PROCESSOR ALLOCATION AND MESSAGE BROADCASTING IN HYPERCYCLE INTERCONNECTION NETWORKS

by

VASSILIOS V. DIMAKOPOULOS
Diploma, University of Patras, 1990

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of

MASTER OF APPLIED SCIENCE

in the Department of Electrical and Computer Engineering

We accept this thesis as conforming
to the required standard

Dr. N.J. Dimopoulos, Supervisor (Dept. of Elec. & Comp. Eng.)

Dr. K.F. Li, Departmental Member (Dept. of Elec. & Comp. Eng.)

Dr. W. Myrvold, Outside Member (Dept. of Comp. Science)

Dr. R. Vahldieck, Graduate Advisor (Dept. of Elec. & Comp. Eng.)

Dr. D.M. Miller, External Examiner (Dept. of Comp. Science)

© VASSILIOS V. DIMAKOPOULOS, 1992
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part by photocopy or other means, without the permission of the author.

Supervisor: Dr. N.J. Dimopoulos

ABSTRACT

Hypercycles are a class of static multiprocessor interconnection networks with the ability of matching the processing node and communication link restrictions of a large number of applications. This is in contrast with other popular interconnection topologies including hypercubes, k-ary n-cubes and generalized hypercubes. The regular structure of hypercycles allows for simple routing strategies which are responsible for the communication between the processing nodes. Broadcasting a message to all the processors in the network is an essential part of the routing engine and an algorithm that implements it for hypercycles constitutes part of this thesis. The proposed algorithm is analysed and proven to be optimal.

Processor allocation which is the second and most important subject of this study is a major issue for achieving high performance in multiprocessor systems. This theoretical problem has topology-specific solutions and various strategies to deal with it for the hypercycles case are presented here. Comparisons between the proposed strategies are made using both analytical and computer simulation tools. Part of this thesis was the implementation of a simulator to help with these comparisons. Hypercubes, being a subset of hypercycles, can benefit from the proposed strategies which exhibited better performance than the ones with similar complexity proposed in the current literature.

Examiners:

Dr. N.J. Dimopoulos, Supervisor (Dept. of Elec. & Comp. Eng.)

Dr. K.F. Li, Departmental Member (Dept. of Elec. & Comp. Eng.)

Dr. W. Myrvold, Outside Member (Dept. of Comp. Science)

Dr. R. Vahldieck, Graduate Advisor (Dept. of Elec. & Comp. Eng.)

Dr. D.M. Miller, External Examiner (Dept. of Comp. Science)

Table of Contents

Title Page **i**

Abstract **ii**

Table of Contents **iv**

List of Tables **vi**

List of Figures **vii**

1 Introduction **1**
1.1 Multiprocessors 2

2 Hypercycles **6**
2.1 Graphs and interconnection networks 6
2.2 Binary and k-ary hypercubes 8
2.3 Mixed-radix number system 9
2.4 Generalized hypercubes 10
2.5 Hypercycles 11
2.6 Properties of hypercycles 13

3 Broadcasting in Hypercycles **18**
3.1 Broadcasting in $C_m<1, 2, \ldots, \rho>$ 19
3.2 Broadcasting in hypercycles 27

4 The Allocation Problem **30**
4.1 Allocation in hypercubes - buddy strategy 33
4.2 Other strategies for hypercubes 38

5 Allocation in Hypercycles **41**
5.1 First fit strategy 42
5.2 Suboptimal strategy 44
5.3 Multiple permutations 56
5.4 Allocation in uniform hypercycles 64

5.5 Matching the load characteristics 70

6 Conclusion **78**
6.1 Future work 79

Bibliography **81**

A The Simulator **85**

List of Tables

5.1 Delays of all sets of two permutations in Q_3 67
5.2 Delays of various strategies in Q_3 71
5.3 Delays of various strategies in Q_4 71
5.4 Delays of various strategies in Q_5 72
5.5 Delays of various strategies in $G^{1,1,1,1}_{3,3,3,3}$ 72

A.1 Simulation parameters 88
A.2 Simulator settings 90
A.3 User commands 91

List of Figures

2.1 Some hypercubes: Q_1 (a), Q_2 (b), Q_3 (c) 8
2.2 Some hypercycles: G_6^1 (a), G_6^2 (b), $G_{4,3}^{1,1}$ (c), $G_{2,5}^{1,1}$ (d) 12

3.1 The circulants $C_{14}\langle 1\rangle$ (a) and $C_{14}\langle 1,2\rangle$ (b) 20
3.2 Algorithm 1 (all numbers are mod m) 23
3.3 Broadcasting in circulant graphs 26
3.4 Algorithm 2 28
3.5 Broadcasting in the hypercycle $G_{5,4}^{1,1}$ 29

4.1 Subcubes $00*$, $1**$ and $*1*$ in Q_3 33
4.2 The address tree of Q_3 35
4.3 The buddy strategy for Q_n 36
4.4 Allocating nodes in Q_3 with the buddy strategy 37
4.5 The gray code address tree for Q_3 39

5.1 The first fit strategy for hypercycles 44
5.2 Allocation example in $Q_{3,3,2}^{1,1,1}$ using the first fit strategy 45
5.3 The suboptimal strategy for hypercycles 49
5.4 Allocation example in $Q_{3,3,2}^{1,1,1}$ using the suboptimal strategy . . . 51
5.5 The same example using the first fit strategy 51
5.6 Counter-example in $Q_{2,3}^{1,1}$ 52
5.7 Average queueing delay in $G_{3,2}^{1,1}$ using the two strategies 53
5.8 Average utilization in $G_{3,2}^{1,1}$ using the two strategies 53
5.9 Average queueing delay in $G_{3,3,2,2}^{1,1,1,1}$ using the two strategies . . . 54
5.10 Average utilization in $G_{3,3,2,2}^{1,1,1,1}$ using the two strategies 54
5.11 Average queueing delay in $G_{5,4,3,2}^{2,2,1,1}$ using the two strategies . . . 55
5.12 Average utilization in $G_{5,4,3,2}^{2,2,1,1}$ using the two strategies 55
5.13 The multiple first fit strategy for hypercycles 57
5.14 The multiple suboptimal strategy for hypercycles 58
5.15 Allocating nodes with the multiple suboptimal strategy 60
5.16 Average queueing delay in $G_{3,2}^{1,1}$ using multiple permutations . . 62
5.17 Average utilization in $G_{3,2}^{1,1}$ using multiple permutations 62

5.18 Average queueing delay in $G^{2,2,1,1}_{5,4,3,2}$ using multiple permutations . 63
5.19 Average utilization in $G^{2,2,1,1}_{5,4,3,2}$ using multiple permutations 63
5.20 Queueing delay in $G^{2,2,1,1}_{5,4,3,2}$, uniform load 75
5.21 Utilization in $G^{2,2,1,1}_{5,4,3,2}$, uniform load 75
5.22 Queueing delay in $G^{2,2,1,1}_{5,4,3,2}$, gaussian load (mean = 20.0, var = 7.5) 76
5.23 Utilization in $G^{2,2,1,1}_{5,4,3,2}$, gaussian load (mean = 20.0, var = 7.5) . . 76
5.24 Queueing delay in $G^{2,2,1,1}_{5,4,3,2}$, gaussian load (mean = 60.5, var = 7.5) 77
5.25 Utilization in $G^{2,2,1,1}_{5,4,3,2}$, gaussian load (mean = 60.5, var = 7.5) . . 77

Chapter 1

Introduction

The evolution of computer systems throughout their sixty years history has been strongly connected with advancements in technology which in turn have been motivated by the needs that such systems have to satisfy. In the beginning, the simple needs of an old society created a simple computational device. As time progressed, new problems required more complex and efficient systems in order to be dealt with. Supercomputers are a typical example of such a development that is based on the interaction between need and technology.

Today, a growing number of applications are responsible for the next generation of computer systems that are about to appear. Problems that have to be solved include image processing, pattern recognition, fluid mechanics, aerodynamics problems and weather forecasting. They require a breakthrough that is not solely dependent on technology but also on architectural improvement of computer systems ([1]). The simple Von Neumann architecture is no longer appropriate for large-scale computations. Concurrency has to be exploited, instead, both in hardware and software, and this gives rise to the concept of *parallel processing*. Systems that emphasize parallel processing have already been built and others are currently in development. Research is being conducted in order to maximize the use of these systems and to suggest more efficient ones.

1.1 Multiprocessors

Parallel processing is possible within any of the SISD, SIMD and MIMD architectural categories ([1]) in the form of pipelined, array processor and multiprocessor systems respectively. Examples of pipelined computers include the Star-100, the IBM 3838, the Cray-1 and the Cyber 205 ([1]). The Illiac-IV ([2]) and the Massively Parallel Processor ([3]) are typical array processors.

In contrast with the SISD and SIMD machines, *multiprocessors* allow concurrent execution of multiple instructions thus providing a higher level of parallelism. They consist of processing elements (PE's) which communicate and cooperate with each other asynchronously through dedicated lines and/or shared memory modules. The whole system is governed, though, by a single operating system in contrast with distributed systems.

Multiprocessor systems can be characterized as either *tightly* or *loosely coupled*. The interconnection in tightly coupled multiprocessors relies on shared memory while each processor may own a small local memory or cache. A complete connectivity exists between the processors and the memory modules through the use of an interconnection network or multiport memory. On the other hand, the PE's in a loosely coupled multiprocessor do not share a common memory. Each one is actually a complete computer module with local memory and an I/O subsystem. The whole system relies heavily on the interconnection structure between these PE's. Examples of multiprocessor systems include the Univac 1100/94, the Cray X-MP, the Intel iPSC, the Cosmic Cube, and the Connection Machine ([1, 4, 5, 6]).

As implied above, one of the primary issues in the design of multiprocessor systems, especially in the loosely coupled ones, is the interconnection scheme between the processing elements. Bus structures and crossbar switches were some of the early ways to serve this purpose. Bus structures (hierarchical or not) are attractive because of their low cost, but they have acceptable performance

only in the case of a system consisting of a few PE's. Crossbar switches, on the other hand, have the property of yielding the best possible performance but their implementation is quite expensive for more than a few PE's as their cost increases with the square of the number of interconnection points.

More sophisticated and cost-effective ways to interconnect PE's have been proposed and implemented, and will be referred to as *multiprocessor interconnection networks*. They can be divided into two classes: *dynamic* and *static* or *point-to-point*. Dynamic networks have been used in SIMD machines and tightly coupled multiprocessors mainly for the interconnection between the PE's and the (shared) memory modules.

Dynamic networks involve one or more layers of switching elements which can be programmed to provide different paths from their inputs to their outputs. If such a network is used for interprocessor communication, both its inputs and outputs are connected to the set of PE's. Since at any time we have processor pairs that want to communicate, the network actually performs a mapping between the processors, i.e. a permutation. Most of these networks are not capable of implementing all the possible permutations and are characterized as *blocking* networks. The Omega ([7]) and Delta ([8]) are examples of such networks. The Benes network ([9]) is an example of a dynamic network that can have its switching elements reprogrammed so that any particular connection pattern (permutation) is implemented. The disadvantage, though, is that an excessive overhead is needed for setting the switches to the appropriate state.

On the other hand, the asynchronous communication nature of loosely coupled multiprocessors have made static interconnection networks more attractive. Such networks consist of point-to-point dedicated links between pairs of PE's and can be either circuit switched or packet switched, much like large scale and local area computer networks. The PE's communicate by messages which traverse links and intermediate PE's until they reach their destination. Usually the system is assumed to be *homogeneous* (i.e. all PE's have the same capabilities)

and the network symmetric. The network is then characterized by its topology and is modeled by a graph where the vertices denote the PE's and the edges denote the links between the PE's. Popular static interconnection networks include binary and k-ary hypercubes, cube connected cycles, rings, etc.

Hypercycles are a class of static interconnection networks that actually includes hypercubes, rings, and some other topologies as subsets. Hypercubes impose restrictions on the number of PE's that may exist in the system, which makes them inappropriate for many applications. Hypercycles, on the other hand, can adapt to any number of PE's thus providing a more flexible way to built a system. Two problems related to multiprocessor interconnection networks are studied in this thesis, namely broadcasting and processor allocation, and solutions are proposed for the case of hypercycles.

One of the primary issues associated with an interconnection network is the design of algorithms to establish the communication paths between the PE's. This is referred to as *routing*. Given a PE (source) that wants to send a message to another PE (destination), routing is responsible for choosing the intermediate PE's and links that have to be traversed in order for the message to travel from the source to destination, if possible, in the minimum time.

Furthermore, *broadcasting* algorithms are necessary in the case that a PE wants to send a message to all the other PE's in the system. This is of paramount importance for many parallel algorithms that are to be executed on the machine, an example being matrix multiplication and other numerical calculations ([1]). In the case that a PE is about to shut down because of a hardware or software error, broadcasting is also needed in order to notify the rest of the PE's so that the system can be reorganized and continue its correct operation. Broadcasting in hypercycles is one of the two problems studied in this thesis.

Another important issue for achieving high performance in multiprocessor systems is the *allocation problem*. In a multitasking environment the system is expected to receive requests for a number of PE's that each program (or

algorithm, or job, or task) needs to execute on. The way in which processors are actually allocated affects the performance of the whole system especially in the cases where the load is not low. The solution to this problem is closely related to the topology of the interconnection network and the routing scheme associated with it. A number of strategies are presented for the hypercycles and this is the second and most important subject of this thesis.

This study is organized as follows:

Chapter 2 introduces the basic graph notation to be used for describing and analyzing the problems associated with interconnection networks. Several topologies such as binary and k-ary hypercubes, generalized hypercubes and hypercycles are also presented in detail.

Chapter 3 presents a broadcasting algorithm for hypercycles. The algorithm is analyzed and proven to be as efficient as possible.

Chapter 4 is an introduction to the processor allocation problem and the terminology associated with it. A simple allocation method that has been proposed for the hypercubes is also described.

Chapter 5 proposes a number of allocation strategies for the hypercycles. Their performance is analyzed and evaluated through the use of both analytical and computer simulation tools.

Chapter 6 concludes this study and discusses future work on hypercycles.

Finally, **Appendix A** presents the simulator we built to study the behavior of the various allocation strategies for the hypercycles.

Chapter 2

Hypercycles

In this section we present some important interconnection topologies: the binary hypercubes, the k-ary n-cubes, the generalized hypercubes and the hypercycles. This thesis is focused on the latter but since hypercubes and generalized hypercubes are subclasses of hypercycles, the theory developed in the next chapters applies to all three topologies. Since these topologies are defined and analyzed using graph theoretic terms, we summarize some important characteristics of graphs in the first section of this chapter. Section 2.2 defines the hypercubes. Section 2.3 introduces the mixed-radix number system which forms the basis for presenting the generalized hypercubes (section 2.4) and the hypercycles (section 2.5). Section 2.6 discusses the properties of hypercycles in more detail.

2.1 Graphs and interconnection networks

A static multiprocessor interconnection network can be modeled by a graph and many of its theoretical properties can be derived from this model. The PE's in such a network are assumed to be identical and will correspond to the graph vertices. The links between the PE's will correspond to the edges of the graph. These links are assumed to be half or full duplex communication lines, so that the use of an undirected graph is justified. The number of I/O ports in each PE, an important cost factor of the system, is equal to the degree of the

corresponding vertex in the graph. One of the measures for determining the efficiency of such a network is the maximum delay that a message has to suffer when traveling between two PE's and this is dependent on the diameter of the underlying graph ([10, 11]). We thus give a summary of the terminology and characteristics associated with graphs that will be useful in the next sections.

A *graph* ([12, 13]) is defined as a set of *vertices* V, interconnected by a set of *edges* E, symbolized as $G = (V, E)$. If the edges have no direction, the graph is called *undirected*, otherwise it is called *directed*. In this thesis we deal only with undirected graphs, so the term 'graph' implies an undirected one.

The edge e that connects vertices v_i and v_j is written as $e = (v_i, v_j)$ and is said to be *incident with* v_i and v_j. If $(v_i, v_j) \in E$, then v_i and v_j are *adjacent to* each other. The *degree* of a vertex v, denoted by $d(v)$, is the number of vertices v is adjacent to. If all the vertices in a graph have the same degree, the graph is called *regular* and the degree of the graph, denoted by $d(G)$, is the degree of each node.

A *path* from v_1 to v_i is a sequence of distinct vertices $P = v_1, v_2, ..., v_i$ such that for every $1 \leq j < i$, the edge (v_j, v_{j+1}) is in E. A *cycle* or *circuit* consists of a path and an additional edge incident with the first and last node of the path. The *length* of a path P, denoted as $|P|$, is equal to the number of edges it contains, which is equal to the number of vertices minus one.

The distance between v_i and v_j, $dist(v_i, v_j)$, is the minimum of the lengths of all the possible paths between v_i and v_j. The *diameter*, D, of the graph is defined as the maximum distance between any pair of nodes, i.e.

$$D = \max\{dist(v_i, v_j) : \quad v_i, v_j \in V\}$$

From now on, we are going to use the term *processing node*, or just *node* to denote either a PE or the corresponding vertex of the underlying graph. Every node will be labeled by a unique name, called the *address* of the node. The set of nodes will be equivalent to the set of their addresses. Finally, the

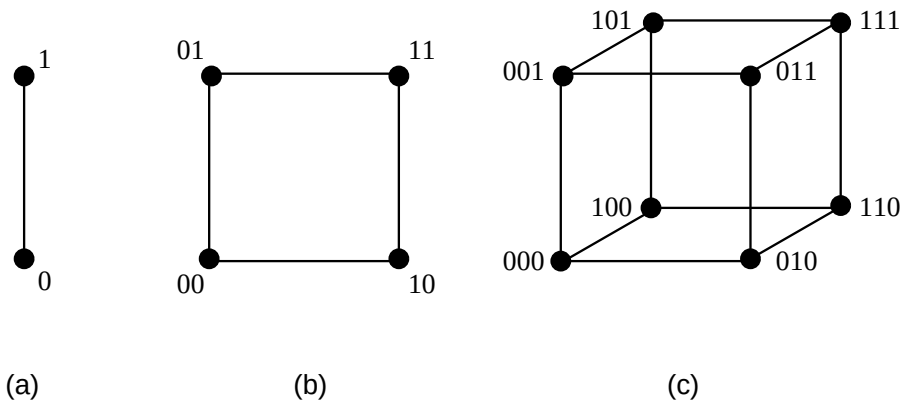


Figure 2.1: Some hypercubes: Q_1 (a), Q_2 (b), Q_3 (c)

communication links will be referred to as either *links* or *edges*.

2.2 Binary and k-ary hypercubes

A *binary hypercube*, or n-cube ([14, 15]), is a graph $Q_n = (V, E)$ with 2^n nodes. The addresses of the nodes range from 0 to $2^n - 1$, hence $V = \{0, 1, ..., 2^n - 1\}$. The edge set is defined by writing the address of each node in the binary representation; two nodes are adjacent if their binary representations differ in only 1 bit. If $a = (a_n a_{n-1} ... a_1)_2$ and $b = (b_n b_{n-1} ... b_1)_2$ then $(a, b) \in E$ if and only if:

$$\begin{array}{ll} \exists j,\ 1 \leq j \leq n, \text{ such that:} & a_j \neq b_j \\ \text{and } \forall i,\ 1 \leq i \leq n,\ i \neq j & a_i = b_i \end{array}$$

Binary hypercubes are regular graphs, the degree of each node being equal to n. The diameter is equal to n and the number of edges is $n2^{n-1}$. A number of n-cubes are shown in Figure 2.1.

If we allow k ($k > 2$) nodes in every dimension of an n-cube, where each node is adjacent to its two closest neighbors in each dimension, we obtain a k-ary n-cube ([14, 16]), a generalization of the binary hypercube. The edge set is defined by writing the address of each node in base k representation. If $a = (a_n a_{n-1} ... a_1)_k$ and $b = (b_n b_{n-1} ... b_1)_k$ then $(a, b) \in E$ if and only if:

$$\begin{array}{ll} \exists j,\ 1 \leq j \leq n, \text{ such that:} & a_j = b_j \pm 1 (\mathrm{mod} k) \\ \text{and } \forall i,\ 1 \leq i \leq n,\ i \neq j & a_i = b_i \end{array}$$

The k-ary n-cubes are regular graphs, with k^n nodes and degree $2n$. Their diameter is equal to $kn/2$ and the number of edges is nk^n.

In this thesis, we use the term *hypercube* to denote a binary hypercube and *k-ary hypercube* to denote a k-ary n-cube.

2.3 Mixed-radix number system

A number $x_r x_{r-1} \ldots x_1$ in base b representation (denoted as $(x_r x_{r-1} ... x_1)_b$) represents the decimal number

$$x_r \times b^{r-1} + x_{r-1} \times b^{r-2} + \cdots + x_2 \times b + x_1$$

where b^{i-1} is referred to as the weight of the i-th digit. As an example, in the binary number system ($b = 2$), we have: $(1011)_2 = 1 + 1 \times 2 + 0 \times 2^2 + 1 \times 2^3 = 1 + 2 + 8 = 11_{10}$.

The *mixed-radix system* ([17, 18]) is a generalization of the base b representation, where each digit of a number may have a different base. Given a number N, factored into r factors $m_1, m_2, ..., m_r$ as

$$N = m_r \times m_{r-1} \times \cdots \times m_1,$$

any number x, $0 \leq x \leq N - 1$, can be written as an r-tuple

$$(x_r x_{r-1} \ldots x_1)_{m_r m_{r-1} \ldots m_1}$$

where m_i is the base of x_i and $0 \leq x_i \leq m_i - 1$ $(i = 1, 2, ..., r)$. The corresponding decimal number can be found as follows:

$$x = \sum_{i=1}^{r} x_i w_i$$

where $w_i = m_{i-1} \times m_{i-2} \times \cdots \times m_1$, the *weight* of the i-th digit (w_1 is always 1). As an example, let $N = 24 = 3 \times 4 \times 2 = m_3 \times m_2 \times m_1$. Then $w_1 = 1, w_2 = 2, w_3 = 8$, and

$$(231)_{3,4,2} = 2 \times w_3 + 3 \times w_2 + 1 \times w_1 = 1 + 6 + 16 = 23$$

If the bases of the mixed radix system are clear from context, we are going to omit them in order to simplify the notation, i.e.

$$(x_r x_{r-1} \ldots x_1) \equiv (x_r x_{r-1} \ldots x_1)_{m_r m_{r-1} \ldots m_1}$$

Finally, we note that if $m_1 = m_2 = \cdots = m_r = b$, then the corresponding weights become $w_i = m_{i-1} \times m_{i-2} \times \cdots \times m_1 = b^{i-1}$, i.e. we obtain the standard base b representation.

2.4 Generalized hypercubes

A generalized hypercube ([18]) is a graph $G = (V, E)$ with N nodes where $N = m_r \times m_{r-1} \times \cdots \times m_1$, generating a mixed radix system. The addresses of the nodes range from 0 to $N - 1$ hence $V = \{0, 1, \ldots, N - 1\}$. The edge set is defined by representing the address of each node in the mixed-radix system generated by the factors of N. If $a = (a_r a_{r-1} \ldots a_1) \in V$ and $b = (b_r b_{r-1} \ldots b_1) \in V$, then $(a, b) \in E$ if and only if:

$$\begin{array}{ll} \exists j,\ 1 \leq j \leq r, \text{ such that:} & a_j \neq b_j \\ \text{and } \forall i,\ 1 \leq i \leq r,\ i \neq j & a_i = b_i \end{array}$$

Generalized hypercubes are a superset of hypercubes. For $N = 2^r$, factored in r factors as $N = 2 \times 2 \times \cdots 2 = 2^r$, we obtain Q_r. Generalized hypercubes are regular graphs with degree

$$d = \sum_{i=1}^{r} (m_i - 1)$$

since for every given node there are $m_i - 1$ other nodes that differ in the i-th digit. The diameter of a generalized hypercube is equal to r and the number of edges is $(dN)/2$.

Compared to binary hypercubes, generalized hypercubes are more flexible since they allow interconnections that are not restricted to a number of nodes that is a power of 2. Although the diameter is low, the large degree of these graphs makes them difficult to realize as practical multiprocessor networks.

2.5 Hypercycles

A hypercycle ([19, 20, 21]) is a graph $G^{\rho_r,\rho_{r-1},\ldots,\rho_1}_{m_r,m_{r-1},\ldots,m_1} = (V, E)$ with N nodes where N is factored as $N = m_r \times m_{r-1} \times \cdots \times m_1$, generating a mixed radix system. The addresses of the nodes range from 0 to $N-1$ hence $V = \{0, 1, \ldots, N-1\}$. After representing the address of each node in the mixed-radix system generated by the factors of N, the edge set is determined by $\rho_1, \rho_2, \ldots, \rho_r$, where

$$\rho_i \leq m_i/2,\ i = 1, 2, \ldots, r$$

as follows: if $a = (a_r a_{r-1} \ldots a_1) \in V$ and $b = (b_r b_{r-1} \ldots b_1) \in V$, then $(a, b) \in E$ if and only if:

$$\begin{array}{cc} \exists \xi_j, 1 \leq \xi_j \leq \rho_j,\ 1 \leq j \leq r: & b_j = (a_j \pm \xi_j) \bmod m_j \\ \text{and } \forall i,\ 1 \leq i \leq r,\ i \neq j & a_i = b_i \end{array} \tag{2.1}$$

Figure 2.2 shows some examples of different hypercycles.

Hypercycles are a superset of generalized hypercubes (hence of binary hypercubes also). If $\rho_i = \lfloor m_i/2 \rfloor$, for all $i = 1, 2, \ldots, r$, then we obtain the generalized hypercube that uses the same factoring of N. For $m_1 = m_2 = \cdots = m_r = 2$ we get Q_r. The k-ary hypercubes are also a subset of hypercycles, the k-ary r-cube obtained for $m_1 = m_2 = \cdots = m_r = k$ and $\rho_1 = \rho_2 = \cdots = \rho_r = 1$. As compared to generalized hypercubes, hypercycles allow a richer set of interconnections, ranging in complexity from the simple rings ($\rho_i = 1$) to the fully connected ones used in the generalized hypercubes. In addition to matching the node requirements (as opposed to hypercubes), they can also match the link constraints (as opposed to generalized hypercubes) of different applications.

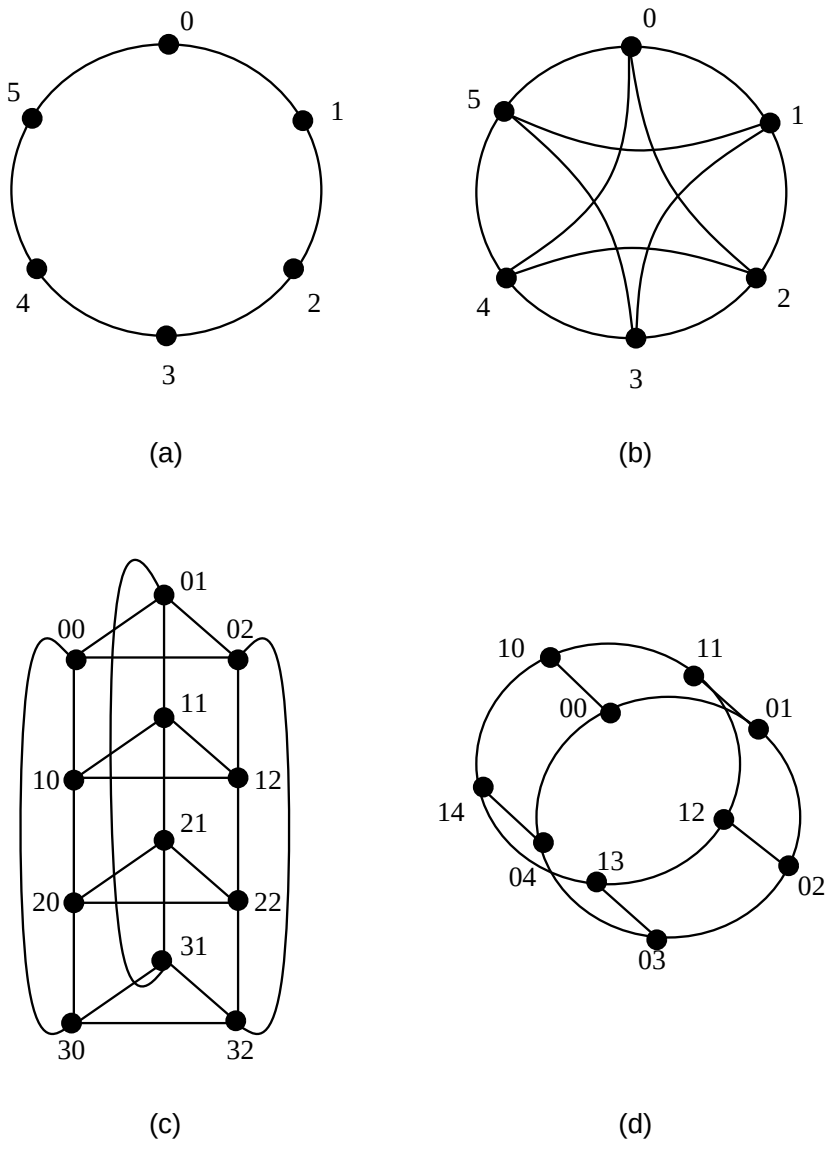


Figure 2.2: Some hypercycles: G_6^1 (a), G_6^2 (b), $G_{4,3}^{1,1}$ (c), $G_{2,5}^{1,1}$ (d)

2.6 Properties of hypercycles

Hypercycles consist of r dimensions with m_i nodes in the i-th dimension. Dimension i constitutes a circulant graph ([22]) $C_{m_i}<1, 2, \ldots, \rho_i>$ which is identical to a graph presented by Harary (quoted in [23]). The hypercycle structure is a product of these simple graphs ([19]).

The circulant graph $C_{m_i}<1, 2, \ldots, \rho_i>$, where $\rho_i \leq m_i/2$, is defined as an m_i-node graph where node j is adjacent to nodes

$$j \pm 1, j \pm 2, \ldots, j \pm \rho_i \,(\mathrm{mod} m_i)$$

Note that the above definition is in accordance with equation 2.1 assuming that the hypercycle has only one dimension ($r = 1$).

For the above type of graph we know ([23, 22]) that it is regular with degree d_i and diameter D_i equal to

$$d_i = \begin{cases} 2\rho_j & \text{if } \rho_j \neq m_j/2 \\ 2\rho_j - 1 & \text{if } \rho_j = m_j/2 \end{cases} \tag{2.2}$$

$$D_i = \lceil \frac{\lfloor m_i/2 \rfloor}{\rho_i} \rceil \tag{2.3}$$

Every node in an r-dimensional hypercycle belongs to r such circulants thus making hypercycles regular graphs with degree ([19])

$$d(G^{\rho_r,\rho_{r-1},\ldots,\rho_1}_{m_r,m_{r-1},\ldots,m_1}) = \sum_{i=1}^{r} d_i \tag{2.4}$$

and diameter

$$D = \sum_{i=1}^{r} D_i \tag{2.5}$$

where d_i and D_i are given by equations 2.2 and 2.3 respectively.

Below, we briefly discuss the routing procedure in hypercycles which is similar to the ones used for hypercubes ([14]) and generalized hypercubes ([18]). At first we consider routing in one dimension. Assume that the distance between nodes $a = (a_r a_{r-1} \ldots a_i \ldots a_1)$ and $a^* = (a_r a_{r-1} \ldots a_i^* \ldots a_1)$, is equal to

l. Define

$$|x_i, y_i| \triangleq \min\{(x_i - y_i) \bmod m_i, (y_i - x_i) \bmod m_i\}$$

and let $a_i^1 = a_i, a_i^l = a_i^*$. Then a path of length l between a and a^* is constructed as follows:

$$\begin{aligned} a = \quad & (a_r a_{r-1} \ldots a_i^1 \ldots a_1), (a_r a_{r-1} \ldots a_i^2 \ldots a_1), \cdots, \\ & (a_r a_{r-1} \ldots a_i^{l-1} \ldots a_1), (a_r a_{r-1} \ldots a_i^l \ldots a_1) \qquad = a^* \end{aligned}$$

such that

$$a_i^{j+1} = \begin{cases} (a_i^j + \rho_i) \bmod m_i & \text{if } [(a_i^l - a_i^j) \bmod m_i = |a_i^j, a_i^l|] > \rho_i \\ (a_i^j + |a_i^j, a_i^l| \bmod \rho_i) \bmod m_i & \text{if } [(a_i^l - a_i^j) \bmod m_i = |a_i^j, a_i^l|] > \rho_i \\ & \text{and } |a_i^j, a_i^l| \bmod \rho_i \neq 0 \\ (a_i^j - \rho_i) \bmod m_i & \text{if } [(a_i^l - a_i^j) \bmod m_i = |a_i^j, a_i^l|] > \rho_i \\ (a_i^j - |a_i^j, a_i^l| \bmod \rho_i) \bmod m_i & \text{if } [(a_i^l - a_i^j) \bmod m_i = |a_i^j, a_i^l|] > \rho_i \\ & \text{and } |a_i^j, a_i^l| \bmod \rho_i \neq 0 \\ a_i^l & \text{if } |a_i^j, a_i^l| \leq \rho_i \end{cases}$$

As explained in [20, 21] the first and third of the above cases constitute a greedy strategy where the maximum step towards the destination is taken. The second and fourth cases form alternate paths by allowing the step in the fifth case to be taken earlier. The above equation will be called the *routing equation* of the hypercycles.

Multidimensional routing involves routing through a single dimension and then selecting another dimension in which the source and destination differ to route through. For example a minimum length path from $a = (a_r a_{r-1} \ldots a_2 a_1)$ to $b = (b_r b_{r-1} \ldots b_2 b_1)$ can be formed as follows

$$\begin{aligned} a = \quad & (a_r a_{r-1} \ldots a_i \ldots a_j \ldots a_1), \\ & (a_r a_{r-1} \ldots x_1 \ldots a_j \ldots a_1), \\ & (a_r a_{r-1} \ldots x_1 \ldots y_1 \ldots a_1), \\ & (a_r a_{r-1} \ldots x_2 \ldots y_1 \ldots a_1), \\ & \ldots \\ & (a_r a_{r-1} \ldots x_2 \ldots b_j \ldots a_1), \\ & \ldots \\ & (b_r b_{r-1} \ldots b_i \ldots b_j \ldots b_1) \qquad = b \end{aligned}$$

where in each of the above lines we route through a single dimension as described previously. We sequentially modify the source address (a), each time substituting a source digit by an intermediate path digit, until we form the destination address (b).

Assuming that $dist(a,b) = l = \sum_{i=1}^{r} l_i$, where l_i is the distance between a and b in dimension i, the above strategy yields in total

$$\frac{l!}{l_1!l_2!\cdots l_r!} \text{ paths.} \tag{2.6}$$

If we choose one of the the above paths at random then the undesirable situation of a *deadlock* may easily occur. Deadlocks ([24, 25, 26]) arise in computer networks when for some reason messages cannot be forwarded to their destination. A direct consequence of such a situation is the huge delays and the performance degradation the system experiences. Deadlocks have different forms depending on the type of communication policy implemented: in a *circuit-switched* network, the complete path has to be allocated to establish communication between a source and a destination node. All the links in the path are then dedicated to these two nodes and are released as soon as the source node decides that the interaction is complete. In a *message-switched* network on the other hand, only one line is dedicated at a time and the message is passed to the next node and stored in a message queue. As soon as the next link is available, the message moves to the next node, and the procedure is repeated until the message reaches its destination. Deadlocks may then occur in the following cases:

- *Circuit-switching case:* a partially-completed path requires a number of edges currently used by another path which in turn requires edges of the first path in order to complete.

- *Message-switching case:* all the message queues are filled up in nodes which lie in a cycle, and each of these nodes wants to communicate with a node other than the next one.

Certain strategies have been proposed to eliminate deadlocks, either by *preventing* them from happening or by *avoiding* them when they occur. For message-switched networks store-and-forward ([25]) and wormhole ([27]) routing are routing schemes that are designed to work for arbitrary networks and provide means of dealing with deadlocks. The *virtual channels* technique emulates multiple links on a single physical link with each of the 'virtual' links having a message queue associated with it. When such a technique is used in conjunction with wormhole routing and the 'virtual' links are ordered, a deadlock-free routing scheme results ([27]). The above routing strategies usually include a higher hardware and software overhead as compared to topology-specific routing strategies.

For hypercubes, the e-cube routing ([14, 27]) which can be applied to both circuit- and message-switched lines, selects only one of the paths in equation 2.6 by ordering the dimensions of the hypercube in ascending order. This way it can be been proven to be deadlock-free. Hypercycles are currently circuit-switched and e-cube routing cannot be applied because of the cyclic nature of the circulant graphs in each dimension. Nevertheless, a modified version can be applied to certain hypercycles ([21]) thus providing a deadlock-free communication between the processing nodes.

The disadvantage of e-cube routing is that it limits the number of paths between two nodes to exactly one, although we saw that there are more than that. To overcome this limitation, a *backtrack-and-retry* routing scheme has been proposed, where the path that is actually going to be used is chosen at random. Deadlocks are then *avoided* as follows: if an intermediate node finds out that the next link required to form the chosen path is busy, we return to a previous node and try again a possibly different path. The simplest version involves returning back to the *source* node. Other possibilities are currently being examined, like backtracking to a random intermediate node of the path, or utilizing the e-cube, where this is possible, for the longer paths and the

backtracking strategy for the shorter paths ([21]). In another study ([28]) the backtrack-to-the-origin-and-retry strategy was implemented on a VLSI chip and exhibited excellent performance.

Chapter 3

Broadcasting in Hypercycles

Transmitting a message from a single node to all the other nodes in the network is called *broadcasting*. As noted in the Chapter 1, the presence of a fast broadcasting method is necessary for the efficient operation of a parallel system. System fault tolerance requires that each processor is aware of the operating state of the other processors. Operating system primitives must be available, at least in part, in all the processors in the machine. Some parallel algorithms such as matrix multiplication include broadcasting as one of the key operations (see for example pages 355-361 in [1]).

Usually the broadcasting algorithm involves attaching a number to the message, called *weight*, and passing it to adjacent nodes. The receiving nodes keep the message and decrease the weight, then transmit the message to other nodes with the new weight. The procedure goes on until some nodes receive a weight of 1 and stop transmitting the message. This is the idea behind the broadcasting algorithms that have been presented for hypercubes [14] and generalized hypercubes [18]. In this section we follow the same principle to develop a broadcasting algorithm for hypercycles.

Broadcasting should be *non-redundant*. This means that a node that already received the message should not receive it again. Broadcasting should also be fast. If the diameter of the graph is D, then in order for the message to

travel from the source node to the furthest node it should go through a path of maximum length D. Broadcasting cannot be completed in less than D steps and a fast broadcasting algorithm should achieve this lower limit.

Section 3.1 presents an algorithm for broadcasting in the circulant graphs that represent a single dimension in hypercycles as described in section 2.6. Then the algorithm is extended for r-dimensional hypercycles in section 3.2. The algorithms are shown to be non-redundant and require the minimum number of steps to complete.

3.1 Broadcasting in $C_m<1, 2, \ldots, \rho>$

From equation 2.3 we know that the diameter of such a circulant graph is equal to

$$D = \lceil \frac{\lfloor m/2 \rfloor}{\rho} \rceil \tag{3.1}$$

For example $C_{14}<1>$, which is a simple ring, in Figure 3.1(a) has a diameter of 7, while $C_{14}<1, 2>$ in Figure 3.1(b) has a diameter of 4. Consider now the first graph and assume that node 0 wants to broadcast a message. To guarantee delivery of the message in 7 steps at most, the message has to be sent both to node 1 and node 13 at first. Node 1 will then transmit it to node 2, and so on, while node 13 will transmit it to node 12, etc. After 6 steps, the message will reach nodes 6 (in the clockwise direction) and 8 (in the counterclockwise direction). At this point, only one of those two nodes should continue the transmission, so that node 7 does not receive the message twice. Arbitrarily, we select node 6 to continue and node 8 to stop, so in the 7th step the broadcasting is complete. In order to make a node aware about when to stop transmitting, we attach a weight to the message. Any node that receives the message checks to see if the weight is equal to 1. If it is equal to 1 the node stops transmitting. Otherwise, it reduces the weight by 1 and transmits the message with the new

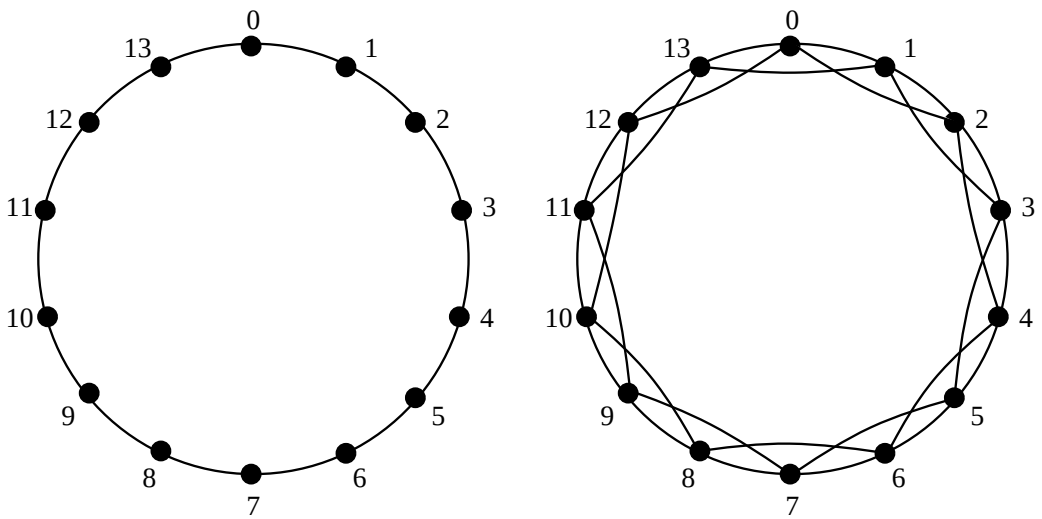


Figure 3.1: The circulants C_{14}<1> (a) and C_{14}<1, 2> (b)

weight to the next node. One can see that node 0 has to give a weight of 7 to node 1 and a weight of 6 to node 13 in order for the broadcasting to complete in 7 steps and for node 7 to receive the message only once.

An observation we can now make is that we can always give weight equal to the diameter, D, of the graph to the neighboring nodes on the clockwise direction since this way we reach the furthest node(s) in exactly D steps and this is the best we can have. We can then select appropriate weights for the adjacent nodes in the counterclockwise direction. Consider now the second circulant in Figure 3.1(b). Since for this graph $D = 4$, we give weight 4 to nodes 1 and 2. Now, these nodes should transmit the message as far as possible and only to one of the nodes they are adjacent to. So node 1 sends it to node 3 and node 2 sends it to node 4. Continuing this way, after exactly 4 steps, nodes 7 and nodes 8 will be receiving the message from nodes 5 and 6 respectively. The weight will at that point be equal to 1, so transmission in the clockwise direction stops. At the same time, though, the message is propagating in the counterclockwise

direction and we must make certain that it reaches up to node 9, so that we do not have redundant recipients. By looking carefully at the graph we can see that nodes 13 and 12 should not receive the same weight from the originating node 0. Node 13 should receive a weight of 3 so that the transmission stops at node 9. Node 12 should should receive a weight of 2 so that the transmission stops at node 10. We now proceed with the general case.

Assume that in the circulant $C_m<1,2,\ldots,\rho>$ the nodes are numbered 0 to $(m-1)$ in a clockwise manner and node 0 wants to broadcast a message to the other nodes. Begin by giving weight D to nodes $1, 2, ..., \rho$ in the clockwise direction. After exactly D steps, nodes $(D-1)\rho+1, (D-1)\rho+2, \ldots, (D-1)\rho+(\rho-1), D\rho$ will be receiving the message with weight 1 and the transmission stops in this direction. Assume, now that node 0 sent in the beginning the message with weight a to nodes $m-1, m-2, \ldots, m-\rho$ in the counterclockwise direction. Then after exactly a steps, the nodes $m-(a-1)\rho-1, m-(a-1)\rho-2, \ldots, m-a\rho$ will be receiving the message.

We now want to make sure that no redundant transmissions are made, in other words, the counterclockwise paths should not meet the clockwise paths. From the above description, after D steps the furthest node we will reach in the clockwise direction is node $D\rho$ as shown below:

$$0 \rightarrow \rho \rightarrow 2\rho \rightarrow \cdots \rightarrow D\rho$$

At the same time, after a steps, the furthest node we reach visiting nodes in the counterclockwise direction from node 0, is node $m-a\rho$:

$$0 \rightarrow m-\rho \rightarrow m-2\rho \rightarrow \cdots \rightarrow m-a\rho$$

What we have to make certain, then, is that $m-a\rho$ is greater than $D\rho$. Solving for the *maximum* integer value of a, we obtain:

$$m-a\rho > D\rho \Rightarrow \tag{3.2}$$

$$\Rightarrow a < \frac{m}{\rho} - D \Rightarrow a = \begin{cases} \frac{m}{\rho} - 1 - D & \text{if } m/\rho \text{ is integer} \\ \lfloor \frac{m}{\rho} \rfloor - D & \text{if } m/\rho \text{ is not integer} \end{cases}$$

or equivalently,

$$a = \lfloor \frac{m-1}{\rho} \rfloor - D$$

Lastly the broadcasting must be complete, i.e. all the nodes receive the message. One can see that the transmission in the clockwise direction stopped at node $D\rho$ while in the opposite direction, it stopped at node $m - a\rho$. Between nodes $D\rho$ and $m - a\rho$ there are exactly $k = m - a\rho - D\rho - 1$ nodes. Hence, after the a-th step, k nodes should continue the transmission for one more step in the counterclockwise direction to let the above nodes receive the message. This can be accomplished by giving, in the beginning, weights of $(a+1)$ to the k closest to the origin nodes in the counterclockwise direction. The remaining nodes receive a weight of a. We note that k always has a legal value, i.e. $k < \rho$ because

$$k = m - a\rho - D\rho - 1 = m - (\lfloor \frac{m-1}{\rho} \rfloor - D)\rho - D\rho - 1 = (m-1) - \lfloor \frac{m-1}{\rho} \rfloor \rho$$

and since $m - 1$ can be written as $m - 1 = q\rho + r \quad (0 \leq r < \rho)$, we have

$$k = q\rho + r - q\rho = r < \rho$$

Based on the above analysis, we give a non-redundant broadcasting algorithm for circulant graphs shown in Figure 3.2, which is completed in exactly D steps. The numbers in the algorithm are all mod m and, a and k have the following values:

$$a = \lfloor \frac{m-1}{\rho} \rfloor - D \tag{3.3}$$

$$k = m - a\rho - D\rho - 1 \tag{3.4}$$

Note that Algorithm 1 is clockwise preferential because of Case 1(a). Algorithm 1 can be written to be counterclockwise preferential by simply interchanging the occurrence of the terms 'clockwise' and 'counterclockwise'. The

Case 1: ORIGINATING NODE (suppose node i)

(a) Transmit the message with weight D to all ρ nodes in the clockwise direction, i.e. nodes $i+1, i+2, \ldots, i+\rho$

(b) Transmit the message with weight $(a+1)$ to the first k nodes counterclockwise, i.e. nodes $i-1, i-2, \ldots, i-k$

(c) Transmit the message with weight a to the remaining $\rho - k$ nodes (counterclockwise), i.e. nodes $i-(k+1), i-(k+2), \ldots, i-\rho$

Case 2: THE OTHER NODES

(Suppose wc is the received weight)

If $(wc = 1)$

then Stop

else
- $wc \leftarrow wc - 1$
- Transmit the message with the new weight wc to the node at distance ρ in the direction the message was received

Figure 3.2: Algorithm 1 (all numbers are mod m)

properties of the algorithms are identical because of the rotational invariance of the circulant graphs.

Theorem 3.1 *The broadcasting described in Algorithm 1 is non-redundant.*

Proof: Without loss of generality (due to the symmetry of the graph) we can assume that the originating node is node 0. We consider the broadcasting clockwise from node 0 (broadcasting on the counterclockwise direction is treated similarly). Case 1(a) of Algorithm 1 generates ρ paths by following the routes:

$$0 \rightarrow \left\{ \begin{array}{ccccccc} 1 & \rightarrow & \rho+1 & \rightarrow & \cdots & \rightarrow & (D-1)\rho+1 \\ 2 & \rightarrow & \rho+2 & \rightarrow & \cdots & \rightarrow & (D-1)\rho+2 \\ \vdots & & & & & & \\ \rho-1 & \rightarrow & \rho+(\rho-1) & \rightarrow & \cdots & \rightarrow & (D-1)\rho+(\rho-1) \\ \rho & \rightarrow & 2\rho & \rightarrow & \cdots & \rightarrow & D\rho \end{array} \right.$$

Node $D\rho$ is reached following the last path. *Any* number j between $(\rho+1)$ and $D\rho$ can be uniquely represented as $j = q\rho + r$, where $0 \leq r < \rho$ is the

remainder of the division by ρ. Since node j receives the message from node $j-\rho=(q-1)\rho+r$ and sends the message to node $j+\rho=(q+1)\rho+r$ (Case 2 of Algorithm 1), we see that the node addresses on any path have the same remainder. Hence, any node up to and including node $D\rho$ belongs to a path and the paths have no node in common. Exactly the same holds for the paths formed in the counterclockwise direction from node 0.

To complete the proof we have to show that while the clockwise paths reach node $D\rho$, the counterclockwise paths reach node $D\rho+1$. We distinguish two cases:

- Case 1: $k=0$

  In this case the latter paths reach node $m-a\rho$:

  $$0 \rightarrow m-\rho \rightarrow m-2\rho \rightarrow \cdots \rightarrow m-a\rho$$

  Since $k=0$, from equation 3.4 we get

  $$k=0=m-a\rho-D\rho-1 \Rightarrow m-a\rho=D\rho+1$$

- Case 2: $k\neq 0$

  In this case the counterclockwise paths reach node $m-k-a\rho$ following the route:

  $$0 \rightarrow m-k \rightarrow m-k-\rho \rightarrow \cdots \rightarrow m-k-a\rho$$

  starting with weight $(a+1)$. But

  $$m-k-a\rho=m-(m-a\rho-D\rho-1)-a\rho=D\rho+1$$

□

Theorem 3.2 *The broadcasting described in Algorithm 1 is completed in exactly D steps (i.e. minimum number of steps).*

Proof: Since we start with weights D, a and $(a+1)$, and after each step the weights are decreased by one, the algorithm is completed in $\max\{D, a, a+1\}$ steps. So, what we have to prove is that $a < D$. From equation 3.1, we obtain

$$D = \lceil \frac{\lfloor m/2 \rfloor}{\rho} \rceil \geq \frac{\lfloor m/2 \rfloor}{\rho} \Rightarrow D\rho \geq \lfloor \frac{m}{2} \rfloor \tag{3.5}$$

Also from the analysis that preceeded Algorithm 1 (equation 3.2), we have

$$m - a\rho > D\rho \Rightarrow a\rho < m - D\rho \Rightarrow a\rho < m - \lfloor \frac{m}{2} \rfloor \tag{3.6}$$

- In the case that $m/2$ is an integer, $m/2 = \lfloor m/2 \rfloor$ and as a result (combining equations 3.5 and 3.6)

$$a\rho < \frac{m}{2} \leq D\rho \Rightarrow a < D$$

- If $m/2$ is not an integer, then $m - \lfloor m/2 \rfloor = \lfloor m/2 \rfloor + 1$ and equations 3.5 and 3.6 give

$$a\rho < m - \lfloor \frac{m}{2} \rfloor = \lfloor \frac{m}{2} \rfloor + 1 \Rightarrow a\rho \leq \lfloor \frac{m}{2} \rfloor \leq D\rho$$

  hence $a < d$ except for the special case that $a\rho = D\rho = \lfloor m/2 \rfloor$. For this case, we have (from equation 3.4)

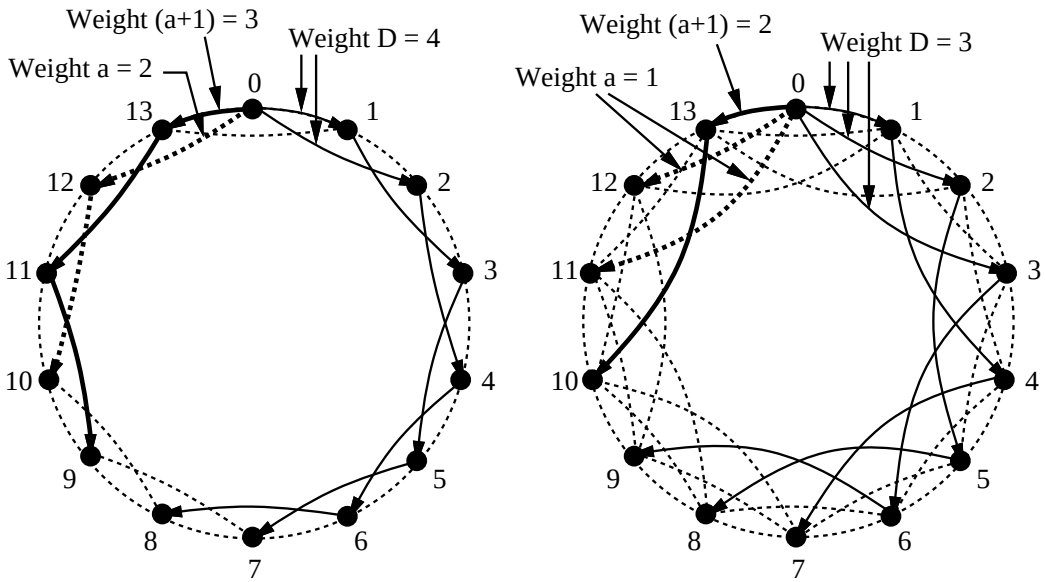
$$k = m - a\rho - D\rho - 1 = m - 2\lfloor \frac{m}{2} \rfloor - 1 = 0$$

  which means that we do not have any paths starting with weights of value $(a+1)$.

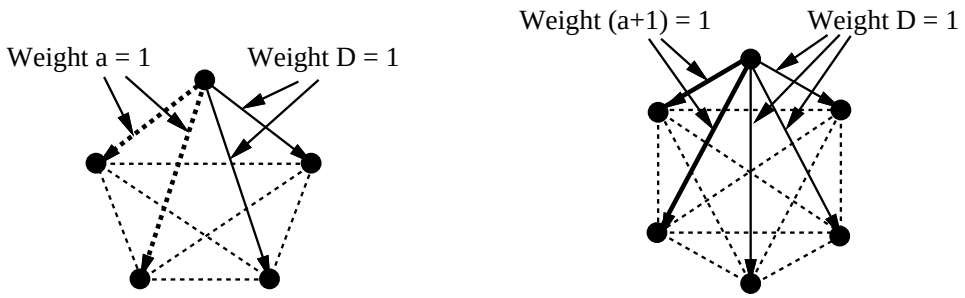
□

Four examples are shown in Figure 3.3.

A comment should be made now about the proposed broadcasting algorithm with respect to the implementation cost. The hardware requirements are minimal since all the intermediate nodes need only compare the received weight to 1



C_{14}<1, 2> C_{14}<1, 2, 3>



C_5<1, 2> C_6<1,2,3>

Figure 3.3: Broadcasting in circulant graphs

and possibly decrement it by 1. All the computations for the different weights have to be performed only once, and only in the originating node. This can be done easily in software or by hardwiring the values of a, k and D to the router subsystem. Also, the weight (which is added to the message header), requires only $\lceil \log_2(D) \rceil$ bits since D is the maximum value it will have. Some broadcasting techniques for larger scale networks ([25, 24]) techniques need a much more costly configuration both in terms of hardware, message length and time. For example, instead of using weights, the originating node could just place its address on the message header. This way the originating node does not need to perform any computations at all. But now the intermediate nodes will have to do a costly calculation in order to discover, based on the sender's address, if they should continue transmitting or not. This requires more complex hardware and introduces delays. Finally, the message header is loaded with $\lceil \log_2(m) \rceil$ bits, which means at least one more bit than the proposed method ($D \leq m/2$ always).

3.2 Broadcasting in hypercycles

The broadcasting of a message to every node in a hypercycle is similar to the one used for hypercubes [14] and generalized hypercubes [18]. The complete procedure is formulated in Algorithm 2, in Figure 3.4, and is completed in the minimum number of steps, equal to the diameter of the hypercycle.

Assuming that the r dimensions of the hypercycle $G^{\rho_r,\rho_{r-1},\ldots,\rho_1}_{m_r,m_{r-1},\ldots,m_1}$ are ordered (in any fixed ordering), the originating node sends the message with a pair of weights attached to it, (wd, wc), where wd is used for propagating through the dimensions and wc is used for broadcasting within a single dimension (circulant graph). The nodes that receive the message with weights (wd, wc) send it to dimensions $i = 1, 2, \ldots, wd - 1$ with weights (i, wc_i), where wc_i is computed according to case 1 of Algorithm 1, because the message has to be broadcast

Case 1: ORIGINATING NODE

For every dimension $i = 1, 2, \ldots, r$ transmit the message with weights (i, wc_i) to all the adjacent nodes. The weight wc_i is to be calculated according to Case 1 of Algorithm 1, applied to the circulant graph of the i-th dimension

Case 2: THE OTHER NODES

(Suppose (wd, wc) is the received weights)

- If $(wc > 1)$ then
  Within dimension wd transmit the message with weights $(wd, wc - 1)$ following Case 2 of Algorithm 1
- If $(wd > 1)$ then
  For every dimension $i = 1, 2, \ldots, wd - 1$ transmit the message with weights (i, wc_i) to all the adjacent nodes. The weight wc_i is to be calculated according to Case 1 of Algorithm 1, applied to the circulant graph of the i-th dimension

Figure 3.4: Algorithm 2

along the whole circulant graph that corresponds to dimension i. They also send it along dimension wd with weights $(wd, wc-1)$ according to case 2 of algorithm 1 since the node is an intermediate node in the broadcasting along dimension wd. This way the message is received by every node with no redundancy. An example is shown in Figure 3.5.

The only thing to note is that since the maximum value for wd is r and the maximum value for wc, when in dimension i, is the diameter D_i of the corresponding circulant graph (Theorem 3.2), the number of steps needed to complete the broadcasting for the whole hypercycle is

$$\sum_{i=1}^{\max\{wd\}} \max\{wc \text{ in dimension } i\} = \sum_{i=1}^{r} D_i$$

which according to equation 2.5 is equal to the diameter of the hypercycle. Hence the algorithm requires the minimum number of steps to complete.

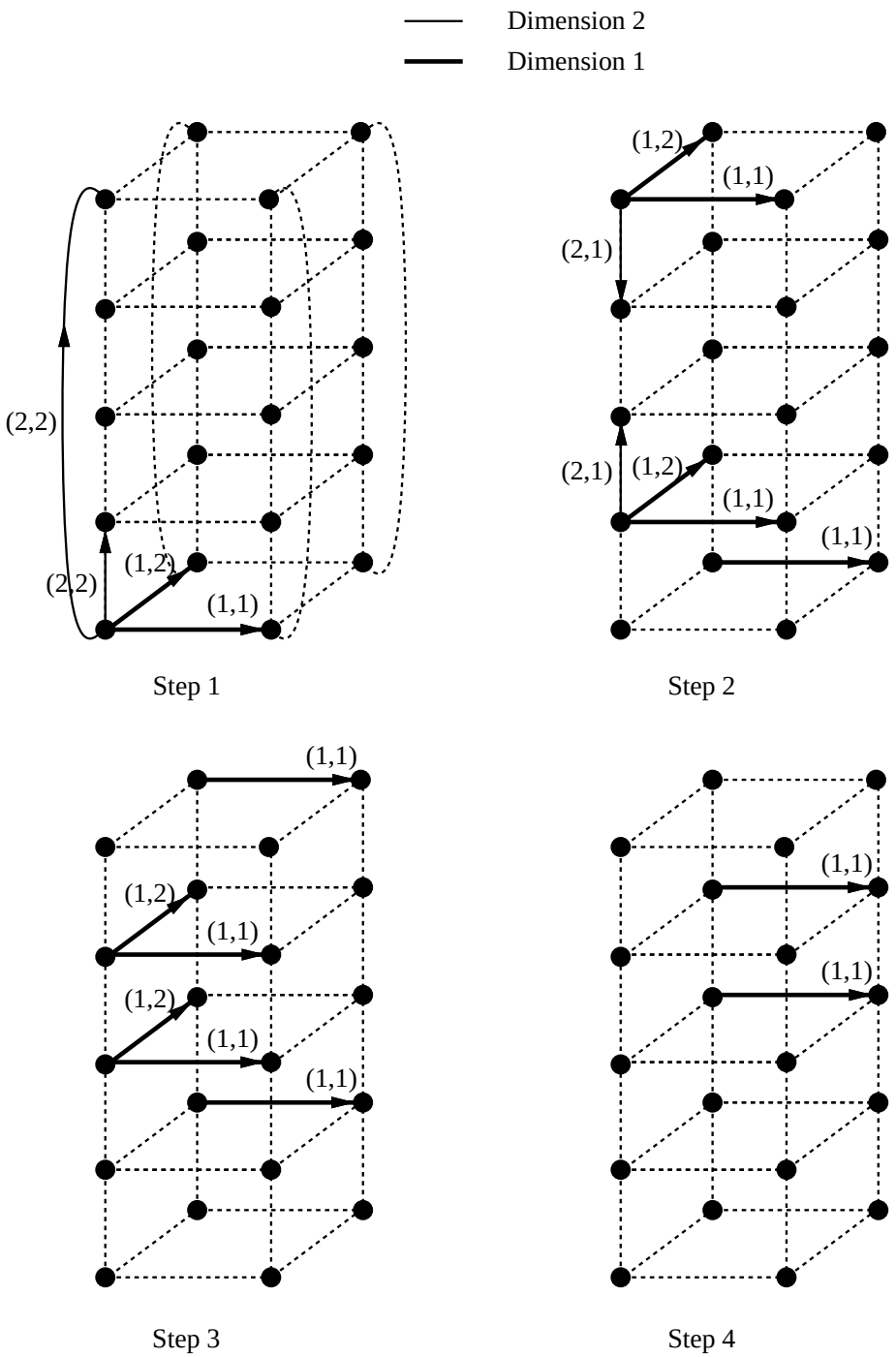


Figure 3.5: Broadcasting in the hypercycle $G^{1,1}_{5,4}$

Chapter 4

The Allocation Problem

For a parallel machine in a multiuser environment, where jobs implement parallel algorithms, it is expected that more jobs (or tasks) than one are to be executed concurrently. According to its flow graph ([1]), each task will require a number of nodes on which to execute. The problem of deciding which processing nodes are to be dedicated to which task is referred to as the *processor allocation problem* and its effective solution is an important issue towards achieving high performance. An inefficient processor allocation scheme can cause fragmentation, meaning that nodes are left isolated, unable to communicate efficiently with each other, thus decreasing the system performance.

In [29] there is a distinction made between "on-line" and "off-line" allocation. In "on-line" allocation, an arriving task gets the number of processing nodes it requests immediately when they become available, regardless of subsequent requests while in the "off-line" case the operating system collects a sufficient number of tasks before actually allocating nodes to them. Since the tasks are known in advance in the "off-line" allocation, one can design an algorithm to optimally allocate all of them at the same time. The decision problem corresponding to "off-line" allocation was shown in [29] to be NP-complete (when applied to hypercubes) which implies that an optimal solution is almost impossible to compute. Apart from this, most of the incoming jobs cannot wait for a

long period of time before actually starting to execute, so "on-line" allocation is necessary for fast allocations. The "on-line" allocation is the only type of allocation that is going to be studied in this thesis.

To begin with, the nodes allocated to a task should be able to communicate efficiently. While one could allocate nodes far apart from each other, the performance of the system will drop since not only will the task take more time to execute but also the intermediate processors allocated to other tasks will have to participate in the communication paths between the isolated nodes. What is actually allocated is in accordance with the routing procedure used for the internode communication, and is a portion of the network that is *closed* under this routing scheme:

Definition 4.1 Given a graph and a routing scheme for the internode communication, a subgraph S_G of this graph is *closed* under this routing scheme iff all the intermediate nodes required to complete a path between a source and a destination in S_G are nodes of S_G.

By allocating subgraphs closed under the routing procedure, there is no interference between independent jobs hence the running times are kept to the minimum possible. For example e-cube [14, 27] is a common routing algorithm for hypercubes. If one applies e-cube routing to a subcube of a hypercube, then all the communication paths lie inside this subcube. If one selects a set of nodes that do not form a cube, the e-cube routing will in some cases require nodes outside this set to form the paths and will consequently interfere with the routing of messages in the other parts of the hypercube. Hence the allocation schemes for hypercubes which have appeared in the literature allocate subcubes to the incoming tasks.

We have discussed the purpose of an allocating method being able to allocate nodes efficiently. The vague term 'efficiently' is to be quantified by considering the delay the tasks have to suffer before actually being allocated the nodes they

need, and the resource utilization of the system. In a lightly loaded situation the tasks should be allocated quickly, the only delay being the complexity of the allocation algorithm. When the load becomes heavier the main delay is due to the time that the tasks have to wait for the required nodes to be freed. In the following sections we assume that the delay because of the allocation algorithm itself is negligible. A separate comment on its time complexity will be given when the algorithm is first presented. We are also assuming that tasks which cannot be directly allocated are placed in a waiting queue that operates on a First-In-First-Out (*FIFO*) policy.

Since the dynamic performance is almost impossible to calculate analytically, simulation will be the main tool for our purposes. Tasks will be assumed to arrive, wait until there are nodes available, execute for a period of time and finally leave, freeing thus nodes for subsequent requests. The deallocation of nodes when a task finishes, will be called *processor relinquishment*.

When we consider an allocation policy *statically*, we deal with a set of incoming requests to be *allocated* only, i.e. without considering the processor relinquishment phase. Although the behaviour in a practical environment will be quite different, static allocation may give some idea about the dynamic performance of the policy.

Definition 4.2 An allocation policy is *statically optimal* if, and only if, it can allocate all sets of requests for which there exists a feasible allocation.

The term 'feasible' means that the total number of requested nodes does not exceed the total number of nodes in the system. A direct consequence of Definition 4.2 is that if a set of requests can be allocated with a statically optimal policy, then any reordering of those requests should also be allocated with this policy. Static optimality is a nice property but, as will be shown in the next chapter, it cannot guarantee a good performance in a dynamic environment. Policies that are not statically optimal may perform better in practice than

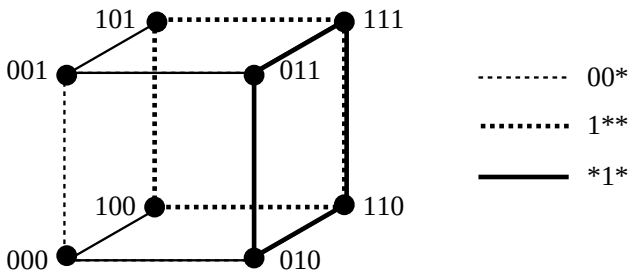


Figure 4.1: Subcubes $00*$, $1**$ and $*1*$ in Q_3

statically optimal ones.

The rest of the chapter summarizes the allocation policies that have been reported for hypercubes. The first of them is presented in section 4.1 which also introduces the basic notation to be used in the following chapter. Section 4.2 presents the rest of the allocation strategies for hypercubes.

4.1 Allocation in hypercubes - buddy strategy

Consider the set of nodes $\{000, 001\}$ in the hypercube Q_3 shown in Figure 4.1. These two nodes form a subcube Q_1 of Q_3 by themselves. Another example is the set of nodes $\{100, 101, 110, 111\}$ which forms a subcube Q_2 of Q_3. We are going to use the symbol '$*$' as a *don't care* symbol, i.e. one that can be either 1 or 0. Then the above subcubes can be represented by a single string as follows

$$00* = \{000, 001\}$$

$$1** = \{100, 101, 110, 111\}$$

which is going to be called the *address of the subcube*. Another example is the subcube $*1*$ which consists of the nodes 010, 011, 110 and 111. Figure 4.1 shows all these examples graphically.

The *dimension* of a subcube is equal to the number of '$*$'s in its address and its *size* is the number of nodes it contains. A subcube of size 2^k will also be referred to as k-subcube. Note that since the address of a subcube can have

'$*$'s in any of the address digits, there are in total $\binom{n}{k}$ possible placements of k stars in n address digits. Also since the remaining $(n-k)$ digits of the address can be either 0 or 1, the total number of k-subcubes in Q_n is equal to

$$\binom{n}{k}2^{n-k} \tag{4.1}$$

Now consider the *address tree* of a hypercube (Figure 4.2). In the bottom of the tree (i.e. the leaves) we list the addresses of the nodes in the graph. These also represent all the 0-subcubes of the hypercube. As we move up, we combine 2 nodes at a time forming higher-dimensioned subcubes. For example in Figure 4.2 the subtree that contains the nodes 000 and 001 actually represents the 1-subcube $00*$, while the subtree that contains the nodes 100, 101, 110 and 111 corresponds to the subcube $1**$. The root represents the whole Q_3. Following a path from the root to a leaf gives the address of the corresponding node, based on the labels of the edges. One can notice that nodes 010 and 011 constitute the subcube $01*$. Nodes 110 and 111 constitute the subcube $11*$. Yet these two subcubes do not have the same father in the address tree, hence *they cannot be combined to form a 2-subcube*. In fact, though, $*1*$ *is* a 2-subcube that contains exactly the above four nodes. In conclusion, we see that the address tree *cannot recognize* all the existing subcubes of the graph. Since in the i-th level of the tree (0-th level being the root node) there are 2^i nodes, and each node at this level represents an $(n-i)$-subcube, the address tree recognizes in total

$$2^i \text{ subcubes of dimension } (n-i)$$

which is much less that the $\binom{n}{n-i}2^i$ subcubes of dimension $(n-i)$ that exist in total, according to equation 4.1. Before proceeding with the description of the buddy strategy, some important definitions will be given.

Definition 4.3 A *region* is a set of consecutive integers $\{j \mid a \leq j \leq b, j \in I^+\}$, symbolized as $\#[a,b]$ (I^+ is the set of positive integers).

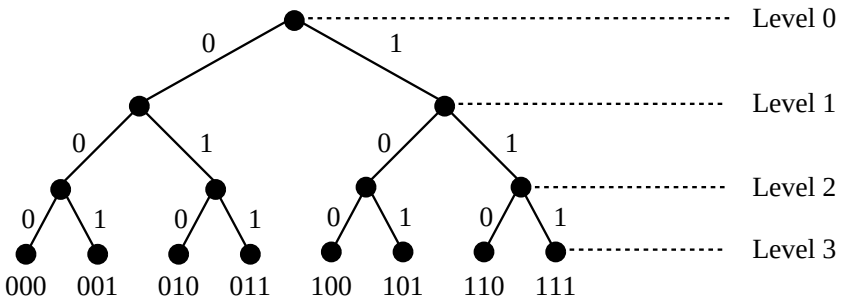


Figure 4.2: The address tree of Q_3

Note that the set of integers {4, 5, 6, 7} which forms a region, in binary notation is written as {100, 101, 100, 111} which represents the subcube $1**$. If $*^k$ means k consecutive $*$'s, then any subcube with address

$$b_n b_{n-1} \dots b_{k+1} *^k$$

is a region according to Definition 4.3 ($b_n, b_{n-1}, \dots, b_{k+1}$ are binary digits, i.e. 0 or 1). Note though that a region is not necessarily a subcube.

We now assume that tasks start to appear asking for a number of nodes to work on. In order to assign the necessary number of nodes to an incoming task we have to make certain that there are enough nodes that are not currently allocated to any task.

Definition 4.4 A *node* is *available* if it is not currently allocated to a job.

We are going to use one bit per node, called the *allocation bit*, to keep track of its availability. If the bit is 0 then the corresponding node is available while if it is 1 the node is unavailable (currently allocated to a task).

Definition 4.5 A *region* is *available* if all the allocation bits associated with the nodes of this region are 0. If there exists at least one such bit which is 1, the region is *unavailable*.

PROCESSOR ALLOCATION
(Suppose there is a request I for a number of nodes)

1. Set k = the dimension of the subcube required to accommodate the request I.
2. Determine the *least* integer m such that all the allocation bits in the region $\#[m2^k, (m+1)2^k - 1]$ are 0's, and set all the bits in this region to 1's.
3. Allocate the node $B_n(i)$, for every $i \in \#[m2^k, (m+1)2^k - 1]$.

PROCESSOR RELINQUISHMENT
(Suppose the request that occupied the subcube q finished)

1. Reset the i-th allocation bit to 0, for every $B_n(i) \in q$.

Figure 4.3: The buddy strategy for Q_n

Based on the above discussion, we present the buddy strategy for allocation in a hypercube ([30]) which consists of two parts, the allocation and the relinquishment of processing nodes, in Figure 4.3. Note that $B_k(i)$ means the k most significant digits in the binary representation of i. For example $19 = 10011_2$, and $B_2(19) = 10$. $B_n(i)$ is the binary representation of i if $i \leq 2^n - 1$.

What the buddy strategy does, is essentially a linear search for a subcube to fit a request I. When I asks for a k-subcube, the $(n-k)$-th level of the address tree is scanned from left to right and the first available subcube is allocated. Thus the buddy strategy is actually a *first fit* strategy. The nodes associated with the allocation bits in the region $\#[m2^k, (m+1)2^k - 1]$ always constitute a subcube with address $B_{n-k}(m)*^k$, in which the allocated nodes are the leaves of the subtree which has as its root the m-th node from the left at the $(n-k)$-th level of the tree.

As an example (see also Figure 4.4) assume that in a Q_3 all the allocation bits are 0 and the sequence of requests $\{I_1, I_2, I_3, I_4\}$ asks for $\{2, 1, 2, 1\}$ nodes. For I_1 a search at the second level of the tree reveals that the subcube $0**$ is

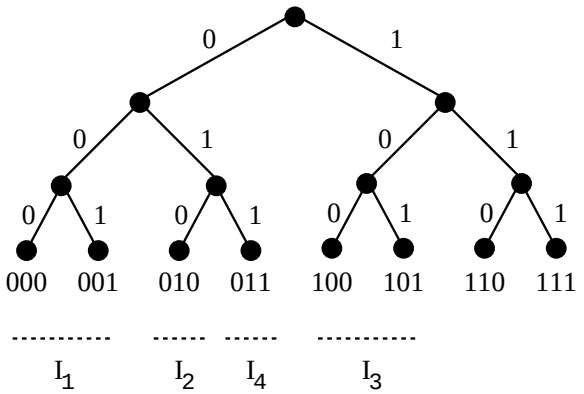


Figure 4.4: Allocating nodes in Q_3 with the buddy strategy

free, or allocation bits 0 and 1 are 0's according to the algorithm in Figure 4.3 (m = 0), so I_1 is allocated the subcube $00*$. The allocation bits now are {1, 1, 0, 0, 0, 0, 0, 0}. When I_2 comes, the search is performed at the third level of the tree. The first node to be found available is the third one, i.e. node 010, and is allocated to I_2. The allocation bits are now {1, 1, 1, 0, 0, 0, 0, 0}. For I_3 the second level of the tree is searched and the subcube $00*$ is found unavailable ($00*$ was allocated to I_1). The subcube $01*$ is also unavailable because one of its nodes (010) is allocated to I_2. Hence I_3 is allocated the next available subcube which is $10*$ and the allocation bits are now {1, 1, 1, 0, 1, 1, 0, 0}. Finally for I_4 the first available node is node 011 and is allocated to it. The final list of allocation bits is {1, 1, 1, 1, 1, 1, 0, 0}.

Assuming that the requests ask for subcubes, i.e. for a power of 2 nodes, the buddy strategy is statically optimal ([30]). This means, according to Definition 4.2 that as long as a sequence of requests $\{I_1, I_2, \ldots, I_j\}$ satisfies the following

$$\sum_{i=1}^{j} |I_i| \leq 2^n,$$

the buddy strategy can allocate all requests in the sequence on Q_n. The symbol $|I_i|$ is the number of nodes that I_i requires, called the *size* of the request. The time complexity of the buddy strategy is $\mathcal{O}(2^n)$, where 2^n is the number of nodes

in Q_n (hence the number of allocation bits).

As noted before, the address tree, hence the buddy strategy, recognizes only 2^{n-k} k-subcubes out of the $\binom{n}{k}2^{n-k}$ in total. This is a disadvantage which causes under-utilization of the processing nodes in a dynamic environment. Suppose that in the above example, after a time period, requests I_2 and I_4 finish, so that nodes 010 and 011 become available. Then a request I_5 comes with $|I_5| = 4$. A search in the first level of the tree will reveal that neither $0**$ nor $1**$ are available since parts of them are still allocated to requests I_1 and I_2. Hence I_5 cannot be allocated, and has to wait until one of the above subcubes becomes available. Note though that the available nodes 010, 011, 110, 111 constitute the 2-subcube $*1*$ which could satisfy I_5, yet the buddy strategy does not recognize it. The following section summarizes the improved allocation strategies for the hypercubes that have appeared in the literature.

4.2 Other strategies for hypercubes

In [30], an improvement over the buddy strategy is given, which uses the gray codes. Simply stated, a gray code is a permutation of the numbers $\{0, 1, \ldots, 2^n - 1\}$ so that the i-th number, when written in binary representation, differs in only one bit from the $(i-1 \bmod 2^n)$-th and the $(i+1 \bmod 2^n)$-th number in the set. For example, the set $\{00, 01, 11, 10\} = \{0, 1, 3, 2\}$ is a gray code. The address tree for Q_3 using node addresses in a gray code sequence is shown in Figure 4.5 and is actually a reorganization of the tree in Figure 4.2.

The main difference from the simple address tree of Figure 4.2 is that at any level of the gray code tree two adjacent nodes are recognized as forming a subcube, *even if they do not have the same parent*. For example, nodes 010 and 110 do not have the same parent but they constitute subcube $*10$. When a request comes for a subcube of dimension k, the gray codes strategy performs exactly the same search that the buddy strategy does but at the $(n-k+1)$-th

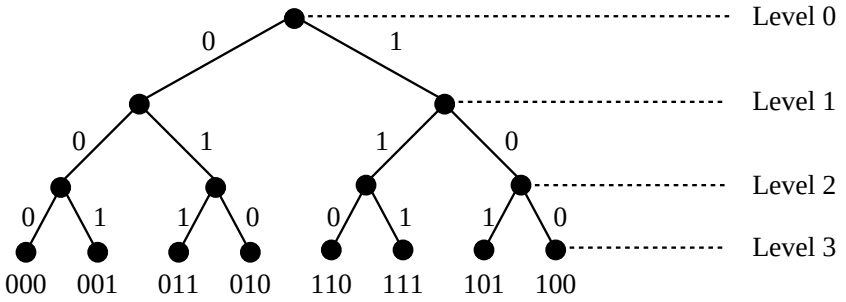


Figure 4.5: The gray code address tree for Q_3

level of the tree. It searches for two adjacent nodes that correspond to free subcubes. This way, the gray code strategy can recognize $2^{n-k+1} = 2 \times 2^{n-k}$ k-subcubes, i.e. *twice* as many as the buddy strategy can recognize. This enhanced subcube recognition ability results naturally in improved performance. The time complexity of the gray codes strategy is $\mathcal{O}(2^n)$, although on the average the buddy strategy is faster by a small constant factor.

Still, though, there remain subcubes unrecognized. Using a procedure described in [30] one can construct many distinct n-bit gray codes and use them concurrently to improve the performance over the single-gray code strategy. The idea is that different gray codes recognize different subcubes hence the subcube recognition ability is enhanced. The authors also showed that in order to recognize all the possible subcubes, one needs in total $\mathcal{O}(\binom{n}{\lfloor n/2 \rfloor})$ different gray codes. The gray code strategy is statically optimal.

A different strategy is proposed in [29] that does not use the address tree and allocation bits. The algorithm keeps a list of *all* the available subcubes. When a request comes for a k-subcube, the list is searched for an available subcube of the same size. If such a subcube is found, it is allocated to the request, otherwise a search is performed for the smallest available m-cube, where $m > k$. The m-subcube is then split in two k-subcubes, an $(k+1)$-, an $(k+2)$-, ..., and an $(m-1)$-subcube. One of the k-subcubes is allocated to the request and the others are placed back in the list. During processor relinquishment, the freed

subcube is put back on the list of available subcubes and a coalescing procedure is applied to the list, trying to combine low dimension subcubes into higher dimension ones. Subcubes are coalesced in such a way that the list always forms a *maximal set of subcubes* (MSS). An MSS contains the most subcubes of higher dimension as compared to any other set of free subcubes. Finding an MSS, however, is an NP-complete problem as shown in [29]. The authors of [29] thus proposed a number of heuristic procedures to approximate an MSS so that the allocation algorithms became usable. The time complexity of the fastest of the proposed algorithms is $\mathcal{O}(n2^n)$. Hence this strategy requires more processing time than the previous two but gives better results.

Finally in another recently proposed allocation scheme ([31]), called the *free list*, there are $n+1$ different lists to keep the available subcubes, list i containing the free i-subcubes, $0 \leq i \leq n$. As in [29], when a request for a k-subcube arrives, if there exists an available subcube of this dimension (sought in the k-th list), it is allocated to the request, otherwise a nearest higher dimension subcube is decomposed. Which subcube is to be decomposed is based on the gray code formed by the most significant digits of the address of the subcubes. While the previous strategy tries to keep an MSS after each allocation, the one proposed in [31] takes no such action and thus it is faster (but not optimal). When processor deallocation occurs, the relinquished subcube is placed to the corresponding list and is combined with others to form higher dimension subcubes. This coalescing scheme is applied to all the lists, possibly decomposing some of the higher dimension subcubes to form new ones using the newly freed subcube. The objective is to make the subcubes mutually disjoint, meaning that their addresses must have hamming distance other than zero (for more details refer to [31]). The free list strategy is statically optimal, and its time complexity is $\mathcal{O}(n2^n)$.

Chapter 5

Allocation in Hypercycles

The first fit allocation strategy proposed in [32] for hypercycles is a generalization of the buddy strategy presented in the previous chapter for hypercubes. In some cases, though, this strategy results in an inefficient partitioning of hypercycles and consequently in wasted processors. To overcome this drawback, a new strategy is proposed here that is not statically optimal but shows superior performance in a dynamic environment.

Both strategies are extended by the use of multiple address trees to further improve their performance. For hypercycles that have all their factors equal to each other (hypercubes fall into this category) we derive a way to choose the best set of address trees in order to maximize the performance of the allocation strategies. Finally, we extend our study to the case where some information about the statistical nature of the requests is available and we derive the optimum address tree to best fit them.

The performance of the various strategies was verified through computer simulation. We have implemented a simulator capable of simulating the above strategies for any hypercycle plus the buddy and the gray codes strategies for the hypercubes. Throughout our simulations, we keep track of the allocated nodes using the list of allocation bits presented in the previous chapter. Tasks are assumed to arrive at either constant or poisson distributed time intervals.

The mean interarrival delay can be specified by the user. Also every task is assumed to execute for a random time period, uniformly distributed in an interval specified by the user. Each task asks for a number of nodes which is also random and can follow a uniform, gaussian or exponential distribution, again all the parameters are user specified. If a task cannot be directly allocated due to the lack of free nodes, it is placed in a FIFO queue and waits until all the tasks in front of it are allocated. The simulation stops either after a user-specified time limit or when the queue fills up. Two values are then computed

- *the average queueing delay* which is the time that a task spent in the queue before actually starting execution, averaged over the total number of *allocated* tasks and
- *the average utilization* which is the fraction of the nodes that remain idle in each clock tick, averaged over the simulation length (total number of clock ticks). Note that if a task is allocated more nodes that it actually needs, the redundant nodes are assumed idle.

A good allocation strategy should keep the queuing delays low and the utilization high. The simulator together with the source code is presented in detail in Appendix A.

5.1 First fit strategy

In this section we summarize the allocation algorithm proposed in [32] for $G^{\rho_r,\rho_{r-1},\ldots,\rho_1}_{m_r,m_{r-1},\ldots,m_1}$. While in the hypercubes the proposed strategies allocate subcubes, the first fit strategy for the hypercycles allocates *fragments*. If we use '$*$' as the don't care symbol, then an '$*$' in the i-th digit of an address means that it can take any value from 0 to $m_i - 1$. A k-fragment contains k '$*$'s in its address exactly like a k-subcube in a hypercube. For example the fragment

$F = b_r b_{r-1} \ldots b_{k+1} *^k$ is the set of nodes

$$\{n \mid n = b_r b_{r-1} \ldots b_{k+1} a_k \ldots a_1, a_j \in \{0, 1, \ldots m_j - 1\},\ j = 1, 2, \ldots, k\}$$

A k-fragment that has k consecutive '$*$'s in the rightmost digits of its address is a region by Definition 4.3 and we are going to call it a *primary fragment* to distinguish it from the fragments that may have arbitrarily placed '$*$'s in their address. As an example, if $N = 2 \times 3 \times 4$, then the nodes {120, 121, 122, 123} constitute the primary 1-fragment $12*$. Note that the size of the fragment $F = b_r b_{r-1} \ldots b_{k+1} *^k$, $|F|$, is equal to the number of nodes it contains, i.e. $|F| = m_k \times m_{k-1} \times \cdots \times m_1$. Fragments are closed under the hypercycle routing presented in section 2.6.

Definition 5.1 The fragments $\{b_r b_{r-1} \ldots a_{k+1} *^k \mid a_{k+1} \in \{0, 1, \ldots, m_{k+1} - 1\}\}$ that comprise the fragment $b_r b_{r-1} \ldots b_{k+2} *^{k+1}$ are called *buddies*.

Definition 5.2 A fragment $b_r b_{r-1} \ldots b_{k+1} *^k$ is called a *hole* if and only if it is available but at least one of its buddies is not.

Requests for primary fragments are allocated by the first fit strategy exactly the way subcubes are allocated by the buddy strategy. The algorithm is shown in Figure 5.1. Note that $MR(i)$ is the mixed radix representation of i and corresponds to the address of node i.

An example is shown in Figure 5.2. The address tree for $G^{1,1,1}_{3,3,2}$ is constructed as for hypercubes the only difference lying in the fact that the i-th level of the tree contains $m_r \times m_{r-1} \times \cdots \times m_{r-i+1}$ nodes and each of these nodes represents a primary $(r-i)$-fragment containing $m_{r-i} \times m_{r-i-1} \times \cdots \times m_1$ nodes. Suppose we receive the following sequence $\{|I_1| = 2, |I_2| = 6, |I_3| = 6, |I_4| = 1\}$. Then for the request I_1 the nodes in the fragment $00*$, i.e. nodes 000 and 001, are allocated because a 1-fragment is needed to satisfy the request and $00*$ is the first available one in the $r - 1 = 3 - 1 =$ 2nd level of the tree. For I_2 we

PROCESSOR ALLOCATION
(Suppose there is a request I for a number of nodes)

1. Set $|F|$ = the size of the smallest primary fragment to accommodate the request I.
2. Determine the *least* integer m such that all the allocation bits in the region $\#[m|F|, (m+1)|F|-1]$ are 0's, and set all the bits in this region to 1's.
3. Allocate the node $MR(i)$, for every $i \in \#[m|F|, (m+1)|F|-1]$.

PROCESSOR RELINQUISHMENT
(Suppose the request that occupied the fragment F finished)

1. Reset the i-th allocation bit to 0, for every $MR(i) \in F$.

Figure 5.1: The first fit strategy for hypercycles

need a 2-fragment ($6 = 3 \times 2$) and we search in the first level of the tree. The fragment $0**$ is unavailable since part of it is allocated to I_1, hence I_2 is given the fragment $1**$. I_3 is then allocated the fragment $2**$ and the first available node (0-fragment) for I_4 is node 010. The final list of the 18 allocation bits is $\{1,1,1,0,0,0,1,1,\ldots,1\}$. The first fit strategy is statically optimal if the requests are assumed to ask for primary fragments.

5.2 Suboptimal strategy

Hypercycles, as mentioned in Chapter 2, were designed with the goal of matching the node and link requirements of various applications. Also the analytical expressions of the routing equations provide a way to smoothly expand an already designed hypercycle to one with more dimensions or more nodes per dimension. While the interconnection topology is flexible, the first fit allocation provides an inefficient partitioning of the graph in cases where requests do not match the fragment sizes. Algorithms that required fragments in a certain hypercycle, when transferred to other hypercycles, may be allocated many

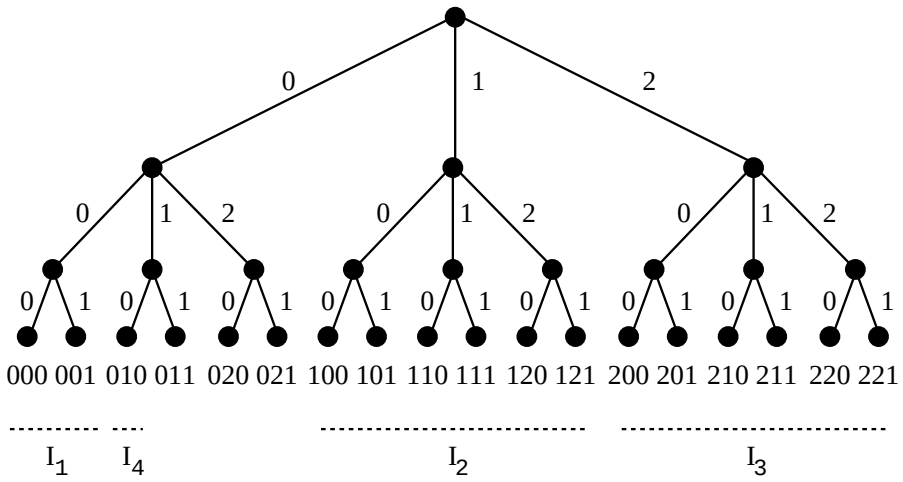


Figure 5.2: Allocation example in $Q^{1,1,1}_{3,3,2}$ using the first fit strategy

more nodes than what they actually need using the strategy of the previous section. For example if a job requires 6 nodes, it is efficiently allocated in a hypercycle where N is factored as $N = m_r \times m_{r-1} \times \cdots \times m_3 \times 3 \times 2$, while if $N = m_r \times m_{r-1} \times \cdots \times m_3 \times 5 \times 3$ it would be allocated 15 nodes. Thus we face the problem of *arbitrary request sizes* were jobs require a number of nodes that is not equal to the number of nodes inside primary fragments.

Consider a request for $m_i \times m_{i-1} \times \cdots \times m_1 + 1$ nodes in a hypercycle where $N = m_r \times m_{r-1} \times \cdots \times m_1$. Such a request represents the worst case of node waste because using the first fit policy a primary i-fragment is not enough to allocate the request. Hence a primary $(i+1)$-fragment is needed, allocating thus $m_{i+1} \times m_i \times \cdots \times m_1$ nodes in total, i.e. approximately about m_{i+1} times the needed nodes. Since every factor is ≥ 2, we allocate *at least* twice as many nodes as the request needs. One can see that in the general case many of the factors are greater than 2 resulting in an even bigger waste of nodes. The same problem exists in the hypercubes where a request for $2^i + 1$ nodes has to be allocated 2^{i+1} nodes with all the strategies presented in Chapter 4. The difference here is that the number of allocated nodes is *at most* two times the needed one, because

all the factors of N are equal to 2. In this section, we present a strategy to reduce the waste of nodes to the same level with that of the hypercubes. This means that the allocated nodes will be, in the worst case, *at most* two times the number of required nodes. The idea is to give away *parts* of fragments, not a whole fragment when it is not needed.

Consider the address tree of $G^{1,1,1}_{3,3,2}$ in Figure 5.2. A request for 4 nodes would result in allocating a whole subtree if the first fit strategy was used, i.e. one has to allocate all three children of a node in the first level. The waste of 2 nodes can be avoided by allocating only two of the children and this is feasible as shown below.

Definition 5.3 A *segment* is a subset of a primary fragment $b_r b_{r-1} \ldots b_{k+1} *^k$ that contains the $(k-1)$-fragments $\{b_r b_{r-1} \ldots b_{k+1} n_1 *^{k-1},\ b_r b_{r-1} \ldots b_{k+1}(n_1 + 1)*^{k-1}, \ldots,\ b_r b_{r-1} \ldots b_{k+1} n_2 *^{k-1}\}$, symbolized as

$$S(b_r b_{r-1} \ldots b_{k+1}[n_1, n_2]) = b_r b_{r-1} \ldots b_{k+1}[n_1, n_2] *^{k-1}$$

where

$$0 \leq n_2 - n_1 < \begin{cases} m_k/2 & \text{if } \rho_k < \lfloor m_k/2 \rfloor \\ m_k - 1 & \text{if } \rho_k = \lfloor m_k/2 \rfloor \end{cases}$$

If dimension k is a fully connected circulant graph ($\rho_k = \lfloor m_k/2 \rfloor$) a segment may consist of any number of $(k-1)$-fragments, i.e. n_1 and n_2 can have any value between 0 and $m_k - 1$. Otherwise, less than half of the m_k $(k-1)$-fragments can be contained in a segment, i.e. $n_2 - n_1 < m_k/2$. The reason for the latter is to guarantee that segments are closed under the hypercycle routing, as proven in Theorem 5.2 below. Also note that if $n_1 = n_2$, then the segment reduces to a $(k-1)$-fragment. As an example, in $G^{1,1,1}_{3,5,4}$ the segment $2[2,3]*$ is a subset of the fragment $2** = \{200, 201, 202, 203, 210, \ldots, 213, \ldots, 223, \ldots, 233, \ldots, 243\}$ (20 nodes) and contains the 1-fragments $22*, 23*$, i.e. the nodes {220, 221, 222, 223, 230, 231, 232, 233}. In $G^{1,1,1}_{3,3,2}$ the segment $0[0,1]*$, a subset of $0**$, contains

the fragments $00*, 01*$, i.e. only two children of the leftmost node in the second level of the tree (Figure 5.2).

Segments offer a much more efficient way to partition a hypercycle because they reduce the number of allocated nodes by including only a portion of the nodes in a fragment. This is formalized with Theorem 5.1 where we prove that in the worst case, a segment contains *at most* twice the number of nodes needed by a request. In addition, allocating segments meets the requirement of efficient internode communication as shown by Theorem 5.2.

Theorem 5.1 *If a request requires R nodes and the smallest segment to accommodate this request contains S nodes, then $S \leq 2R$.*

Proof: Let $F_k = \prod_{i=1}^{k} m_i$, the size of a primary k-fragment. Then there exists a k such that $F_k \leq R < F_{k+1} = m_{k+1}F_k$. Moreover, $\exists j,\ 1 \leq j < m_{k+1}$, such that

$$jF_k \leq R < (j+1)F_k \leq m_{k+1}F_k$$

- *Case 1:* $1 \leq j < m_{k+1}/2$

  The smallest segment to fit the request is, then, of the form

  $$b_r b_{r-1} \ldots b_{k+2}[n_1, n_2]*^k$$

  with $n_2 - n_1 = j - 1$ and size $S = jF_k$. Thus

  $$2R \geq 2jF_k > jF_k \Rightarrow 2R > S.$$

  Note that since $j < m_{k+1}/2$, according to Definition 5.3, $(n_2 - n_1)$ has a legal value, whatever the value of ρ_{k+1} is.

- *Case 2:* $m_{k+1}/2 \leq j < m_{k+1}$

  In the worst case, the smallest segment to fit the request will be a whole $(k+1)$-fragment, with size $S = m_{k+1}F_k$. But then

  $$2R \geq 2jF_k \geq m_{k+1}F_k \Rightarrow 2R \geq S.$$

□

Theorem 5.2 *Segments are closed under the hypercycle routing scheme.*

Proof: Suppose that $x = b_r b_{r-1} \ldots b_{k+1} x_k \ldots x_1$ and $y = b_r b_{r-1} \ldots b_{k+1} y_k \ldots y_1$ are two nodes in the segment $b_r b_{r-1} \ldots b_{k+1}[n_1, n_2]*^{k-1}$. Based on the hypercycle routing described in section 2.6, the path from a source to a destination is formed by modifying the address digits in which the two nodes differ. This means that we have to route only through dimensions $1, 2, \ldots, k$.

Routing through dimensions $1, 2, \ldots, (k-1)$ uses only nodes in the segment to form a path because the segment consists of $(k-1)$-fragments and fragments are closed under the hypercycle routing scheme ([32]). Hence we only have to prove that the segment is closed when routing in dimension k.

- In the case that $\rho_k = \lfloor m_k/2 \rfloor$, i.e. a fully interconnected dimension, then every node can communicate with every other node in this dimension using only 1 link. This means that the communication paths consist only of two nodes, the source and the destination, thus trivially the segment is closed.

- In the case that $\rho_k < \lfloor m_k/2 \rfloor$, from Definition 5.3 we know that $n_2 - n_1 < m_k/2$. Hence in dimension k the segment contains less than half of the nodes, and these nodes are consecutive. Based on the routing equation (section 2.6), we know that the router chooses a greedy (i.e. shortest) path. It is clear that since the nodes of the segment in dimension k, constitute less than a half-cycle all the shortest paths lie among them.

□

The strategy proposed here allocates segments instead of primary fragments and works like the first fit strategy by linearly searching for the first segment to fit the requested nodes. The algorithm is presented in Figure 5.3.

PROCESSOR ALLOCATION
(Suppose there is a request I for a number of nodes)

1. Set $|S|$ = the size of the smallest segment to accommodate the request I. Also let $|f|$ be the size of the largest primary fragment so that $|S| \geq |f|$ and $|F|$ the size of the smallest primary fragment so that $|S| < |F|$.
2. Determine the *least* integers m, m_s such that all the allocation bits in the region $\#[m|F| + m_s|f|, m|F| + m_s|f| + |S| - 1]$ are 0's, and set all the bits in this region to 1's.
3. Allocate the node $MR(i)$, for every $i \in \#[m|F| + m_s|f|, m|F| + m_s|f| + |S| - 1]$.

PROCESSOR RELINQUISHMENT
(Suppose the request that occupied the segment S finished)

1. Reset the i-th allocation bit to 0, for every $MR(i) \in S$.

Figure 5.3: The suboptimal strategy for hypercycles

If an i-fragment would be required for a request I using the first fit strategy, then with the new strategy a search is performed in the $(r - i + 1)$ level of the tree instead of the $(r - i)$-th level. The nodes that are examined in this level are all children of a primary i-fragment, the i-th fragment chosen from left to right (minimum 'm' in Figure 5.3). The children are also examined from left to right to locate the first available segment (minimum 'm_s') to fit the request. We progress in steps of the size of the immediately smaller fragment since a segment is a collection of such fragments. If all of them are needed then the whole i-fragment is allocated.

For example, consider the address tree of $G^{1,1,1}_{3,3,2}$ in Figure 5.4 and the sequence of requests $\{|I_1| = 4, |I_2| = 4, |I_3| = 2\}$. For I_1, we search the 2-fragment $0**$ for a segment of 4 nodes. The two leftmost children of this fragment are free, which means that we give the segment $0[0, 1]*$ to I_1. In the same manner, I_2 is allocated the segment $1[0, 1]*$, because the leftmost fragment to contain a

free 4-node segment is the fragment $1**$. For I_3 we need a 2-node segment, or equivalently a 1-fragment. The first such segment to be found free is $0[2,2]*$ which is the fragment $02*$.

It should be clear that the proposed strategy behaves significantly better than the first fit strategy. In the previous example, the first fit strategy would have allocated 14 nodes in total, instead of the 10 requested, as can be seen in Figure 5.5. For I_1 and I_2 we need two whole 2-fragments, these being the fragments $0**$ and $1**$ respectively. For I_3, then, we allocate the 1-fragment $20*$.

Theorem 5.3 *The proposed allocation strategy is not statically optimal.*

Proof: Assuming that the requests ask for nodes that match the segment sizes, we give a counter-example by constructing a request sequence which cannot be allocated. Consider the address tree of $G^{1,1}_{2,3}$ in Figure 5.6 and the request sequence $\{|I_1| = 2, |I_2| = 2, |I_3| = 2\}$ and observe that $\sum_{j=1}^{3} |I_j| = 6$, i.e. there are enough nodes in the graph for all the requests. I_1 is then allocated the segment $0[0,1]$ and I_2 the segment $1[0,1]$. While there are two free nodes remaining in the graph, the address tree does not place them inside the same fragment, hence there is no segment recognized to allocate I_3.

□

It should be noted that if requests match the sizes of fragments, the proposed strategy reduces exactly to the first fit strategy (and is statically optimal for these cases). The proposed strategy is going to be referred to as the *suboptimal* strategy from now on.

We simulated both strategies and present the results in Figures 5.7 to 5.12. The simulations lasted for a period of 20000 clock ticks. The tasks generated in this period would execute on the processing nodes for a period uniformly distributed in between 3 and 7 clock ticks. They are assumed to arrive in a poisson manner with mean interarrival delay of λ, or equivalently with mean

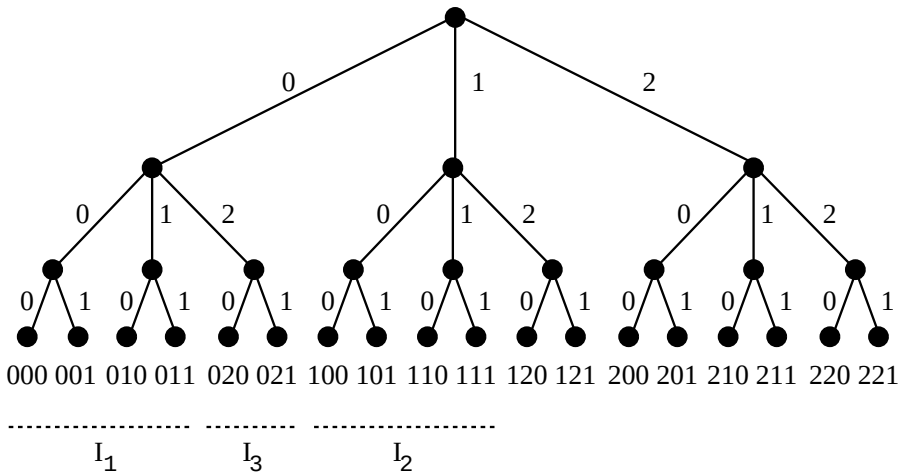


Figure 5.4: Allocation example in $Q^{1,1,1}_{3,3,2}$ using the suboptimal strategy

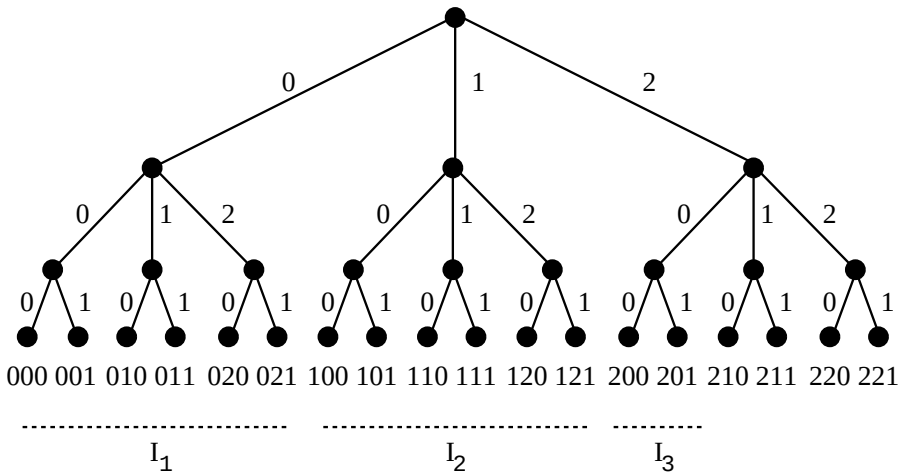


Figure 5.5: The same example using the first fit strategy

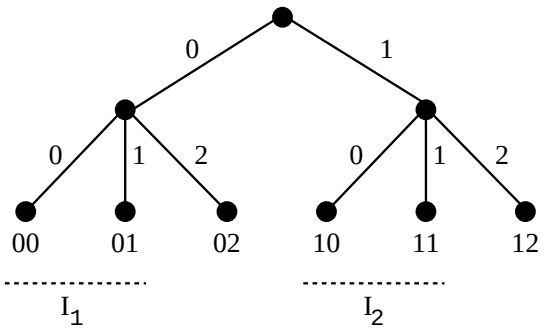


Figure 5.6: Counter-example in $Q^{1,1}_{2,3}$

interarrival rate of $1/\lambda$. We repeated the simulations using both the first fit and the suboptimal strategies for different values of λ. For a large λ, the load is light since tasks arrive far apart from each other. For smaller λ the load becomes heavier because tasks arrive before the older ones finish their execution. Thus tasks are filling up the queue increasing the queueing delays. The utilization of the processors is increased at the same time because the processors are constantly being occupied by tasks. The utilization, though, tends to an upper bound which is determined by the average waste of nodes. In each of the Figures 5.7 to 5.12 the average queueing delay and the average utilization is given versus the mean task interarrival rate.

Figures 5.7 and 5.8 show the results in terms of average queueing delay and utilization respectively, for the hypercycle $G^{1,1}_{3,2}$. One can see that for every interarrival rate the queueing delay when using the suboptimal strategy is much less than when using the first fit strategy. The utilization of processors is also significantly higher for the suboptimal strategy. The same conclusions are derived from Figures 5.9 and 5.10 for $G^{1,1,1,1}_{3,3,2,2}$ and Figures 5.11 and 5.12 for $G^{2,2,1,1}_{5,4,3,2}$. These three hypercycles were chosen because they represent a wide variety of sizes.

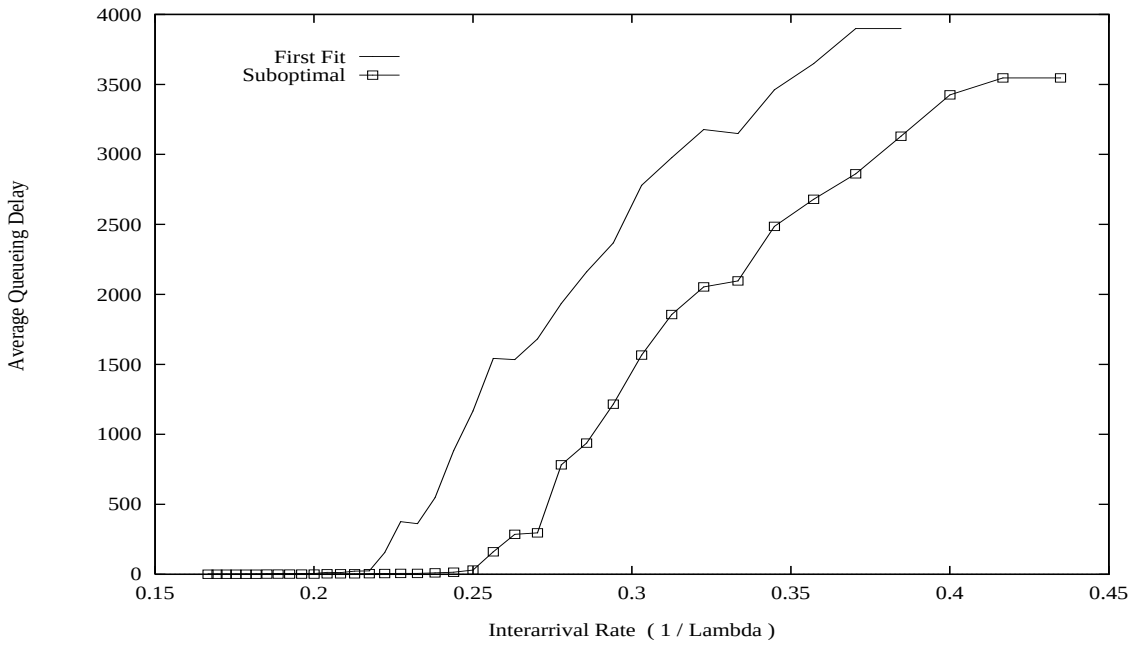


Figure 5.7: Average queueing delay in $G_{3,2}^{1,1}$ using the two strategies

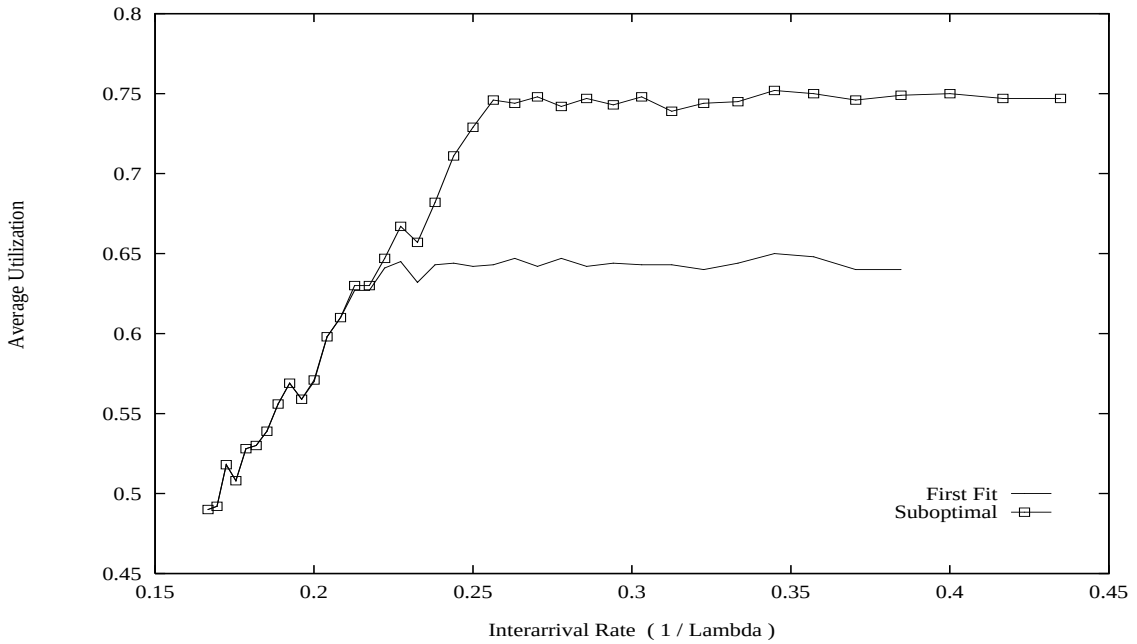


Figure 5.8: Average utilization in $G_{3,2}^{1,1}$ using the two strategies

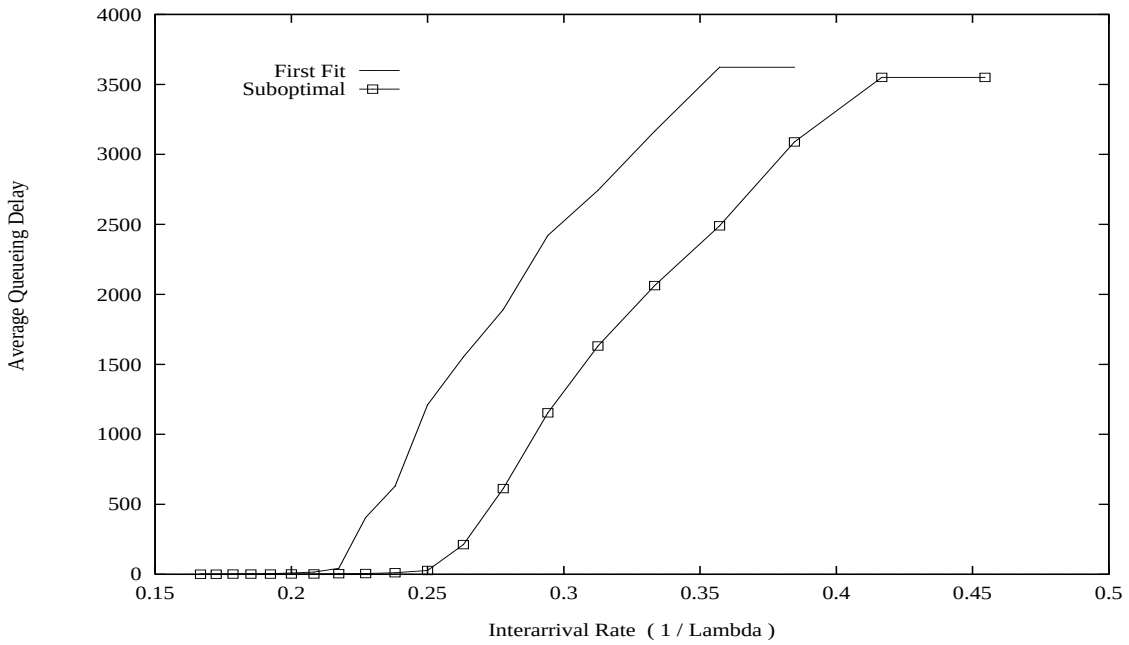


Figure 5.9: Average queueing delay in $G^{1,1,1,1}_{3,3,2,2}$ using the two strategies

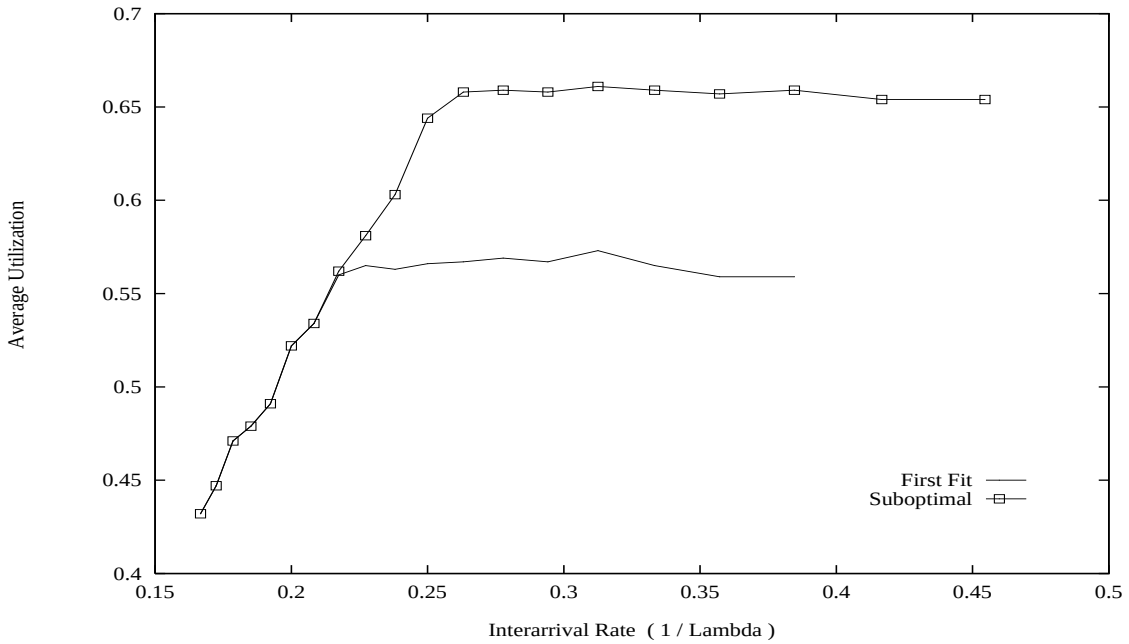


Figure 5.10: Average utilization in $G^{1,1,1,1}_{3,3,2,2}$ using the two strategies

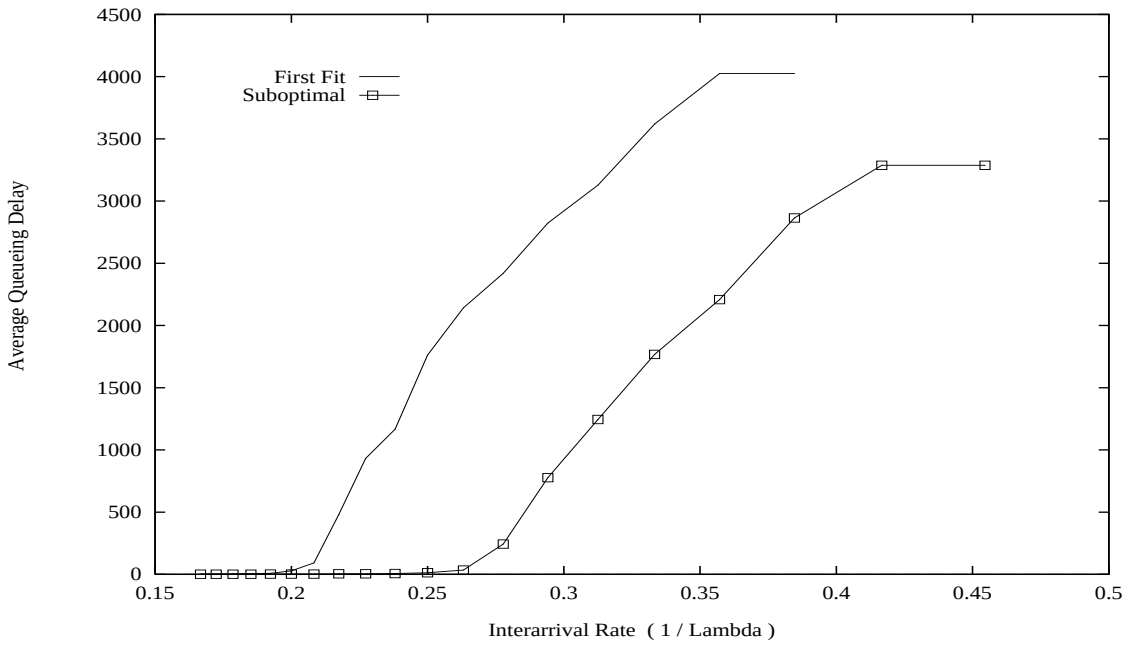


Figure 5.11: Average queueing delay in $G^{2,2,1,1}_{5,4,3,2}$ using the two strategies

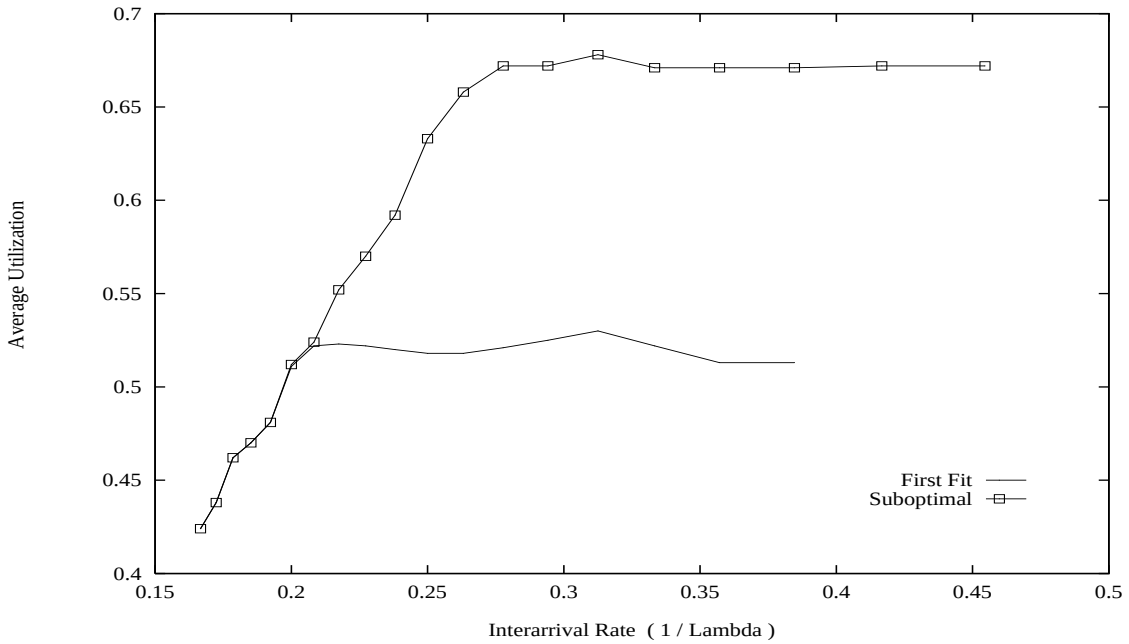


Figure 5.12: Average utilization in $G^{2,2,1,1}_{5,4,3,2}$ using the two strategies

5.3 Multiple permutations

Much like hypercubes, the address tree of a hypercycle cannot recognize all the fragments (or segments) that exist. Since a '$*$' can be in any position of an r-digit address, there are in total $\binom{r}{k}$ placements of k '$*$'s inside r digits. Depending in which positions these '$*$'s are, the rest of the digits can take up a different number of possible values. For example the j-th digit can have up to m_j different values, (from 0 to $m_j - 1$).

Let p_i^k denote the i-th placement of the k '$*$'s, i.e. the set of positions where '$*$'s lie. For example, if we consider a hypercycle with $N = m_3 \times m_2 \times m_1$, the possible 2-fragments have one of the three forms (1): $b_3 * *$, (2): $*b_2*$, (3): $**b_1$, and $p_1^2 = \{1, 2\}$, $p_2^2 = \{1, 3\}$, $p_3^2 = \{2, 3\}$. Since, there are in total $\binom{r}{k}$ such placements, the total number of different k-fragments is equal to

$$\sum_{i=1}^{\binom{r}{k}} \Big(\prod_{\substack{j=1 \\ j \notin p_i^k}}^{r} m_j \Big) \tag{5.1}$$

The address tree can recognize only $m_r \times m_{r-1} \times \cdots \times m_{k+1}$ k-fragments which have to be *primary*, i.e. of the form $b_r b_{r-1} \ldots b_{k+1} *^k$, which gives $p^k = \{1, 2, \ldots, k\}$. Hence, it recognizes only one of the $\binom{r}{k}$ terms of the sum in equation 5.1, just like the buddy strategy recognizes only $1/\binom{n}{k}$ of the total subcubes in Q_n.

To improve the fragment recognition ability, we are going to use more than one address trees concurrently, each address tree representing a permutation of the r factors of N. Note that a permutation of the factors of N, results in new fragment sizes. For example if $N = 3 \times 2$, then 1-fragments of the form b_2* consist of 2 nodes, while the same form of fragments in $N = 2 \times 3$ contain 3 nodes. Actually a permutation of the factors of N creates a new placement of the '$*$'s in the fragment addresses, causing new terms of the sum in equation 5.1 to be recognized.

PROCESSOR ALLOCATION
(Suppose there is a request I for a number of nodes)

For all permutations $\pi = 1, 2, \ldots, P$ do

1. Set $|F|$ = the size of the smallest primary fragment to accommodate the request I.
2. Determine the *least* integer m such that all the allocation bits in the region $\#[m|F|, (m+1)|F| - 1]$ are 0's, and set all the bits in this region to 1's.
3. If such a region was found

   then Allocate the node $MR_{\pi^{-1}}(i)$, for every $i \in \#[m|F|, (m+1)|F| - 1]$ and Stop.

   else Continue with next π.

PROCESSOR RELINQUISHMENT
(Suppose the request that occupied the set of nodes F finished)

1. Reset the i-th allocation bit to 0, for every $MR(i) \in F$.

Figure 5.13: The multiple first fit strategy for hypercycles

The allocation algorithm for the first fit and the suboptimal strategies remains essentially the same but is repeated for the different permutations until we find one that is able to allocate the request. Note that we still allocate *primary fragments* with respect to each permutation, but a fragment that is primary for one permutation, is no longer primary for another because of the re-positioning of the '$*$'s. We give the algorithm for the modified first fit policy, called *multiple first fit*, in Figure 5.13. The *multiple suboptimal* strategy appears in Figure 5.14.

A number of things have to be noted about the algorithms in Figures 5.13, 5.14. First of all, P is the number of chosen permutations. One can include as many permutations as desired, the maximum of course being $r!$. This provides a trade-off between the fragment recognition ability and the complexity of the

PROCESSOR ALLOCATION
(Suppose there is a request I for a number of nodes)

For all permutations $\pi = 1, 2, \ldots, P$ do

1. Set $|S|$ = the size of the smallest segment to accommodate the request I. Also let $|f|$ be the size of the largest primary fragment so that $|S| \geq |f|$ and $|F|$ the size of the smallest primary fragment so that $|S| < |F|$.
2. Determine the *least* integers m, m_s such that all the allocation bits in the region $\#[m|F| + m_s|f|, m|F| + m_s|f| + |S| - 1]$ are 0's, and set all the bits in this region to 1's.
3. If such a region was found

   then Allocate the node $MR_{\pi^{-1}}(i)$, for every $i \in \#[m|F| + m_s|f|, m|F| + m_s|f| + |S| - 1]$ and Stop.

   else Continue with next π.

PROCESSOR RELINQUISHMENT
(Suppose the request that occupied the set of nodes S finished)

1. Reset the i-th allocation bit to 0, for every $MR(i) \in S$.

Figure 5.14: The multiple suboptimal strategy for hypercycles

algorithm. The more permutations used, the more fragments will be recognized but also the more processing time is required for the algorithm, its complexity being $\mathcal{O}(P \times N)$. Finally, $MR_{\pi^{-1}}(i)$ is the address of node i in the original (unpermuted) sequence of factors.

If the original factoring of N is $m_r \times m_{r-1} \times \cdots \times m_1$, denoted by the permutation $(r, r-1, \ldots, 1)$, called the *original permutation*, then

$$\pi = (n_r, n_{r-1}, \ldots, n_1) \text{ where } \pi(i) = n_i, i = 1, 2, \ldots, r$$

defines a new factoring of N as follows:

$$m_{\pi(r)} \times m_{\pi(r-1)} \times \cdots \times m_{\pi(1)}$$

Then a node i with address $MR(i) = (b_r b_{r-1} \dots b_1)_{m_r, m_{r-1}, \dots, m_1}$ in the unpermuted base, has the following address in the permuted factoring of N:

$$MR_\pi(i) = (b_{\pi(r)} b_{\pi(r-1)} \dots b_{\pi(1)})_{m_{\pi(r)}, m_{\pi(r-1)}, \dots, m_{\pi(1)}}$$

Conversely, if π^{-1} is the inverse permutation of π, then node i with address $MR(i) = (b_r b_{r-1} \dots b_1)_{m_{\pi(r)}, m_{\pi(r-1)}, \dots, m_{\pi(1)}}$, has original (unpermuted) address

$$MR_{\pi^{-1}}(i) = (b_{\pi^{-1}(r)} b_{\pi^{-1}(r-1)} \dots b_{\pi^{-1}(1)})_{m_r, m_{r-1}, \dots, m_1}$$

As an example, let $N = 4 \times 3 \times 5$ and let $\pi = (2, 1, 3)$. The new factoring of N, using permutation π, is

$$m_{\pi(3)} \times m_{\pi(2)} \times m_{\pi(1)} = m_2 \times m_1 \times m_3 = 3 \times 5 \times 4$$

The primary fragment $F = 01*$ in the new permutation has address $F_\pi = 1*0$, and it is no longer primary with respect to the original permutation. The inverse permutation of π is $\pi^{-1} = (1, 3, 2)$ which gives

$$m_{\pi^{-1}(2)} \times m_{\pi^{-1}(3)} \times m_{\pi^{-1}(1)} = m_3 \times m_2 \times m_1 = 4 \times 3 \times 5$$

i.e. the original permutation. The address $(203)_{3,5,4}$ actually represents the node $(320)_{4,3,5}$ in the original permutation.

We should also note that allocating nodes by using a permutation of factors is legal, which means that allocated fragments and segments (which are no more fragments or segments in the original address tree) are still closed under the hypercycle routing. This holds because the graphs resulting from different permutations of the factors of N are isomorphic as proved in [19]. Two graphs are isomorphic if there exists a one-to-one mapping between the vertices of the two graphs that preserves the edges. Simply stated, one can redraw one of the graphs in a certain way so that a figure identical to the other graph results.

Consider the two address trees for $G_{2,3}^{1,1}$ in Figure 5.15. The tree on the left corresponds to the original factoring of N as $N = 2 \times 3$. The tree on

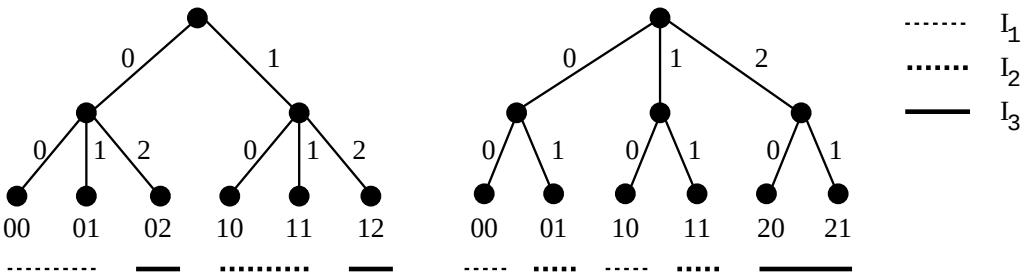


Figure 5.15: Allocating nodes with the multiple suboptimal strategy

the right results from factoring N as $N = 3 \times 2$. We are going to use both permutations to allocate the request sequence $\{|I_1| = 2, |I_2| = 2, |I_3| = 2\}$, using the multiple suboptimal strategy. The list of the allocation bits is initially {0, 0, 0, 0, 0, 0}. The original permutation is $\pi_1 = (2, 1)$ and the second one is $\pi_2 = (1, 2)$. Based on the algorithm in Figure 5.14, π_1 is used first to allocate I_1. Hence, the address tree on the left is searched for a segment of 2 nodes and is allocated the segment $0[0, 1]$. The affected nodes on the second address tree are $(0[0, 1])_\pi = [0, 1]0 = \{00, 10\}$. The allocation bits are now {1, 1, 0, 0, 0, 0}. For I_2, the same procedure results in the segment $1[0, 1]$ being allocated using the original tree. Translating the allocated segment to find the corresponding nodes in the tree on the right, we see that nodes $(1[0, 1])_\pi = [0, 1]1 = \{01, 11\}$ have been allocated. The allocation bits are {1, 1, 0, 1, 1, 0}. Next, the request I_3 arrives and the address tree on the left cannot allocate it because no free segment of two nodes is recognized. Hence the 'for' loop in the algorithm continues with π_2. The second address tree is scanned from left to right and reveals the free segment $2[0, 1]$. In order to update the list of the allocation bits and find the corresponding nodes for π_1, we apply the inverse permutation of π_2, $\pi_2^{-1} = (1, 2)$. Hence $(2[0, 1])_{\pi_2^{-1}} = [0, 1]2 = \{02, 12\}$ in the original permutation. The $(02)_{2,3} = 2$nd and $(1, 2)_{2,3} = 5$th allocation bits are set to 1, hence the final allocation bits are {1, 1, 1, 1, 1, 1}, having the whole graph allocated. It is worth noting that the same sequence of requests could not be allocated using

only π_1 as shown in Theorem 5.3.

As noted above, permuting the factors of the hypercycle results in different fragment and segment sizes. In the previous example, if the multiple first fit strategy was used instead, the third request could not be allocated. The reason is that since π_1 is examined first, the primary fragments to fit the first two requests are of size 3. Hence all of the 6 nodes would be allocated to them, leaving no nodes free for the third request. On the other hand, if π_2 was used first, all three requests would be satisfied because the primary 1-fragments in this permutation have 2 nodes, matching exactly the request sizes.

The obvious improvement to the algorithms in Figures 5.13 and 5.14 is to choose the order in which the different permutations are examined, based on their *redundancy*. When a request comes, we check all the permutations to see how many nodes each permutation requires to allocate it. The permutations are then sorted by increasing order of required nodes. The least redundant is used first. This is the approach that has been implemented in our simulator. The logic behind this is that the fewer nodes we allocate, the greater the probability of allocating a subsequent request.

We simulated both multiple-permutations strategies using randomly selected permutations and compared them to each other and to the first fit and the suboptimal strategies of the previous section. The results, as expected, in terms of both queueing delay and utilization, confirm the improved performance of the new strategies over the earlier ones. The more permutations we use, the better performance we get. Still, the suboptimal strategies exhibit the best behaviour. The results are presented in Figures 5.16 - 5.19 for the hypercycles $G^{1,1}_{3,2}$ and $G^{2,2,1,1}_{5,4,3,2}$.

In the next sections we develop a framework for choosing appropriate permutations. This is a very important issue in order to achieve maximum performance. An important case is examined first, namely the case of all factors being equal to each other. Hypercubes fall in this category. The method developed,

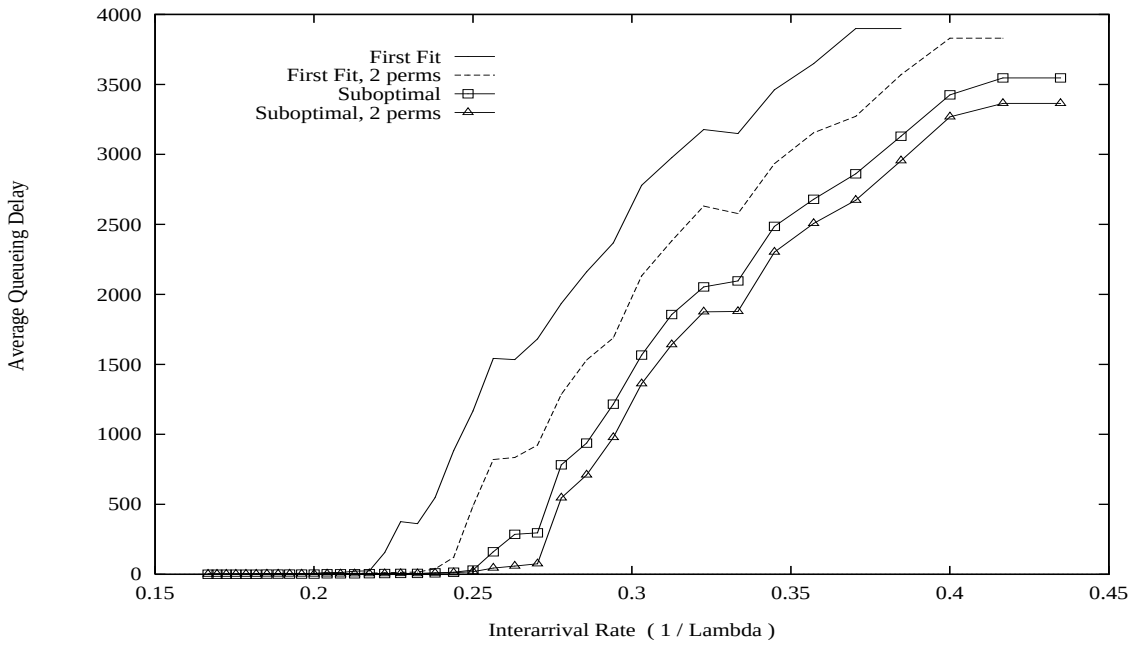


Figure 5.16: Average queueing delay in $G_{3,2}^{1,1}$ using multiple permutations

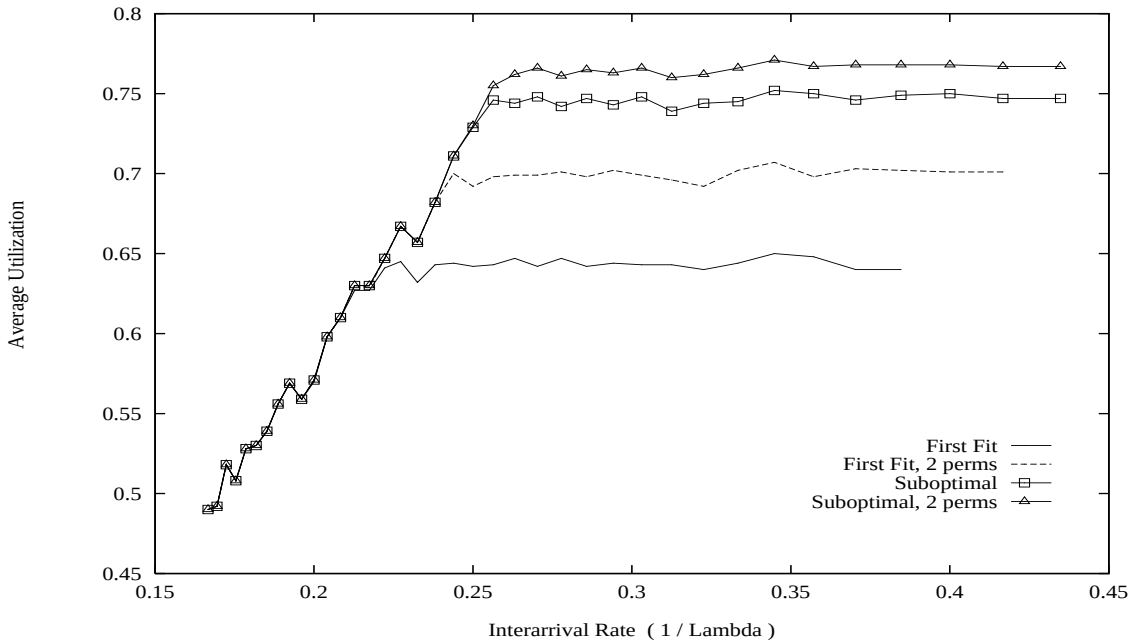


Figure 5.17: Average utilization in $G_{3,2}^{1,1}$ using multiple permutations

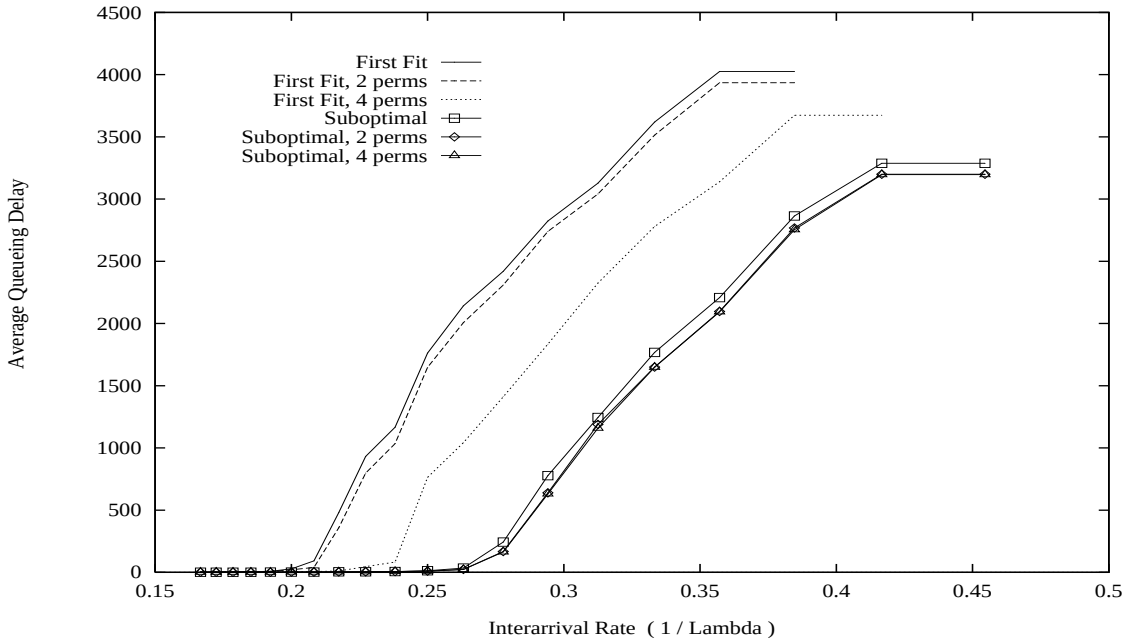


Figure 5.18: Average queueing delay in $G^{2,2,1,1}_{5,4,3,2}$ using multiple permutations

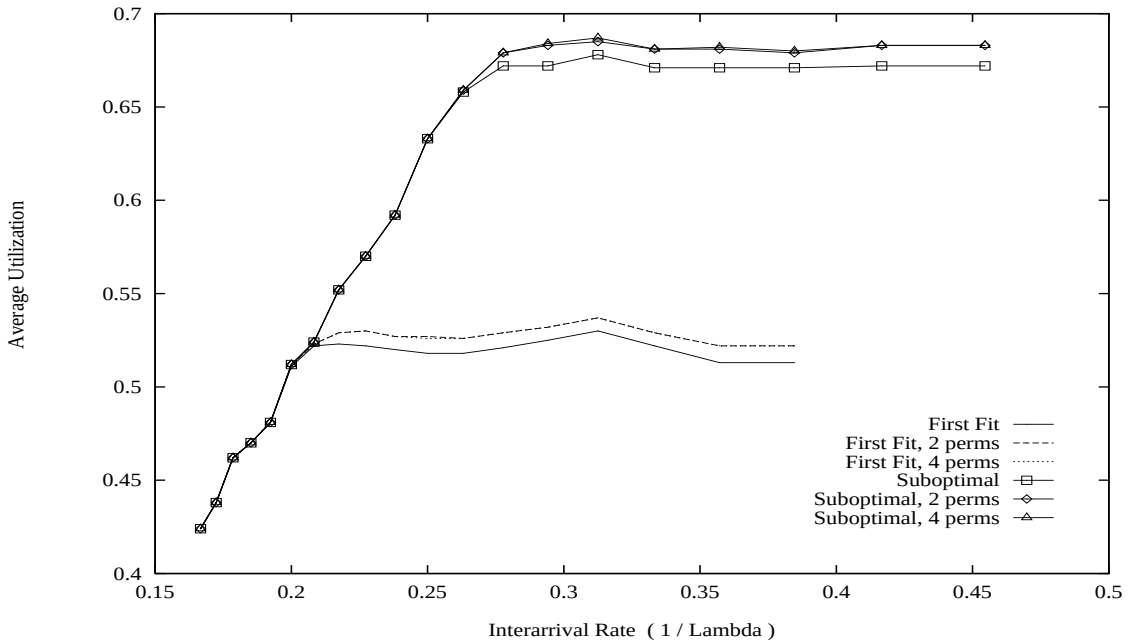


Figure 5.19: Average utilization in $G^{2,2,1,1}_{5,4,3,2}$ using multiple permutations

maximizes the fragment recognition ability of the strategies and for the special case of hypercubes results in better performance over the gray code strategy. The following section concentrates on the rest of the hypercycles, deriving a way to choose the optimal permutation to allocate a series of requests for which some facts about their statistical nature are known beforehand.

5.4 Allocation in uniform hypercycles

In the previous section we introduced an allocation method that used multiple permutations of the factors of N in order to recognize more fragments. When the factors of N are not equal to each other, each permutation results in different primary fragment sizes and consequently different behavior. Thus we have to have a method of choosing the best possible permutations so that the performance is maximized. In this section, we present such a method for the important case of uniform hypercycles.

Definition 5.4 A hypercycle is *uniform* if all the factors of N are equal.

Hypercubes are uniform hypercycles, of the form $G^{1,1,\ldots,1}_{2,2,\ldots,2}$. k-ary hypercubes are uniform hypercycles of the form $G^{1,1,\ldots,1}_{k,k,\ldots,k}$. Uniform hypercycles have the property that *every permutation of the factors of* N results in the same fragment sizes, and recognizes a constant number of fragments. If $N = m^r = m \times m \times \cdots \times m$, then, for every factor permutation, a k-fragment (primary or not) has m^k nodes and there are m^{r-k} k-fragments recognized by the corresponding address tree, out of $\binom{r}{k} m^{r-k}$ possible k-fragments.

The purpose of using multiple permutations is to recognize more fragments than the single-permutation first fit or suboptimal strategies. The objective now is to *maximize* the number of recognized fragments. Consider the hypercycle $G^{1,1,1}_{3,3,3}$ and the factor permutations $\pi_1 = (3,2,1)$ and $\pi_2 = (2,3,1)$ used concurrently. Although each permutation recognizes 3^{3-k} k-fragments, there

are fragments common to both permutations, reducing the effective number of recognized k-fragments below $2 \times 3^{3-k}$. As an example the primary fragment $00*$ in π_1 is translated as $(00*)_{\pi_2} = 00*$ in π_2, i.e. it is a primary fragment in π_2, hence π_2 recognizes it, too. In order to maximize the performance we have to select permutations that recognize completely different primary fragments. The next theorem gives a way to achieve that.

Lemma 5.1 *A permutation $\pi = (n_r, n_{r-1}, \ldots, n_1)$ recognizes different primary k-fragments from the original permutation, if, and only if, $\{n_1, n_2, \ldots, n_k\} \neq \{1, 2, \ldots, k\}$.*

Proof: First we are going to prove that if $\{n_1, n_2, \ldots, n_k\} = \{1, 2, \ldots, k\}$, then there exists at least one common primary fragment to the two permutations. Consider the node with address $b_r b_{r-1} \ldots b_{k+1} b_k \ldots b_1$ in the original permutation. The same node will have $b_{n_r} b_{n_{r-1}} \ldots b_{n_{k+1}} b_{n_k} \ldots b_{n_1}$ as its address in π, where $b_{n_1}, b_{n_2}, \ldots, b_{n_k}$ are the digits $b_1, b_2, \ldots, b_k$, in a different order, i.e. none of the former is digit b_i, for $i > k$. Then the fragment $b_r b_{r-1} \ldots b_{k+1} *^k$ is the fragment $b_{n_r} b_{n_{r-1}} \ldots b_{n_{k+1}} *^k$ in π. The fragment $00 \ldots 0*^k$ is consequently a primary fragment in both permutations, hence both recognize it.

Conversely, if $\{n_1, n_2, \ldots, n_k\} \neq \{1, 2, \ldots, k\}$, then we will prove that there is no fragment in common. If there was one, then based on the first fit strategy, it will have to be a primary fragment both in the original permutation and π. Hence its address must be of the form $F = b_r b_{r-1} \ldots b_{k+1} *^k$ in the former and $F_\pi = b_{n_r} b_{n_{r-1}} \ldots b_{n_{k+1}} *^k$ in the latter permutation. But then the k least significant digits of F are all mapped in the k least significant digits of F_π, i.e. no '$*$' appears in digits $k+1, k+2, \ldots, r$ of F_π. This means that $\{n_1, n_2, \ldots, n_k\} = \{1, 2, \ldots, k\}$ which is a contradiction to our hypothesis that $\{n_1, n_2, \ldots, n_k\} \neq \{1, 2, \ldots, k\}$. Hence there is no common primary fragment between the original permutation and π.

□

Definition 5.5 Two permutations are called *fragment disjoint* if they do not have any primary k-fragment in common, for all $k = 1, 2, \ldots, r-1$.

Theorem 5.4 *A permutation $\pi = (n_r, n_{r-1}, \ldots, n_1)$ is fragment disjoint with the original permutation, if, and only if, $\{n_1, n_2, \ldots, n_k\} \neq \{1, 2, \ldots, k\}$, for all $k = 1, 2, \ldots, r-1$.*

Proof: Apply Lemma 5.1 for all $k = 1, 2, \ldots, r-1$.

□

We can now decide which permutations should be selected to maximize the number of recognized fragments. For example $\pi_1 = (3, 2, 1)$ and $\pi_2 = (1, 2, 3)$ do not have any primary fragments in common, while $\pi_1 = (3, 2, 1)$ and $\pi_2 = (3, 1, 2)$ have some 2-fragments in common because the sets $\{1,\ 2\}$ and $\{2,\ 1\}$ of their rightmost 2 elements contain the same numbers.

Simulations confirmed the expected improvement in performance that results from choosing sets of fragment disjoint permutations. An example is given in Table 5.1. We simulated the multiple first fit strategy using all possible sets of 2 permutations in Q_3. Note that for the case of hypercubes, the first fit strategy is identical to the buddy strategy, since fragments represent subcubes. The suboptimal strategy allocates subcubes too, and is thus also identical to the buddy strategy. In these experiments we chose to let one request appear every clock tick, i.e. in a constant, not a poisson, manner. Although this is a crude approximation to reality, many of the results for hypercubes that have appeared in the literature use it. T is the length of the simulation and in our experiments it was 100 clock ticks. E is the interval in which the execution time of the incoming tasks is uniformly distributed. We let tasks execute for a random number of clock ticks between 2 and 6 (inclusive). Finally, we repeated the simulations 3,000 times and averaged the results. The queueing delay was measured, averaged over the number of allocated tasks during the 3,000 tests. Also, the requests are assumed to ask for subcubes, up to dimension 2, i.e.

$T = 100$, $E \in [2, 6]$, Repeats = 3000

PERMUTATIONS	*DELAY*
(3,2,1), (3,1,2)	12.744
(3,2,1), (1,3,2) †	12.710
(3,2,1), (1,2,3) †	12.750
(3,2,1), (2,1,3) †	12.362
(3,2,1), (2,3,1)	12.375

† Fragment disjoint

Table 5.1: Delays of all sets of two permutations in Q_3

there is no request for the whole graph. Note that in Table 5.1 the best set of permutations (the one with the lowest delay) is the set {(3, 2, 1), (2, 1, 3)}, that includes two fragment disjoint permutations.

From Table 5.1 one can see that not all the sets of fragment disjoint permutations result in lower delays. This is a result of the way the algorithms allocate subcubes (or fragments). Permutation $\pi_1 = (3, 2, 1)$ is always examined first. The second permutation, π_2, is to be examined only if one of the following conditions was true:

1. There was *not* enough space in the graph, so π_1 was not able to allocate the request.

2. There *was* enough space but π_1 could not recognize a fragment able to allocate the request.

In case (1) π_2 will also be unable to allocate the request. In case (2) though, π_2 will be able to allocate the request only if it recognized a fragment that included the needed free nodes. When π_1 allocates a fragment, it actually allocates one of the buddies (Definition 5.1) of a higher dimensioned fragment, leaving the rest of them as holes. If π_1 cannot allocate a subsequent request, although there exists a fragment in some permutation to fit the request, it is because the holes

left in the graph are not buddies. Hence π_2 will be useful only if it can combine holes that are not buddies in π_1, to form a fragment, i.e. make the holes buddies. So, even if π_2 recognizes completely different fragments, it may not be able to allocate a request that π_1 could not allocate.

Fragments that cannot be allocated by the original permutation due to case (2) above, have a chance to be allocated only if fragments that are not buddies can be combined by the second permutation to form higher dimensioned fragments. This can be accomplished only with generalized shifts, as proven in Theorem 5.5.

Definition 5.6 A permutation $\pi = (n_r, n_{r-1}, \ldots, n_1)$ is called a *generalized shift* if $\{1, 2, \ldots, k\} \subset \{n_1, n_2, \ldots, n_{k+1}\}$, for every $k = 1, 2, \ldots, r-1$.

Generalized shifts map the set of numbers $\{1, 2, \ldots, k\}$ to their first $k+1$ digits. Permutations (3, 2, 1, 4) and (4, 2, 3, 1) are examples of generalized shifts. Generalized shifts have the unique property that any k-fragment of the original permutation (permutation (4, 3, 2, 1)) belongs in a $(k+1)$-fragment in the generalized shift, as proven in the following theorem.

Theorem 5.5 *A primary k-fragment in the original permutation belongs to a primary $(k+1)$-fragment in a permutation π, for every $k = 1, 2, \ldots, r-1$, if and only if π is a generalized shift.*

Proof: Consider the fragment $F = b_r b_{r-1} \ldots b_{k+1} *^k$.

At first assume that π is a generalized shift. Then we will have $F_\pi = b_{n_r} b_{n_{r-1}} \ldots b_{n_{k+2}} a_{n_{k+1}} \ldots a_{n_1}$ where the digits $a_{n_1}, a_{n_2}, \ldots, a_{n_{k+1}}$ include the k '$*$'s of F, according to definition 5.6. Then the fragment $b_{n_r} b_{n_{r-1}} \ldots b_{n_{k+2}} *^{k+1}$ in π includes F.

Conversely, if F belongs in a primary fragment $F_\pi = b_{n_r} b_{n_{r-1}} \ldots b_{n_{k+2}} *^{k+1}$ in a permutation π, then the k '$*$'s of F must be included in the $k+1$ least significant digits of the address of F_π, which means that π maps numbers $1, 2, \ldots, k$

into its first $k+1$ digits. Since k can take any value from 1 to $r-1$, by definition 5.6 π is a generalized shift.

□

Also, one should notice that a generalized shift has to recognize completely different fragments from the original permutations because otherwise, fragments that are buddies in the original permutation, may be combined in the same fragment in the generalized shift. Hence, a generalized shift that has the additional property of being fragment disjoint with the original permutation, should give the best results.

Theorem 5.6 *The only generalized shift that is fragment disjoint with the original permutation is the clockwise rotation* $\pi = (r-1, r-2, \ldots, 1, r)$.

Proof: We use Theorem 5.4 and Definition 5.6 to construct the permutation that satisfies both. Let $\pi = (n_r, n_{r-1}, \ldots, n_1)$.

In order for it to be fragment disjoint with the original permutation, we must have: $n_1 \neq 1$ and $\{n_1, n_2\} \neq \{1, 2\}$. Also, in order to be a generalized shift, $\{n_1, n_2\} \supset \{1\}$. Hence n_2 has to be equal to 1.

Assume now that $n_2 = 1, n_3 = 2, \ldots, n_k = k-1$. We are going to show that $n_{k+1} = k$. The following have to be satisfied at the same time:

$$\{n_1, n_2, \ldots, n_k\} \neq \{1, 2, \ldots, k\}$$

which means that n_1 cannot be equal to k, and

$$\{n_1, n_2, \ldots, n_{k+1}\} \supset \{1, 2, \ldots, k\}$$

which means that n_{k+1} has to be equal to k.

Thus for every $k = 2, 3, \ldots, r$, $n_k = k-1$. Then n_1 has to be equal to the only number left, r. Hence $\pi = (r-1, r-2, \ldots, 1, r)$.

□

Tables 5.2 - 5.5 present the results of the simulations done for a number of graphs. Choosing the clockwise rotation as the second permutation shows the best performance in all cases. Tables 5.2 - 5.4 give the results for the hypercubes Q_3, Q_4 and Q_5 respectively. The results confirm the theory developed above. Choosing a clockwise rotation as the second permutation outperforms the gray codes strategy also by a significant factor. Finally, the results for the uniform hypercycle $G^{1,1,1,1}_{3,3,3,3}$ are given in Table 5.5 and once more the theoretic results are confirmed.

5.5 Matching the load characteristics

In section 5.3 we discussed the use of various permutations to improve the performance of the first fit and the suboptimal strategies. We also noted that the request sequence {2, 2, 2} could be allocated on $G^{1,1}_{2,3}$ if we used the permutation $\pi = (1, 2)$ because this results in primary fragments of 2 nodes each. The reasoning was that the permutation that allocates the least nodes to a request should be preferred because the probability that there is enough space for subsequent requests is higher. But if we knew beforehand that the requests would ask for 3 nodes instead of 2, then the permutation $\pi = (2, 1)$ is better because with the first fit strategy a request for 3 nodes would be allocated the whole graph if $\pi = (1, 2)$ was used. Hence we conclude that determining which permutation is best, actually depends on the nature of the requests.

In this section, we derive a heuristic procedure to find the best permutation to be used by the single-permutation first fit strategy when we know in advance some statistical characteristics of the types of requests. We assume that the probability for a request to ask for i nodes is known and is equal to p_i, for every $i = 1, 2, \ldots, N$, satisfying $\sum_{i=1}^{N} p_i = 1$. We use the term *load characteristics* to refer to this statistical information. Note that the following theory does not apply to uniform hypercycles because in this case each permutation results in

$T = 100$, $E \in [2, 6]$, Repeats = 3000

STRATEGY		*DELAY*
Buddy (First Fit)		12.791
Gray Codes		12.572
2 permutations	(3,2,1), (1,2,3)	12.750
	(3,2,1), (3,1,2)	12.744
	(3,2,1), (1,3,2)	12.710
	(3,2,1), (2,3,1)	12.375
	(3,2,1), (2,1,3) †	12.362

† Clockwise Rotation

Table 5.2: Delays of various strategies in Q_3

$T = 100$, $E \in [2, 6]$, Repeats = 3000

STRATEGY		*DELAY*
Buddy (First Fit)		7.048
Gray Codes		6.798
2 permutations	(4,3,2,1), (1,4,3,2)	7.022
	(4,3,2,1), (3,1,4,2)	6.534
	(4,3,2,1), (1,3,4,2)	7.028
	(4,3,2,1), (2,4,1,3)	6.883
	(4,3,2,1), (1,4,2,3)	7.029
	(4,3,2,1), (2,1,4,3)	6.995
	(4,3,2,1), (1,2,4,3)	7.036
	(4,3,2,1), (2,3,1,4)	6.952
	(4,3,2,1), (3,1,2,4)	6.543
	(4,3,2,1), (1,3,2,4)	7.035
	(4,3,2,1), (2,1,3,4)	6.995
	(4,3,2,1), (1,2,3,4)	7.038
	(4,3,2,1), (3,2,1,4) †	6.520

† Clockwise Rotation

Table 5.3: Delays of various strategies in Q_4

$T = 100$, $E \in [3, 7]$, Repeats = 3000

STRATEGY		*DELAY*
Buddy (First Fit)		9.092
Gray Codes		8.814
2 permutations	(5,4,3,2,1), (1,2,3,4,5)	9.091
	(5,4,3,2,1), (1,3,2,4,5)	9.091
	(5,4,3,2,1), (1,5,4,3,2)	9.088
	(5,4,3,2,1), (2,1,5,4,3)	9.080
	(5,4,3,2,1), (4,3,2,1,5) †	8.497

† Clockwise Rotation

Table 5.4: Delays of various strategies in Q_5

$T = 300$, $E \in [5, 8]$, Repeats = 3000

STRATEGY		*DELAY*
First Fit		6.229
2 permutations	(4,3,2,1), (1,4,3,2)	6.228
	(4,3,2,1), (3,1,4,2)	6.187
	(4,3,2,1), (1,3,4,2)	6.229
	(4,3,2,1), (2,4,1,3)	6.223
	(4,3,2,1), (1,4,2,3)	6.229
	(4,3,2,1), (2,1,4,3)	6.246
	(4,3,2,1), (1,2,4,3)	6.229
	(4,3,2,1), (2,3,1,4)	6.231
	(4,3,2,1), (3,1,2,4)	6.187
	(4,3,2,1), (1,3,2,4)	6.229
	(4,3,2,1), (2,1,3,4)	6.247
	(4,3,2,1), (1,2,3,4)	6.229
	(4,3,2,1), (3,2,1,4) †	6.186

† Clockwise Rotation

Table 5.5: Delays of various strategies in $G_{3,3,3,3}^{1,1,1,1}$

the same primary fragment sizes.

We know that for $N = m_r \times m_{r-1} \times \cdots \times m_1$ there are in total $r!$ possible permutations of the factors, each permutation resulting in different fragment sizes. Suppose that the j-th permutation, π_j, factors N as $N = m_{\pi_j(r)} \times m_{\pi_j(r-1)} \times \cdots \times m_{\pi_j(1)}$. Denote the number of nodes that are needed to accommodate a request for i nodes, using π_j, by $|R_{\pi_j}(i)|$. If the first fit strategy is used then $|R_{\pi_j}(i)|$ will be equal to the size of a primary fragment, e.g. $|R_{\pi_j}(i)| = m_{\pi_j(k)} \times m_{\pi_j(k-1)} \times \cdots \times m_{\pi_j(1)}$ if a k-fragment is needed. If the suboptimal strategy is used then $|R_{\pi_j}(i)|$ will be equal to the size of a segment. Then the *average number of allocated nodes per request* for permutation π_j is given by

$$A_{\pi_j} = \sum_{i=1}^{N} p_i |R_{\pi_j}(i)| \tag{5.2}$$

The best permutation is the one that *minimizes* A_{π_j} because it allocates less nodes on the average, thus increasing the probability of finding free space for subsequent requests. Hence the preferred permutation is permutation π where

$$\pi \in \{\pi_k \mid A_{\pi_k} = \min\{A_{\pi_j}\},\ j = 1, 2, \ldots, r!\} \tag{5.3}$$

For example, suppose we use the first fit strategy in the hypercycle $G^{1,1}_{2,3}$, and let $p_1 = 0.1$, $p_2 = 0.2$, $p_3 = 0.4$, $p_4 = p_5 = p_6 = 0.1$, where requests ask frequently for 2 or 3 nodes. If $\pi_1 = (2, 1)$ is used then for a request of 1 node, 1 node exactly will be allocated, hence $|R_{\pi_1}(1)| = 1$. For a request of 2 nodes, a 3-node fragment will be allocated, hence $|R_{\pi_1}(2)| = 3$. Going on in this manner we get $|R_{\pi_1}(3)| = 3$, $|R_{\pi_1}(4)| = |R_{\pi_1}(5)| = |R_{\pi_1}(6)| = 6$. If $\pi_2 = (1, 2)$ is used we will have $|R_{\pi_2}(1)| = 1$, $|R_{\pi_2}(2)| = 2$, $|R_{\pi_2}(3)| = |R_{\pi_2}(4)| = |R_{\pi_2}(5)| = |R_{\pi_2}(6)| = 6$. Computing A_{π_j} for the two permutations, we get

$$A_{\pi_1} = \sum_{i=1}^{6} p_i |R_{\pi_1}(i)| = 3.7$$

$$A_{\pi_2} = \sum_{i=1}^{6} p_i |R_{\pi_2}(i)| = 4.7$$

Hence permutation π_1 is best. If we had, though, $p_3 = 0.1$ and $p_2 = 0.5$, the same calculations would give π_2 as the best permutation.

To confirm that the proposed heuristic actually reveals the best permutation, we simulated the first fit strategy in many hypercycles, using various load characteristics. We exhaustively simulated all the possible permutations on some cases and the results were in agreement with the above heuristic. Figures 5.20 to 5.25 present the results for $G^{2,2,1,1}_{5,4,3,2}$ where we compare the 'best' permutation with another permutation. We chose the unpermuted basis (permutation (4, 3, 2, 1)) arbitrarily as the latter. In Figures 5.20 and 5.21, we assume that the load characteristics are uniform i.e. $p_i = 1/120$, for every $i = 1, 2, \ldots, 120$. Equations 5.2 and 5.3 gave permutation (1, 2, 3, 4) as the best and the results confirm that. In Figures 5.22 and 5.23, the requests are assumed to ask for a number of nodes that follows the gaussian distribution with mean 20.0 and variance 7.5. This is a case where requests show a preference towards a small number of nodes. The calculated best permutation (2, 1, 3, 4) indeed performs better. Finally, in Figures 5.24 and 5.25 the preference is for a large number of nodes, obeying the gaussian distribution with mean 60.5 and variance 7.5. The best permutation for this case is (1, 4, 3, 2) and the results confirm that. One can see that the improvement in performance is dramatic in all the cases.

In conclusion we see that the formulas developed in this section do indeed provide us with a means of finding the best permutation to match the load characteristics. Of course we made the assumption that these characteristics were given. Although in a new environment one may not be able to calculate explicitly the probabilities p_i, a rough approximation can usually be predicted. One would then end up with more that one permutations with low A_{π_j} but any of those permutations should perform better that a randomly chosen one. Also, the operating system could gather statistics about the incoming requests and after an acceptable period of time, the statistics could be used to dynamically change the permutation used for allocating the processors in the machine.

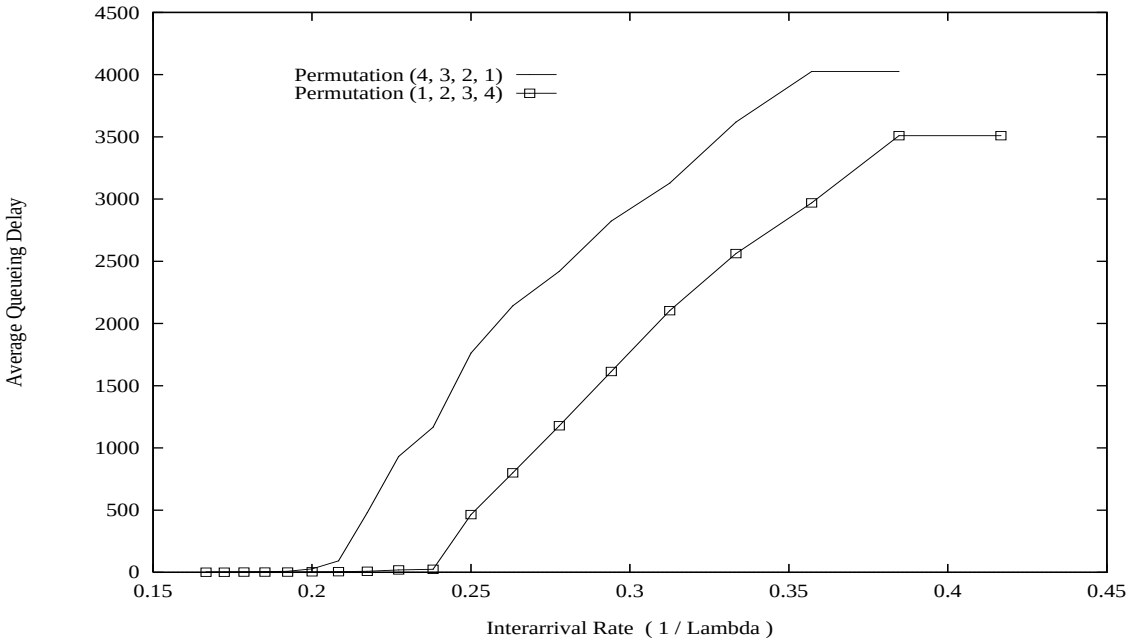


Figure 5.20: Queueing delay in $G^{2,2,1,1}_{5,4,3,2}$, uniform load

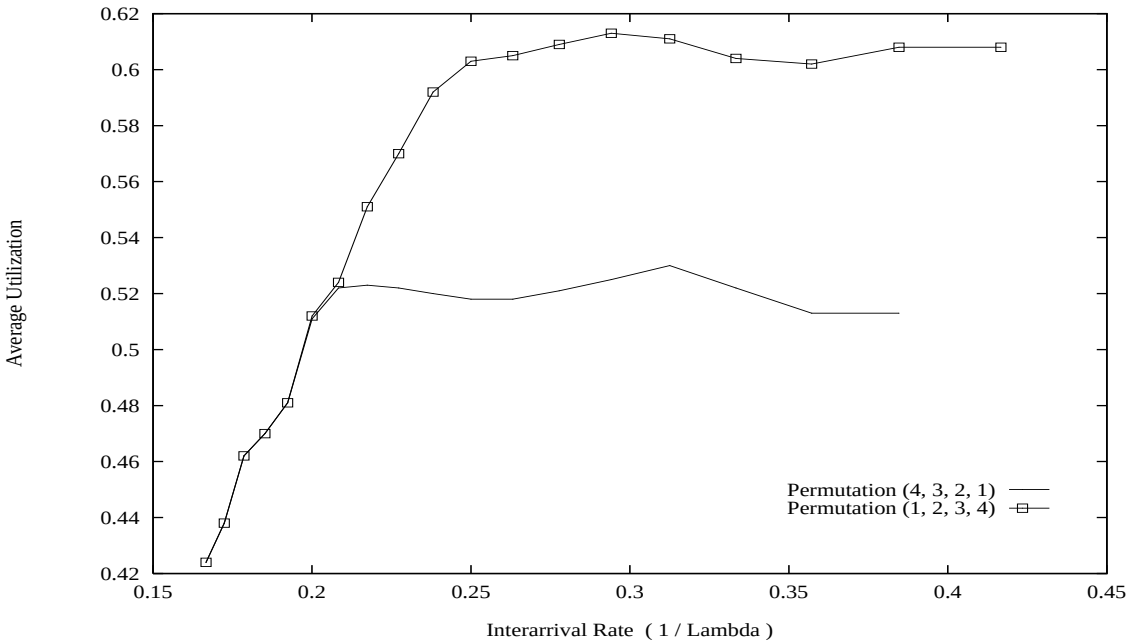


Figure 5.21: Utilization in $G^{2,2,1,1}_{5,4,3,2}$, uniform load

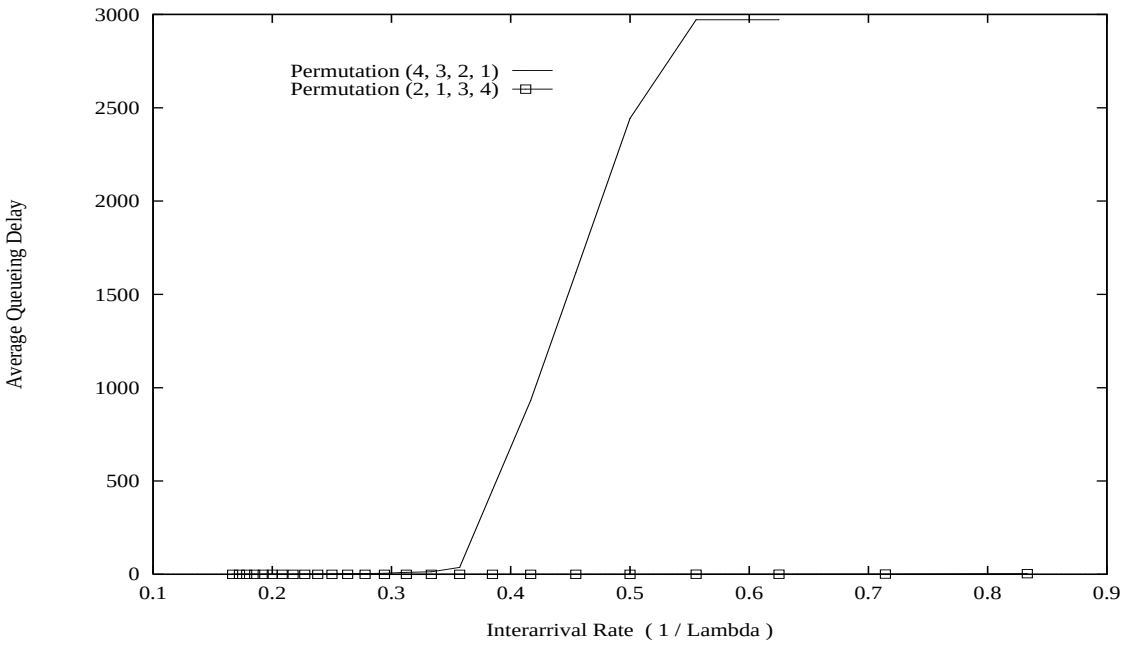


Figure 5.22: Queueing delay in $G^{2,2,1,1}_{5,4,3,2}$, gaussian load (mean = 20.0, var = 7.5)

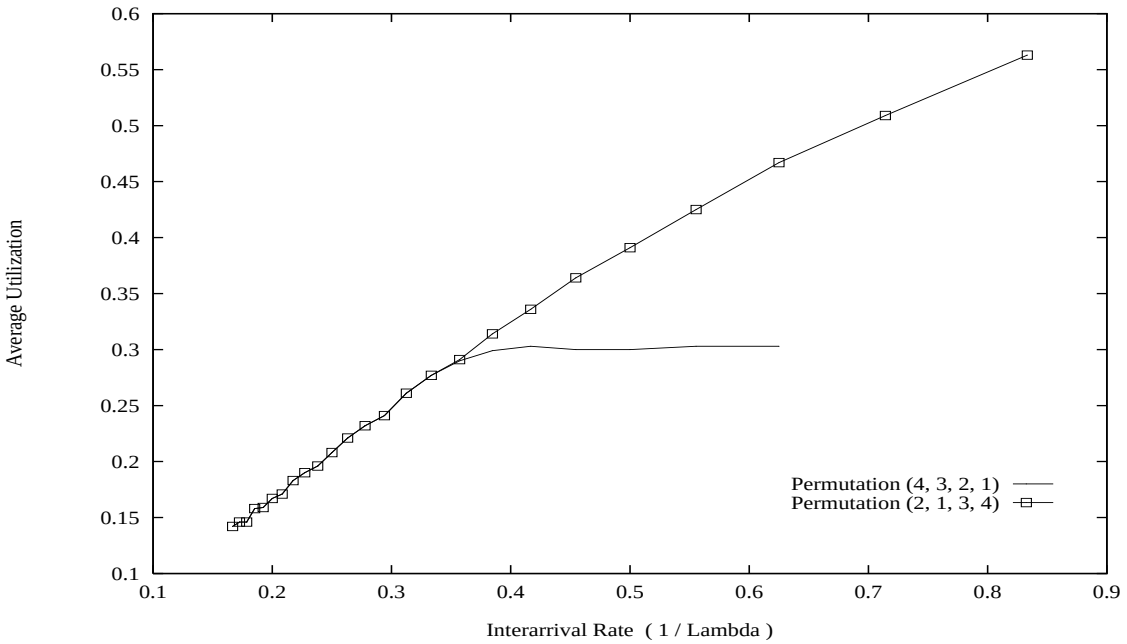


Figure 5.23: Utilization in $G^{2,2,1,1}_{5,4,3,2}$, gaussian load (mean = 20.0, var = 7.5)

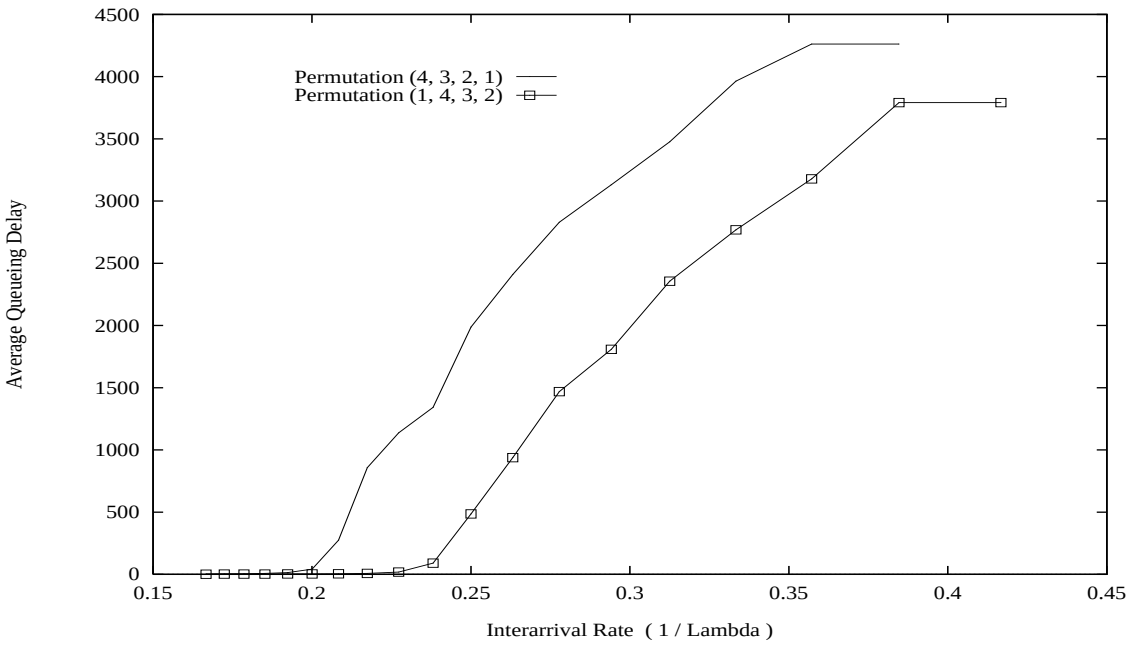


Figure 5.24: Queueing delay in $G^{2,2,1,1}_{5,4,3,2}$, gaussian load (mean = 60.5, var = 7.5)

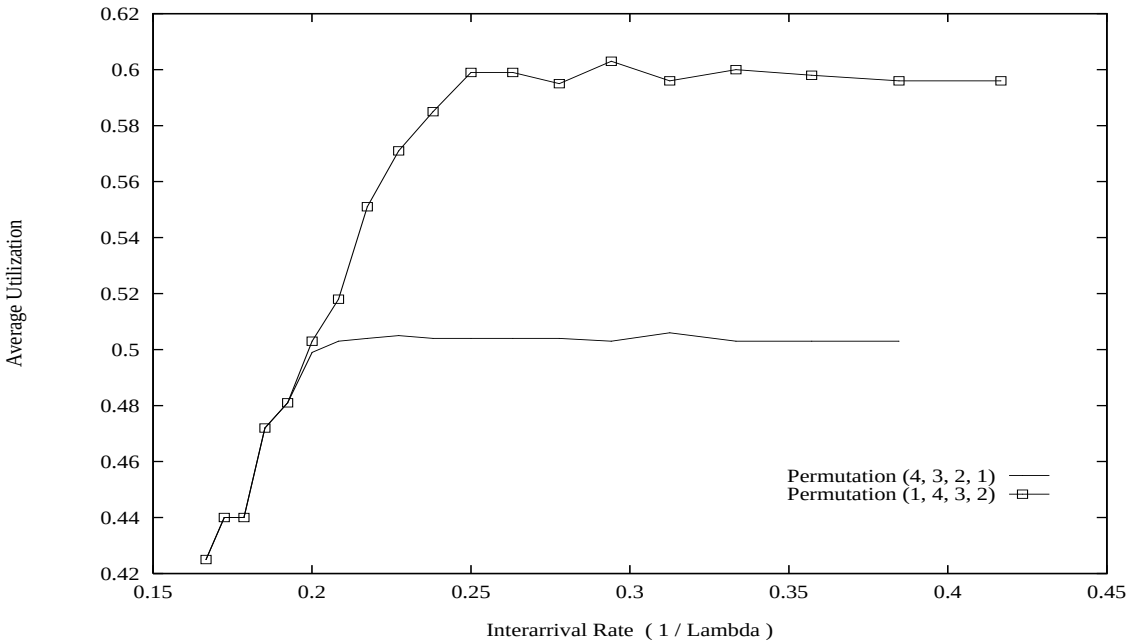


Figure 5.25: Utilization in $G^{2,2,1,1}_{5,4,3,2}$, gaussian load (mean = 60.5, var = 7.5)

Chapter 6

Conclusion

In this thesis we dealt with two different problems, namely the broadcasting of messages and the allocation of processors in hypercycles. Hypercycles, being a wide class of static multiprocessor interconnection networks, offer a higher degree of flexibility as compared to other popular interconnection structures. The availability of analytical expressions for the routing function provides a means for comprehensive mathematical treatment and easy hardware implementation ([28]). The two subjects of this study were meant to provide a background based on which a practical implementation of a hypercycle-based parallel system can be used as efficiently as possible.

Broadcasting is closely associated with the topology of the interconnection network. It is an essential part of the hardware portion of the system that takes care of the routing and in general the low-level communication between the processors. We developed a broadcasting algorithm that guarantees the nonredundant delivery of a message to all nodes in the graph in the minimum time. The hardware requirements for this algorithm were also found to be very small.

We then discussed the general problem of allocating nodes in a parallel machine and we proposed a number of strategies to efficiently solve this problem with respect to hypercycles. A framework was also developed, based on which,

one is provided with a means of choosing between alternative strategies. A part of the results can be applied to hypercubes which are a subset of hypercycles, resulting in better performance as compared to the strategies of the same complexity proposed in literature.

Finally, we built a simulator to examine all the allocation strategies and confirm the theoretical results. The simulator is presented in Appendix A together with the source code, written in the C programming language.

6.1 Future work

There are, still, many issues to be examined for hypercycles. Two important theoretical problems that are related to processor allocation, are finding the best hypercycle for a given set of requests and balancing the load on the processors. The first one is closely related to the problem of matching the load characteristics, explored in Section 5.5. One is given in advance some statistical information about the tasks that are going to appear and the objective is to find the best hypercycle to use. This involves trade-offs between the queueing delays of the tasks and the cost of the graph (being attributed to the number of nodes, links etc.).

In the load balancing problem, each processing node is actually allocated more than one job at a time. In most cases, some nodes will have a large number of jobs allocated while others remain idle or lightly loaded. One has to derive an efficient algorithm to reallocate the jobs on the processors so that the load is uniformly distributed over them. This will result in better utilization of the resources and potentionaly shorter running times. Also, the broadcasting algorithm presented in this thesis should be useful in a situation like this because a processor can determine which are the lightly loaded nodes by exchanging messages with all of them.

Finally, the current status of the hypercycle router chip ([28]) has to be

advanced in order to incorporate the broadcasting principles developed here. Given the simplicity of the proposed algorithm, the implementation should actually take a relatively small amount of effort.

Bibliography

[1] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Procesing.* McGraw-Hill, 1984.

[2] G. H. Barnes et al., "The Illiac IV computer," *IEEE Transactions on Computers*, vol. C-17, pp. 144 – 151, August 1968.

[3] K. E. Batcher, "Design of a massively parallel processor," *IEEE Transactions on Computers*, vol. C-29, pp. 836 – 840, September 1980.

[4] J. Rattner, "Concurrent processing: A new direction in scientific computing," in *1985 National Computer Conference*, vol. 54, pp. 764 – 771, AFIPS Press, 1985.

[5] C. L. Seitz, "The cosmic cube," *Communications of the ACM*, vol. 28, pp. 22–33, January 1985.

[6] W. D. Hillis, *The Connection Machine: A Computer Architecture Based on Cellular Automata.* Physica, 1984.

[7] D. H. Lawrie, "Access and alignemnet of data in an array processor," *IEEE Transactions on Computers*, vol. C-24, pp. 1145 – 1155, December 1975.

[8] J. H. Patel, "Performance of processor-memory interconnections for multiprocessors," *IEEE Transactions on Computers*, vol. C-30, pp. 771–780, October 1981.

[9] V. E. Benes, *Mathematical Theory of Connecting Networks and Telephone Traffic.* New York: Academic Press, 1965.

[10] D. A. Reed and H. D. Schwetman, "Cost-performance bounds for multimicrocomputer networks," *IEEE Transactions on Computers*, vol. C-32, pp. 83 – 95, January 1983.

[11] D. A. Reed and D. C. Grunwald, "The performance of multicomputer interconnection networks," *IEEE Computer*, vol. 20, pp. 63 – 73, June 1987.

[12] A. Gibbons, *Algorithmic Graph Theory*. Cambridge: Cambrige University Press, 1989.

[13] N. Deo, *Graph Theory with Applications to Engineering and Computer Science*. Prentice-Hall, 1974.

[14] H. Sullivan and T. R. Bashkow, "A large scale, homogenous, fully distributed parallel machine, I," in *4th Symposium on Computer Architecture*, pp. 105 – 117, April 1977.

[15] Y. S. Saad and M. H. Schultz, "Topological properties of hypercubes," *IEEE Transactions on Computers*, vol. C-37, pp. 867 – 872, July 1988.

[16] W. Dally, "Performance analysis of k-ary n-cube interconnection networks," *IEEE Transactions on Computers*, vol. C-39, pp. 775 – 784, June 1990.

[17] L. N. Bhuyan and D. P. Agrawal, "Design and performance of generalized interconnection networks," *IEEE Transactions on Computers*, vol. C-32, pp. 1081 – 1090, December 1983.

[18] L. N. Bhuyan and D. P. Agrawal, "Generalized hypercube and hyperbus structures for a computer network," *IEEE Transactions on Computers*, vol. C-33, pp. 323 – 333, April 1984.

[19] N. J. Dimopoulos, R. D. Rasmussen, G. S. Bolotin, B. F. Lewis, and R. M. Manning, "Hypercycles, interconnection networks with simple rout-

ing strategies," in *Canadian Conference on Electrical and Computer Engineering*, (Vancouver, B.C., Canada), pp. 577 – 580, November 1988.

[20] N. J. Dimopoulos, D. Radvan, and K. F. Li, "Performance evaluation of the backtrack to the origin and retry routing for hypercycle based interconnection networks," in *10th International Conference on Distributed Systems*, (Paris, France), pp. 279 – 284, June 1990.

[21] N. J. Dimopoulos, M. Chowdhury, R. Sivakumar, and V. Dimakopoulos, "Routing in hypercycles. Deadlock free and backtracking strategies," in *PARLE '92 Parallel Architectures and Languages Europe*, (France), June 1992.

[22] F. Boesch and R. Tindell, "Circulants and their connectivities," *Journal of Graph Theory*, vol. 8, pp. 487 – 499, 1984.

[23] R. S. Wilkov, "Analysis and design of reliable computer networks," *IEEE Transactions on Communications*, vol. COM-20, pp. 660 – 678, June 1972.

[24] L. Kleinrock, *Queueing Systems, Vol. 2.* Wiley, 1976.

[25] A. S. Tanenbaum, *Computer Networks.* Englewood Cliffs: Prentice-Hall, 1981.

[26] D. Bertsekas and R. Gallager, *Data Networks.* Englewood Cliffs: Prentice-Hall, 1987.

[27] W. J. Dally and C. L. Seitz, "Deadlock-free message routing in multiprocessor interconnection networks," *IEEE Transactions on Computers*, vol. C-36, pp. 547 – 553, May 1987.

[28] S. Radhakrishnan, "VLSI implementation of a router for the backtrack-to-the-origin-and-retry routing scheme of the hypercycle base interconnection networks," Master's thesis, University of Victoria, 1991.

[29] S. Dutt and J. P. Hayes, "Subcube allocation in hypercube computers," *IEEE Transactions on Computers*, vol. C-40, pp. 341 – 352, March 1991.

[30] M.-S. Chen and K. G. Shin, "Processor allocation in an n-cube multiprocessor," *IEEE Transactions on Computers*, vol. C-36, pp. 1396 – 1407, December 1987.

[31] J. Kim, C. R. Das, and W. Lin, "A top-down processor allocation scheme for hypercube computers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, pp. 20 – 30, January 1991.

[32] N. J. Dimopoulos, S. Radhakrishnan, and D. Radvan, "Routing and processor allocation on a hypercycle-based multiprocessor," in *International Conference on Supercomputing*, (Cologne, Germany), pp. 105–114, 1991.

[33] A. Agarwal, "Limits on interconnection network performance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, pp. 398 – 412, October 1991.

[34] B. W. Aren and H. Lee, "Analysis of chordal ring network," *IEEE Transactions on Computers*, vol. C-30, pp. 291 – 295, April 1981.

[35] A. Al-Dhelaan and B. Bose, "A new strategy for processors allocation in an n-cube multiprocessor," in *International Phoenix Conference on Computers and Communication*, pp. 114 – 118, March 1989.

[36] F. P. Preparata and J. Vuillemin, "The cube-connected cycles: A versatile network for parallel computation," *Communications of the ACM*, vol. 24, May 1981.

[37] H. J. Siegel, "Analysis techniques for SIMD machine interconnection networks and the effects of processor address masks," *IEEE Transactions on Computers*, vol. C-26, pp. 153 – 161, February 1977.

Appendix A

The Simulator

In this appendix we give a brief description of the simulator we implemented to study the behavior of the various allocation strategies. The simulator was written in the C programming language and the source files follow this discussion.

The simulator consists of three logical layers, each one independent of the other. The top layer (layer number 3) is the user interface and consists of the files `main.c`, `screen.c` and `defs.h`. The file `defs.h` contains the necessary definitions and declarations for this layer. The file `screen.c` is responsible for reading the user requests and displaying the simulation results. The file `main.c` initializes the simulator, calls the routines in `screen.c` whenever the user has a request or the simulator has produced some results, and finally calls the simulator routines in layer 2. Parameters like the network interconnection, the task types and their interarrival patterns are set by the user through the routines of layer 3. A simulation session may consist of several separate simulations. The first option is to repeat the same simulation. This is necessary in cases where the simulation time is relatively short and the results may not be reliable. By repeating the simulation, the random number generators produce different outputs hence the simulated hypercycle experiences different requests and yields different results. In this case the simulator returns the average of the results. The second option is to repeat the same simulation for different task interarrival

times, in order to study how a strategy behaves when the load becomes heavier. This is accomplished by specifying a step by which the interarrival times should be decreased. In this case the simulator repeats the current simulation with increasingly smaller interarrival delays, until the interarrival delays become zero or the task waiting queues overflow due to excessive delays.

Layer number 2 is responsible for simulating every clock tick, based on the parameters passed by layer 3, and consists of the files `simulate.c` and `simulate.h`. The necessary definitions and declarations can be found in the file `simulate.h`. The file `simulate.c` contains the actual routines to perform the simulation. It uses the routines provided by layer 1 to allocate a request and to create a random new one if necessary. The task queue, the allocation bits and the statistical information are handled by this layer.

Finally, layer number 1 contains the primitive routines used by the upper layers. It consists of the files `allocate.c`, `permute.c` and `random.c`. The file `allocate.c` contains the allocation strategies. Every strategy is given the list of the allocation bits and a request, and decides which nodes should be allocated to the request. No knowledge of the task running times or the task waiting queue is available. The file `permute.c` contains routines to produce permutations of a set of integers and `random.c` contains random number generators for various distributions.

Layer 2 has access to the routines in layer 1 through file `layer1.h`. This is the only interface provided to layer 2. In the same manner layer 3 has access to layer 2 only through file `layer2.h`. By organizing the code into layers, we have the advantages of easy debugging, well-defined interfaces, and shorter compilation times. The latter means that one can modify the routines in any layer independently of the other layers and recompile only the altered files instead of the whole simulator. The `make` utility driven by the file `Makefile` takes care of updating the simulator according to the changes in the various files. It should be noted that all the above advantages are not at the expense of simulator

performance.

The user interface is given in Tables A.1 - A.3. The notation used for the arguments of each command have the following meaning: *int* means that an integer number should be given; *real* means that a real number should be given; <text> means that the word 'text' should be given and '|' represents 'or' (i.e. alternative arguments). Table A.1 contains the parameters that control a single simulation. Some additional settings to control a series of simulations and the format of the results are given in Table A.2. Finally, Table A.3 gives the commands available to the user.

Table A.1: Simulation parameters

COMMAND	COMMENT
`arrdelay real`	the interarrival delay between the incoming tasks
`arrmode <poisson | constant>`	tasks come in either constant or poisson distributed intervals
`durdis <uniform | gaussian | exponential>`	tasks may execute for uniform, gaussian or exponential distributed time intervals (duration distribution)
`durmax int`	maximum allowed task duration
`durmean real`	mean of task duration, used only if `durdis` is gaussian or exponential
`durmin int`	minimum allowed task duration
`durvar int`	variance of task duration, used only if `durdis` is gaussian
`graph int int ...`	the hypercycle factors
`queue int`	the size of the waiting queue for the incoming tasks. If the queue overflows, simulation stops
`reqdis <uniform | gaussian | exponential>`	requests may ask for uniform, gaussian or exponential distributed number of nodes (request distribution)
`reqmax int`	maximum number of nodes allowed to a request
`reqmean real`	mean of the number of nodes asked by requests, used only if `reqdis` is gaussian or exponential
`reqmin int`	minimum number of nodes allowed to a request
`reqmode <fragments | arbitrary>`	requests may either match the fragment sizes or ask for an arbitrary number of nodes
`reqvar real`	variance of the number of nodes asked by requests, used only if `reqdis` is gaussian
`simlength int`	the number of clock ticks to be simulated

Table A.1: (continued)

COMMAND	COMMENT
`strategy <optimal | suboptimal | graycodes>`	the possible allocation strategies. If the graph is a hypercube, the optimal and suboptimal strategies are identical to the buddy strategy
`trees int`	the number of permutations to be used concurrently, applies only to the optimal and suboptimal allocation strategies

Table A.2: Simulator settings

COMMAND	COMMENT
`arrcountdown <yes | no>`	if 'yes' then after the simulation ends, `arrdelay` will be decreased by `arrstep`, and the simulation is repeated. If `arrstep` is not specified, `arrdelay` will be decreased by 1. This simulation loop will stop when `arrdelay` becomes zero, or the task queue overflows
`arrstep real`	how much `arrdelay` should be decreased in the next simulation
`display <yes | all>`	the same as `save` but for displaying the results on screen. The final, averaged result is always displayed
`perm int1 = int int ...`	define the 'int1'-th permutation, 0 is the first permutation. The integers following the '=' should be a rearrangement of 1, 2, ..., R, where R is the number of factors in the hypercycle. If `trees` is > 1, and not enough `perms` are specified, random permutations will be used
`repeats int`	how many times the same simulation should be repeated
`save <yes | no | all>`	if 'yes' or 'all' the final, averaged results will be saved to a file named 'RESULTS'. If 'all', then if `repeats` is > 1, the results of every repetition of the simulation will also be saved

Table A.3: User commands

COMMAND	COMMENT
`echo "message"`	display a message
`exit`	quit the simulator
`go`	start a simulation
`help`	a file named 'allocate.help' is displayed to assist the user with the commands
`show`	show all the parameters and settings
`showperms`	show the permutations
`start`	the same as `go`
`quit`	the same as `exit`

```
/* ALLOC-SIM: A simulator for the allocation problem in HyperCycles.
 *
 * Driver file for the simulation - layer 3 (MAIN.C)
 *
 * V. Dimakopoulos  –  Victoria 1991
 */

#include  <stdio.h>
#include  <string.h>
#include  <math.h>
#include  "layer2.h"
#include  "defs.h"
#include  "screen.c"

USERDATA  data;

main()
{
  int   i, x, t, rep, inforate;
  char  str[200];

  /* Install the default values
   */
  FileSave = YES;
  Testing = NO;
  Display = YES;
  data.SimulationLength = 20000;
  data.QueueSize = 3000;
  RepeatNum = 1;
  data.ArrivalMode = POISSON;
  data.ArrivalDelay = 5.0;
  CountDown = NO;
  ArrivalStep = 0.2;
  data.N = 120;       /* total # nodes */
  data.R = 4;
  data.Factors[1] = 2;    /* factors */
  data.Factors[2] = 3;
  data.Factors[3] = 4;
  data.Factors[4] = 5;
  for (i = 1; i <= data.R; i++)
    data.Perms[0][i] = i;
  data.NumOfPerms = 1;
  data.Strategy = OPTIMAL;
  data.RequestMode = ARBITRARY;
  data.DurationDis = UNIFORM;
```

```
  data.DurationMin = 3;
  data.DurationMax = 7;
  data.DurationMean = 5.0;
  data.DurationVar = 1.0;
  data.RequestDis = UNIFORM;
  data.RequestMin = 1;
  data.RequestMax = data.N;
  data.RequestMean = ((double) (data.R)) / 2.0;
  data.RequestVar = ((double) (data.R)) / 8.0;

  puts("                    ********************************");
  puts("                    **                            **");
  puts("                    **  V.D. - HyperCycles Group  **");
  puts("                    **   University of Victoria   **");
  puts("                    **      Victoria, 1991        **");
  puts("                    **                            **");
  puts("                    ********************************");
  MAINLOOP:
    printf("ALLOC-SIM> ");
    if (fgets(str, 199, stdin) != NULL)
      switch (ParseLine(str))
      {
        case OK:  goto MAINLOOP;
        case GO:  DoSimulation();
                  goto MAINLOOP;
        case END: break;
      }
    else
      puts(" ");   /* CTRL-D pressed, or EOF of input file */
  ENDLOOP;
}



DoSimulation()
{
  int  i, rep, x, inforate;

  if (data.RequestDis == GAUSSIAN)       /* Prepare discrete probabilities */
    PrepareDiscreteGaussian(data.RequestMean, data.RequestVar,
                            data.RequestMin, data.RequestMax);
  rep = RepeatNum + 1;
  for (i = 0; i < data.NumOfPerms; i++)
    PermTimes[i] = 0L;
```

```
DOSIMLOOP:

  rep--;
  InitSimulator(data);

  /* Now enter the simulation loop
   */
  inforate = 100;  /* How often the clock is displayed on screen */
  if (Display == ALL)
    fprintf(stderr, "\n\nClock:      00");

  while (1)    /* Main Sim Loop: until FINISHED or QOVERFLOW */
  {
    if ((inforate--) == 0)
    {
      if (Display == ALL)
        fprintf(stderr, "\b\b\b\b\b\b%6ld", ClockTick);
      inforate = 99;
    }
    if (Testing == YES)
    {
      puts("Press a key for next clock tick (`q' quits) ..");
      x = getchar();
      getchar();
      if (((char) x) == 'q')
        return;
    }
    x = (Testing == YES) ? DebugNextTick() : SimulateNextTick();
    switch (x)
    {
      case QOVERFLOW: puts("\nOverflow occured.");
                      break;
      case FINISHED:        /* puts("\nSimulation finished."); */
                      break;
    }
    if (x != CONTINUE)
      break;
  }  /* Main Sim Loop */

  /* Statistics
   */
  if (Display == ALL)
    ShowResults(rep);
```

```
    if (FileSave == ALL)
      SaveResults(YES);
    AvDelays[rep] = AverageDelay;
    AvAllocations[rep] = AverageAllocation;
    AvUtilizations[rep] = AverageUtilization;
    for (i = 0; i < data.NumOfPerms; i++)
      PermTimes[i] += PermUsage[i];

    if (rep == 1)
    {
      /* Final Statistics
       */
      FinalResults();
      if ((CountDown == YES) && (data.ArrivalDelay >= 1.0 + ArrivalStep)
          && (x != QOVERFLOW))
      {
        data.ArrivalDelay -= ArrivalStep;
        rep = RepeatNum + 1;
        goto DOSIMLOOP;
      }
      return;
    }
    goto DOSIMLOOP;
  ENDLOOP;
} /* DoSimulation */


Statistics(values, num, mean, var)
double values[];
int    num;
double *mean, *var;
{
  double  av = 0.0, v = 0.0;
  int     i;

  for (i = 1; i <= num; i++)
  {
    av += values[i];
    v  += (values[i] * values[i]);
  }
  *mean = av / ((double) num);
  *var  = (v / ((double) num)) - (*mean)*(*mean);
}
```

```
FinalResults()
{
  double  m, v;

  if (CountDown == YES)
    printf("\n\nInterarrival delay = %lf\n", data.ArrivalDelay);
  printf("\n\nAVERAGE BEHAVIOUR:\n");
  printf("==================\n\n");
  Statistics(AvDelays, RepeatNum, &m, &v);
  printf("Average waiting delay in the queue:  %lf (variance = %lf)\n",
         m, v);
  Statistics(AvAllocations, RepeatNum, &m, &v);
  printf("Average allocation of nodes:         %lf\n", m);
  Statistics(AvUtilizations, RepeatNum, &m, &v);
  printf("Average utilization of nodes:        %lf\n", m);
  if (FileSave != NO)
    SaveResults(ALL);
}
```

```
/* File: SCREEN.C
 *
 * File for collecting the simulation data and printing the results.
 * (user interface - layer 3)
 *
 * V. Dimakopoulos  -  Victoria 1991
 */

#include <time.h>


int  SetCountDown(), SetArrivalDelay(), SetArrivalMode(),
     SetArrivalStep(), SetDisplay(), SetDurationDis(),
     SetDurationMax(), SetDurationMean(), SetDurationMin(),
     SetDurationVar(), Echo(), Quit(), Go(),
     SetFactors(), Help(), SetPermutation(),
     SetQueueSize(), Quit(), SetRepeatNum(),
     SetRequestDis(), SetRequestMax(), SetRequestMean(),
     SetRequestMin(), SetRequestMode(), SetRequestVar(),
     SetFileSave(), ShowData(), ShowPerms(),
     SetSimulationLength(), Go(), SetStrategy(),
     SetTesting(), SetNumOfPerms();

struct cf
  {
    char *command;
    int  (*function)();
  };

struct cf Commands[] =       /* User Commands */
  {
    { "arrcountdown", SetCountDown    },
    { "arrdelay",     SetArrivalDelay },
    { "arrmode",      SetArrivalMode  },
    { "arrstep",      SetArrivalStep  },
    { "display",      SetDisplay      },
    { "durdis",       SetDurationDis  },
    { "durmax",       SetDurationMax  },
    { "durmean",      SetDurationMean },
    { "durmin",       SetDurationMin  },
    { "durvar",       SetDurationVar  },
    { "echo",         Echo            },
    { "exit",         Quit            },
    { "go",           Go              },
    { "graph",        SetFactors      },
```

```
    { "help",         Help            },
    { "perm",         SetPermutation  },
    { "queue",        SetQueueSize    },
    { "quit",         Quit            },
    { "repeats",      SetRepeatNum    },
    { "reqdis",       SetRequestDis   },
    { "reqmax",       SetRequestMax   },
    { "reqmean",      SetRequestMean  },
    { "reqmin",       SetRequestMin   },
    { "reqmode",      SetRequestMode  },
    { "reqvar",       SetRequestVar   },
    { "save",         SetFileSave     },
    { "show",         ShowData        },
    { "showperms",    ShowPerms       },
    { "simlength",    SetSimulationLength },
    { "start",        Go              },
    { "strategy",     SetStrategy     },
    { "test",         SetTesting      },
    { "trees",        SetNumOfPerms   }
  };
int  NumOfCommands = 33;
int  argnum;
char *args[30];


/* Convert a string to (args, argc)
 */
GetTokens(str, argc, args)
char *str;
int  *argc;
char *args[];
{
  int comment;

  *argc = 0;
  while (1)
  {
    while ((*str == ' ') || (*str == '\t') || (*str == '\n'))
      str++;
    if (*str == 0)
      return;
    comment = (*str == '\"');
    args[(*argc)++] = (comment) ? str+1 : str;
    while (1)
```

```
    {
      str++;
      if (comment)
      {
        if (*str == '\"')
          break;
      }
      else
      {
        if ((*str == ' ') || (*str == '\t') || (*str ==  '\n'))
          break;
      }
      if (*str == 0)
        return;
    }
    *str = 0;
    str++;
  }
}


ParseLine(str)
char *str;
{
  int  i;

  GetTokens(str, &argnum, args);
  if (argnum)
  {
    for (i = 0; i < NumOfCommands; i++)
      if (strcmp(args[0], Commands[i].command) == 0)
        return ((*(Commands[i].function))());
    puts("Incorrect syntax or unknown entry. Check `help' ...");
  }
  return (OK);
}


SetCountDown()
{
  if ((argnum == 1) || (strcmp(args[1], "yes") == 0))
    CountDown = YES;
  else
    if (strcmp(args[1], "no") == 0)
```

```
      CountDown = NO;
    else
      puts("Usage: arrcountdown <yes | no>");
  return (OK);
}


SetArrivalDelay()
{
  double i;

  if ((argnum != 3) || strcmp(args[1], "="))
  {
    puts("Incorrect suntax. Check `help'.");
    return (OK);
  }
  i = atof(args[2]);
  if ((i <= 0.0) || (i >= 100.0))
    puts("Lambda must be between 0 and 100.");
  else
    data.ArrivalDelay = i;
  return (OK);
}


SetArrivalMode()
{
  if ((argnum != 3) || strcmp(args[1], "="))
  {
    puts("Incorrect suntax. Check `help'.");
    return (OK);
  }
  if (strcmp(args[2], "constant") == 0)
    data.ArrivalMode = CONSTANT;
  else
    if (strcmp(args[2], "poisson") == 0)
      data.ArrivalMode = POISSON;
    else
      puts("Unknown arrivals pattern: check `help' ...");
  return (OK);
}


SetArrivalStep()
```

```
{
  double i;

  if ((argnum != 3) || strcmp(args[1], "="))
  {
    puts("Incorrect suntax. Check `help'.");
    return (OK);
  }
  i = atof(args[2]);
  if ((i <= 0.0) || (i >= data.ArrivalDelay))
    printf("The countdown step should be between 0 and %f.\n",
           data.ArrivalDelay);
  else
    ArrivalStep = i;
  return (OK);
}


SetDisplay()
{
  if ((argnum == 1) || (strcmp(args[1], "yes") == 0))
    Display = YES;
  else
    if (strcmp(args[1], "all") == 0)
      Display = ALL;
    else
      puts("Usage: display <yes | all>");
  return (OK);
}


SetDurationDis()
{
  if ((argnum != 3) || strcmp(args[1], "="))
  {
    puts("Incorrect suntax. Check `help'.");
    return (OK);
  }
  if (strcmp(args[2], "uniform") == 0)
    data.DurationDis = UNIFORM;
  else
    if (strcmp(args[2], "gaussian") == 0)
      data.DurationDis = GAUSSIAN;
    else
```

```
      puts("Unknown task duration distibution: check `help' ...");
  return (OK);
}


SetDurationMax()
{
  int i;

  if ((argnum != 3) || strcmp(args[1], "="))
  {
    puts("Incorrect suntax. Check `help'.");
    return (OK);
  }
  i = atoi(args[2]);
  if ((i <= data.DurationMin ) || (i > MAX_DUR))
    printf("Maximum duration must be between %d and %d.\n",
           data.DurationMin, MAX_DUR);
  else
    data.DurationMax = i;
  return (OK);
}


SetDurationMean()
{
  double i;

  if ((argnum != 3) || strcmp(args[1], "="))
  {
    puts("Incorrect suntax. Check `help'.");
    return (OK);
  }
  i = atof(args[2]);
  if ((i <= 0.0) || (i > MAX_DUR.0))
    printf("Mean of duration must be between 1 and %d.\n", MAX_DUR);
  else
    data.DurationMean = i;
  return (OK);
}


SetDurationMin()
{
```

```
  int i;

  if ((argnum != 3) || strcmp(args[1], "="))
  {
    puts("Incorrect suntax. Check `help'.");
    return (OK);
  }
  i = atoi(args[2]);
  if (i <= 0 )
    printf("Minimum duration must be greater than 0 ticks.\n");
  else
  {
    if (i > data.DurationMax)
      printf("Warning: maximum duration (durmax) set to %d.\n",
             data.DurationMax = i);
    data.DurationMin = i;
  }
  return (OK);
}


SetDurationVar()
{
  double i;

  if ((argnum != 3) || strcmp(args[1], "="))
  {
    puts("Incorrect suntax. Check `help'.");
    return (OK);
  }
  i = atof(args[2]);
  if (i <= 0.0)
    puts("Variance of duration must be greater than 0");
  else
    data.DurationVar = i;
  return (OK);
}


Echo()
{
  if (argnum > 1)
    puts(args[1]);
  return (OK);
```

```
}


Go()
{
  return (GO);
}


SetFactors()
{
  int  c, i, f[MAX_FACTORS+1];

  if ((argnum <= 3) || strcmp(args[1], "="))
  {
    puts("Incorrect synatax or less than 2 factors. Check `help'");
    return (OK);
  }
  c = argnum - 2;
  if (c > MAX_FACTORS)
    c = MAX_FACTORS;
  for (i = 0; i < c; i++)
    if ((f[i] = atoi(args[i+2])) < 2)
    {
      printf("Error: the %d-th factor is less than 2.\n", i+1);
      return (OK);
    };
  data.R = c;
  data.N = 1;
  data.NumOfPerms = 1;
  for (i = 1; i <= c; i++)
  {
    data.Perms[0][i] = i;
    data.Factors[i] = f[i-1];
    data.N *= f[i-1];
    if (i > 1)
      if ((data.Strategy == GRAYCODE) &&
          ((data.Factors[i] != data.Factors[i-1]) || (data.Factors[i] != 2)))
        data.Strategy = OPTIMAL;
  }
  data.RequestMin = (data.RequestMode == ARBITRARY);
  data.RequestMax = (data.RequestMode == ARBITRARY) ?
                    data.N : data.R;
  data.RequestMean = (double) data.RequestMax / 2.0;
```

```
  data.RequestVar = data.RequestMean / 4.0;
  return (OK);
}


Help()
{
  FILE *fp;
  char str[200];

  if ((fp = fopen("allocate.help", "r")) == NULL)
    puts("File `allocate.help' could not be found.");
  else
  {
    while (fgets(str, 199, fp) != NULL)
      printf(str);
    fclose(fp);
  }
  return (OK);
}


SetPermutation()
{
  int i, j, index, used[MAX_FACTORS+1], seq[MAX_FACTORS+1];

  if ((argnum != (data.R+3)) || strcmp(args[2], "="))
  {
    puts("Syntax: perm index = permutation of numbers 1 2 ... R");
    return (OK);
  }
  index = atoi(args[1]);
  if ((index < 0) || (index >= data.NumOfPerms))
  {
    puts("Permutations index should be between 0 and (#trees - 1).");
    return (OK);
  }
  for (i = 1; i <= data.R; i++)
    used[i] = 0;
  for (i = 1; i <= data.R; i++)
  {
    j = atoi(args[i+2]);
    if ((j < 1) || (j > data.R))
    {
```

```
      printf("The %dth number (%d) is illegal.\n", i, j);
      return (OK);
    }
    if (used[j])
    {
      printf("The %dth number (%d) has been already used.\n", i, j);
      return (OK);
    }
    seq[i] = j;
    used[j] = 1;
  }
  for (i = 1; i <= data.R; i++)
    data.Perms[index][i] = seq[i];
  return (OK);
}


SetQueueSize()
{
  int i;

  if ((argnum != 3) || strcmp(args[1], "="))
  {
    puts("Incorrect suntax. Check `help'.");
    return (OK);
  }
  i = atoi(args[2]);
  if ((i <= 0) || (i > MAX_WAITING))
    printf("Queue size must be between 1 and %d\n", MAX_WAITING);
  else
    data.QueueSize = i;
  return (OK);
}


Quit()
{
  return (END);
}


SetRepeatNum()
{
  int i;
```

```
  if ((argnum != 3) || strcmp(args[1], "="))
  {
    puts("Incorrect suntax. Check `help'.");
    return (OK);
  }
  i = atoi(args[2]);
  if ((i >= MAX_REPEAT) || (i < 1))
    printf("This number should be between 1 and %d.\n",
           MAX_REPEAT-1);
  else
    RepeatNum = i;
  return (OK);
}


SetRequestDis()
{
  if ((argnum != 3) || strcmp(args[1], "="))
  {
    puts("Incorrect suntax. Check `help'.");
    return (OK);
  }
  if ((data.RequestMode == FRAGMENTS) &&
      (data.Strategy == SUBOPTIMAL))
    return;
  if (strcmp(args[2], "uniform") == 0)
    data.RequestDis = UNIFORM;
  else
    if (strcmp(args[2], "gaussian") == 0)
      data.RequestDis = GAUSSIAN;
    else
      if (strcmp(args[2], "exponential") == 0)
        data.RequestDis = EXPONENTIAL;
      else
        puts("Unknown request distribution: check `help' ...");
  return (OK);
}


SetRequestMax()
{
  int i, mi, ma;
```

```
  if ((argnum != 3) || strcmp(args[1], "="))
  {
    puts("Incorrect suntax. Check `help'.");
    return (OK);
  }
  i = atoi(args[2]);
  mi = data.RequestMin;
  ma = (data.RequestMode == FRAGMENTS) ? data.R : data.N;
  if ((i < mi ) || (i > ma))
    printf("Maximum request must be between %d and %d.\n", mi, ma);
  else
    data.RequestMax = i;
  return (OK);
}


SetRequestMean()
{
  int    mi, ma;
  double i;

  if ((argnum != 3) || strcmp(args[1], "="))
  {
    puts("Incorrect suntax. Check `help'.");
    return (OK);
  }
  i = atof(args[2]);
  mi = (data.RequestMode == ARBITRARY);
  ma = (data.RequestMode == FRAGMENTS) ? data.R : data.N;
  if ((i < (double) mi ) || (i > (double) ma))
    printf("Mean of request must be between %d and %d.\n", mi, ma);
  else
    data.RequestMean = i;
  return (OK);
}


SetRequestMin()
{
  int i, mi, ma;

  if ((argnum != 3) || strcmp(args[1], "="))
  {
    puts("Incorrect suntax. Check `help'.");
```

```
    return (OK);
  }
  i = atoi(args[2]);
  mi = (data.RequestMode == ARBITRARY);
  ma = data.RequestMax;
  if ((i < mi ) || (i > ma))
    printf("Minimum request must be between %d and %d.\n", mi, ma);
  else
    data.RequestMin = i;
  return (OK);
}


SetRequestMode()
{
  if ((argnum != 3) || strcmp(args[1], "="))
  {
    puts("Incorrect suntax. Check `help'.");
    return (OK);
  }
  if (strcmp(args[2], "fragments") == 0)
    data.RequestMode = FRAGMENTS;
  else
    if (strcmp(args[2], "arbitrary") == 0)
      data.RequestMode = ARBITRARY;
    else
    {
      puts("Unknown request mode: check `help' ...");
      return (OK);
    };
  if ((data.RequestMode == FRAGMENTS) &&
      (data.Strategy == SUBOPTIMAL))
    data.RequestDis = UNIFORM;
  data.RequestMin = (data.RequestMode == ARBITRARY);
  data.RequestMax = (data.RequestMode == FRAGMENTS) ?
                    data.R : data.N;
  data.RequestMean = (double) data.RequestMax / 2.0;
  data.RequestVar = data.RequestMean / 4.0;
  return (OK);
}


SetRequestVar()
{
```

```
  int    mi, ma;
  double i;

  if ((argnum != 3) || strcmp(args[1], "="))
  {
    puts("Incorrect suntax. Check `help'.");
    return (OK);
  }
  i = atof(args[2]);
  mi = 1;
  ma = (data.RequestMode == FRAGMENTS) ? data.R / 2 : data.N / 2;
  if (i <= 0.0)
    puts("Variance of request must be greater than 0");
  else
    data.RequestVar = i;
  return (OK);
}


SetFileSave()
{
  if ((argnum == 1) || (strcmp(args[1], "yes") == 0))
    FileSave = YES;
  else
    if (strcmp(args[1], "no") == 0)
      FileSave = NO;
    else
      if (strcmp(args[1], "all") == 0)
        FileSave = ALL;
      else
        puts("Usage: save <yes | no | all>");
  return (OK);
}


ShowData()
{
  int  i;

  printf("Factors, m1 first     (graph) = ");
  for (i = 1; i <= data.R; i++)
    printf("%d ", data.Factors[i]);
  printf("\n");
  printf("  (Nodes in the graph:  %d)\n", data.N);
```

```
  printf("Task queue size      (queue)  = %d tasks\n", data.QueueSize);
  printf("Allocation method  (strategy) = %s\n",
         (data.Strategy == OPTIMAL) ? "optimal" :
         ((data.Strategy == SUBOPTIMAL) ? "suboptimal" :
         "graycodes"));
  printf("# address trees used  (trees) = %d\n", data.NumOfPerms);
  printf("Simulation length (simlength) = %d ticks\n",
         data.SimulationLength);
  printf("Task arrival mode   (arrmode) = %s\n",
         (data.ArrivalMode == CONSTANT) ? "constant" : "poisson");
  printf("Interarrival time  (arrdelay) = %lf ticks\n",
         data.ArrivalDelay);
  printf("Decrement time (arrcountdown) = %s\n",
         (CountDown) ? "yes" : "no");
  printf("Countdown step      (arrstep) = %lf\n", ArrivalStep);
  printf("Task duration distr. (durdis) = %s\n",
         (data.DurationDis == UNIFORM) ? "uniform" : "gaussian");
  printf("Minimum duration     (durmin) = %d ticks\n", data.DurationMin);
  printf("Maximum duration     (durmax) = %d ticks\n", data.DurationMax);
  printf("Mean of durations   (durmean) = %lf \n", data.DurationMean);
  printf("Var. of durations    (durvar) = %lf \n", data.DurationVar);
  printf("Node request mode   (reqmode) = %s\n",
          (data.RequestMode == FRAGMENTS) ? "fragments(/segments)" :
          "arbitrary");
  printf("Request distribution (reqdis) = %s\n",
         (data.RequestDis == UNIFORM) ? "uniform" :
         ((data.RequestDis == GAUSSIAN) ? "gaussian" :
         "exponential"));
  printf("Minimum request      (reqmin) = %d\n", data.RequestMin);
  printf("Maximum request      (reqmax) = %d\n", data.RequestMax);
  printf("Mean of requests    (reqmean) = %lf\n", data.RequestMean);
  printf("Var. of requests     (reqvar) = %lf\n", data.RequestVar);
  printf("Repeat simul. for   (repeats) = %d times\n", RepeatNum);
  printf("Display results     (display) = %s\n", (Display == YES) ?
         "yes (only the final)" : "all (for every repeat)");
  printf("Save results           (save) = %s\n", (FileSave == NO) ? "no" :
         ((FileSave == YES) ? "yes (only the final)" :
                              "all (for every repeat)"));
  printf("Testing mode           (test) = %s\n", (Testing == YES) ? "yes" :
         "no");
  return (OK);
}
```

```
ShowPerms()
{
  int i, j;

  for (i = 0; i < data.NumOfPerms; i++)
  {
    printf("Permutation %3d: ", i);
    for (j = 1; j <= data.R; j++)
      printf("%d ", data.Factors[ data.Perms[i][j] ]);
    printf("\n");
  }
  return (OK);
}


SetSimulationLength()
{
  int i;

  if ((argnum != 3) || strcmp(args[1], "="))
  {
    puts("Incorrect suntax. Check `help'.");
    return (OK);
  }
  i = atoi(args[2]);
  if ((i <= 0) || (i > MAX_TIME))
    printf("Simulation should last between 1 and %d ticks\n",
           MAX_TIME);
  else
    data.SimulationLength = i;
  return (OK);
}


SetStrategy()
{
  int i;

  if ((argnum != 3) || strcmp(args[1], "="))
  {
    puts("Incorrect suntax. Check `help'.");
    return (OK);
  }
  if (strcmp(args[2], "optimal") == 0)
```

```
    data.Strategy = OPTIMAL;
  else
    if (strcmp(args[2], "suboptimal") == 0)
      data.Strategy = SUBOPTIMAL;
    else
      if (strcmp(args[2], "graycode") == 0)
      {
        for (i = 1; i < data.R; i++)
          if (data.Factors[i] != data.Factors[i+1])
          {
            puts("Gray-codes strategy does not apply to this graph.");
            return (OK);
          };
        data.Strategy = GRAYCODE;
        data.NumOfPerms = 1;
      }
      else
        puts("Unknown strategy: check `help' ...");
  return (OK);
}


SetTesting()
{
  if ((argnum == 1) || (strcmp(args[1], "yes") == 0))
    Testing = YES;
  else
    if (strcmp(args[1], "no") == 0)
      Testing = NO;
    else
      puts("Usage: test <yes | no>");
  return (OK);
}


SetNumOfPerms()
{
  int i, j, temp[MAX_FACTORS+1], SavePerm();

  if ((argnum != 3) || strcmp(args[1], "="))
  {
    puts("Incorrect suntax. Check `help'.");
    return (OK);
  }
```

```
  i = atoi(args[2]);
  if ((i < 1) || (i > factorial(data.R)))
  {
    printf("The number of trees should be between 1 and %d.\n",
           factorial(data.R));
    return;
  }
  if (i > data.NumOfPerms)
  {
    for (j = 1; j <= data.R; j++)
      temp[j] = j;
   Permute(temp, data.R, factorial(data.R), SavePerm);
  }
  data.NumOfPerms = i;
  return (OK);
}


/* Called from 'Permute()'
 */
int SavePerm(elems, r, p)
int elems[], r, p;
{
  int i;

  for (i = 1; i <= r; i++)
    data.Perms[p][i] = elems[i];
}


ShowResults(rep)
int rep;
{
  printf("SIMULATION RESULTS (%d remaining)\n", rep-1);
  puts("==================");
  printf("Clock at finishing time:   %d\n",
         ClockTick);
  printf("Created tasks:             %d\n",
         AllotedTasks + UnallotedTasks);
  printf("Allocated tasks:           %d\n",
         AllotedTasks);
  printf("Average queueing delay:    %.3lf%%\n",
         AverageDelay * 100.0);
  printf("Average node allocation:   %.3lf%%\n",
```

```
        AverageAllocation * 100.0);
  printf("Average node utilization:  %.3lf%%\n\n",
        AverageUtilization * 100.0);
}


SaveResults(w)
int w;
{
  int     i, m;
  time_t  t, time();
  FILE    *fp;
  long    all;
  double  m1, m2, m3, v1, v;

  fp = fopen("RESULTS.all", "a");
  t = time(0);
  fprintf(fp, "################################### %s",
              ctime(&t));

  /* Data */
  fprintf(fp, "Factors: ");
  for (i = 1; i <= data.R; i++)
    fprintf(fp, "%d ", data.Factors[i]);
  fprintf(fp, "\b, Nodes: %d, Trees: %d, Strategy: %s, Arrival: %s\n",
              data.N, data.NumOfPerms, (data.Strategy == OPTIMAL) ?
              "opt" : ((data.Strategy == SUBOPTIMAL) ? "sub" : "gc."),
              (data.ArrivalMode == CONSTANT) ? "const" : "poiss");
  fprintf(fp, "Requests: %s, %s, ",
              (data.RequestMode == FRAGMENTS) ?
              "fragments" : "arbitrary",
              (data.RequestDis == UNIFORM) ? "unif" :
              ((data.RequestDis == GAUSSIAN) ? "gauss" : "expon"));
  fprintf(fp, "%d(min), %d(max), %.2lf(mean), %.2lf(var)\n",
              data.RequestMin, data.RequestMax, data.RequestMean,
              data.RequestVar);
  fprintf(fp, "Durations: %s, ",
              (data.DurationDis == UNIFORM) ? "unif" : "gauss");
  fprintf(fp, "%d(min), %d(max), %.2lf(mean), %.2lf(var)\n",
              data.DurationMin, data.DurationMax, data.DurationMean,
              data.DurationVar);

  /* Results */
  if (w == ALL)
```

```
  {
    Statistics(AvDelays, RepeatNum, &m1, &v1);
    Statistics(AvAllocations, RepeatNum, &m2, &v);
    Statistics(AvUtilizations, RepeatNum, &m3, &v);
    fprintf(fp, "CLOCK %ld, AV.DELAY %.3lf, AV.ALLOC. %.3lf, ",
                ClockTick, m1, m2);
    fprintf(fp, "AV.UTIL. %.3lf (DELAY VAR. %.3lf)\n", m3, v1);
  }
  else
    fprintf(fp,
      "CLOCK %ld, AV.DELAY %.3lf, AV.ALLOC. %.3lf, AV.UTIL. %.3lf\n",
      ClockTick, AverageDelay, AverageAllocation, AverageUtilization);
  fprintf(fp, "PERM. USAGE (times used):");
  for (i = 0; i < data.NumOfPerms; i++)
    fprintf(fp, "(%d) %ld, ", i, PermTimes[i]);
  fprintf(fp, "\n");
  fclose(fp);
  SaveTabbed(w);
}


SaveTabbed(w)
int w;
{
  FILE    *fp;
  double  m1, m2, v;

  fp = fopen("RESULTS.tab", "a");
  if (w == ALL)
  {
    Statistics(AvDelays, RepeatNum, &m1, &v);
    Statistics(AvAllocations, RepeatNum, &m2, &v);
    fprintf(fp, "%.3lf\t%.3lf\t%.3lf\n", data.ArrivalDelay, m1, m2);
  }
  else
    fprintf(fp, "%.3lf\t%.3lf\t%.3lf\n", data.ArrivalDelay,
          AverageDelay, AverageUtilization);
  fclose(fp);
}


test_screen()
{
  /* For actul code look at the file
```

```
    * 'testing.c' which is no more used in this
    * simulator.
    */
}
```

```
/* File: DEFS.H
 *
 * Interface file for the Simulator (layer 3)
 *
 * V. Dimakopoulos  -  Victoria 1991
 */


/* Definitions
 */
#define  MAX_REPEAT  999999

#define  NO              0   /* Command Line Constants */
#define  YES             1
#define  ALL             2
#define  OK              0
#define  GO              1
#define  END             2
#define  ENDLOOP;

USERDATA data;

/* Global Variables
 */
int    FileSave;    /* for saving the results */
int    Display;     /* for displaying the results */
int    Testing;     /* for the testing screen */
int    RepeatNum;   /* how many times a simulation should be repeated */
double ArrivalStep; /* for the arrival countdowns */
int    CountDown;   /* if we are to decrement lambda by one until overflow */
double AvDelays[MAX_REPEAT];
double AvAllocations[MAX_REPEAT];
double AvUtilizations[MAX_REPEAT];
long   PermTimes[5040];
```

```
/* File: LAYER2.H
 *
 * Interface file for the 2nd layer routines
 *
 * V. Dimakopoulos  -  Victoria 1991
 */

/* Definitions
 */
#define  MAX_NODES       500        /* Limmits */
#define  MAX_FACTORS     7
#define  MAX_TIME        999999
#define  MAX_WAITING     9999
#define  MAX_DUR         99

#define  UNIFORM         0          /* Constants */
#define  GAUSSIAN        1
#define  EXPONENTIAL     2
#define  CONSTANT        0
#define  POISSON         1
#define  FINISHED        0
#define  QOVERFLOW       1
#define  CONTINUE        2
#define  OPTIMAL         0
#define  SUBOPTIMAL      1
#define  GRAYCODE        2
#define  FRAGMENTS       0
#define  ARBITRARY       1


struct userdata
  {
    int    N;                          /* # nodes */
    int    R;                          /* # factors */
    int    Factors[MAX_FACTORS+1];     /* the factors of N */
    int    NumOfPerms;                 /* # permutations used */
    int    Perms[5040][MAX_FACTORS+1]; /* the permutations */

    long   SimulationLength;           /* how long must the simulation be */
    int    QueueSize;                  /* the size of the task queue */
    int    Strategy;                   /* allocation policy */
    int    ArrivalMode;                /* constant or poisson arrivals */
    double ArrivalDelay;               /* interrar. delay (mean if poisson) */

    int    DurationDis;                /* uniform or gaussian distribution */
```

```
    int    DurationMin;                /* minimum task duration */
    int    DurationMax;                /* maximum task duration */
    double DurationMean;               /* mean; for gaussian */
    double DurationVar;                /* variance; for gaussian */

    int    RequestMode;                /* fragments or segments */
    int    RequestDis;                 /* uniform, gauss., or exponetial */
    int    RequestMin;                 /* minimum request size */
    int    RequestMax;                 /* maximum request size */
    double RequestMean;                /* mean; for gaussian, exponential */
    double RequestVar;                 /* variance; for gaussian */
  };
typedef  struct userdata  USERDATA;


/* Variables
 */
extern long   ClockTick;              /* current clock value */
extern int    AllocList[MAX_NODES];   /* the allocation bits */
extern int    CurrentFree;            /* # currently UNALLOCATED nodes */
extern int    CurrentWasted;          /* # currently WASTED nodes */
extern long   SumFree;                /* sum of CurrentFree up to now */
extern long   SumWasted;              /* sum of CurrentWasted up to now */
extern long   AllotedTasks;           /* upto now allocated tasks */
extern long   UnallotedTasks;         /* finally how many are in the queue */
extern long   TotalDelay;             /* for the already allocated tasks */
extern double AverageDelay;           /* FINAL statistics */
extern double AverageAllocation;      /*       —          */
extern double AverageUtilization;     /*       —          */
extern long   PermUsage[5040];        /* # times each perm. was used */

/* Functions
 */
extern int InitSimulator();         /* initializes & checks validity */
extern int SimulateNextTick();      /* the simulation loop */
extern int DebugNextTick();         /* the simulation loop plus debugging */


/* Some layer 1 functions that may be accessed
 */
extern int     poisson(/* double lambda */);
extern double  gaussian(/* double mean, var */);
extern double  exponential(/* double mean */);
extern double  uniform(/* double a, b */);
```

```
extern          PrepareDiscreteGaussian();
extern          DiscreteProbability(/* int min, max */);
extern          GetProbabilities(/* double Probs[], num */);
extern          SetProbabilities(/* double Probs[], num */);

extern int      Permute(/* int elems[], n, nu_perms_wanted, save_perm() */);
extern int      RandomPermute(/* same as Permute() */);
extern int      factorial(/* int n */);
extern int      combinations(/* int n, k */);
```

```
/* File: SIMULATE.C
 *
 * Layer 2: Actions to be taken in every clock tick of the simulation
 *
 * V. Dimakopoulos  -  Victoria 1991
 */

#include "simulate.h"

static TASK debugtask;   /* a copy of the most recently created task */


int InitSimulator(data)
USERDATA data;
{
  register int i, j;
  int          *p[5040], *w[5040], *f[5040];

  N = data.N;
  R = data.R;
  for (i = 1; i <= R; i++)
    Factors[i] = data.Factors[i];
  NumOfPerms       = data.NumOfPerms;
  SimulationLength = data.SimulationLength;
  QueueSize        = data.QueueSize;
  Strategy         = data.Strategy;
  ArrivalMode      = data.ArrivalMode;
  ArrivalDelay     = data.ArrivalDelay;
  DurationDis      = data.DurationDis;
  DurationMin      = data.DurationMin;
  DurationMax      = data.DurationMax;
  DurationMean     = data.DurationMean;
  DurationVar      = data.DurationVar;
  RequestMode      = data.RequestMode;
  RequestDis       = data.RequestDis;
  RequestMin       = data.RequestMin;
  RequestMax       = data.RequestMax;
  RequestMean      = data.RequestMean;
  RequestVar       = data.RequestVar;

  /* Weights and fragments of the permutations
   */
  for (i = 0; i < NumOfPerms; i++)
  {
    PermUsage[i]    = 0L;
```

```
    Perms[i][1]     = data.Perms[i][1];
    Weights[i][1]   = 1;
    Fragments[i][0] = 1;
    Fragments[i][1] = Factors[ Perms[i][1] ];
    p[i] = &(Perms[i][0]);
    w[i] = &(Weights[i][0]);
    f[i] = &(Fragments[i][0]);
    for (j = 1; j < R; j++)
    {
      Perms[i][j+1]     = data.Perms[i][j+1];
      Weights[i][j+1]   = Weights[i][j] * Factors[ Perms[i][j] ];
      Fragments[i][j+1] = Fragments[i][j] * Factors[ Perms[i][j+1] ];
    }
  }

  /* Various initializations
   */
  for (i = 0; i < N; i++)
    AllocList[i] = NodeInfo[i] = 0;
  TaskQueue.head     = 0;
  TaskQueue.contains = 0;
  TaskQueue.size     = QueueSize;
  TotalDelay         = 0L;
  AllotedTasks       = 0L;
  CurrentFree        = N;
  CurrentWasted      = N;
  SumFree            = 0L;
  SumWasted          = 0L;
  ClockTick          = 0L;
  NextInterval       = 1;   /* because we first decrement it */

  /* For the allocation functions
   */
  SetAllocationStuff(N, R, Factors, NumOfPerms, p, w, f, AllocList);
}


SimulateNextTick()
{
  TASK  ttask;
  int   free_space;

  /* Check if simulation finished */
  if (ClockTick >= SimulationLength)
```

```
{
  int i;

  UnallotedTasks     = TaskQueue.contains;
  AverageDelay       = ((double) (TotalDelay)) /
                       ((double) (AllotedTasks));
  AverageAllocation  = 1.0 - ((double) (SumFree)) /
                             ((double) (N * (ClockTick)));
  AverageUtilization = 1.0 - ((double) (SumWasted)) /
                             ((double) (N * (ClockTick)));
  return (FINISHED);
}

/* Check for any released nodes */
free_space = CheckFree();

/* check for creation of new task */
if ((--(NextInterval)) == 0)
{
  CreateTask(&ttask);
  debugtask.requested = ttask.requested;
  debugtask.duration  = ttask.duration;
  if (push(&(TaskQueue), &ttask) == 0)
    return (QOVERFLOW);

  /* calculate the next interarrival interval */
  if (ArrivalMode == CONSTANT)
    NextInterval = (int) (ArrivalDelay + 0.5);
  else
  {
    if ((NextInterval = poisson(ArrivalDelay)) <= 0)
      NextInterval = 1;
  }
}

if (free_space)
  Accomodate();

/* update the statistics */
SumFree   += CurrentFree;
SumWasted += CurrentWasted;

/* advance clock */
ClockTick++;
```

```
  return (CONTINUE);
}


/* See if anybody finished or there are free nodes
 */
CheckFree()
{
  register int  i, space_created;

  /* Due to the way we allocate bits, we do not have to explicitly
   * keep track of the task owning each node; relinquishment
   * hapens after the appropriate time, by decrementing the quantity
   * that the allocation 'bit' contains (see Accomodate()).
   */
  space_created = 0;
  for (i = 0; i < N; i++)
    if (AllocList[i])
    {
      if ((-- (AllocList[i])) == 0)  /* someone finished */
      {
        space_created = 1;
        /* if a new task is allocated, current_wasted will decrease */
        CurrentFree += 1;
        CurrentWasted += 1 - NodeInfo[i];
        NodeInfo[i] = 0;
      }
    }
    else
      space_created = 1;

  return (space_created);
}


/* Allocates as many tasks as possible, starting from the head of
 * the queue.
 */
Accomodate()
{
  TASK  *ctask;

  if (TaskQueue.contains)
    /* FIFO : check the head of the queue */
```

```
    while (TryOne(&(TaskQueue.task[ TaskQueue.head ])))
    {
      /* Update queue */
      pop(&(TaskQueue));
      if (TaskQueue.contains == 0)
        break;
    };
}


/* Try to allocate only 1 task
 */
TryOne(ctask)
TASK  *ctask;
{
  int  i;

  ctask->alloted = (Strategy == OPTIMAL) ?
                   Optimal(ctask->requested, &(ctask->address))
                           :
                   ((Strategy == SUBOPTIMAL) ?
                   Suboptimal(ctask->requested, &(ctask->address))
                           :
                   GrayCode(ctask->requested, &(ctask->address)));
  i = (int) ctask->alloted / (N+1);   /* The permutation used */
  ctask->alloted -= (i * (N+1));      /* The actually allocated nodes */
  if (ctask->alloted)
  {
    (PermUsage[i])++;                 /* Kept for statistics */
    /* Store to each allocated bit the duration of the task,
     * so when it becomes zero later on, the nodes are
     * automatically released.
     */
    for (i = 0; i < N; i++)
      if (AllocList[i] == -1)  /* find which bits were */
      {                              /* just allocated */
        AllocList[i] = ctask->duration;
        NodeInfo[i]  = 0;
      }
    /* Keep the number of wasted nodes in the first node of the region */
    NodeInfo[ctask->address] = ctask->alloted - ctask->requested;
    ctask->starting_time = ClockTick;
    /* Kept for statistics */
    CurrentFree -= ctask->alloted;
    CurrentWasted -= ctask->requested;
```

```
    TotalDelay += (ctask->starting_time - ctask->creation_time);
    AllotedTasks ++;
    return (1);
  }
  return (0);
}


/* assumption: q is not empty
 */
pop(q)
QUEUE  *q;
{
  q->head = (q->head + 1) % (q->size);   /* circular */
  q->contains --;
}


/* Returns 0 if buffer is full
 */
push(q, t)
QUEUE  *q;
TASK   *t;
{
  TASK  *tail;

  if (q->contains == q->size)
    return (0);
  tail = &(q->task[(q->head + q->contains) % q->size]);  /* circular */
  tail->creation_time = t->creation_time;
  tail->duration = t->duration;
  tail->requested = t->requested;
  q->contains ++;
  return (1);
}


CreateTask(ttask)
TASK  *ttask;
{
  int  d, k;

  ttask->creation_time = ClockTick;
  /* in the uniform distribution, we use the interval
   * [DurationMin - 0.5, DurationMax + 0.5] instead of [1, max_duration]
```

```
 * in order to make up for the loss of uniformity at
 * the ends of the latter */
switch (DurationDis)
{
  case UNIFORM:
    d = (int) uniform((double) DurationMin, (double) (DurationMax + 1));
    break;

  default:
    d = (int) (gaussian(DurationMean, DurationVar) + 0.5);
}
if (d < DurationMin)
  d = DurationMin;
if (d > DurationMax)
  d = DurationMax;
ttask->duration = d;

switch (RequestDis)
{
  case UNIFORM:
    d = (int) uniform((double) RequestMin, (double) (RequestMax + 1));
    break;

  case GAUSSIAN:
    d = DiscreteProbability(RequestMin, RequestMax);
    break;

  default:
    d = (int) (exponential((double) RequestMean) + 0.5);
}
if (d < RequestMin)
  d = RequestMin;
if (d > RequestMax)
  d = RequestMax;

if (RequestMode == FRAGMENTS)
{
  if ((Strategy == SUBOPTIMAL) && (d < R))
  {      /* uniform distr. is assumed for segments */
    ttask->requested = (int) uniform(1.0, (double) Factors[d+1]);
    if (ttask->requested < 1)
      ttask->requested = 1;
    if (ttask->requested > Factors[d+1] - 1)
      ttask->requested = Factors[d+1] - 1;
```

```
    }
    else
      ttask->requested = 1;
    if (d)
      for (; d > 0; d--)
        ttask->requested *= Factors[d];
  }
  else
    ttask->requested = d;
}


/* For debugging the simulator.
 * Directly and with no changes at all, is used instead of
 * SimulateNextTick().
 */
DebugNextTick()
{
  int    i, rr;
  static int prevnext = 0;

  rr = SimulateNextTick();
  printf("Clock Tick.....%ld (just finished) \n", ClockTick-1);
  printf("Nodes..........%d free, %d allocated, %d wasted\n",
         CurrentFree, N - CurrentFree, CurrentWasted);
  printf("Tasks..........%d served, %d waiting\n", AllotedTasks,
         TaskQueue.contains);
  printf("Statistics.....delay...%lf, utilization...%lf\n",
         ((double) TotalDelay) / ((double) AllotedTasks),
         1.0 - ((double) SumWasted) / ((double) (N * (ClockTick))));
  if (NextInterval < prevnext)
    printf("New task in....%d ticks\n", NextInterval);
  else
    printf("New task.......requested...%d, runtime...%d\n",
           debugtask.requested, debugtask.duration);
  prevnext = NextInterval;
  printf("Allocation bits:\n");
  for (i = 0; i < N; i++)
    printf("%d", (AllocList[i] != 0));
  printf("\n");
  return (rr);
}
```

```
/* File: SIMULATE.H
 *
 * Definitions and declarations for 2nd layer file(s).
 * (not available to layer 3)
 *
 * V. Dimakopoulos  -  Victoria 1991
 */

#include "layer1.h"   /* interface to layer 1 */


/* Definitions
 */
#define  MAX_NODES       500        /* Limmits */
#define  MAX_FACTORS     7
#define  MAX_TIME        999999
#define  MAX_WAITING     9999
#define  MAX_DUR         99

#define  UNIFORM         0          /* Constants */
#define  GAUSSIAN        1
#define  EXPONENTIAL     2
#define  CONSTANT        0
#define  POISSON         1
#define  FINISHED        0
#define  QOVERFLOW       1
#define  CONTINUE        2
#define  OPTIMAL         0
#define  SUBOPTIMAL      1
#define  GRAYCODE        2
#define  FRAGMENTS       0
#define  ARBITRARY       1


/* Data Structures
 */
struct tsk                      /* task model */
  {
    long creation_time;         /* obvious */
    long starting_time;         /* obvious */
    int  duration;              /* obvioud */
    int  requested;             /* actual number of nodes needed */
    int  alloted;               /* number of nodes allocated */
    int  address;               /* start address of the allocated fragment */
  };
```

```
typedef  struct tsk  TASK;


struct tq                       /* the task queue model */
  {
    int   head;                 /* obvious */
    int   contains;             /* current number of enqueued tasks */
    int   size;                 /* obvious */
    TASK  task[MAX_WAITING];    /* obvious */
  };
typedef  struct tq  QUEUE;


struct userdata
  {
    int    N;                           /* # nodes */
    int    R;                           /* # factors */
    int    Factors[MAX_FACTORS+1];      /* the factors of N */
    int    NumOfPerms;                  /* # permutations used */
    int    Perms[5040][MAX_FACTORS+1]; /* the permutations */

    long   SimulationLength;            /* how long must the simulation be */
    int    QueueSize;                   /* the size of the task queue */
    int    Strategy;                    /* allocation policy */
    int    ArrivalMode;                 /* constant or poisson arrivals */
    double ArrivalDelay;                /* interrarival delay (mean if poisson) */

    int    DurationDis;                 /* uniform or gaussian distribution */
    int    DurationMin;                 /* minimum task duration */
    int    DurationMax;                 /* maximum task duration */
    double DurationMean;                /* mean; for gaussian */
    double DurationVar;                 /* variance; for gaussian */

    int    RequestMode;                 /* fragments or segments */
    int    RequestDis;                  /* uniform, gauss., or exponetial */
    int    RequestMin;                  /* minimum request size */
    int    RequestMax;                  /* maximum request size */
    double RequestMean;                 /* mean; for gaussian, exponential */
    double RequestVar;                  /* variance; for gaussian */
  };
typedef  struct userdata  USERDATA;


/* Layer 2 variables, transparent to upper Layers
```

```
 */
static int    N;
static int    R;
static int    Factors[MAX_FACTORS+1];
static int    NumOfPerms;
static int    Perms[5040][MAX_FACTORS+1];
static long   SimulationLength;
static int    QueueSize;
static int    Strategy;
static int    ArrivalMode;
static double ArrivalDelay;
static int    DurationDis;
static int    DurationMin;
static int    DurationMax;
static double DurationMean;
static double DurationVar;
static int    RequestMode;
static int    RequestDis;
static int    RequestMin;
static int    RequestMax;
static double RequestMean;
static double RequestVar;
static int    NextInterval;           /* when the next arrival will be */
static int    Weights[5040][MAX_FACTORS+1];    /* the weights */
static int    Fragments[5040][MAX_FACTORS+1];  /* the fragment sizes */
static QUEUE  TaskQueue;              /* tasks waiting for allocation */
static int    NodeInfo[MAX_NODES];    /* needed to keep how many wasted
                                       * nodes per allocated region exist.
                                       * Only the first node of the region
                                       * keeps this number, the others
                                       * contain 0 */


/* Layer 2 variables, accessible to upper Layers
 */
long   ClockTick;             /* current clock value */
int    AllocList[MAX_NODES];  /* the allocation bits */
int    CurrentFree;           /* # currently UNALLOCATED nodes */
int    CurrentWasted;         /* # currently WASTED nodes */
long   SumFree;               /* sum of CurrentFree up to now */
long   SumWasted;             /* sum of CurrentWasted up to now */
long   AllotedTasks;          /* upto now allocated tasks */
long   UnallotedTasks;        /* FINALY, how many are still in the queue */
long   TotalDelay;            /* for the already allocated tasks */
double AverageDelay;          /* FINAL statistics */
```

```
double AverageAllocation;       /*       —          */
double AverageUtilization;      /*       —          */
long   PermUsage[5040];         /* how many times each perm. was used */
```

```
/* File: LAYER1.H
 *
 * Interface to the Layer 1 routines
 *
 * V. Dimakopoulos  -  Victoria 1991
 */

/* From ALLOCATE.C
 */
extern        SetAllocationStuff();
extern int    Optimal(),
              Suboptimal(),
              GrayCode(),
              Release();

/* From RANDOM.C
 */
extern int    poisson();
extern double gaussian(),
              exponential(),
              uniform();
extern        PrepareDiscreteGaussian(),
              DiscreteProbability(),
              GetProbabilities(),
              SetProbabilities();

/* From PERMUTE.C
 */
extern int    Permute();
extern int    RandomPermute();
extern int    factorial();
extern int    combinations();
```

```
/* File: ALLOCATE.C
 *
 * Part of layer1: Optimal, suboptimal and gray-codes allocation procedures
 *
 * V. Dimakopoulos  -  Victoria 1991
 */

/*  Locally global variables for speed & ease in handling
 */
static int N,
           R,
           NumOfPerms,
           *Factors,
           *Perms[10000],
           *Weights[10000],
           *Fragments[10000],
           *AllocBits;


SetAllocationStuff(n, r, factors, permnum, perms, weights, frags, alloc_list)
int n, r, factors[], permnum, *perms[], *weights[], *frags[], alloc_list[];
{
  int  i;

  N = n;
  R = r;
  Factors = factors;
  NumOfPerms = permnum;
  for (i = 0; i < NumOfPerms; i++)
  {
    Perms[i] = perms[i];
    Weights[i]= weights[i];
    Fragments[i] = frags[i];
  }
  AllocBits = alloc_list;
}


/*  Optimal allocation strategy.
 *  Multiple address trees can be used if needed.
 */
int  Optimal(request, address)
int  request, *address;
{
  int  p, i, j, list[1000], size;
```

```
  if ((request < 1) || (request > N))
    return (0);

  if (request > 1)
  {
    /* First build list[] based on the redudancy of the coresponding
     * fragments */
    for (p = 0; p < NumOfPerms; p++)
    {
      i = 1;
      while (Fragments[p][i] < request)
        i++;
      list[p] = Fragments[p][i];  /* the size of the fragment */
    }

    /* Now try the trees with increasing order of redudancy */
    i = NumOfPerms;
    while (i--)
    {
      /* find the less redudant tree */
      size = N+1;  /* so that a smaller can be found */
      for (j = 0; j < NumOfPerms; j++)
        if (list[j] < size)
        {
          size = list[j];
          p = j;
        };

      /* Try to allocate it with this tree */
      if (optimal_one(size, p, address))
        return (size + (N+1)*p);   /* we need not know the permutation also */
      list[p] = N+2;  /* so that we do not choose it again */
    }
    return (0);  /* No tree was able to allocate it */
  }
  else     /* request = 1 */
    return (optimal_one(1, 0, address));
}


/*  Optimal allocation in one tree (i.e. sequence of factors)
 */
```

```
int  optimal_one(size, perm_num, address)
int  size, perm_num, *address;
{
  int  i, j, free;

  for (i = 0; i < N; i += size)
  {
    free = 1;
    /* check this region */
    for (j = 0; j < size; j++)
      /* Here we must use the maping */
      if (AllocBits[ map(i+j, perm_num) ])   /* region not available */
      {
        free = 0;
        break;
      };
    if (free)
      break;
  }

  /* Allocate this region by puting -1 to the region bits, not 1 */
  if (free)
  {
    for (j = 0; j < size; j++)
      AllocBits[ map(i+j, perm_num) ] = -1;
    *address = map(i, perm_num);   /* i.e. the returned address is absolute,
                                      not specific to the permutation */
    return (1);
  }
  return (0);
}


/*  Suboptimal allocation strategy.
 *  Multiple address trees can be used if needed.
 */
int  Suboptimal(request,address)
int  request, *address;
{
  int  p, i, j, k, size, child, list[1000],
       childs[1000], fragms[1000];

  if ((request < 1) || (request > N))
    return (0);
```

```
if (request > 1)
{
  /* First build list[] based on the redudancy of the coresponding
   * segments */
  for (p = 0; p < NumOfPerms; p++)
  {
    i = 1;
    /* Find the fragment size */
    while (Fragments[p][i] < request)
      i++;
    child = (i > 1) ? Fragments[p][i - 1] : 1;

    /* Find the 'size' of the request ('segment') */
    k = request / child;
    if ((child*k) < request)
      k++;

    list[p] = k * child;  /* the size of the segment */
    childs[p] = child;
    fragms[p] = Fragments[p][i];
  }

  /* Now try the trees with increasing order of redudancy */
  i = NumOfPerms;
  while(i--)
  {
    /* find the less redudant tree */
    size = N+1;  /* so that a smaller can be found */
    for (j = 0; j < NumOfPerms; j++)
      if (list[j] < size)
      {
        size = list[j];
        p = j;
      };

    /* Try to allocate it with this tree */
    j = (size == fragms[p]) ?
          optimal_one(size, p, address) :          /* for speed */
          suboptimal_one(fragms[p], childs[p], size, p, address);
    if (j)
      return (size + (N+1)*p);   /* we need not know the permutation also */
    list[p] = N+2;  /* so that we do not choose it again */
  }
```

```
    return (0);
  }
  else    /* request = 1 */
    return (optimal_one(1, 0, address));   /* to be faster */
}


/*  Suboptimal allocation in one tree (i.e. sequence of factors)
 */
int  suboptimal_one(fragment, child, size, perm_num, address)
int  fragment, child, size, perm_num, *address;
{
  int  i, j, l, s_addr, free;

  /* Search in every fragment for a free segment */
  for (i = 0; i < N; i += fragment)
  {
    /* check this fragment, for a free region of 'size' nodes */
    /* advancing in steps of 'child' nodes                    */
    for (j = 0; j <= fragment - size; j += child)
    {
      free = 1;
      s_addr = i + j;
      /* check this segment inside the fragment */
      for (l = 0; l < size; l++)
        if (AllocBits[map(s_addr+l, perm_num)]) /* region not available */
        {
           free = 0;
          break;
        };
      if (free)
        break;
    }
    if (free)
      break;
  }

  /* Allocate the region */
  if (free)
  {
    for (l = 0; l < size; l++)
      AllocBits[ map(s_addr + l, perm_num) ] = -1;
    *address = map(s_addr, perm_num);
    return (1);
```

```
  }
  return (0);
}


/* Both for optimal and sub-optimal allocation
 */
void release(address, size)
int  address,
     size;
{
  int i;

  for (i = 0; i < size; i++)
    AllocBits[address + i] = 0;
}


/*  Mapping of a number in a permuted factors numbering system to
 *  a number in the unpermuted one.
 *  Basically, we permute the corresponding tuple exactly the way
 *  the factors are permuted and then calculate the value of the
 *  new tuple.
 */
map(number, perm_num)
register int  number, perm_num;
{
  register int  i, tuple1[10], tuple2[10];

  /* find tuple-representation of 'num' */
  maketuple(number, tuple1, perm_num);
  /* inverse mapping */
  for (i = 1; i <= R; i++)
    tuple2[ Perms[perm_num][i] ] = tuple1[i];
  /* return the new value, using the -1-th permutation (i.e. no permut.) */
  return (makenum(tuple2, -1));
}


/*  Convert the tuple-representation to a number
 */
makenum(tuple, perm_num)
register int  tuple[], perm_num;
{
  register int  num, i, w;
```

```
  num = tuple[1];    /* first weight is always 1 */
  if (perm_num != -1)
  {
    for (i = 2; i <= R; i++)
      num += tuple[i] * Weights[perm_num][i];
  }
  else   /* use no permutation */
  {
    w = 1;
    for (i = 2; i <= R; i++)
    {
      w *= Factors[i-1];
      num += tuple[i] * w;
    }
  }
  return (num);
}


/*  Convert a number to tuple-representation
 */
maketuple(num, tuple, perm_num)
register int  num, tuple[], perm_num;
{
  register int  i;

  for (i = R; i >= 2; i--)
  {
    tuple[i] = num / Weights[perm_num][i];
    num -= tuple[i] * Weights[perm_num][i];
  }
  tuple[1] = num;
}


/*  Gray-codes allocation strategy - only 1 tree used.
 */
int  GrayCode(request, address)
int  request, *address;
{
  int  i, size;

  if ((request < 1) || (request > N))
```

```
    return (0);

  i = 1;
  if (request > 1)
  {
    while (Fragments[0][i] < request)
      i++;
    size = Fragments[0][i];
  }
  else
    size = 1;

  if (gray_one(size, address))
    return (size);

  return (0);  /* No tree was able to allocate it */
}


/*  Gray-codes allocation in one tree (i.e. sequence of factors)
 *  Although we use Gray codes, WE DO NOT NEED THE GRAY CODE!
 *  EXPLANATION:
 *  We examine the allocation bits i, where i in [m*size/2, (m+2)*size/2].
 *  Suppose we find this region free. Then we set these bits to 1.
 *  We then allocate the *NODES* that correspond to the above bits, which
 *  of course have addresses gcaddress(i). Since this is a SIMULATOR that
 *  only keeps track of the allocation bits, gcaddress(i) is useless.
 */
int  gray_one(size, address)
int  size, *address;
{
  int  i, j, free, step;

  if (size == 1)
  {
    /* special case */
    for (i = 0; i < N; i++)
      if (AllocBits[i] == 0)
      {
        AllocBits[i] = -1;
        *address = i;
        return (1);
      };
    return (0);
  }
```

```
  step = size >> 1;   /* = size / 2 = 2 ^ (k-1) */
  for (i = 0; i < N; i += step)
  {
    free = 1;
    /* check this region */
    for (j = 0; j < size; j++)
      if (AllocBits[(i+j) % N])   /* region not available */
      {
        free = 0;
        break;
      };
    if (free)
      break;
  }

  /* Allocate this region by puting -1 to the region bits, not 1 */
  if (free)
  {
    for (j = 0; j < size; j++)
      AllocBits[(i+j) % N] = -1;
    *address = i;
    return (1);
  }
  return (0);
}


/* Create a g.c. - not needed in this simulator
 */

int gcmapper[1024];

makegcmap(bits)
int bits;
{
  if (bits == 1)
  {
    gcmapper[0] = 0;
    gcmapper[1] = 1;
    return;
  }
  makegcmap(bits-1);
  {
    int i, c = 0, add, n;
```

```
    n = 1 << bits;
    add = 1 << (bits-1);
    for (i = (n-1); i > 0; i-=2)
      if (c)
      {
        gcmapper[i-1] = gcmapper[i / 2];
        gcmapper[i] = gcmapper[i-1] + add;
        c = 0;
      }
      else
      {
        gcmapper[i] = gcmapper[i / 2];
        gcmapper[i-1] = gcmapper[i] + add;
        c = 1;
      };
  }
}



/* Vertical instead of horizontal search.
 * Only for hypercubes. After testing, it
 * seems less efficient. Currently left
 * out of the simulator.
 */
int  AltBuddy(request, address)
int  request, *address;
{
  int size, i, j, p, free;

  if ((request < 1) || (request > N))
    return (0);

  if (request > 1)
  {
    i = 1;
    while (Fragments[0][i] < request)
      i++;
    size = Fragments[0][i];
  }
  else
    size = 1;

  /* Different order of search than optimal_one()
```

```
   */
  for (i = 0; i < N; i += size)
    for (p = 0; p < NumOfPerms; p++)
    {
      free = 1;
      for (j = 0; j < size; j++)
        if (AllocBits[ map(i+j, p) ])
        {
          free = 0;
            break;
        };
      if (free)
      {
        /* Allocate the region */
        for (j = 0; j < size; j++)
          AllocBits[ map(i+j, p) ] = -1;
        *address = map(i, p);
        return (size + (N+1)*p);
      }
    };
  return (0);
}
```

```
/* File: PERMUTE.C
 *
 * Part of layer 1: Permutations, factorials, combinations
 *
 * The permutation algorithms were adapted from:
 *    Robert Sedgewick, "Permutation Generation Methods",
 *    Computing Surveys, Vol. 9, No 2, June 1977
 *
 * V. Dimakopoulos  -  Victoria 1991
 */

#define  MAX_PERM  10  /* maximum number of elements */


/* JOHNSON - TROTTER ALGORITHM
 */
int Permute(elements, n, nu_of_perms, UsePerm)
int elements[];
int n;
int nu_of_perms;
int (*UsePerm)();  /* user function to save each permutation. It is called
                    * from here as 'UsePerm(elements, n, CurrentPermNum)'
                    */
{
  int  temp, i, per_num, x, k, c[MAX_PERM], d[MAX_PERM];

  if ((n > MAX_PERM) || (n < 2))
    return (-1);
  i = 1;
  for (i = 2; i <= n; i++)
    c[i] = d[i] = 1;
  c[1] = 0;
  per_num = 0;
  (*UsePerm)(elements, n, per_num++);  /* just save the permutation */
  while (per_num < nu_of_perms)
  {
    i = n;
    x = 0;
    while (c[i] == i)
    {
         if (!d[i])
           x++;
         d[i] = !d[i];
         c[i] = 1;
         i--;
```

```
    }
    if (i <= 1)
         break;
    k = (d[i]) ? c[i] + x : i - c[i] + x;
    temp = elements[k];
    elements[k] = elements[k+1];
    elements[k+1] = temp;
    (*UsePerm)(elements, n, per_num++);
    c[i]++;
  }
  return (per_num - 1);
}


/* R. DURSTENFELD ALGORITHM
 */
int RandomPermute(elements, n, nu_of_perms, UsePerm)
int elements[];
int n;
int nu_of_perms;
int (*UsePerm)();  /* user function to save each permutation. It is called
                    * from here as 'UsePerm(elements, n, CurrentPermNum)'
                    */
{
  int x, i, temp, c[MAX_PERM];

  if ((n > MAX_PERM) || (n < 2))
    return (-1);
  for (x = 0; x < nu_of_perms; x++)
  {
    /* Fill c[] with random numbers */
    for (i = 2; i <= n; i++)
         c[i] = random(i);  /* between 0 and i-1 */
    /* Now do the inverse mapping */
    for (i = n; i >= 2; i--)
    {
         temp = elements[i];
         elements[i] = elements[c[i]+1];
         elements[c[i]+1] = temp;
    }
    (*UsePerm)(elements, n, x);
  }
  return (nu_of_perms);
}
```

```
/* Factorial computation
 */
int factorial(i)
int i;
{
  int f = 1;

  if (i)
    for (; i >= 2; f *= (i--))
      ;
  return (f);
}


/* Combinations C(n,k)
 *
 * 1. C(n,k) = C(n,n-k) and
 *
 *                  n!          (n-1)!*n          n          (n-1)!
 * 2. C(n,k) = ------- = ------------- = --- * ---------- =
 *               k!*(n-k)!   (k-1)!*k*(n-k)!     k      (k-1)!*(n-k)!
 *                 n
 *            = --- * C(n-1, k-1) = integer
 *                 k
 * hence we can assure that (n * C(n-1, k-1)) / k is an integer,
 * i.e. we do not loose accuracy by using floating point numbers,
 * to caclulate n/k first. Also we do no explicitly calculate the
 * factorials because of the huge numbers involved.
 */
int combinations(n, k)
int n, k;
{
  register int i, c, t;

  if ((k > n) || (n <= 0) || (k < 0))
    return (0);
  if (n - k < n)
    k = n - k;                /* 1st property mentioned above */
  if (k == 0)
    return (1);
  t = n - k;
  c = t + 1;
  for (i = 2; i <= k; i++)  /* based on 2nd property above */
```

```
    c += (c*t)/i;  /* simple calculations on 'c = (c * (n-k+i)) / i' */
  return (c);
}
```

```
/* File: RANDOM.C
 *
 * Part of layer 1: Implementation of random number arithmetic
 *
 * V. Dimakopoulos  -  Victoria 1991
 */

#include <math.h>
#define  RND_MAX  2147483647

double uniform();

/* Returns a number of events, poisson distributed with
 * mean 'lambda'.
 * 'lambda' should be >= 0.
 */
int poisson(lambda)
double lambda;
{
  register double  c;
  register double  product;
  register int     events;

  c = -lambda;
  c = exp(c);
  product = 1;
  events = 0;
  while (events < 500)      /* lets not overdo this */
  {
    product *= uniform(0.0, 1.0);
    if (product < c)    /* reached the condition */
      break;
    events++;
  }
  return (events);
}


/* Returns a random value, with gaussian distribution,
 * with given mean ('mean') and variance ('var').
 * 'var' should be > 0.
 */
double gaussian(mean, var)
double mean, var;
{
```

```
  register int    i;
  register double sum;

  sum = -6.0;
  for (i = 0; i < 12; i++)    /* MUST be 12 */
    sum += uniform(0.0, 1.0);
  sum *= var;
  return(sum + mean);
}


/* Returns a random value, exponentially distributed,
 * with mean 'mean' (equal to 1 / lambda)
 */
double exponential(mean)
register double  mean;
{
  return (-mean*log(uniform(0.0, 1.0)));
}


/* Returns a random value, uniformly distributed over
 * the interval [a, b].
 * Can be used instead of the standard function random()
 * of the system, since it is more versatile, with the same
 * 'randomness'.
 * There should hold:  b > a.
 */
double uniform(a, b)
register double  a, b;
{
  register double  t;

  /* use system-specific functions to get a random number
   * between 0 and RND_MAX, and scale it in [0, 1] */
  t = ((double) random()) / ((double) RND_MAX);
  /* scale it in [a, b] */
  t *= (b-a);
  return (a + t);
}


/* DISCRETE PROBABILITIES
 */
```

```
#define TESTNUM 100000
double  ProbabilityTable[1000];


PrepareDiscreteGaussian(mean, var, min, max)
double mean, var;
int    min, max;
{
  register int i, tests[1000], ir, out;

  if ((max − min) > 999)
    return (0);
  out = 0;
  for (i = 0; i <= (max−min); i++)
    tests[i] = 0;
  for (i = 0; i < TESTNUM; i++)
  {
    /* Try not to have negative numbers */
    ir = (int) (gaussian((double) (1000.0 + mean), (double) var) + 0.5);
    ir −= 1000;
    if ((ir < min) || (ir > max))
      out++;
    else
      tests[ir−min]++;
  }
  for (i = 0; i <= (max−min); i++)
    ProbabilityTable[i] = ((double) tests[i]) /
                          (TESTNUM.0 − ((double) out));
  for (i = 1; i <= (max−min); i++)
    ProbabilityTable[i] += ProbabilityTable[i−1];
  return (1);
}


/* Universal function, whatever the distribution is.
 * The entries in the ProbabilityTable[] are assumed to
 * be loaded using the appropriate PrepareDiscreteXxx()
 * function.
 */
DiscreteProbability(min, max)
int min, max;
{
  register int    i;
  register double x;
```

```
  x = uniform(0.0, 1.0);
  for (i = 0; i < (max−min); i++)
    if (x <= ProbabilityTable[i])
      return (i+min);
  return (max);
}


/* Return the probabilities for each entry of the ProbabilityTable[]
 */
GetProbabilities(Probs, num)
double Probs[];
int    num;
{
  int i;

  if (num < 999)
  {
    Probs[0] = ProbabilityTable[0];
    for (i = 1; i < num; i++)
      Probs[i] = ProbabilityTable[i] − ProbabilityTable[i−1];
  }
}


/* Set-up a user table of probabilities
 */
SetProbabilities(Probs, num)
double Probs[];
int    num;
{
  int    i;
  double sum;

  if (num < 999)
  {
    /* First check for validity
     */
    sum = 0.0;
    for (i = 0; i < num; i++)
      sum += Probs[i];
    if ((sum > 0.999) && (sum < 1.001))   /* should be close to 1 */
    {
      Probs[0] = ProbabilityTable[0];
```

```
      for (i = 1; i < num; i++)
        Probs[i] = ProbabilityTable[i] - ProbabilityTable[i-1];
      return (1);
    }
  }
  return (-1);
}
```

```
# MAKEFILE
# For the Allocation Simulator
#
# V. Dimakopoulos — 1992

MAIN = Objects/main
SIMULATE = Objects/simulate
RANDOM = Objects/random
PERMUTE = Objects/permute
ALLOCATE = Objects/allocate
OBJS = $(MAIN).o $(SIMULATE).o $(RANDOM).o $(PERMUTE).o $(ALLOCATE).o

allocate: $(OBJS)
        cc -o allocate $(OBJS) /lib/libm.a
$(MAIN).o: main.c defs.h screen.c layer2.h
        cc -c -o $(MAIN).o main.c
$(SIMULATE).o: simulate.c simulate.h layer1.h
        cc -c -o $(SIMULATE).o simulate.c
$(ALLOCATE).o: allocate.c
        cc -c -o $(ALLOCATE).o allocate.c
$(PERMUTE).o: permute.c
        cc -c -o $(PERMUTE).o permute.c
$(RANDOM).o: random.c
        cc -c -o $(RANDOM).o random.c
```