Straightforward Method for Computer System with Variable Concurrent Programs

By

Mr. Pipat Liutanakul

Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science
in Telecommunications Science
Assumption University

May, 2003

Straightforward Method for Computer System with Variable Concurrent Programs

by

Mr. Pipat Liutanakul

Submitted in Partial Fulfillment of the Requirements for Degree of Master of Science in Telecommunication Science Assumption University

May, 2003

The Faculty of Science and Technology

Master Thesis Approval

Thesis Title	Straightforward Method for Computer System with Variable Concurrent Programs
By	Mr. Pipat Liutanakul
Thesis Advisor	Asst. Prof. Dr. Chanintorn Jittawiriyanukoon
Academic Year	3/2002

The Department of Telecommunications Science, Faculty of Science and Technology of Assumption University has approved this final report of the **twelve** credits course. **TS7000 Master Thesis**, submitted in partial fulfillment of the requirements for the degree of Master of Science in Telecommunications Science.

Approval Committee:

[illegible]
(Asst.Prof.Dr. Chanintorn J. Nukoon)
Advisor

[illegible]
(Asst.Prof.Dr. Dobri Batovski)
Committee Member

[illegible]
(Dr. Jirapun Daengdej)
Committee Member

[illegible]
(Asst.Prof.Dr. Surapong Auwatanamongkol)
Representative of Ministry of University Affairs

Faculty Approval:

[illegible]
(Asst.Prof.Dr. Dobri Batovski)
Program Director

[illegible]
(Asst.Prof. Supavadee Nontakao)
Dean

May / 2003

ACKNOWLEDGEMENTS

I would like express my most sincere appreciation and thanks to Asst.Prof.Dr. Chanintorn J. Nukoon, my master thesis advisor, for the valuable comments, encouragements, and suggestions during the development and writing of this thesis.

I would like to extend my gratitude to Asst.Prof.Dr. Pratit Santiprabhob who had given me a big chance to study for a Master's Degree in this field, MSTS. I also thank Dr. Jirapun Daengdej and Asst.Prof.Dr. Thotsapon Sortrakul who recommended solutions to any problems and cheer me up.

I am also grateful to Ms. Phikul Laemsuwanchuen, and Mr. Keattisak Chatusen for the support and help. Another person is, Ms. Pattra Sarai who has supported me in every way that she can.

Furthermore, I would like to give many thanks to Ms. Aschara Sukitjanont and Ms. Noppacha Tuansa-ard for spending their time to check, reorganize, and correct my thesis report and presentation. I am also grateful to Mr. Prateep Kurasakulwong, Ms. Sudaporn Ouppapansettee, Ms. Kuntalin Khositseth who gave me a lot of suggestions. Their cheerful and friendly nature encouraged me a lot.

There are many other friends, in my working office, who gave me valuable help especially, Mr. Teerawat Chaiyakijpichet, who gave me many suggestions and helped me to quickly understand the fundamentals and complexities of programming in C language.

Most specially, I would like to thank my family for their encouragement, for giving me support in doing everything during my MSTS study period.

Mr. Pipat Liutanakul

21 May 2003

ABSTRACT

Several methods have been proposed for evaluating the performance of parallel processing system with a set of concurrent programs such as the decomposition approximation method, which is applicable only to fixed concurrent programs. Another method, the average concurrency method, determines the average of the overall concurrency. The average concurrency method can be executed on a system with a small memory capacity, and can be executed higher speed. The average concurrency method does not suit the different of contents each vector. It is suitable for a case where is the variance of concurrent levels does not deviate much from the average concurrent level.

This thesis emphasizes on solving some problems in Multiple Level Concurrent Program (MLCP). There is a main contribution resultant from this thesis:

- Proposing an effective method for calculating MLCP

The proposed method allows the result of calculation to be close to the simulation result.

The proposed method has two main advantages. First is to provide more accurate result when compiling the calculation with MLCP. Second is this method can be applied to all concurrent programs. However, it should be noted that the main disadvantages of this method are it takes more time to calculate, and it needs more CPU power.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF EQUATIONS

CHAPTER 1

Introduction

The hardware and software are the components which can characterize many important aspects of computer systems. The computer systems trend focuses more and more on computational speed so the most satisfying alternative is parallel processing. Many programs complete for a limited number of computer resources. A group of programs will compete for and access the hardware resources (such as CPU, RAM, Buses, I/O). Therefore, computer hardware resources could not be shared sufficiently when many programs are running simultaneously. So, the notion of parallel and distributed processing is gaining increased importance. The parallel and distributed processing system is one of the latest in advanced computer technology.

1.1 Research Issues Related to Parallel Processing System

In the present computer technology, the program or software size is very large and consumes a lot of computer hardware resources such as CPU power, I/O access time, main memory, buses, and so on. To solve this problem, queuing network method is used. Although the queuing method is used, it will be inevitable to diminish this status. Delay in the queuing system resources service still occurs in the case of predicting the real system's characteristics, such as resource utilizations, queue lengths and throughputs, a representative model was proposed to show the network of interconnected queues.

1.2 Objective of the Thesis

An important factor that must be emphasized in design, development, implementation, and tuning system, is performance measurement. During the entire life cycle of a computer system [2], quantitative evaluation of computer performance is required. The queuing network model is an analytical model so the characteristics (utilizations, queue lengths, throughputs) may be found mathematically. The most suitable cost-effective tool for examining the computational speed of computer systems is the queuing network. This is an important methodology in computer performance modeling. The direction on queuing theory is influenced and driven by this application [10], [20], [21]. To investigate this technique compile to parallelism, the program needs to be partitioned into small independent programs, so a piece of the program will be executed on one or more CPUs.

1.3 Scope of Work

This document will concentrate both on the result of applying the method and the method itself as well. The scope of this document involves many terms of parallel processing with concurrent programs. Many publication papers that focus on this topic have released [1], [6], [7], 8], [9]. These publication papers [1], [10], [11], [12] introduced an approximation technique for multiprogramming computer system. Many techniques will show and evaluate the characteristics of network. For instance, *decomposition approximation* technique [13] proposed, could evaluate the characteristics of a network based upon the mean value analysis (MVA) algorithm [14]. This method is widely used but there is a limitation to the method. It can only evaluate the unchangeable concurrent level. Another technique, the *average concurrency* technique was proposed to solve the limitation of *decomposition approximation* technique but it still has a disadvantage that is the higher percentage of errors.

This document will also explain and show the main three types of concurrent program: *Different Concurrent Programs* (DCP), *Cycle-dependent Concurrent Programs* (CCP), and *Variable Concurrent Programs* (VCP). In the next chapter, it will discuss the existing methods and the restriction of the methods.

1.4 Solution to the Problems

Chapter 3 provides another solution for evaluation computer performance, the *straightforward method.* It also discusses the advantage and disadvantage of the *straightforward method.* The advantage is that if this thesis topic is proved, the result should be much more precise as it is the direct method to calculate computer performance with VCP concurrent level. It should also support evaluation computer performance with DCP and CCP concurrent level. The disadvantage will be the calculation time. It may be a little bit slower compared to the *average concurrency method.* This is due to state space number incretion and number of iteration. The computation is much longer, extending the method with changeable concurrent programs. C programming language is chosen for the implementation of the algorithm and computation of the characteristics of network (Mean Queue Length [MQL], Utilization, and Throughput) for the proposed method.

1.5 Principal Contribution

The contribution of this document is:

- To present program types which have nonproductive-form solution [2].
- To propose a more precise computational algorithm for evaluating the characteristics of the network with changeable concurrent programs.

- To obtain the performance measurement of a multiprocessing system with changeable concurrent programs by exposing the proposed method as an application tool.

In Chapter 4 this document will prove and validate the proposed method (*straightforward method*) by comparing the results obtained by the proposed method with the results both obtained by the *average concurrency method* and the *simulation method.* Comparing the number of accurate percentages to the methods will show the advantage of the proposed method. Conclusion will be presented in Chapter 5.

1.6 Abbreviations

CPU	Central Processing Unit
I/O	Input/Output
FCP	Fixed Concurrent Programs
DCP	Different Concurrent Programs
CCP	Cycle-Dependent Concurrent Programs
VCP	Variable Concurrent Programs
MLCP	Multiple Level Concurrent Programs

CHAPTER 2

Existing Performance Analysis Method

2.1 Introduction

The tools for performance prediction that are needed to support parallel and distributed processing system are cost-effective tools. For instance, one of the cost-effective tools for performance analysis of computer systems is an analytic queuing network model. At present, there are many analytic queuing network models, which can predict the performance of different computer components such as CPU power, I/O, Buses and so on. Before measuring the parallelism and minimizing the overhead of synchronization among the tasks, each task needs to be divided into a limited large number of independent tasks [1]. In order to get the number, first, assume each task requires multiple accesses to many different computer components before it communicates with the others. The performance measurement of parallel processing systems for concurrent program has been studied more and more in recent years. Heidelberger et al [14] proposed a method for evaluation parallel processing system for a fixed concurrent program, the decomposition approximation, which is based on the mean value analysis algorithm [13]. Comparing the results to simulation results proves the approximation to be accurate. However, this method is applicable only to fixed concurrent programs.

2.2. Analytic Model for Parallel Processing

This section will provide an overview model as well as some assumptions for analysis. First, assume that the program group is composed of N completely independent programs. Each is called primary task. The analytic system model is shown in Figure 2-1 [2].

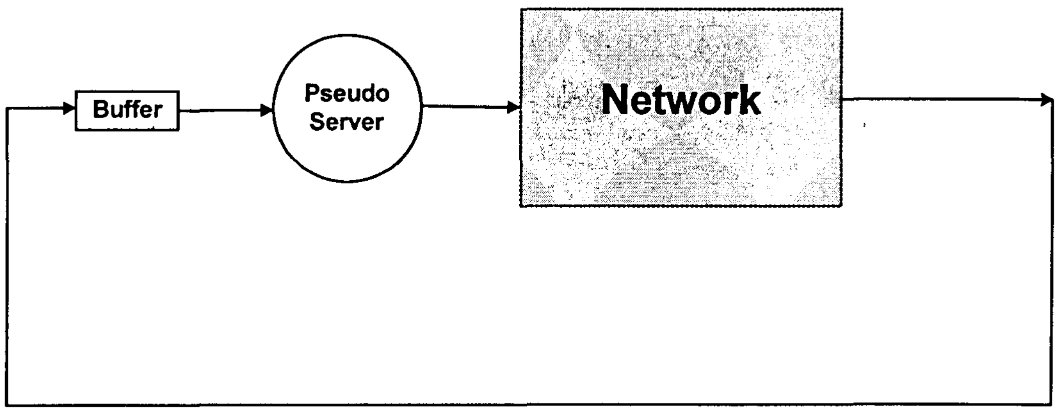


Figure 2-1 – Analytic Model for Parallel Processing

According to Figure 2-1, the model is composed of several servers (such as CPU and I/O devices) and pseudo-server. The primary task enters into the pseudo-server regardless of the service time. Then the primary task is divided into several secondary tasks, called "*sibling*". In the network, those divided secondary tasks will be executed. Assume that there is independent data among the tasks, except for the effect of the queue before entering into the server. Assume the server is visited by several tasks forming Markov Chain. When the task is completely executed by the server, it will be sent to the buffer, which is located in front of the pseudo-server. If another secondary task is not completely executed, the other tasks will wait in the buffer until the end of execution is reached. All tasks are ready in the buffer after all the secondary tasks are completely executed. Then they are combined into a primary task, and flow into the pseudo-server again. The synchronization of the secondary tasks are achieved by the above process, and the process will be repeated in the proposed system. The counting of the cycle of the process starts from the time when a primary task enters a pseudo-server to the time when it returns to the pseudo-server. When a program (with identifier i) is divided in a pseudo-server into X_i tasks, it is defined that the tasks have concurrency X_i. In general, X_i changes with each cycle. The concurrency vector for each cycle will be shown as Equation 2-1 [1].

$$\bar{C}_i = \{X_{i1}, X_{i2}, X_{i3}, \ldots, X_{iM_i}\}$$

Equation 2-1 – Concurrency Vector

2.3 Concurrent Programs

The features of the programs with varying concurrency (DCP, CCP and VCP) will be described in this section.

- Programs in which the concurrency level in each program differs (DCP)
- Programs in which the concurrency level changes cyclically (CCP)
- Programs with both characteristics of DCP and CCP (VCP)

The *decomposition approximation method* can only analyze DCP concurrency level. However, this method is not suitable for computation when the number of state spaces increases rapidly with the increasing number of programs. In addition, this traditional method cannot handle the CCP and VCP concurrency level.

The method, which can handle CCP and VCP is "*average concurrency method*". This method will reduce the time for computation, number of state spaces, and computer resources usage. However, it will present a higher percentage of errors.

First, consider all three types of concurrent programs and the states of execution.

2.3.1 Different Concurrent Programs (DCP)

The actual realistic system programs are executed in parallel. The concurrency level is fixed independently of the program, is not always true. Consider the case where the concurrency vector is different for each program and does not change with the cycle. The example of DCP is illustrated in Figure 2-2. Three programs are given. The concurrency tasks 2, 3, and 4 for the first, the second and the third program respectively, are executed. According to Figure 2-2, EXE is the state of secondary tasks existing in the network, where it is either executed by the server or waiting in front of the server, and WAIT is the state of secondary tasks waiting in the buffer of the pseudo-server for the execution of a sibling [2]".

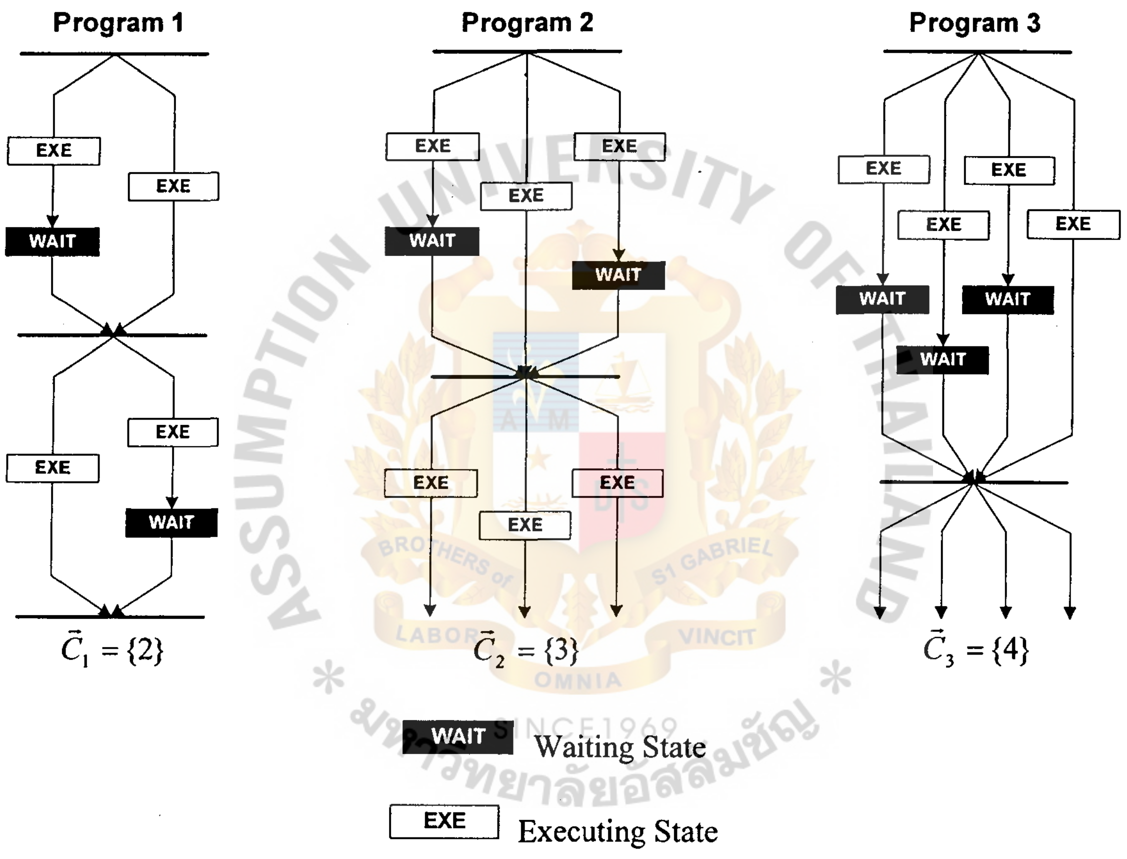
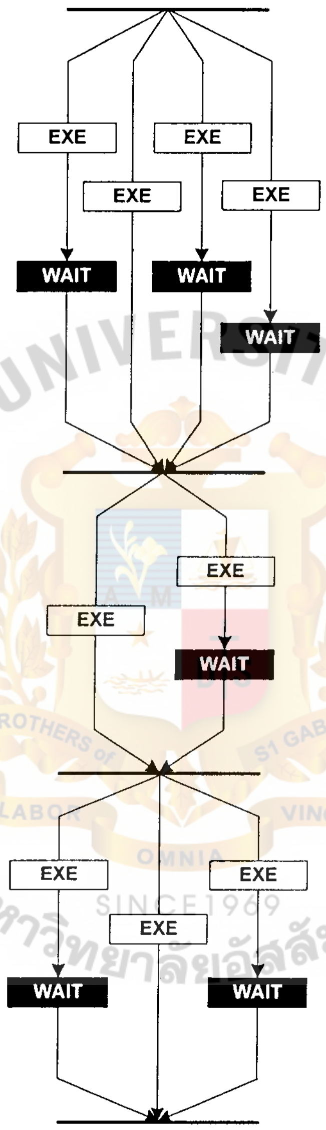


Figure 2-2 – State of Executing DCP

2.3.2 Cycle-dependent Concurrent Programs (CCP)

In this case, the concurrency vector changes with time. For instance, the concurrency vector {1,2,3} changes at the beginning of each cycle. The program shows that the concurrency vector changes as 1→2→ 3→1→2→3.

A program in which the concurrency vector will change cyclically is called CCP, but the concurrency vector of each program still remains the same. For instance, in Figure 2-3, the CCP's concurrency vector {4,2,3} is executed.



$$\vec{C}_1 = \vec{C}_2 = \vec{C}_3 = \vec{C}_4 = \{4,2,3\}$$

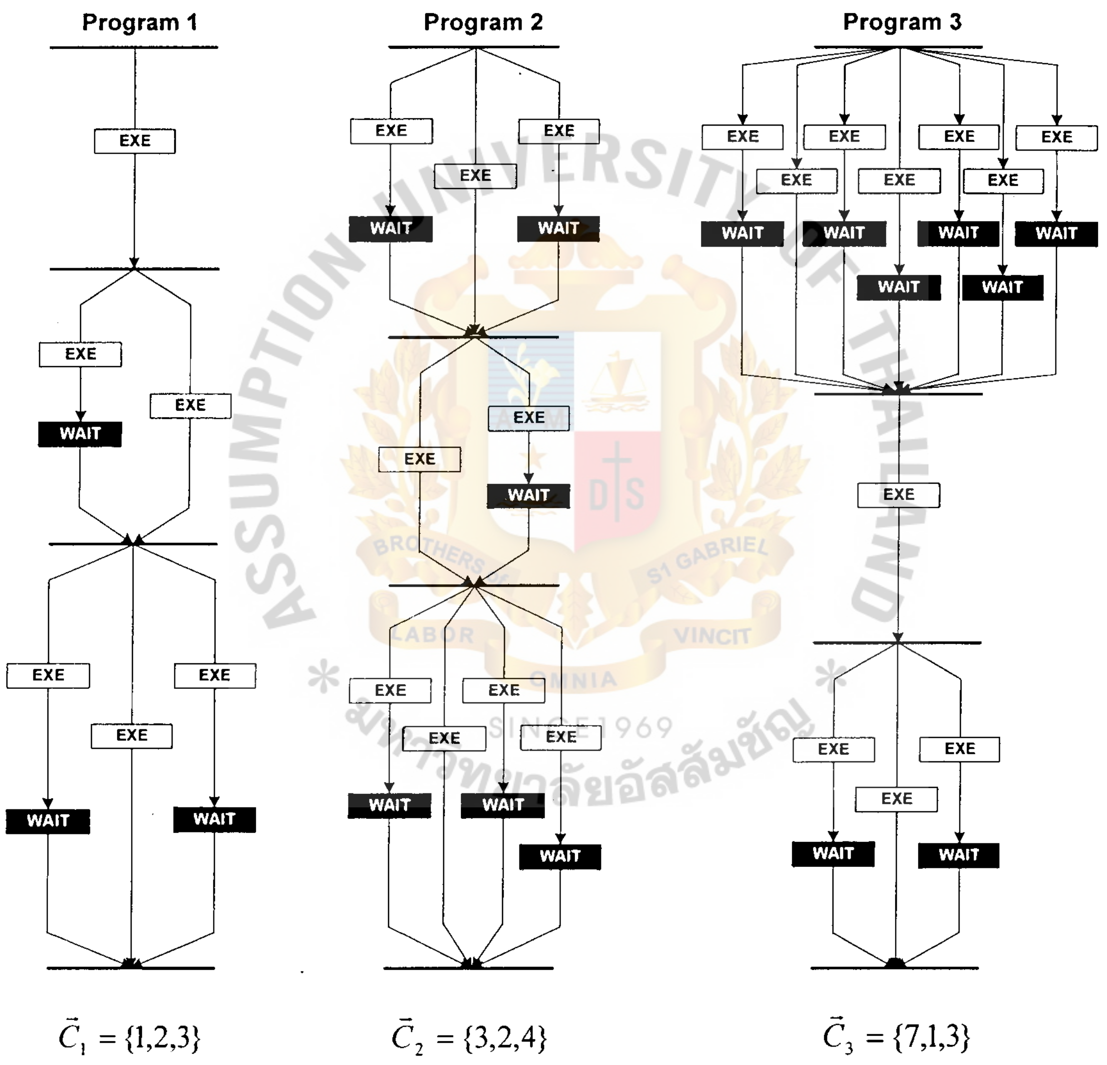
WAIT Waiting State

EXE Executing State

Figure 2-3 -- State of Executing CCP

2.3.3 Variable Concurrent Programs (VCP)

The most general case that is close to the realistic system is VCP, which is a hybrid of DCP and CCP. The concurrency vector changes with the cycle and the program. In Figure 2-4, for instance, the execution of VCP's concurrency vector {1,2,3}, {3,2,4}, and {7,1,3} for program 1, 2, and 3 respectively, is shown below.



WAIT Waiting State

EXE Executing State

Figure 2-4 -- State of Executing VCP

2.4 Existing Performance Evaluation Method

Many recent research topics are related to the measurement of the performance of parallel processing systems for concurrent programs.

2.4.1 Decomposition Approximation Method

Decomposition approximation is based upon the mean value analysis (MVA) algorithm for fixed concurrent programs [13]. It was proposed by Heidelberger et al. [14]. This method is applicable only to the programs in which the concurrency is the same for any program and is invariant in regard to time (FCP, fixed concurrent programs). The study compared the accurate percentage of the approximation result with the simulation results.

According to the traditional decomposition approximation method, it could not handle the CCP and VCP concurrent level (program group). It can only support DCP concurrent level. But the high accuracy of the method is demonstrated. This method is true only for the system which is shared by a large number of users [1].

2.4.2 Average Concurrency Method

Another method, average concurrency, which was introduced in [7], will be introduced in this section for a theoretical approximate analysis. This method is an improved technique for evaluating the characteristic parameters of the systems with Multiple Level Concurrent Programs and it focuses on the evaluation of the system performance when DCP, CCP, or VCP is executed in parallel.

This assumption allows deriving a two-step simpler calculation by decomposing the average concurrency into upper and lower bounds [1]. The lower and the upper bound sets of interesting characteristic parameters (utilization, mean queue length, and throughput) can be obtained separately in two sets. Then averaging the two sets with weight will get the final solution for the observed system.

To obtain the average concurrency of the overall programs, calculate the average concurrent level of program group $\vec{C}$ by using the integers $\vec{C}_1$ and $\vec{C}_2 (\vec{C}_2 = \vec{C}_1 + 1)$ bounding $\vec{C}$. Also, use decomposition approximation twice on two levels (lower level and upper level) bounding the average concurrent level, and the characteristic parameters (such as throughput, mean queue length, utilization rate). Then by the proportional assignment of $\vec{C}$ based on the deviation from $\vec{C}_1$, the characteristic parameters for $\vec{C}$ are computed [2]. The method is divided into 3 steps.

1. Average concurrent level calculation
2. Decomposition approximation
3. Characteristic parameters calculation

This approximate method is outlined in the following.

2.4.2.1 Average Concurrent Level

First, calculate average concurrent level ($\vec{C}$) as follows for DCP, CCP, and VCP.

DCP

$$C_i = \{X_{i1}\} \text{ for } i = 1,2,3,\ldots,N$$

$$\vec{C} = \frac{1}{N}\sum_{i=1}^{N} X_{i1}$$

CCP

$$C_i = \{X_{11}, X_{12}, X_{13}, \ldots, X_{1M_i}\} \text{ for } i = 1,2,3,\ldots,N$$

$$\vec{C} = \frac{1}{M_1}\sum_{j=1}^{M_i} X_{1j}$$

VCP

$$C_i = \{X_{i1}, X_{i2}, X_{i3}, ..., X_{iM_i}\} \text{ for } i = 1,2,3,\ldots,N$$

$$\vec{C} = \frac{\sum_{i=1}^{N}\sum_{j=1}^{M_i} X_{ij}}{\sum_{i=1}^{N} M_1}$$

Equation 2-2 – Average Concurrent Level

2.4.2.2 Decomposition Approximation

The concurrent programs deal with the approximation methods. Decomposition approximation is an approximation method. This method will be described in this section. $\vec{\mathrm{C}}$ is in general a non-integer. There are two parts of $\vec{\mathrm{C}}$; the first part is integer part *I* and the second part is fractional part *F* as shown in [1]. The concurrent level for each primary task is fixed and uses the alphabet *K* for representation. *K* secondary tasks are produced for each primary task.

$$\vec{C} = I + F = (1 - F)I + F(I + 1)$$

$$\vec{C} = (1 - \mathrm{F})\vec{\mathrm{C}}_1 + F\vec{C}_2$$

Equation 2-3 -- Decomposition Approximation

$\vec{C}_1$ represents the lower level of average concurrent level $\vec{\mathrm{C}}$

$\vec{C}_2$ represents the upper level of average concurrent level $\vec{\mathrm{C}}$

2.4.2.3 Characteristic Parameters Calculation

The characteristic parameters is computed by using $\vec{C}_1$ and $\vec{C}_2$(substituted by K in the above section). These are the characteristic parameters, $\rho_n(\vec{C}_1)$, $\rho_n(\vec{C}_2)$,

$L_n(\vec{C}_1)$, $L_n(\vec{C}_2)$, $\lambda_0(\vec{C}_1)$, and $\lambda_0(\vec{C}_2)$. They can be used for calculation to determine the $\rho_n(\vec{C})$, $L_n(\vec{C})$, $\lambda_0(\vec{C})$ for $\vec{C}$ as shown below, [1].

$$\rho_n(\vec{C}) = (1 - \mathrm{F})\rho_{\mathrm{n}}(\vec{C}_1) + (F)\rho_n(\vec{C}_2)$$

$$L_n(\vec{C}) = (1 - \mathrm{F})\mathrm{L}_{\mathrm{n}}(\vec{C}_1) + (F)L_n(\vec{C}_2)$$

$$\lambda_n(\vec{C}) = (1 - \mathrm{F})\lambda_{\mathrm{n}}(\vec{C}_1) + (F)\lambda_n(\vec{C}_2)$$

Equation 2-4 -- Calculation of Characteristic Parameters for $\vec{C}$

To get the idea of average concurrency method, consider the algorithm as shown below.

```
/**************************** ALGORITHM *************************/
BEGIN /** Computation for average concurrent level $\vec{C}$ **/
INPUT
    Parameters : $C_i$ (concurrency vector of the I-th program)
               : N (number of programs)
               : $M_i$ (number of elements in $C_i$)
IF DCP
    THEN    $\vec{C} = \frac{1}{N}\sum_{i=1}^{N} X_{i1}$
    ELSE IF CCP
        THEN    $\vec{C} = \frac{1}{M_1}\sum_{j=1}^{M_i} X_{1j}$
        ELSE IF VCP
```

```
            THEN    $\vec{C} = \frac{\sum_{i=1}^{N}\sum_{j=1}^{M_i} X_{ij}}{\sum_{i=1}^{N} M_1}$
        END IF
END IF
END /*** $\vec{C}$ is solved by these equations ***/
BEGIN /*** Initialize $C_l$ and $C_u$ ***/
    /*** I is integral part of $\vec{C}$ ***/
    /*** F is fractional part of $\vec{C}$ ***/
INITIALIZE    $C_l = I$
              $C_u = I + 1$
END
BEGIN /*** Use MVA to compute the performance for all states when concurrent
        level is $C_l$ and $C_u$ ***/
    FOR K = $C_l$ TO $C_u$ DO
        Compute all states (S) for concurrent level K
        FOR f=1 to S DO
        Compute state transition matrix $Q(\vec{A}_f)$
        Compute state probability matrix $\rho(\vec{A}_f)$
        Use MVA for computing $\rho_n(\vec{A}_f)$, $L_n(\vec{A}_f)$, $\lambda_0(\vec{A}_f)$
         END FOR
         Compute $\rho_n(K)$, $L_n(K)$, $\lambda_0(K)$
    END FOR
```

```
END
BEGIN /*** Final computation ***/
        $\rho_n(C) = (1-F)\rho_n(C_l) + (F)\rho_n(C_u)$
        $L_n(C) = (1-F)L_n(C_l) + (F)L_n(C_u)$
        $\lambda_n(C) = (1-F)\lambda_n(C_l) + (F)\lambda_n(C_u)$
END
/********************** END OF ALGORITHM ********************/
```

Figure 2-5 Algorithm of Average Concurrency Method

2.4.3 Simulation Method

The simulation is any representation or imitation of reality. It is an instructional strategy used to teach problem solving, procedures, or operations by immersing cases in situations resembling reality. The case actions can be analyzed, feedback about specific errors will provided, and performance can be scored. They provide safe environments for users to practice real-world skills. They can be especially important in situations where real errors would be too dangerous or too expensive. It is a research or teaching technique that reproduces actual events and processes under test conditions. It is an application that simulates real-world activities using mathematics or models. Real-world objects are turned into mathematical models and executing the formulas simulates their actions. EZ simulation, for instance, is a simulation tool that is used to evaluate parallel processing system performance in the computer. The other two simulations are PANACEA [18] or QNAP [19]. These simulation tools are used for analyzing the queuing network model. These simulation tools do not support analysis of the system with concurrent programs. Virtually, any objects with known characteristics can be modeled and simulated.

Simulations use enormous calculations and often require high-speed computers speed. As personal computers become more powerful, more laboratory experiments will be converted into computer models that can be interactively examined by researchers without the risk and cost of the actual experiments. The results had also been widely acceptable for high accuracy.

2.5 Restriction of Existing Method

The restriction of the existing method, decomposition approximation, is only applicable to programs which have the unchangeable set of concurrent programs. It is true only for a system which is shared by a large number of users. It provides a highly accurate method. But the other method, average concurrency method, provides higher percentage of errors compared with the results of decomposition approximation and the results of simulation method. This method can be applied to various types of changeable concurrent programs. The average concurrency method is not suitable for the contents of each vector which are different from each other (such as $\vec{C}_1 = \{2,3,4\}$, $\vec{C}_2 = \{5,15,150\}$). It is suitable for the case where the variance of concurrent levels does not deviate much from the average concurrent level (such as $\vec{C}_1 = \{2,3,4\}$, $\vec{C}_2 = \{1,2,4\}$).

CHAPTER 3

Proposed Method for Multiple Level Concurrent Programs (MLCP)

3.1 Introduction

In recent years, there have been many published papers study about the parallel and distributed processing system. These published papers present partitioning program into a number of tasks and then execute them. Queuing network models that are introduced in [15], [16], [17] are applied to use with concurrent programs. A model for programs with concurrency, which is executed on a hierarchically structured multiprocessor, has been formulated by Herzog et al. [7]. In the model, any queuing delays are insignificant.

The application of queuing theory and simulation technique by computer, for the approximate analysis, is used for investigating the bottleneck and efficiency improvement of those systems. The result of simulation is close to the realistic system but computation takes time, needs powerful CPU, and it is expensive. For these reasons, the approximate analysis is intended for study. Also PANACEA [18] or QNAP [19], for instance, are software tools used for analyzing the queuing network model but they do not support analyzing the system with concurrent programs.

3.2 Proposed Method (Straightforward Method)

However, the proposed method (straightforward method) will extend the range of the traditional decomposition approximation method that is restricted to FCP. So the straightforward method is applicable to the programs where the concurrency of a program differs or even changes with time.

The algorithm of proposed method will be discussed. It will be used to analyze the VCP concurrent level directly by separating each VCP program group into several small parts (as combination of DCP concurrent level). Then the characteristic parameters of each separated part will be calculated. After the results are obtained, the probability of each case (after extracting the VCP concurrent level to be the combinations of DCP concurrent level) will be multiplied with those results. And then they are all summarized to be the final result.

For the straightforward method, VCP will be substituted with the combinations of DCP. This method is a direct solution for solving VCP case. In N programs the concurrency vector of program i has M_i elements [3]. Then all the combinations (Z) of DCP can be computed through the equation below.

$$Z = \prod_{i=1}^{N} M_i$$

Equation 3-1 – All Combinations (Z) of DCP

That means the computation needed for Z times. Let $pr.[D = r, y \in Z]$ be the probability that the network contains r tasks when the y th combination is considered.

In this document, the proposed method will provide more flexibility for the concurrency program that is different from any programs. This method can evaluate the system, which cannot be handled by the traditional decomposition approximation. Finally, the results of several models obtained by the proposed method would be compared with the results of the simulation method. This could be indicated for realizing a sufficient accuracy.

3.3 Analytical Model for Evaluation

According to Figure 3-1, the model has a central server that is composed of six components with First-Come First-Served (FCFS) service (except component number 0). All the components are labeled from number 0-5. The component number 0 is computer or server regardless of service time. Components number 1-5 are interconnected in the network. Component number 1 is represented as the main router or gateway to the system. The service time distribution of the main router or gateway is exponential distribution with mean service time of 0.01. The components number 2-5 represent the four other routers in the network, connected with the component number 1, are shared equally by all tasks. The service time distributions of components number 2-5 are exponential distribution with mean service time of 0.04 seconds. The branching probability among these components is shown in Table 3-1.

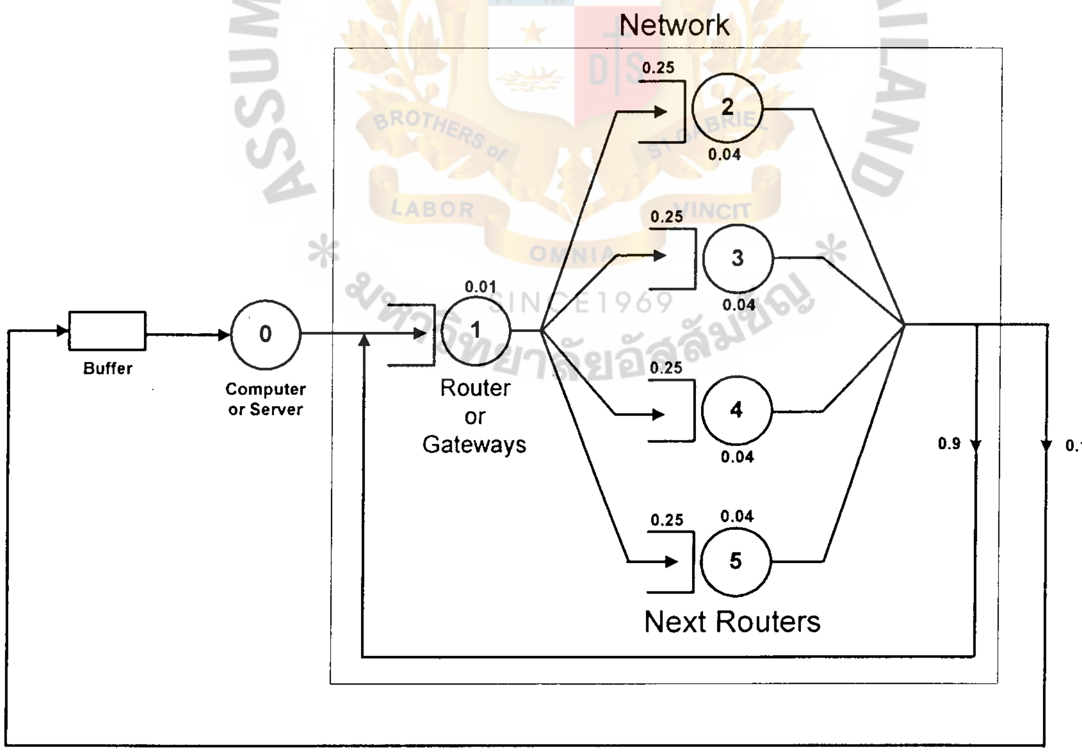


Figure 3-1 -- Closed Queuing Network

In the model, primary tasks or files that need to be sent to another computer will be entered into the computer or server and is spawned to be several independent small tasks or packets, called siblings. Then the tasks are passed in queue to component number 1, main router or gateway, for execution. After that, the executed task is passed to component number 2-5 (next routers of hopping) by equal probability sharing. The model is controlled as a closed network by assigning the probability of the complete execution task at only 0.1. And the probability of the incomplete execution task is 0.9 to reenter into component number 1 (main router or gateway). This could make the system more stable. The higher probability of the incomplete execution tasks, the more likely this model becomes closed queuing network.

Server	**Server**					
	0	1	2	3	4	5
0	0	1	0	0	0	0
1	0	0	0.25	0.25	0.25	0.25
2	0.1	0.9	0	0	0	0
3	0.1	0.9	0	0	0	0
4	0.1	0.9	0	0	0	0
5	0.1	0.9	0	0	0	0

Table 3-1 Branching Probability

Through the proposed method, the decomposition approximation's limitation will be solved. This method also produces a weak point. It requires more time for execution, and more CPU power for calculation. But at present, CPU power is not a big deal. There are many CPU models in the market that could calculate this method. So the time of execution would be reduced. After the results are obtained from the calculation, they will be entered into the table for comparison with the average concurrency method, and the simulation method.

3.4 Proposed Method's Algorithm

To get the idea of the proposed method (straightforward method), consider the algorithm as shown below.

```
/************************* ALGORITHM ***************************/
BEGIN /** Compute for straightforward concurrent level C **/
INPUT
        Parameters : $\vec{C}_i$ (concurrency vector of the i-th program)
                   : N (number of programs)
                   : $M_i$ (number of elements in $\vec{C}_i$)
        FOR 0 to ($M_i$*$M_N$) DO
           Separate the combination to be several DCP concurrent vectors
           Find the probability of each DCP vector
        END FOR
END /** C is solved by these equations **/
        /** Calculate the parameters of each DCP vector iteratively **/
        BEGIN /*** Use MVA to compute the performance for all states ***/
            Compute all states (S) for concurrent level K
            FOR f=1 to S DO
            Compute state transition matrix $Q(\vec{A}_f)$
            Compute state probability matrix $\rho(\vec{A}_f)$
            Use MVA for computing $\rho_n(\vec{A}_f)$, $L_n(\vec{A}_f)$, $\lambda_0(\vec{A}_f)$
        END FOR
        Compute $\rho_n(K)$, $L_n(K)$, $\lambda_0(K)$
    END FOR
END
BEGIN /*** Final computation ***/
            $\rho_n(C) = (1-F)\rho_n(C_l) + (F)\rho_n(C_u)$
            $L_n(C) = (1-F)L_n(C_l) + (F)L_n(C_u)$
            $\lambda_n(C) = (1-F)\lambda_n(C_l) + (F)\lambda_n(C_u)$
END
/********************** END OF ALGORITHM ********************/
```

Figure 3-2 Algorithm of Straightforward Method

First step of the algorithm is separating VCP vector into combinations of DCP vector. For instance, if the input of VCP vector is:

N=4, $\vec{C}_1 = \{1,2,3,4\}, \vec{C}_2 = \{1,2,3\}, \vec{C}_3 = \{1,1,2\}, \vec{C}_4 = \{1,2,4\}$

Then all the combinations of DCP vector will be shown in Appendix B (part 1). After getting all the combinations of DCP vector, the next step is finding the probability of summary of each DCP as will be shown in Appendix B (part 2).

The probability of content summary of each DCP, in the above, could be explained. For the probability of case, in which each DCP vector has at least one task or the content summary of each DCP vector is equal to 4. It is 2 from108 or 2/108. The probability of case, in which the content summary of each DCP vector is equal to 5, is 7 from 108 or 7/108. Another instance, the probability of the last case, in which the content summary of each DCP vector is equal to 13, is 1 from 108 or 1/108.

After getting the probability of content summary of each DCP, next step is calculating the characteristic parameters of each probability iteratively. In this case, the loop needs to start from 1 to 13. Each probability is multiplied by the results of each loop and all of the results are summarized, after multiplication, as the final result.

CHAPTER 4

Evaluation and Validation of the Proposed Method

4.1 Introduction

The validation of the proposed method (straightforward method) will be presented in this chapter. Assume the MLCP (DCP, CCP, and VCP) are executed through the analytical model for evaluation, which is introduced in Chapter 3, and is composed of CPU and several slower I/O devices. Then the characteristic parameters will be determined. First, the input parameters will be described. The results are obtained through the proposed method. The results will be compared to the results obtained through the simulation method, and the average concurrency method. The approximation accuracy of the proposed method will be evaluated. This chapter will also describe the advantage of the proposed method over the decomposition approximation in terms of applicability to all types of concurrent programs and also will point out the disadvantage of the proposed method.

4.2 Input Parameters

The proposed method will be investigated for validation by using two examples of each of CCP, and VCP.

The two examples for CCP is investigated as follows:

Case 1: N=4, $\vec{C}_1 = \vec{C}_2 = \vec{C}_3 = \vec{C}_4 = \{2,3,3,3\}$

Case 2: N=5, $\vec{C}_1 = \vec{C}_2 = \vec{C}_3 = \vec{C}_4 = \vec{C}_5 = \{2,2,2,3,3\}$

For VCP as follows:

Case 1: N=4, $\vec{C}_1 = \{1,2,3,4\}, \vec{C}_2 = \{1,2,3\}, \vec{C}_3 = \{1,1,2\}, \vec{C}_4 = \{1,2,4\}$

Case 2: N=5, $\vec{C}_1 = \{1,1,2\}, \vec{C}_2 = \{1,1,3\}, \vec{C}_3 = \{1,2,3\}, \vec{C}_4 = \{2,2,3\}, \vec{C}_5 = \{1,3,3\}$

The mean queue length at a server, server utilization and throughput at the pseudo-server will be calculated for evaluation by using the algorithm in Figure 3-2.

4.3 Reference Method for Comparison

To validate the results of the proposed method, comparison will be made between the results obtained by the proposed method with the results obtained by simulation method and average concurrency method. The result will be shown in the table for comparison as follows.

CCP				
	Server	**CASE 1**		
		STRAIGHT	**AVERAGE**	**SIMULATION**
Mean Queue Length	1	1.70999	1.71	1.79
	2	1.70999	1.71	1.69
	3	1.70999	1.71	1.71
	4	1.70999	1.71	1.65
	5	1.70999	1.71	1.73
Utilization	1	0.61999	0.62	0.61
	2	0.61999	0.62	0.61
	3	0.61999	0.62	0.62
	4	0.61999	0.62	0.60
	5	0.61999	0.62	0.62
Throughput	0	0.96999	0.97	1.02

Table 4-1 -- Comparison of Results Against Simulation (CCP CASE 1)

CCP				
	Server	CASE 2		
		STRAIGHT	AVERAGE	SIMULATION
Mean Queue Length	1	1.86	1.86	1.91
	2	1.86	1.86	1.81
	3	1.86	1.86	1.87
	4	1.86	1.86	1.91
	5	1.86	1.86	1.86
Utilization	1	0.64999	0.65	0.65
	2	0.64999	0.65	0.64
	3	0.64999	0.65	0.65
	4	0.64999	0.65	0.65
	5	0.64999	0.65	0.65
Throughput	0	0.85999	0.86	0.91

Table 4-2 -- Comparison of Results Against Simulation (CCP CASE 2)

VCP				
	Server	CASE 1		
		STRAIGHT	AVERAGE	SIMULATION
Mean Queue Length	1	1.50136	1.54	1.66
	2	1.50136	1.54	1.50
	3	1.50136	1.54	1.60
	4	1.50136	1.54	1.60
	5	1.50136	1.54	1.62
Utilization	1	0.56544	0.58	0.58
	2	0.56544	0.58	0.56
	3	0.56544	0.58	0.59
	4	0.56544	0.58	0.58
	5	0.56544	0.58	0.59
Throughput	0	1.02362	1.05	1.07

Table 4-3 -- Comparison of Results Against Simulation (VCP CASE 1)

VCP				
	Server	**CASE 1**		
		STRAIGHT	**AVERAGE**	**SIMULATION**
Mean Queue Length	1	1.706834	1.72	1.82
	2	1.706834	1.72	1.75
	3	1.706834	1.72	1.76
	4	1.706834	1.72	1.75
	5	1.706834	1.72	1.75
Utilization	1	0.615255	0.62	0.63
	2	0.615255	0.62	0.63
	3	0.615255	0.62	0.63
	4	0.615255	0.62	0.63
	5	0.615255	0.62	0.63
Throughput	0	0. 932803	0.94	0.90

Table 4-4 -- Comparison of Results Against Simulation (VCP CASE 2)

To investigate all combinations of DCP, the state numbers and probabilities [D=r, y$\in$Z] of straightforward method versus the average concurrency method are listed in the following example.

Case 1 : N=2, $\vec{C}_1 = \{2,2,3\}, \vec{C}_2 = \{1,2,3\}$

Straightforward Method						
Combinations Z=9	**State No.**	**Probabilities [D=r, y$\in$Z]**				
		r=2	**r=3**	**r=4**	**r=5**	**r=6**
$\vec{C}_1 = \{2\}$ $\vec{C}_2 = \{1\}$	3	.107	.143	-	-	-
$\vec{C}_1 = \{1\}$ $\vec{C}_2 = \{2\}$	3	.107	.143	-	-	-
$\vec{C}_1 = \{3\}$ $\vec{C}_2 = \{1\}$	7	.1	.05	.1	-	-
$\vec{C}_1 = \{2\}$ $\vec{C}_2 = \{2\}$	6	.1	.1	.05	-	-
$\vec{C}_1 = \{2\}$ $\vec{C}_2 = \{2\}$	6	.1	.1	.05	-	-
$\vec{C}_1 = \{3\}$ $\vec{C}_2 = \{2\}$	14	.068	.054	.068	.059	-
$\vec{C}_1 = \{2\}$ $\vec{C}_2 = \{3\}$	14	.068	.054	.068	.059	-
$\vec{C}_1 = \{2\}$	14	.068	.054	.068	.059	-

$\vec{C}_2 = \{3\}$						
$\vec{C}_1 = \{3\}$ $\vec{C}_2 = \{3\}$	23	.032	.019	.027	.017	.013
Total	90	.352	.330	.207	.096	.013

Average Concurrency Method						
Combinations Z=2	**State No.**	**Probabilities [D=r, y=1]**				
		r=2	**r=3**	**r=4**	**r=5**	**r=6**
$\vec{C}_1 = \{2\}$ $\vec{C}_2 = \{2\}$	6	.333	.333	.166	-	-
$\vec{C}_1 = \{3\}$ $\vec{C}_2 = \{3\}$	23	.049	.029	.041	.026	.020
Total	29	.382	.362	.208	.026	.020

Table 4-5 -- Average Concurrency Method versus Proposed Method (Case 1)

Case 2 : N=2, $\vec{C}_1 = \{1,2\}, \vec{C}_2 = \{2,3\}$

Straightforward Method						
Combinations Z=4	State No.	Probabilities [D=r, y∈Z]				
		r=2	r=3	r=4	r=5	r=6
$\vec{C}_1 = \{2\}$ $\vec{C}_2 = \{2\}$	6	.1	.1	.05	-	-
$\vec{C}_1 = \{1\}$ $\vec{C}_2 = \{2\}$	3	.107	.143	-	-	-
$\vec{C}_1 = \{1\}$ $\vec{C}_2 = \{3\}$	7	.1	.05	.1	-	-
$\vec{C}_1 = \{2\}$ $\vec{C}_2 = \{3\}$	14	.068	.054	.068	.059	-
Total	30	.375	.347	.218	.059	-

Average Concurrency Method						
Combinations Z=2	State No.	Probabilities [D=r, y=1]				
		r=2	r=3	r=4	r=5	r=6
$\vec{C}_1 = \{2\}$ $\vec{C}_2 = \{2\}$	6	.4	.4	.2	-	-
Total	6	.4	.4	.2	-	-

Table 4-6 -- Average Concurrency Method versus Proposed Method (Case 2)

4.4 Consideration for Evaluation Results

It is clear that the relative result error of mean queue lengths, server utilization and throughput are little. Using the proposed method, the maximum number of state (S_S) is increased dramatically. But using the average concurrency method, the maximum state number (S_A) decreases and explosion of the state number can be avoided. The following table will be shown for comparison of the state number and computation cost of proposed method bound against average concurrency method.

The computational cost for solving the balance equation is approximately equal to $O(S^3)$. The computational cost is definitely more expensive than that of the average concurrency method.

DCP	Straightforward Method	Average Concurrency Method
Maximum State Number	$Z = \prod_{i=1}^{N} M_i$	if C is integer $S_A=(N+1)^C$
	$S_S=Z$	if C is real $S_A=(N+2)*(N+1)^{C}{}_{1}$
Maximum Cost	$O(S_S^3)$	$O(S_A^3)$

Table 4-7 Comparison of State Number and Computation Cost Bound Against Average Concurrency Method

DCP	Number of all states	
	Average Concurrency Method	Straightforward Method
$\vec{C}_1 = \{1\}$ $\vec{C}_2 = \{2\}$ $\vec{C}_3 = \{3\}$	10	**34**
$\vec{C}_1 = \{2\}$ $\vec{C}_2 = \{3\}$ $\vec{C}_3 = \{4\}$	54	**103**
$\vec{C}_1 = \{2\}$ $\vec{C}_2 = \{2\}$ $\vec{C}_3 = \{4\}$ $\vec{C}_4 = \{4\}$	105	**845**
$\vec{C}_1 = \{2\}$ $\vec{C}_2 = \{4\}$ $\vec{C}_3 = \{4\}$ $\vec{C}_4 = \{4\}$	695	**1563**

Table 4-8 Comparison of Exact State Number Against Average Concurrency Method

CHAPTER 5

Conclusion

5.1 Introduction

Firstly, a closed queuing network model of fixed concurrent programs was introduced. The decomposition approximation, proposed by [14], allows a primary program to separate into two or three concurrent tasks. All tasks possibly sharing with various system resources are executed in parallel. It consists of a hierarchical decomposition, which can only be applicable to a system with fixed concurrent program.

In Chapter 2, three types of Multiple Level Concurrent Programs (MLCP) were introduced. It has changeable concurrent level, including DCP, CCP, and VCP. Traditional decomposition approximation can analyze DCP type of concurrent level. However, it cannot handle the CCP and VCP type of concurrent level.

So another method, average concurrency method, was introduced to solve the limitation of decomposition approximation. The first step is to calculate the average concurrent level. The next step is to employ the decomposition approximation twice to two levels bounding the average concurrent level. Then calculate the characteristic parameters. After that estimate weight to the obtained two sets of parameters. This method can evaluate the performance of computer systems with MLCP, which cannot be handled by traditional decomposition approximation. The average concurrency method can be executed on a system with a small memory capacity, and can be executed with higher speed.

In the chapter 3, the proposed method is employed for the central server model with MLCP. The proposed method's accuracy result is found by checking results against the results obtained by the more exact simulation method. Table 4-1 ~ 4-4 show clearly that in estimating the server utilization, throughput, and mean queue length, nearly all the results are obtained through the simulation method. The values obtained by the proposed method agree very favorably with the values obtained by the simulation method. It can be concluded that the proposed method validates for MLCP. Considering the maximum number of states to be calculated in DCP, the proposed method is compared with the average concurrency. The number of states in the average concurrency level increases in proportion. On the other hand, the required number of states in the proposed method increases dramatically. Since the computational cost is proportional to the cube of the number of states, there is a greater difference concerning the computational cost [2]. The proposed algorithm needs to be executed on a system with high power CPU, more memory capacity, and consumes a calculation time when compared with the average concurrency method.

In the case of comparison between proposed method and average concurrency method, the average concurrency method is not suitable for the contents in each vector which differ (such as $\vec{C}_1 = \{2,3,4\}$, $\vec{C}_2 = \{5,15,150\}$). It is suitable for a case where variance of concurrent levels does not deviate from the average concurrent level much (such as $\vec{C}_1 = \{2,3,4\}$, $\vec{C}_2 = \{1,2,4\}$). The proposed method can solve this weak point of the average concurrency method.

5.2 Future Research

In the parallel processing system, the granularity of a program is critical for speeding up the overall system performance. So the next step of work for future research is the program partitioning. Also another one that remains for future research is the dynamic task-scheduling problem. This could appear that it is a difficult problem. The only feasibility study approach is the seeking of optimal solutions through effective heuristic algorithms. The next challenging topic is the designing and analysis of the dynamic scheduling algorithm. Also the program partitioning could be the next future research. For instance, if the topic is trying to partition the incoming tasks at component number 2 and send to the next components (routers), this could be the next research topic for performance evaluation of parallel processing system.

BIBLIOGRAPHY

[1] C. Jittawiriyanukoon, "Approximate Performance Evaluation of Parallel Processing Systems Based on Reducible State Spaces Computation," *Proceedings of the Twentieth IASTED International Conference on Applied Informatics (AI 2001)*, Innsbruck, Austria, 19-22, February, 2001.

[2] C. Jittawiriyanukoon, *A Performance Evaluation Method for Parallel Processing Systems.* PhD thesis, Dept. Comm. Eng., Osaka Univ., Osaka, December, 1989.

[3] C. Jittawiriyanukoon and et al, "Performance evaluation of systems processing concurrent programs," *Systems and Computers in Japan*, pp. 21-31, 20,11,1990.

[4] C. Jittawiriyanukoon, T. Watanabe, H. Nakanishi, and Y. Tezuda, "Approximate Analytic Method for Computer System with Multiple Level Concurrent Programs," in *Proceedings of IEEE INFOCOM 89*, J. W. Mark, ed., Ottawa, Canada, pp. 82-90, April, 1989.

[5] C. Jittawiriyanukoon, T. Watanabe, H. Nakanishi, Sanada, and Y. Tezuka. "Approximate Performance Evaluation of Parallel Processing Systems Using Average Concurrency," *EIC Trans Japan*, vol. J71-D, no. 9, pp. 1623-1632, 1988.

[6] D. Towsley, K. M. Chandy, and J. C. Browne, "Models for parallel processing within programs: Applications to CPU:I/O and I/O:I/O overlap," *ACM*, vol. 21, pp. 821-831, 1978.

[7] U. Herzog, W. Hoffmann, and W. Kleinoder, "Performance modeling and evaluation for hierarchically organized multiprocessor computer systems," in *Int. Conference on Parallel Processing*, pp. 103-114, 1979.

[8] C. H. Sauer, "Approximate solution of queueing networks with simultaneous resource possession," *IBM J. Res. & Dev.*, vol. 25, pp. 894-903, 1981.

[9] P. A. Jacobson and E. D. Lazowska, "Analyzing queueing networks with simultaneous resource possession," *CACM*, vol. 25, pp. 142-151, 1981.

[10] J. P. Buzen, *Queueing network models of multiprogramming*. PhD thesis, Div. Eng. Appl. Sci., Harvard Univ., Cambridge, MA, 1971.

[11] J. C. Browne, K. M. Chandy, J. Hogarth, and C. C. A. Lee, "The effect on throughput of multiprocessing in a multiprogramming environment," *IEEE Trans. Comput.,* vol. C-22, pp. 728-735, 1973.

[12] B. Avi-Itzhak and D. P. Heyman, "Approximate queueing models for multiprogramming computer systems," *Oper. Res.,* vol. 21, pp. 1212-1230, 1973.

[13] M. Reiser and S. S. Lavenberg, "Mean-value analysis of closed multi-chain queueing networks," *ACM*, vol. 27, no. 2, pp. 313-322, 1980.

[14] P. Heidelberger and K. S. Trivedi, "Analytic queueing models for programs with internal concurrency," *IEEE Trans. Coput.,* vol. C-32, pp. 73-82, 1983.

[15] C. H. Sauer and K. M. Chandy, "The impact of distributions and disciplines on multiple processor systems," *CACM*, vol. 22, pp. 25-34, 1979.

[16] P. Heidelberger and K. S. Trivedi, "Queueing network models for parallel processing with asynchronous tasks," *IEEE Trans. Comput.,* vol. C31, pp. 1099-1109, 1982.

[17] G. S. Graham, "Queueing network models of computer system performance," *ACM Computing Surveys,* vol. 10, no. 3, pp. 219-224, 1978.

[18] K. G. Ramakrishnan and D. Mitra, "An overview of panacea: A software package for analyzing Markovian queueing networks," *The Bell Syst. Tech. Journal*, vol. 61, no. 10, pp. 2849-2872, 1982.

[19] K. M. Chandy and C. H. Sauer, “Approximate methods for analyzing queueing network models of computer systems,” *ACM Computing Surveys*, vol. 10, no. 3, pp. 218-317, 1978.

[20] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative System Performance: Computer System Analysis Using Queuing Network Models.* Englewood Cliffs, NJ: Prentice-Hall, 1984.

[21] P. Heidelberger and S. Lavenberg, “Computer performance evaluation methodology,” *IEEE Trans, Comput.*, vol. c-33, pp. 1195-1220, 1984.

APPENDIX A : SOURCE CODE

```
#include<conio.h>

#include<ctype.h>

#include<stdlib.h>

#include<string.h>

#include<stdio.h>

#include<sys\stat.h>

#include<fcntl.h>

#include<io.h>

#include<dir.h>


struct data_record

{         unsigned char record[10];

          unsigned char amount;

} all_data[10];

long progress = 0;

long all_happen=1;

long all_dcp_happen=0;

long dcp_order = 1;

long vcp_order = 1;

long c_order = 0;

char dir_name[255];

char path_name[255],tpath_name[255];


float w1,w2,n;
float prob[500];

int numberOfprob = 0;

float MQLArray[200];

float UtilArray[200];

float ThruArray[200];

int rCount = 0;

unsigned char amount_all_data;


/* Progressing process notification */

void processNotify()

{

        printf("#");

}


/* Finding summary values */

long findSumValue(char* content)

{

   int size,j,i;

   long value = 0;

   char strInt[255];

   size = strlen(content);

   j = -1;

   for(i=0; i<size; i++)

   {

        switch(content[i])
        {
         case '=' : j = 0;
                  break;
         case ',' :
                  strInt[j] = '\0';
                  value+=atoi(strInt);
                  j = -1;
                  break;
         default : if( j >=0 ) strInt[j++] = content[i];
        }
    }
    strInt[j] = '\0';
    value+=atoi(strInt);

    return value;
}
int compareDCPValue(char* compare1, char* compare2)
{
   return (findSumValue(compare1) == findSumValue(compare2) ? 1 : 0);
}
int compareVCPValue(char* compare1, char* compare2)
{
   return (findSumValue(compare1) == findSumValue(compare2) ? 1 : 0);
}
```

```
/* Convert characters to integer */

int charToInt(char d)

{

        char tmp_str[2];

        tmp_str[0]=d;

        tmp_str[1]='\0';

        return atoi(tmp_str);

}


/* Write all the DCP cases into file */

void printDCPSampleSpace(FILE *dcp, FILE *rawdcp,int order,int index,int data[])

{

        char buffer[255];

        char rawbuffer[255];

        int checkValue = 0;

        switch(index)

        {

                case 1 :

                        if( data[0] >= amount_all_data)

                        {

                                checkValue = 1;

                        }

                        break;

                case 2 :

                        if( (data[0]+data[1]) >= amount_all_data)
```

```
          {
                    checkValue = 1;
          }
          break;
case 3 :
          if( (data[0]+data[1]+data[2]) >= amount_all_data)
          {
                    checkValue = 1;
          }


          break;
case 4 :
          if((data[0]+data[1]+data[2]+data[3]) >= amount_all_data)
          {
                    checkValue = 1;
          }
          break;
case 5 :
          if((data[0]+data[1]+data[2]+data[3]+data[4]) >=
amount_all_data)
          {
                    checkValue = 1;
          }
     break;
case 6 :
```

```
                    if((data[0]+data[1]+data[2]+data[3]+data[4]+data[5]) >=
amount_all_data)
                    {
                         checkValue = 1;
                    }
                    break;
               case 7 :
                    if((data[0]+data[1]+data[2]+data[3]+data[4]+data[5]+data[6])
>= amount_all_data)
                    {
                         checkValue = 1;
                    }
                    break;
               case 8 :
          if((data[0]+data[1]+data[2]+data[3]+data[4]+data[5]+data[6]+data[7]) >=
amount_all_data)
                    {
                         checkValue = 1;
                    }
                    break;
               case 9 :
          if((data[0]+data[1]+data[2]+data[3]+data[4]+data[5]+data[6]+data[7]+data[8])
>= amount_all_data)
                    {
                         checkValue = 1;
                    }
                    break;
               case 10 :
          if((data[0]+data[1]+data[2]+data[3]+data[4]+data[5]+data[6]+data[7]+data[8]
     +data[9]) >= amount_all_data)
                    {
                         checkValue = 1;
                    }
                    break;
          }
          if(checkValue == 1)
          {
               switch(index)
               {
                    case 1 :
                         sprintf(buffer,"%ld. C1_%d=%d",
                         dcp_order++,
                         order,data[0]
                         );
                         sprintf(rawbuffer,"C1_%d=%d",
                         order,data[0]
                         );
                         break;
                    case 2 : sprintf(buffer,"%ld. C1_%d=%d,C2_%d=%d",
                         dcp_order++,
                                        order,data[0],
                                        order,data[1]
                                        );
                                        sprintf(rawbuffer,"C1_%d=%d,C2_%d=%d",
                                        order,data[0],
                                        order,data[1]
                                        );
                                        break;
                              case 3 : sprintf(buffer,"%ld.
C1_%d=%d,C2_%d=%d,C3_%d=%d",
                                        dcp_order++,
                                        order,data[0],
                                        order,data[1],
                                        order,data[2]
                                        );
sprintf(rawbuffer,"C1_%d=%d,C2_%d=%d,C3_%d=%d",
                                        order,data[0],
                                        order,data[1],
                                        order,data[2]
                                        );
                                        break;
                              case 4 : sprintf(buffer,"%ld.
C1_%d=%d,C2_%d=%d,C3_%d=%d,C4_%d=%d",
                                        dcp_order++,
                                        order,data[0],
                                        order,data[1],
                                        order,data[2],
                                        order,data[3]
                                        );
sprintf(rawbuffer,"C1_%d=%d,C2_%d=%d,C3_%d=%d,C4_%d=%d",
                                        order,data[0],
                                        order,data[1],
                                        order,data[2],
                                        order,data[3]
                                        );
                                        break;
                                case 5 : sprintf(buffer,"%ld.
C1_%d=%d,C2_%d=%d,C3_%d=%d,C4_%d=%d,C5_%d=%d",
                                        dcp_order++,
                                        order,data[0],
                                        order,data[1],
                                        order,data[2],
                                        order,data[3],
                                        order,data[4]
                                        );
sprintf(rawbuffer,"C1_%d=%d,C2_%d=%d,C3_%d=%d,C4_%d=%d,C5_%d=%d",
                                        order,data[0],
                                        order,data[1],
                                        order,data[2],
                                        order,data[3],
                    order,data[4]
                    );
                    break;
            case 6 : sprintf(buffer,"%ld.
C1_%d=%d,C2_%d=%d,C3_%d=%d,C4_%d=%d,C5_%d=%d,C6_%d=%d",
                    dcp_order++,
                    order,data[0],
                    order,data[1],
                    order,data[2],
                    order,data[3],
                    order,data[4],
                    order,data[5]
                    );
sprintf(rawbuffer,"C1_%d=%d,C2_%d=%d,C3_%d=%d,C4_%d=%d,C5_%d=%d,C6
_%d=%d",
                    order,data[0],
                    order,data[1],
                    order,data[2],
                    order,data[3],
                    order,data[4],
                    order,data[5]
                    );
                    break;
                    case 7 : sprintf(buffer,"%ld.
C1_%d=%d,C2_%d=%d,C3_%d=%d,C4_%d=%d,C5_%d=%d,C6_%d=%d,C7_%d=
%d",
                          dcp_order++,
                          order,data[0],
                          order,data[1],
                          order,data[2],
                          order,data[3],
                          order,data[4],
                          order,data[5],
                          order,data[6]
                          );
sprintf(rawbuffer,"C1_%d=%d,C2_%d=%d,C3_%d=%d,C4_%d=%d,C5_%d=%d,C6
_%d=%d,C7_%d=%d",
                          order,data[0],
                          order,data[1],
                          order,data[2],
                          order,data[3],
                          order,data[4],
                          order,data[5],
                          order,data[6]
                          );
                          break;
```

```
                case 8:  sprintf(buffer,"%ld.
C1_%d=%d,C2_%d=%d,C3_%d=%d,C4_%d=%d,C5_%d=%d,C6_%d=%d,C7_%d=
%d,C8_%d=%d",
                         dcp_order++,
                         order,data[0],
                         order,data[1],
                         order,data[2],
                         order,data[3],
                         order,data[4],
                         order,data[5],
                         order,data[6],
                         order,data[7]
                         );
sprintf(rawbuffer,"C1_%d=%d,C2_%d=%d,C3_%d=%d,C4_%d=%d,C5_%d=%d,C6
_%d=%d,C7_%d=%d,C8_%d=%d",
                         order,data[0],
                         order,data[1],
                         order,data[2],
                         order,data[3],
                         order,data[4],
                         order,data[5],
                         order,data[6],
                         order,data[7]
                         );
                         break;
```

```
                    case 9: sprintf(buffer,"%ld.
C1_%d=%d,C2_%d=%d,C3_%d=%d,C4_%d=%d,C5_%d=%d,C6_%d=%d,C7_%d=
%d,C8_%d=%d,C9_%d=%d",
                         dcp_order++,
                         order,data[0],
                         order,data[1],
                         order,data[2],
                         order,data[3],
                         order,data[4],
                         order,data[5],
                         order,data[6],
                         order,data[7],
                         order,data[8]
                         );
sprintf(rawbuffer,"C1_%d=%d,C2_%d=%d,C3_%d=%d,C4_%d=%d,C5_%d=%d,C6
_%d=%d,C7_%d=%d,C8_%d=%d,C9_%d=%d",
                         order,data[0],
                         order,data[1],
                         order,data[2],
                         order,data[3],
                         order,data[4],
                         order,data[5],
                         order,data[6],
                         order,data[7],
                         order,data[8]
                    );
                    break;
               case 10: sprintf(buffer,"%ld.
C1_%d=%d,C2_%d=%d,C3_%d=%d,C4_%d=%d,C5_%d=%d,C6_%d=%d,C7_%d=
%d,C8_%d=%d,C9_%d=%d,C10_%d=%d",
                    dcp_order++,
                    order,data[0],
                    order,data[1],
                    order,data[2],
                    order,data[3],
                    order,data[4],
                    order,data[5],
                    order,data[6],
                    order,data[7],
                    order,data[8],
                    order,data[9]
                    );
sprintf(rawbuffer,"C1_%d=%d,C2_%d=%d,C3_%d=%d,C4_%d=%d,C5_%d=%d,C6
_%d=%d,C7_%d=%d,C8_%d=%d,C9_%d=%d,C10_%d=%d",
                    order,data[0],
                    order,data[1],
                    order,data[2],
                    order,data[3],
                    order,data[4],
                    order,data[5],
                                        order,data[6],
                                        order,data[7],
                                        order,data[8],
                                        order,data[9]
                                        );
                                        break;
                              default :
                                       printf("This function does not support this index");
                    }
                    fprintf(dcp,"%s\n",buffer);
                    fprintf(rawdcp,"%s\n",rawbuffer);
                    all_dcp_happen++;
          }
}
void putAndCount(FILE *out,FILE *chk, char *odata)
{
   fprintf(out,"\n%d. ",vcp_order++);
   fprintf(out,odata);
   fprintf(chk,"\n");
   fprintf(chk,odata);
}

/* Print the result of DCP into file name DCP.TXT*/
void printDCP(int amount, int data[])
{
        int i,order=1,loop[10];

        char fileName[12];

        FILE *dcpraw,*dcp;

        sprintf(fileName,"%s//DCP.TXT",dir_name);

        if((dcp = fopen(fileName,"a+"))== NULL)

        {

                printf("Cannot open output file\\n");

        }

        sprintf(fileName,"%s//DCPTEMP.TXT",dir_name);

        if((dcpraw = fopen(fileName,"a+"))== NULL)

        {

                printf("Cannot open output file\\n");

        }

        for(loop[0] = 0; loop[0] <= data[0]; loop[0]++)

        {  if(amount == 1) printDCPSampleSpace(dcp,dcpraw,order++,amount,loop);

          for(loop[1] = 0; loop[1] <= data[1]; loop[1]++)

          {

           if(amount == 2) printDCPSampleSpace(dcp,dcpraw,order++,amount,loop);

           for(loop[2] = 0; loop[2] <= data[2]; loop[2]++)

                {

                if(amount == 3)

printDCPSampleSpace(dcp,dcpraw,order++,amount,loop);


                for(loop[3] = 0; loop[3] <= data[3]; loop[3]++)

                {
```

```
                if(amount == 4)
printDCPSampleSpace(dcp,dcpraw,order++,amount,loop);
                for(loop[4] = 0; loop[4] <= data[4]; loop[4]++)
                {
                    if(amount == 5)
printDCPSampleSpace(dcp,dcpraw,order++,amount,loop);
                    for(loop[5] = 0; loop[5] <= data[5]; loop[5]++)
                    {
                     if(amount == 6)
printDCPSampleSpace(dcp,dcpraw,order++,amount,loop);
                     for(loop[6] = 0; loop[6] <= data[6]; loop[6]++)
                     {
                       if(amount == 7)
printDCPSampleSpace(dcp,dcpraw,order++,amount,loop);
                       for(loop[7] = 0; loop[7] <= data[7]; loop[7]++)
                       {
                          if(amount == 8)
printDCPSampleSpace(dcp,dcpraw,order++,amount,loop);
                          for(loop[8] = 0; loop[8] <= data[8]; loop[8]++)
                          {
                           if(amount == 9)
printDCPSampleSpace(dcp,dcpraw,order++,amount,loop);
                           for(loop[9] = 0; loop[9] <= data[9]; loop[9]++)
                           {
```

```
                                        if(amount == 10)
printDCPSampleSpace(dcp,dcpraw,order++,amount,loop);
                              }
                            }
                        }
                      }
                    }
               }
             }
         }
        }
      }
      fprintf(dcp,"\n");
      fclose(dcp);
      fprintf(dcpraw,"\n");
      fclose(dcpraw);
}

void separateDCPDataAndPrint(char* compdata)
{
  int size,k,j,i;
  int data[10];
  char strInt[255];
  if(strstr(compdata,"END") != NULL) return;
  size = strlen(compdata);
```

```
    k = 0;
    j = -1;
    for(i=0; i < 10 ; i++)
    {
      data[i]=0;
    }
    for(i=0; i<size; i++)
    {
           switch(compdata[i])
           {
             case '=' : j = 0;
                          break;
             case ',' :
                          strInt[j] = '\0';
                          data[k++] = atoi(strInt);
                          j = -1;
                          break;
             default : if( j >=0 ) strInt[j++] = compdata[i];
           }
    }
    strInt[j] = '\0';
    data[k++] = atoi(strInt);
    printDCP(k,data);
}
```

```
/* Count the number of VCP after separate to be each DCP */
long countVCPList(FILE *cal)
{
   char cdata[255],compdata[255];
   FILE *tmp,*chk,*equallist;
   long c = 1;
   int handle,percent;
   double sample;
   sprintf(path_name,"%s%s",dir_name,"//CHK.TXT");
   if((chk = fopen(path_name,"rt"))== NULL)
   {
         printf("Cannot open output file\\n");
   }
   sprintf(path_name,"%s%s",dir_name,"//TMP.TXT");
   if((tmp = fopen(path_name,"wt"))== NULL)
   {
         printf("Cannot open output file\\n");
   }
   sprintf(path_name,"%s%s",dir_name,"//TMPCOUNT.TXT");
   if((equallist = fopen(path_name,"wt"))== NULL)
   {
         printf("Cannot open output file\\n");
   }
   if(!feof(chk)) fscanf(chk, "%s",compdata);
   if(all_happen == 1)
```

```
        fprintf(cal,"%d. %s \n",1,compdata);
    if(all_happen  == 1 || strstr(compdata,"END") != NULL)
    {
        c_order++;
        separateDCPDataAndPrint(compdata);
        fclose(chk);
        fclose(tmp);
        processNotify();
        return 0;
    }
    fprintf(equallist,"%s",compdata);
    separateDCPDataAndPrint(compdata);
    while(!feof(chk))
    {
        fscanf(chk, "%s",cdata);

        if(strstr(cdata,"END") != NULL)
        {
            fprintf(tmp,"\nEND");
            break;
        }
        else
        if(compareVCPValue(compdata,cdata))
        {
            c++;
```

```
                separateDCPDataAndPrint(cdata);
                if(strstr(compdata,cdata) == NULL)
                {
                  fprintf(equallist,"\n%s",cdata);
                }
        }
        else
        {
                fprintf(tmp,"\n%s",cdata);
        }
}
fclose(equallist);
sprintf(path_name,"%s%s",dir_name,"//TMPCOUNT.TXT");
if((equallist = fopen(path_name,"rt"))== NULL)
{
      printf("Cannot open output file\\n");
}
while(!feof(equallist))
{
      fscanf(equallist, "%s",cdata);
      c_order++;
      fprintf(cal,"%ld. %s\n",c_order,cdata);
}
fprintf(cal,"Probability = %ld/%ld\n",c,all_happen );
fclose(equallist);
```

```
    fclose(tmp);

    fclose(chk);

    processNotify();

    sprintf(path_name,"%s%s",dir_name,"//TMPCOUNT.TXT");

    remove(path_name);

    sprintf(path_name,"%s%s",dir_name,"//CHK.TXT");

    remove(path_name);

    sprintf(path_name,"%s%s",dir_name,"//TMP.TXT");

    sprintf(tpath_name,"%s%s",dir_name,"//CHK.TXT");

    rename(path_name,tpath_name);

    sprintf(path_name,"%s%s",dir_name,"//CHK.TXT");

    handle = open(path_name,S_IREAD);


    c = filelength(handle);

    close(handle);

    return c;

}


/* Count all the number of DCP cases*/

long countDCPList(FILE *cal)

{

    char cdata[255],compdata[255];

    FILE *tmp,*chk,*equallist;

    long c = 1;

    int handle,percent;
```

```
double sample;
sprintf(path_name,"%s%s",dir_name,"//DCPTEMP.TXT");
if((chk = fopen(path_name,"rt"))== NULL)
{
      printf("Cannot open output file\\n");
}
sprintf(path_name,"%s%s",dir_name,"//TMP.TXT");
if((tmp = fopen(path_name,"wt"))== NULL)
{
      printf("Cannot open output file\\n");
}
sprintf(path_name,"%s%s",dir_name,"//TMPCOUNT.TXT");
if((equallist = fopen(path_name,"wt"))== NULL)
{
      printf("Cannot open output file\\n");
}
if(!feof(chk)) fscanf(chk, "%s",compdata);
if(strstr(compdata,"END") != NULL)
{
      fclose(chk);
      fclose(tmp);
      fclose(equallist);
      return 0;
}
fprintf(equallist,"%s",compdata);
```

```
while(!feof(chk))
{
        fscanf(chk, "%s",cdata);
        if(strstr(cdata,"END") != NULL)
        {
                fprintf(tmp,"\nEND");
                break;
        }
        else
        if(compareDCPValue(compdata,cdata))
        {
                c++;
                if(strstr(compdata,cdata) == NULL)
                {
                  fprintf(equallist,"\n%s",cdata);
                }
        }
        else
        {
                fprintf(tmp,"\n%s",cdata);
        }
}
fclose(equallist);
sprintf(path_name,"%s%s",dir_name,"//TMPCOUNT.TXT");
if((equallist = fopen(path_name,"rt"))== NULL)
```

```
{
        printf("Cannot open output file\\n");
}
while(!feof(equallist))
{
        fscanf(equallist, "%s",cdata);
        fprintf(cal,"%s\n",cdata);
}
fprintf(cal,"Probability = %ld/%ld\n",c,all_dcp_happen );
fclose(equallist);
fclose(tmp);
fclose(chk);
processNotify();
sprintf(path_name,"%s%s",dir_name,"//TMPCOUNT.TXT");
remove(path_name);
sprintf(path_name,"%s%s",dir_name,"//DCPTEMP.TXT");
remove(path_name);
sprintf(path_name,"%s%s",dir_name,"//TMP.TXT");
sprintf(tpath_name,"%s%s",dir_name,"//DCPTEMP.TXT");
rename(path_name,tpath_name);
sprintf(path_name,"%s%s",dir_name,"//DCPTEMP.TXT");
handle = open(path_name,S_IREAD);

c = filelength(handle);
```

```
   close(handle);


   return c;
}


/* Print all the number of the combinations of DCP */
void printSampleSpace(unsigned char index,struct data_record data[],unsigned char
loop[], FILE *out,FILE *chk)
{
        char buffer[255];
        int percent;

        switch(index)
        {
                case 1 : sprintf(buffer,"C1=%d",
                         data[0].record[loop[0]]);
                         break;
                case 2 : sprintf(buffer,"C1=%d,C2=%d",
                         data[0].record[loop[0]],
                         data[1].record[loop[1]]);
                         break;
                case 3 : sprintf(buffer,"C1=%d,C2=%d,C3=%d",
                         data[0].record[loop[0]],
                         data[1].record[loop[1]],
                         data[2].record[loop[2]]);
```

```
                    break;
            case 4 : sprintf(buffer,"C1=%d,C2=%d,C3=%d,C4=%d",
                    data[0].record[loop[0]],
                    data[1].record[loop[1]],
                    data[2].record[loop[2]],
                    data[3].record[loop[3]]);
                    break;
            case 5 : sprintf(buffer,"C1=%d,C2=%d,C3=%d,C4=%d,C5=%d",
                    data[0].record[loop[0]],
                    data[1].record[loop[1]],
                    data[2].record[loop[2]],
                    data[3].record[loop[3]],
                    data[4].record[loop[4]]);
                    break;
            case 6 :
sprintf(buffer,"C1=%d,C2=%d,C3=%d,C4=%d,C5=%d,C6=%d",
                    data[0].record[loop[0]],
                    data[1].record[loop[1]],
                    data[2].record[loop[2]],
                    data[3].record[loop[3]],
                    data[4].record[loop[4]],
                    data[5].record[loop[5]]);
                    break;
            case 7 :
sprintf(buffer,"C1=%d,C2=%d,C3=%d,C4=%d,C5=%d,C6=%d,C7=%d",
                    data[0].record[loop[0]],
                    data[1].record[loop[1]],
                    data[2].record[loop[2]],
                    data[3].record[loop[3]],
                    data[4].record[loop[4]],
                    data[5].record[loop[5]],
                    data[6].record[loop[6]]);
                    break;
          case 8:
sprintf(buffer,"C1=%d,C2=%d,C3=%d,C4=%d,C5=%d,C6=%d,C7=%d,C8=%d",
                    data[0].record[loop[0]],
                    data[1].record[loop[1]],
                    data[2].record[loop[2]],
                    data[3].record[loop[3]],
                    data[4].record[loop[4]],
                    data[5].record[loop[5]],
                    data[6].record[loop[6]],
                    data[7].record[loop[7]]);
                    break;
          case 9:
sprintf(buffer,"C1=%d,C2=%d,C3=%d,C4=%d,C5=%d,C6=%d,C7=%d,C8=%d,C9=
%d",
                    data[0].record[loop[0]],
                    data[1].record[loop[1]],
                    data[2].record[loop[2]],
                    data[3].record[loop[3]],
                    data[4].record[loop[4]],
                    data[5].record[loop[5]],
                    data[6].record[loop[6]],
                    data[7].record[loop[7]],
                    data[8].record[loop[8]]);
                    break;
               case 10:
sprintf(buffer,"C1=%d,C2=%d,C3=%d,C4=%d,C5=%d,C6=%d,C7=%d,C8=%d,C9=
%d,C10=%d",
                    data[0].record[loop[0]],
                    data[1].record[loop[1]],
                    data[2].record[loop[2]],
                    data[3].record[loop[3]],
                    data[4].record[loop[4]],
                    data[5].record[loop[5]],
                    data[6].record[loop[6]],
                    data[7].record[loop[7]],
                    data[8].record[loop[8]],
                    data[9].record[loop[9]]);
                    break;
               default :
                    printf("This function does not support this index");
     }
     if(!(!strcmp(buffer,""))) putAndCount(out,chk,buffer);
```

```
        processNotify();
}
void strToDataRecord(char *c,struct data_record *d)
{       int len,i;
        char *tmp_data = c;
        len = strlen(c);
        d->amount = 0;
        for(i=0; i < len; i++)
        {
                if(!(*tmp_data == ','))
                {
                 d->record[d->amount] = charToInt(*tmp_data);
                 rCount++;
                 d->amount++;
                }
                tmp_data++;
        }
}

/* Print out and count all the number of possible cases of DCP into file name
DCPSummary.TXT*/
void makeDCPSummary()
{
        FILE *dcpSum;
        long file_size;
        sprintf(path_name,"%s%s",dir_name,"//DCPSummary.TXT");
        if ((dcpSum = fopen(path_name, "wt"))== NULL)
        {
          printf("Cannot open output file/n");
          exit(0);
        }
        do{
          file_size = countDCPList(dcpSum);
        }while(file_size > 0);

        fclose(dcpSum);
}

/* Print out and count all the number of possible cases of VCP into file name
VCPSummary.TXT*/
void makeVCPSummary()
{
        FILE *cal;
        long file_size;
        sprintf(path_name,"%s%s",dir_name,"//CAL.TXT");
        if((cal = fopen(path_name,"w+t"))== NULL)
        {
          printf("Cannot open output file/n");
          exit(0);
        }
        do{
                file_size = countVCPList(cal);
        }while(file_size > 0 && c_order != all_happen);
        fclose(cal);
        sprintf(path_name,"%s//DCPTEMP.TXT",dir_name);
        if((cal = fopen(path_name,"a+"))== NULL)
        {
          printf("Cannot open output file/n");
          exit(0);
        }
        fprintf(cal,"END");
        fclose(cal);
}

/* Function that will try to remove the unused files after calculation */
void manageTemporaryFile()
{
 sprintf(path_name,"%s%s",dir_name,"//CHK.TXT");
 remove(path_name);
 sprintf(path_name,"%s%s",dir_name,"//TMP.TXT");
 remove(path_name);
 sprintf(path_name,"%s%s",dir_name,"//VCP.TXT");
 remove(path_name);
 sprintf(path_name,"%s%s",dir_name,"//OUT.TXT");
 sprintf(tpath_name,"%s%s",dir_name,"//VCP.TXT");
```

```
  rename(path_name,tpath_name);

  sprintf(path_name,"%s%s",dir_name,"//VCPSUM.TXT");

  remove(path_name);

  sprintf(path_name,"%s%s",dir_name,"//CAL.TXT");

  sprintf(tpath_name,"%s%s",dir_name,"//VCPSUM.TXT");

  rename(path_name,tpath_name);

  sprintf(path_name,"%s%s",dir_name,"//TMPCOUNT.TXT");

  remove(path_name);

  sprintf(path_name,"%s%s",dir_name,"//DCPTEMP.TXT");

  remove(path_name);

}

float findW(float t,float p_array[],int numberOfArray)

{

        float value = 0;

        int i;

        if(numberOfArray > 3)

        {

                for(i = 0; i < 3; i++)

                {

                        value+=p_array[i];

                        //printf("1,%f",p_array[i]);

                }

                for(i = 3; i < numberOfArray; i++)

                {

                        value+=(float)(i-1)*p_array[i];
```

```
                    //printf("1,%f",p_array[i]);
               }
          }
          else
          {
               for(i = 0; i < 3; i++)
               {
                    value+=p_array[i];
                    //printf("2,%f",p_array[i]);
               }
          }
          return t*value;
}

/* Read the probability of each DCP case from file VCPSUM.TXT*/
void readDCPProbability(void)
{
          FILE *dcpSum;
          long file_size;
          char cdata[255];
          char *ptr_cdata;
          int pos_slash;
          numberOfprob = 0;
          sprintf(path_name,"%s%s",dir_name,"//VCPSUM.TXT");
          if ((dcpSum = fopen(path_name, "r"))== NULL)
```

```
	{
	  printf("Cannot open output file/n");
	  exit(0);
	}
	do
	{
		fscanf(dcpSum, "%s",cdata);
		if((ptr_cdata=strstr(cdata,"/")) != NULL)
		{
			pos_slash = ptr_cdata-cdata;
			cdata[pos_slash] = '\0';
			prob[numberOfprob] =
((float)atoi(cdata)/(float)atoi(ptr_cdata+1));
			numberOfprob++;
		}
	}
	while(!feof(dcpSum));
	fclose(dcpSum);
}

/* Calculate W1, W2, MQT, MQL, Util, lam (λ), Thru, n */
void calculateVcpValue(FILE *out, int r, float W1, float W2)
{
	float MQT, MQL, Util, lam, thru;
	MQT = W1 - 0.01;
```

```
        MQL = MQT*1/0.01;

        MQLArray[r-1] = MQL;

        Util = W1 * r/ 20 * 100;

        UtilArray[r-1] = Util;

        lam = 1/(W1+W2);

        thru = lam * 0.1;

        ThruArray[r-1] = thru;

        n = 1+(20 * 0.9 * (W1 + W2));

        fprintf(out,

        "R=%d, W1=%.2f, W2=%.2f, MQT=%.2f, MQL=%.2f, Util=.%.2f,
Lamda=%.2f, Thru=%.2f, n=%.2f \n"

        ,r,W1,W2,MQT,MQL,Util,lam,thru,n);

}

/* End Calculate W1, W2, MQT, MQL, Util, lam (λ), Thru, n */


/* The steps for making result summary */

void makeResultSummary(void)

{

        FILE *cal;

        int i;

        n = 1;

        sprintf(path_name,"%s%s",dir_name,"//CalResult.TXT");

        if((cal = fopen(path_name,"w+t"))== NULL)

        {

          printf("Cannot open output file/n");
          exit(0);
        }
        for(i = 1; i <= rCount; i++)
        {
                calculateVcpValue(cal,i,w1*n,w2*n);
        }
        fclose(cal);
}
/* End the steps for making result summary */


/* Calculate MQT and Util and Thru and write them into file name
MeanSummary.TXT */
void makeResultMQTAndUtilAndThru()
{
        FILE *cal;
        int i;
        sprintf(path_name,"%s%s",dir_name,"//MeanSummary.TXT");
        if((cal = fopen(path_name,"w+t"))== NULL)
        {
          printf("Cannot open output file/n");
          exit(0);
        }
        float MQLResult=0,UtilResult=0,ThruResult=0;
        for(i = 0; i < rCount; i++)
        {
                MQLResult += MQLArray[i]*prob[i];

        }

        for(i = 0; i < rCount; i++)

        {

                UtilResult += UtilArray[i]*prob[i];

        }

        for(i = 0; i < rCount; i++)

        {

                ThruResult += ThruArray[i]*prob[i];

        }

        fprintf(cal,"MQL = %.2f\n",MQLResult);

        fprintf(cal,"Utilization = %.2f\n",UtilResult);

        fprintf(cal,"Throughput = %.2f",ThruResult);

        fclose(cal);

}

/* End Calculate MQT and Util and Thru and write them into file name

MeanSummary.TXT */


int main(void)

{

        unsigned char i,j,loop[10];

        char tmp_char;

        char tmp_data[255];

        long file_size;

        FILE *out,*chk;
```

```
clrscr();
printf("Enter Directory Name : ");
scanf("%s",dir_name);
mkdir(dir_name);
sprintf(path_name,"%s%s",dir_name,"//OUT.TXT");
if ((out = fopen(path_name, "wt"))
== NULL)
{
  printf("Cannot open output file/n");
  return 1;
}
sprintf(path_name,"%s%s",dir_name,"//CHK.TXT");
if((chk = fopen(path_name,"wt"))== NULL)
{
  printf("Cannot open output file/n");
  return 1;
}
sprintf(path_name,"%s%s",dir_name,"//DCPTEMP.TXT");
remove(path_name);
sprintf(path_name,"%s%s",dir_name,"//DCP.TXT");
remove(path_name);
fprintf(out,"<-----BeginOfResult---->");
do
{
        printf("Please insert the number of program(s) : ");
        tmp_char = getche();
        printf("\n");
        if(!isdigit(tmp_char)) printf("\nPlease enter only 1-10\n");
    }while(!isdigit(tmp_char));
    amount_all_data=charToInt(tmp_char);
    printf("Enter the data and seperate each number by comma (,). Do not type the
space!!!\n");
    for(i=0; i <amount_all_data; i++)
    {
        printf("The number of concurrent vector #%d = ",i+1);
        scanf("%s",tmp_data);
        strToDataRecord(tmp_data,&all_data[i]);
    }
    printf("Enter W1 = ");
    tmp_data[0]='\0';

    scanf("%s",tmp_data);
    w1 = atof(tmp_data);
    printf("Enter W2 = ");
    tmp_data[0]='\0';
    scanf("%s",tmp_data);
    w2 = atof(tmp_data);
    for(i=0; i <amount_all_data; i++)
    {
        fprintf(out,"\nThe concurrent %d = ",i);
        for(j=0; j<all_data[i].amount; j++)

        {

             fprintf(out,"%d ",all_data[i].record[j]);

        }

        all_happen *= all_data[i].amount;

}

fprintf(out,"\n");

for(loop[0] = 0; loop[0] < all_data[0].amount; loop[0]++)

{

 if(amount_all_data == 1) printSampleSpace(1,all_data,loop,out,chk);

 for(loop[1] = 0; loop[1] < all_data[1].amount; loop[1]++)

 {

  if(amount_all_data == 2) printSampleSpace(2,all_data,loop,out,chk);

  for(loop[2] = 0; loop[2] < all_data[2].amount; loop[2]++)

  {

   if(amount_all_data == 3) printSampleSpace(3,all_data,loop,out,chk);

   for(loop[3] = 0; loop[3] < all_data[3].amount; loop[3]++)

   {

       if(amount_all_data == 4) printSampleSpace(4,all_data,loop,out,chk);

       for(loop[4] = 0; loop[4] < all_data[4].amount; loop[4]++)

       {

        if(amount_all_data == 5) printSampleSpace(5,all_data,loop,out,chk);

        for(loop[5] = 0; loop[5] < all_data[5].amount; loop[5]++)

        {
               if(amount_all_data == 6)
printSampleSpace(6,all_data,loop,out,chk);
               for(loop[6] = 0; loop[6] < all_data[6].amount; loop[6]++)
               {
                if(amount_all_data == 7)
printSampleSpace(7,all_data,loop,out,chk);
                for(loop[7] = 0; loop[7] < all_data[7].amount; loop[7]++)
                {
                  if(amount_all_data == 8)
printSampleSpace(8,all_data,loop,out,chk);
                  for(loop[8] = 0; loop[8] < all_data[8].amount; loop[8]++)
                  {
                   if(amount_all_data == 9)
printSampleSpace(9,all_data,loop,out,chk);
                   for(loop[9] = 0; loop[9] < all_data[9].amount; loop[9]++)
                   {
                      if(amount_all_data == 10)
printSampleSpace(10,all_data,loop,out,chk);
                   }
                  }
                }
               }
              }
             }
           }
```

```
            }
           }
          }
    fprintf(out,"\n<-----EndOfResult---->");
    fprintf(chk,"\nEND");
    fclose(out);
    fclose(chk);
    makeVCPSummary();
    makeDCPSummary();
    manageTemporaryFile();
    readDCPProbability();
    makeResultSummary();
    makeResultMQTAndUtilAndThru();
    printf("\nCompleted.");
    printf("\nPress enter to continue...");
    getch();
    return 0;
}
```

APPENDIX B : RESULT FILE

Part I

The concurrent vector after user enters the value follows VCP case 1 as shown in Page 26. The result would be as below.

1 The concurrent vector 1 (C_1) = {1, 2, 3, 4}

2 The concurrent vector 2 (C_2) = {1, 2, 3}

3 The concurrent vector 3 (C_3) = {1, 1, 2}

4 The concurrent vector 4 (C_4) = {1, 2, 4}

All cases (108 cases) after the combination of each concurrent vectors 1-4 are listed:

1. C_1=1, C_2=1, C_3=1, C_4=1
2. C_1=1, C_2=1, C_3=1, C_4=2
3. C_1=1, C_2=1, C_3=1, C_4=4
4. C_1=1, C_2=1, C_3=1, C_4=1
5. C_1=1, C_2=1, C_3=1, C_4=2
6. C_1=1, C_2=1, C_3=1, C_4=4
7. C_1=1, C_2=1, C_3=2, C_4=1
8. C_1=1, C_2=1, C_3=2, C_4=2
9. C_1=1, C_2=1, C_3=2, C_4=4
10. C_1=1, C_2=2, C_3=1, C_4=1
11. C_1=1, C_2=2, C_3=1, C_4=2
12. C_1=1, C_2=2, C_3=1, C_4=4
13. C_1=1, C_2=2, C_3=1, C_4=1
14. C_1=1, C_2=2, C_3=1, C_4=2

15. C_1=1, C_2=2, C_3=1, C_4=4

16. C_1=1, C_2=2, C_3=2, C_4=1

17. C_1=1, C_2=2, C_3=2, C_4=2

18. C_1=1, C_2=2, C_3=2, C_4=4

19. C_1=1, C_2=3, C_3=1, C_4=1

20. C_1=1, C_2=3, C_3=1, C_4=2

21. C_1=1, C_2=3, C_3=1, C_4=4

22. C_1=1, C_2=3, C_3=1, C_4=1

23. C_1=1, C_2=3, C_3=1, C_4=2

24. C_1=1, C_2=3, C_3=1, C_4=4

25. C_1=1, C_2=3, C_3=2, C_4=1

26. C_1=1, C_2=3, C_3=2, C_4=2

27. C_1=1, C_2=3, C_3=2, C_4=4

28. C_1=2, C_2=1, C_3=1, C_4=1

29. C_1=2, C_2=1, C_3=1, C_4=2

30. C_1=2, C_2=1, C_3=1, C_4=4

31. C_1=2, C_2=1, C_3=1, C_4=1

32. C_1=2, C_2=1, C_3=1, C_4=2

33. C_1=2, C_2=1, C_3=1, C_4=4

34. C_1=2, C_2=1, C_3=2, C_4=1

35. C_1=2, C_2=1, C_3=2, C_4=2

36. C_1=2, C_2=1, C_3=2, C_4=4

37. C_1=2, C_2=2, C_3=1, C_4=1

38. C_1=2, C_2=2, C_3=1, C_4=2

39. C_1=2, C_2=2, C_3=1, C_4=4

40. C_1=2, C_2=2, C_3=1, C_4=1

41. C_1=2, C_2=2, C_3=1, C_4=2

42. C_1=2, C_2=2, C_3=1, C_4=4

43. C_1=2, C_2=2, C_3=2, C_4=1

44. C_1=2, C_2=2, C_3=2, C_4=2

45. C_1=2, C_2=2, C_3=2, C_4=4

46. C_1=2, C_2=3, C_3=1, C_4=1

47. C_1=2, C_2=3, C_3=1, C_4=2

48. C_1=2, C_2=3, C_3=1, C_4=4

49. C_1=2, C_2=3, C_3=1, C_4=1

50. C_1=2, C_2=3, C_3=1, C_4=2

51. C_1=2, C_2=3, C_3=1, C_4=4

52. C_1=2, C_2=3, C_3=2, C_4=1

53. C_1=2, C_2=3, C_3=2, C_4=2

54. C_1=2, C_2=3, C_3=2, C_4=4

55. C_1=3, C_2=1, C_3=1, C_4=1

56. C_1=3, C_2=1, C_3=1, C_4=2

57. C_1=3, C_2=1, C_3=1, C_4=4

58. C_1=3, C_2=1, C_3=1, C_4=1

59. C_1=3, C_2=1, C_3=1, C_4=2

60. C_1=3, C_2=1, C_3=1, C_4=4

61. C_1=3, C_2=1, C3=2, C_4=1

62. C_1=3, C_2=1, C_3=2, C_4=2

63. C_1=3, C_2=1, C_3=2, C_4=4

64. C_1=3, C_2=2, C_3=1, C_4=1

65. $C_1=3, C_2=2, C_3=1, C_4=2$

66. $C_1=3, C_2=2, C_3=1, C_4=4$

67. $C_1=3, C_2=2, C_3=1, C_4=1$

68. $C_1=3, C_2=2, C_3=1, C_4=2$

69. $C_1=3, C_2=2, C_3=1, C_4=4$

70. $C_1=3, C_2=2, C_3=2, C_4=1$

71. $C_1=3, C_2=2, C_3=2, C_4=2$

72. $C_1=3, C_2=2, C_3=2, C_4=4$

73. $C_1=3, C_2=3, C_3=1, C_4=1$

74. $C_1=3, C_2=3, C_3=1, C_4=2$

75. $C_1=3, C_2=3, C_3=1, C_4=4$

76. $C_1=3, C_2=3, C_3=1, C_4=1$

77. $C_1=3, C_2=3, C_3=1, C_4=2$

78. $C_1=3, C_2=3, C_3=1, C_4=4$

79. $C_1=3, C_2=3, C_3=2, C_4=1$

80. $C_1=3, C_2=3, C_3=2, C_4=2$

81. $C_1=3, C_2=3, C_3=2, C_4=4$

82. $C_1=4, C_2=1, C_3=1, C_4=1$

83. $C_1=4, C_2=1, C_3=1, C_4=2$

84. $C_1=4, C_2=1, C_3=1, C_4=4$

85. $C_1=4, C_2=1, C_3=1, C_4=1$

86. $C_1=4, C_2=1, C_3=1, C_4=2$

87. $C_1=4, C_2=1, C_3=1, C_4=4$

88. $C_1=4, C_2=1, C_3=2, C_4=1$

89. $C_1=4, C_2=1, C_3=2, C_4=2$

90. C_1=4, C_2=1, C_3=2, C_4=4

91. C_1=4, C_2=2, C_3=1, C_4=1

92. C_1=4, C_2=2, C_3=1, C_4=2

93. C_1=4, C_2=2, C_3=1, C_4=4

94. C_1=4, C_2=2, C_3=1, C_4=1

95. C_1=4, C_2=2, C_3=1, C_4=2

96. C_1=4, C_2=2, C_3=1, C_4=4

97. C_1=4, C_2=2, C_3=2, C_4=1

98. C_1=4, C_2=2, C_3=2, C_4=2

99. C_1=4, C_2=2, C_3=2, C_4=4

100. C_1=4, C_2=3, C_3=1, C_4=1

101. C_1=4, C_2=3, C_3=1, C_4=2

102. C_1=4, C_2=3, C_3=1, C_4=4

103. C_1=4, C_2=3, C_3=1, C_4=1

104. C_1=4, C_2=3, C_3=1, C_4=2

105. C_1=4, C_2=3, C_3=1, C_4=4

106. C_1=4, C_2=3, C_3=2, C_4=1

107. C_1=4, C_2=3, C_3=2, C_4=2

108. C_1=4, C_2=3, C_3=2, C_4=4

Part II

The probability of the case that the summarizes of all $C_{(1\text{-}4)}$ [$\sum_{i=1}^{N} C_i$] is equal to 4

1. $C_1=1, C_2=1, C_3=1, C_4=1$

Probability = 2/108 = 0.0185185

The probability of the case that the summarizes of all $C_{(1\text{-}4)}$ [$\sum_{i=1}^{N} C_i$] is equal to 5

2. $C_1=1, C_2=1, C_3=1, C_4=2$

3. $C_1=1, C_2=1, C_3=2, C_4=1$

4. $C_1=1, C_2=2, C_3=1, C_4=1$

5. $C_1=1, C_2=2, C_3=1, C_4=1$

6. $C_1=2, C_2=1, C_3=1, C_4=1$

7. $C_1=2, C_2=1, C_3=1, C_4=1$

Probability = 7/108 = 0.0648148

The probability of the case that the summarizes of all $C_{(1\text{-}4)}$ [$\sum_{i=1}^{N} C_i$] is equal to 6

8. $C_1=1, C_2=1, C_3=2, C_4=2$

9. $C_1=1, C_2=2, C_3=1, C_4=2$

10. $C_1=1, C_2=2, C_3=1, C_4=2$

11. $C_1=1, C_2=2, C_3=2, C_4=1$

12. $C_1=1, C_2=3, C_3=1, C_4=1$

13. $C_1=1, C_2=3, C_3=1, C_4=1$

14. $C_1=2, C_2=1, C_3=1, C_4=2$

15. $C_1=2, C_2=1, C_3=1, C_4=2$

16. C_1=2, C_2=1, C_3=2, C_4=1

17. C_1=2, C_2=2, C_3=1, C_4=1

18. C_1=2, C_2=2, C_3=1, C_4=1

19. C_1=3, C_2=1, C_3=1, C_4=1

20. C_1=3, C_2=1, C_3=1, C_4=1

Probability = 13/108 = 0.1203704

The probability of the case that the summarizes of all $C_{(1\text{-}4)}$ [$\sum_{i=1}^{N} C_i$] is equal to 7

21. C_1=1, C_2=1, C_3=1, C_4=4

22. C_1=1, C_2=2, C_3=2, C_4=2

23. C_1=1, C_2=3, C_3=1, C_4=2

24. C_1=1, C_2=3, C_3=1, C_4=2

25. C_1=1, C_2=3, C_3=2, C_4=1

26. C_1=2, C_2=1, C_3=2, C_4=2

27. C_1=2, C_2=2, C_3=1, C_4=2

28. C_1=2, C_2=2, C_3=1, C_4=2

29. C_1=2, C_2=2, C_3=2, C_4=1

30. C_1=2, C_2=3, C_3=1, C_4=1

31. C_1=2, C_2=3, C_3=1, C_4=1

32. C_1=3, C_2=1, C_3=1, C_4=2

33. C_1=3, C_2=1, C_3=1, C_4=2

34. C_1=3, C_2=1, C_3=2, C_4=1

35. C_1=3, C_2=2, C_3=1, C_4=1

36. C_1=3, C_2=2, C_3=1, C_4=1

37. C_1=4, C_2=1, C_3=1, C_4=1

38. C_1=4, C_2=1, C_3=1, C_4=1

Probability = 19/108 = 0.1759259

The probability of the case that the summarizes of all $C_{(1-4)}$ [$\sum_{i=1}^{N} C_i$] is equal to 8

39. C_1=1, C_2=1, C_3=2, C_4=4

40. C_1=1, C_2=2, C_3=1, C_4=4

41. C_1=1, C_2=2, C_3=1, C_4=4

42. C_1=1, C_2=3, C_3=2, C_4=2

43. C_1=2, C_2=1, C_3=1, C_4=4

44. C_1=2, C_2=1, C_3=1, C_4=4

45. C_1=2, C_2=2, C_3=2, C_4=2

46. C_1=2, C_2=3, C_3=1, C_4=2

47. C_1=2, C_2=3, C_3=1, C_4=2

48. C_1=2, C_2=3, C_3=2, C_4=1

49. C_1=3, C_2=1, C_3=2, C_4=2

50. C_1=3, C_2=2, C_3=1, C_4=2

51. C_1=3, C_2=2, C_3=1, C_4=2

52. C_1=3, C_2=2, C_3=2, C_4=1

53. C_1=3, C_2=3, C_3=1, C_4=1

54. C_1=3, C_2=3, C_3=1, C_4=1

55. C_1=4, C_2=1, C_3=1, C_4=2

56. C_1=4, C_2=1, C_3=1, C_4=2

57. C_1=4, C_2=1, C_3=2, C_4=1

58. C_1=4, C_2=2, C_3=1, C_4=1

59. C_1=4, C_2=2, C_3=1, C_4=1

Probability = 21/108 = 0.1944444

The probability of the case that the summarizes of all $C_{(1\text{-}4)}$ [$\sum_{i=1}^{N} C_i$] is equal to 9

60. $C_1=1, C_2=2, C_3=2, C_4=4$

61. $C_1=1, C_2=3, C_3=1, C_4=4$

62. $C_1=1, C_2=3, C_3=1, C_4=4$

63. $C_1=2, C_2=1, C_3=2, C_4=4$

64. $C_1=2, C_2=2, C_3=1, C_4=4$

65. $C_1=2, C_2=2, C_3=1, C_4=4$

66. $C_1=2, C_2=3, C_3=2, C_4=2$

67. $C_1=3, C_2=1, C_3=1, C_4=4$

68. $C_1=3, C_2=1, C_3=1, C_4=4$

69. $C_1=3, C_2=2, C_3=2, C_4=2$

70. $C_1=3, C_2=3, C_3=1, C_4=2$

71. $C_1=3, C_2=3, C_3=1, C_4=2$

72. $C_1=3, C_2=3, C_3=2, C_4=1$

73. $C_1=4, C_2=1, C_3=2, C_4=2$

74. $C_1=4, C_2=2, C_3=1, C_4=2$

75. $C_1=4, C_2=2, C_3=1, C_4=2$

76. $C_1=4, C_2=2, C_3=2, C_4=1$

77. $C_1=4, C_2=3, C_3=1, C_4=1$

78. $C_1=4, C_2=3, C_3=1, C_4=1$

Probability = 19/108 = 0.1759259

The probability of the case that the summarizes of all $C_{(1-4)}$ [$\sum_{i=1}^{N} C_i$] is equal to 10

79. $C_1=1, C_2=3, C_3=2, C_4=4$

80. $C_1=2, C_2=2, C_3=2, C_4=4$

81. $C_1=2, C_2=3, C_3=1, C_4=4$

82. $C_1=2, C_2=3, C_3=1, C_4=4$

83. $C_1=3, C_2=1, C_3=2, C_4=4$

84. $C_1=3, C_2=2, C_3=1, C_4=4$

85. $C_1=3, C_2=2, C_3=1, C_4=4$

86. $C_1=3, C_2=3, C_3=2, C_4=2$

87. $C_1=4, C_2=1, C_3=1, C_4=4$

88. $C_1=4, C_2=1, C_3=1, C_4=4$

89. $C_1=4, C_2=2, C_3=2, C_4=2$

90. $C_1=4, C_2=3, C_3=1, C_4=2$

91. $C_1=4, C_2=3, C_3=1, C_4=2$

92. $C_1=4, C_2=3, C_3=2, C_4=1$

Probability = 14/108 = 0.1296296

The probability of the case that the summarizes of all $C_{(1-4)}$ [$\sum_{i=1}^{N} C_i$] is equal to 11

93. $C_1=2, C_2=3, C_3=2, C_4=4$

94. $C_1=3, C_2=2, C_3=2, C_4=4$

95. $C_1=3, C_2=3, C_3=1, C_4=4$

96. $C_1=3, C_2=3, C_3=1, C_4=4$

97. $C_1=4, C_2=1, C_3=2, C_4=4$

98. $C_1=4, C_2=2, C_3=1, C_4=4$

99. C_1=4, C_2=2, C_3=1, C_4=4

100. C_1=4, C_2=3, C_3=2, C_4=2

Probability = 8/108 = 0.0740741

The probability of the case that the summarizes of all $C_{(1\text{-}4)}$ [$\sum_{i=1}^{N} C_i$] is equal to 12

101. C_1=3, C_2=3, C_3=2, C_4=4

102. C_1=4, C_2=2, C_3=2, C_4=4

103. C_1=4, C_2=3, C_3=1, C_4=4

104. C_1=4, C_2=3, C_3=1, C_4=4

Probability = 4/108 = 0.0370370

The probability of the case that the summarizes of all $C_{(1\text{-}4)}$ [$\sum_{i=1}^{N} C_i$] is equal to 13

105. C_1=4, C_2=3, C_3=2, C_4=4

Probability = 1/108 = 0.0092593