

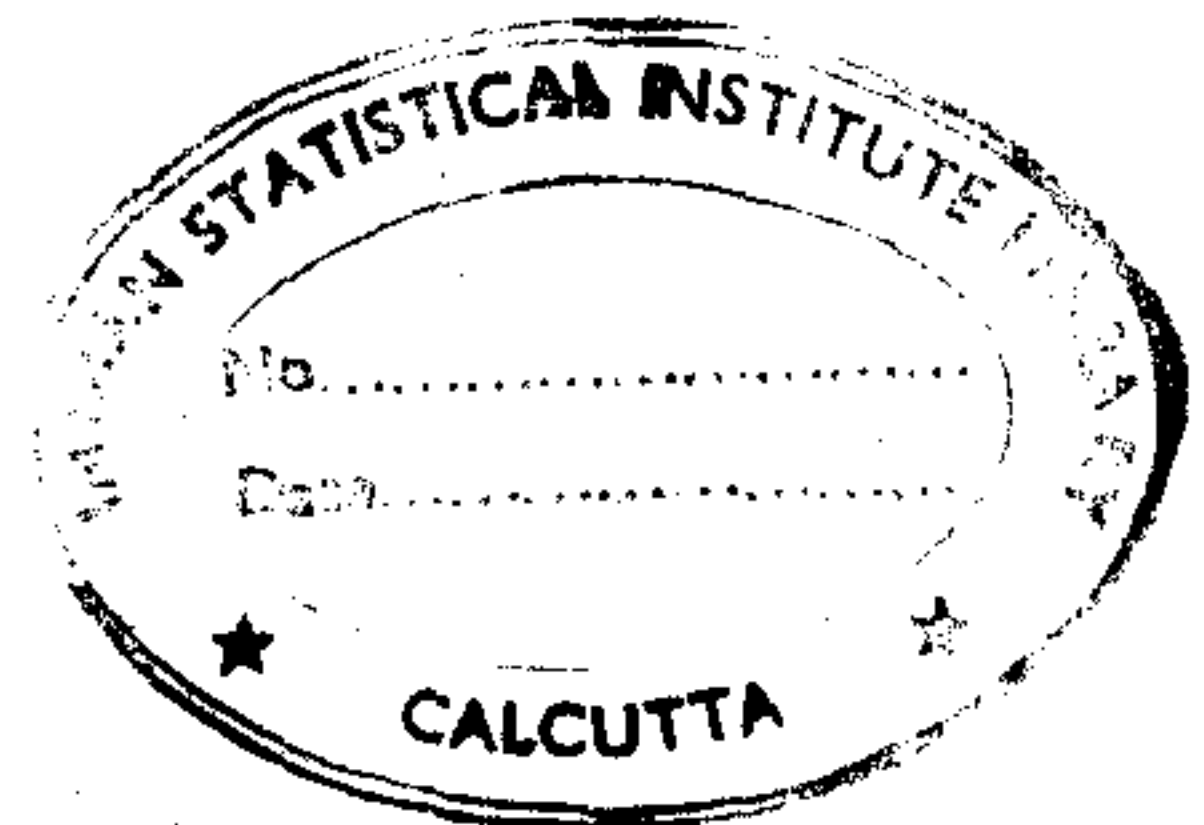
Scheduling Memory in a Multiprocessing Environment

A Dissertation submitted in partial fulfilment of the requirements for the award of M.Tech
(Computer Science) degree from Indian Statistical Institute, Calcutta.

By
Piyush Kumar Srivastava

Under the supervision of
Prof. Bhabhani Prasad Sinha
Advanced Computing and Microelectronics Unit
Indian Statistical Institute
203, Barrackpore Trunk Road,
Calcutta- 700035

July 29, 1999




Certificate of Approval

This is to certify that the thesis titled **Scheduling of Memory in a Multi-processing Environment** submitted by **Mr. Piyush Kumar Srivastava**, towards partial fulfilment of the requirements for the degree of **M.tech Computer Science** at the **Indian Statistical Institute, Calcutta**, embodies the work done under my supervision.

Date :

28.7.99

 28.7.99

Prof. Bhabani P. Sinha,
Head,
Electronics Unit,
Indian Statistical Institute,
Calcutta-700035

Acknowledgements

I take this opportunity to thank my supervisor *Prof.B.P.Sinha* who not only guided me during this work but also taught me a lot during the M.Tech course. I feel honoured to have worked with him and will always remain indebted to him for all his valuable suggestions and guidance.

I am grateful to the staff of the Computer Centre at ISI for helping me to avail the computing facilities and giving valuable advice whenever required. I thank Mr. Pradip Laha and Mr. Dibyendu for their timely help.

Lastly, I would like to thank all my class mates without whose co-operation and suggestions this project would have remained a dream. I would like to thank my friend Mr. P. Swamy Dhoss who stood by my side always to help me out of difficulties. I would also thank Mr. Arnab Nandy and Mr. Parveen Gupta for their help during this course. The love of all my friends and teachers have made this two year period a memorable one.

Scheduling Memory in a Multiprocessor Environment

Piyush Kumar Srivastava

July 23, 1999

Contents

1 Introduction	3
2 Model Used in Analysis	10
3 An Expression for Bandwidth	14
4 Analysis of (NH, NF) and (H, NF) Classes	16
4.1 (NH, NF) Class	16
4.2 (H, NF) Class	17
5 Analysis of (NH, F) Class	19
5.1 Equal Number of Processors and Modules	19
5.2 Number of Processors more than Number of Modules	21

6	Analysis of (H, F) Class	23
7	Implementational Aspects of Our Simulation	26
8	Results	30
9	Conclusions	32

Chapter 1

Introduction

Many real-life applications such as Digital Signal Processing, Image Processing, weather forecasting, neural processing, etc. require large amounts of computations to be performed in an acceptable time frame. It is here that parallel machines provide much superior performance compared to conventional sequential machines (Von-Neumann Architecture). The demand for increased performance appears to have no upper-bound in the present day context. Research and Development activities world wide have led to faster and faster processors for meeting ever-increasing demand for better performance. Major technological breakthroughs in the form of VLSI have brought about a revolution in miniaturising component size and achieving even higher speeds. But there is a fundamental physical limit to speed whereby nothing can move faster than light. With today's technology, computers have reached close to this fundamental limit. Hence, there is a limit on the number of floating point operations that can be executed on such processors per second. Parallel Processing has been accepted as the most important architectural approach to overcome the technological barrier.

Parallel Processing is currently a hot research topic with wide research being carried out by world's leading scientists and professors. There remains a lot of scope for improvements and more efficient algorithms keep coming notwithstanding the fact that a quantum leap has already been made in design of parallel machines. Parallel machines are being foreseen as the future generation machines. While the architecture for supporting parallel computation is an issue, it is certainly not the major problem. There are also issues relating to the programmability of the parallel machines. For effectively utilizing the features offered by parallel architectures, the synergism among architectures, algorithm and programming should be properly fostered.

Parallel Processing has been inherently used in some form or the other to increase performance of computers ever since their emergence as computing devices. It is the performance that sets an index to commercial existence of a computer. In this direction, capabilities for bit parallel arithmetic, development of I/O co-processors, cache and content addressable memories and applications of pipelining, vector processing, time sharing, multiprogramming, multiprocessing, all strive to achieve optimal performance from a computer exploiting inherent parallelism.

In Parallel Processing we use multiple processors to execute a single task. This is achieved by parallel algorithms which distribute a single task among all processors. These processors then communicate with each other using shared variables in shared memory modules after calculating the intermediate results. Proper synchronization is needed to preserve semantic dependence in order to guarantee the expected results. This is what makes the parallel computation more complicated. The interconnection mechanism used to connect processors to memory modules must be fast enough to guarantee that the time taken for communication does not nullify the advantage achieved by parallel computation.

The following are the major issues to be considered before one decides to go ahead with parallel system :-

- Firstly, we must decide whether the parallel system should support the illusion that the processor or system is actually executing the program using a single control point.
- Secondly, we must find whether a particular problem or task can be implemented effectively in a parallel environment. At the same time we need to preserve the system functionality of the task. An algorithm must be found to distribute a big task among several processors and synchronize the execution.
- Thirdly, we need to devise a control mechanism to coordinate parallel activities.
- and Lastly, we need to exercise special care to implement an application efficiently, especially if the system's parallelism is not completely hidden by the illusion of a single point of execution.

A shared memory multiprocessor system consists of a set of processors $\{P_1, P_2, \dots, P_n\}$ and a set of memory modules $\{M_1, M_2, \dots, M_n\}$. The processor memory communications are established either through a set of switches or through a global shared (multi) bus. Both types of connections will be called under the generic name *interconnection network*. A logical diagram of such a shared memory system is shown in fig. 1.1.

The interconnection mechanism should allow efficient resource sharing among the processors. Conflict arises when more than one processors attempt to access memory modules using the same path or switching module in the network. The primary goal of interconnection mechanism is to minimize this conflict. In our discussions we assume that the interconnection mechanism is free from any contentions.

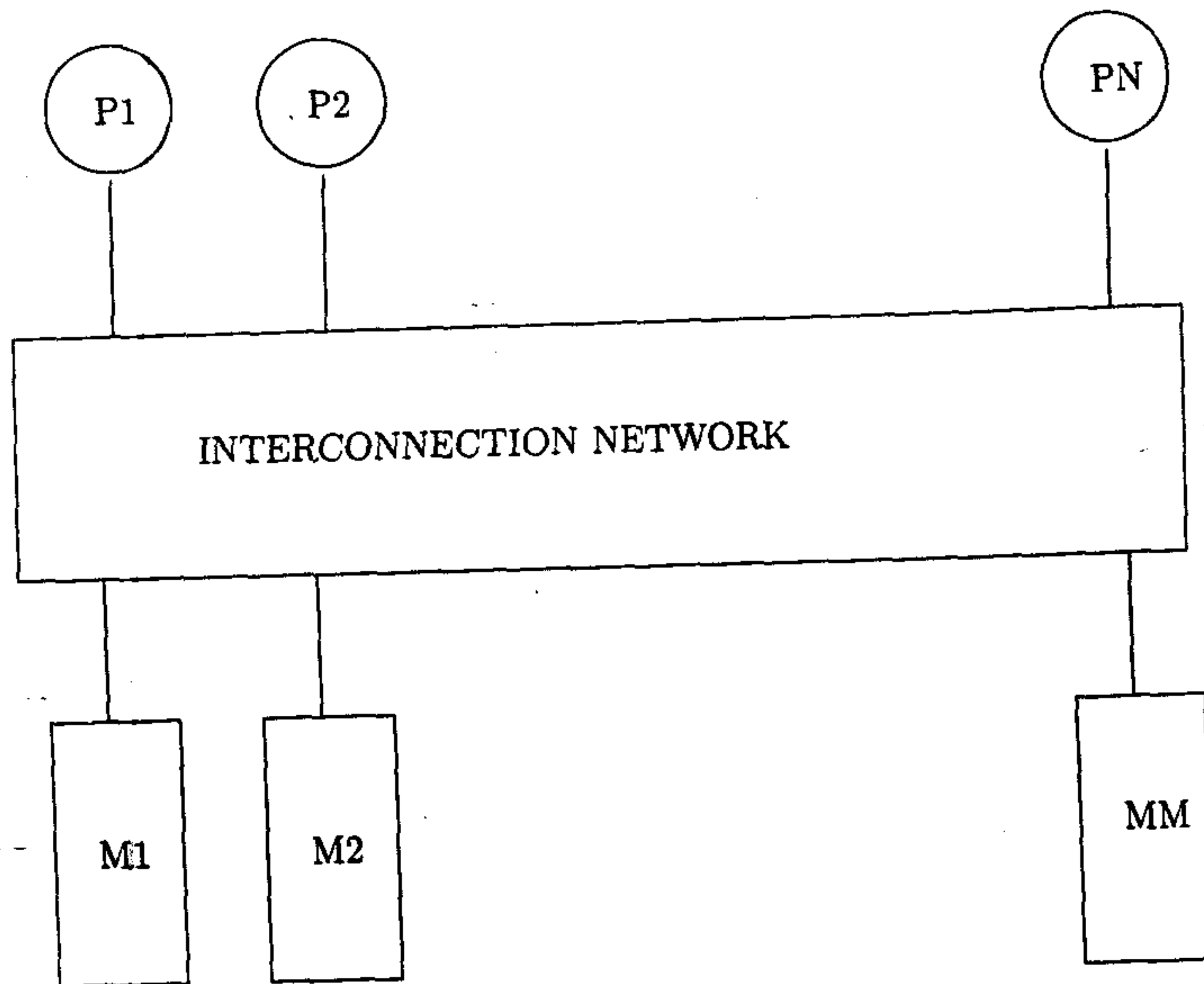


Figure 1.1: Logical Diagram of Processor Memory Interconnection

Bandwidth is defined as the average number of busy memory modules in each memory cycle. Ofcourse the Bandwidth (henceforth denoted by BW) is also dependent on the request rate of the processors. It gives an indication of the rate at which the requests are getting served.

The memory reference patterns of the processors can be *uniform* or *non-uniform*. The reference pattern is uniform if all processors have the same probability of accessing any memory module. This is a valid assumption if address interleaving on the lower order address bits are used. Without this kind of interleaving, however, the memory reference pattern will

be non-uniform in most cases and will depend on the locality of reference.

Uniform Memory Access Patterns (UMA)

Here the physical memory is uniformly shared by all the processors. All processors have equal access time to all memory words. Each processor may also use a private cache. Peripherals are also shared in the same fashion. When all processors have equal access to all peripheral devices, the system is called a *symmetric* multiprocessor. The UMA model is suitable for general purpose and time sharing applications by multiple users. It can also be used to speed up the execution of a single large program in time-critical applications. To coordinate parallel events, synchronization and communications among the processors are done using shared variables.

Non-Uniform Memory Access Patterns (NUMA)

The NUMA multiprocessor system is a shared memory system in which the access time varies with the location of the memory word. The shared memory is physically distributed to all processors, called *local memories* of the processors. The collection of all local memories forms a global address space accessible by all the processors. Obviously, any processor accesses its local memory faster than it accesses any other memory module. The delay will be due to the interconnection network.

In a shared memory multiprocessor system having N processors and M memory modules, contention arises when more than one processor generate requests for the same module,

in the same memory cycle. This contention is measured in terms of memory bandwidth of the system ,ie., the number of modules busy in the same cycle.

Some authors have used a uniform memory-reference model in determining the analytical performance of the system [1]–[3] . But practical situations show that memory references in a multiprocessor system are not necessarily uniform. The non-uniform memory access patterns can be *intra-cycle* or *inter-cycle* [8]. There are four types of intra-cycle non-uniformities. In the first type, each processor may have a *favorite* module which it accesses more frequently than other modules [4] [5]. In the second type, all the processors may access a particular module or a class of memory modules more often than others. This particular module(s) is(are) called *Hotspots*. This phenomenon was first observed by Pfister [6]. The third type assumes that the module that a processor is accessing in present cycle will be accessed by the same processor in the next cycle with a probability $\gamma > \frac{1}{M}$. The fourth type, also called spacial non-uniformity, is defined when the events that a particular module has atleast one request are not independent of each other [7].

We classify each memory module as one of the following types, H (Hot), F (Favorite), NH (Non-Hot), NF (Non-Favorite). Now we can have an environment with any one of the following four combinations of memory modules : (NH, NF), (H, NF), (NH,F) and (H, F).

In this work we deal with the problem of memory access contentions for a shared memory multiprocessor system for all the four types of memory modules environments, with the assumption that the processor-memory interconnection network does not create any bottleneck. We use an *Approximate Queueing Model* for our analysis. In an earlier year, analysis of (NH, NF) and (H, NF) classes has been carried out by Mr. Rajarshi Choudhary [9]. We suggest small improvements in the expressions derived earlier and proceed to carry out the analysis further for the (NH, F) and (H, F) environments. We show that the presence of multiple hot spots in the system leads to an overall improvement in the system

performance in the (HF) environment. We also show that the Bandwidth increases with no. of hot spots only to a particular value. This limit is called saturation value. We also show that if processors access their favorite memories with high probability, an improvement in Bandwidth is observed.

We have also simulated a shared memory multiprocessor environment where N processors produce requests to memory modules at the beginning of a memory cycle. The requests generated are in a random order and independent of requests generated in any earlier cycle. Implementational aspects are discussed in brief in chapter 7. We then compare the results obtained by analytical expressions with those obtained by simulation. These are shown graphically in Appendix A.

Chapter 2

Model Used in Analysis

This chapter provides the basic model used and the assumptions made for the remainder of discussions. It also lists the symbols used in the work along with their meanings. The assumptions we make are as follows:

1. The system consists of n identical processors and M globally accessible identical memory modules. Usually, $N \geq M$.
2. There is no contention in the interconnection mechanism.
3. The system is synchronous, ie., all requests made by the processors are at the beginning of a memory cycle with a probability m .
4. All memory modules have a constant cycle time.
5. Each processor generates random and independent memory requests, If two or more processors make simultaneous requests directed to the same memory module, only one

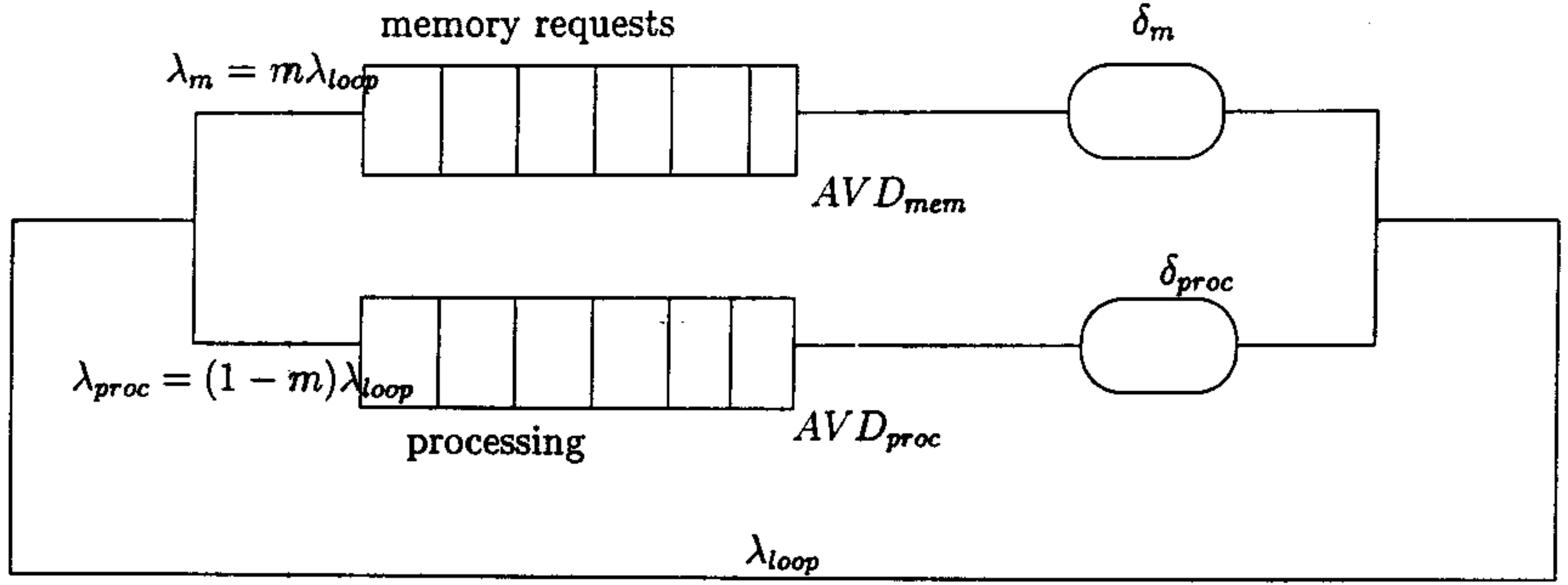


Figure 2.1: Aggregated Queueing Model

of these will be served in that cycle. The remaining requests will be queued up in the memory queue of that module. Obviously a processor produces next request only when its current request has been served.

6. If there are K hot modules in the system, they are all identical and memory reference is uniform among them. So, a processor accesses a particular hot module with a probability $\frac{1}{M}$ provided that it produces a hot request.
7. The reference pattern among non-hot, non-favorite modules is also uniform.

The nomenclature we use in the following chapters are as shown in table 2.1.

The model we use in our analysis is the Approximate Queueing model shown in fig. 2.1. We use the flow equivalent aggregation to find the average delay of memories as a function of memory request rate. For simplicity we assume $m=1$, ie., the probability of a processor producing a request at the beginning of a memory cycle is 1. We also assume that request arrivals at the memory modules follow a Poisson's distribution.

Table 2.1: Nomenclature used in the Analysis

Symbol	Description
N	Number of Processors
M	Number of Memory Modules
K	Number of Hot Modules
m	Probability with which a Processor produces a Memory request at the beginning of a cycle
β	Memory Cycle Time
P_h	Probability with which a Processor produces a Hot Memory Request (in other words Hot Memory Request Rqte for each Processor)
γ	Favorite Memory Request Rate for each Processor.
BW	Bandwidth
AVD_h	Load-dependent average delay in a Hot Module as seen by each Processor
AVD_f	Load-dependent average delay in a Favorite Module, as seen by each Processor
AVD_{nhf}	Load-dependent average delay in Non-Hot,Non-Favorite Module, (as seen by each Processor)
λ_h	Total arrival rate of request at each Hot Module
λ_{nh}	Total arrival rate of request at each Non-Hot Module
λ_{nhf}	Total arrival rate of request at each Non-Hot,Non-Favorite Module
δ_i	Total service rate of requests at i^{th} Memory Queue
ν	Utilization Factor

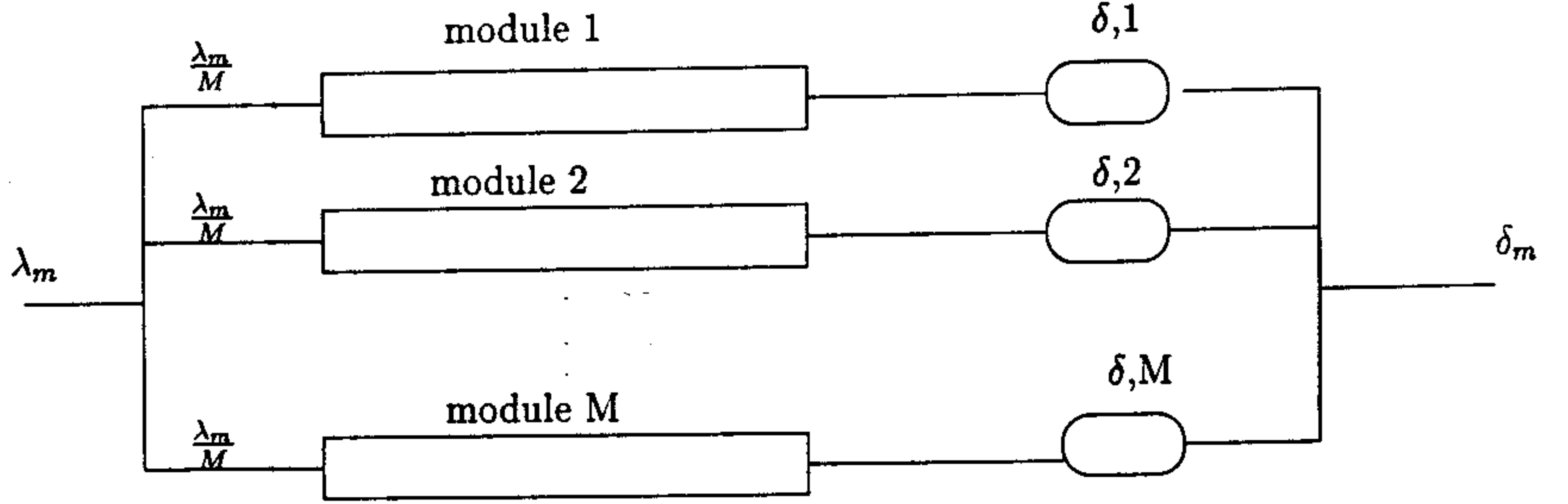


Figure 2.2: Decomposed Memory Model

All processors place their request for memory at the beginning of a cycle. If more than one request arrive at a memory module, only one of them is served and the rest are queued up in the memory buffer (queue). In fig. 2.1, δ_m is the rate of service of memory requests from the aggregated memory queue. δ_{proc} is the rate at the processor end. λ_m is the rate at which a processor sends a request to the aggregated memory queue. If λ_{loop} is the rate of request across the processor-memory loop then $\lambda_m = m\lambda_{loop}$.

Fig 2.2. shows the decomposed memory queue. Note that this is the decompsed model for the case that there are no Hot or Favorite memory modules. Then the rate of request at each memory module will depend on the reference pattern among processors. This will be dealt with in detail in subsequent chapters.

Chapter 3

An Expression for Bandwidth

In this chapter we lay the foundation for our analysis in terms of basic expressions which will be used in future discussions. We state an equation for bandwidth in terms of the average request rate of each processor. We start by finding an expression for average delay seen by each processor.

The probability that there are j requests in a memory queue is given as $p_j = (1 - \nu_i)\nu_i^j$ [10], where ν_i is the utilization factor for the i^{th} memory module. ν_i is given by $\nu_i = \frac{\lambda_i}{\delta_i}$, where λ_i is the rate at which request arrive at the queue and δ_i is the service rate at the queue front. Usually we have, $\nu_i \leq 1$. This is obvious because otherwise the queue will go on getting filled indefinitely.

The total delay that a memory request experiences when there are already j request in the queue is $(j + 1)\beta$, where β is the memory cycle time. Then, the average delay in that

module is given by

$$AVD_m = \sum_{j=0}^{\infty} p(j) \text{delay}(j)$$

On simplification we get in general,

$$AVD_i = \frac{\beta}{1 - \nu_i}$$

We have $\beta = 1/\delta_i$, assuming that all modules have same service time β . So we have,

$$AVD_i = \frac{\beta}{1 - \lambda_i \beta} \quad (3.1)$$

Total average delay across the loop is given by

$$AVD_{loop} = m AVD_m + (1 - m) AVD_{proc}$$

If λ_{loop} is the loop rate, then, it is given by,

$$\lambda_{loop} = \frac{1}{AVD_{loop}}$$

Since we assumed $m=1$, we get

$$\lambda_m = \lambda_{loop} = \frac{1}{AVD_m}$$

So, in general we may write,

$$\frac{1}{\lambda_m} = AVD_m = \sum_{i=1}^M p(i) AVD_i \quad (3.2)$$

where, AVD_i is given by eq. 3.1.

A simple expression for Bandwidth follows once we have calculated λ_m . Note that since a processor produces a request once its previous request is served, the rate at which requests are served will be the same as the rate at which they are effectively produced by processors. So Bandwidth is given by

$$BW = N \lambda_m \quad (3.3)$$

Chapter 4

Analysis of (NH, NF) and (H, NF) Classes

In this chapter we discuss the environments when there are no favorite modules. We use the results derived in last chapter and find expressions for throughput of the multiprocessor system which is ofcourse measured by Bandwidth. We also see that the presence of Hot-spots improves the performance by increasing the effective bandwidth. We see that beyond a saturation value, there is no increase in performance if we increase the no. of hot spots. Let us start with the simplest case when no Hot-spot and no Favorite modules are present.

4.1 (NH, NF) Class

We work on the assumption that all processors and memory modules are identical. Thus, when a processor produces a request, the probability that it goes to a particular module is

$\frac{1}{M}$. So, referring to equation 3.2, $p(i)$'s for all modules is $1/M$. The rate of request arriving at all modules is the same. When a processor sends a request, the delay it sees is due to the request rates of the remaining $N-1$ processors. So,

$$\lambda_i = \frac{(N-1)\lambda_m}{M}$$

Thus,

$$\frac{1}{\lambda_m} = \frac{\beta}{1 - \lambda_i \beta} = \frac{\beta}{1 - \frac{(N-1)\lambda_m \beta}{M}}$$

Or,

$$\lambda_m = \frac{M}{(M + N - 1)\beta}$$

Thus,

$$BW = \frac{MN}{(M + N - 1)\beta}$$

4.2 (H, NF) Class

Again all the Hot memory modules are identicle. Requests coming to Hot modules are uniformly distributed over all Hot modules. Similarly for non-Hot modules. The part of the request generated by a processor going to Hot modules is $p_h \lambda_m$ and that to non-Hot modules is $(1 - p_h) \lambda_m$.

We now consider a single processor sending request at a rate λ_m . The average delay this processor sees is given by,

$$AVD_m = p_h AVD_h + (1 - p_h) AVD_{nh} \quad (4.1)$$

where, AVD_h and AVD_{nh} are the average delays seen in Hot and non-Hot modules respectively. This equation also follows from equaion 3.2 since $p(i)$'s for all hot modules is p_h and

$p(i)$'s for all non-Hot modules is $(1 - p_h)$. From equation 3.1 we calculate the average delays as,

$$AVD_h = \frac{\beta}{1 - \lambda_{ih}\beta}, AVD_{nh} = \frac{\beta}{1 - \lambda_{inh}\beta} \quad (4.2)$$

The request rate at any hot and non-hot modules due to $N-1$ processors are respectively given by,

$$\lambda_{ih} = \frac{(N-1)\lambda_m p_h}{K}, \lambda_{inh} = \frac{(N-1)\lambda_m(1-p_h)}{M-K} \quad (4.3)$$

Solving equations 4.1, 4.2 and 4.3 we get a quadratic equation of the form

$$a\lambda_m^2 + b\lambda_m + c = 0$$

where,

$$a = p_h(1-p_h)\beta^2(N-1)(M+N-1)$$

$$b = -\beta K(M+N-1-K) - \beta p_h(N-1)(M-2K)$$

$$c = K(M-K)$$

We then get

$$\lambda_m = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

The other solution is neglected as it does not fall in the range $[0,1]$. Bandwidth is now given by $BW = N\lambda_m$. By plotting the values of bandwidth against the number of Hot modules we observe that it increases with number of hot modules upto a saturation value. Thereafter it rapidly falls. The calculated and simulated results have been shown in appendix A.

Chapter 5

Analysis of (NH, F) Class

In this chapter we analyze the case when only Favorite modules are present. We start with a special case that the number of processors and number of modules are both same and then proceed to the more general case. Each module is favorite to only one processor. The probability with which a processor accesses its favorite module is γ .

5.1 Equal Number of Processors and Modules

There are N Processors and N Memory Modules. Each processor accesses its favorite module with a probability γ and any non-Favorite memory with probability $\frac{(1-\gamma)}{N-1}$. When a processor sends a request, it encounters delay due to requests from other processors. When it sends a request to its favorite module, the number of requests arriving at that module from other $N-1$ processors is much less than the those arriving from it. So the rate of request at its favorite module due to oher processor is quite less. Thus it sees lesser delay whenever it

request its favorite module. But when the processor sends request to a non-favorite module, the rate of request there is high. Most of it comes from the processor which favours that particular module. So it sees a larger delay. The average delay it sees is thus given by,

$$AVD_m = \gamma AVD_f + (1 - \gamma) AVD_{nf} \quad (5.1)$$

where, AVD_f and AVD_{nf} are respectively the delays seen in favorite and non-favorite modules. Let λ_{if} be the rate of request arrival at any module from those $N-1$ processors which do not favour it. This rate will contribute to delay seen by a favorite request. Let λ_{inf} be the rate of request at a module from the processor favoring it and from remaining $N-2$ processors (since this excludes the processor which makes the request). This will be the rate as seen by a processor requesting a non-favorite module. λ_{if} is given by,

$$\lambda_{if} = \frac{(N-1)\lambda_m(1-\gamma)}{N-1} = \lambda_m(1-\gamma) \quad (5.2)$$

since the requests come from $N-1$ processors and are distributed uniformly among $N-1$ modules. λ_{inf} is given by,

$$\lambda_{inf} = \frac{(N-2)\lambda_m(1-\gamma)}{N-1} + \lambda_m\gamma \quad (5.3)$$

Again the average delays are given by,

$$AVD_f = \frac{\beta}{1-\lambda_{if}}, AVD_{nf} = \frac{\beta}{1-\lambda_{inf}} \quad (5.4)$$

Solving equations 5.1 5.2 and 5.4, we get a quadratic equation

$$a\lambda_m^2 + b\lambda_m + c = 0$$

where,

$$a = pq\beta^2 + q\gamma\beta^2 + (1-\gamma)p\beta^2$$

$$b = -\beta[1+p+q]$$

$$c = 1$$

and $p = (1 - \gamma)$ and $q = \left[\frac{(N-2)(1-\gamma)}{N-1} + \gamma\right]$

λ_m is again found as in earlier cases and the expression for BW follows. If we consider the case when $\gamma = 1$ we notice that, λ_{if} becomes 0 and we get $\lambda_m = 1$. This is expected because in this case all processors request their respective modules and produce new request in each memory cycle. They see no delay due to the other modules. Then we get $BW = N$.

5.2 Number of Processors more than Number of Modules

We assume that there are N processors and M modules ($N \geq M$). Each module is favorite to only one processor, and each processor has only one favorite module. Remaining $M-N$ processors do not have favorite memories. This case is more complicated as the processors are not all having identicle reference patterns. Let λ_m and λ_{nm} be the average request rates of processors with favorite modules and those without them respectively. Then bandwidth is given by,

$$BW = M\lambda_m + (N - M)\lambda_{nm} \quad (5.5)$$

λ_{if} and λ_{inf} get modified as follows:

$$\lambda_{if} = \frac{(M-1)(1-\gamma)\lambda_m}{M-1} + \frac{(N-M)\lambda_{nm}}{M} = (1-\gamma)\lambda_m + \frac{(N-M)\lambda_{nm}}{M} \quad (5.6)$$

$$\lambda_{inf} = \left[\gamma + \frac{(M-2)(1-\gamma)}{M-1} \right] \lambda_m + \frac{(N-M)\lambda_{nm}}{M} \quad (5.7)$$

λ_{inm} is the rate of request at a module due to all remaining processors as seen by a

processor having no favorite module. This is given by,

$$\lambda_{inm} = \left[\gamma + \frac{(M-1)(1-\gamma)}{M-1} \right] \lambda_m + \frac{(N-M-1)\lambda_{nm}}{M} = \lambda_m + \frac{(N-M-1)\lambda_{nm}}{M} \quad (5.8)$$

The average delay seen by a processor having favorite modules is given by,

$$\frac{1}{\lambda_m} = \frac{\gamma\beta}{1 - \lambda_{if}\beta} + \frac{(1-\gamma)\beta}{1 - \lambda_{inf}\beta} \quad (5.9)$$

$$\frac{1}{\lambda_{nm}} = \frac{\beta}{1 - \lambda_{inm}\beta} \quad (5.10)$$

Solving equations 5.6 to 5.10 we get two quadratic equations in two variables which can be solved to obtain expressions for λ_m and λ_{nm} . The bandwidth will then be given by equation 5.5. The values of bandwidth found by solving the above equations (by computer program) have been plotted against the simulated results.

Chapter 6

Analysis of (H, F) Class

In this chapter we analyse a more general case. The memory modules could be either Hot or Favorite or neither of them. We however assume that a particular module can be favorite to one processor only. Similarly each processor has only one favorite module if it at all has. There are M modules out of which K are Hot, F are Favorite and rest $M - F - K$ are neither of these. Among N processors, F will have corresponding favorite modules and rest $N - F$ have no favorite memories. Again as in (NH, F) case, the processors have different reference patterns. So, the effective rate of request for the F processors having favorite modules will be different from the effective rate of request from the other $N - F$ processors. Let these be respectively λ_f and λ_{nf} .

Average delay seen by any of F processors (AVD_f) is given by,

$$\begin{aligned}
\frac{1}{\lambda_f} &= AVD_f \\
&= \gamma AVD_{(f,1)} + (1-\gamma)(1-p_h) \left[\frac{M-F-K}{M-K-1} AVD_{nhf} + \frac{F-1}{M-K-1} AVD_{(f,f-1)} \right] \\
&\quad + (1-\gamma)p_h AVD_h
\end{aligned} \tag{6.1}$$

where, $AVD_{(f,1)}$ is the average delay seen by the processor in its single favorite module, AVD_{nhf} is delay in non-hot, non-favorite module, $AVD_{(f,f-1)}$ is the delay seen by it in remaining $F-1$ modules and AVD_h is delay in hot module. If the processor does not produce a request to favorite or hot module (with probability $(1-\gamma)(1-p_h)$) then it gets distributed uniformly among $M-K-1$ modules which are neither hot nor its favorite. This explains the above equation.

Similarly the expression for average delay seen by any of the $N-F$ processors is given as,

$$\frac{1}{\lambda_{nf}} = (1-p_h) \left[\frac{M-F-K}{M-K} AVD_{nhf} + \frac{F}{M-K} AVD_{(nf,f)} \right] + p_h AVD_h \tag{6.2}$$

Note that $AVD_{(nf,f)}$ and $AVD_{(f,f-1)}$ which are request rates from $N-F$ processors to favorite modules, and from F processors to $F-1$ non favorite modules are approximately the same. The expressions for average delays mentioned are based on equation 3.1 with corresponding λ_i values being different for each of the cases above. Following the same subscript notation we calculate the λ_i values as follows:

$$\lambda_{i(f,1)} = \left[\frac{(N-F)(1-p_h)}{M-K} \right] \lambda_{nf} + \left[\frac{(1-\gamma)(1-p_h)(F-1)}{M-K-1} \right] \lambda_f \tag{6.3}$$

$$\lambda_{i(f,f-1)} = \lambda_{i(n,f)} = \left[\frac{(N-F)(1-p_h)}{M-K} \right] \lambda_{nf} + \left[\gamma + \frac{(1-\gamma)(1-p_h)(F-1)}{M-K-1} \right] \lambda_f \quad (6.4)$$

$$\lambda_{ih} = \left[\frac{(N-F)p_h}{K} \right] \lambda_{nf} + \left[\frac{(1-\gamma)p_h F}{K} \right] \lambda_f \quad (6.5)$$

$$\lambda_{inhf} = \left[\frac{(N-F)(1-p_h)}{M-K} \right] \lambda_{nf} + \left[\frac{(1-\gamma)(1-p_h)F}{M-K-1} \right] \lambda_f \quad (6.6)$$

We observe that the average delays AVD_{inhf} and $AVD_{i(f, \blacksquare 1)}$ are very nearly the same. If we treat them as same we get two cubic equations in two variables (λ_f and λ_{nf}) after simplification of equations 6.1 to 6.6 and using equation 3.1 as a general expression for average delays. The cubic equation involves cross-products of variables and can best be solved numerically (by programming). The Bandwidth is then given by

$$BW = F\lambda_f + (N-F)\lambda_{nf}$$

The results obtained by simulation are compared with those obtained by numerically solving the above equations and shown graphically in appendix A.

Chapter 7

Implementational Aspects of Our Simulation

In this chapter we present a brief description of our simulation which we used to compare our analytical results with. The simulation was done on Sun Microsystem workstation which provide facilities to run more than one program at the same time and view the intermediate outputs on the same screen. Debugging and testing during intermediate stages would not have been possible without these facilities. The simulation consists of mainly two programs (written in C) which run simultaneously in synchronism. One program, *process.c*, simulates the processor side by producing requests for memory modules in a random fashion. The other program *memory.c* simulates the memory queues by serving the requests and also counting the number of requests served. The two processes share some common files described later. They also share common variables which are included in *header.h* file. To simulate the four different classes, the same basic structure was used with small modifications in different directories. We now describe the the two programs in brief.

process.c

The program generates random requests from N processors (labelled from 1 to N) to M memory modules (labelled from 1 to M). It accepts a *seed* value as an argument to executable file which enables to produce different sequences of random numbers each time it is run. The process calls a delay routine *request_delay* which produces a delay of a fixed duration (defined in *header.h*) each time before producing the next set of requests. This is used to synchronize its execution with the memory process. It calls the routine *place_req* which generates the random request set and places it in the file *Requests*. The routine *place_req* takes care of the different reference patterns among the processors. We present in figure 7.1 a part of this routine for the (H, F) environment. The code for the rest of cases is similar to this one.

The code shown in 7.1 assumes that the first NO_FAVMOD memory modules are favorite, next NO_HOTMOD modules are hot spots and the remaining are neither. Similarly the first NO_FAVMOD processors are having favorite modules and the rest have none. Before producing the requests, the routine reads the *Ready* file which contains the information about the previous request produced by processors. If their earlier request is served then they produce new one, else they produce a request to 0th module (which is nonexistent and so) later discarded.

The main program puts a request in a file called *Requests* and puts 1 in a *Status* file. It then waits until the memory process has read the new request set. It then produces new request set. This is repeated over a pre-defined number of cycles.

```

place_req()
{
for(i=1;i<=NO_PRCSRS;i++)
{
fseek(ready_fp,i*sizeof(int),0);
fread(&prc_ready,sizeof(int),1,ready_fp);
if(prc_ready==1)
{
seek(ready_fp,i*sizeof(int),0);
fwrite(&zero,sizeof(int),1,ready_fp);
rewind(ready_fp);
if (i<=NO_FAVMOD)
{
if((prob=drand48())<PROB_FAVMOD)
rand_no=i;
else if((prob=drand48())<PROB_HOTMOD)
rand_no=((int)(drand48()*NO_HOTMOD)+1) + NO_FAVMOD ;
else
do
rand_no=((int)(drand48()*NO_MEMMOD)+1) ;
while((rand_no==i)||((rand_no>NO_FAVMOD)&&(rand_no<=NO_FAVMOD+NO_HOTMOD)));
}
else
{
if((prob=drand48())<PROB_HOTMOD)
rand_no=((int)(drand48()*NO_HOTMOD)+1) + NO_FAVMOD ;
else
do
rand_no=(int)(drand48()*NO_MEMMOD+1) ;
while((rand_no>NO_FAVMOD)&&(rand_no<=NO_FAVMOD+NO_HOTMOD));
} /* end if */
requests[i]=rand_no;
}
else requests[i]=0;
} /* end if */
fwrite(requests,sizeof(int),NO_PRCSRS+1,req_fp);
}

```

Figure 7.1: A Code Segment to Generate Random Requests

memory.c

This program reads the request-profile of the processors and adds the requests to respective memory queues along with the processor numbers. It then calls the routine *process_request* which deletes one request from each queue and also counts the no. of requests served in each call. It also places appropriate values in the *Ready* file to indicate that a processor whose request was just served is now ready to produce another request in the next cycle. The routine also calls a delay routine to properly synchronize with the process program. Total number of requests served are counted by the main program which also keeps a track of total number of cycles to find the average Bandwidth. This program also places a 0 in the *Status* file to indicate to process routine that it has read this request and is ready to receive the next set.

The average is calculated over 300-500 cycles. All the parameters such as number of Hot modules, Probabilities of references, cycletime etc. are kept flexible to get bandwidth variation with several parameters. All these are shown in Appendix A.

Chapter 8

Results

The values obtained by analytical calculations have been plotted along with those obtained by simulation. These plots indicate the variation of Bandwidth with various parameters. They also give a measure of accuracy of our model and the correctness of the expressions we derived. We note that for (NH, F) and (H, F) classes, the equations obtained were involved but can be solved by programming.

We observe from the graphs in appendix A. that our Approximate Queueing Model slightly underestimates the Bandwidth. The reason is that, when we calculated the expressions for average delay seen by a processor at a memory module, which is caused due to request rate of all other processors, we assumed that its request was the last to be served among those which were produced in that particular cycle. This assumption was essential to avoid too many complications. We however get values quite close to the simulation values. Also the nature of the curves are identical for both the plots.

We observe that Bandwidth increases significantly with no. of memory modules. This is certainly expected. Bandwidth also increases with no. of processors. But this increase is

significant only for small values. For large no. of processors the increase is not significant.

For the HNF environment, we observe that Bandwidth increases significantly with number of Hot modules upto a saturation value and then falls rapidly. This is because when the no. of hot modules become large, the probability that any one of them is referred becomes significantly less. The very advantage of having hot modules is then lost. We also observe that as probability p_h increases, Bandwidth increases and then decreases slightly. This is because, higher probability of reference to Hot module causes crowding of memory queues and consequently more average delay.

In NHF environment, we see that as the probability of reference to favorite modules is increased, bandwidth increases and when $\gamma = 1$, λ_m also becomes 1 (for the case that number of modules is same as number of processors). This is indicated in the corresponding graph. This is expected as each processor sends requests to its module only and there is no contentions from other processors.

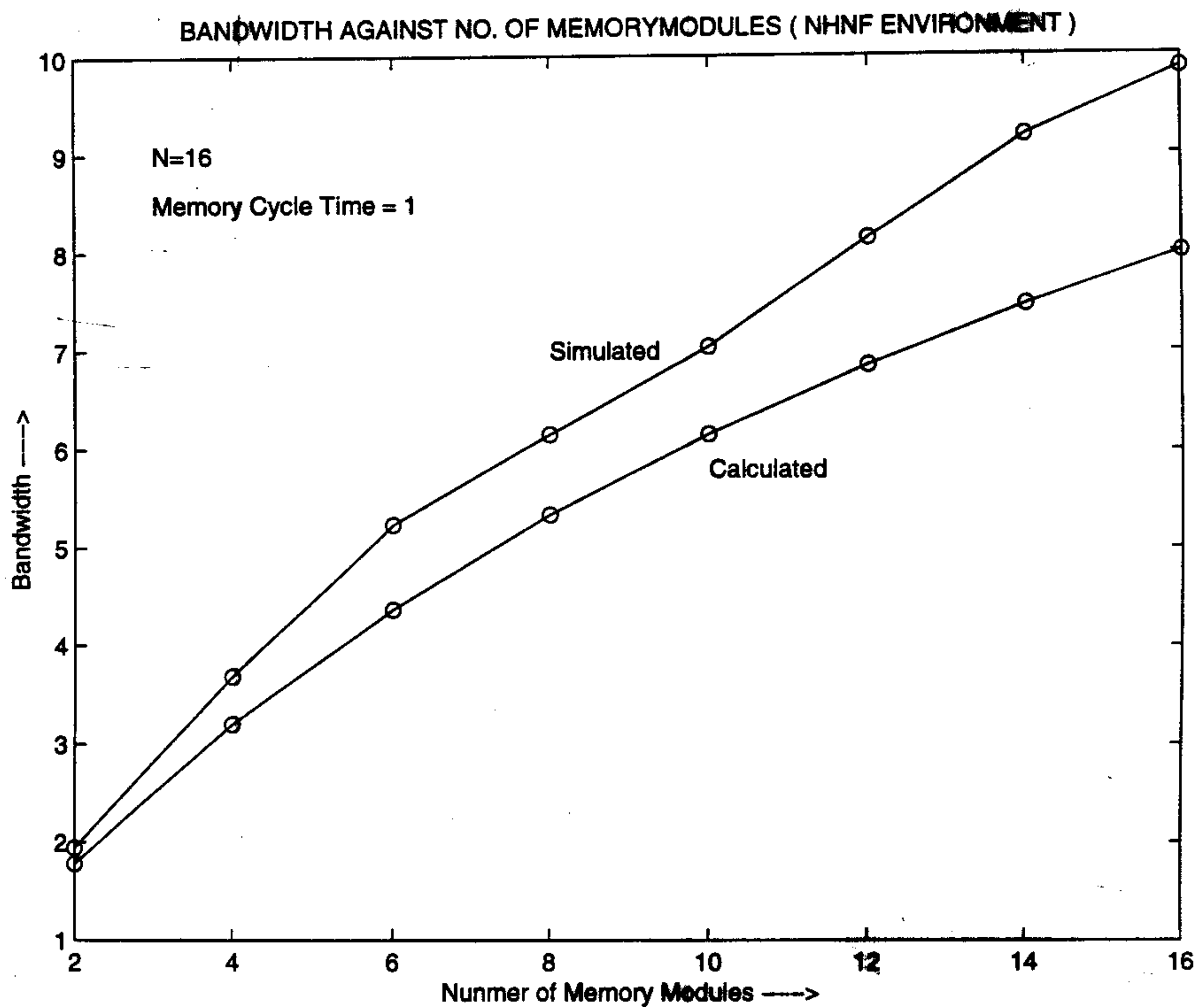
In the HF environment we observe that Bandwidth increases with increase in probability of favorite module reference γ , keeping p_h fixed. It varies only slightly with probability of hot-module reference p_h , when we fix γ . Infact it decreases slightly if p_h is high since it causes crowding of hot modules queues. Also, if we increase the number of hot modules we get significant increase in the bandwidth. But at larger values, the increase is not significant.

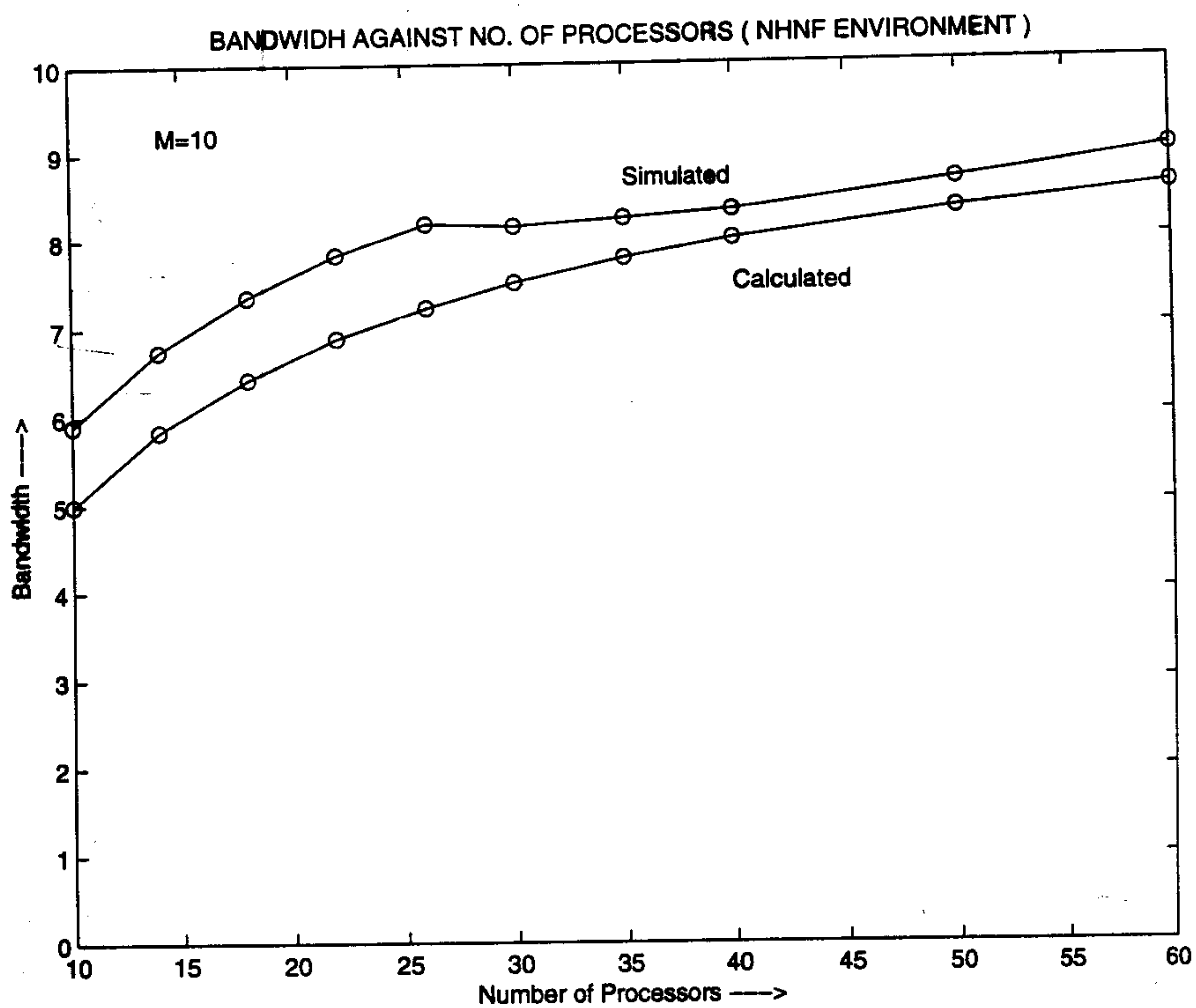
Chapter 9

Conclusions

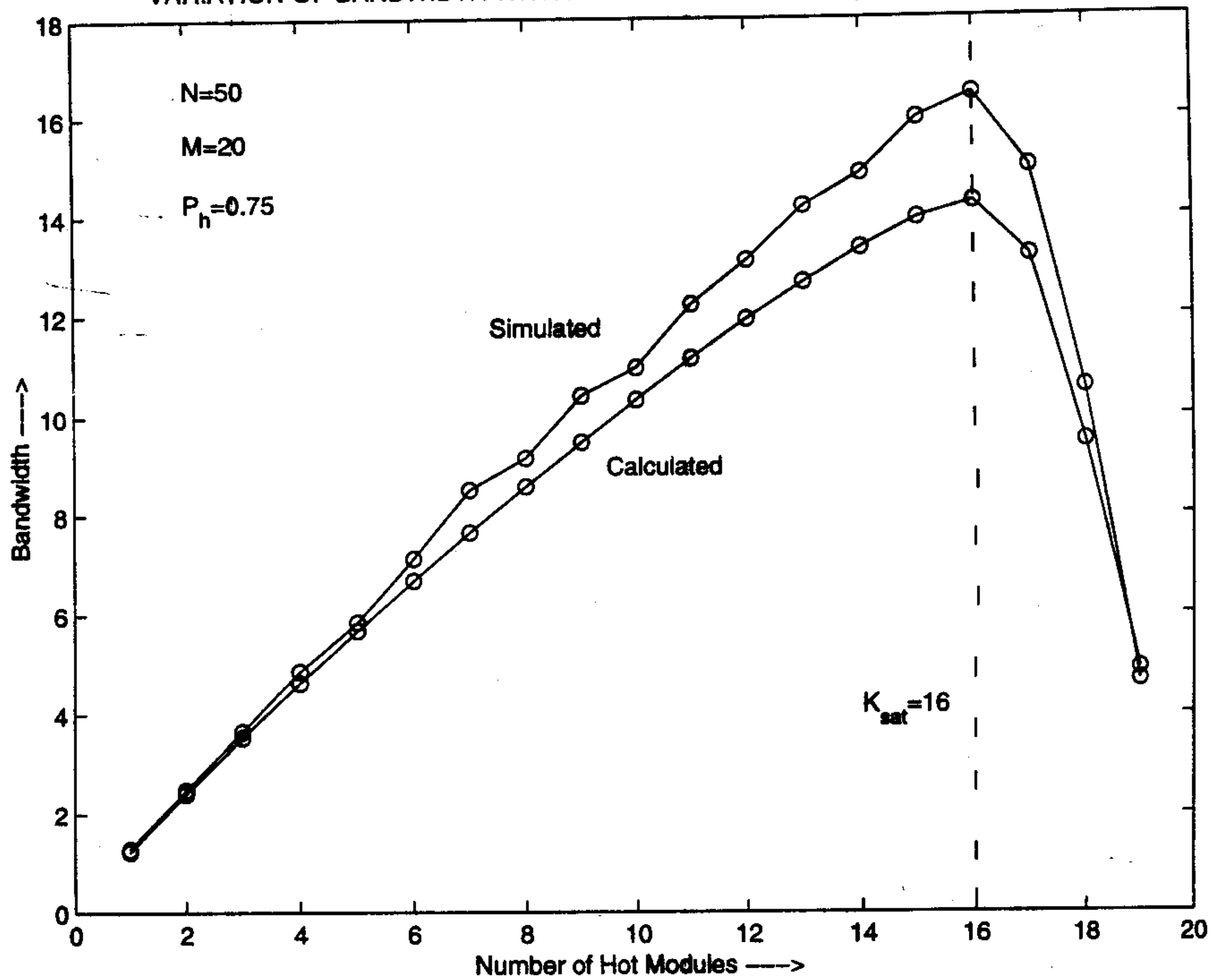
In this work we have studied the performance of general (N,M) multiprocessor system under various forms of non-uniform memory access patterns. We used the Approximate Queueing Model approach to analyse the system performance and proposed expressions for bandwidth for NHNF and HNF classes. For the NHF and HF classes we proposed equations in terms of known parameters, which can be solved numerically to obtain the value of Bandwidth. The analytical results were found to be closely in agreement with the simulated values as is evident in graphs in appendix A. We notice that the bandwidth increases if we have more hot modules in the system provided saturation value is not reached. Bandwidth increases if we have more processors having favorite modules. So, we conclude that in a system with hot spots, it is better to distribute shared variables in more than one module as long as saturation is not reached. Also, if a processor needs a certain set of variables more frequently than other processors, it is best to have them all in one module, which becomes its favorite module.

APPENDIX A

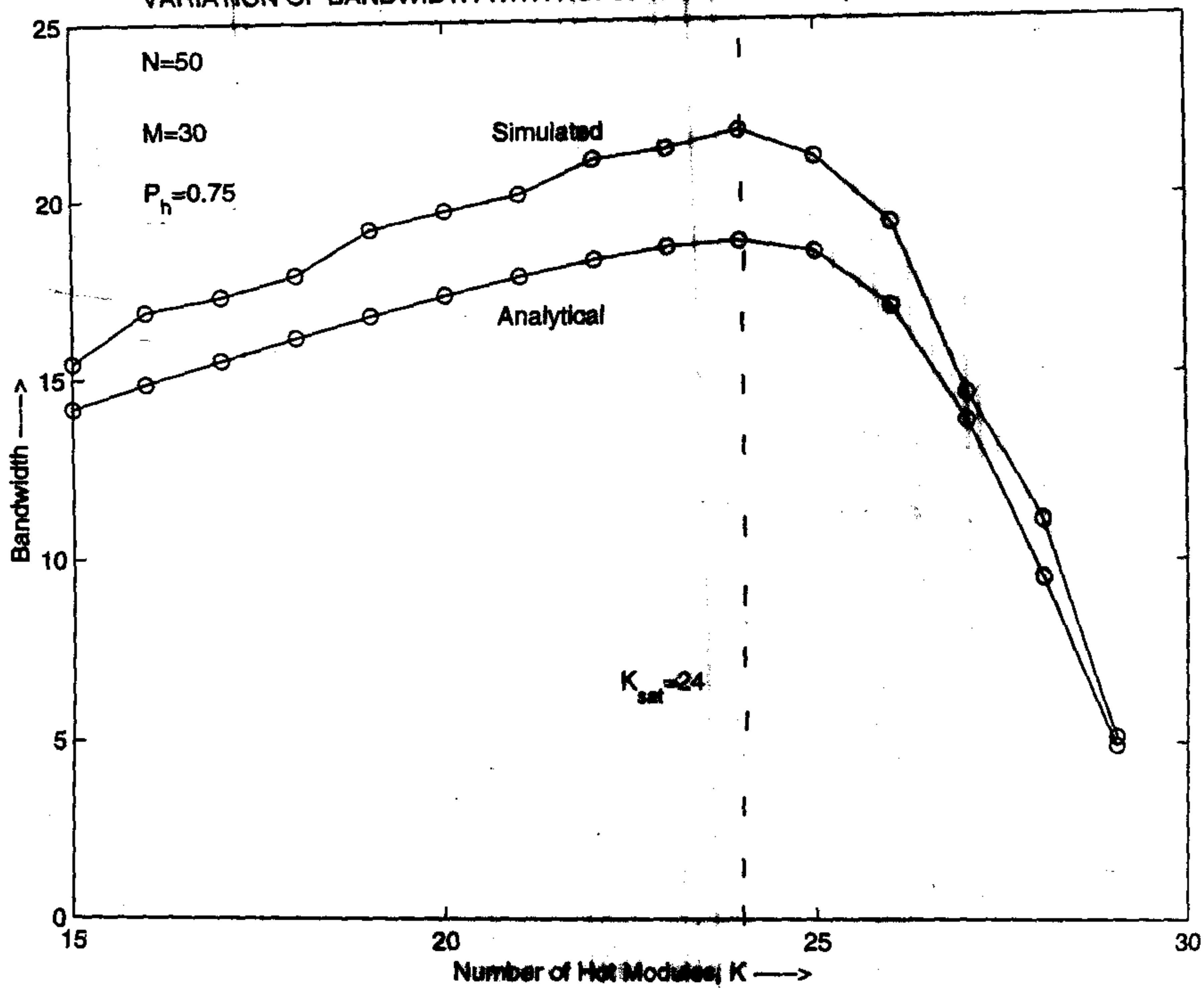




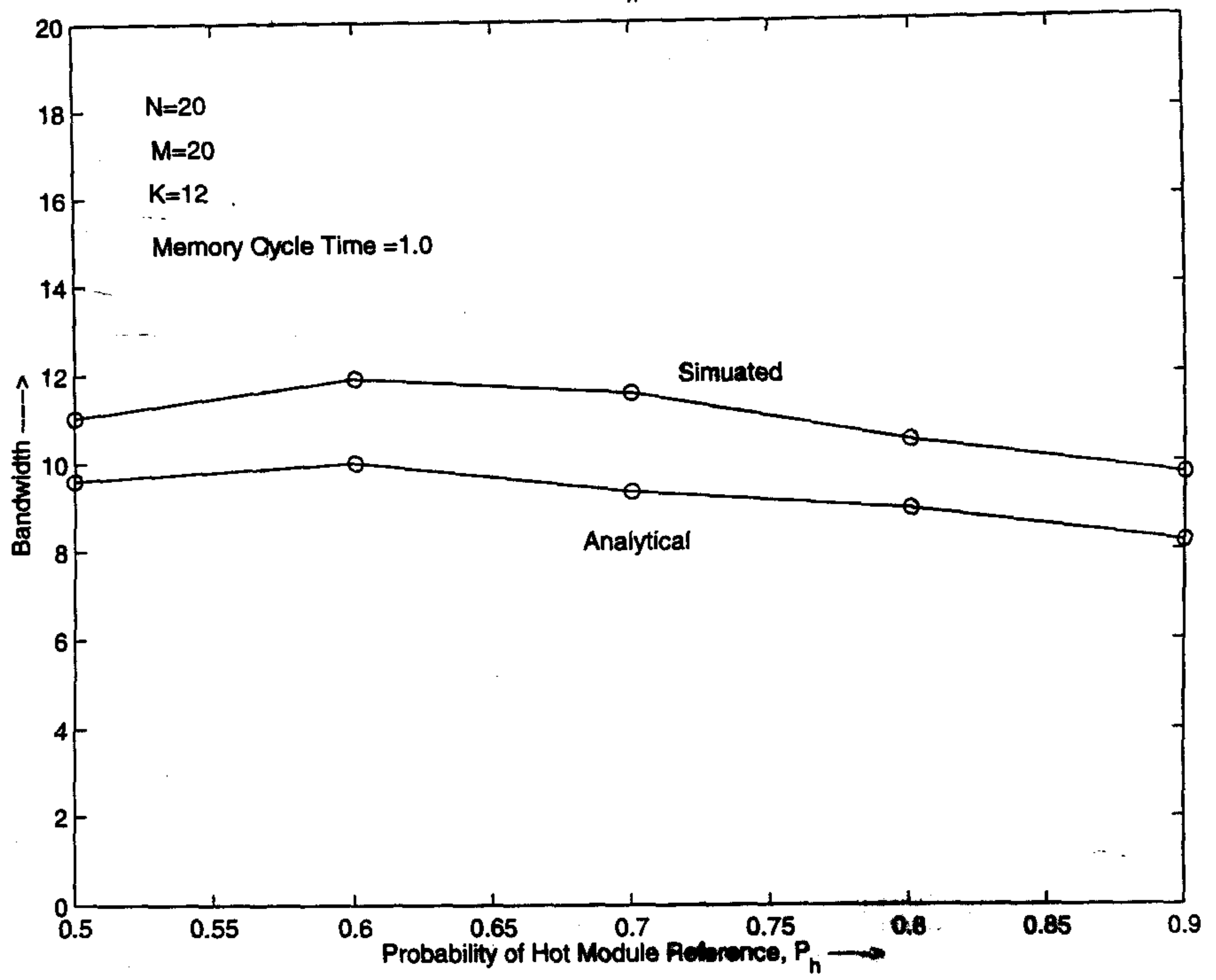
VARIATION OF BANDWIDTH WITH NO. OF HOT MODULES (HNF ENVIRONMENT)

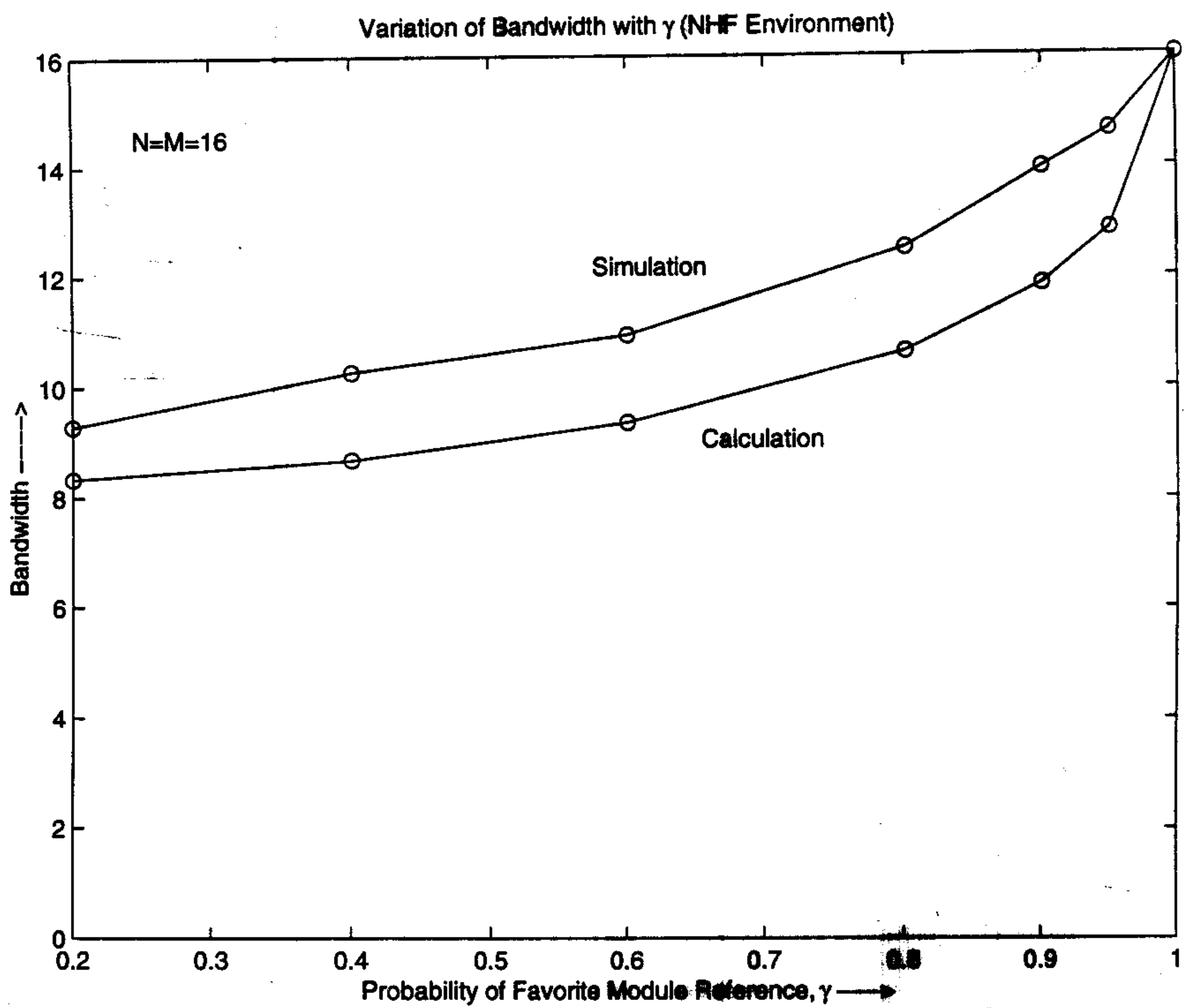


VARIATION OF BANDWIDTH WITH NO. OF HOT MODULES (HNF ENVIRONMENT)

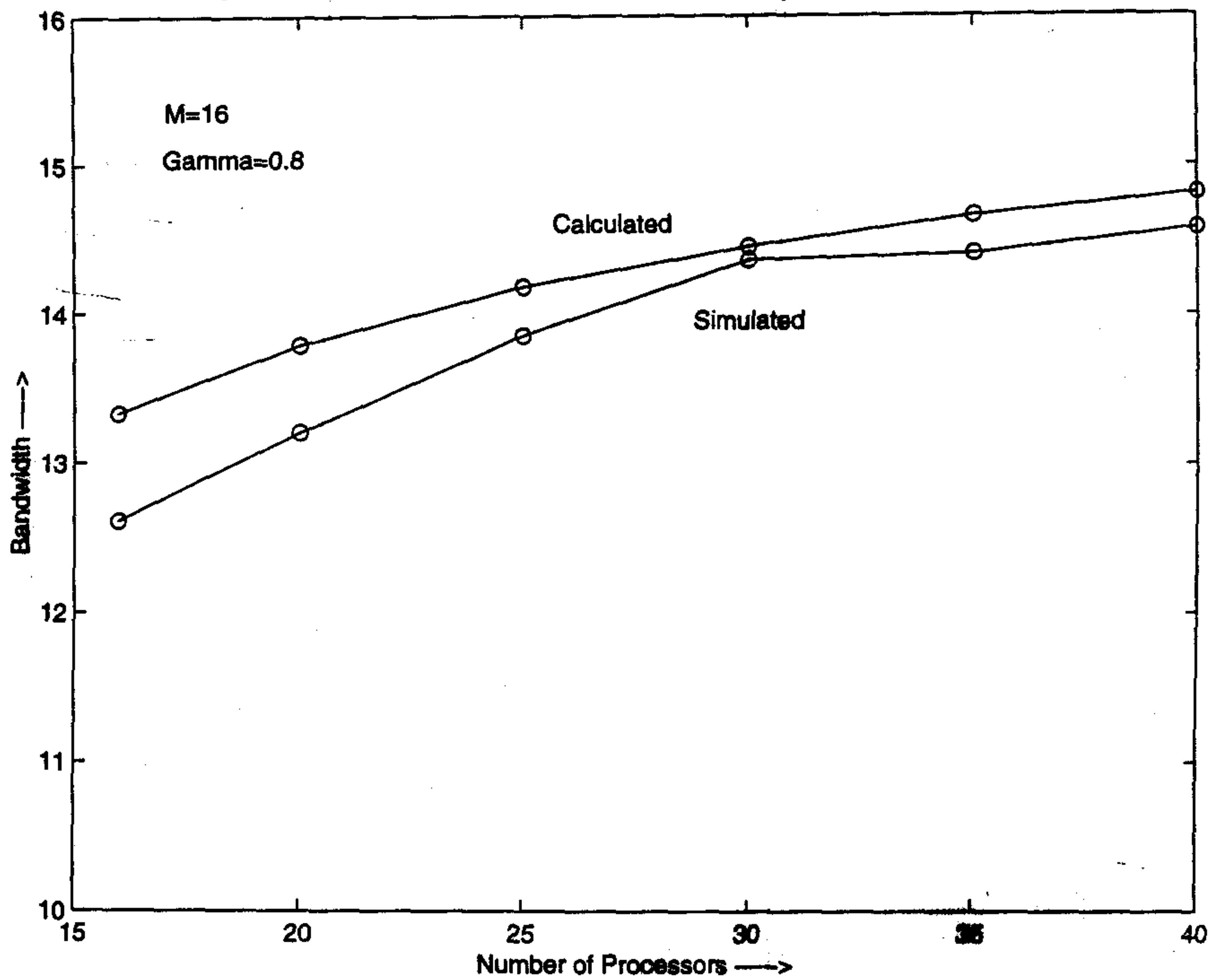


BANDWIDTH AGAINST P_h (HNF ENVIRONMENT)

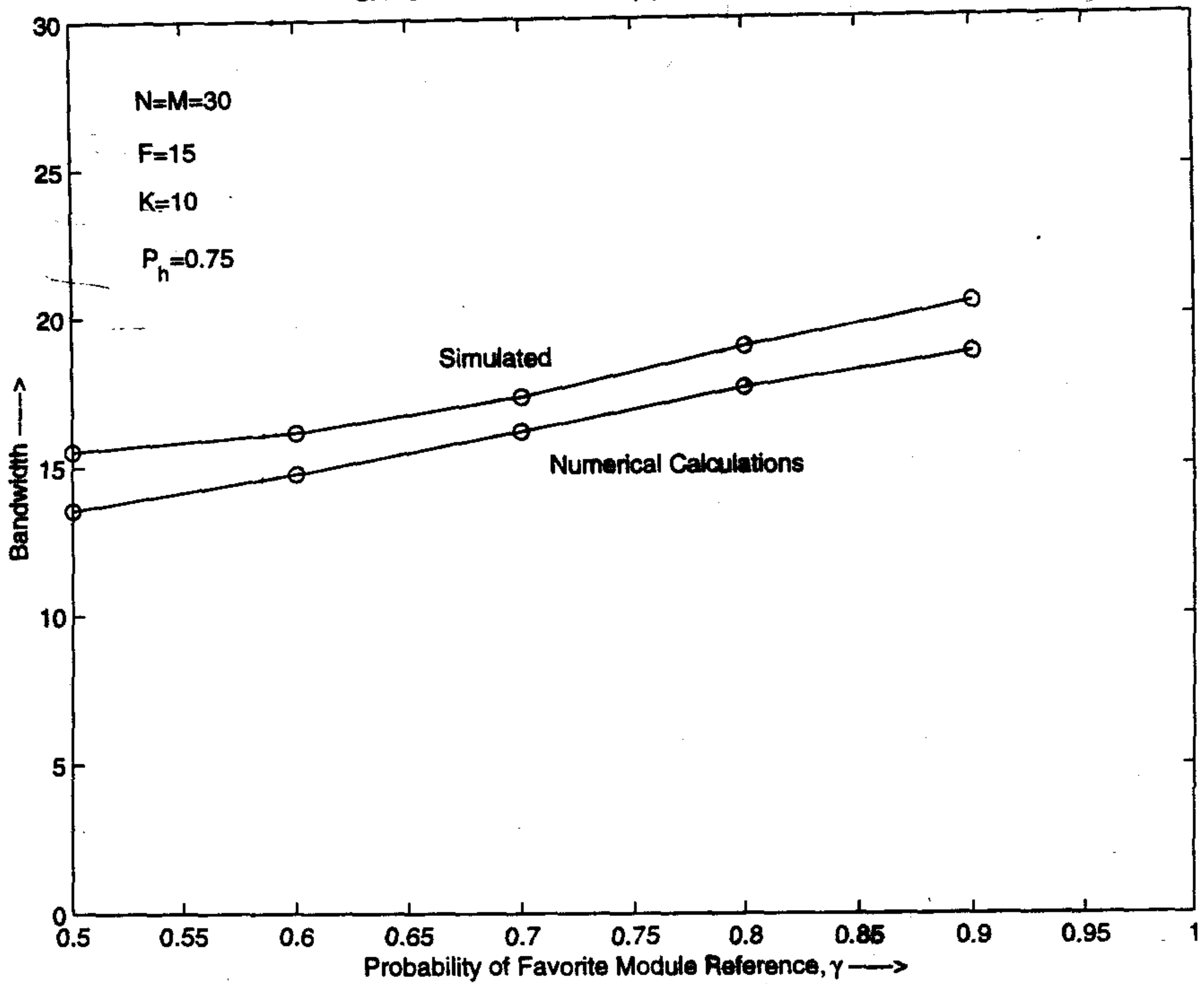




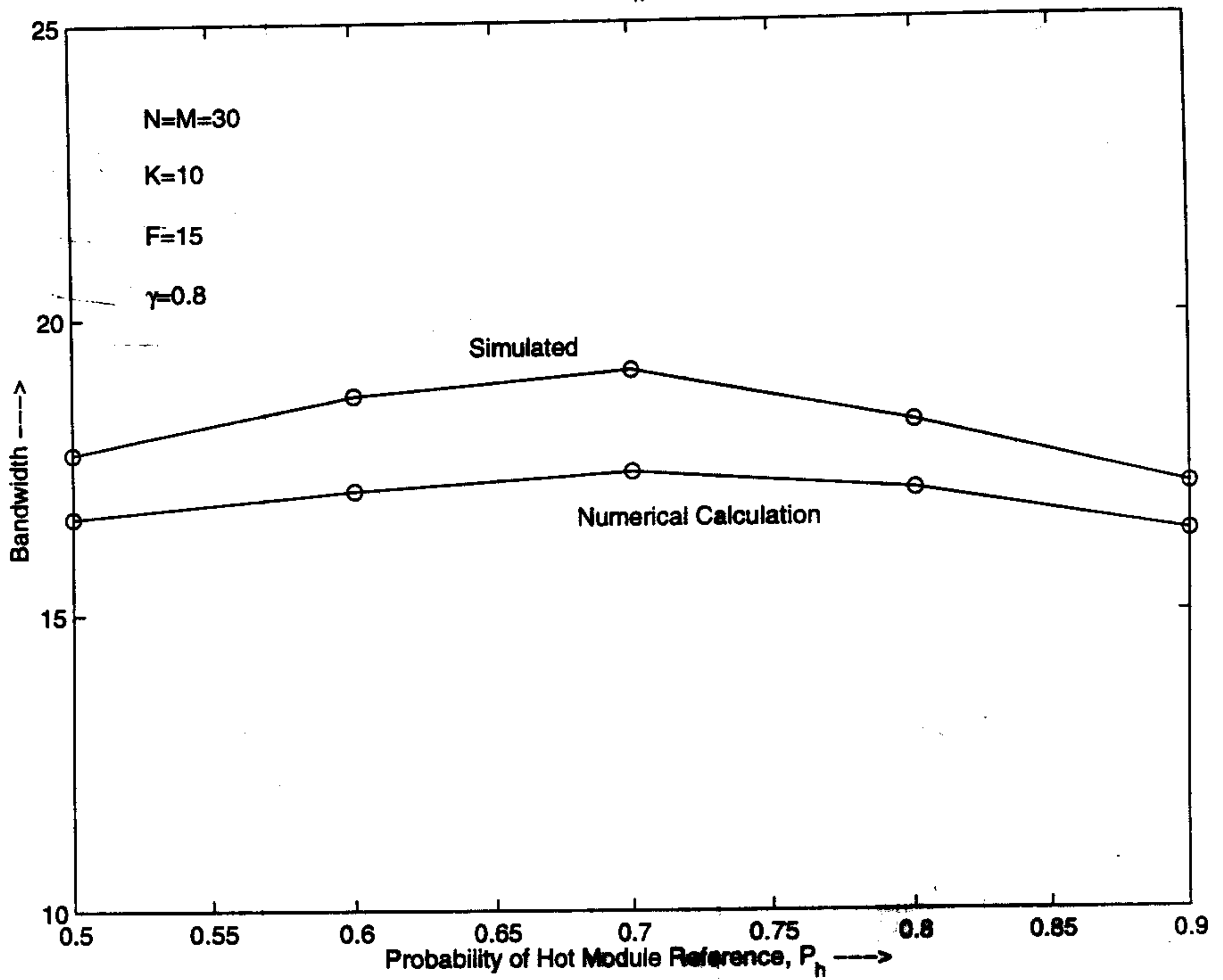
BANDWIDTH AGAINST NO. OF PROCESSORS (NHF ENVIRONMENT)

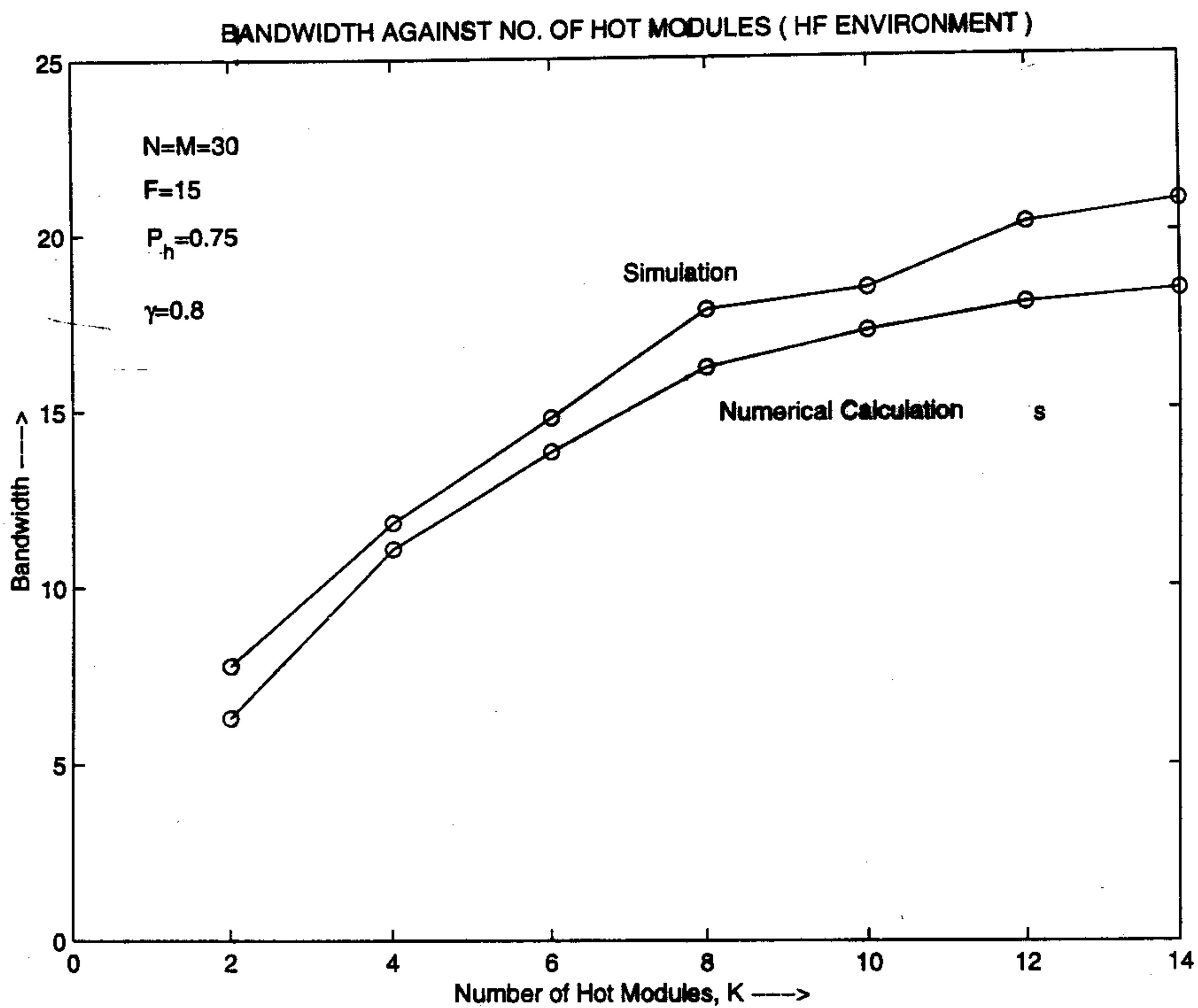


BANDWIDTH AGAINST γ (HF ENVIRONMENT)

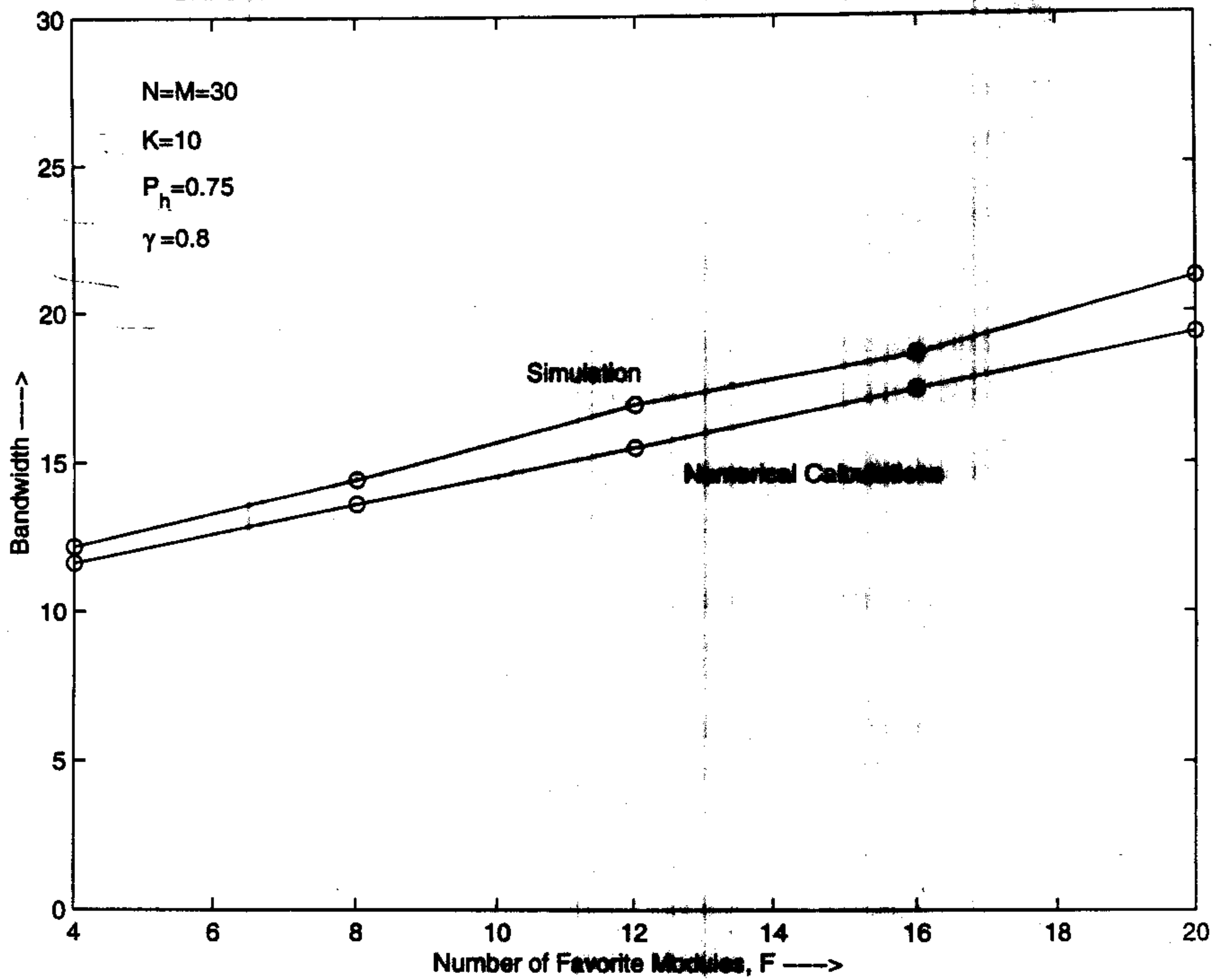


BANDWIDTH AGAINST P_h (HF ENVIRONMENT)





BANDWIDTH AGAINST NO. OF FAVORITE MODULES, F (HF ENVIRONMENT)



Bibliography

- [1] .C.Liu et al. " Effective memory bandwidth and processor blocking probability in multile-bus systems ", IEEE Trans. Comput., Vol. C-36, pp.761-764, June 1987
- [2] .Lang. et al. " Bandwidth of cross bar and multiple bus connections for multiprocessors ", IEEE Trans. Comput., Vol. C-31, pp. 1227-1234, Dec. 1982
- [3] .W.L.Yen, J.H.Patel and E.S.Davidson, " Memory interference in synchronous multiprocessor system ", IEEE Trans. Comput., Vol. C-31, 1116-1121, Nov. 1982
- [4] .R.Das and L.N.Bhuyan, " Bandwidth availability of multi-bus multiprocessors ", IEEE Trans. Comput. Vol. C-34, pp. 918-926, Oct. 1985
- [5] .L.N.Bhuyan , " An analysis of processor-memory interconnection networks ", IEEE Trans. Comput. Vol. C-34, pp. 279-283, March, 1985
- [6] F.Pfister and V.A.Norton , " Hot-spot contention and combining in multistage interconnection networks " ,IEEE Tras. Comput. Vol. C-34, pp. 943-948, Oct. 1985
- [7] .N.Mudge, J.P.Hayes, G.D.Buzzard and D.C.Winsor , " Analysis of multiple bus interconnection networks " , Journal of Parallel and Distributed Computing, Vol.3, pp. 328-313, 1986

- [8] **.K.Sen**, " Analysis of memory interference in buffered multiprocessor systems in presence of hot spots and favorite memories ", Master's thesis, Dept. of Computer Science, University of North Texas, Denton, Aug. 1995
- [9] **.Rajarshi Chowdhary** " Performance Analysis of Shared Memory Multiprocessor Systems ", M.Tech. Dissertation Series, Advanced Computing and Microelectronics Unit, ISI, Calcutta, 1996
- [10] **.Klienrock** , " Queueing systems ", Vol.1, Newyork:John Wiley, 1958