NOTE TO USERS

Université d'Ottawa · University of Ottawa

Université d'Ottawa · University of Ottawa

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

Xin LU

AUTEUR DE LA THÈSE - AUTHOR OF THESIS

Master of Computer Science

GRADE - DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT - FACULTY, SCHOOL, DEPARTMENT

TITRE DE LA THÈSE - TITLE OF THE THESIS

An Efficient Paralell Optimization Algorithm for the Token Bocket Control Mechanism

N. Ahmed

DIRECTEUR DE LA THÈSE - THESIS SUPERVISOR

CO-DIRECTEUR DE LA THÈSE - THESIS CO-SUPERVISOR

EXAMINATEURS DE LA THÈSE - THESIS EXAMINERS

C. Huang

D. Inkpen

J.-M. De Koninck, Ph.D.

LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

DEAN OF THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

An Efficient Parallel Optimization Algorithm for the Token Bucket Control Mechanism

by

Xin Lu

A thesis submitted in conformity with the requirements
for the degree of
Master of Computer Science

School of Graduate Studies and Research
Ottawa-Carleton Institute for Computer Science
School of Information Technology and Engineering (SITE)
University of Ottawa

May 2004

Your file Votre référence
ISBN: 0-494-01539-X
Our file Notre référence
ISBN: 0-494-01539-X

Abstract

The Token Bucket algorithm, one of the most widely used control mechanism nowadays, has been widely studied to ensure the QoS needs of various applications. However, one main drawback of current models of this algorithm is that most of them have focused on a single Token Bucket system.

In this thesis, based on previous research efforts, we propose a parallel solution to the multiple Token Bucket model. We also develop a Reduced Memory Algorithm to decrease the algorithm's memory requirements at the cost of extra computation time. We test our parallel processing algorithm using two sets of traces. Our numerical results show that the model can effectively solve the multiple Token Bucket problems. Besides showing the benefits of using a parallel processing platform, our results also provide us with the guidelines to configure the parallel processing platform.

Acknowledgements

I would like to express my deeply-felt gratitude to my thesis supervisors Dr L. Orozco-Barbosa and Dr. N.U.Ahmed, for their guidance, understanding and inspiration throughout the completion of my thesis, and especially for Dr L. Orozco-Barbosa's and Dr. N.U.Ahmed's perseverance during iterations of this dissertation.

My special thanks are due to Ms. Bo Li for her patient and valuable discussions and suggestions to this project.

Contents

List of Tables

List of Figures

ACRONYMS

CBR	Constant Bit Rate
DiffServ	Differentiated Services
DP	Dynamic Programming
GA	Genetic Algorithm
GOP	Group Of Pictures
MPI	Message Passing Interface
PE	Processing Element
PGA	Parallel Genetic Algorithm
PVM	Parallel Virtual Machine
QoS	Quality of Service
TB	Token Bucket
VBR	Variable Bit Rate

NOTATIONS

B_i	TB capacities
c	Memory in bytes for one state
C	Output link rate to the network
d	Time of calculating one state on one PE
D_i	Link rate between i_{th} TB and the multiplexor
g	Number of generations of GA
J	System losses
L	Losses at multiplexor
k	Time index
m	Number of states in one stage
n	Number of stages
N	Number of TBs
p	Number of PEs
q	Waiting losses at multiplexor
Q	Multiplexor capacity
r_i	Losses at TBs
$rank$	The index of a PE
s	Population size in GA
T	Token size
$u(k)$	TB generation policy at time t_k
u	A vector represents TB generation policy from time t_0 to t_{n-1}

x	Skips in Reduced Memory Algorithm
X	A vector stores the states of TBs and multiplexer
α	Weight to TB losses
β	Weight to multiplexor losses
γ	Weight to waiting losses
τ	Average time interval of the arrival of one packet

Chapter 1

Introduction

1.1 Background

During the last decade, we have witnessed the design and deployment of a large number of novel computer-based applications. We have also been witnesses of the development of the Internet: the network of networks. The development and deployment of these two complementary technologies have given birth to the world-wide information highway. Even though we now use this infrastructure on a day-to-day basis, there are still a large number of open issues to be addressed. Due to the huge and increasing demand for its services and the nature of the applications being deployed, traffic control is one of those major issues to be addressed.

In order to understand the trends on the design of traffic control mechanisms, it is essential to take a quick look to the development in the area of information technologies and communications of the last few years. Multimedia communications, such as video conferences and voice communications are now widely deployed. These applications are characterized by their stringent Quality of Service (QoS) requirements and high network-resources demands. The wide deployment of these high resource-demanding applications combined with the development of

high-performance computer communications has spurred the need of designing novel traffic control mechanisms. It is well understood by developers and designers that these mechanisms have to be both simple and effective in meeting the application requirements when deployed in networks supporting a wide variety of applications. Many different network control schemes have been studied to meet different levels of QoS requirements. The Token Bucket Algorithm, one of the best-known control schemes, has proved to be an effective method to shape and police the network traffic.

In [15] [23] [32], N.U.Ahmed, L.Orozco Barbosa, BoLi, H. Yan and Q. Wang developed a dynamic model based on the Token Bucket Algorithm. The model comprises two traffic sources and two Token Buckets connected to a multiplexor located at the access network. An objective function was defined regarding the possible packet losses in the network; a key performance metric for a system aiming to provide various QoS levels. Under the proposed model, the token generation rate is varied taking into account several system parameters, such as the state of controller (the Token Bucket occupancy), the queue length of the multiplexor, among others, as well as the statistical characteristics of the incoming traffic. The proposed algorithm optimizes the token generation rate using the principle of Dynamic Programming (DP). Due to the huge size of the search space, the search process was enhanced by using a Genetic Algorithm (GA). Numerical results showed the effectiveness of the optimization methodology. The GA proved particularly useful in speeding up the optimization process. The results also showed the effectiveness of the Token Bucket mechanism.

1.2 Motivation

The motivation of this research is to develop a dynamic parallel optimization algorithm for

the Token Bucket control mechanism. The study represents a step further on previous research efforts. The main highlights of the work herein are:

First, most studies in [17] [9] [4] [5] focused on a single Token Bucket. In [15] [23] [32], the authors have developed a model with two buckets and one multiplexor. However, due to the huge memory and processing requirements the number of Token Buckets was limited to two. In this thesis, a parallel algorithm has been developed allowing to solve larger systems (multiple Token Buckets).

Second, previous research has not attempted enough to optimize the computation time and memory requirements. By using multiple processor elements (PEs), the computation procedure can be speeded up. Furthermore, the Reduced Memory Algorithm is introduced into the model to decrease the memory demands at the cost of an extra fixed rate computation.

Third, past studies assumed MPEG-1 as the prevalent digital video encoding scheme on the Internet. However, MPEG-4 is the main trend nowadays.

Fourth, past works have not endeavored to investigate in-depth the Token Bucket parameters. How parameters, such as the token size, influence the dynamics of the controller are studied.

In summary, our contribution for this research is to establish a general Token Bucket control system. Meanwhile, space and time requirements of this system are optimized by the parallel processing and Reduced Memory Algorithm. Furthermore, experiments are evaluated with main tread traffics and in-depth study on parameters is conducted.

1.3 Thesis Organization

This thesis is organized as the follows. Chapter 1 is the introduction and our motivation for this research. Chapter 2 reviews the principles of Dynamic Programming and Genetic Algorithm,

Reduced Memory Algorithm and Parallel Computing. Chapter 3 reviews the work done in the area of the Token Bucket traffic control mechanism, followed by an analysis of the mathematical model developed by N.U.Ahmed and BoLi et al. [15] [23] [32] and briefly analyzes its complexity. Chapter 4 details the proposed parallel Token Bucket optimization algorithms. It also analyzes the computation time and memory requirements of the optimization algorithm. Chapter 5 provides and analyzes our numerical results. Chapter 6 summarizes the thesis and provides a list of open issues for further study.

Chapter 2

Optimization Methodologies and Parallel Processing Principles

In this chapter, we review the principles on Dynamic Programming, Genetic Algorithms, Reduced Memory Algorithms, and Parallel Processing technologies. These principles together with the Token Bucket Model, developed by N.U.Ahmed and BoLi et al. [15] [23] [32], to be reviewed in the next chapter, are the basis of the efficient parallel algorithm being developed in this thesis.

2.1 Dynamic Programming

Dynamic programming (DP) was first introduced in [3] by Richard Bellman in 1957. Even today, DP is still a powerful optimization methodology. The essence of dynamic programming is Richard Bellman's *Principle of Optimality* [3]:

An optimal policy has the property that whatever the initial state and the initial decisions are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

The proof of Bellman Equation is given in *Appendix A*.

The basic principle of Dynamic Programming (DP), consists in rewriting discrete-time dynamic optimization problems in an equivalent format. More specifically, DP solves a multi-variable problem by redefining a set of single-variable problems. DP obtains the overall solution bottom up by synthesizing it from small acquired sub-solutions. It tries many possible subsets and choices before determining the optimal value.

There are a number of common characteristics in DP problems: i) The overall problem is divided into time-dependent *stages* with a optimal decision required at each stage. ii) Each stage has a number of associated *states*. iii) The decision at the i_{th} stage transforms one state into another state of the next stage $i+1$. iv) Optimal decisions for the current state are independent of the previous states. v) There exists a recursive relationship between the decision for the $i+1_{th}$ stage and the given solution of the i_{th} stage. vi) The final stage must be solvable by itself. The last two characteristics are closely tied up into a recursive relationship. Through these characteristics, it is easy to see that DP involves two main steps to obtain the optimal solution.

- The Filling or calculation step. The algorithm computes state decision values stage by stage and saves them into a matrix data structure.
- The Traceback step. This step consists of constructing an optimal solution using the information stored in the matrix.

The recursive relationship between states of neighboring stages is defined as follows:

$$\begin{cases} X(k+1) \equiv F(k, X(k), u(k)) & k = 0, 1, 2, \cdots, n-1; \\ X(0) = X_0 & . \end{cases}$$

where, X denotes the state variable at stage k, $u(k)$ is the decision value and F denotes a function on the variable $k, X(k)$ and $u(k)$. However, this equation only defines the state transition between stages k_{th} and $k+1_{th}$. In order to minimize a performance metric, such as the number of lost packets, by varying the decision value (token generation rate) $u(k)$, we need to define a recursive cost function. Let V denote the cost value and W denote the expected cost at the end of stage n. Using the basic arguments of dynamic programming one can prove that the value function V must satisfy the following recursive equation:

$$\begin{cases} V(k, X(k)) \equiv \inf\{V(k, X(k), u(k)) + V(k+1, F(k, X(k), u(k))\} & k = 0, 1, 2, \cdots, n-1; \\ V(n, X(n)) \equiv W(n, X(n)) & . \end{cases}$$

This is known as the Bellman equation of dynamic programming.

A simple example for a dynamic programming application is calculating fibonacci numbers. A generalization of the Fibonacci number F are defined recursively by:

$$\begin{cases} F(0) & = 0 \\ F(1) & = 1 \\ F(i+2) & = F(i+1) + F(i) \end{cases}$$

Fibonacci number starts with $F(0)$ and $F(1)$, and then produces the next Fibonacci number by adding the two previous Fibonacci numbers. The i_{th} fibonacci number is calculated on the formula $F(i) = F(i-1) + F(i-2)$. Without dynamic programming, the recursive pseudo-code is given by:

```
function F(i)
    if (n is 0 or 1)
        return n
    otherwise
        return the value of F(i − 1) + F(i − 2)
end function
```

For Fibonacci number, the recurrence relation is evident thus dynamic programming is applicable. It is probably obvious that the above function wastes computation on recalculating the values of fibonacci numbers like $F(0)$ and $F(1)$. Dynamic programming can eliminate this need. Thus, the following pseudo-code is generated:

function $F(i)$

    allocate array F[ ]
    $F[0] = 0$
    $F[1] = 1$

    for $i = 2$ to i do
        $F[i] = F[i-1] + F[i-2]$
    end for
    return $F(i)$
end function

By applying dynamic programming, this algorithm is much more efficient with the cost of a fixed memory requirement. This new function breaks the overall problem $F(i)$ into multiple stages from stage 0 to stage i. In each stage, it optimizes the current $F(i)$. To Fibonacci number, this optimization value is the sum of $F[i-1]$ and $F[i-2]$. This example shows the basic idea of dynamic programming.

Due to the popularity and effectiveness of DP as an optimization methodology, extensive research efforts have been undertaken aiming to reduce its memory and running time requirements. For instance, due to the high dimensional feature of the Token Bucket algorithm, N.U.Ahmed and BoLi et al. used an Iterative Dynamic Programming algorithm combined with the use of Genetic Algorithms to solve this equation. The experiments reported in BoLi et al. [15] showed that this approach can significantly reduce the time required to find the optimal solution.

Regarding the memory requirement issues, in one of the most relevant works, a family of checkpoint-based reduced space algorithms have been developed by J A. Grice et al. [12]. These algorithms have been used to solve various DP-related problems, such as finding the optimal

character-by-character correspondence between two sequences. The family of reduced space algorithms typically requires a slowdown on the runtime in exchange for less memory space. In [12], the authors described various algorithms, such as, the 2-Level checkpoints, Viterbi checkpoints, local and semi-local checkpoints, and multi-level checkpoint algorithms.

In this thesis, we go a step further towards solving the optimization problem of the Token Bucket algorithm by addressing its memory and running time requirements. We are particularly interested in solving the multiple-token bucket system. In the following and for the sake of completeness, we reviewed the principles of two methodologies aiming to reduce the time and space requirements of dynamic programming: Genetic Algorithms and Reduced Memory Algorithms.

2.2 Genetic Algorithm

Genetic algorithm (GA) [10], a brainchild of John Holland and his students at the University of Michigan in the 1970s, was a computational model inspired by evolution ideas. This algorithm encodes potential solutions on a chromosome-like data structure called "population". Operators can recombine individuals (chromosomes) in the population and create children or new individuals, then insert them into the population. GA can solve a problem that does not have a precisely defined solution, or a problem that takes long time when following the traditional method. Therefore, many applications have been oriented or interested in GA as one of the most powerful optimization tools.

GA consists of the following steps [24]:

1. Encoding potential solutions into chromosomes. The most common scheme is binary encoding which presents the solution in a string of bits;
2. Generate a random population that is a small set of solutions of chromosomes;
3. Evaluate the objective function $f(z)$ for each chromosome z in the population;

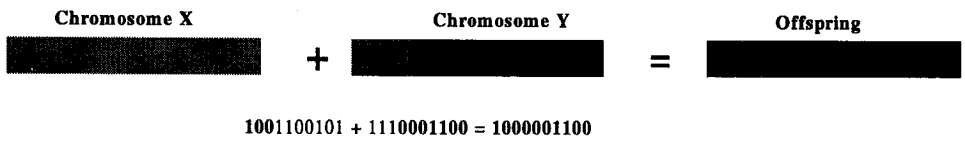


Figure 2.1: Recombination

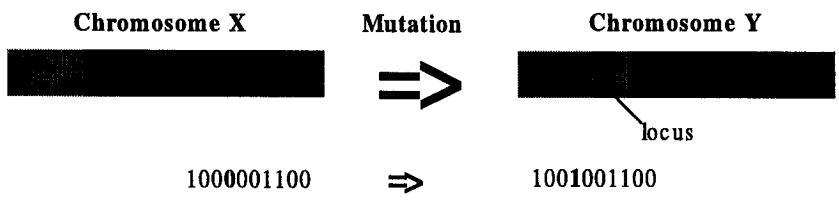


Figure 2.2: Random Mutation

4. If the stopping criteria is satisfied, then the algorithm stops, and returns the solution;

5. If the stopping criteria is not satisfied, then the algorithm restarts to generate a new population;

   5.1 Selection: selecting two parent chromosomes in a population. The better the fit, the bigger the chance of the selection.

   5.2 Recombination: crossing over the parents to form a new offspring. An example is shown in Figure 2.1.

   5.3 Random Mutation: mutating a new offspring at the locus (the position in the chromosome). In the binary encoding, the value of the locus might flip from 0 to 1, or 1 to 0. Figure 2.2 shows a example of the mutation operation.

   5.4 Reinsertion: placing new offsprings into the population.

6. Use the newly generated population and go back to step 3.

GA has been used in various applications in the area of computer networks to explore an

optimal way of utilizing restricted resources. In [18], Fernandez et. al used GA and Wang-Mendel to optimize parameters of fuzzy controllers, which is carried out to handle traffics at DiffServ network nodes for achieving guaranteed QoS. Their results showed that, with the help of GA and Wang-Mendel, the system can attain better controller parameters and performance.

GA has two most-common variations:

- Simple GA: In simple GA, the entire set of the population is replaced by its children. This means, the next generation completely takes the place of its parents.
- Stead State GA: This GA only replaces a few individuals in each generation. Normally the percentage of the replacement can be specified by the user.

To speed up the search period, BoLi et al. [15] apply a steady-state GA to solve the optimal traffic control problem. In this case, the number of generations and replaced individuals was specified.

2.3 Reduced Memory Algorithms

In [12], J A. Grice et al. introduced a DP variation with the *checkpoint* method to decrease the space requirement for sequence alignment problems. Unlike the traditional DP algorithm, Reduced Space Algorithm only stores parts of the interim information at the cost of an extra fixed-rate computation. Checkpoints are interim states such as decision and cost values that are actually saved in the memory during the DP filling procedure.

In the Reduced Space Algorithm, the available memory space M is divided into two parts: M_{fill} and M_{trace}. M_{fill} is the space for the checkpoints in the filling step; M_{trace} is the amount of the memory required during the computation in the traceback step. The Reduced Space Algorithm behaves slightly different from the classical DP algorithm:

- In the filling step, the algorithm allocates a tabular with a smaller size than the traditional matrix and does not store data of all stages.
    1. After the computation of the i_{th} stage, the algorithm saves the stage state in M_{fill} as a checkpoint.
    2. Then in the following stages, the algorithm does the normal computation whereas the interim data may not be saved. Let x denote the interval or *skip* between two checkpoints, interim data between the i_{th} stage and the $i + x + 1_{th}$ stage is discarded.
    3. Afterwards, another checkpoint for the $i + x + 1_{th}$ stage is saved in M_{fill}.
    4. This process is repeated until the algorithm reaches the ending stage $n - 1$.

- In the traceback step, the algorithm has to recalculate the omitted data before it constructs the optimal solution from stage $n - 1$ to stage 0.
    1. During the traceback step, whenever the algorithm encounters a checkpoint pertaining to the i_{th} stage , it restores the omitted interim data from the $i - x_{th}$ stage to the $i - 1_{th}$ stage by using the checkpoint for the $i - x - 1_{th}$ stage. This procedure works in a similar way to the traditional DP filling process with M_{trace} as the temporary memory space.
    2. Then, the algorithm constructs the optimal decision path between the i_{th} stage and the $i - x - 1_{th}$ stage.
    3. Afterwards, the algorithm does another recalculation for the $i - x - 1_{th}$ stage.
    4. This procedure is repeated until the stage 0 is reached. The feature of skipping some stages overcomes the limitation of the memory requirement of the classical DP.

The choice of x plays a major role on the algorithm requirements on space and running time. In a 2-Level algorithm, the optimal number of stages that can be calculated within M space

is $n = M_{trace}(M_{fill}+1)/2$. This optimal number is obtained when $M_{fill}=M_{trace}= M/2$ or $x \approx \sqrt{n}$. Hence, the computation time is approximately $3nm = O(nm)$, where n is the number of stages m is the number of states. Thus, with a constant 1.5 slowdown, the amount of required memory for n stages decreases to $O(m\sqrt{n})$. This algorithm can also be constructed in multiple levels. In [12], the number of stages that can be calculated in a L-level and M space is given by:

$$r_L(M) = \frac{(M+L-1)!}{(M-1)!L!} \tag{2.1}$$

This equation provides a view in the space-time tradeoff. When $L = \log n$, the required memory space can be reduced at $O(n/\log n)$ with an $O(\log n)$ slowdown.

2.4 Basic Parallel Processing Principles

2.4.1 Introduction

Some large computation projects, such as combinatorial problems, optimization issues and graphic processing require high-performance computers to speed up the computation tasks. Nowadays, commercial multiprocessor, consists of tens or even hundreds of processors.

Parallel computation can be implemented at diverse levels. Simply speaking, a parallel algorithm is an algorithm in which the computation can be carried out simultaneously on multiple processing elements (PEs). In this thesis, we only considered parallel algorithms written for multiple PEs.

The degree of parallelism, referred to as *granularity*, which is measured as the ratio between computation time and communication time, is strongly influenced by the architecture of the

parallel machine and the implementation of the algorithm. A *fine-grain* approach partitions the problem into many small tasks. A *coarse-grain* approach consists of a few larger tasks that require less inter-process communication. The choice of the fine-grain or coarse-grain granularity is determined by the algorithm and the properties of the parallel machine. Another primary attribute in the parallel algorithm is the *rank*. It is an index of the PE assigned by parallel software packages. Through the *rank*, we can distinguish among different PEs in the system. Consequently, it is possible to assign workloads to them.

2.4.2 Parallel Processing Algorithms

Load Balancing is a technique used to map a set of concurrent tasks, which are decomposed from the overall problem, into a set of PEs. Load Balancing is a key issue in the design of parallel processing algorithms. This section overviews several models used to describe parallel algorithms [6]. They are as follows:

- Pipeline model – the task decomposition: This model requires dependencies between sub-tasks and is now a part of the VLSI technology. Sub-tasks or sub-procedures are executed on multiple PEs in the scheme similar to a computation chain.
- Asynchronous model: Asynchronous algorithms can be executed simultaneously on multiple PEs. Each PE knows its local tasks and only needs negligible communication during the computation time. Fully asynchronous algorithms are rarely implemented.
- Partitioning model – the data decomposition: The model shares the computation among PEs. Each PE handles sub-tasks that are divided from the original problem, then figures out sub-solutions and combines them into the final result.

- Master/Slave model: One of the most classical ways to parallelize an algorithm is to give the control to a single PE, called the *Master*, and to let this PE synchronize the computation procedure with all other PEs called *Slaves*.

These models are not mutually exclusive and are often combined for designing parallel algorithms. There are several popular parallel softwares that support these models. Details of two of the most widely used parallel platforms, the Message Passing Interface (MPI) and the Parallel Virtual Machine (PVM) are given in *Appendix B*. Throughout this research, due to its availability in the experimental platform being used, we have used MPI.

2.4.3 Performance Metrics

There are several important performance metrics for the analysis of the parallel algorithm: the *computation time*, the *communication cost* and most importantly the *speedup* factor. The computation time, which greatly depends on the processing speed of the slowest PE, is the time required to complete the computation of the algorithm. Communication cost is one of the major overheads due to the need transferring data from one PE to another one during the execution of a parallel program. In developing efficient parallel processing systems, it is important to reduce as much as possible the interaction overhead.

The speedup is given by the ratio between the time required to find a solution with the sequential algorithm and the time taken to accomplish the computation with the parallel version. If we let $T(p)$ be the time required to complete the parallel computation on p PEs, T_s be the sequential time on a SISD machine, the speedup can be defined as the ratio:

$$s(p) = \frac{T_s}{T(p)} \tag{2.2}$$

However, in most practical experiments, T_s may be unknown. Hence, the scaled speedup factor

is simply given by:

$$s'(p) = \frac{T(1)}{T(p)} \tag{2.3}$$

In the following chapter, we will consider the scaled speedup as the parallel speedup factor. Once having available the speedup $s(p)$ and the number of PEs p, the parallel scaled efficiency is defined by:

$$E'(p) = \frac{s'(p)}{p} \tag{2.4}$$

The linear speedup $s(p) = p$ is the optimal value in an ideal system. However, due to the overhead introduced by the communication and synchronization tasks, the speedup $s(p)$ is less than p which implies that in practice the efficiency is a value between 0 and 1.

According to Amdahl's Law [1]: If f is the inherently sequential fraction of a computation solved by p PEs, then the speedup $s(p)$ is limited by the following bound:

$$s(p) \leq \frac{1}{f + \frac{1-f}{p}} \tag{2.5}$$

This expression shows that the fraction of the sequential computation decreases with the increase of the problem size. Based on the above, the parallel performance – the percent of parallelism on the observed speedup can be defined as:

$$pp = \frac{\frac{1}{s'(p)} - 1}{\frac{1}{p} - 1} \tag{2.6}$$

The aforementioned concepts have to be careful considered whenever designing a high-performance

parallel processing system. To achieve a shorter running time and a better speedup, the overheads in the system must be minimized.

Chapter 3

Traffic Control Mechanisms and Modelling

The chapter is organized into two main parts. The first part deals with the principles and main research efforts in the area of traffic control, with particular emphasis on the leaky and token bucket algorithms. The second part reviews the Token Bucket model and the optimization algorithm developed by N.U.Ahmed and BoLi et al. [15] [23] [32] which form the basis of our research on the development of an efficient parallel optimization algorithm for a multiple token bucket dynamic model.

3.1 The Leaky and Token Bucket Algorithms

Over the last two decades, we have witnessed the development of a large number of multimedia applications. The wide deployment of these applications will mainly depend on the quality of service provided by the underlying communications support. Towards this end, it is essential

to develop efficient yet simple traffic control mechanisms. The design of such mechanisms requires a clear understanding of the traffic patterns and the QoS requirements of a wide variety of applications.

In a broad sense, the traffic patterns can be divided into two main categories: i) Constant Bit Rate (CBR), which is widely used in telephone networks, characterized by a fixed bit rate as its name implies; ii) Variable Bit Rate (VBR) prevails on computer networks, such as local area networks (LAN) and the Internet. Most multimedia applications produce VBR traffic. The QoS requirements depend on many factors, such as, the minimum bandwidth required, end-to-end delays, maximum-tolerable packet-loss rates among others. To support VBR traffic streams, a network controller should be able to allocate and maintain the network resources. At times, a controllers may have to reject traffic flows. The process of deciding to accept or reject part of the flows is called *admission control.* In this section, we review one of the most classical traffic enforcement mechanisms namely the *Leaky Bucket Algorithm.*

The Leaky Bucket Algorithm together with its variant the Token Bucket Algorithm is one of the most well-known traffic control schemes used nowadays to regulate the transmission rate at the access nodes to the network.

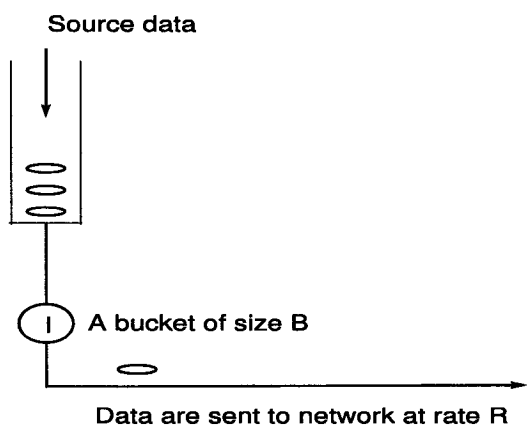


Figure 3.1: Leaky Bucket

The Leaky Bucket Algorithm depicted in Figure 3.1 was first introduced by Turner [13] in 1986. The algorithm is composed of two main elements: a (Token) Bucket and Tokens. The Leaky Bucket system can be viewed as a single-server queuing system with a constant service time. The algorithm enforces a constant output rate, R, regardless of the burstiness of the traffic generated by the data source. However, in most scenarios, speeding up the service rate in response to the arrival of a burst can avoid losing large quantities of traffic data and make use of network resources more efficiently. In order to overcome this problem, the *Token Bucket algorithm* depicted in Figure 3.2 has been introduced as an improved version of the Leaky Bucket Algorithm [29].

The operation of the Token Bucket Algorithm can be simply stated as follows. Tokens are generated and saved in the token bucket. In the case that the bucket becomes full, the Token Bucket Algorithm discards newly generated tokens. The saved tokens can then be used to grant access to bursts arriving at a later time. In this way, the traffic controller can accommodate the temporal burstiness characterizing many multimedia applications. Two variants of the Token bucket Algorithm have been defined (see Figure 3.2): 1) the *bufferless* Token Bucket Algorithm; and the *buffered* Token Bucket Algorithm. As their names imply, the difference between the two types of algorithms rely on the use of a buffer to temporally hold the incoming data traffic.

In summary, there are several specifications for the Token Bucket Algorithm:

- If there are enough tokens in the bucket, the packet is granted access to the network. Otherwise, the packet has to be dropped or buffered corresponding to one of the two different versions of the Token Bucket Algorithm;
- Tokens can be preserved in the bucket. However, when the bucket is full, the arriving tokens are discarded;
- The generation rate of the tokens can be different from one bucket to another and from one time to another;

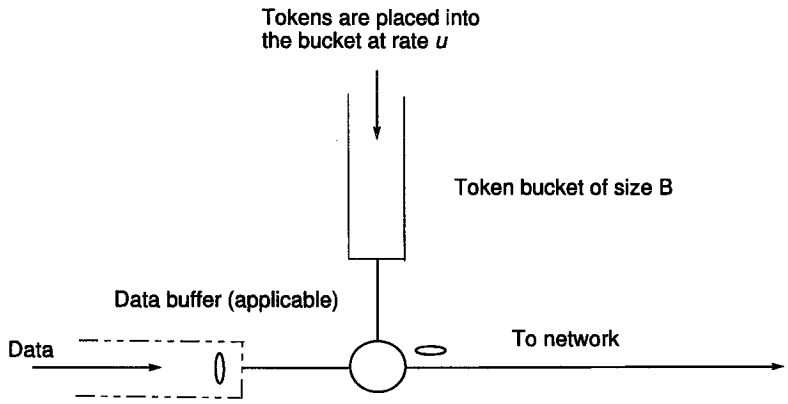


Figure 3.2: Token Bucket

- Packets can be saved in the buffer of a buffered Token Bucket. If the buffer becomes full, the incoming packets are discarded.

Over the last decade, with the development of new technologies for voice and video transmissions, the Leaky Bucket and the Token Bucket algorithms have been extensively studied.

In 1999, Tang and Tai [21] conducted a research study focusing on the definition of the optimal token bucket parameters taking into account various traffic patterns. Their study considered both versions of the Token Bucket algorithms, i.e., the bufferless and the buffered versions. They were particularly interested in the ability of the Token Bucket algorithm to allocate the network resources in real-time required by several different applications.

In 2000, Bruno et al. in [5] evaluated the parameters of the Token Bucket algorithm for Voice over IP (VoIP) applications. The procedure was based on a queuing system modelling a Differentiated Services (DiffServ) network. Their study allow them to estimate the Token Bucket parameters meeting the application requirements.

In 2001, Scogolio and Bruni [4] optimized the traffic generation of VBR MPEG streams in a client-server architecture. They optimized the system operation at the server side satisfying the Token Bucket controls. Optimal packet losses and service delays were obtained by specifying

the output rates of the video streams once having gone through the Token Bucket. The main objective of their research was to ensure the best possible quality of the video stream under the restrictions imposed by the Token Bucket controller.

In [17], Lombardo et al. conducted a research study whose main aim was the dimensioning of a smoother buffer and the system parameters defined by Intserv Working Group to meet the requirements of a stored-video communications system. The Token Bucket worked as a traffic shaper and could be used to figure out the discarding probability of non-confirming data.

Recently [9], Procissi et al. studied the impact of multimedia traffics exhibiting a long-range dependence (LRD) property on the Token Bucket parameters. They developed a LRD model to determine the optimal Token Bucket parameters by introducing the optimization criteria and cost functions into their model.

In 2002 [30], V. Guffens, G. Bastin and H. Mounier designed a fluid flow model for the token buffer. They designed a loop feedback control mechanism guaranteeing the boundedness of the buffer length.

Besides the aforementioned research efforts, various Token Bucket algorithms have been investigated by many other researchers. In [25] 1999, Toshihisa Ozawa introduced a packet-based Leaky Bucket algorithm with a parameter optimization method for ATM networks. In [16] 2000, Frank Yong Li considered the use of the Token Bucket mechanism to control the traffic in 3rd generation wireless networks. In [26] 2002, Park and Ko developed an algorithm to evaluate the traffic parameters for a video stream characterized by a specific QoS claim.

3.2 A Dynamic Model of the Token Bucket Algorithm

In the following, we review the Token Bucket model developed by N.U.Ahmed and BoLi et al. [15] [23] [32]. This model consists of N ($N \in [1 \cdots \infty]$) traffic sources, each one comprising a

bufferless Token Bucket, and one multiplexor to which all Token Buckets connect. The following symbols are defined and are used throughout the model specifications.

1. $(a \wedge b) = Min(a, b)$, $(a \vee b) = Max(a, b)$;
2. the Boolean function $I(S)$ is defined as:

$$I(S) = \begin{cases} 1 & \text{if } S \text{ is } true; \\ 0 & \text{if } S \text{ is } false. \end{cases}$$

3. $\lceil S \rceil = ceil(S)$, $\lfloor S \rfloor = floor(S)$.

Figure 3.3 depicts our network model consisting of multiple Token Buckets placed at the access node. To emulate a network system, this model contains one multiplexor and N traffic sources that are connected to the bufferless Token Buckets. Token Buckets with size B_i are clustered into a multiplexor through a link having bandwidth D_i. The multiplexor is assumed to be located at the edge of the network and to have a buffer of size Q. The outgoing link rate from the multiplexor to the network is C. Packets that cannot be served by the Token Buckets will be dropped promptly. If the multiplexor cannot transmit a packet immediately, it will be temporally queued.

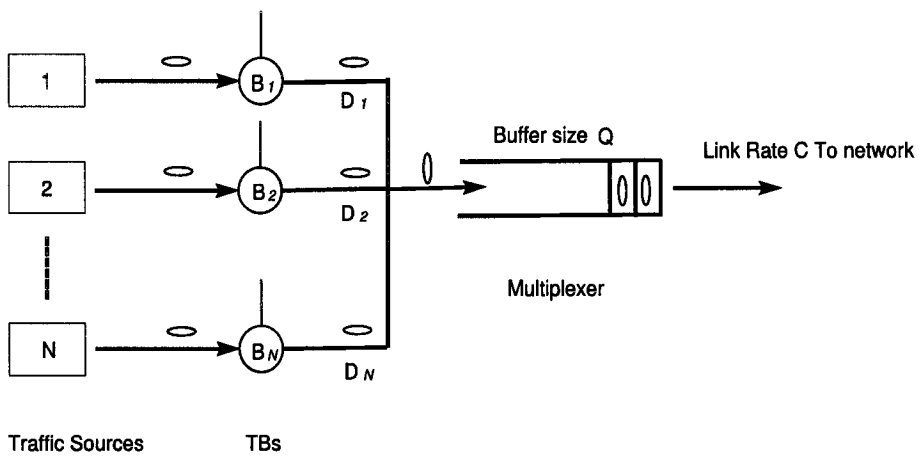


Figure 3.3: System Model

The traffic model is built upon the Variable Bit Rate (VBR) traffic scheme. Let $v(t_k), k = 0, 1, 2, ...,$ denote the incoming traffic at time t_k. The time interval $[t_{k-1}, t_k)$ is defined to be small enough to ensure that at most one packet arrives at the Token Bucket during this period. In [15] [32], N.U Ahmed, BoLi and Q. Wang et al have a deep study on this model. Our mathematics equations are based on the previous works.

The state of Token Buckets at time t_k by the vector equation:

$$\begin{aligned}\rho(t_k) &= \rho(t_{k-1}) + \{u(t_k) \wedge [B - \rho(t_{k-1})]\} \\ &\quad -v(t_k)I(v(t_k) \leq \{[\rho(t_{k-1}) + [u(t_k) \wedge [B - \rho(t_{k-1})]] \wedge D\tau\}) \end{aligned} \tag{3.1}$$

where $\rho(t_k) \equiv (\rho_1(t_k), \rho_2(t_k), \cdots, \rho_N(t_k))$. $\rho_i(t_k)$ is the token occupancy of the i_{th} Token Bucket at time t_k. The input values of this function are the control vector $u(t_k) \equiv (u_1(t_k), u_2(t_k), \cdots, u_N(t_k))$, known as the token generation rates at time t_k. $B \equiv (B_1, B_2, \cdots, B_N)$ denotes the bucket sizes of the Token Buckets. The size of incoming packets at time t_k is $v(t_k) \equiv (v_1(t_k), v_2(t_k), \cdots, v_N(t_k))$. $D \equiv (D_1, D_2, \cdots, D_N)$ is the vector denoting the link rates between the i_{th} Token Bucket and the multiplexor and τ is the time interval during which at most one packet may arrive.

The conforming traffic, denoted by $g(t_k)$ at time t_k, is given by:

$$g(t_k) = v(t_k)I(v(t_k) \leq \{[\rho(t_{k-1}) + [u(t_k) \wedge [B - \rho(t_{k-1})]] \wedge D\tau\}) \tag{3.2}$$

where $g(t_k) \equiv (g_1(t_k), g_2(t_k), \cdots, g_N(t_k))$ denotes the conforming traffic at time t_k. Thus, the non-conforming traffic is given by:

$$r(t_k) = v(t_k) - g(t_k). \tag{3.3}$$

Let $q(t_k)$ denote the state of the multiplexor at time t_k. This state is presented by the number of packets in the buffer waiting for service at the multiplexor. It is governed by the following equation:

$$q(t_k) = \{[q(t_{k-1}) - C\tau] \vee 0\} + \left\{\left[\sum_{i=1}^{N} g_i(t_k)\right] \wedge [Q - ([q(t_{k-1}) - C\tau] \vee 0)]\right\} \tag{3.4}$$

where C is the constant link rate between the multiplexor and the network subscriber. The first term on the right side of the equation represents the leftover packets in the queue at time t_k. The second term describes the accepted conforming traffic during the same period.

Because of limited resources such as the buffer size and the outgoing link rate at the multiplexor, the multiplexor cannot serve all the conforming traffic. The traffic lost is given by:

$$L(t_k) = \left[\sum_{i=1}^{N} g_i(t_k)\right] - \left\{\left[\sum_{i=1}^{N} g_i(t_k)\right] \wedge [Q - ([q(t_{k-1}) - C\tau] \vee 0)]\right\} \tag{3.5}$$

Under the proposed model, the possible packet losses are comprised of two parts: losses at the Token Buckets and losses at the multiplexor. In addition, there is the penalty caused by service delay at the multiplexor. Let $\alpha_i(t_k)$, $\beta(t_k)$ and $\gamma(t_k)$ $(i \in [1 \cdots N])$ denote the weights assigned according to various scenarios. The objective function is expressed as:

$$J(u(k)) = \sum_{i=1}^{N} \alpha_i(t_k) r_i(t_k) + \beta(t_k) l(t_k) + \gamma(t_k) q(t_k) \tag{3.6}$$

$$J(u) = \sum_{k=0}^{n-1} J(u(k)). \tag{3.7}$$

In 3.6, the first term represents the sum of weighted losses at the Token Buckets; the second term denotes the weighted traffic losses at the multiplexor and the last term describes the weighted losses due to service delay. The first two terms are "real" losses because packets are actually dropped, while the last term represents the delay due to the waiting time of the packets at the multiplexor. The Token Bucket generation rate $u(t_k)$ affects all values of $r(t_k)$, $L(t_k)$ and $q(t_k)$. The main objective is to find an optimal solution u to minimize J from time t_0 to t_{n-1}, where $u \equiv (u(t_0), u(t_2), \cdots, u(t_{n-1}))$.

BoLi et al. have developed a sequential control algorithm with 2 Token Buckets. The details of this algorithm is explained in [15]. This algorithm can guarantee the optimal control values. However, it requires remarkable running time and a huge amount of memory. Meanwhile, this algorithm only processes 2 Token Buckets. Therefore, we propose a more efficient parallel optimization algorithm in this thesis.

3.3 Computational Complexity

The Token Bucket control system is a high combinatorial problem. It requires a large amount of memory and huge processing requirements to accomplish the computation. The complexity of the algorithm just described was evaluated using a PC running at 400 MHz, the main characteristics of the configuration evaluated were:

- Number of Token Buckets: 2
- Sizes of Token Buckets: 20
- Size of the multiplexer: 40
- Average running time: 10.7 hours
- Average memory allocation: 211 MBytes

The space complexity of the sequential algorithm is $nmc + 2mc = O(nm)$; the time complexity is nmd, where n is the number of stages, m is the number of states, c is the required memory for one state and d is the time of calculating one state. From above data, we can see that in practice, the sequential algorithm cannot be deployed because it requires long time and a large memory space. Hence, it is necessary to reduce both time and space complexities by introducing a parallel algorithm.

Distributing memory allocations and computation workloads to multiple PEs is one essential characteristic of a parallel algorithm. As we know, the forward filling DP phase consists of two steps of computation. In the first step, because of the non-breakable sequential relationship between neighboring stages, all stages have to be computed in time order. Memories for costs and control values spread over multiple PEs. Without Reduced Memory Algorithm, each PE has to record data of the following amount of stages:

$$S_{PE} = \lfloor \frac{n+p-1}{p} \rfloor - I(rank \geq (n \bmod p)) \tag{3.8}$$

In this equation, *rank* is the index of the PE, n is the number of stages and p is the number of PEs. In case we are unable to distribute stages uniformly, the PEs with small ranks, that is $rank < (n \ mod \ p)$, will have more tasks than others. However, if we use the Reduced Memory Algorithm, the memory allocation behavior will be different. Actually, this algorithm only stores parts of stage data instead of all. Let x denote the skip in Reduced Memory Algorithm, the space complexity is approximately $\frac{nmc}{xp} = O(\frac{mn}{xp})$ on each PE. The details will be described in the next chapter.

The Parallel algorithm also balances workloads among multiple PEs. In the second step, the number of independent states, which can be viewed as GAs, is given by $\prod_{i=1}^{N}(B_i + 1)$. Minimizing the communication overhead is very important for an efficient parallel algorithm. Therefore, we adopt *partition model* to reduce the compulsive overhead that may be present in the normal case. All sub-GA workloads are uniformly distributed among the *master* and the other *slaves* with the following amount:

$$[(Q + 1)\prod_{i=1}^{N}(B_i + 1) + p - 1]/p \tag{3.9}$$

Based on it, ideally, the time complexity of the parallel algorithm is $\frac{nmd}{p} = O(\frac{mn}{p})$. However, because of the communication overheads, the computation time in practice is longer than this value. Moreover, the number of PEs also affects the execution time. Details will be discussed in the next chapter.

Chapter 4

A Multiple Token Bucket Optimization Algorithm

In this chapter, we present the details of an efficient parallel optimization algorithm for the Token Bucket model developed by N.U.Ahmed and BoLi et al. [15] [23] [32]. The algorithm has been defined based on the optimization methodology and heuristics described in the previous chapter. The parallel optimization algorithm determines the token generation rate vector u minimizing the cost J in Equation 3.7. The optimization methodology is supplemented by a Reduced Memory Algorithm and a Genetic Algorithm. Due to the stringent memory and time requirements of the algorithm, we propose the use of a parallel processing system for its solution. Besides including a complete description and detailed flowchart of the algorithm, the chapter provides an in-depth analysis of the time, memory and communication overhead requirements of the parallel algorithm.

4.1 The Reduced-Memory Dynamic Programming Algorithm

Combinatorial problems are characterized by their huge time and space requirements. The integration of a Reduced Memory Algorithm into the optimization algorithm can overcome one of the main drawbacks of Dynamic Programming, i.e., its huge memory requirements. In general, the memory requirements of Dynamic Programming are of the order of $O(nm)$, where n and m are the number of stages and the number of states, respectively.

As previously explained, in classical Dynamic Programming, the optimization algorithm requires to build a matrix data structure. The elements in the matrix are used to store interim data in a stage by stage basis and for every valid state. The main purpose of a Reduced Memory Algorithm is to relax this requirement by skipping some stages. The size of these gaps is called "skip". For instance, Figure 4.1 exhibits the memory allocation for a skip value of 1. The solid lines represent the elements actually present in the matrix.

Let c be the memory required per state; x, which takes a value between 0 and $n-1$, denote the skip in the algorithm; M be the available memory in the system; M_{fill} be the memory used to hold the checkpoints; and M_{trace} be the space used to carry out the traceback phase. Using these definitions, we can state the following relations:

$$M_{trace} = \begin{cases} mc(x-1) & x > 0; \\ 0 & x = 0. \end{cases} \tag{4.1}$$

$$M_{fill} = \frac{nmc}{x+1}, \tag{4.2}$$

$$M = M_{fill} + M_{trace}. \tag{4.3}$$

Based on the previous relations, it is possible to optimize the memory requirements. First,

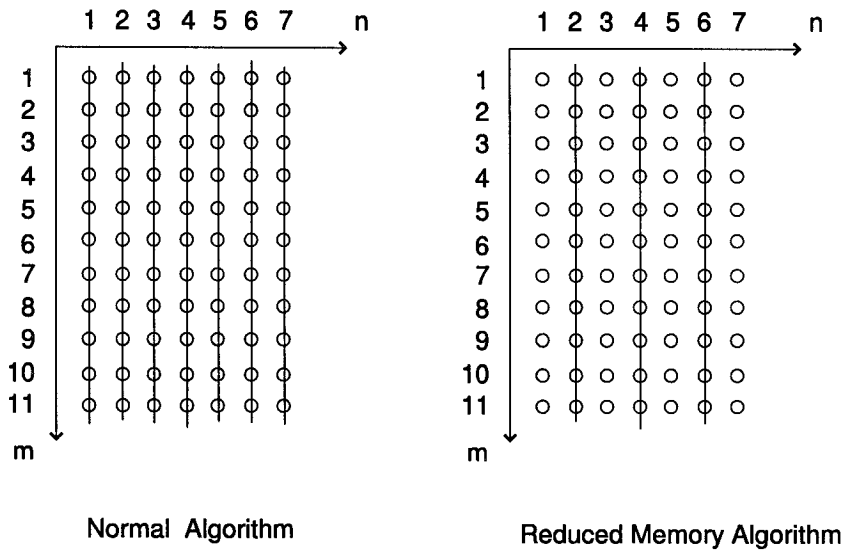


Figure 4.1: Reduced Memory Algorithm

the system memory M attains its optimal value when x is $\sqrt{n}-1$, i.e., when x is approximately equal to $\sqrt{n}$. The proof can be simply stated as follows:

Proof 1 $M = M_{fill} + M_{trace} = mc(x-1) + \frac{nmc}{x+1}$. Because m, c and n are constants, we can define the function $f(x) = mc(x-1) + \frac{nmc}{x+1}$ of the variable x. Hence, $f'(x) = mc - \frac{nmc}{(x+1)^2}$. To obtain the optimal value of $f(x)$, $f'(x)$ must be 0. Meanwhile, $f''(x) = \frac{2nmc}{(x+1)^3} > 0$. Therefore, when $x = \sqrt{n}-1$, $f(x)$ attains the optimal value.

Second, for a given memory size of M, the optimal skip size x is $\sqrt{n}$.

Proof 2 With constants m, c, and M, the optimal value of x can be reached when $M_{fill} = M_{trace} = M/2$. Thus, we can obtain the equation $\frac{nmc}{x+1} = mcx$. This equation can be transformed to $x^2 + x - n = 0$. It is simple to conclude that the solution should be $x = \frac{\sqrt{4n+1}-1}{2}$. Therefore, when $x \approx \sqrt{n}$, then x attains the optimal value.

From the above proofs, it follows that by fixing $x = \sqrt{n}$, the memory can be used most efficiently if the processing time is not an issue to the system. However, the decision of the

skip deeply influences the computation time. A wrong choice could result on extremely long computation times. The tradeoff can be simply explained by the following numerical example:

Recall that the total number of states per stage is given by:

$$m = (Q+1)\prod_{i=1}^{N}(B_i+1) \tag{4.4}$$

For example, for a system consisting of three token buckets, each with a capacity of 30 tokens, and a multiplexor with a capacity of 90 packets, there are a total of $(30+1)^3 \times (90+1) = 2710981$ states per stage. It is obvious that we should avoid having to recalculate this huge number of states during the traceback phase. After considering the space-time tradeoff, we have decided setting $x = 1$, as demonstrated in *Appendix C.*

In our implementation, checkpoints with even indexes are saved during the global filling phase. In the odd stages, the algorithm works following the normal procedure but it discards the interim data. In the even stages, the algorithm saves the control values and states. The algorithm proceeds this way until it reaches the final stage. Before constructing the optimal solution, the Reduced Memory Algorithm has to recalculate the information of those omitted stages. However, since the algorithm has memorized the interim data during the filling phase, the values of costs and control values can be obtained by recalculating only a single sub-GA when the index of the stage is even.

Because of the tradeoff between the time and the memory space, this algorithm cannot execute on a machine having a memory unit smaller than $O(nm)$. In the sequential algorithm developed by BoLi et al. [15], the memory requirement is nm. By using the Reduced Memory Algorithm, our optimization algorithm requires only half of that amount. This is the main benefit achieved by our algorithm.

4.2 Search space reduction by Genetic Algorithm

The number of states in one stage can easily reach millions, even hundreds of millions as the number of Token Buckets increases. Classical optimization methods such as the open-loop algorithm will require extremely long times searching for the optimal state. Therefore, it is crucial to optimize the search space operations. The use of Genetic Algorithms provides an efficient way to perform this high-demanding task.

In one stage, the total number of states can be expressed by Equation 4.4. By considering the same example from the previous section, the number of states in one stage is 2710981. In the case of one Token Bucket, the available control values are $[0 \cdots (B_i - \rho_i(t_k)]$, where ρ_i denotes the state of the i_{th} Token Bucket at time t_k. For the case of N Token Buckets, the search space of one state is given by:

$$\prod_{i=1}^{N}(B_i - \rho_i(t_k) + 1) \tag{4.5}$$

Thus, the average number of admissible control values for one state is 4096. Clearly, the search space in a stage is 4096 $\times$ 2710981.

Finding an optimal solution by the traditional search method like the open-loop algorithm in such a large space will take a remarkably long time. The open-loop algorithm has to traverse all possible combinations in a loop scheme, calculates the cost values, compare the stored minimal value with the current cost, and then save the new minimal cost and corresponding control values. In contrast, GA uses selected individuals to evaluate the costs so that it can avoid going through all possible values in the search space. When the problem has a small search space, the open-loop algorithm is more efficient since it is simple whereas GA is complicated. However, GA is much more efficient when there is a large search space. It shortens the execution time and can be finished quickly. Hence, for minimizing the execution time, there will be a tradeoff of the choice of two algorithms. N.U.Ahmed and BoLi et al. addressed this tradeoff value in the

study [15]. When the search space is smaller than this value, the open-loop algorithm is used. Otherwise, we use GA to obtain the optimal control values. We will discuss and verify the value in experiments in the next chapter.

A substantial reduction of the search time can be achieved by integrating a steady-state GA into the optimization algorithm. For the Token Bucket control problem, the "search space" is the entire set of admissible control values u_{ad}. The population is a small subset from the whole set. Each individual is a chromosome in the population that represents a combination of control values of the Token Buckets. If N Token Buckets are in the system, then each chromosome contains N genes, and each gene maps a control value of one of the Token Buckets. The optimal cost of the current state is the sum of two values: the first one is the current cost at stage k_{th}; the second is the cost from the stage $k + 1_{th}$ to $n - 1$ which satisfies the current state and this value can be retrieved from the matrix. By using a Genetic Algorithm, the search time will not increase exponentially as the problem size does. Experiments in the next chapter will show the benefits of using a Genetic-Algorithm space-search approach.

4.3 Parallel Processing

In this thesis, the algorithm has been designed to run on a MIMD parallel machine which can be either a super multiple-processor computer or a set of workstations interconnected by a high-speed network. This sort of loosely coupled structure is very flexible and nowadays widely deployed. The design issues of our parallel algorithm are explained in this section.

To use the resource efficiently, we start by assigning small divided tasks to the PEs using the parallel partition model. We assume that the PEs have similar computation capacities, therefore, the algorithm distributes the workload evenly among the PEs as specified by Equation 3.9. In practice, the largest workload difference at a given stage is the computation of one state.

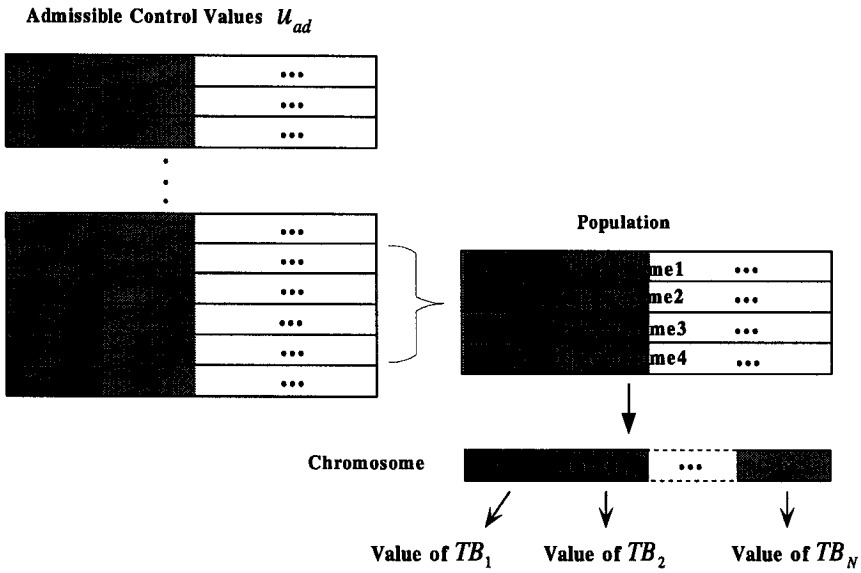


Figure 4.2: Genetic Algorithm

Whenever the number of PEs is larger than the number of states, those PEs with high ranks will remain free.

4.3.1 Parallel Reduced Memory Algorithm

The parallel Reduced Memory Algorithm behaves differently from the sequential version, especially on the memory distribution of the system. The parallel also results in different space and time complexities between the two algorithms. As we have discussed Equation 3.8 of the previous chapter, memories for recorded costs and control values are allocated approximately uniformly on multiple PEs with the following amount:

$$M_{PE} = \begin{cases} \lceil \frac{mnc}{2p} \rceil & rank < (n \bmod p) \quad (4.6a) \\ \lfloor \frac{mnc}{2p} \rfloor & rank \geq (n \bmod p) \quad (4.6b) \end{cases}$$

In the case when it is not possible to evenly distribute the stages, the small-indexed PEs, that is those whose $rank < (n \ mod \ p)$, will be assigned a heavier load. For instance, using the previous example, for a system comprising three PEs, the load (states) distribution is depicted in Figure 4.3. The Master PE takes care of stages 1 to 3, but it only saves the data of stage 2; PE 1 needs to record the data of stage 4 while PE 2 keeps the data for stage 6.

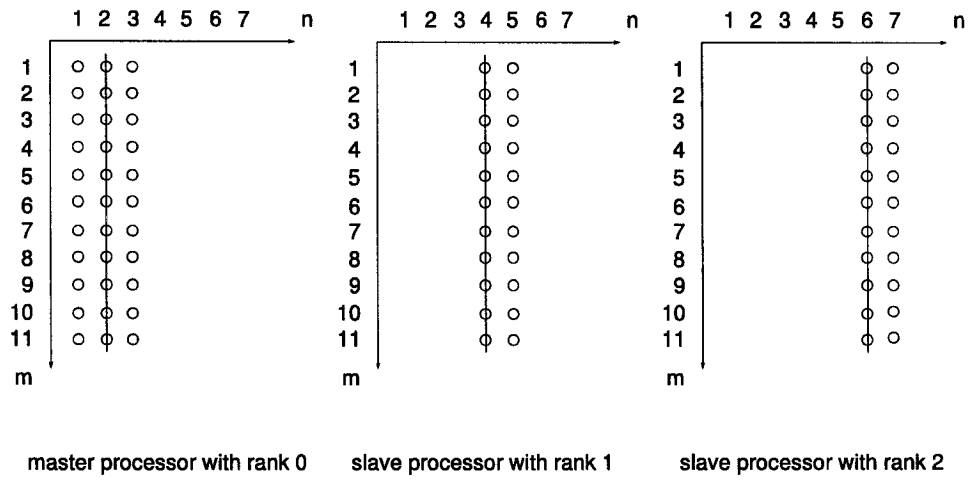


Figure 4.3: Parallel Memory Allocation

From the previous explanation, it is clear that the amount of memory required is not the same for all the PEs. The largest difference on the memory allocation assigned to each PE is equivalent to the space occupied by a checkpoint per stage. This can be easily demonstrated as follows:

Proof 3 The maximum possible difference on the memory distribution is 1 when $rank \geq (n \ mod \ p)$. Let $\zeta = \lfloor \frac{n+p-1}{p} \rfloor$. There are two possible cases: i) ζ is even. ii) ζ is odd. We consider the two cases separately:

i)ζ is even. The worst case is one PE has ζ stages and another PE has $\zeta - 1$. Because only stages with even indexes are treated as checkpoints, the odd number $\zeta - 1$ leads to two possible values of checkpoints: $\frac{\zeta}{2}$ and $\frac{\zeta}{2} - 1$. Meanwhile, the PE with ζ has $\frac{\zeta}{2}$ checkpoints. Therefore, the largest possible difference is 1.

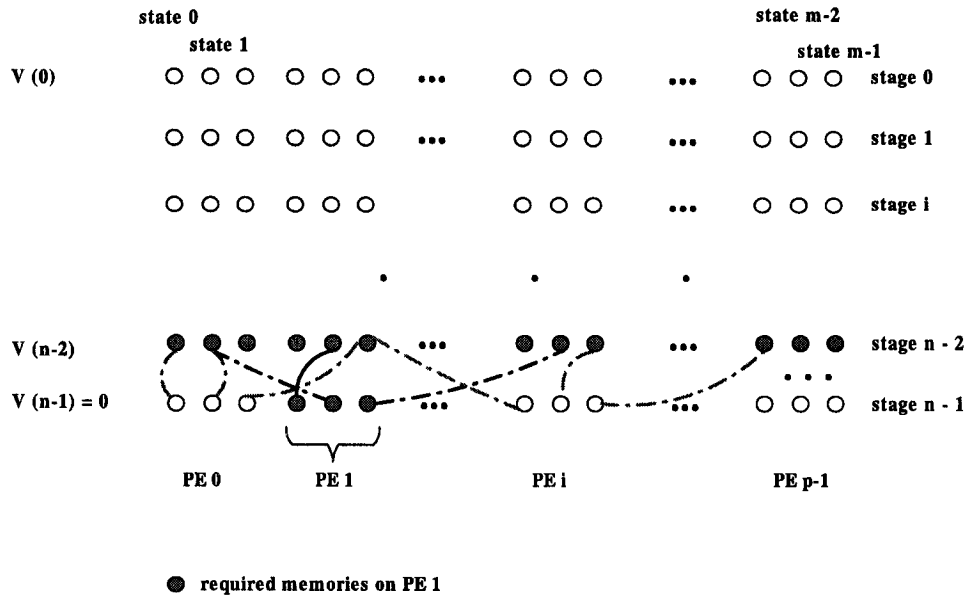


Figure 4.4: Parallel Reduced Memory Algorithm

ii)ζ is odd. The worst case is one PE has ζ stages while another PE has $\zeta - 1$. Because only stages with even indexes will be treated as checkpoints, the even number $\zeta - 1$ leads to the maximum possible value of checkpoints $\lfloor \frac{\zeta}{2} \rfloor$. Meanwhile, the PE with ζ has $\lfloor \frac{\zeta}{2} \rfloor$ or $\lfloor \frac{\zeta}{2} \rfloor + 1$ checkpoints. The above demonstrates that the largest possible difference is 1.

Through the above proof, we can ensure that the memory difference on multiple PEs will not exceed the space for 1 stage. The computation process is shown in Figure 4.4.

Another major issue to consider is the amount of overhead introduced by the inter PEs communication. One of the major source of the communication is due to the nature of the dynamic programming algorithm. Since the problem is solved backwards, it is not known which vector will eventually satisfy a given initial state. Therefore, the algorithm requires computing each admissible state. Furthermore, during time t_k, the values of $J(u(k))$ have to be made available to all PEs, in order to compute the optimal cost at time t_{k+1}. Hence, the best costs of

one stage must be broadcasted to and recorded by all PEs. Consequently, the system requires to store on each PE the values $J(u(k))$ corresponding to the states at a given stage. Besides the memory required to store the interim data, the PEs require enough memory space for computing the current stage. Thus, during the filling procedure, one PE must have at least $mc+\frac{mc}{p}$ available spaces to support such operations. To get the best control sequence, the algorithm traces to the last stage from the starting stage 0. During this phase, if the index of the last available stage on the PE is odd, the next PE has to do a extra GA with the coming state. Vice versa, when the index of the first stage on the PE is even, it must prepare to receive states from the previous PE and calculate an extra sub-GA. Summarizing the above analysis, the memory requirements of a PE is $mc+\frac{mc}{p}+\lceil\frac{mnc}{2p}\rceil = O(\frac{mn}{p}+m)$. The system computation time approximately reduces to $O(\frac{mn}{p}+m)$. We have a more precise time analysis in the next section. This algorithm can be developed further when it only records parts of control values and costs.

4.4 Algorithm Flowcharts

In this section, we present the flowcharts of the algorithm consisting of two main phases depicted in Figure 4.5: the tabular filling and the traceback. At first, the master PE reads the system configuration parameters and broadcasts this information to all slaves using the MPI primitive by MPI_Bcast. Afterwards, based on the information conveyed by the messages, the PEs figure out their local sub-GAs and prepare to perform their assigned tasks. Following this, the algorithm uses the MPI_Allgather primitive to collect the data and broadcast them in an all-to-all mode. Thereafter, only the appropriate PE records the interim data such as costs and control values. During the second phase depicted in Figure 4.6, the algorithm processes the stages sequentially, starting the computation from the master PE, which reads the initial state.

When there is no more necessary information on the local PE, the values of current state will be delivered to the next PE. The algorithm follows this scheme until it arrives at the final stage or the last PE.

4.5 System Analysis

In the parallel system, the number of PEs and the communication overheads influence the parallel speedup greatly because they have strong effects on the computation time. Therefore, it is important to analyze them in details.

4.5.1 Optimal Number of PEs

Normally, whenever more PEs are involved in a parallel system, a faster calculation speed is expected. However, as the number of PEs increase, there is an increase on the communication requirements which translates into higher communication overheads. An improper dimensioning of a parallel system may adversely affect the overall system performance. Therefore, there is a clear need to closely analyze and somehow optimize the number of PEs ensuring the best possible system performance.

Suppose there are m states in one stage; the link capacity among PEs is v; c is the required memory of one state in bytes; d is the time for calculating the solution of one state on a single PE, then the communication cost measured in seconds is $\frac{mc(p-1)}{v}$. The computation time of states in one stage on the PE is $\frac{md}{p}$ when the system reaches the optimal speedup p. Therefore, the execution time of one stage is equal to $\frac{mc(p-1)}{v}+\frac{md}{p}$. To minimize this formula with the variable p, we can transform it to:

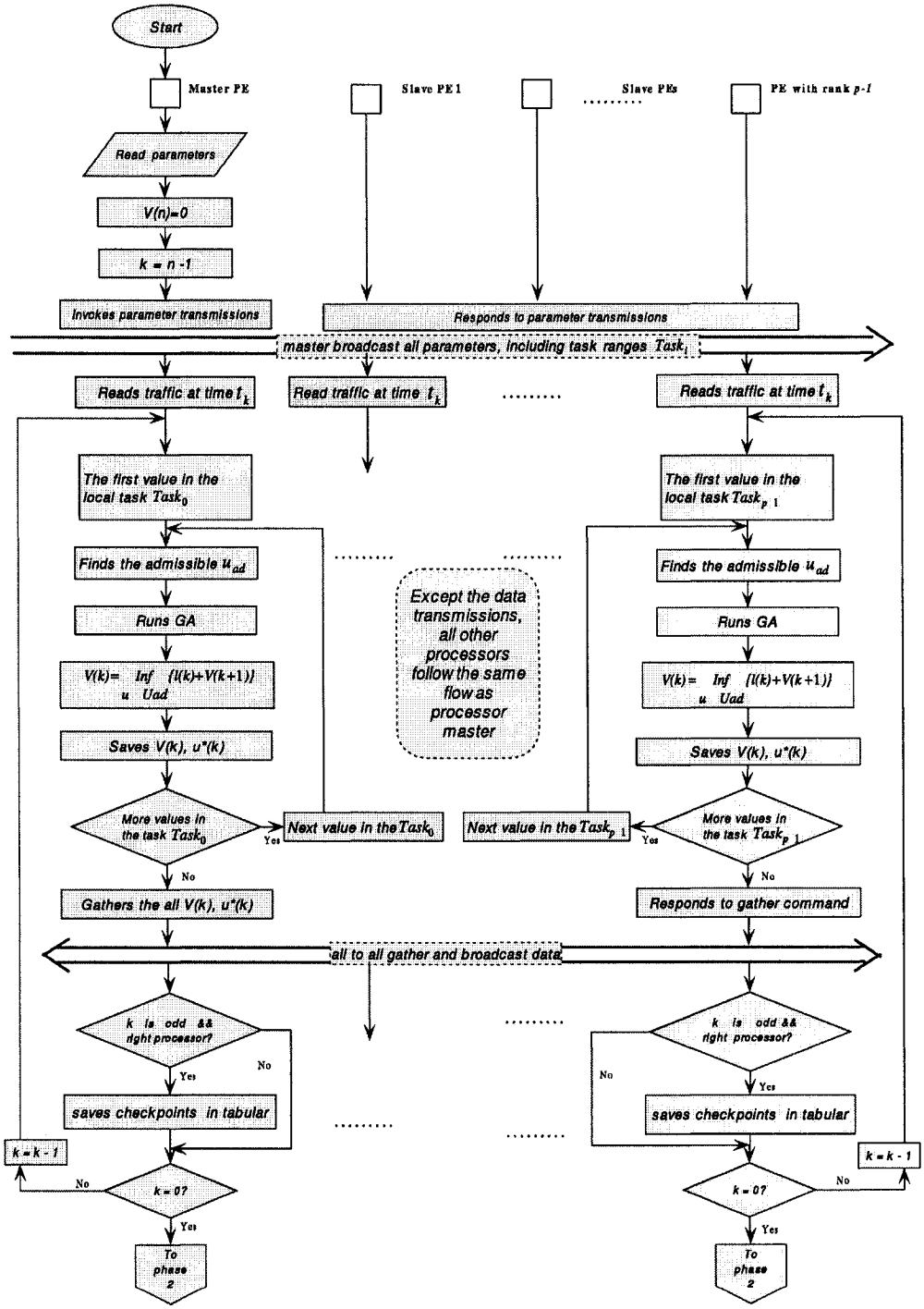


Figure 4.5: Phase 1 of Parallel TB Algorithm

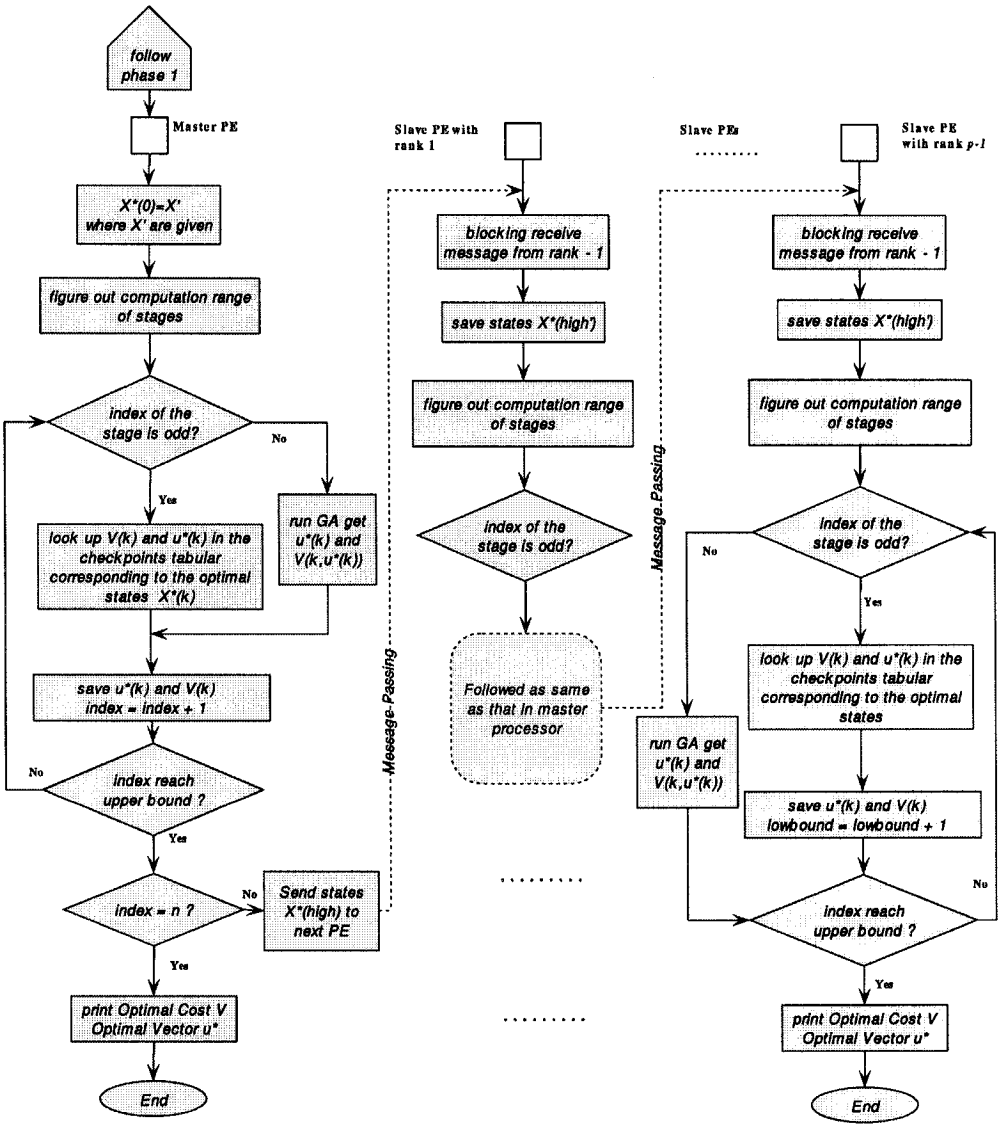


Figure 4.6: Phase 2 of Parallel TB Algorithm

$$f(p) = m\{\frac{cp}{v} + \frac{d}{p} - \frac{c}{v}\} \tag{4.7}$$

After some simple algebra, we can figure out that when

$$p = p_o = \sqrt{\frac{dv}{c}} \tag{4.8}$$

our algorithm runs in the optimal speed.

Proof 4 Because m, c, v and d are constants, the objective is to minimize the equation $\frac{cp}{v}+\frac{d}{p}$ with the variable p. Suppose $f(p) = \frac{cp}{v}+\frac{d}{p}$, then $f'(p) = \frac{c}{v}+d\frac{d}{dp}(p^{-1}) = \frac{c}{v}-\frac{d}{p^2}$. To obtain the minimal value of $f(p)$, it must be $f'(p) = \frac{c}{v}-\frac{d}{p^2} = 0$. Meanwhile, $f''(p) = \frac{2d}{p^3} > 0$. Therefore, when $p = \sqrt{\frac{dv}{c}}$, $f(p)$ has the optimal value.

We know that when p can minimize Equation 4.7, the system can obtain the shortest running time. However, $p = \sqrt{\frac{dv}{c}}$ is the optimal number of PEs in the ideal scenario where the optimal speedup can be reached. In normal cases, it is very hard to achieve the linear speedup p. Therefore, this p may not reveal the optimal number of PEs in practice. However, still it is a useful guide to determine the range of optimal number of PEs. Once we obtain the optimal number of PEs $p = \sqrt{\frac{dv}{c}}$, by Equation 4.7, the shortest execution time can be deduced with the following equation:

$$T = nm\{2\sqrt{\frac{cd}{v}} - \frac{c}{v}\} \tag{4.9}$$

4.5.2 Communication Cost

In the parallel algorithm, the main part of the communication overheads is the data gathering and broadcasting operations after the computation of one stage. Suppose each PE has φ bytes for broadcasting in one stage and there are p PEs in the system, one PE needs to gather

$(p-1)\varphi$ bytes from others. Therefore, the communication cost for one stage is $(p-1)\varphi p = (p^2-p)\varphi$ bytes. In the initial period of the algorithm, the master PE has to broadcast the parameters such as traffic data and system configurations to other PEs. This contributes to $O(n)+O(1) = O(n)$. During the traceback phase of the algorithm, states and other parameters such as the losses at TBs and the losses at multiplexor are exchanged among PEs with ranks ranging from 0 to $p-1$ and contribute to $O(p)$. Therefore, the overall cost of n stages is $(p^2-p)\varphi n + O(n) + O(p)$. In addition, φ actually can be obtained from $\varphi = O(m)/p$. From the above, we can figure out that the communication overhead of the TB control algorithm are $(p-1)O(m)n + O(n) = O(mn)(p-1) + O(n) + O(p)$. Because our algorithm is designed for coarse-grain parallel machines, the value of p is actually very small compared to n and m. Therefore, the cost is equal to $O(mn) + O(n) = O(mn)$. The primary reason for which the communication cost still keeps at $O(mn)$ is the strong sequential relationship between stages and state transitions that can not be forecasted.

Chapter 5

Numerical Results

The program was written in ANSI C++ and were complied using the SUN Forte 6 C/C++ with the fast optimization level. The standard MPICH-2.0 [20], a freely available, portable implementation of MPI, was used as the message-passing library. In this chapter, numerical results are produced from the experiments based on MPEG-4 transmissions and self-similar transmissions.

5.1 Hardware Platform

Experiments done in this thesis were carried out on the SunFire [28] cluster of the High Performance Computing Virtual Laboratory (HPCVL). The cluster is connected by a Gigabit Ethernet, consisting of eight SunFire 6800 servers, SFNODE0 to SFNODE7, a symmetric multiprocessor (SMP) system based on the UltraSPARC-III processor and the Solaris 9 Operating Environment. SFNODE0 to SFNODE3 are each configured with 24 $\times$ 1.05 GHz processors and 96 GB of memory. SFNODE4 to SFNODE7 are configured with 24 $\times$ 900 Mhz. Details of the

Sun Microsystems environment can be viewed in [11]. Parallel jobs are submitted via the Grid Engine, which allocates resources such as the memory and the processor time to different jobs. With this scheme, parallel jobs wait in the queue of the Grid Engine for service. A job requiring more processors implies longer time in the queue and less opportunities to execute. Therefore, it is impossible to utilize all processor and memory resources simultaneously. Meanwhile, the Sun Grid Engine does not allow the parallel jobs to claim more than 1 GB of memory space. Based on the above limits, the conducted experiments on the SunFire cluster were limited to a maximum of 24 PEs.

5.2 Traffic Traces Specifications

Throughout this study, we have used two sets of traffic traces to the parallel TB algorithm. One is a group of MPEG-4 encoded video traces. The other is a set of self-similar traces.

5.2.1 MPEG-4 Traces

Nowadays, more and more multimedia services, such as video and voice, are supported through high speed network systems. Video communications applications require abundant resources to ensure QoS. In this section, we briefly describe the characteristics of four MPEG-4 traffic traces: a video encoding scheme widely used nowadays.

A standard MPEG (MPEG-1, MPEG-2 and MPEG-4) encoder generates three types of compressed frames: *I, B and P* [7]. The *I* frames are compressed by using an intra-coded scheme; the *P* frames are similar to the *I* frames but with additional information regarding the previous *I*- or *P*- frames; *B* frames are encoded similarly to the *P* frames but using a bi-directional prediction mechanism. Typically, the *I* frames exhibit the largest size among the

three types of frames; and the B frames, the smallest, since they are encoded using the maximal compression rates. The frames are arranged in a deterministic order called a "group of pictures" (GOP, e.g., 'IBBPBBPBBPBB') pattern. Unlike the frame-based encoding of MPEG-1 and MPEG-2, MPEG-4 makes use of an object-based scheme.

Throughout our study, the video streams being used have been encoded by setting the number of objects to 1. The video streams have been captured on a clock basis like the frame-based standard. The traffic traces have been generated from four video sequences with a typical length of 60 minutes each. The details of the video encoding procedures can be found in the Technical Report TKN 00-06 [7]. Table 5.1 summarizes the relevant statistics of the video streams used throughout our study, namely, *Mr. Bean*, a movie, 1997; *The Simpsons*, a cartoon show, 2001; *Formula 1*, one sport event, 2000; *ARD News 2000*, a news sequence.

Figures 5.1 shows that MPEG-4 traces have their own special patterns. Frames belonging to the cartoon video, *The Simpsons*, are larger than those of any other traces. The appearance of its I frames is not as distinct as I frames in other traces. The news video, trace 4, exhibits peak periods and low periods on its I frames: the frames between 67 and 169 are clearly larger than those between 1 and 67. All traffic traces follow the GoP pattern: one big I frame appears every 12 frames. The trace statistics are given in Table 5.1.

Traffic Trace	**Mean bit rate**	**Peak bit rate**	**Peak/Mean rate**	**Mean Frame Size**
Mr. Bean	0.58Mbps	3.1Mbps	5.24	2909.92 Bytes
Simpsons	1.3Mbps	8.8Mbps	6.75	6513.69 Bytes
Formula 1	0.84Mbps	2.9Mbps	3.45	4177.77 Bytes
ARD News	0.72Mbps	3.4Mbps	4.72	3612.81 Bytes

Table 5.1: MPEG-4 specification of traffic traces

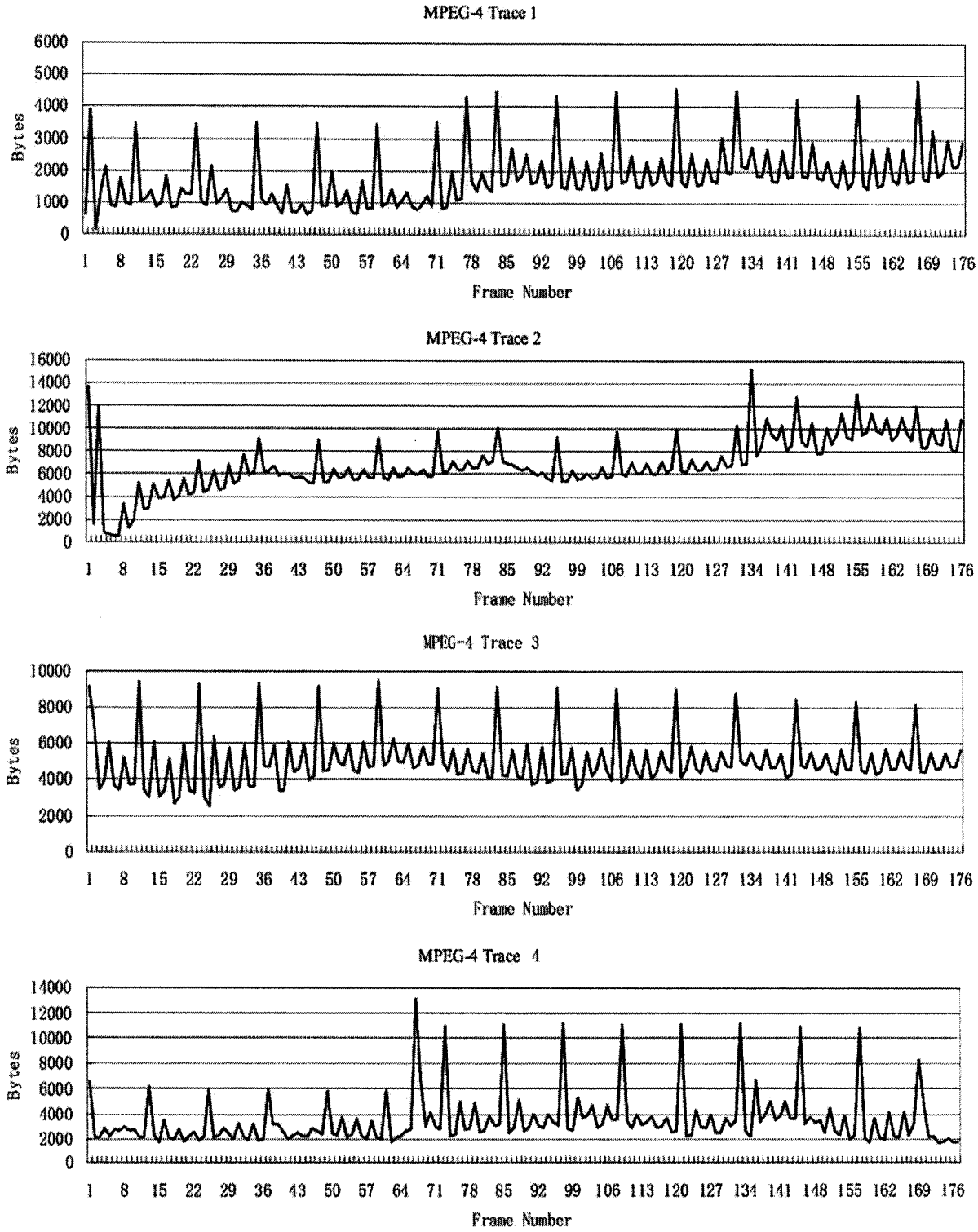


Figure 5.1: MPEG-4 Traffic Traces

5.2.2 Self-Similar Traces

The traffic traces generated by Web, FTP, video conference applications among others, cannot be specified by traditional models, such as the Poisson Process. This sort of traffic has highly similar-looking bursts occurring within a time scale spanning from milliseconds to minutes or hours.

The three self-similar traces [2] used in our study were obtained from the archive *Traces in The Internet Traffic* available from Bellcore. These traces have been widely used as representative traffic traces exhibiting a self-similar statistical characteristics [14]. W. E. Leland et al. [14] have formally demonstrated that Ethernet traffic is statistically self-similar. Each trace contains over a million records of packet arrivals. Figure 5.2 depicts snapshots of the three self-similar traffic traces used in our study. The first two traces shown in the figure correspond to LAN traffics (with a small portion of WAN traffic). The third trace is a WAN traffic trace. All the packets in the traffic traces have a minimum size of 64 bytes and a maximum size of 1518 bytes, values that comply to the Ethernet MAC protocol.

The trace statistics are given in Table 5.2.

Traffic Trace	**Mean bit rate**	**Peak bit rate**	**Peak/Mean rate**
trace 1	1.11Mbps	4.56Mbps	4.11
trace 2	2.9Mbps	7.63Mbps	2.63
trace 3	0.03Mbps	1.61Mbps	53.68

Table 5.2: **Self-Similar specification of traffic traces**

From the trace figures and statistics, we see that Trace 2 has the heaviest workload; and the average bit rate of the third trace is much smaller than the one of the other two traces.

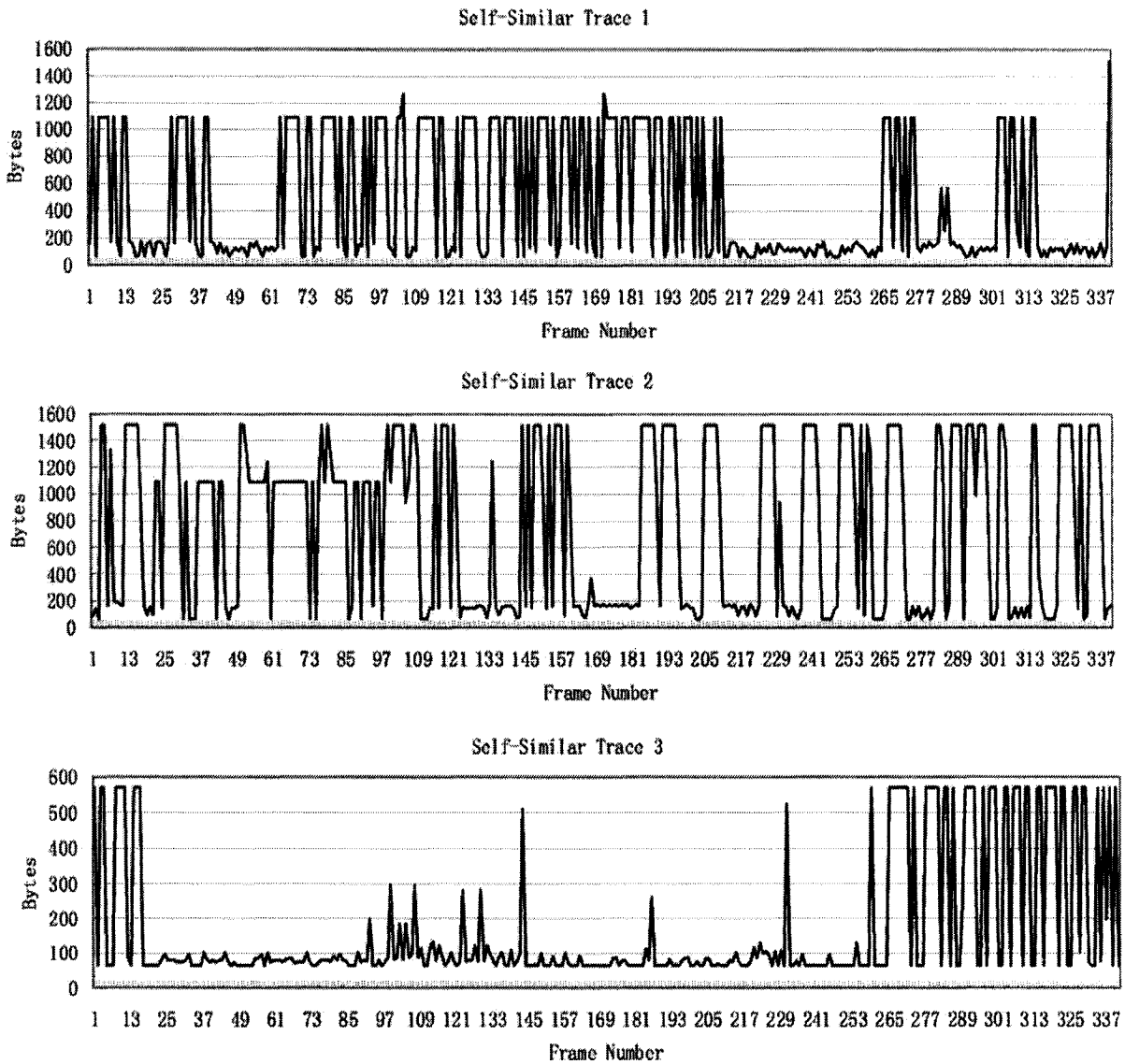


Figure 5.2: Self-Similar Traffic Traces

5.3 Numerical Results and Analysis

To evaluate our parallel algorithm, relevant weights have been assigned to the different types of losses as follows:

- $\alpha(t_i) = 10$, weight assigned to Token Bucket losses;
- $\beta(t_i) = 5$, weight assigned to multiplexor losses;
- $\gamma(t_i) = 1$, weight assigned to waiting losses at the multiplexor.

By assigning a heavier weight to one type of loss, we are setting up our preferences, i.e., a heavier weight translates into a lower loss probability for that component with respect to other components. In our particular case, we have preferred to drop a packet at the Token Buckets rather than having to discard it at the multiplexor, due to lack of space. Furthermore, keeping packets waiting at the queue of the multiplexor is seen as a better option than unnecessary rejecting packets at the Token Bucket. Once a packet has been admitted into the network, i.e., once having already passed the Token Bucket, the system should make its best for its successful transmission. Otherwise, dropping the packet at the multiplexor means that the system has unnecessarily wasted network resources by first admitting it into the network and then rejecting it at the multiplexor. In other words, if the system has to reject a packet, it is preferable to do it at the Token Bucket than at the multiplexor.

5.3.1 Numerical Results for MPEG-4 Traffic

In the first part of our study, four MPEG-4 traffic traces have been used, each one being controlled by a Token Bucket. In order to emulate the characteristics of a typical network system, we have had to define some key parameters. In the network set up, we assume that all

the packets conform to the UPD/IP protocol using an Ethernet LAN. Although UDP allows the use of a maximum packet size of 64kbytes, Ethernet imposes a limit of 1518 bytes per packet. Therefore, as a general rule, the packet size is assumed to have a size between 64 bytes and 1500 bytes. In addition, we assume that a video frame will be transmitted using the longest admissible packet size, i.e., a video frame can be divided into up to 9 packets which translates into a maximum frame size of 13500 *bytes*. This in turn defines a maximum admissible rate for a video stream of 2.7Mbps.

- $B_i = 2000\ Bytes, i = 1, 2, \cdots, N$, the capacity of the bucket i, this value must be larger than the maximum packet size;
- $Q = \sum B_i\ Bytes$, the capacity of the buffer in the multiplexor, where N is the number of Token Buckets, $N \in [1, 2, 3, 4]$;
- $T = 100\ Bytes$, the size of one token;
- $C = 3.44M\ bps$, the output link capacity from the multiplexor to network subscribers, this capacity is large enough to support most traces peak rates
- $D_i = 3.6M\ bps$, the link capacity between the i_{th} bucket and the multiplexor, which is large enough to support most traces peak rates;
- $\tau = 4.44msec$, the average interval corresponding to the arrival of a single packet.

We also define the following GA parameters:

- $p = 6$, the population size;
- $g = 4$, the number of generations.

Running Time and Speedup

In this section, we study the speedup of the parallel optimization algorithm through four cases. Each case is characterized by comprising a different numbers of Token Buckets, ranging from 1 to 4. Because of the exponential increase on the running time and memory requirements, the numbers of stages in the four cases are different from each other: for the 1 Token Bucket configuration the number of stages has been set to 720 stages; for the 2 Token Buckets case, to 162 stages; for the 3 Token Buckets case, to 18 stages, and for the 4 Token Buckets case to 9 stages. The buffer size of the multiplexor has been always set to be able to contain all possible packets coming from the Token Buckets, i.e., equal to $\sum B_i$.

Case 1:

- Number of Token Buckets: 1
- Number of stages: 720
- Number of states per stage: 441
- Average number of admissible states per stage: 11

Case 2:

- Number of Token Buckets: 2
- Number of stages: 162
- Number of states per stage: 18081
- Average number of admissible states per stage: 221

Case 3:

- Number of Token Buckets: 3
- Number of stages: 18
- Number of states per stage: 564921

- Average number of admissible states per stage: 4631

Case 4:

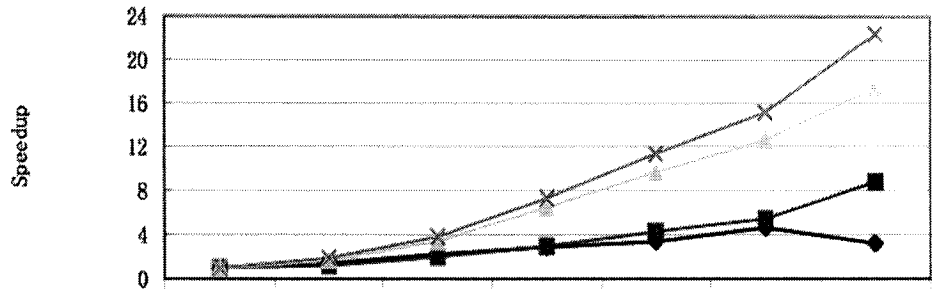
- Number of Token Buckets: 4
- Number of stages: 9
- Number of states per stage: 15752961
- Average number of admissible states per stage: 97241

	Number of PEs						
	1 PE	**2 PEs**	**4 PEs**	**8 PEs**	**12 PEs**	**16 PEs**	**24 PEs**
Case 1	8.88	6.29	3.93	3.07	2.61	1.91	2.75
Case 2	477.37	368.22	236.31	160.7	111.16	87.59	54.29
Case 3	5538.82	3230.6	1616.05	852.8	572.59	439.66	318.11
Case 4	93742.4	48960.3	24452.2	12773.6	8217.36	6159.2	4170.9

Table 5.3: Running Time of MPEG4 Traces (in seconds)

The numerical results shown in Table 5.3 depict the execution time for the four cases (in seconds). As we have anticipated, the computation time for the four cases increases exponentially corresponding to the number of Token Buckets. The average calculation times for one stage using 1 PE are: 0.012, 2.95, 307.71, and 10385.8 seconds, respectively, i.e., increasing ratios of: 246, 104, 34. This shows that the increase ratios are not linear to the sizes of the Token Buckets. In fact, the values gradually decrease to the new TB size of 21. Two reasons contribute to this. First, more Token Buckets means a larger search space for a state. Thus, the state search operation delays the execution time. Second, in our implementation, when the state search space is smaller than a certain point, we replace the sub-GA routine by an open-loop algorithm, which shortens the time to find the optimal value. Obviously, 1 and 2 TBs have more chances to run using an open-loop scheme than the other two cases. Hence, the numerical results illustrates the descending ratios.

From the data in Table 5.3, we can obtain speedups of multiple TBs in Figure 5.3. Figure 5.3 shows that the speedup of the 1 Token Bucket case first increases then decreases slightly. This implies that for this case, we have reached the maximum possible value for the speedup. However, for Cases 2, 3 and 4, the speedups continuously increase as we increase the number of processors. These results show that by using up to 24 PEs, we have not reached the maximal speedup values for these cases. These results should allow us to define the "the optimal number of PEs", which will be discussed in a later section.



	1	2	4	8	12	16	24
1 TB	1	1.41	2.26	2.89	3.4	4.65	3.23
2 TBs	1	1.21	2.02	2.97	4.29	5.45	8.79
3 TBs	1	1.71	3.43	6.49	9.67	12.6	17.41
4 TBs	1	1.91	3.83	7.32	11.37	15.18	22.41

Number of Processors

Figure 5.3: Speedup with MPEG-4 traces

We can also observe from Figure 5.3 that the speedup factor deviates from the ideal case. Especially for the cases 1 and 2 at light workloads, the algorithm shows low performances. For instance, the maximal speedup is only 4.65 with 16 PEs in case 1. The speedup of 2 Token Buckets is slightly better. However, cases 3 and 4 exhibit better performance than the previous two cases, nearly reaching the ideal speedup values. In case 4, the speedup is 3.83 with 4 PEs and 7.32 with 8 PEs. The parallel algorithm for the 1 or 2 Token Bucket cases cannot be executed efficiently since the granularity, the ratio between computation time and communication time, is too low, which is not sufficient for the parallel computation. This obeys Amdahl's Law: the

fraction of sequential computation decreases with the problem size.

	Parallel Efficiency	**Percent Parallel**
Case 1	0.39	73.27%
Case 2	0.42	73.47%
Case 3	0.81	95%
Case 4	0.94	99%

Table 5.4: **Average Parallel Efficiency and Percent Parallel**

We have not been able to evaluate systems with more than 4 Token Buckets due to the large memory requirements. As we have indicated in the previous chapter, each PE must record the $J(u(k))s$ corresponding to the states of 1 stage. Therefore, to support multi-TB operations, the memory capacity of one PE must be more than mc. This is the part of the memory which cannot be distributed. There are two reasons to keep us from distributing this part of the memory space. First, GA demands local interim data. The sub-GA uses best costs of previous stages to optimize current state vectors. If the data is not stored locally, the PE must retrieve it from other PEs. Meanwhile, each stage is composed of hundreds, thousands, and even millions of sub-GAs. Therefore, to lower the computation speed of the sub-GA will be disastrous to the time performance. Second, the idea of distributing the $J(u(k))s$ among multiple PEs implies more communication overheads. Such operations with numerous states can cause extremely long transmission delays, which reversely affect the computation speed greatly. Accordingly, the centralized interim data model seems a more suitable approach for our parallel algorithm than a distributed one. This centralized scheme imposes a limit on the number of Token Buckets that the parallel machine can support since the memory requirement augments exponentially as the number of Token Buckets increases. For instance, the case with 4 Token Buckets demands approximately 567 MB of memory space. A 5 Token Buckets system requires a minimal memory demand per PE of 4.95GB. This demand increases to 41GB when 6 TBs are involved. For this reason, we have not been able to evaluate the system with more than 4 TBs. However, in theory

our algorithm can work with any number of Token Buckets as long as hardware environments can provide enough support.

Parallel vs. Sequential Algorithms Comparison

In this section, we focus on the time and memory issues of the parallel algorithm by comparing it to the sequential algorithm. Experiments have been conducted by varying the number of stages, ranging from 162 to 1440. Since the sequential algorithm only considers a network system involving 2 Token Buckets, we have limited this study to this number.

Running Time Analysis

In this evaluation, we have considered the use of 1, 4, 8 and 16 PEs:

- Number of TBs: 2
- MPEG-4 sequences: "Mr Beans", "Simpson"
- Number of PEs involved: 1, 4, 8, 16

  Case 1: $n = 162$ stages

  Case 2: $n = 360$ stages

  Case 3: $n = 720$ stages

  Case 4: $n = 1440$ stages

The main focus of this section is to compare the running time between the two algorithms. Through the numerical results shown in Figure 5.5, we have obtained the average computation time per stage when using multiple PEs. These results are depicted in Figure 5.4. The figure shows that the execution time of the parallel algorithm with 1 PE is approximately two times longer than that of the sequential algorithm. Several factors contribute to this. i) The sequential

algorithm only considers the case with two Token Buckets and does not focus on the case with multiple Token Buckets. However, the parallel version takes into account the multiple Token Bucket case in view of *generalization*. Hence, it penalizes the code efficiency when algorithms only confront the system consisting of 2 Token Buckets. ii) To efficiently use the memory, the Reduced Memory Algorithm takes extra time. iii) The MPI routines spend extra time, not only with MPI Calls, but also with the latency taken to initialize the parallel environment and to synchronize the computation procedure. The parallel algorithm demands 4 or more PEs to reach the sequential time. When the PE number is 16, the execution time is approximately one third of that with the sequential algorithm. Based on the speedup analysis and the parallel theorem, we can estimate that by increasing the number of Token Buckets, the parallel algorithm exhibits better efficiency than the sequential one.

	Parallel Algorithm				**BoLi's Algorithm**
	Number of PEs				
	1 PE	**4 PEs**	**8 PEs**	**16 PEs**	**1 PE**
Case 1	477.373	236.31	160.7	87.59	222.14
Case 2	1062.16	594.517	363.688	177.11	500.46
Case 3	2128.73	1070.98	603.27	373.288	1001.42
Case 4	4236.66	2120.13	1382.22	697.949	1722

Table 5.5: Time Comparison of Two Algorithms (in seconds)

Memory Requirements Analysis

The study on the memory requirements comparison between the two algorithms has been conducted using a two Token Buckets system and by setting the number of stages to 1440. Figure 5.5 reveals the benefits of the Reduced Memory Algorithm and the use of parallel processing

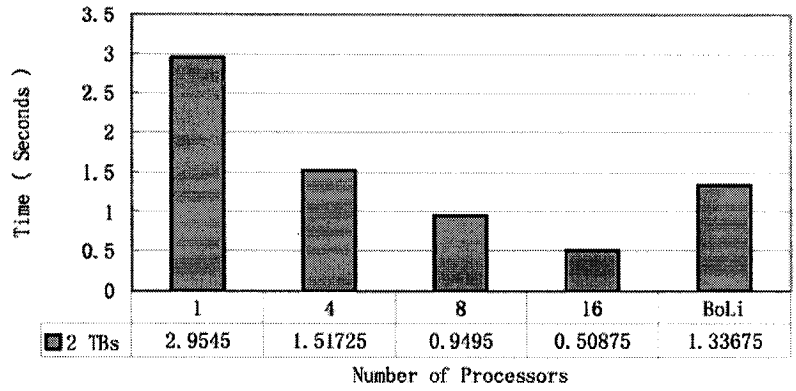


Figure 5.4: Average Time of One Stage

system. The Reduced Memory Algorithm decreases the memory space at the expense of extra processing. For instance, when 1 PE is used, the parallel version requires only 105MB, half of the amount of the memory that the sequential algorithm demands. One benefit of the parallel algorithm is that it can distribute the memory among multiple PEs. Thus, the amount of the allocated memory on one PE decreases as the number of PEs increases. In the figure, each PE holds 14MB when 8 PEs are in the system; one PE needs only 8MB if there are 16 PEs. The other benefit of the parallel implementation is the small memory increase with the number of stages n. That is, the additional memory of one PE is only $\lceil \frac{mc}{2p} \rceil$ when n increases by 1. This is particularly useful because normally the stage number n holds a large value. Based on the above advantages, our parallel algorithm can solve bigger problems that cannot be processed on a single PE in the sequential model.

Token Size Effects on Packet Losses

In this section, we study the effects of the token size on the packet losses in the network. In general, a smaller size of the token implies a larger search space per sub-GA; the token with a larger size decreases the computation workloads on both the quantity of sub-GAs and the search

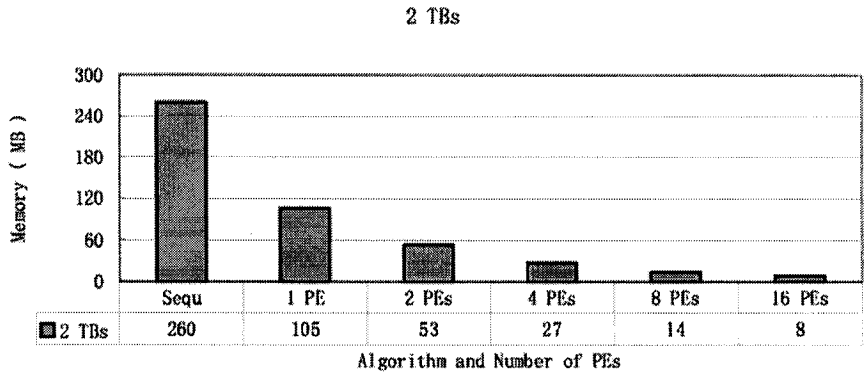


Figure 5.5: Memory Comparison on MPEG-4

space within a sub-GA.

Case 1:

- $T = 50Bytes$, $\frac{1}{2}$ of the original size;

Case 2:

- $T = 100Bytes$, the original size;

Case 3:

- $T = 200Bytes$, 2 times of the original size.

Figure 5.6 shows that the size of the token has a slight impact on the packet losses. The token with 100 bytes provides the lowest Token Bucket and Total losses. However, the waiting losses are larger than those obtained for the 50 bytes token size. Case 1 represents a more elaborate control mechanism. However, this does not mean that it is able to guarantee that the algorithm is able to minimize the loss count. The figure also shows that Case 1 decreases the waiting losses at the cost of increasing the losses at the Token Buckets and the multiplexor. Case 3, which uses a 200-bytes token size, loses more packets than the previous two cases by allowing more traffic to get into the system.

It is important to note that the use of a token size of 200 bytes drastically shortens the

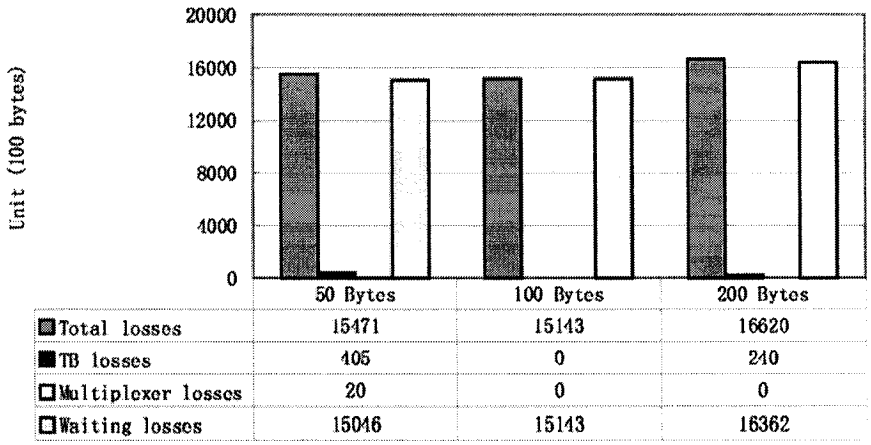


Figure 5.6: Token Size vs. Packet Losses

running time; in our case study to only 73.86 seconds from 30169.2 seconds for Case 1 and 2118.69 seconds for Case 2. These results show that there is an important tradeoff between reducing the execution time by increasing the token size and the level of granularity of the control mechanism. The final choice will very much depend on an in-depth analysis of the requirements of each type of applications.

Weights Effects on Packet Losses

In this section, we investigate how weights may affect the system performance, namely the different types of losses. Overall, we have considered four cases by varying the weight assignments:

Case 1:

- $\alpha(t_k) = 5$, $\beta(t_k) = 5$, $\gamma(t_k) = 5$, no special preference;

Case 2:

- $\alpha(t_k) = 10$, $\beta(t_k) = 5$, $\gamma(t_k) = 1$, preference on waiting losses;

Case 3:

- $\alpha(t_k) = 1$, $\beta(t_k) = 5$, $\gamma(t_k) = 10$, preference on Token Bucket losses;

Case 4:

- $\alpha(t_k) = 10$, $\beta(t_k) = 1$, $\gamma(t_k) = 5$, preference on multiplexor losses;

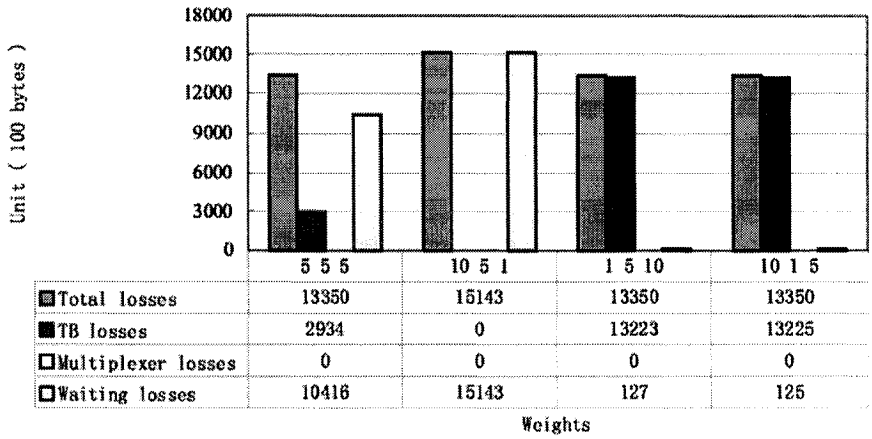


Figure 5.7: Weights' effects on Packet Losses

Numerical results in Figure 5.7 show the effects of the weights on the distribution of the packet losses. The results illustrate that the overall loss rate is very similar for the four different weight assignments being analyzed. In all cases, there were no multiplexor losses, i.e., losses due to buffer overflow. Even for Case 4, for which we have assigned the lightest weight to the multiplexor losses, no packets were discarded due to the lack of buffer space at the multiplexor. In Case 1, where no particular preference has been given to any of the components, the system allows a large number of packets to get into the system. This results on a high loss rate due to the excessive waiting times at the multiplexor. These two results are on line with the main design objectives of the Token Bucket traffic control mechanism. In the former case, Case 4, the

system has been configured in a very conservative setup. The controller refuses access to packets that must likely be dropped due to the unavailability of the network to fulfill the applications timing requirements. In the latter case, Case 1, the controller is less conservative and allows the packets to enter the network with the ultimate dropping decisions left to the network. Case 2 represents the worst case. In this case, the Token Bucket mechanism has practically no role to play. Packets are left to get into the network and all losses are waiting losses at the multiplexor. As seen from the figure, this results in a significant higher packet loss. Figure 5.7 shows that Case 2, with a weight of "10 5 1", possesses the largest packet losses.

Optimization using Genetic Algorithm

In this section, we study the benefits of using a Genetic Algorithm on the time required to find the optimal solution. Herein, four cases are considered, Cases 1 to 4 corresponding to systems consisting of one to four Token Buckets, respectively. Throughout this evaluation, standard values for the key parameters have been used.

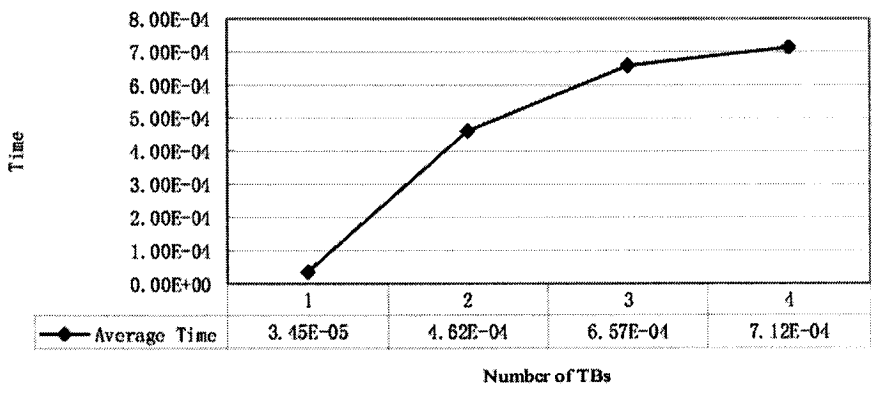


Figure 5.8: Average Search Time of One State

Figure 5.8 depicts the average state calculation time corresponding to different TB numbers. Through experiments we found that when the search space is smaller than 300, the open-loop algorithm runs more efficiently than GA. However, if the search space is larger than this value, GA shows more benefits than the open-loop algorithm. From the data in Figure 5.8, we can see the benefits of genetic algorithm and the tradeoff of the choice between two algorithms. Less TBs in the system indicates that there are more states with small search spaces. Hence, most states in the case 1 runs with the open-loop algorithm. Therefore, the state calculation time in case 1 is much shorter than other 3 cases. Meanwhile, according to the parameters, with the increase of the TB number, the search space in the case i is 20 times lager than the space in the case $i-1$. However, in the figure, instead of 20 times longer, the average time only increases 2.0E-04 seconds from 2 TBs to 3 TBs or 0.5E-04 seconds from 3 TBs to 4 TBs. This shows the GA is an especially useful technique to reduce the computation time for this highly combinatorial problem.

Optimal Number of Processors

From the analysis on the optimal number of PEs presented in the previous chapter and the speedup experiment above, we can better argue that the saying *"more processors mean less time"* is not always true. In this section, we determine the optimal number of PE to run the algorithm.

According to Equation 4.8, if d, the time to search the solution of a state on a single PE, is given, we can figure out the optimal number of PEs for the system. For instance, the average time to search the solution of one state is $3.45e-5$ for a network consisting of 1 Token Bucket. Based on Equation 4.7, the optimal number of PEs is $p = \sqrt{\frac{dv}{c}} \approx 18$. Figure 5.9 depict our numerical results for a network consisting of 1 Token Bucket and 720 stages. Figure 5.9 indicates that the algorithm reaches its best time when 19 PEs are used. This number is in close agreement with

the expected value of 18 PEs. In the figure, 18 PEs has the second shortest time to accomplish 720 stages. Applying a similar reasoning for the networks consisting of 2, 3 and 4 Token Bucket controllers, we obtain the results shown in Table 5.6.

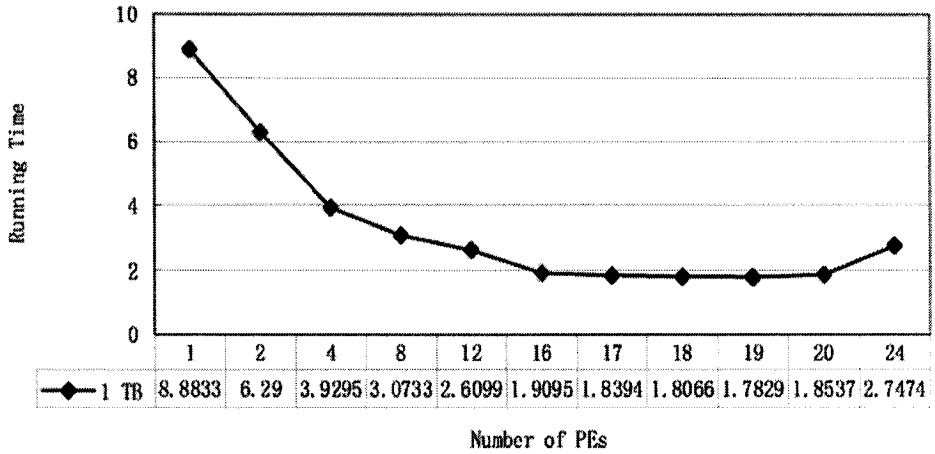


Figure 5.9: Optimal Number of PEs – 1 TB

	1 TB	2 TBs	3 TBs	4 TBs
Number of PEs	18	68	81	84

Table 5.6: Optimal Number of PEs

In addition, using Equation 4.9, the best execution time with 1 Token Bucket can be worked out to be 1.15 seconds. In practice, this time will be slightly longer than anticipated.

5.3.2 Numerical Results for the Self-Similar Traffic

In a second set of experiments, we have used the three aforementioned self-similar traffic traces from Bellcore. Since the most important performance metrics of the algorithm have already been analyzed in the previous sections, in the following we will focus on two important

properties of all parallel processing systems: the speedup and the memory requirements. Unlike the MPEG traffic, self-similar traces are measured in packets instead of frames. Hence, the system parameters are redefined as follows:

- $B_i = 2000\ Bytes, i = 1, 2, \cdots, N$, where N is the number of TB;
- $Q = \sum B_i\ Bytes$, the capacity of the buffer in the multiplexor
- $T = 64\ Bytes$, the size of one token;
- $C = 5.12\ Mbps$;
- $D_i = 4.608\ Mbps$,
- $\tau = 5.02 msec$, the average interval corresponding to the arrival of one single packet.

Running Time and Speedup

In this section we study the execution time and the speedup of the algorithm with self-similar traces. Similar to the MPEG-4 experiments, the number of stages are different corresponding to the number of Token Buckets. Herein, three cases are analyzed:

Case 1:

- Number of Token Buckets: 1
- Number of stages: 720
- Number of states per stage: 1024
- Average number of admissible states per stage: 17

Case 2:

- Number of Token Buckets: 2
- Number of stages: 162
- Number of states per stage: 64512
- Average number of admissible states per stage: 512

Case 3:

- Number of Token Buckets: 3
- Number of stages: 18
- Number of states per stage: 3080192
- Average number of admissible states per stage: 16384

	Number of PEs						
	1 PE	**2 PEs**	**4 PEs**	**8 PEs**	**12 PEs**	**16 PEs**	**24 PEs**
Case 1	15.76	11.35	6.87	4.04	3.08	2.61	3.22
Case 2	3615.88	2573.88	1184.15	654.88	474.74	380.34	231.19
Case 3	34774.3	18639.1	9390	4657.2	3027.3	2260.2	1493.1

Table 5.7: Running Time of Self-Similar Traces (in seconds)

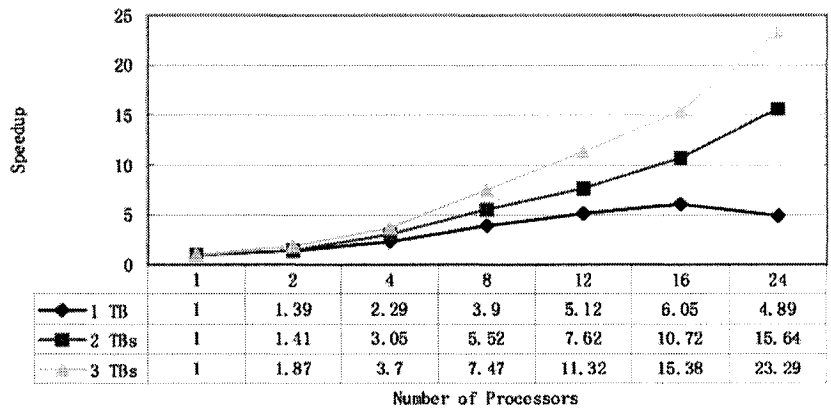


Figure 5.10: Speedup with Self-Similar Traces

Table 5.7 summarizes the running times for the three cases being considered. As expected, the running times as significantly longer than those obtained for the MPEG-4 study. Then, we can attain the speedup of the algorithm in Figure 5.10. The speedup of the algorithm with self-similar traces in Figure 5.10 reveals the same properties of those with MPEG-4 traces: the speedup increase with the TB number. The token size, which is smaller than that of MPEG-4 traces, results in heavy workloads and enhances the ratio between the computation time and the communication time. Therefore, the speedup of self-similar traces appears to be better and the parallel program needs longer time to accomplish the computation compared to MPEG-4 traces.

Memory Requirements

In this section, we study the behavior of the memory requirement in two algorithms with self-similar traces. The case below is with 2 TBs and 1440 stages. Figure 5.11 shows the benefits of parallel algorithm compared to the sequential version. The memory requirement is only 25MB when the parallel program runs with 16 PEs. Because the reduced value of the token size, which is smaller than MPEG-4 traces, our algorithm requires more memories. This also shows that the token size has great influence on the system memory demand.

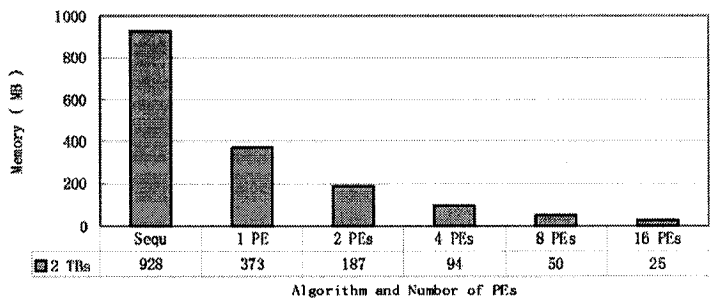


Figure 5.11: Memory Comparison on Self-Similar

Chapter 6

Conclusion

A parallel optimization algorithm for the Token Bucket traffic control mechanism has been designed and evaluated in this thesis. We have started by an in-depth analysis of the optimization algorithm developed by N.U.Ahmed and BoLi et al. [15] [23] [32]. This algorithm is based on a dynamic programming approach. The optimization process is supplemented by a genetic algorithm. The use of the genetic algorithm shortens the search time. Taking as starting point this algorithm, we have proposed the use of parallel processing to further speed up the processing of the optimization algorithm. Furthermore, due to the huge memory requirement of the optimization algorithm, we designed and developed a Reduced Memory Algorithm to reduce the memory requirement. The design of this algorithm was derived from a family of Reduced Memory Algorithms first introduce by J A. Grice et al. [12].

Numerical results illustrate the effectiveness of the proposed parallel algorithm. From our results we have been able to derive some useful guidelines for the settings of the parallel processing system, such as the number of optimal processors, among others. We have also been able to get an insight into the setting of the parameters of the Token Bucket control system, such as the token size, weights assigned to the various types of losses, and others.

Due to the huge resource requirements of the optimization algorithm, there is still a need to further study potential ways to reduce its memory and processing requirements. In order to address these issues, we foresee to study the following issues:

i) To reduce memory demands, we can consider developing the Reduce Memory Algorithm further. The system obtains the minimum space when the skip x is equal to $\sqrt{n}$. Due to the compromise between the time and the memory space, the value of x is set to 1. The development of faster processors may render possible to consider other values, by keeping the computation time at an acceptable level, the memory may be utilized more efficiently.

ii) A *Traffic Prediction* mechanism may be helpful to decrease the processing time. In this case, the parallel algorithm should ignore some states that are irrelevant to the current state or to the optimal solution.

ii) A *Feedback Control Strategy based on Previous Control Decisions* can be applied to the model. Under this control policy, the algorithm can adjust its packet dropping policy according to the current control trend. This could render the model more stable and less sensitive to sporadic spikes.

Besides the aforementioned solutions to improve the performance of the algorithm, there is still the need of studying the buffered version of the Token Bucket. Furthermore, many other proposals have been developed towards the development of a Token Bucket capable of providing various QoS levels. Through the above, we can see that although extensive work has been explored in the area of the Token Bucket control mechanism, there are still many open issues to be further studied.

Bibliography

[1] Online course, http://ikpe1101.ikp.kfa-juelich.de/briefbook_data_analysis/node4.html.

[2] online – Internet, http://ita.ee.lbl.gov/html/contrib/BC.html.

[3] R. Bellman, *Dynamic programming*, Princeton University Press, Princeton, New Jersey, 1957.

[4] C. Bruni and C. Scoglio, *An optimal rate control algorithm for guaranteed services in broadband network*, Computer Networks 37 (2001), 331–344.

[5] R. Bruno, R.G. Garroppo, and S. Giordano, *Estimation of token bucket parameters of VoIP traffic*, Proc. of IEEE ATM Workshop 2000 (1999), 353–356.

[6] J. Digalakis, *Parallelisation of memetic algorithms*, citeseer.nj.nec.com/digalakis03parallelisation.html.

[7] F. Fitzek and M. Reisslein, *MPEG–4 and H.263 traces for network performance evaluation (extended version)*, Tech. Report TKN–00–06, Technical University Berlin, Dept. of Electrical Eng., Germany, October 2000, Traces available at `http://www.tkn.tu-berlin.de/research/trace/trace.html` and `http://www.eas.asu.edu/trace`.

[8] M.J. Flynn, *Some computer organizations and their effectiveness*, IEEE Trans. computers **C-21** (1972), no. 9.

[9] A. Garg M.Y. Sanadidi G. Procissi and M. Gerla, *Token bucket characterization of long-range dependent traffic*, Computer Communications (2002), 1009–1017.

[10] J. Holland, *Adaptation in natural and artificial systems*, 1975.

[11] online – Internet, http://www.hpcvl.org/sun/index.html.

[12] R. Hughey J. A. Grice and D. Speck, *Reduced space sequence alignment*, CABIOS **13(1)** (1997), 45–53.

[13] J.Turner, *New directions in communications (or which way to the information age)*, IEEE communications Magazine **24** (1986).

[14] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson, *On the self-similar nature of ethernet traffic (extended version)*, IEEE/ACM Transactions on Networking **2** (1994), no. 1, 1–15.

[15] B. Li, *On the optimization of the token bucket control mechanism*, Master's thesis, System Science, University of Ottawa, October 2002.

[16] F. Y. Li, *Local and global qos-aware token bucket parameters determination for traffic conditioning in 3rd generation wireless networks*, Proc. of European Wireless (EW) (2002), no. 25-28, 362–368.

[17] A. Lombardo, G. Schembra, and G. Morabito, *Traffic specifications for the transmission of stored mpeg video on the internet*, IEEE Transactions on Multimedia **3** (2001), no. 1, 5–17.

[18] P. Pedroza M. Porto Fernandez, A. de Castro and J. Ferreira de Rezende, *Optimizing fuzzy controllers with genetic algorithms for qos improvement*, 2002.

[19] M. M. Mobius, *Dynamic programming*, http://icg.harvard.edu/ ec2030/lecture/lecture13.pdf.

[20] online – Internet, http://www-unix.mcs.anl.gov/mpi/mpich/.

[21] *Network traffic characterization using token bucket model*, vol. 1, New York, Mar. 1999, the Conference on Computer Communications.

[22] M. Nowostawski and R. Poli, *Parallel genetic algorithm taxonomy*, 1999, citeseer.nj.nec.com/nowostawski99parallel.html.

[23] N.U.Ahmed, Hong Yan, and Luis Orozco-Barbosa, *Performance analysis of the token bucket control mechanism subject to stochastic traffic*, To appear in Dynamics of Continuous, Discrete and Impulsive Systems, http:monotone.uottawa.ca/ journal.

[24] M. Obitko, *Introduction to genetic algorithms*, 1998.

[25] T. Ozawa, *Performance characteristics of a packet-based leaky-bucket algorithm for atm networks*, IEICE Transactions on Communications **E82-B** (1999), no. 1, 305–308.

[26] S-H Park and S-J Ko, *Evaluation of token bucket parameters for vbr mpeg video transmission over the internet*, IEICE TRANS. COMMUN. **E85-B** (2002), no. 1.

[27] H. Pohlheim, *Genetic and evolutionary algorithms: Principles, methods and algorithms*, July 1997, http://geneticalgorithms.ai-depot.com/jump.pl?url=http://www.geatbx.com/docu/algindex.html.

[28] online – Internet, http://www.sun.com/servers/midrange/sunfire6800/.

[29] A.S. Tanenbaum, *Computer networks third edition*, Prentice Hall PTR Upper Saddle River, New Jersey, 1996.

[30] G. Bastin V. Guffens and H. Mounier, *Using token leaky bucket with feedback control for guaranteed boundedness of buffer queue.*

[31] M. Wall, *A c++ library of genetic algorithm components*, online – Internet, http://lancet.mit.edu/ga/.

[32] Q. Wang, *A systems approach to modelling the token bucket algorithm in computer networks. mathematical problems in engineering*, Master's thesis, University of Ottawa, 2000.

Appendix A

Proof of Bellman Equation

According to Bellman's study, if any DP problem can be modelled into Bellman Equation, the solution of this problem is optimal. To prove this, we study a general dynamic optimization problem shown in [19] by Markus:

Find $v(y)$ such that

$$v(y_0) = \sup_{\{y_{t+1}\}_{t=0}^{\infty}} \sum_{t=0}^{\infty} \sigma^t F(y_t, y_{t+1}) \tag{A.1}$$

subject to $y_{t+1} \epsilon \Gamma(y_t)$ with given y_0.

In the equation A.1, y_t is the state at time t; σ is the exponential discount factor; $F(y_t, y_{t+1})$ is the flow payoff function at time t; sup is the supremum function; each state y_t is mapped to a new state y_{t+1}. Suppose $v(y_{t+1})$ is continuation payoff, Bellman Equation can be expressed as:

$$v(y_t) = \sup_{y_{t+1} \epsilon \Gamma(y_t)} \{F(y_t, y_{t+1}) + \sigma v(y_{t+1})\} \tag{A.2}$$

We can readily prove Bellman Equation by substituting y_0 for y_t as the follows:

Proof

$$
\begin{aligned}
v(y_0) &= \sup_{y_{t+1}\epsilon\Gamma(y_t)} \{F(y_0, y_1) + \sigma v(y_1)\} \\
&= \sup_{y_{t+1}\epsilon\Gamma(y_t)} \{F(y_0, y_1) + \sigma F(y_1, y_2) + \sigma^2 v(y_2)\} \\
&\cdots \\
&= \sup_{y_{t+1}\epsilon\Gamma(y_t)} \{F(y_0, y_1) + \cdots + \sigma^{n-1} F(y_{n-1}, y_n) + \sigma^n v(y_n)\} \\
&= \sup_{y_{t+1}\epsilon\Gamma(y_t)} \{\sigma^0 F(y_0, y_1) + \sigma^1 F(y_2, y_2) + \cdots + \sigma^{n-1} F(y_{n-1}, y_n) + \sigma^n v(y_n)\} \\
&= \sup_{\{y_{t+1}\}_{t=0}^{\infty}} \sum_{t=0}^{\infty} \sigma^t F(y_t, y_{t+1})
\end{aligned}
$$

Through the above proof, we can understand that a solution to Bellman Equation is also a solution to the original dynamic optimization problem, vice versa.

Appendix B

MPI versus PVM

The MPI, which is the industry standard for developing the parallel application, was specified by a committee of about forty computing experts from research and industry. The message passing is the kernel of the MPI, which has the flexible capacities to deploy a wide variety of parallel applications. There are different versions of MPI, such as LAM, MPICH and CHIMP, and so on. After recompiled, MPI programs can execute on all MPI implementations due to its platform independent characteristic. MPI has a loose architecture with distributed PEs and memory, especially suitable for MIMD algorithms. MPI introduces the definition of *communicator*, which can be thought of as a binding of a communication context to a group of PEs. This context can be considered as a virtual topology. Group information is collective and the information of the group membership is distributed instead of centralized. In addition, values of the rank range from 0 to $p-1$, where p is the number of PEs. Moreover, MPI provides versatile ways for message communications. Firstly, in point to point communication, it has both blocking and nonblocking "send" and "receive" routines. Secondly, some helper functions such as "receive status" which is used to retrieve unknown lengths of incoming data, are also included. Thirdly, the collective communication functions, such as "barrier", "gather", "allgather", and "scatter", are collected in MPI. Hence, it can provide sufficient support both for simple programs and complicated

system developments. MPI can create multiple virtual PEs on one single PE and it can execute programs on them as well. In order to increase efficiency and avoid context swapping, each PE usually runs one single process that is in charge of computations and communications. This sort of programming scheme is particularly suitable for coarse-grain applications.

In 1991, Bob Manchek joined a research on heterogenously distributed computing and implemented a portable, robust version of PVM design (PVM 2.0). Unlike MPI with the definition of "message passing", the kernel of PVM was the notion of "virtual machines" – a set of heterogeneous hosts connected by the network which appears logically to the user as a single large parallel computer. The parallel computation of the PVM computing model is based on the idea that an application consists of several tasks. Each of them is responsible for a part of the application's computational workloads. All PVM tasks are identified by an integer task identifier (TID), by which messages are sent to and received from different PEs. The PVM includes the concept of user named groups that are designed to be very general and transparent to the user. Furthermore, The PVM uses sockets to implement message exchange and does not move beyond the basic message passing paradigm.

By comparing two parallel softwares, we can discern difference between them. PVM only supports some simple data exchange functions because it is based on the idea of "virtual machines". Additionally, the PVM does not implement remote-memory operations and does not define clear parallel-I/O operations.

Concerning the flexibility and the structure of the parallel machine, we choose MPI as the parallel platform for the implementation. It provides sophisticated communication routines to the program.

Appendix C

Decision on Space-Time Tradeoff

Through the system analysis, we summarize some formulas on the time and memory issues here. Equations of time issues can be defined as the follows:

$$T_{trace}\begin{cases} = [\frac{(x-1)nm}{x+1} + \frac{n}{x+1}]d & x > 0 \quad \text{(C.1a)} \\ \approx 0 & x = 0 \quad \text{(C.1b)} \end{cases}$$

$$T_{fill} = nm\{\frac{cp}{\upsilon} + \frac{d}{p} - \frac{c}{\upsilon}\} \quad \text{(C.2)}$$

$$T = (T_{fill} + T_{trace})/60 \quad Minutes \quad \text{(C.3)}$$

The memory requirement can be illustrated as the below list:

$$M_{trace} = \begin{cases} mc(x-1) & x > 0 \quad \text{(C.4a)} \\ 0 & x = 0 \quad \text{(C.4b)} \end{cases}$$

$$M_{fill} = \frac{nmc}{x+1} \quad \text{(C.5)}$$

$$M = (M_{fill} + M_{trace})/10^6 \quad MB \tag{C.6}$$

Hence, the space-time Tradeoff function, which considers both the time and memory issues can be defined herein:

$$f(x) = aT + bM \tag{C.7}$$

Weights a, b are assigned according to different scenarios. The larger weight to one component implies we prefer that the value of this part would be smaller.

The objective function for the skip x is $f(x) = a\frac{T_{fill}+T_{trace}}{60} + b\frac{M_{fill}+M_{trace}}{10^6}$. The value of $f(x)$ is optimal when $f'(x) = 0$. From the equation C.7, we can know that when

$$x = \sqrt{n - \frac{(2 - \frac{1}{m})adn10^5}{6bc}} - 1 \qquad x > 0 \tag{C.8}$$

$f(x)$ can obtain its optimal value.

Proof $f(x) = a\frac{T_{fill}+T_{trace}}{60} + b\frac{M_{fill}+M_{trace}}{10^6}$. Then, $f'(x) = \frac{ad(2m-1)n}{(x+1)^2 60} - \frac{bmcn}{(x+1)^2 10^6} + \frac{mcb}{10^6} = \frac{ad(2m-1)n10^5 - bmcn6 + 6mcb(x+1)^2}{(x+1)^2 10^6} = 0$. Meanwhile, to minimize $f(x)$, $f''(x) = \frac{nmbc - adn(2m-1)}{(x+1)^3 30} > 0$ must be true. It is easy to figure out that Equation C.8 is the optimal solution of the function.

Solution C.8 illustrates that with the increase of the number of states m, the value of x decreases; c can be viewed as a constant with no strong influence on x; a higher value of d can result in a smaller value of x.

The relationship between two weights a and b can be one of the following three conditions:

$$\begin{cases} \frac{a}{b} < 1 & \text{the memory is more important than the time;} \\ \frac{a}{b} = 1 & \text{the memory is as important as the time;} \\ \frac{a}{b} > 1 & \text{the time is more important than the memory.} \end{cases}$$

In the experimental environment, we concern the time as a more precious resource than the memory space especially when the tradeoff is between 1 minutes and 1 MB. Therefore, $\frac{a}{b}$

should be large enough as long as $[1 - \frac{(2-1/m)ad10^5}{6bc}] > 0$. Otherwise, x will have no optimal value. Normally, $\frac{1}{m} \approx 0$ because the value of m is much larger than 1. In addition, we can use approximate values from experiments for following variables: $c = 10$; $d = 3 \times 10^{-4}$. Hence, Equation C.8 can be transformed to $x = \sqrt{n - \frac{an}{b}} - 1$. To optimize the skip x, $\frac{a}{b}$ must be less than 1. However, we also expect the value of $\frac{a}{b}$ can be as large as possible. Furthermore, because m is greatly larger than 1, $f''(x) \approx \frac{nmbc-2adnm}{(x+1)^3 30} > 0$. This requires $\frac{c}{2d} > \frac{a}{b}$. It is easy to figure out this equation is true with above $\frac{a}{b}, c$, and d. Therefore, the optimal value of x is the constant 1 when $x > 0$. Then, the possible choice is between values 0 or 1.

$$f(x) = \begin{cases} \frac{aT_{fill}}{60} + \frac{bnmc}{10^6} & x = 0 \\ \frac{a(\frac{nd}{2} + T_{fill})}{60} + \frac{bnmc}{2 \times 10^6} & x = 1 \end{cases}$$

Apply $d = 3 \times 10^{-4}$ into this formula, the choice becomes values of $10n$ when $x = 1$ and $2bnmc$ when $x = 0$. Clearly, when the skip x is set to 1, the system can have a better tradeoff between the space and the time.

Appendix D

Source code of Parallel TB Control System

```
#include <ctime>
#include <mpi.h>
#include <zlib.h>

using namespace std;

#define NUM_OF_CACHE_STAGES  2

#define SIZE_OF_SHORT    sizeof(unsigned short)
#define SIZE_OF_INT    sizeof(int)

#define TRAFFIC(row, col)  (row * NUMBER_OF_TOKENS + col )
#define TABLEELEMENT(col)  (col * SIZEOF_OF_TABLEELEMENT)
#define STATES(col)    (col)
#define CONTROLS     0
```

```
#define COST        SIZE_OF_INT/SIZE_OF_SHORT

class ParaTB;

struct checkPointItem{
  unsigned int controlsValue;
  unsigned int cost;
};


/* define a stucture which can maintain the element of the value table*/
struct tableElement
{
  unsigned short *states;
  unsigned short *control;
  int  cost;
};

struct optimizeData{
  unsigned short *states;
  unsigned short *address;
  ParaTB*  objPointer;
};


class ParaTB{
```

```
public:
  //retrieve parameters from configuration, treated as constants.
  unsigned short NUMBER_OF_TOKENS;
  unsigned short SIZEOF_OF_TABLEELEMENT;
  unsigned short COMPRESSCOMMUNICATION;

  double SERVICERATE; // the output link service rate
  unsigned short A; // the weight alpha
  unsigned short B; // the weight betta
  unsigned short C; // the weight gamma
  double TIMEINTERVAL;// the time between two stages
  unsigned short QUEUESIZE;  // the queue capacity of the multiplexor
  unsigned short *LINKCAPACITY; //the link rates from TB to multiplexor
  unsigned short *TBSIZE;  // the capacity of TBs


  unsigned short population; // the GA population size
  unsigned short generation; // number of generations of GA

  unsigned short STAGE; // record the total number of stages
  unsigned short (*Traffic); //all traffic from all TBs and all stages

private:

  //member variables
  unsigned short *confirmedTraffic; // conforming traffic of each state
  short stage; // keep the change of stage, must be signed
```

```
unsigned short acceptedMul; // the accepted bytes by the multiplexor
unsigned short *initialState; // store the inital states of TB and Queue
int totalState; // keeps track of the number of possible states
int optimalIndex; //keeps track of the array index of the optimal value

//path route information, called checkpoint
struct checkPointItem **checkPoints;

//pow_base for reducing the time
double *pow_base;
double *control_pow_base;

/* this is for local cache of states, control and costs in all processor */
// please know that processor 0 also occupies localStatesPerCPU in this array.
//struct tableElement *cacheStates;
unsigned short *cacheStates;

/* this is for sending out the local result */
//struct tableElement *tmpStatesPerCPU;
unsigned short *tmpStatesPerCPU;

/* because can not divide workload preciously, so there is some more than needed
*/
int actualStatesPerCPU ;
int localStatePerCPU;

//variables store information of command arguments
```

```
  int ac;
  char **av;

  //the range of local states in one stage
  int   queue_down_bound,  queue_up_bound;

  //the current state
  unsigned short  *states;

  //variables for memory distribution
  int localStages, indexOfCheckpoints;
  int numOfCheckpoints;

  //GA parameters
  GAParameterList params;


public:
  unsigned int seed;

protected:

  /* for testing to exit program */
  void programExit();
  /* compute the variance of the traffic trace*/
  double variance(unsigned short,unsigned short);
  /* compute the standard Deviation of the traffic trace*/
```

```
double StandardDeviation(double);
/* compute objective1 function without running GA*/
float objective1(unsigned short G[], struct optimizeData* optData);
/* compute the current conforming traffic*/
unsigned short confirmTraffic(unsigned short []);
/* search the optimal value for the table */
int searchNextV(unsigned short []);
/* compute the losses at the multiplexor*/
int LostAtmultiplexor(unsigned short);
/* compute the total losses at TBs*/
int TotalLostAtTB();
/* compute the losses at a single TB*/
int LostAtTB(unsigned short, unsigned short);
/* compute the waiting losses at the Queue*/
int WaitingLost(unsigned short,unsigned short);
/* compute the accepted traffic by multiplexor
   calling by WaitingLost() and LostAtMultiplier()*/
int AcceptedTrafficByMul(unsigned short ,unsigned short);

/* calculate the control value into index */
inline int controls2index(unsigned short *ptr);

/* reverse the index into the control value*/
inline void index2controls(int index, unsigned short *ptr);

/* get the appropriate rank according to the stage */
int getRank(int stage);
```

```
public:

  void  SetCommandParameters(int argc, char **argv)
  {
    ac = argc;
    av = argv;
  }


  /* trace the optimal value from the initial state*/
  int traceBack();

  /* for processor root to append data into current buffer */
  void appendMemoryDump();

  // optimize one state by given parameters
  int optimizeOneState(struct optimizeData *optData, unsigned short *G);

  ParaTB() {
    timeOld = MPI_Wtime();
    start = timeOld;
    A = 0;
    B = 0;
    C = 0;
    seed = 0;
    totalState=1;
```

```
  optimalIndex=0;

  checkPoints = NULL;

  pow_base = NULL;
  control_pow_base = NULL;

  actualStatesPerCPU = NULL ;
  localStatePerCPU = NULL;

  bnonGA = false;

}
~ParaTB() {
}

//figure out the local task range
void  PartitionLoad();
//allocate memories
void  AllocateMemory();
//set values of parameters for checkpoints
void  SetupCheckpoints();
//set values of parameters for GA
void  SetupGAParameters();
//set values of other variables
void  SetupOthers();
//compute all stages in backward scheme
```

```
  void  ComputeStages();

  //release all memory required by the system
  void  FreeMemory();

  /* read number of stages from a file*/
  void readStage();
  /* read traffic traces from a file*/
  void readTraffic();
  /* read all parameters from a file*/
  void readParameters();

  //parameters of configuration
  void  TransferConfiguration();
  //values of traffic data
  void  TransferTraffic();


public:

  //variables relative to parallel environment
  int size,rank;

private:
  //local variable only for debugging and performance

  double timeOld, timeNew, timeElapsed;
```

```
  double   start, finish;
     double   elapsed_time;

public:
  /* compute objective function with GA*/
  friend float objective(GAGenome & c);

};


/*============================================================================
    start of the main program
============================================================================*/
int main(int argc, char **argv)
{

  ParaTB  paraTBObj;

  /* read a seed value from the console argument*/
  for(int i=1; i<argc; i++) {
    if(strcmp(argv[i++],"seed") == 0) {
      paraTBObj.seed = atoi(argv[i]);
    }
  }

  paraTBObj.SetCommandParameters(argc, argv);
  /*****Parallel Modification*****/
```

```
MPI_Init(&argc, &argv);
  //MPI::Init(argc, argv);

MPI_Comm_size(MPI_COMM_WORLD, &(paraTBObj.size) );
    MPI_Comm_rank(MPI_COMM_WORLD, &(paraTBObj.rank) );



/*****Parallel Modification end*****/
if (paraTBObj.rank==0){
  /* read the number of stages */
  paraTBObj.readStage();
  /* read all parameters from a file*/
  paraTBObj.readParameters();
}

paraTBObj.TransferConfiguration();
paraTBObj.TransferTraffic();



paraTBObj.PartitionLoad();
paraTBObj.AllocateMemory();

paraTBObj.SetupCheckpoints();
paraTBObj.SetupGAParameters();
paraTBObj.SetupOthers();

paraTBObj.ComputeStages();
```

```
  int rankFinished = paraTBObj.traceBack();


  //do not need MPI routines now, realease resources now.
  MPI_Finalize();


  paraTBObj.FreeMemory();

  return 0;
}
/*============================================================================
            End of the main program
============================================================================*/

void ParaTB::TransferConfiguration()
{
  /** broadcast all parameters to all other processors**/

  MPI_Bcast(&STAGE, 1, MPI_UNSIGNED_SHORT, 0, MPI_COMM_WORLD);
  MPI_Bcast(&QUEUESIZE, 1, MPI_UNSIGNED_SHORT, 0, MPI_COMM_WORLD);
  MPI_Bcast(&population, 1, MPI_UNSIGNED_SHORT, 0, MPI_COMM_WORLD);
  MPI_Bcast(&generation, 1, MPI_UNSIGNED_SHORT, 0, MPI_COMM_WORLD);
  MPI_Bcast(&COMPRESSCOMMUNICATION, 1, MPI_UNSIGNED_SHORT, 0, MPI_COMM_WORLD);
  MPI_Bcast(&NUMBER_OF_TOKENS, 1, MPI_UNSIGNED_SHORT, 0, MPI_COMM_WORLD);
```

```
  MPI_Bcast(&A, 1, MPI_UNSIGNED_SHORT, 0, MPI_COMM_WORLD);
  MPI_Bcast(&B, 1, MPI_UNSIGNED_SHORT, 0, MPI_COMM_WORLD);
  MPI_Bcast(&C, 1, MPI_UNSIGNED_SHORT, 0, MPI_COMM_WORLD);

  MPI_Bcast(&SERVICERATE, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast(&TIMEINTERVAL, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);


  if (rank != 0)
  {
    TBSIZE = new unsigned short [NUMBER_OF_TOKENS];
    LINKCAPACITY = new unsigned short [NUMBER_OF_TOKENS];
    initialState = new unsigned short [NUMBER_OF_TOKENS+1];
  }

  MPI_Bcast(LINKCAPACITY, NUMBER_OF_TOKENS, MPI_UNSIGNED_SHORT, 0, MPI_COMM_WORLD);
  MPI_Bcast(TBSIZE, NUMBER_OF_TOKENS, MPI_UNSIGNED_SHORT, 0, MPI_COMM_WORLD);
  MPI_Bcast(initialState, NUMBER_OF_TOKENS+1, MPI_UNSIGNED_SHORT, 0,
MPI_COMM_WORLD);
}


void ParaTB::TransferTraffic()
{
    Traffic = new unsigned short [STAGE * NUMBER_OF_TOKENS];
    if (rank==0)
```

```
{
  /* read traffic traces to the two dimentional array*/
  readTraffic();
}

confirmedTraffic = new unsigned short[NUMBER_OF_TOKENS];

//SIZEOF_OF_TABLEELEMENT = NUMBER_OF_TOKENS * 2 + 1 + SIZE_OF_INT/SIZE_OF_SHORT;
SIZEOF_OF_TABLEELEMENT = 2 * SIZE_OF_INT/SIZE_OF_SHORT;

if (rank != 0 ) {
  for(short int i3=0;i3<NUMBER_OF_TOKENS;i3++)
    totalState=totalState*(TBSIZE[i3]+1);
  totalState=totalState*(QUEUESIZE+1);
}

//declare datatype for Traffic communication
//MPI::Datatype trafficType;
MPI_Datatype trafficType;

if (NUMBER_OF_TOKENS > 1){
  //trafficType = MPI::UNSIGNED_SHORT.Create_contiguous(NUMBER_OF_TOKENS);
  //trafficType.Commit();
  MPI_Type_contiguous(NUMBER_OF_TOKENS, MPI_UNSIGNED_SHORT, &trafficType);
  MPI_Type_commit(&trafficType);
}
else
```

```
    //trafficType = MPI::UNSIGNED_SHORT;
    trafficType = MPI_UNSIGNED_SHORT;

    ////////////////////////////
    /** broadcast Traffice to others **/
    //MPI::COMM_WORLD.Barrier();
  MPI_Bcast(Traffic, STAGE, trafficType, 0, MPI_COMM_WORLD);
  //MPI::COMM_WORLD.Bcast(Traffic, STAGE, trafficType, 0);

  if (NUMBER_OF_TOKENS > 1)
    //trafficType.Free();
    MPI_Type_free(&trafficType);

  return;

}

void ParaTB::PartitionLoad()
{
  /* uniformly divide workload, to use processor 0*/
  int skip = totalState/size;
  int surplus = totalState % size;
  //adjust workload because no leak between two Processor
  if ( totalState % size != 0)
    skip = skip + 1;

  //calculate the up and lower bound for each processor.
```

```
  //change this to totalState later. Then use while in each stage.
  //if (rank!=0){
    queue_up_bound = rank*skip;
    queue_down_bound = (rank+1)*skip - 1;
  //}
  if ( (rank==size-1) && ( queue_down_bound >= totalState ) )
    queue_down_bound = totalState - 1;

  //if (rank!=0)
  //  cout << "queue up bound "<<queue_up_bound << " and down bound " <<
queue_down_bound << " pow_base:"<<pow_base<<" at rank " << rank <<endl;

  /* calculate out how many states in each CPU, because we divide workload
     by Multiplexor size */
  localStatePerCPU = skip;
  //calculate actual number of states for gathering
  actualStatesPerCPU = localStatePerCPU * size;
}

void ParaTB::AllocateMemory()
{
  /* allocate enough memory for all states*/
  states = new unsigned short [NUMBER_OF_TOKENS+1];
  /* allocate memory for communicating states in each processor  */
  tmpStatesPerCPU = new unsigned short [localStatePerCPU * SIZEOF_OF_TABLEELEMENT];
  if (tmpStatesPerCPU == NULL)
    cout << "memory allocation error at rank " << rank << endl;
```

```
  /* allocate memory to our states tables in cache */
  cacheStates = new unsigned short
[NUM_OF_CACHE_STAGES*SIZEOF_OF_TABLEELEMENT*actualStatesPerCPU];
  if (cacheStates == NULL)
    cout << "cache memory allocation error at rank " << rank << endl;
}

void ParaTB::SetupCheckpoints()
{
  // variables for choosing which PE to store checkpoints
  localStages = (STAGE + size - 1) / size;
  if ( (STAGE%size) && (rank >= STAGE % size) )
    localStages--;
  int countStages = 0;

  //filling procedure from the last stage n-1 to the first stage 0
  //so checkpoints saved in the same direction
  int rankIndex = size - 1;
  numOfCheckpoints = ceil((double)localStages/2);

  checkPoints = new struct checkPointItem* [numOfCheckpoints];
  for (int i=0; i<numOfCheckpoints; i++)
    checkPoints[i] = NULL;

  indexOfCheckpoints = 0;
}
```

```
void ParaTB::SetupGAParameters()
{
  /* create GAParameterList object and set ga paremeters */
  GASteadyStateGA::registerDefaultParameters(params);
  params.set(gaNpopulationSize, population);
  params.set(gaNelitism,gaTrue);
  params.set(gaNpCrossover, 0.9);
  params.set(gaNpMutation, 0.04);
  params.set(gaNnGenerations, generation);
  params.set(gaNselectScores,GAStatistics::Minimum);
  params.parse(ac, av, gaFalse);
}


void ParaTB::SetupOthers()
{
  //need one more elelment in this array.
  pow_base = new double [NUMBER_OF_TOKENS + 1];
  pow_base[NUMBER_OF_TOKENS] = 1;
  for (int i=NUMBER_OF_TOKENS-1; i>=0; i--)
    if (i==NUMBER_OF_TOKENS-1)
      pow_base[i] = pow_base[i+1] * (QUEUESIZE+1);
    else
      pow_base[i] = pow_base[i+1] * (TBSIZE[i+1]+1);
```

```
  control_pow_base = new double [NUMBER_OF_TOKENS];
  for (int i=0; i<NUMBER_OF_TOKENS; i++)
    if (i==0)
      control_pow_base[i] = 1;
    else
      control_pow_base[i] = control_pow_base[i-1] * (TBSIZE[i]+1);

  timeNew = MPI_Wtime();
  timeElapsed = timeNew - timeOld;

}

void ParaTB::ComputeStages()
{

  /* IMPORTANT ==== the running of the whole optimal control process tarts*/
  for ( stage=STAGE-1; stage>=0; stage--)
  {
    if (rank == 0)
      cout<<"running stage : "<<stage<<" at rank " << rank << endl;

    int n = 0;  //keeps the index of the totalstates
    int localOptimal = 9999999;

    /* find where the optimal values of the next stage stores.*/
    if(optimalIndex==0)
    {
```

```
  optimalIndex=1;
}
else optimalIndex=0;

int counter = queue_up_bound;
while (counter <= queue_down_bound)
{
  double tmpStates = counter;
  counter++;

  for (int i=0; i<NUMBER_OF_TOKENS+1; i++)
  {
    states[i] = (unsigned short)(tmpStates/pow_base[i]);
    tmpStates = tmpStates - states[i] * pow_base[i];
  }

  /* an array of control values */
  unsigned short *G = new unsigned short [NUMBER_OF_TOKENS];

  struct optimizeData optData;
  optData.states = states;
  optData.objPointer = this;

  unsigned short otherIndex;

  if (optimalIndex==0) otherIndex=1;
    else otherIndex=0;
```

```
      optData.address = cacheStates + TABLEELEMENT(otherIndex*actualStatesPerCPU);

      localOptimal = optimizeOneState(&optData, G);

      unsigned short *ptr = tmpStatesPerCPU + TABLEELEMENT(n);

      int controlValues = controls2index(G);

      memcpy(ptr + CONTROLS, &controlValues, SIZE_OF_INT);
      //to avoid the alignment problems
      memcpy(ptr + COST, &localOptimal, SIZE_OF_INT);

      delete [] G;
      n++;

    }

    MPI_Barrier(MPI_COMM_WORLD);

    MPI_Allgather(tmpStatesPerCPU,
localStatePerCPU*SIZEOF_OF_TABLEELEMENT*SIZE_OF_SHORT,
      MPI_CHAR, (cacheStates+TABLEELEMENT(optimalIndex*actualStatesPerCPU)),
      localStatePerCPU*SIZEOF_OF_TABLEELEMENT*SIZE_OF_SHORT,
      MPI_CHAR, MPI_COMM_WORLD);

    //find the appropriate processor to save the current information
```

```
int  rankIndex = 0;
if (rank < STAGE%size)
{
  if (stage<((STAGE%size)*localStages))
    rankIndex = stage / localStages;
  else
    rankIndex = (stage - STAGE%size) / (localStages - 1);
}
else
{
  if ( stage < ((STAGE%size)*(localStages+1)) )
    rankIndex = stage / (localStages + 1);
  else
    rankIndex = (stage - STAGE%size ) / localStages;
}

if (stage%2 == 1 && rank == rankIndex)
{
  if ( (indexOfCheckpoints < numOfCheckpoints) )
  {
    cout << "stages "<< stage << " save checkpoints at PE " << rank << endl;
    appendMemoryDump();
  }

  // else  cout << "index " << indexOfCheckpoints << " overflow with " <<
numOfCheckpoints << " at rank " << rank << endl;
}
```

```
    if (rank == rankIndex)
      cout << "stage " << stage << " at PE " << rank << " with index " << rankIndex
<< endl;

    if (rank == 0)
      cout<<"finished stage "<<stage<<" at rank " << rank << endl;
  }

}

void ParaTB::FreeMemory()
{

  //free all memory used in the programme
  delete [] states;
  delete [] Traffic;
  delete [] confirmedTraffic;
  delete [] pow_base;
  delete [] cacheStates;
    delete [] tmpStatesPerCPU;

  for (int i=0; i<numOfCheckpoints; i++)
    if (checkPoints[i] != NULL)
      delete checkPoints[i];
  delete [] checkPoints;
}
```

```
/*** for testing **/
void ParaTB::programExit()
{
  delete []Traffic;
  delete []tmpStatesPerCPU;

  //MPI::Finalize();
  MPI_Finalize();
  exit (0);
}

void ParaTB::appendMemoryDump()
{
  static int stage_count = STAGE-1;

  int index = indexOfCheckpoints;


  //one of integer for control value, another for optimal cost
  checkPoints[index] = new struct checkPointItem [actualStatesPerCPU];

    //unsigned short *currentStates = new unsigned short [ NUMBER_OF_TOKENS + 1];
    unsigned short *currentControls = new unsigned short [ NUMBER_OF_TOKENS];
    //unsigned short *nextStates = new unsigned short [ NUMBER_OF_TOKENS + 1];
```

```
    for (int i=0; i< totalState - 1; i++)
    {
      //assuming no compression in this algorithm

      //get the current state in this stage;
      unsigned short *ptr = cacheStates +
TABLEELEMENT(actualStatesPerCPU*optimalIndex) + TABLEELEMENT(i);
      //for (int j=0; j<NUMBER_OF_TOKENS+1; j++)
      //  currentStates[j] = *(ptr+STATES(j));
      int controlsValue = 0;
      memcpy(&controlsValue, ptr + CONTROLS, SIZE_OF_INT);
      index2controls(controlsValue, currentControls);

      //for (int j=0; j<NUMBER_OF_TOKENS; j++)
      //  currentControls[j] = *(ptr+CONTROLS(j));

      int controlValue = controls2index(currentControls);

      //if (stage_count==STAGE-1 && currentControls[0]==6 && currentControls[1]==15)
      //  cout << "at last stage, index is " << index << endl;

      //sotre control values
      checkPoints[index][i].controlsValue = controlValue;
      //store cost values
      memcpy( &(checkPoints[index][i].cost),ptr + COST, SIZE_OF_INT);

    }
```

```
    delete [] currentControls;

  indexOfCheckpoints = index + 1;
  stage_count--;
}

int ParaTB::optimizeOneState(struct optimizeData *optData, unsigned short *G)
{
  /*  find the boundary for control u  */
  /* Declare MaxU[3] to store the boundary for u */
  unsigned short *MaxU = new unsigned short [NUMBER_OF_TOKENS];

  /* initially define use ga to find the best value for current state*/
  bool runGa=true;

  /* initially define the number of admissible control values*/
  unsigned short totalU=1;

  float o;

  int localOptimal = 9999999;
  /* compute the Maximum number of token generating  */
  for(unsigned short j=0; j<NUMBER_OF_TOKENS; j++)
  {
  /* the Maxcimum number of token generating can't
    exceed the available space in TB */
```

```
    MaxU[j] = TBSIZE[j] - states[j];
  }

  /* compute the total number of admissable control values */
  for(unsigned short u=0;u<NUMBER_OF_TOKENS;u++)
  {
    totalU=totalU*(MaxU[u]+1);
  }

  /* if the search space smaller than 300, we do not use GA
  to find the optimal value, since from several testings,
    we found that running speed is faster without using GA
    if the search space is smaller than 300*/
  if(totalU<300)
  {
    runGa=false;
  }

  /* run GA */
  if(runGa==true)
  {

    /* define the search space */
    GAAlleleSet<unsigned short> *a = new GAAlleleSet<unsigned short>
[NUMBER_OF_TOKENS];
    GAAlleleSetArray<unsigned short> alleles;
```

```
for(unsigned short n1=0;n1<NUMBER_OF_TOKENS;n1++)
{
  for(unsigned short h=0; h<=MaxU[n1]; h++)
  {
    a[n1].add(h);
  }
  alleles.add(a[n1]);
}

/* define a genome object */
GA1DArrayAlleleGenome<unsigned short> genome(alleles, objective, optData);

/* define a GA object */
GASteadyStateGA ga(genome);
/* add all ga parameters to ga*/
ga.parameters(params);
/* tell ga compute the minimum value*/
ga.minimize();
/* initilaize the first generation of population*/
ga.initialize(seed);

/* run ga until reach the number of generation
  pre-defined*/
int g=0;
while(g<generation)
{
  ga.step();
```

```
    g++;
  }

  /* assign the value of the best individual to genome*/
  genome=ga.population().best();

  /* assign the each value of control to corresponding
    TBs to an array*/
  for(unsigned short i=0;i<NUMBER_OF_TOKENS;i++)
  {
    G[i]= genome.gene(i);
  }
  /* compute the best value corresponding to
    the best genome GA found*/
  o=objective1(G, optData);
  localOptimal = (int)o;

  //if (localOptimal < 0)
  //  cout << "optimal value less than 0 in GA at rank " << rank<<endl;
  delete [] a;
}
/* if the search space is small, find the best value
  without GA*/
else
{
  unsigned short index = 0;
  unsigned short *c = new unsigned short [NUMBER_OF_TOKENS];
```

```
int optimalV;

int min=999999;
int minIndex=0;


// go through all the admissable control values
// now this must be done dynamically

int arraysIndex=NUMBER_OF_TOKENS-1;
unsigned short *arrayStates = new unsigned short [NUMBER_OF_TOKENS];

for (int a=0; a<NUMBER_OF_TOKENS; a++)
  arrayStates[a] = 0;

for (int index=0; index<totalU; index++)
{
  optimalV =(int)(objective1(arrayStates, optData));

  if(optimalV <= min)
  {
    min=optimalV;
    minIndex=index;

    for (int b=0; b<NUMBER_OF_TOKENS; b++)
      c[b] = arrayStates[b];
```

```
}

bool blnReset = false;
bool blnStop = false;

//adjust the arrayIndex and arrayStates for next step
do {
  //increase the state
  arrayStates[arraysIndex]++;

  //check whether the state exceed the maximal value
  if ( arrayStates[arraysIndex] > MaxU[arraysIndex] )
  {
    blnReset=true;
    arraysIndex--;
    //cout << " set arrayFlags to false" << endl;
  }else blnStop = true;
}
while (blnStop==false && arraysIndex >=0 );

if (blnReset==true)
{
  //adjust all states behind the current one.
  for (int b=arraysIndex+1; b<NUMBER_OF_TOKENS; b++)
    arrayStates[b] = 0;

  //always keep the last one firstly exhausted
```

```
        arraysIndex = NUMBER_OF_TOKENS - 1;
      }

    }

    delete [] arrayStates;

    for (int b=0; b<NUMBER_OF_TOKENS; b++)
      G[b] = c[b];

    localOptimal=min;

    delete []c;
  }

  delete [] MaxU;

  return localOptimal;

}

/* read how many stages */
void ParaTB::readStage()
{
      fstream file;
      file.open("Traffic.txt", ios::in);
      file>>STAGE;
```

```
      file.close();
}

void ParaTB::readTraffic()
{
  fstream file;
  file.open("Traffic.txt",ios::in);
  file>>STAGE;

  for(unsigned short l=0;l<STAGE;l++)
  {
    for(unsigned short l1=0;l1<NUMBER_OF_TOKENS;l1++)
    {
      file>>Traffic[TRAFFIC(l,l1)];
      cout<<Traffic[TRAFFIC(l,l1)]<<",";
    }
    cout<<endl;
  }
  file.close( );
}

double ParaTB::variance(unsigned short mean, unsigned short  source)
{

  int v=0;
  for(int i=0; i<STAGE;i++)
  {
```

```
    v=v+ pow((double)(mean-Traffic[TRAFFIC(i, source)]),2);
  }

  return v/STAGE;
}

double ParaTB::StandardDeviation(double variance)
{
  return sqrt(variance);
}

void ParaTB::readParameters()
{
  fstream f;
  f.open("new_parameters.txt",ios::in);
  char l[50];
  f.getline(l,50);
  f>>population;
  f>>generation;
  f.getline(l,50);
  f.getline(l,50);
  f>>A;
  f>>B;
  f>>C;
  f.getline(l,50);
  cout<<"===Alpha "<<A <<", beta="<<B<<", gamma="<<C<<endl;
  f.getline(l,50);
```

```
//  cout<<l<<endl;

  f>>TIMEINTERVAL;
  cout<<"===TimeInterval "<<TIMEINTERVAL<<endl;

  f.getline(l,50);
  f.getline(l,50);

  f>>COMPRESSCOMMUNICATION;
  cout<<"===Compress Communication "<<COMPRESSCOMMUNICATION<<endl;

  f.getline(l,50);
  f.getline(l,50);
//  cout<<l<<endl;

  f>>SERVICERATE;
  cout<<"===ServiceRate "<<SERVICERATE<<endl;

  f.getline(l,50);
  f.getline(l,50);
//  cout<<l<<endl;
  f>>QUEUESIZE;
  cout<<"===QueueSize " <<QUEUESIZE<<endl;

  f.getline(l,50);
  f.getline(l,50);
```

```
  f>>NUMBER_OF_TOKENS;
  cout<<"===Number of Token Buckets "<<NUMBER_OF_TOKENS<<endl;

  f.getline(1,50);
  f.getline(1,50);

//  cout<<1<<endl;
//  cout<<"******"<<QUEUESIZE<<endl;
  TBSIZE = new unsigned short [NUMBER_OF_TOKENS];
  for(int i=0;i<NUMBER_OF_TOKENS;i++)
  {

    f>>TBSIZE[i];
    cout<<"******TBSIZE["<<i<<"] "<< TBSIZE[i]<<".";
  }
  cout<<endl;
  f.getline(1,50);
  f.getline(1,50);

//  cout<<1<<endl;
  LINKCAPACITY = new unsigned short [NUMBER_OF_TOKENS];
  for(int i2=0;i2<NUMBER_OF_TOKENS;i2++)
  {
    f>>LINKCAPACITY[i2];
    cout<<"******LINKCAPACITY["<<i2<<"] "<<LINKCAPACITY[i2]<<",";
  }
  cout<<endl;
```

```
  f.getline(l,50);
  f.getline(l,50);
  cout<<l<<endl;

  initialState = new unsigned short [NUMBER_OF_TOKENS+1];
  for(int i1=0;i1<NUMBER_OF_TOKENS;i1++)
  {
    f>>initialState[i1];
    cout<<"******TOKEN INITIAL["<<i1<<"] "<<initialState[i1]<<endl;
  }

  f>>initialState[NUMBER_OF_TOKENS];
  cout<<"******multiplexor INITIAL "<<initialState[NUMBER_OF_TOKENS]<<endl;

  for(short int i3=0;i3<NUMBER_OF_TOKENS;i3++)
  {
    totalState=totalState*(TBSIZE[i3]+1);
  }
  totalState=totalState*(QUEUESIZE+1);
  cout<<"total state "<<totalState<<endl;
  f.close();

}

/* this function is used by no GA computation*/
float ParaTB::objective1(unsigned short G[], struct optimizeData* optData)
{
```

```
float total=0; // it stores the total cost

/* define and initial the state of the next stage correspondin to
   the current state*/
unsigned short *nextState = new unsigned short [NUMBER_OF_TOKENS+1];
for (int i=0; i<NUMBER_OF_TOKENS+1; i++)
  nextState[i]=0;

/* copmute the conforming traffic*/
unsigned short totalConfirmed=confirmTraffic(G);

/* compute the cost current stage*/
total =B*TotalLostAtTB()+A*LostAtmultiplexor(totalConfirmed);
unsigned short waiting=WaitingLost(totalConfirmed,states[NUMBER_OF_TOKENS]);

total= total+C*waiting;


/* seach the corresponding optimal cost from the next stage to
   the terminal stage*/
if(stage!=STAGE-1)
{
  /* compute the state of the next stage according to the
     state transition function */
  for(unsigned short l=0;l<NUMBER_OF_TOKENS;l++)
  {
```

```
    nextState[l]=states[l]+G[l]-confirmedTraffic[l];

    if (nextState[l]>TBSIZE[l])
    {
      nextState[l]=TBSIZE[l];
    }
  }

  nextState[NUMBER_OF_TOKENS]=waiting;

  unsigned short otherIndex;

  if(optimalIndex==0) otherIndex=1;
  else otherIndex=0;

  int index = searchNextV(nextState);

  /* compute the total cost from current stage to the terminal stage*/
  //calculate out the memory address first.
  unsigned short *Pt = optData->address + TABLEELEMENT(index);

  //for integer alignment reason, we have to use memory copy.
  int val;
  memcpy(&val, Pt+COST, SIZE_OF_INT);

  total = total + val;
}
```

```
  delete [] nextState;
  return total;
}

unsigned short ParaTB::confirmTraffic(unsigned short Genome[])
{
  unsigned short totalConfirmed=0;

  for (unsigned short i=0 ;i<NUMBER_OF_TOKENS;i++)
  {
    if (Traffic[TRAFFIC(stage,i)]<=(states[i]+ Genome[i]))
    {
      confirmedTraffic[i]=Traffic[TRAFFIC(stage,i)];
    }

    else confirmedTraffic[i]=0;//states[stage][i]+Genome[i];
    if (confirmedTraffic[i]>TIMEINTERVAL*LINKCAPACITY[i])
    {
      confirmedTraffic[i]=0;
    }
    totalConfirmed=totalConfirmed+confirmedTraffic[i];
  }
  return totalConfirmed;

}
```

```
/* this function used by GA */
float objective(GAGenome & c)
{
  GA1DArrayAlleleGenome<unsigned short> & genome = (GA1DArrayAlleleGenome<unsigned
short>&)c;

  struct optimizeData *optData = (struct optimizeData *)genome.userData();
  ParaTB*  objPointer = optData->objPointer;

  unsigned short *nextState = new unsigned short [objPointer->NUMBER_OF_TOKENS+1];

  for (int i=0; i<objPointer->NUMBER_OF_TOKENS+1; i++)
    nextState[i] = 0;

  float total=0;
  unsigned short *G= new unsigned short [objPointer->NUMBER_OF_TOKENS];

  for(unsigned short i=0;i<objPointer->NUMBER_OF_TOKENS;i++)
  {
    G[i]=genome.gene(i);
  }

  /* comupute the conforming traffic*/
  unsigned short totalConfirmed=objPointer->confirmTraffic(G);

  /* compute the cost of current stage */
  total =(objPointer->B)*(objPointer->TotalLostAtTB())+(objPointer->A)
```

```
*(objPointer->LostAtmultiplexor(totalConfirmed));
  unsigned short waiting=objPointer->WaitingLost(totalConfirmed,objPointer->
states[objPointer->NUMBER_OF_TOKENS]);

  total = total+objPointer->C*waiting;

  /* search the optimal total cost from next stage to the terminal stage*/
  if(objPointer->stage!=objPointer->STAGE-1)
  {
    /* computet the state of the next stage by state transition function */
    for(unsigned short l=0;l<objPointer->NUMBER_OF_TOKENS;l++)
    {
      nextState[l]=objPointer->states[l]+G[l]-objPointer->confirmedTraffic[l];
      if (nextState[l]>objPointer->TBSIZE[l])
      {
        nextState[l]=objPointer->TBSIZE[l];
      }
    }
    nextState[objPointer->NUMBER_OF_TOKENS]=waiting;
    unsigned short otherIndex;
    if(objPointer->optimalIndex==0) otherIndex=1;
    else otherIndex=0;

    int index = objPointer->searchNextV(nextState);

    /* compute the total cost from current stage to the terminal stage*/
    //calculate out the memory address first.
```

```
    unsigned short *Pt = optData->address + index *(
objPointer->SIZEOF_OF_TABLEELEMENT);

    //for integer alignment reason, we have to use memory copy.
    int val;
    memcpy(&val, Pt+COST, SIZE_OF_INT);

    total=total + val;
  }
  delete [] nextState;
  delete [] G;
  return total;
}


/* search the index of the optimal value at next stage corresponding to
   the current state of the current stage*/
int ParaTB::searchNextV(unsigned short nextState[])
{
  int index = 0;

  for (int i=0; i<NUMBER_OF_TOKENS+1; i++)
    index = index + pow_base[i] * nextState[i];

  return index;

}
```

```
int ParaTB::LostAtmultiplexor(unsigned short totalConfirmed)
{
  return
totalConfirmed-AcceptedTrafficByMul(totalConfirmed,states[NUMBER_OF_TOKENS]);
}

int ParaTB::TotalLostAtTB()
{
  int total = 0;
  for(unsigned short i=0;i<NUMBER_OF_TOKENS;i++)
  {
    total += LostAtTB(confirmedTraffic[i], Traffic[TRAFFIC(stage,i)]);
  }
  return total;
}


int ParaTB::LostAtTB(unsigned short confirmedTrafficValue, unsigned short
TrafficValue) //calling by TotalLostAtTB()
{
  return TrafficValue - confirmedTrafficValue;
}


int ParaTB::AcceptedTrafficByMul(unsigned short totalConfirmed,unsigned short
```

```
stateAtMul)

{

  //int totalT=0;

  unsigned short leftSpace=0;

  /*for(int i=0;i<Num;i++)

  {

    totalT += confirmedTraffic[i];

  }*/

  int QueueState = stateAtMul-SERVICERATE*TIMEINTERVAL;

  if(QueueState<0)

  {

    QueueState =0;

  }

  leftSpace=QUEUESIZE-QueueState;

  if(totalConfirmed>leftSpace)
```

```
    return leftSpace;
  else return totalConfirmed;
}

int ParaTB::WaitingLost(unsigned short confirmed, unsigned short stateAtMul)
{

  int QueueState = stateAtMul - SERVICERATE*TIMEINTERVAL;

  unsigned short goTraffic;

  if(QueueState >0)
  {
    QueueState=QueueState;
  }

  else
  {
    QueueState = 0;
  //  goTraffic =
AcceptedTrafficByMul(confirmed,stateAtMul);//-SERVICERATE*TIMEINTERVAL;

  }

  goTraffic=AcceptedTrafficByMul(confirmed,stateAtMul);

  return QueueState + goTraffic;//AcceptedTrafficByMul();
```

```
}

int ParaTB::controls2index(unsigned short *ptr)
{
  int index = 0;

  for (int i=0; i<NUMBER_OF_TOKENS; i++)
    index = ptr[i]*control_pow_base[i]+index;

  return index;
}

void ParaTB::index2controls(int index, unsigned short *ptr)
{
  int tmpIndex = index;
  for (int i=NUMBER_OF_TOKENS-1; i>=0; i--)
  {
    ptr[i] = (unsigned short)(tmpIndex/control_pow_base[i]);
    tmpIndex = tmpIndex - ptr[i] * control_pow_base[i];
  }

}


int ParaTB::traceBack()
{
```

```
int rankFinished = -1;
//with no information stored locally, then return
if ( rank>STAGE && localStages == 1 && indexOfCheckpoints==0) return rankFinished;


for (int i=0; i< NUMBER_OF_TOKENS+1; i++)
  states[i] = 0;

int totalOptimal;
int totallost=0;
unsigned short totalConfirmed=0;
int lostAtTB=0;
int currentStage;
unsigned short waiting=0;

if (rank==0)
{
  //begin to trace spontaneously
  for(unsigned short j=0;j<=NUMBER_OF_TOKENS;j++)
    states[j]=initialState[j];
  currentStage = 0;
}
else
{
  //get the latest states from processor rank - 1
  MPI_Status status;
```

```
    MPI_Recv(states, NUMBER_OF_TOKENS+1, MPI_UNSIGNED_SHORT,  rank-1, MPI_ANY_TAG,
MPI_COMM_WORLD, &status);

    //receive other sum data, like: totallost, totoalOptimal

    MPI_Recv(&totalOptimal, 1, MPI_INT,  rank-1, MPI_ANY_TAG, MPI_COMM_WORLD,
&status);
    MPI_Recv(&totallost, 1, MPI_INT,  rank-1, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    MPI_Recv(&lostAtTB, 1, MPI_INT,  rank-1, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    MPI_Recv(&waiting, 1, MPI_UNSIGNED_SHORT,  rank-1, MPI_ANY_TAG, MPI_COMM_WORLD,
&status);
    MPI_Recv(&totalConfirmed, 1, MPI_UNSIGNED_SHORT,  rank-1, MPI_ANY_TAG,
MPI_COMM_WORLD, &status);

    //in the case that there is some remainder
    if (rank < STAGE % size)
      currentStage = rank * localStages;
    else
      currentStage = rank * localStages + ( STAGE % size);

  }

  //means we have more processors than the stage
  if (currentStage > STAGE - 1)
  {
    delete [] states;
    return rankFinished;
```

```
}

//begin to work on trace procedure
fstream f;

if (rank==0)
  f.open("result.txt",ios::out);
else
  f.open("result.txt",ios::app);

int *stagelost = new int [NUMBER_OF_TOKENS];

int controlsValue = 0;
int oddCount = indexOfCheckpoints - 1;

/* adjust the loop count for the last processor */
if (rank == size - 1 && currentStage + localStages > STAGE - 1)
  localStages = STAGE - currentStage;

int loopCount = localStages;

int lastStage = currentStage + localStages - 1;
/* adjust the count of loop based on the distributed architecture
 * if last stage of the processor is even and not equal to stage - 1
   it should send the data to the processor with rank + 1*/
if (lastStage < STAGE - 1 && lastStage % 2 == 0)
  loopCount -= 1;
```

```
  /* if the stage number of beginning is odd, it must receive the data from
     processor rank - 1.
  */
  if (currentStage % 2 == 1 )
  {
    currentStage = currentStage - 1;
    loopCount += 1;
  }

  cout << " adjusted traceback range at PE " << rank << " is " << currentStage << "
with number " << loopCount << endl;

  for (stage=currentStage; stage<STAGE && stage<currentStage+loopCount; stage++)
  {

    unsigned short *confirmed = new unsigned short [NUMBER_OF_TOKENS];
    unsigned short totalConfirmed=0;
    int cost;
    unsigned short *currentControls = new unsigned short [NUMBER_OF_TOKENS];

    int index = searchNextV(states);

    f<<"==============At stage "<<stage<<":"<<endl;
    if (stage%2==0)
    {
      //run local GA again.
```

```
  struct optimizeData optData;
  optData.states = states;
  optData.objPointer = this;

  //if this is the last stage in whole, no checkPoints information.
  if (stage==STAGE-1)
    optData.address = NULL;
  else
    optData.address = (unsigned short *)(&(checkPoints[oddCount][0]));


  cost = optimizeOneState(&optData, currentControls);

  if (stage==0)
  {
    //get the cost from optimal cost in the cache
    totalOptimal = cost;
  }

}
else
{
  //get control values automatically.
  controlsValue = checkPoints[oddCount][index].controlsValue;
  cost = checkPoints[oddCount][index].cost;
  index2controls(controlsValue, currentControls);
  oddCount--;
```

```
}

//cout << "get control values sucessfully at rank " << rank << endl;
f<<"cost is " << cost << endl;

for(short int k=0;k<NUMBER_OF_TOKENS;k++)
{
  f<<"control:"<<currentControls[k]<<";";
}
f<<endl;

for(short int k1=0;k1<=NUMBER_OF_TOKENS;k1++)
{
  f<<"states:"<< states[k1]<<";";
}

f<<endl;

for (short int s=0 ;s<NUMBER_OF_TOKENS;s++)

{

  if (Traffic[TRAFFIC(stage,s)]<=(states[s]+ currentControls[s]))

  {
```

```
    confirmed[s]=Traffic[TRAFFIC(stage,s)];

  }

  else confirmed[s]=0;//optimalState[i][s]+ optimalControl[i][s];

  if(confirmed[s]>LINKCAPACITY[s]*TIMEINTERVAL)

  {

    confirmed[s]=0;

  }

  totalConfirmed=totalConfirmed+confirmed[s];

}



f<<"totalConfirmed :"<< totalConfirmed<<endl;

unsigned short waiting=0;
waiting=WaitingLost(totalConfirmed,states[NUMBER_OF_TOKENS]);

for(short int k2=0;k2<NUMBER_OF_TOKENS;k2++)
{
```

```
      stagelost[k2]=LostAtTB(confirmed[k2],Traffic[TRAFFIC(stage,k2)]);
      f<<"lost TB:"<<stagelost[k2]<<",";
    }

    f<<endl;

    for(short int k3=0;k3<NUMBER_OF_TOKENS;k3++)
    {
      lostAtTB=lostAtTB+stagelost[k3];
      totallost=totallost+stagelost[k3];
    }

    f<<"totallost1: "<<totallost<<endl;

    totallost=totallost+totalConfirmed-
AcceptedTrafficByMul(totalConfirmed,states[NUMBER_OF_TOKENS]);

    f<<"totallost:"<<totallost<<endl;

    if(stage==STAGE-1)
    {
      f<<"waiting : "<<waiting<<endl;
      totallost=totallost;//+waiting;
      break;
    }

    for(short int l=0;l<NUMBER_OF_TOKENS;l++)
    {
```

```
      states[l]=states[l]+currentControls[l]-confirmed[l];
      if (states[l]>TBSIZE[l])
      {
        states[l]=TBSIZE[l];
      }
    }
    states[NUMBER_OF_TOKENS]=waiting;

    delete [] currentControls;
    delete [] confirmed;

  }


  //send out the next state
  if ( ( (stage==currentStage+loopCount && stage%2==1) ||
(stage==currentStage+loopCount && stage%2==0) )
     && stage<STAGE )
  {
    //cout << "send out next states at rank " << rank << endl;

    MPI_Send(states, NUMBER_OF_TOKENS+1, MPI_UNSIGNED_SHORT, rank+1, rank,
MPI_COMM_WORLD);

    MPI_Send(&totalOptimal, 1, MPI_INT, rank+1, rank, MPI_COMM_WORLD);
    MPI_Send(&totallost, 1, MPI_INT, rank+1, rank, MPI_COMM_WORLD);
    MPI_Send(&lostAtTB, 1, MPI_INT, rank+1, rank, MPI_COMM_WORLD);
```

```
  MPI_Send(&waiting, 1, MPI_UNSIGNED_SHORT, rank+1, rank, MPI_COMM_WORLD);
  MPI_Send(&totalConfirmed, 1, MPI_UNSIGNED_SHORT, rank+1, rank, MPI_COMM_WORLD);

  delete [] states;
  //delete [] nextStates;

  return rankFinished;
}


delete [] states;
delete [] stagelost;
//delete [] nextStates;


if (stage>=STAGE-1)
{
  //print out the overall statistics.
  int totalTraffic=0;
  for(int t=0;t<STAGE;t++)
  {
    for(int t1=0;t1<NUMBER_OF_TOKENS;t1++)
      totalTraffic=totalTraffic+Traffic[TRAFFIC(t,t1)];
  }

  int waiting=totalOptimal-A*(totallost-lostAtTB)-B*lostAtTB;
  f<<"TOAL OPTIMAL VALUE ==============> "<<totalOptimal<<endl;
```

```
    f<<"None waiting losses ============> "<<totallost<<endl;
    f<<"REAL lost at TB ================> "<<lostAtTB<<endl;
    f<<"Weighted lost at TB ============> "<<B*lostAtTB<<endl;
    f<<"REAL lost at multiplexor ========> "<<(totallost-lostAtTB)<<endl;
    f<<"Weighted lost at multiplexor ====> "<<A*(totallost-lostAtTB)<<endl;
    f<<"REAL waiting lost ===============> "<<waiting/C<<endl;
    f<<"Weighted wating lost ============> "<<waiting<<endl;
    f<<"Throughtput =====================>
"<<(totalTraffic-totallost)/(SERVICERATE*TIMEINTERVAL*STAGE)<<endl;

    rankFinished = rank;
  }


  f.close();

  //finish = MPI::Wtime();
  finish = MPI_Wtime();
      elapsed_time = finish - start ;

  if (rank == rankFinished)
    cout << " Program takes " << elapsed_time << " seconds totally at rank " << rank
<< endl;

  return rankFinished;
}
```