}$ as all zeros and insert it into the right hand side of the iterative function below to get a newer set of the $\vec{\alpha}$ on the left hand

side. Substitute the newer $\vec{\alpha}$ into the right hand side again and so on.

$$\vec{\alpha}^{k+1} = \vec{\alpha}^{k} - \mathcal{J}^{-1}(\vec{\alpha}^{k})\vec{f}(\vec{\alpha}^{k}) \tag{4.70}$$

After the $\vec{\alpha}$ is obtained, one can calculate the speedup and the makespan (finish time).

$$Makespan = \alpha_0 w_0 T_{cp} \tag{4.71}$$

$$Speedup = \frac{w_0 T_{cp}}{\alpha_0 w_0 T_{cp}} = \frac{1}{\alpha_0} \tag{4.72}$$

Figure 4.11 and 4.12 show how the makespan and speedup change as the number of processors increases with simultaneous distribution, staggered start and third order nonlinearity.

### 4.4.5 Second order communication, third order computation, sequential distribution, staggered start

In this case, the communication time is the square of the size of the load and the computation time is the cubic of the size of the load. One has the timing

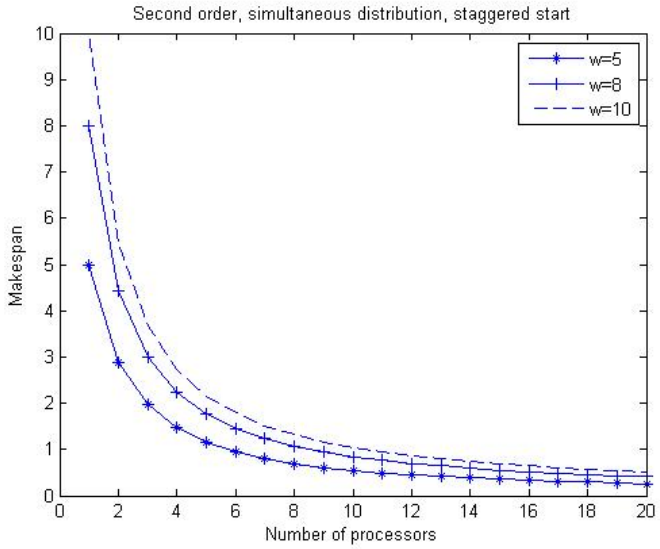


Figure 4.11: Makespan - Simultaneous distribution staggered start (third order communication)

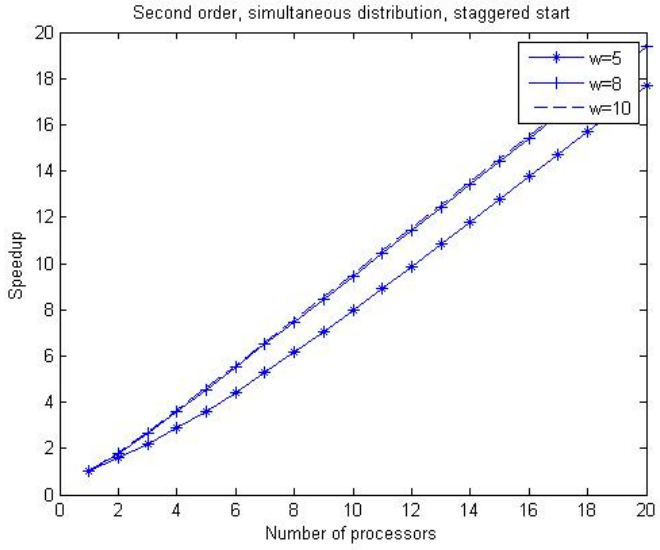


Figure 4.12: Speedup - Simultaneous distribution staggered start (third order communication)

equations:

$$\vec{f} = \begin{bmatrix} (\alpha_0 T_{cp})^3 w_0 - (\alpha_1 T_{cm})^2 z_1 - (\alpha_1 T_{cp})^3 w_1 \\ (\alpha_1 T_{cp})^3 w_1 - (\alpha_2 T_{cm})^2 z_2 - (\alpha_2 T_{cp})^3 w_2 \\ (\alpha_2 T_{cp})^3 w_2 - (\alpha_3 T_{cm})^2 z_3 - (\alpha_3 T_{cp})^3 w_3 \\ \vdots \\ (\alpha_{N-1} T_{cp})^3 w_{N-1} - (\alpha_N T_{cm})^2 z_N - (\alpha_N T_{cp})^3 w_N \\ \alpha_0 + \alpha_1 + \alpha_2 + \cdots + \alpha_N - 1 \end{bmatrix} = 0 \qquad (4.73)$$

Let

$$\epsilon_i = 3\alpha_i^2 T_{cp}^3 w_i \qquad (4.74)$$

and

$$\theta_i = 2\alpha_i T_{cm}^2 z_i \qquad (4.75)$$

The Jacobian of $\vec{f}$ is:

$$\mathcal{J}_{\vec{f}}(\vec{\alpha}) = \begin{bmatrix} \frac{\partial f_0}{\partial \alpha_0}(\vec{\alpha}) & \cdots & \frac{\partial f_0}{\partial \alpha_N}(\vec{\alpha}) \\ \frac{\partial f_1}{\partial \alpha_0}(\vec{\alpha}) & \cdots & \frac{\partial f_1}{\partial \alpha_N}(\vec{\alpha}) \\ \frac{\partial f_2}{\partial \alpha_0}(\vec{\alpha}) & \cdots & \frac{\partial f_2}{\partial \alpha_N}(\vec{\alpha}) \\ \vdots & \vdots & \vdots \\ \frac{\partial f_N}{\partial \alpha_0}(\vec{\alpha}) & \cdots & \frac{\partial f_N}{\partial \alpha_N}(\vec{\alpha}) \end{bmatrix}$$

$$= \begin{bmatrix} \epsilon_0 & -\epsilon_1 - \theta_1 & 0 & \cdots & 0 & 0 \\ 0 & \epsilon_1 & -\epsilon_2 - \theta_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & & & \\ 0 & 0 & 0 & \cdots & \theta_{N-1} & -\epsilon_N - \theta_N \\ 1 & 1 & 1 & \cdots & 1 & 1 \end{bmatrix} \tag{4.76}$$

One can set the initial $\vec{\alpha}$ as all zeros and insert it into the right hand side of the iterative function below to get a newer set of the $\vec{\alpha}$ on the left hand side. Substitute the newer $\vec{\alpha}$ into the right hand side again and so on.

$$\vec{\alpha}^{k+1} = \vec{\alpha}^k - \mathcal{J}^{-1}(\vec{\alpha}^k)\vec{f}(\vec{\alpha}^k) \tag{4.77}$$

After the $\vec{\alpha}$ is obtained, one can calculate the speedup and the makespan (finish time).

$$Makespan = \alpha_0 w_0 T_{cp} \tag{4.78}$$

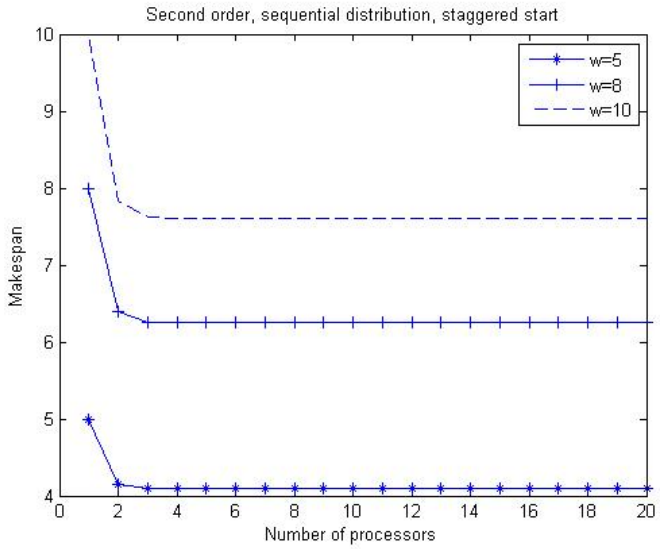


Figure 4.13: Makespan - Nonlinear communication (second order) and nonlinear computation (third order), sequential distribution staggered start

$$Speedup = \frac{w_0 T_{cp}}{\alpha_0 w_0 T_{cp}} = \frac{1}{\alpha_0} \tag{4.79}$$

Figure 4.13 and 4.14 show how the makespan and speedup change as the number of processors increases with sequential distribution, staggered start, second order communication and third order computation.

## 4.5 Conclusion

Scheduling divisible loads with nonlinear commmunication time has many aerospace applications including fast Fourier transform, line detection using

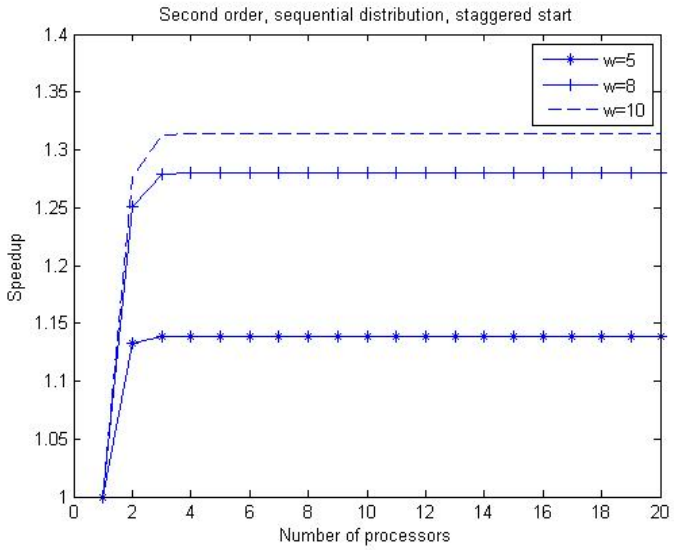


Figure 4.14: Speedup - Nonlinear communication (second order) and nonlinear computation (third order), sequential distribution staggered start

the Hough transform and pattern recognition using 2D hidden Markov models. This chapter proposes an iterative method to find the optimal scheduling for single level tree networks under different distribution policies. Quadratic and cubic nonlinearity examples are used to demonstrate the proposed algorithm. The testing results show that this scheduling algorithm can provide an optimal solution for parallel and distributed systems with divisible and nonlinear communication time loads. Such optimal solutions can maximize the responsiveness of critical data processing where time is of the essence.

# Chapter 5

# Utilization and Progress Performance Measures for Divisible Load Scheduled Trees

## 5.1 Introduction

In any performance evaluation of a computing or communication system performance measures, numerical quantities indicating system performance [41] [42] [43] [44], play an important role.

Over a hundred journal papers [45] describing the performance of divisible load scheduling have been published since the original work of Agarwal and Jagadish [46] in 1988 and Cheng and Robertazzi [15] the same year. Divisible loads are computing and communication loads that are perfectly partitionable among processors and links. One may have a very long linear

data file of numbers whose sum is desired. Fragments of the file can be sent to different processors over an interconnection network, intermediate sums computed and returned to a processor for final summation. This is similar in spirit to the Map/Reduce algorithm [47]. Divisible load theory provides analytical and/or numerical means to determine the amount of load to schedule on links and processors to achieve a minimal time solution for processing the load. The basic problem is to determine the optimal scheduling of load given the interconnection and processor network topology, scheduling policy, processor and link speeds, and computing and communication intensities.

Cheng and Robertazzi considered finish time, the time for all load distribution and computation to end. Other workers have called this performance measure makespan. Later in 1995 [48] Bharadwaj, Ghose and Mani introduced the use of speedup to the divisible load scheduling problem. Speedup is a well-established parallel processing performance measure. It is the ratio of the time to solve a computational problem on one processor divided by the time to solve the problem on N (homogeneous) processors. In 2010 Drozdowski and Wielebski proposed the use of isoefficiency maps for divisible computations [49]. Efficiency is a normalized speedup (speedup divided by the number of processors considered, $N$). Isoefficiency maps are plots of contours of equal efficiency versus such quantities as $N$ and problem size $V$.

In this chapter we compute two novel performance measures for representative sequential and simultaneous distribution scheduling policies for both single-level and multi-level tree networks. One is utilization. This is the fraction of time processors are busy processing computational load. The second

is progress. Progress is the percentage of load processed at a given time. Earlier work on utilization and progress for single level tree networks can be found in the 2008 PhD thesis of Milton Jackson [50].

These two performance measures, progress and utilization, are of interest for real time data processing (How fast is the data processed?), scheduling policy comparative evaluation (Which is the best scheduling policy in a given situation?) and resource allocation (What percentage of time is a processor actually used for processing and what percentage of time it is idle?).

This chapter is organized as follows. In section 5.2, we introduce model and notations of divisible load theory on multi-level tree. In section 5.3, we discuss the solution, utilization of the processors and the percentage of the load that have been processed at a given time $T$ for single level tree networks with divisible loads. In section 5.4, we discuss solution, utilization and progress for multi-level tree networks with divisible loads. In section 5.5, we make a conclusion.

## 5.2 Model and notation

### 5.2.1 Symbols for Single Level Tree

$\alpha_i$: The load fraction assigned to the $i$th child processor from the root node.

$w_i$: The inverse of the computing speed of the $i$th child processor.

$z_i$: The inverse of the link speed of the link connects $i$th processor to the

root node.

$T_{cp}$: Computing intensity constant:

The entire load is processed in $w_i T_{cp}$ seconds by the $i$th child processor.

$T_{cm}$: Communication intensity constant:

The entire load can be transmitted in $z_i T_{cm}$ seconds to the $i$th child processor from the

root node.

$P_i$: $i$th node of the tree.

$T_i$: time point at which the $i$th processor start computing.

$\hat{T}_i$: $i$th smallest element of sorted vector $T$

$T_f$: The finish time. Time at which the last processor ceases computation.

$U_i$: The utilization of the $i$th node in the $i$th level of the tree.

$\alpha\%T$: The percentage of load that has been processed at a time point T.

### 5.2.2 Symbols for Multi-level Tree

$\alpha_{i,j}$: The load fraction assigned to the $j$th node in the $i$th level from its parent.

$w_{i,j}$: The inverse of the computing speed of the $j$th processor in the $i$th level.

$z_{i,j}$: The inverse of the link speed of the link connects $j$th processor in $i$th level to its parent.

$T_{cp}$: Computing intensity constant:

The entire load is processed in $w_{i,j}T_{cp}$ seconds by the $j$th processor in $i$th level.

$T_{cm}$: Communication intensity constant:

The entire load can be transmitted in $z_{i,j}T_{cm}$ seconds to the $j$th processor in the $i$th level.

$T_f$: The finish time. Time at which the last processor ceases computation.

$P_{i,j}$: $j$th node in the $i$th level of the tree.

$U_{i,j}$: the utilization of the $j$th node in the $i$th level of the tree.

$\alpha\%T$: the percentage of load that has been processed at time T.

## 5.3 Single-level tree networks with divisible loads

Basic idea: The assumption is that the communication speed is faster than the computation speed, otherwise there is no need to distribute computational load to other processors. Thus the optimality condition is that all the processors stop computing at the same time. If this was not true, the load could be transferred from the busy processors to the idle ones to increase the computation speed.

There are two strategies to distribute the load: simultaneous distribution and sequential distribution. With the sequential distribution, the root processor distributes load to its children in sequence. With the simultaneous distribution, the root processor distributes load to all of its children at the

same time.

Meanwhile, there are two strategies to start the child processors: simultaneous start and staggered start. With simultaneous start, the child processors start to process when they start to receive load. With the staggered start, the child processors start to process after they have received all the load that they are supposed to process.

### 5.3.1 Simultaneous distribution, staggered start, root node does processing

In this strategy, the root processor keeps a fraction of $\alpha_0$ to process and distributes the other fractions $\alpha_1$, $\alpha_2$, $\cdots$ ,$\alpha_N$ (suppose the root processor has $N$ children processors) to its children processors concurrently. The assignment of the $\alpha$'s is decided by the computing speed of the processors and the communication speed from root processor to the children processors. The children processors start to compute after they have received all the load and they finish computing at the same time.

#### Utilization

From figure 5.1

$$T_{f_1} = \alpha_1 z_1 T_{cm} + \alpha_1 w_1 T_{cp} \tag{5.1}$$

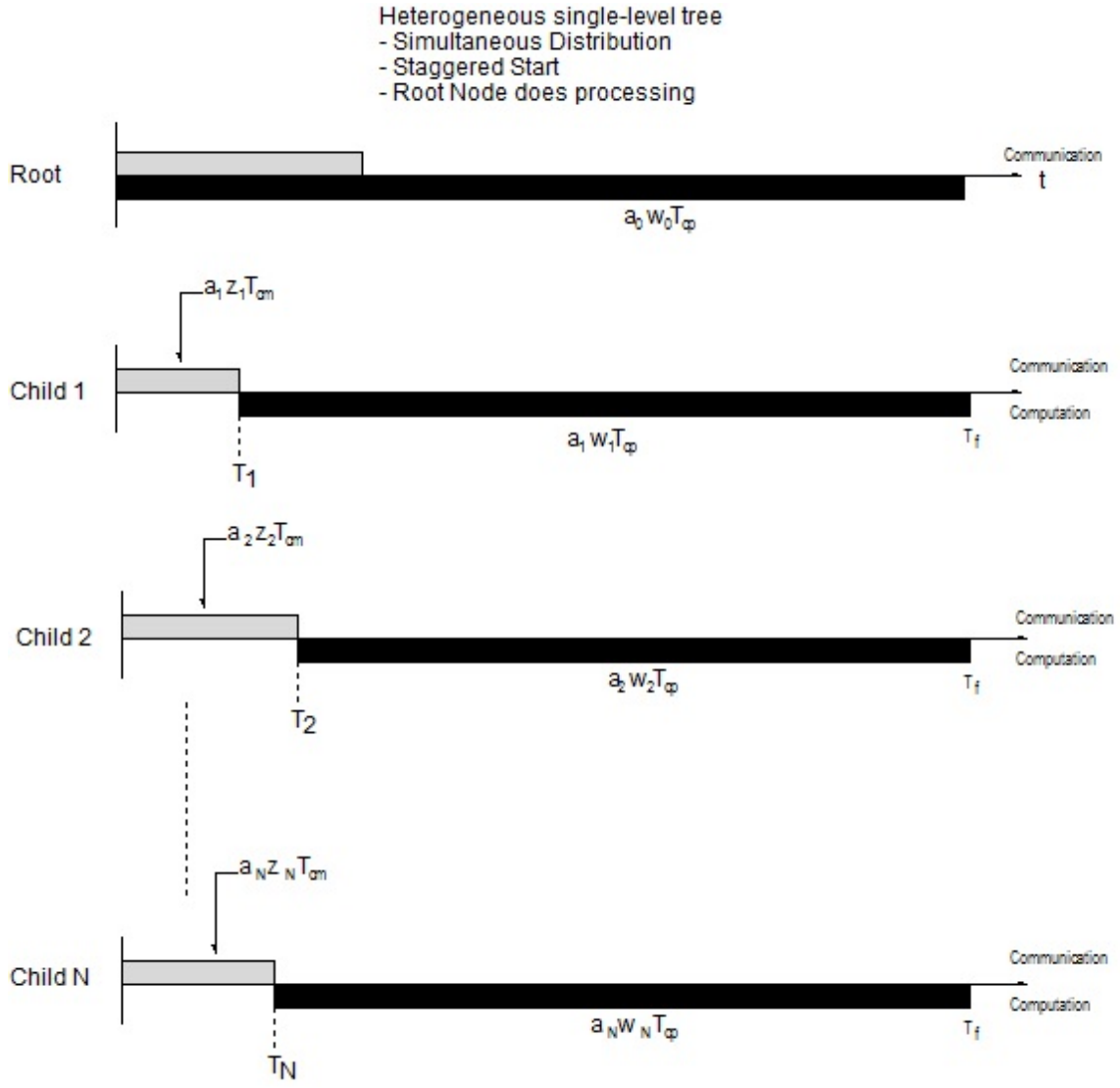


Figure 5.1: Simultaneous distribution, staggered start, root does processing.

$$U_1 = \frac{\alpha_1 w_1 T_{cp}}{\alpha_1 z_1 T_{cm} + \alpha_1 w_1 T_{cp}} \tag{5.2}$$

$$T_{f_2} = \alpha_2 z_2 T_{cm} + \alpha_2 w_2 T_{cp} \tag{5.3}$$

$$U_2 = \frac{\alpha_2 w_2 T_{cp}}{\alpha_2 z_2 T_{cm} + \alpha_2 w_2 T_{cp}} \tag{5.4}$$

$$T_{f_3} = \alpha_3 z_3 T_{cm} + \alpha_3 w_3 T_{cp} \tag{5.5}$$

$$U_3 = \frac{\alpha_3 w_3 T_{cp}}{\alpha_3 z_3 T_{cm} + \alpha_3 w_3 T_{cp}} \tag{5.6}$$

.....

.....

.....

$$T_{f_i} = \alpha_i z_i T_{cm} + \alpha_i w_i T_{cp} \tag{5.7}$$

$$U_i = \frac{\alpha_i w_i T_{cp}}{\alpha_i z_i T_{cm} + \alpha_i w_i T_{cp}} \tag{5.8}$$

The utilization is the ratio of a processor's computational time to the makespan (finish time).

$$U_i = \frac{\alpha_i w_i T_{cp}}{T_f} = \frac{\alpha_i w_i T_{cp}}{\frac{1}{k_1}\alpha_1 w_0 T_{cp}} = k\frac{\alpha_i w_i}{\alpha_1 w_0} \tag{5.9}$$

The general form for the utilization is:

$$AvgU = \frac{1}{N+1}\sum_{i=0}^{N} U_i \tag{5.10}$$

Figure 5.2 is the simulation result of utilization for simultaneous distribution, staggered start, root does processing case.

**Progress**

As above, the progress is the percent of the total load that has been processed within a interval of time $T$. Here,

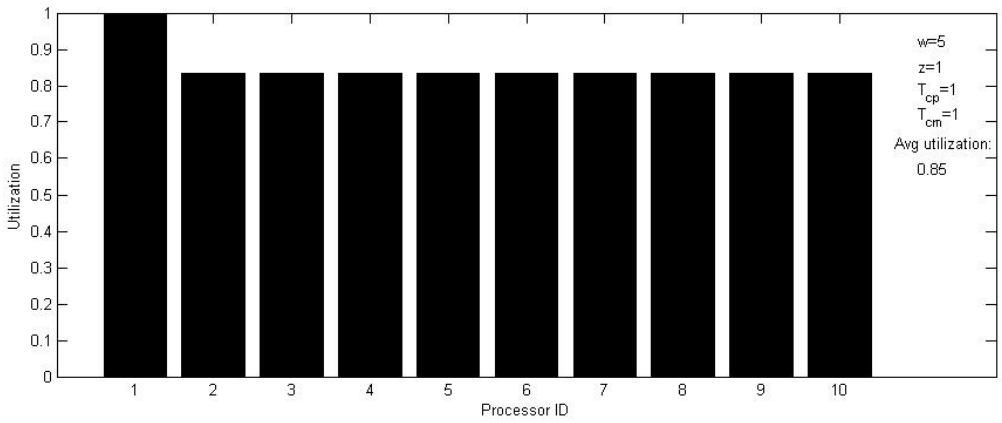


Figure 5.2: Utilization - Simultaneous distribution, staggered start, root does processing.

$$T_1 = \alpha_1 z_1 T_{cm} \tag{5.11}$$

$$T_2 = \alpha_2 z_2 T_{cm} \tag{5.12}$$

$$T_3 = \alpha_3 z_3 T_{cm} \tag{5.13}$$

.....

.....

.....

$$T_i = \alpha_i z_i T_{cm} \tag{5.14}$$

One can sort the vector and obtain $\hat{T}$ such that the elements in $\hat{T}$ are from small the large. Also, record the indexes and use it to creat a corresponding vector of $\hat{\alpha}$.

The percent of the load that processors $P_1$ through $P_N$ have computed within an interval can be given by the following equations.

$$\hat{T}_1 \leq T \leq \hat{T}_2 \qquad \frac{T - \hat{T}_1}{T_f - \hat{T}_1}\hat{\alpha}_1 \tag{5.15}$$

$$\hat{T}_2 \leq T \leq \hat{T}_3 \qquad \frac{T - \hat{T}_1}{T_f - \hat{T}_1}\hat{\alpha}_1 + \frac{T - \hat{T}_2}{T_f - \hat{T}_2}\hat{\alpha}_2 \tag{5.16}$$

$$\hat{T}_3 \leq T \leq \hat{T}_4 \qquad \frac{T - \hat{T}_1}{T_f - \hat{T}_1}\hat{\alpha}_1 + \frac{T - \hat{T}_2}{T_f - \hat{T}_2}\hat{\alpha}_2 + \frac{T - \hat{T}_3}{T_f - \hat{T}_3}\hat{\alpha}_3 \tag{5.17}$$

..... (5.18)

..... (5.19)

..... (5.20)

$$\alpha T\%_i = \sum_{m=1}^{i} \frac{T - \hat{T}_i}{T_f - \hat{T}_i}\hat{\alpha}_i \qquad \text{for } i = 1, 2, \cdots, N \tag{5.21}$$

$$\alpha T\% = \sum_{m=1}^{N} \alpha T\%_i \tag{5.22}$$

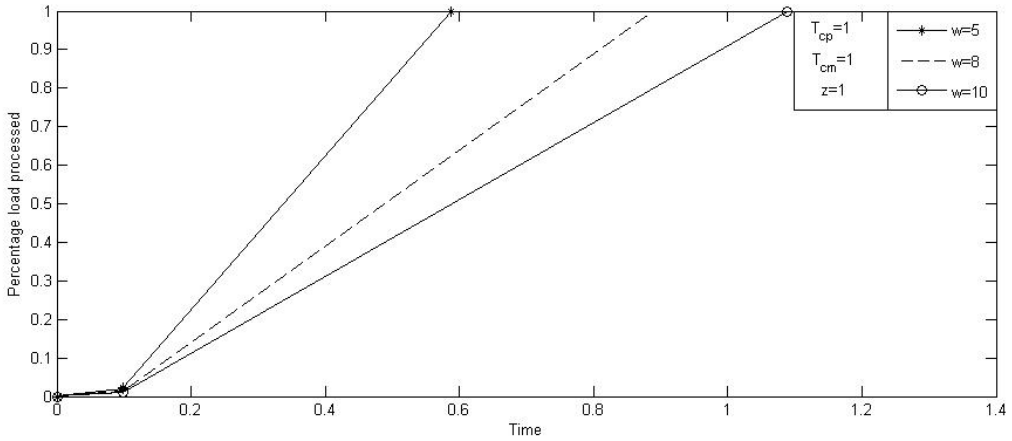


Figure 5.3: Progress - Simultaneous distribution, staggered start, root does processing

Where $T$ is in the interval of

$$\hat{T}_i \leq T \leq \hat{T}_{i+1} \qquad \text{for } i = 1, 2, \cdots, N-1 \tag{5.23}$$

And $T$ is in the interval of

$$\hat{T}_N \leq T \leq T_f \qquad \text{for } i = N \tag{5.24}$$

Figure 5.3 is the simulation of progress for simultaneous distribution, staggered start, root does processing case.

### 5.3.2 Sequential distribution, staggered start, root node does processing

#### Utilization

In Figure 5.4, the loads are sequentially distributed by the parent processor to the children processors. The children processors do not have front end processing. One can solve for utilization by the processors' computing time and finish time.

From figure 5.4

$$T_{f_1} = \alpha_1 z_1 T_{cm} + \alpha_1 w_1 T_{cp} \tag{5.25}$$

$$U_1 = \frac{\alpha_1 w_1 T_{cp}}{T_{f_1}} \tag{5.26}$$

$$U_1 = \frac{\alpha_1 w_1 T_{cp}}{\alpha_1 z_1 T_{cm} + \alpha_1 w_1 T_{cp}} \tag{5.27}$$

$$T_{f_2} = \alpha_1 z_1 T_{cm} + \alpha_2 z_2 T_{cm} + \alpha_2 w_2 T_{cp} \tag{5.28}$$

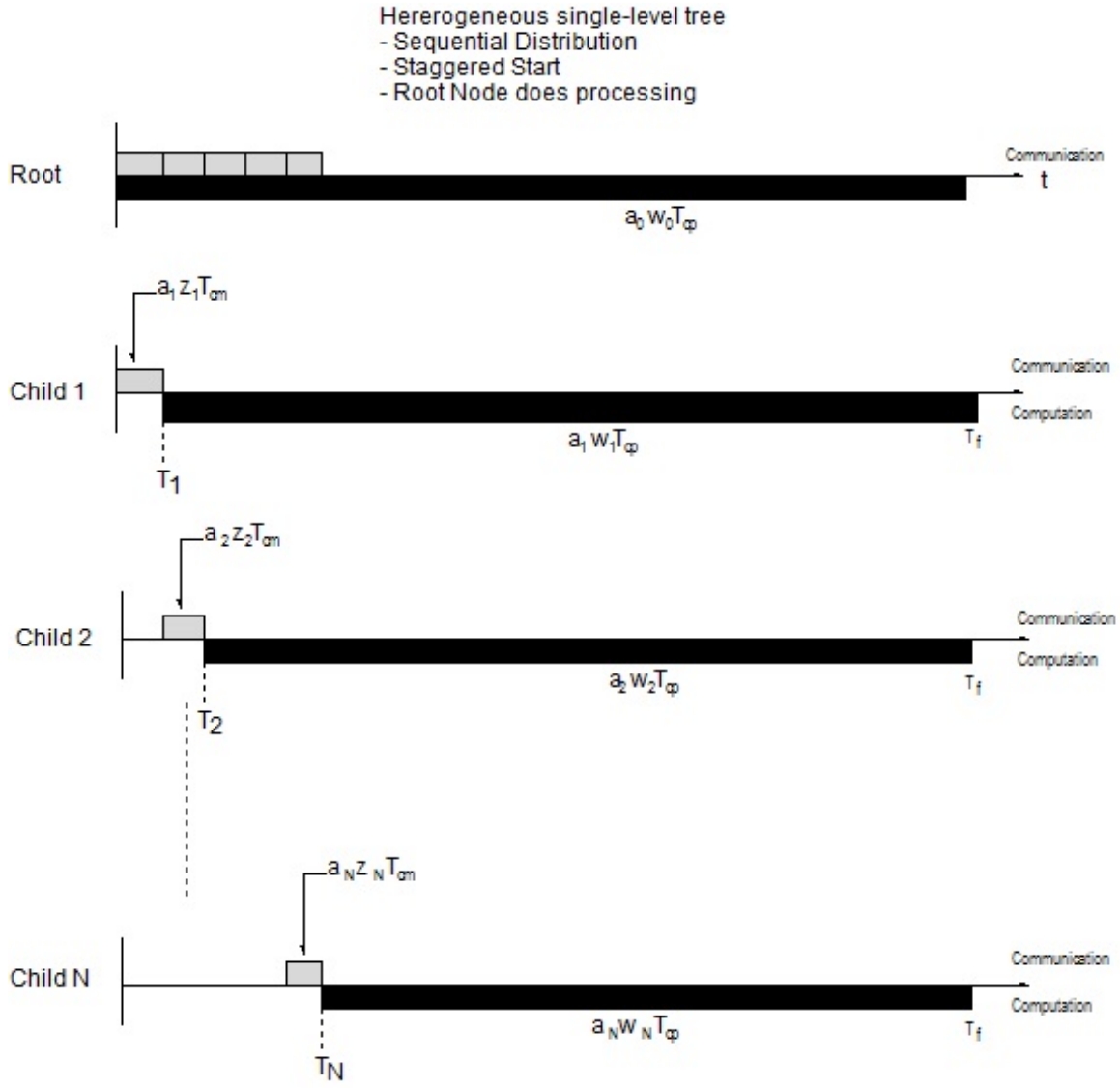


Figure 5.4: Sequential distribution, staggered start, root does processing.

$$U_2 = \frac{\alpha_2 w_2 T_{cp}}{T_{f_2}} \tag{5.29}$$

$$U_2 = \frac{\alpha_1 w_1 T_{cp}}{\alpha_1 z_1 T_{cm} + \alpha_2 z_2 T_{cm} + \alpha_2 w_2 T_{cp}} \tag{5.30}$$

$$T_{f_3} = \alpha_1 z_1 T_{cm} + \alpha_2 z_2 T_{cm} + \alpha_3 z_3 T_{cm} + \alpha_3 w_3 T_{cp} \tag{5.31}$$

$$U_3 = \frac{\alpha_3 w_3 T_{cp}}{T_{f_3}} \tag{5.32}$$

$$U_3 = \frac{\alpha_2 w_2 T_{cp}}{\alpha_1 z_1 T_{cm} + \alpha_2 z_2 T_{cm} + \alpha_3 z_3 T_{cm} + \alpha_3 w_3 T_{cp}} \tag{5.33}$$

.....

.....

.....

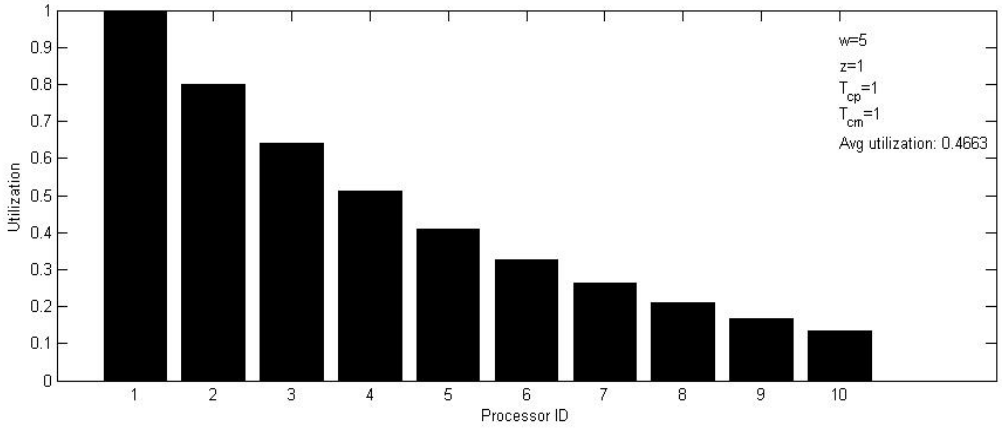


Figure 5.5: Utilization - Sequential distribution, staggered start, root does processing.

$$T_{f_i} = \alpha_i w_i T_{cp} + \sum_{p=1}^{i} \alpha_p z_p T_{cm} \tag{5.34}$$

$$U_i = \frac{\alpha_i w_i T_{cp}}{\alpha_i w_i T_{cp} + \sum_{p=1}^{i} \alpha_p z_p T_{cm}} \qquad \text{for i}=1, 2, \cdots, N \tag{5.35}$$

The general form for the utilization is:

$$AvgU = \frac{1}{N+1} \sum_{i=0}^{N} U_i \tag{5.36}$$

Figure 5.5 is the simulation result of utilization for sequential distribution, staggered start, root does processing case.

**Progress**

Here $T_i$ represents time point that processor $i$ starts to compute the load. Since this distribution has a staggered start, the loads are completely transmitted before the processors begin computations. So the load $\alpha_1$ of $T_1$ is entirely transmitted over link $z_1$ in the time $\alpha_1 z_1 T_{cm}$ before processor $P_1$ starts processing at time $T_1$. This process continues until all values from $T_1$ to $T_N$ are transmitted.

$$T_0 = 0 \tag{5.37}$$

$$T_1 = (\alpha_1 z_1) T_{cm} \tag{5.38}$$

$$T_2 = (\alpha_1 z_1 + \alpha_2 z_2) T_{cm} \tag{5.39}$$

$$T_3 = (\alpha_1 z_1 + \alpha_2 z_2 + \alpha_3 z_3) T_{cm} \tag{5.40}$$

$$\cdots\cdots \tag{5.41}$$

$$\cdots\cdots \tag{5.42}$$

$$\cdots\cdots \tag{5.43}$$

$$T_{f_i} = \alpha_i w_i T_{cp} + \sum_{p=1}^{i} \alpha_p z_p T_{cm} \tag{5.44}$$

The percent of the load that processors $P_1$ through $P_N$ have computed within an interval is defined as $\alpha T\%$, and given by the following equations.

$$T_0 \leq T \leq T_1 \qquad \frac{T-T_0}{T_f-T_0}\alpha_0 \tag{5.45}$$

$$T_1 \leq T \leq T_2 \qquad \frac{T-T_0}{T_f-T_0}\alpha_0 + \frac{T-T_1}{T_f-T_1}\alpha_1 \tag{5.46}$$

$$T_2 \leq T \leq T_3 \qquad \frac{T-T_0}{T_f-T_0}\alpha_0 + \frac{T-T_1}{T_f-T_1}\alpha_1 + \frac{T-T_2}{T_f-T_2}\alpha_2 \tag{5.47}$$

$$T_3 \leq T \leq T_4 \qquad \frac{T-T_0}{T_f-T_0}\alpha_0 + \frac{T-T_1}{T_f-T_1}\alpha_1 + \frac{T-T_2}{T_f-T_2}\alpha_2 \tag{5.48}$$

$$+\frac{T-T_3}{T_f-T_3}\alpha_3 \tag{5.49}$$

$$\cdots\cdots \tag{5.50}$$

$$\cdots\cdots \tag{5.51}$$

$$\cdots\cdots \tag{5.52}$$

$$\alpha T\%_i = \sum_{m=1}^{i} \frac{T-T_i}{T_f-T_i}\alpha_i \qquad \text{for } i = 0, 1, 2, \cdots, N \tag{5.53}$$

$$\alpha T\% = \sum_{m=0}^{N} \alpha T\%_i \tag{5.54}$$

Where $T$ is in the interval of

$$T_i \leq T \leq T_{i+1} \qquad \text{for } i = 0, 1, 2, \cdots, N-1 \tag{5.55}$$

And $T$ is in the interval of

$$T_N \leq T \leq T_f \qquad \text{for } i = N \tag{5.56}$$

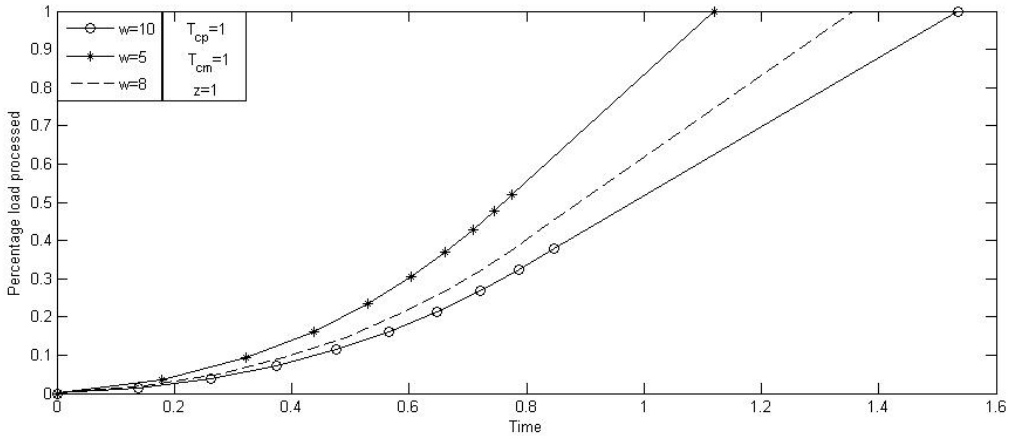


Figure 5.6: Progress - Sequential distribution, staggered start, root does processing.

Figure 5.6 is the simulation result of progress for sequential distribution, staggered start, root does processing case.

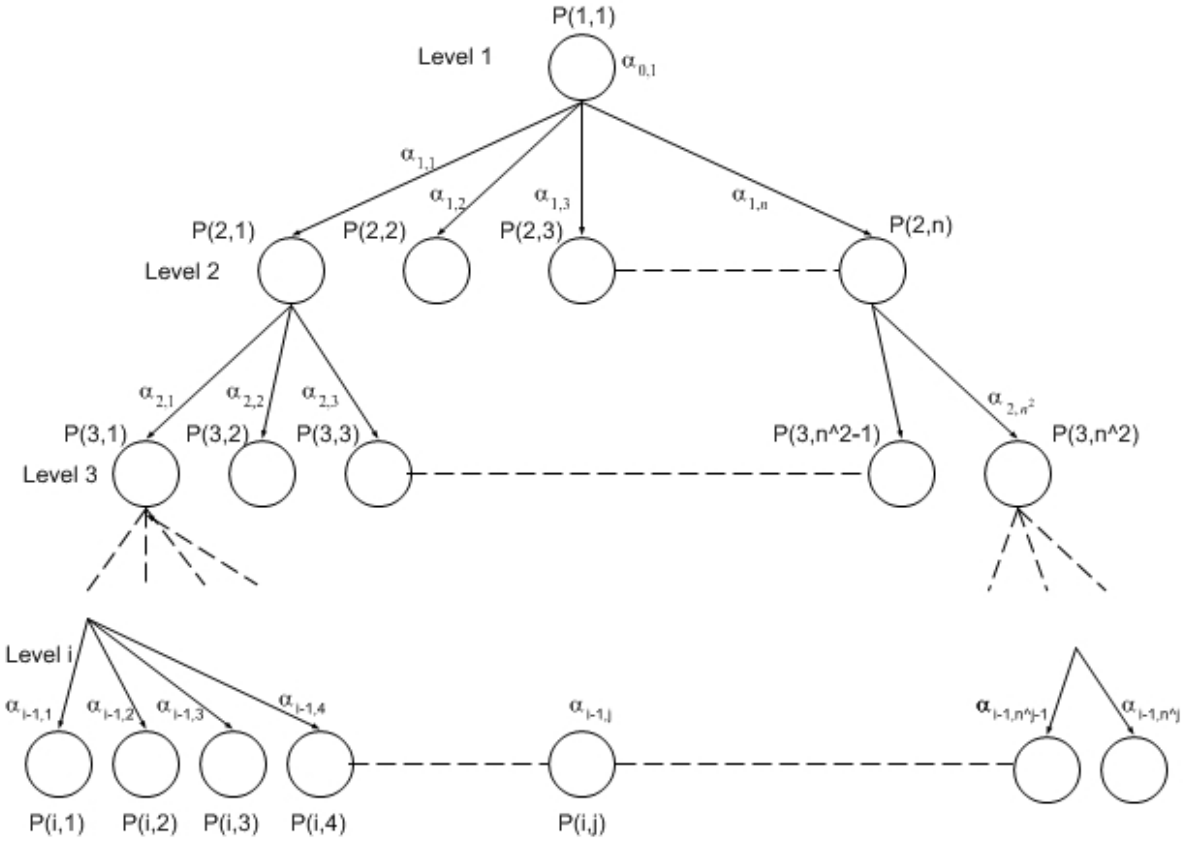


Figure 5.7: Multi-level tree

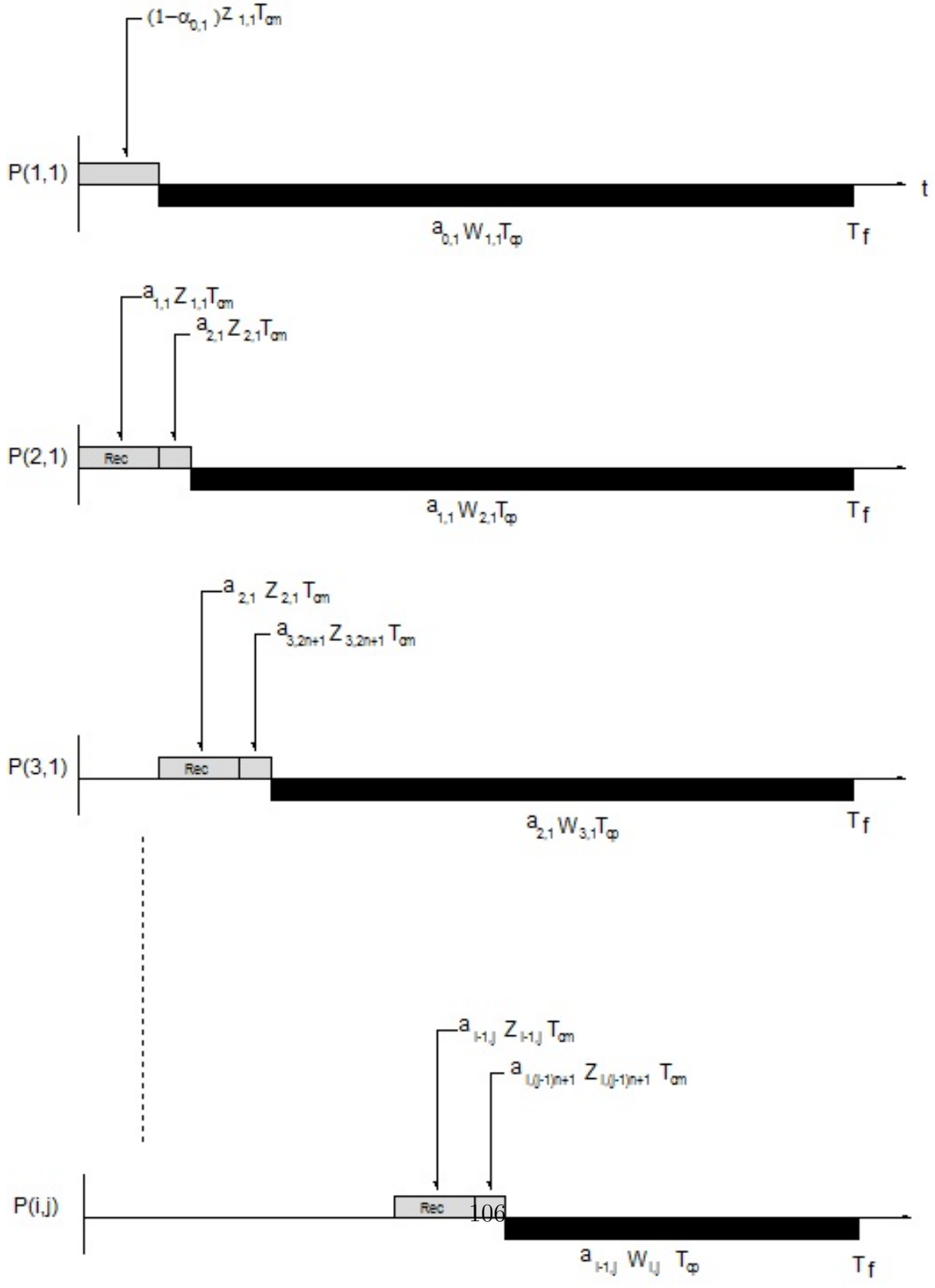


Figure 5.8: Simultaneous distribution, staggered start, root does processing.

## 5.4 Multi-level tree networks with divisible loads

### 5.4.1 Simultaneous distribution, staggered start

#### Optimal solution

For the first $N$ children in $ith$ level, one has the timing equations:

$$\beta_{i-1,1}z_{i-1,1}T_{cm} + \beta_{i-1,1}w_{eq_{i,1}}T_{cp} = \beta_{i-1,2}z_{i-1,2}T_{cm} + \beta_{i-1,2}w_{eq_{i,2}}T_{cp}$$
$$= \beta_{i-1,3}z_{i-1,3}T_{cm} + \beta_{i-1,3}w_{eq_{i,3}}T_{cp} = \cdots = \beta_{i-1,N}z_{i-1,N}T_{cm} + \beta_{i-1,N}w_{eq_{i,N}}T_{cp} \tag{5.57}$$

Here, $i$ is the number of level and $j = 1toN$. Here also, $N$ is the number of children nodes in a single subtree.

Also:

$$\beta_{i-1,1} + \beta_{i-1,2} + \beta_{i-1,3} + \cdots + \beta_{i-1,N} = 1 \tag{5.58}$$

So one has:

$$\beta_{i-1,j} = \frac{z_{i-1,j-1}T_{cm} + w_{eq_{i,j-1}}T_{cp}}{z_{i-1,j}T_{cm} + w_{eq_{i,j}}T_{cp}}\beta_{i-1,j-1} \tag{5.59}$$

and

$$\beta_{i-1,1} + \frac{z_{i-1,1}T_{cm} + w_{eq_{i,1}}T_{cp}}{z_{i-1,2}T_{cm} + w_{eq_{i,2}}T_{cp}}\beta_{i-1,1} + \cdots + \frac{z_{i-1,1}T_{cm} + w_{eq_{i,1}}T_{cp}}{z_{i-1,N}T_{cm} + w_{eq_{i,N}}T_{cp}}\beta_{i-1,1} = 1 \tag{5.60}$$

So

$$\beta_{i-1,1} = \frac{1}{\sum_{a=1}^{N} \frac{z_{i-1,1}T_{cm} + w_{eq_{i,1}}T_{cp}}{z_{i-1,a}T_{cm} + w_{eq_{i,a}}T_{cp}}} \tag{5.61}$$

One finds the $\beta$s (fraction of load for subtrees by themselves). For the connected tree one multiplies the $\beta$s to find the $\alpha$s.

### Utilization

Here, we assume the time for the data transferred to the next level is the time to transfer the load to its first child. The finish time equations are:

$$T_{f_{1,1}} = \alpha_{0,1} z_{0,1} T_{cm} + \alpha_{0,1} w_{1,1} T_{cp} \tag{5.62}$$

$$T_{f_{2,1}} = \alpha_{1,1} z_{1,1} T_{cm} + \alpha_{2,1} z_{2,1} T_{cm} + \alpha_{1,1} w_{2,1} T_{cp} \tag{5.63}$$

$$T_{f_{2,2}} = \alpha_{1,2} z_{1,2} T_{cm} + \alpha_{2,n+1} z_{2,n+1} T_{cm} + \alpha_{1,2} w_{2,2} T_{cp} \tag{5.64}$$

$$T_{f_{2,3}} = \alpha_{1,3} z_{1,3} T_{cm} + \alpha_{2,2n+1} z_{2,2n+1} T_{cm} + \alpha_{1,3} w_{2,3} T_{cp} \tag{5.65}$$

$$\cdots\cdots \tag{5.66}$$

$$\cdots\cdots \tag{5.67}$$

$$\cdots\cdots \tag{5.68}$$

$$T_{f_{2,j}} = \alpha_{1,j} z_{1,j} T_{cm} + \alpha_{2,jn+1} z_{2,jn+1} T_{cm} + \alpha_{1,j} w_{2,j} T_{cp} \tag{5.69}$$

$$T_{f_{3,j}} = \alpha_{1,ceil(j/n)} z_{1,ceil(j/n)} T_{cm} + \alpha_{2,j} z_{2,j} T_{cm} + \alpha_{3,jn+1} z_{3,jn+1} T_{cm} \tag{5.70}$$

$$+\alpha_{2,j} w_{3,j} T_{cp} \tag{5.71}$$

$$T_{f_{4,j}} = \alpha_{1,ceil(j/n^2)} z_{1,ceil(j/n^2)} T_{cm} + \alpha_{2,ceil(j/n)} z_{2,ceil(j/n)} T_{cm} \tag{5.72}$$

$$+\alpha_{3,j} z_{3,j} T_{cm} + \alpha_{4,jn+1} z_{4,jn+1} T_{cm} + \alpha_{3,j} w_{4,j} T_{cp} \tag{5.73}$$

$$\cdots\cdots \tag{5.74}$$

$$\cdots\cdots \tag{5.75}$$

$$\cdots\cdots \tag{5.76}$$

$$T_{f_{i,j}} = \alpha_{1,ceil[j/n^{(i-2)}]} z_{1,ceil[j/n^{(i-2)}]} T_{cm} + \alpha_{2,ceil[j/n^{(i-3)}]} z_{2,ceil[j/n^{(i-3)}]} T_{cm} \tag{5.77}$$

$$+\alpha_{3,ceil[j/n^{(i-4)}]} z_{3,ceil[j/n^{(i-4)}]} T_{cm} + \cdots + \alpha_{i-1,j} z_{i-1,j} T_{cm} \tag{5.78}$$

$$+\alpha_{i,jn+1} z_{i,jn+1} T_{cm} + \alpha_{i-1,j} w_{i,j} T_{cp} \tag{5.79}$$

$$= \sum_{L=1}^{i-1} \alpha_{L,ceil[j/n^{(i-L-1)}]} z_{L,ceil[j/n^{(i-L-1)}]} T_{cm} \tag{5.80}$$

$$+\alpha_{i,jn+1} z_{i,jn+1} T_{cm} + \alpha_{i-1,j} w_{i,j} T_{cp} \tag{5.81}$$

The utilization is:

$$U_{i,j} = \alpha_{i-1,j} w_{i,j} T_{cp} / T_{f_{i,j}} \tag{5.82}$$

The average utilization is:

$$\begin{aligned} AvgU &= \frac{1}{1+n+n^2+n^3+\cdots+n^{L-1}} \sum_{i=1}^{L} \sum_{1}^{n^{i-1}} U_{i,j} \\ &= \frac{1-n^L}{1-n} \sum_{i=1}^{L} \sum_{1}^{n^{i-1}} U_{i,j} \end{aligned} \tag{5.83}$$

Here, $L$ is the depth of the tree.

Figure 5.9 is the simulation result of utilization for simultaneous distribution, staggered start, root does processing case.

**Progress**

As previously:

$$\begin{aligned} T_{i,j} &= \alpha_{1,ceil[j/n^{(i-2)}]} z_{1,ceil[j/n^{(i-2)}]} T_{cm} + \alpha_{2,ceil[j/n^{(i-3)}]} z_{2,ceil[j/n^{(i-3)}]} T_{cm} \\ &+ \alpha_{3,ceil[j/n^{(i-4)}]} z_{3,ceil[j/n^{(i-4)}]} T_{cm} + \cdots + \alpha_{i-1,j} z_{i-1,j} T_{cm} + \alpha_{i,jn+1} z_{i,jn+1} T_{cm} \\ &= \sum_{L=1}^{i-1} \alpha_{L,ceil[j/n^{(i-L-1)}]} z_{L,ceil[j/n^{(i-L-1)}]} T_{cm} + \alpha_{i,jn+1} z_{i,jn+1} T_{cm} \end{aligned} \tag{5.84}$$

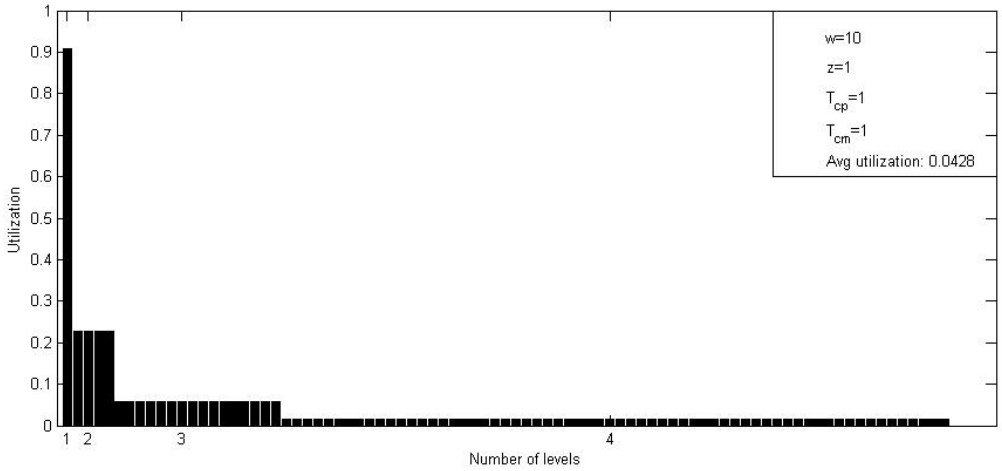


Figure 5.9: Utilization - Simultaneous distribution, staggered start, root does processing.

so:

$$\alpha T\% \, in \, level \, x = \sum [(T - T_{x,s_x})/(T_f - T_{x,s_x})\alpha_{x,s_x}] \tag{5.85}$$

And:

$$\begin{aligned} \alpha T\% &= \alpha T\% \, in \, level \, 1 + \alpha T\% \, in \, level \, 2 + \cdots + \alpha T\% \, in \, level \, x \\ &= \sum_{L=1}^{x} \sum [(T - T_{L,s_L})/(T_f - T_{L,s_L})\alpha_{L,s_L}] \end{aligned} \tag{5.86}$$

Figure 5.10 is the simulation result of progress for simultaneous distribution, staggered start, root does processing case.

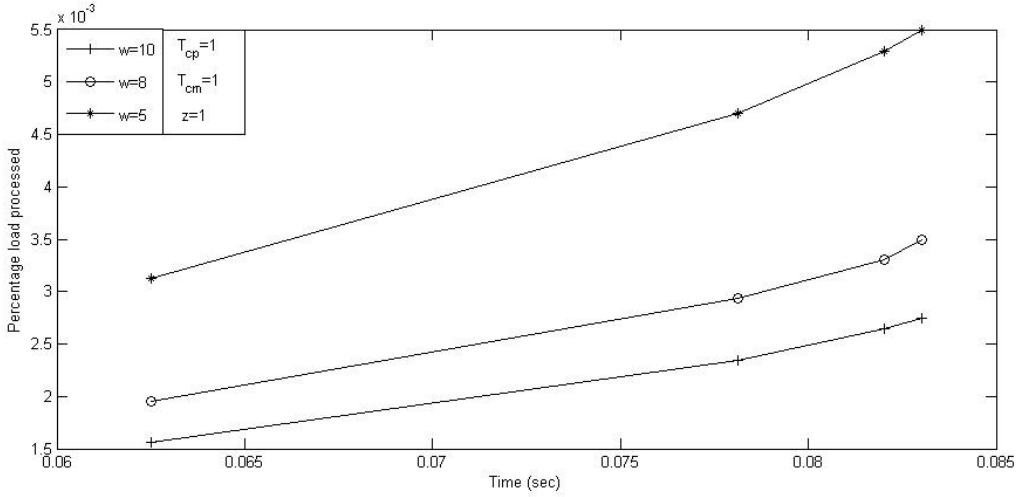


Figure 5.10: Progress - Simultaneous distribution, staggered start, root does processing.

### 5.4.2 Sequential distribution, staggered start

#### Optimal solution

One has the timing equations:

$$\beta_{i-1,j} z_{i-1,j} T_{cm} + \beta_{i-1,j} w_{eq_{i,j}} T_{cp} = \beta_{i-1,j-1} w_{eq_{i,j-1}} T_{cp} \tag{5.87}$$

Here, $i$ is the number of level and $j$=1 to $N$. Here also, $N$ is the number of children nodes in a single subtree.

that is:

$$\beta_{i-1,j} = \frac{w_{eq_{i,j-1}} T_{cp}}{z_{i-1,j} T_{cm} + w_{eq_{i,j}} T_{cp}} \beta_{i-1,j-1} = Q_{i,j-1} \beta_{i-1,j-1} \tag{5.88}$$

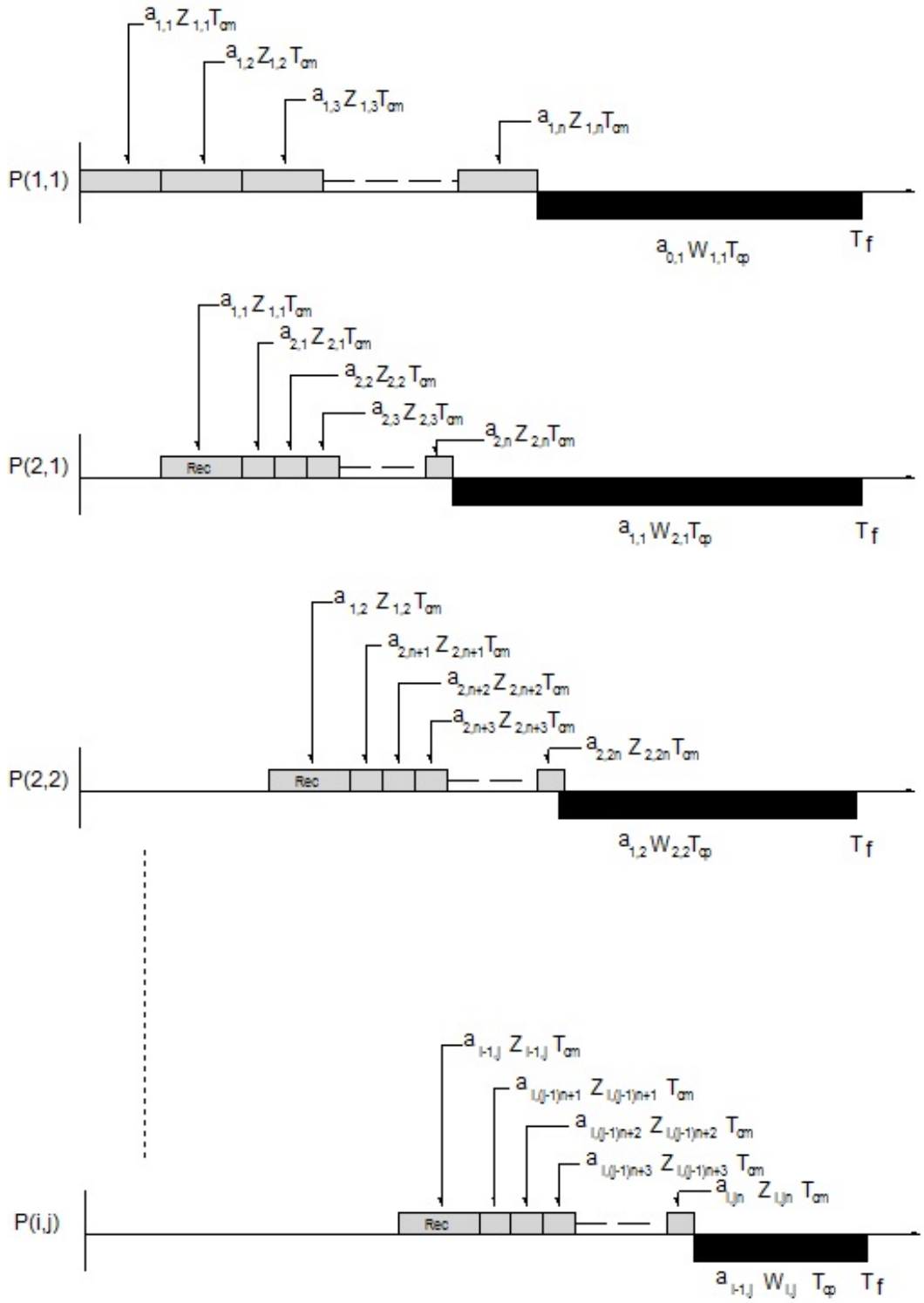




Figure 5.11: Sequential distribution, staggered start, root does processing.

where

$$Q_{i,j-1} = \frac{w_{eq_{i,j-1}} T_{cp}}{z_{i-1,j} T_{cm} + w_{eq_{i,j}} T_{cp}} \tag{5.89}$$

and the normalization equation:

$$\beta_{i-1,1} + \beta_{i-1,2} + \cdots + \beta_{i-1,N} = 1 \tag{5.90}$$

So one has:

$$\beta_{i-1,1} + Q_{i,2}\beta_{i-1,1} + Q_{i,2}Q_{i,3}\beta_{i-1,1} + \cdots + \prod_{N=1}^{n-1} Q_{i,N}\beta_{i-1,1} = 1 \tag{5.91}$$

$$\beta_{i-1,1} = \frac{1}{\sum_{N_1=1}^{n-1} \prod_{N_2=1}^{N_2=N_1} Q_{i,N_2}} \tag{5.92}$$

and

$$\beta_{i-1,j} = \prod_{N=1}^{j-1} Q_{i,N}\beta_{i-1,1} \tag{5.93}$$

From the $\beta$s of all the single level trees, one can get the $\alpha$s by multiplying the corresponding $\beta$s. Other subtrees can be solved by the same method.

**Utilization**

The finish time equations are:

$$T_{f_{1,1}} = \alpha_{1,1}z_{1,1}T_{cm} + \alpha_{1,2}z_{1,2}T_{cm} + \alpha_{1,3}z_{1,3}T_{cm} + \cdots \alpha_{1,n}z_{1,n}T_{cm} + \alpha_{0,1}w_{1,1}T_{cp}$$

$$T_{f_{2,1}} = \alpha_{1,1}z_{1,1}T_{cm} + \alpha_{2,1}z_{2,1}T_{cm} + \alpha_{2,2}z_{2,2}T_{cm} + \cdots + \alpha_{2,n}z_{2,n}T_{cm} + \alpha_{1,1}w_{2,1}T_{cp}$$

$$T_{f_{2,2}} = \alpha_{1,1}z_{1,1}T_{cm} + \alpha_{1,2}z_{1,2}T_{cm} + \alpha_{2,n+1}z_{2,n+1}T_{cm} + \alpha_{2,n+2}z_{2,n+2}T_{cm}$$
$$+\alpha_{2,n+3}z_{2,n+3}T_{cm} + \cdots + \alpha_{2,2n}z_{2,2n}T_{cm} + \alpha_{1,2}w_{2,2}T_{cp}$$

$$T_{f_{2,3}} = \alpha_{1,1}z_{1,1}T_{cm} + \alpha_{1,2}z_{1,2}T_{cm} + \alpha_{1,3}z_{1,3}T_{cm} + \alpha_{2,2n+1}z_{2,2n+1}T_{cm}$$
$$+\alpha_{2,2n+2}z_{2,2n+2}T_{cm} + \alpha_{2,2n+3}z_{2,2n+3}T_{cm} + \cdots + \alpha_{2,3n}z_{2,3n}T_{cm} + \alpha_{1,3}w_{2,3}T_{cp}$$

$$T_{f_{2,j}} = \alpha_{1,1}z_{1,1}T_{cm} + \alpha_{1,2}z_{1,2}T_{cm} + \alpha_{1,3}z_{1,3}T_{cm} + \cdots + \alpha_{1,j}z_{1,j}T_{cm}$$
$$+\alpha_{2,(j-1)*n+1}z_{2,(j-1)*n+1}T_{cm} + \alpha_{2,(j-1)*n+2}z_{2,(j-1)*n+2}T_{cm}$$
$$+\alpha_{2,(j-1)*n+3}z_{2,(j-1)*n+3}T_{cm} + \cdots + +\alpha_{2,j*n}z_{2,j*n}T_{cm} + \alpha_{1,j}w_{1,j}T_{cp}$$

$$T_{f_{3,j}} = \alpha_{1,1}z_{1,1}T_{cm} + \alpha_{1,2}z_{1,2}T_{cm} + \alpha_{1,3}z_{1,3}T_{cm} + \cdots + \alpha_{1,ceil(j/n)}z_{1,ceil(j/n)}T_{cm}$$
$$+\alpha_{2,[ceil(j/n)-1]*n+1}z_{2,[ceil(j/n)-1]*n+1}T_{cm} + \alpha_{2,[ceil(j/n)-1]*n+2}z_{2,[ceil(j/n)-1]*n+2}T_{cm}$$
$$+\cdots + \alpha_{2,j}z_{2,j}T_{cm} + \alpha_{2,[ceil(j/n)-1]*n+3}z_{2,[ceil(j/n)-1]*n+3}T_{cm}$$
$$+\alpha_{3,(j-1)*n+1}z_{3,(j-1)*n+1}T_{cm} + \alpha_{3,(j-1)*n+2}z_{3,(j-1)*n+2}T_{cm}$$
$$+\alpha_{3,(j-1)*n+3}z_{3,(j-1)*n+3}T_{cm} + \cdots + \alpha_{3,j*n}z_{3,j*n}T_{cm} + \alpha_{2,j}w_{3,j}T_{cp}$$
$$= \sum_{L=1}^{ceil(j/n)} \alpha_{1,L}z_{1,L}T_{cm} + \sum_{L=[ceil(j/n)-1]*n+1}^{j} \alpha_{2,L}z_{2,L}T_{cm} + \sum_{L=(j-1)*n+1}^{j*n} \alpha_{3,L}z_{3,L}T_{cm}$$
$$+\alpha_{2,j}w_{3,j}T_{cp}$$

$$T_{f_{4,j}} = \alpha_{1,1}z_{1,1}T_{cm} + \alpha_{1,2}z_{1,2}T_{cm} + \alpha_{1,3}z_{1,3}T_{cm} + \cdots + \alpha_{1,ceil(j/n^2)}z_{1,ceil(j/n^2)}T_{cm}$$
$$+\alpha_{2,[ceil(j/n^2)-1]*n+1}z_{2,[ceil(j/n^2)-1]*n+1}T_{cm} + \alpha_{2,[ceil(j/n^2)-1]*n+2}z_{2,[ceil(j/n^2)-1]*n+2}T_{cm}$$
$$+\alpha_{2,[ceil(j/n^2)-1]*n+3}z_{2,[ceil(j/n^2)-1]*n+3}T_{cm} + \cdots + \alpha_{2,ceil(j/n(}z_{2,ceil(j/n)}T_{cm}$$
$$+\alpha_{3,[ceil(j/n)-1]*n+1}z_{3,ceil(j/n)-1]*n+1}T_{cm} + \alpha_{3,[ceil(j/n)-1]*n+2}z_{3,ceil(j/n)-1]*n+2}T_{cm}$$
$$+\alpha_{3,[ceil(j/n)-1]*n+3}z_{3,ceil(j/n)-1]*n+3}T_{cm} + \cdots + \alpha_{3,[ceil(j/n)}z_{3,ceil(j/n)}T_{cm}$$
$$+\alpha_{4,(j-1)*n+1}z_{4,(j-1)*n+1}T_{cm} + \alpha_{4,(j-1)*n+2}z_{4,(j-1)*n+2}T_{cm}$$
$$+\alpha_{4,(j-1)*n+3}z_{4,(j-1)*n+3}T_{cm} + \cdots + \alpha_{4,j*n}z_{4,j*n}T_{cm} + \alpha_{3,j}w_{4,j}T_{cp}$$
$$= \sum_{}^{ceil(j/n^2)} \alpha_{}z_{}T_{} + \sum_{}^{ceil(j/n)} \alpha_{}z_{}T_{}$$

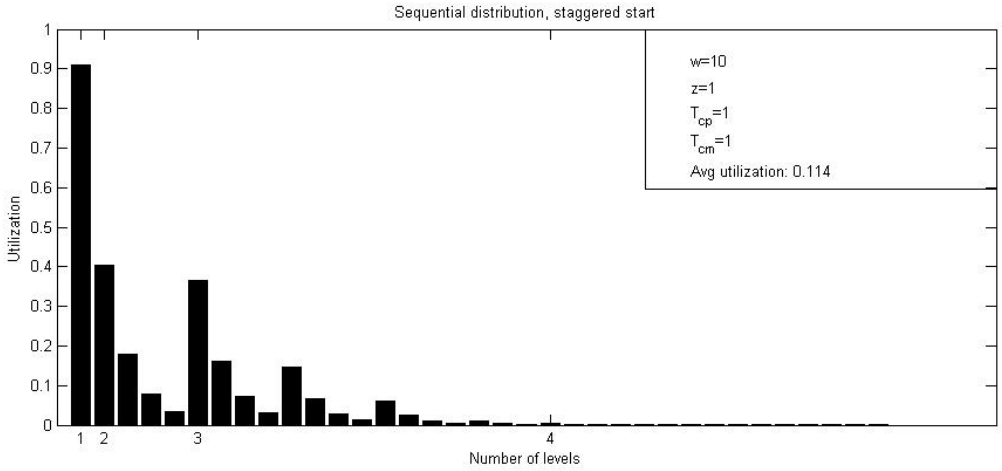


Figure 5.12: Utilization - Sequential distribution, staggered start, root does processing.

The utilization is:

$$U_{i,j} = \frac{\alpha_{i-1,j} w_{i,j} T_{cp}}{T_{f_{i,j}}} \tag{5.124}$$

The average utilization is:

$$\begin{aligned} AvgU &= \frac{1}{1 + n + n^2 + n^3 + \cdots + n^{L-1}} \sum_{i=1}^{L} \sum_{1}^{n^{i-1}} U_{i,j} \\ &= \frac{1 - n^L}{1 - n} \sum_{i=1}^{L} \sum_{1}^{n^{i-1}} U_{i,j} \end{aligned} \tag{5.125}$$

Here, $L$ is the depth of the tree.

Figure 5.12 is the simulation result of utilization for sequential distribution, staggered start, root does processing case.

**Progress**

$$T_{i,j} = \sum_{p=ceil[j/n^{(i-1)}]n+1}^{ceil[j/n^{(i-2)}]} \alpha_{1,p} z_{1,p} T_{cm} + \sum_{p=ceil[j/n^{i-2}]n+1}^{ceil[j/n^{i-3}]} \alpha_{2,p} z_{2,p} T_{cm} + \sum_{p=ceil[j/n^{(i-3)}]n+1}^{ceil[j/n^{(i-4)}]} \alpha_{3,p} z_{3,p} T_{cm}$$
$$+ \cdots + \sum_{p=ceil(j/n^2)n+1}^{ceil(j/n)} \alpha_{i-2,p} z_{i-2,p} T_{cm} + \sum_{p=ceil(j/n)n+1}^{j} \alpha_{i-1,p} z_{i-1,p} T_{cm} + \sum_{p=jn+1}^{(j+1)n} \alpha_{i,p} z_{i,p} T_{cm}$$
$$= \sum_{L=1}^{i-1} \sum_{p=ceil[j/n^{(i-L)}]n+1}^{ceil[j/n^{(i-1-L)}]} \alpha_{L,p} z_{L,p} T_{cm} + \sum_{p=jn+1}^{(j+1)n} \alpha_{i,p} z_{i,p} T_{cm} \tag{5.126}$$

$$\alpha T\% = \sum_{L=1}^{x} \alpha T\% \, in \, level \, L \tag{5.127}$$

As previously, $x$ is the largest number of levels which have been running and can be got from $T$ and $T_{i,j}$

$$\alpha T\% \, in \, level \, 1 = \frac{T - T_{1,1}}{T_f - T_{1,1}} \alpha_{0,1} \tag{5.128}$$

$$\alpha T\% \, in \, level \, 2 = \frac{T - T_{2,1}}{T_f - T_{2,1}} \alpha_{1,1} + \frac{T - T_{2,2}}{T_f - T_{2,2}} \alpha_{1,2} + \cdots + \frac{T - T_{2,m_2}}{T_f - T_{2,m_2}} \alpha_{1,m_2}$$
$$= \sum_{p=1}^{m_2} \frac{T - T_{2,p}}{T_f - T_{2,p}} \alpha_{1,p} \tag{5.129}$$

.....

.....

.....

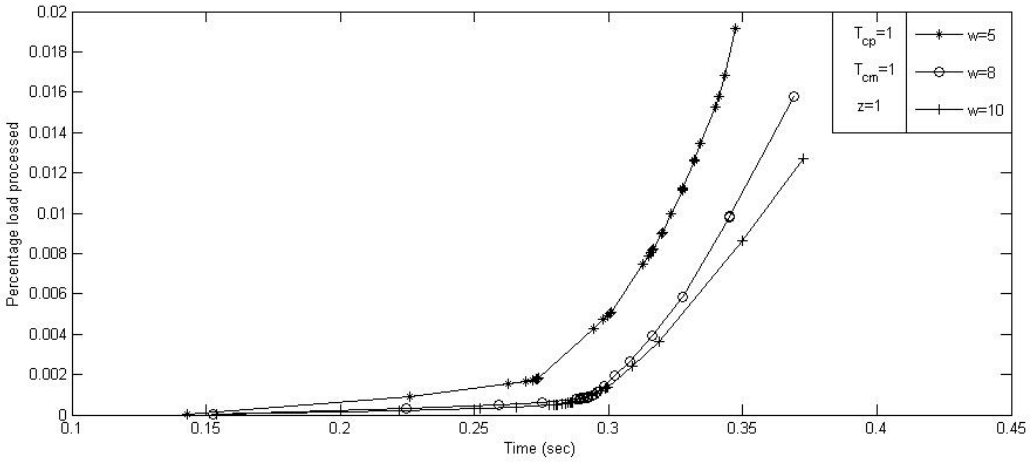


Figure 5.13: Progress - Sequential distribution, staggered start, root does processing.

$$\alpha T\% \, in \, level \, x = \sum_{p=1}^{m_x} \frac{T - T_{x,p}}{T_f - T_{x,p}} \alpha_{x-1,p} \tag{5.130}$$

So we get:

$$\begin{aligned} \alpha T\% &= \alpha T\% \, in \, level \, 1 + \alpha T\% \, in \, level \, 2 + \cdots + \alpha T\% \, in \, level \, x \\ &= \sum_{L=1}^{x} \sum_{p=1}^{m_L} [\frac{T - T_{L,p}}{T_f - T_{L,p}} \alpha_{L-1,p}] \end{aligned} \tag{5.131}$$

Here, $m$ is the set of processors that have been running in the $i$th level and can be got by $T$ and $T_{i,j}$.

Figure 5.13 is the simulation result of progress for sequential distribution, staggered start, root does processing case.

## 5.5 Conclusion

In this chapter, we discussed calculating two parameters for performance evaluation for single-level and multi-level trees with divisible loads: utilization and progress. For multilevel tree network, it was found that the utilization decreases as the number of levels increases. This indicates explicitly, as other authors have noticed, that there is a point of diminishing returns for speedup and make span in adding additional levels of processors in multi-level tree networks.

It was also found that the processors with staggered start have lower utilizations than the ones with simultaneous start. For the processors with simultaneous distribution, all the processors in the same level have same utilization. For the processors with sequential distribution, the processors in precedence have larger utilization.

The percentage of the load that has been processed has two periods: a transient one and a stable one. In the transient period, the processors are getting started level by level (simultaneous distribution) or one by one (sequential distribution). So the total processing speed increases during this short period of time. In the stable period, all the processors have been started and the processing speed is a constant.

Future work could involve are utilization and progress for other types of networks such as rings, 2D meshes, 3D meshes and hypercubes.

# Chapter 6

# Conclusion

In this thesis, the optimization for electrical grid systems and grid computing systems are studied. Chapter 2 introduces single phase load balancing problem. Different algorithms including brute force searching, greedy algorithm, heuristic algorithms and a dynamic programming algorithm we developed are implemented to solve this problem. These algorithms are compared and we found the dynamic programming is the most promising one in terms of its optimality and reasonable running time. In chapter 3, we extended the dynamic programming algorithm to solve three phase load balancing problem with spatial consideration. More details for three phase balancing problem are modeled and several variations on the basic algorithm are discussed. The heuristic algorithms can be superseded by the dynamic programming algorithm we proposed to solve three phase balancing problem. In chapter 4, a DLT - based scheduling algorithm for divisible loads with nonlinear communication time is investigated. This algorithm is implemented on single

level tree network with divisible loads and quadratic and cubit internal communication time for different distribution policies. An iterative method is used to obtain the numerical solutions. The simulation results show that this algorithm provides optimal solutions for grid computing systems where the makespan is the most crucial factor. In chapter 5, two novel performance measures for grid computing systems are proposed: utilization and progress. Single level and multi-level divisible loads scheduled trees with various distribution policies are studied and it was found that these two performance measures are of interest for grid computing systems.

The key lesson is that electrical companies can write software based on our dynamic programming algorithm to automatically analyze customers' electricity demand and make the optimal phase balancing arrangement instead of workers' intuitive decision, greedy algorithm and heuristic algorithms. Also, the Divisible Load Theory as an analytical tool is applied and optimal scheduling is obtained. It is found that DLT is a promising mathematical tool for grid computing systems and further studies are needed.

# Bibliography

[1] J. Zhu, MY. Chow and F. Zhang. IEEE Transaction on Power Systems, " Phase balancing using Mixed-Integer Programming". Vol. 13, No. 4, November 1998, pp. 1487-1492.

[2] J. Zhu, G. Bilbro and M. Chow. IEEE Transactions on Power Systems, "Phase Balancing using Simulated Annealing". Vol. 14, No. 4, November 1999, pp. 1508-1513.

[3] M. N. Gaffney. http://www.ndia-mich.org/workshop/Papers, "Intelligent Power Management: Improving Power Distribution in the Field".

[4] Y. David and R. Hasharon. United States Patent. "Apparatus for and method of evenly distributing an electrical load across an N-phase power distribution network". US Patent Num. 6018203. Jan. 25. 2000.

[5] Y. David. et al. United States Patent. "Apparatus for and method of evenly distributing an electrical load across a three phase power distribution network". US Patent Num. 5604385. Feb. 18. 1997.

[6] M. Gandomkar. 39th International Universities Power Engineering Conference, "Phase Balancing Using Genetic Algorithm". Sept, 2004, pp. 377-379.

[7] T. H Chen and J. T. Cherng. IEEE Transactions on Power Systems, "Optimal Phase Arrangement of distribution Transformers Connected to a Primary Feeder for System Unbalance Improvement and Loss Reduction Using a Genetic Algorithm". Vol. 15, NO. 3, August 2000, pp 994-1000.

[8] Chia-Hung Lin, Chao-Shun Chen, Hui-Jen Chuang and Cheng-Yu Ho. IEEE Transactions on Power Systems, "Heuristic rule-based phase balancing of distribution systems by considering customer load patterns". VOL. 20, NO. 2, May 2005. pp 709-716.

[9] M.-Y. Huang, C.-S. Chen, C.-H. Lin, M.-S. Kang, H.-J. Chuang and C.-W. Huang. IET Generation, Transmission and Distribution, "Three-phase balancing of distribution feeders using immune algorithm". 17th August 2007, pp. 383-392.

[10] Steven Skiena. "The Algorithm Design Manual" 2nd edition. Springer, 2008.

[11] K. Wang, S. Skiena and T.G. Robertazzi, Electric Power System Research, "Phase Balancing Algorithms", Volume 96, March 2013, Pages 218-224.

[12] Nikhil Gupta, Anil Swarnkar and K. R. Niazi. Power and Energy Society General Meeting, 2011 IEEE. "A novel strategy for phase balancing in three phase four wire distribution systems". July 2011.

[13] R.O. Duda and P.E. Hart, "Use of the Hough Transform to Detect Lines and Curves in Pictures, Communications of the ACM, vol. 15, 1972, pp. 11-15.

[14] http://www.ece.sunysb.edu/∼tom/.

[15] Cheng, Y.C. and Robertazzi, T.G., "Distributed Computation with Communication Delays", IEEE Transactions on Aerospace and Electronic Systems, Vol. 24, No. 6, Nov.1988, pp. 700-712.

[16] Agrawal, R. and Jagadish, H.V., "Partitioning Techniques for Large Grained Paralleism", IEEE Transactions on Computers, Vol. 37, No. 12, Dec. 1988, pp. 1627-1634.

[17] Robertazzi, T.G, "Ten Reasons to Use Divisible Load Theory", IEEE Computer, vol, 36, no. 5, pp. 63-68, 2003.

[18] Y.C. Cheng and T.G. Robertazzi, "Distributed Computation for a Tree Network with Communication Delays". IEEE Transactions on Aerospace and Electronic Systems, 26 511-516, 1990.

[19] S. Bataineh and T.G. Robertazzi, "Bus Oriented Load Sharing for a Network of Sensor Driven Processors". IEEE Transactions on Systems, Man and Cybernetics, 21 1202-1205, 1991.

[20] J. Blazewicz and M. Drozdowski, "Scheduling Divisible Jobs on Hypercubes". Parallel computing, 21 1945-1956, 1996.

[21] J. Blazewicz and M. Drozdowski, "The Performance Limits of a Two Dimensional Network of Load Sharing Processors". Foundations of Computing and Decision Sciences, 21 3-15,1996.

[22] T.G. Robertazzi. "Processor Equivalence for a Linear Daisy Chain of Load Sharing Processors". IEEE Transactions on Aerospace and Electronic Systems, 29:1216-1221, 1993.

[23] V. Bharadwaj, D. Ghose, V. Mani, "Multi-installment Load Distribution in Tree Networks with Delays". IEEE Transactions on Aerospace and Electronic Systems, 31 555-567 (1995).

[24] Y. Yang, H. Casanova, "UMR: A Multi-Round Algorithm for Scheduling Divisible Workloads". Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03), Nice, France, 2003.

[25] O. Beaumont, A. Legrand, and Y. Robert, "Optimal Algorithms for Scheduling Divisible Workloads on Heterogeneous Systems". 12th Heterogeneous Computing Workshops HCW'2003, 2003.

[26] J. Blazewicz and M. Drozdowski, "Distributed Processing of Distributed Jobs with Communication Startup Costs. Discrete Applied Mathematics", 76 21-41, 1997.

[27] A.L. Rosenberg, "Sharing Partitionable Workloads in Heterogeneous NOWs: greedier is not better". In D.S. Katz, T. Sterling, M. Baker, L. Bergman, M. Paprzycki, and R. Buyya, editors. Cluster Computing 2001 pp. 124-131, 2001.

[28] D. Ghose and H. J. Kim, "Load Partitioning and Trade-Off Study for Large Matrix Vector Computations in Multicast Bus Networks with Communication Delays", Journal of Parallel and Distributed Computing, vol. 54, 1998.

[29] P.F. Dutot, "Divisible Load on Heterogeneous Linear Array". Proceeding of the International Parallel and Distributed Processing Symposium (IPDPS'03), Nice, France, 2003.

[30] M. Drozdowski and P. Wolniewicz, "Out-of-core Divisible Load Processing", IEEE Transactions on Parallel and Distributed Systems, vol. 14, 2003, pp. 1048-1056.

[31] J.T. Hung and T.G. Robertazzi, "Scheduling Nonlinear Computational Loads", IEEE Transactions on Aerospace and Electronic Systems, vol. 44, no. 3, July 2008, pp. 1169-1182.

[32] S. Suresh, C. Run, H.J. Kim, T.G. Robertazzi and Y.-I. Kim, "Scheduling Second Order Computational Load in Master-Slave Paradigm", IEEE Transactions on Aerospace and Electronic Systems, vol. 48, no. 1, Jan 2012, pp. 780-793.

[33] S. Suresh, H.J. Kim, C. Run, and T.G. Robertazzi, "Scheduling Nonlinear Divisible Loads in a Single Level Tree Network", The Journal of Supercomputing, vol. 61, no. 3, Sept. 2012, pp. 1069-1088.

[34] Wong H.M. and B. Veeravalli, "Aligning Biological Sequences on Distributed Bus Networks: A Divisible Load Scheduling Approach," IEEE Transactiosn on Information Technology in BioMedicine, vol. 9, no. 4, Dec. 2005, pp. 489-501.

[35] Ping, D.L.H., Veeravalli, B. and Bader, D., "On the Design of High-Performance Algorithms for Aligning Multiple Protein Sequences in Mesh-Based Multiprocessor Architectures," Journal of Parallel and Distributed Computing, vol. 67, no. 9, 2007, pp. 1007-1017.

[36] Barlas, G.D., "An Analytical Approach to Optimizing Parallel Image Registration," IEEE Transactions on Parallel and Distributed Systems, vol. 21, no. 8, Aug. 2010, pp. 1074-1088.

[37] Berlinska, J. and Drozdowski, M., "Scheduling Divisible MapReduce Computations," Journal of Parallel and Distributed Computing, vol. 71, no. 3, March 2011, pp. 450-459.

[38] Gamboa, C.F. and Robertazzi, T.G., "Simple Performance Bounds for Multicore and Parallel Channel Systems," Parallel Processing Letters, vol. 21, no. 4, Dec 2011, pp. 439-459.

[39] Barlas, G., "Cluster-Based Optimized Parallel Video Transcoding," Parallel Computing, vol. 38, March 2012, pp. 226-244.

[40] D. Ghose,"A Feedback Strategy for Load Allocation in Workstation Clusters with Unknown Network Resource Capabilities Using the DLT Paradigm", Proc. 2002 Int'l Conf. Parallel and Distributed Processing Techniques and Applications (PDPTA 02), vol. 1, CSREA Press, 2002, pp. 425-428

[41] J. Dongarra and W. Gentzsch, Computer Benchmarks. NorthHolland, 1993.

[42] D. Feitelson, "Performance Evaluation Links," http://www.cs.huji.ac.il/ feit/perf.html.Mar.2009.

[43] R.W.Hockney, The Science of Computer Benchmarking. SIAM, 1996.

[44] R. Jain, The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling. John Wiley Sons, 1991.

[45] http://www.ece.sunysb.edu/∼tom/.

[46] Agrawal, R. and Jagadish, H.V., "Partitioning Techniques for Large Grained Paralleism", IEEE Transactions on Computers, Vol. 37, No. 12, Dec. 1988, pp. 1627-1634.

[47] Berlinska, J. and Drozdowski, M., "Scheduling Divisible MapReduce Computations, Journal of Parallel and Distributed Computing, vol. 71, no. 3, March 2011, pp. 450-459.

[48] Bharadwaj, V., Ghose, D. and Mani, V., "Multi-installment Load Distribution in Tree Networks with Delays", IEEE Transactions on Aerospace and Electronic Systems, Vol. 31, No. 2, April 1995, pp. 555-567.

[49] Drozdowski, M. and Wielebski, L., "Isoefficiency Maps for Divisible Computations," IEEE Transactions on Parallel and Distributed Systems, vol. 21, no. 6, June 2010, pp. 872-880.

[50] http://dspace.sunyconnect.suny.edu/bitstream/handle/1951/45461/000000387.sbu.pdf?seque