Join Algorithms for Parallel Computers for Relations Based on Interpolation Based Grid File

by

Salahadin Adem Mohammed

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

January, 1991

JOIN ALGORITHMS FOR PARALLEL COMPUTERS FOR RELATIONS BASED ON INTERPOLATION BASED GRID FILE

BY

SALAHADIN ADEM MOHAMMED

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE
In

COMPUTER SCIENCE

JANUARY 1991

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHRAN 31261, SAUDI ARABIA

COLLEGE OF GRADUATE STUDIES

This thesis, written by MOHAMMED SALAHADIN ADEM under the direction of his Thesis Advisor and approved by his Thesis Committee, has been presented to and accepted by the Dean of the College of Graduate studies, in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE in INFORMATION AND COMPUTER SCIENCE.

Spec
A
1
M67
C-2
1068380/1068386

THESIS COMMITTEE

DR. M. BOZYIGIT MBozyigit
Thesis Advisor

DR. S. GHANTA S. Ghanta
Member

DR. M.A. AL-TAYYEB [illegible]
Member

Member

[illegible]
Department Chairman
DR. M.A. AL-TAYYEB

[illegible]
Dean, College of Graduate Studies
DR. ALA AL-RABEH

كلية الدراسات العليا
COLLEGE OF GRADUATE STUDIES

24-3-91
Date

This thesis is dedicated

To

My beloved parents, brothers & sisters

for the sacrifices they made to educate me.

ACKNOWLEDGMENT

Acknowledgment is due to the king Fahd University of Petroleum and Minerals for extending all facilities and providing financial support.

I would like to offer my indebtedness and sincere appreciation to my committee chairman Dr M. Bozyigit who has been a constant source of help and encouragement during this research. I greatly appreciate the invaluable co-operation and support extended by Dr. S. Ghanta and Dr. M. A. Al-Tayyeb who served as members of thesis committee. I express my thanks to Dr. M. Ouksel and Dr. Al-Suwaiyel for their help in the initial stage of this work. I also thank A/Aleem, Ilias, Limalia, Khaled, Sami, Abdullah, Purushothaman, M. Al-Barrak, Afa , Ibrahim Idris, Ibrahim Mohammed, Lula and Jemea Ali-Shekh for their extensive help during the time of this thesis.

I owe my father and my mother an expression of gratitude for their patience, understanding and encouragement.

Lastly, but not the least, I am thankful to all faculty, colleagues and friends who made my stay at the university a memorable and valuable experience.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

THESIS ABSTRACT

Title : **Join Algorithms for Parallel Computers for Relations Based on IBGF**

By : **Salahadin Adem Mohamed**

Major field : **Computer Science and Engineering**

The recent advances in parallel and distributed processing and its applications to database operations such as join have initiated extensive research in this field. Investigations on hash based join algorithms compared to methods such as join-index join and merge-sort have given encouraging results. However, They involve a costly data partitioning phase prior to the join. This costly partitioning phase can be avoided if file structures that keep data already partitioned in the secondary storage are used. Interpolation Based Grid File (IBGF) is such a file structure. In this thesis new join algorithms for parallel computers for relations based on IBGF are investigated. Different algorithms are used for uniform relations and nonuniform relations. The efficiencies of these algorithms based on relation and architecture, have been studied using simulation. However the comparison of different techniques is not within the scope of this work.

MASTER OF SCIENCE

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS

DHAHRAN, SAUDI ARABIA

JANUARY 1991

خلاصة الرسالة

عنوان الرسالة: خوارزمية ضم العلاقات المعتمدة على الملفات الشبكية التوليدية في الحاسبات المتوازية ٠

اسم الطالب : صلاح الدين آدم محمد

التخصص : علوم وهندسة الحاسب الآلي

تاريخ الدرجة : كانون ثاني (يناير) ١٩٩١م ٠

مع التطور الحاصل في مجال المعالجة الموزعة والمتوازية وتطبيقاتها فى عمليات قواعد المعلومات كعملية الضم مثلاً ، بدأت ابحاث كثيرة فى هذا المجال .

ادت نتائج الدراسات المنفذة على خوارزية الضم المعتمدة على التفكيك الى نتائج مشجعة مقارنة مع خوارزميات الضم المعتمدة على الترتيب الدمجى وعلى الضم المعاملى . لكن هذه الخوارزميات اظهرت حاجتها الى مرحلة تجزئة مكلفة قبل عملية الضم . ويمكن الاستغناء عن مرحلة التجزئة هذه اذا تم استعمال ملفات تحفظ البيانات مجزئة فى الذاكرة الثانوية للحاسب . ويسمى هذا النوع من الملفات « الملفات الشبكية التوليدية » .

يقدم هذا البحث خوارزمية جديدة للضم للعلاقات المعتمدة على الملفات الشبكية التوليدية فى الحاسبات المتوازية . وقد تم دراسة خوارزميات مختلفة للعلاقات المتماثلة والعلاقات غير التماثلة . كما تم دراسة فعالية هذه الخوارزميات على اسا العلاقات والتراكيب باستخدام المحاكاة .

درجة الماجستيرفي العلوم

جامعة الملك فهد للبترول والمعادن

الظهران - المملكة العربية السعودية

كانون ثاني (يناير) ١٩٩١

CHAPTER 1

INTRODUCTION

In relational database queries, *join* (⋈) is an operator involving the retrieval of more than one relation. Moreover, the join operation has a number of applications besides query processing in relational databases. These include computation of paths in directed graphs and recursive queries in deductive databases. Thus its optimization is essential. Several algorithms have been proposed for implementing join operations in relational databases. Well known examples are *nested loops*, *sort merge* and *hash based* join algorithms [*DEW*85]. With the advance of multicomputer and multiprocessor architectures, join algorithms have been reinvestigated to exploit underlying parallelism of these computers, thus reducing the completion time of join operation.

Several join algorithm strategies have been introduced for parallel machines, but almost all these strategies involve a costly operation of data space partitioning prior to the actual join [*BIT*83], [*DEW*85]. Reading unpartitioned data from the secondary memory and then partitioning causes a significant I/O overhead, making it a bottleneck for relations too big to fit into the main memory as join operands. The I/O bottleneck problem of non-resident data can not be remedied by building large and sophisticated machines because there is no architectural solution to mapping presumably infinite data in the secondary memory to a limited main memory. To reduce the severity of this I/O bottleneck, it is necessary to organize the data itself in a way that allows efficient partitioning of the data space on the secondary memory. This will reduce the virtual address space to finite non-overlapping subspaces. Obviously that kind of partitioning scheme has to be multidimensional to be able to use it for any prospective join attribute. And it should be able to preserve order within the dimensions to avoid processing of incompatible partition pairs and thus to exploit the parallel processing power provided by database machines. Interpolation Base Grid File (IBGF) satisfies the above mentioned properties [*OUK*85*a*].

IBGF breaks down the virtual address space into manageable subspaces. The motivation behind this is to reduce the gap between the secondary and the main memory capacities and to cluster the data while assuring disjointedness of the subspaces to achieve linear complexity. To perform a join between relations $R(A_1, A_2, \dots, A_n)$ and $S(B_1, B_2, \dots, B_m)$ having been partitioned into subspaces $R_1, R_2, \dots, R_u$ and $S_1, S_2, \dots, S_v$ respectively, the global join is broken down into a set of subjoins

$$R_i \; \Xi_{A_k = B_l} \; S_j, \text{ for all } i \text{ and } j, \;\; 1 \leq i \leq u, \;\; 1 \leq j \leq v$$

Because the partitions are disjoint, the union of subjoins is equivalent to the global join. The partitioning and the clustering properties of IBGF guarantees that

$$R_i \; \Xi_{A_k = B_l} \; S_j = \text{Null, for all } i \neq j \qquad \textbf{(1.1)}$$

and the subjoin reduces to

$$R_i \; \Xi_{A_k = B_l} \; S_i \quad \text{for all } 1 \leq i \leq u \; i \qquad (1.2)$$

between the compatible partitions R_i and S_i.

The hash based schemes expend their costliest first in hashing and subsequent sifting and distribution of data to achieve (1.1) and (1.2). Where as in IBGF the conditions (1.1) and (1.2) are readily satisfied for all k and l without restoring to any form of processing or reorganization. The reason for this is the way the data file and the directories are dynamically organized [*OZK*88].

The main characteristics of IBGF are [*OUK*85*a*]:

a. Search time is constant.

b. Directories and data have identical structure and search characteristics.

c. No overflow buckets or chaining is needed.

d. Directories and data files can be stored any where because their file address is kept in the directory.

e. No free space needs to be kept in the directory and the data file.

f. The range of values the attribute domain can take is known and fixed.

A review of parallel computers and some existing join algorithms is discussed in Chapter 2. In Chapter 3 IBGF is discussed. The hardware model used in this research is discussed in Chapter 4. Join algorithms for parallel computers using IBGF for relations with uniformly distributed records is discussed in Chapter 5. Join algorithms for relations with nonuniformly distributed records are discussed in Chapter 6. In Chapter 7 the simulation of the algorithms of Chapters 5 and 6 is discussed. The simulation of the join algorithms is discussed in Chapter 7. Analysis of the join algorithms discussed in Chapters 5 and 6 is given in Chapter 8. Conclusion and advice for further research in this area is given in Chapter 9.

CHAPTER 2

LITRETURE REVIEW

2.1. Parallel Computers

Parallel computers are classified by their memory organization, processor organization, and the number of instruction stream [*GAJ*85].

A parallel computer in which at least two processors share a common memory or a common memory address space is called a multiprocessor. A multicomputer on the other hand is a parallel computer which has neither a shared memory or a shared memory address. In a multicomputer, if a processor wants to use a data stored in the memory of a remote processor, it must explicitly bring the data into its local memory. This and all other inter-processor communication is done by passing messages via a network among the processors [*MIC*88].

How a multicomputer is organized is determined by its network organization. Parallel computers are organized in various topologies. Some of the common networks are tree, ring, mesh, and hypercube [*FEN*81]. The organization of tree ring and mesh is self explanatory, but a hypercube needs some elaboration.

In a hypercube of dimension n we have 2^n processors labelled 0 to $2^n - 1$. Processors P_i and P_j are directly connected (neighbours) if their labels differ in only one bit, when written in binary. Each computer has n output buffers labelled 0 to n-1 starting from the right [*GAJ*85].

The two common categories of instruction and data streams in parallel computers are single instruction and multiple data (SIMD) and multiple instruction and multiple data (MIMD). In SIMD parallel computer, all processors execute the same instruction synchronously while in MIMD parallel computer different processors may execute different instructions asynchronously. In this thesis a MIMD computer is assumed to work in SIMD mode [*FEN*81].

Two terms - *speedup* and *Processor-use efficiency* are commonly used to evaluate the efficiency of a parallel computer program.

Definition : Let T_0 be the time required to solve a problem using the fastest single processor program for a problem. Let T_k be the time required by the parallel computer program when k processors are in use. The *speedup* obtained by the parallel computer is $S_k = \frac{T_0}{T_k}$. The *processor-use efficiency*, E, is $E = \frac{S_k}{k}$. Barring any anomalous behavior, the strive is for $S_k = k$ and $E = 1$. In practice because of the inter-processor communication overhead, S_k will generally be less than k and E less than 1 [*GAJ*85].

2.2. Parallel join algorithms

In this section three parallel join algorithms are discussed. The first one is based on hybrid hash (HH) [*OMI*88], [*TIE*87] and the second one is based on join index (JI) on both hypercube (n-cube) and ring architecture [*OMI*88],

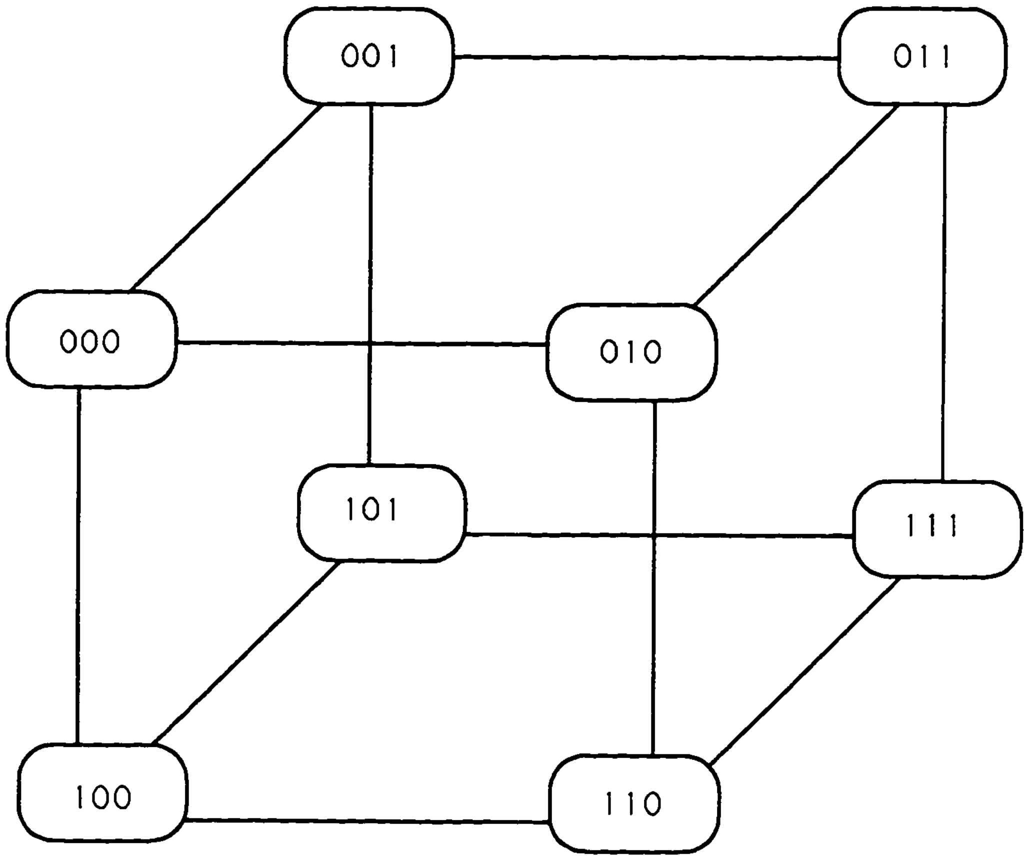


Figure 2.1 :   3-cube (hypercube architecture)

[*PAT*87], [*SAN*88]. The third one is parallel nested-loop join algorithm [*CHA*87]. The hardware models used are described in the following subsection. For the first two algorithms a uniform data distribution is assumed, i.e. tuples from each relation are evenly distributed among the processors.

2.2.1. Ring & cube connected multicomputers

The boolean n-cube model used here is like the hypercube discussed above. Every node has $\log_2 N$ neighbours (links), where N (power of 2) is the total number of nodes in the system. Each neighbour has its address differing in one bit and there are $n = \log_2 N$ bits in an address [*OMI*88]. Figure 2.1 shows a 3-cube with 8 nodes.

In contrast each node in the ring connected multicomputer has 2 neighbours. They will be specified as the left and right neighbours for each node. Figure 2.2 shows a ring with 8 nodes.

Each node consists of a processor, main memory, and secondary storage. Each node is assumed to have independent communication processors capable of simultaneously receiving/transmitting packets along two separate links from and to each neighbour. Each node is assumed to contain two registers called the tuple count registers (TCR's). Nodes are allowed to write into their own TCR's TCR1 and TCR2, and the two TCR's can be read simultaneously by any two of the n neighbors of a node. Nodes exchange data via variable site packets with an upper bound imposed on packet size. The upper bound is determined based on the various hardware parameter values such as CPU speed, communication

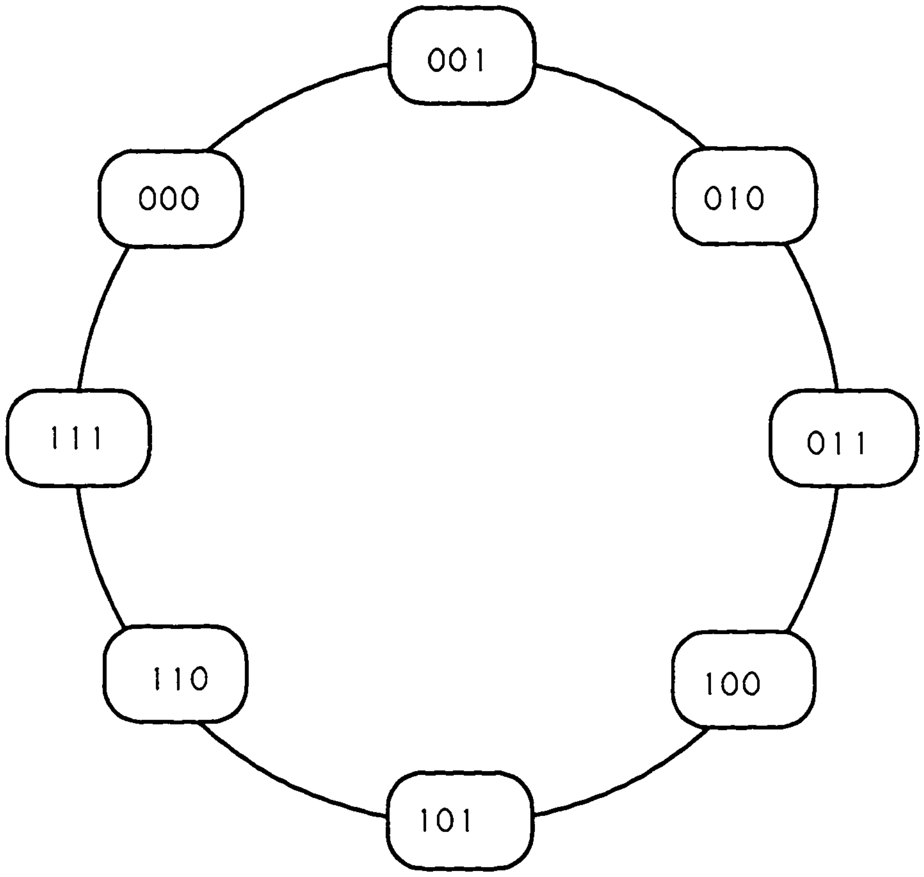


Figure 2.2 : A ring of 8 processors

speed etc. Parallel computers based on the cube connection are already in existence, e.g. cosmic cube [*SEI*85], the IPSC [*INT*85] and NCUBE [*HAY*86].

2.2.2. Parallel HH Join Algorithm

In general, hash join algorithms partition the source relations R and S into disjoint subsets called buckets: $R_0, R_1, \ldots R_{n-1}$ and $S_0, S_1, \ldots S_{n-1}$. Tuples of a relation with the same join value will share the same bucket. Since the same partition scheme is used for both relation, tuples in bucket R_i will only have to be joined with tuples in bucket S_i. Hence, a join of two large relations is reduced to separate joins of many small disjoint subsets of each relation.

The uniprocessor version of hybrid hash-join consists of two phases. In the first phase, both source relations R and S, are partitioned one at a time into buckets $R_0, R_1, \ldots R_{n-1}$ and $S_0, S_1, \ldots S_{n-1}$. While R being partitioned, bucket R_0 is actually used to build a hash table. Thus when S is partitioned, tuples belonging to S_0 can be used to probe the hash table immediately. In the second phase, the remaining n-1 corresponding buckets are processed. That is, tuples in bucket R_i are used to build a hash table and tuples in bucket S_i are used to probe the hash table.

2.2.2.1. HH join for cube connected multicomputers

Each node allocates one output buffer for each neighbour and excess memory is used to build a hash table for those tuples hashing into the bucket assigned to this node.

The algorithm proceeds as follows:

Each P_i

1. Forms an empty hash table.
2. Reads a tuple from the smaller relation.
3. Hashes the join attribute value.
4. Add the tuple to the hash table if the tuple hashes to it, otherwise sends it to an appropriate buffer together with its destination address.
5. Repeats steps 2, 3, 4 for all the tuples of the smaller relation residing in its storage element.
6. Sends tuples from one of its buffers to one of its neighbours and accepts tuples from it.
7. Directs each tuple received from a neighbour to one of its buffers or to its hash table depending on the address of the tuple.
8. Repeats steps 6 and 7 for all the tuples, till all its buffers are empty.
9. Reads a tuple from the bigger relation.
10. Hashes its join-attribute value.
11. Performs join on the tuples of the smaller relation already on its hash table and this new tuple, if this new tuple hashes to it, otherwise it sends this tuple to one of its buffers.
12. Repeats steps 9, 10 and 11 till all the tuples of the bigger relation, in its own storage element, are done.
13. Sends tuples from one of its buffers to one of its neighbours and accepts tuples from it.
14. Directs each tuple received from a neighbour to one of its buffers or to join

with the tuples of the smaller relation which are already in the hash table.

15. Repeats steps 13 and 14 for all the tuples, till all its buffers are empty.

2.2.2.2. HH Join for a ring

The algorithm for the ring is identical to that of the hypercube except we have to deal with one output buffer here instead of *n* output buffers.

The algorithm performs better in a hypercube, for tuples have to travel shorter distances hence sending and receiving times are much fewer in a hypercube than in a ring architecture.

2.2.2.3. HH Join drawbacks

Reading the unpartitioned and unclustered nonresident data causes on line I/O bottleneck problem. Prior to the join, a costly partitioning step was under taken by the algorithm. Both the I/O bottleneck problem and the partitioning step add to the cost of the join algorithm drastically. Also assumption of limitless capacity of buffers is not realistic [*DEW*85].

2.2.3. Parallel JI Join Algorithm

Each entry in a join index is a pair of surrogates (tuple identifiers) which correspond to a pair of joining tuples from two relations as shown in Table 2.1. The join index join algorithm accesses matching tuples of two relations using the join index In a uniprocessor system, a naive version of this algorithm proceeds as follows: Read in the join index page by page, read in the matching tuples for

Relation R(parts)

Sur	Number	Name
1	240	Nail
2	300	Hammer
3	250	Screw
4	450	Wire
5	100	Tape
6	530	Lock
-	-	-

Relations S(Suplier-part)

Sur	Sname	Pnumber
1	Salah	200
2	Ismail	300
3	Khaled	100
4	Sami	100
5	Hussain	530
6	Yahya	450
7	Jaweed	200

Join index for R

Rsur	Ssur
2	2
3	1
3	7
4	6
5	3
5	4
6	5

Join index of S

Ssur	Rsur
1	3
2	2
3	5
4	5
5	6
6	4
7	3

Table 2.1 : Join indices of R and S.

both relations and perform the join operation.

In a multicomputer, two join indices are assumed to be maintained on every node to facilitate the operation, each one is based on one of the two joining relations. For example, if R and S are the two relations to be joined, the join index based on R for a node will be a collection of surrogate pairs composed of the local R tuples with corresponding joining S tuples. Because the corresponding S tuple may not be stored within the same node, we assume each S surrogate has been tagged with the node address in the join index for R. This is useful in the parallel version of the join index algorithm, since we need this information to deliver corresponding joining tuples to their appropriate nodes. Each node has a buffer to store the index for a relation, a buffer to store the join index, a buffer to optimize the page access and a buffer to store the matching tuples.

2.2.3.1. Join Index (JI) Join Algorithm for a hypercube

In JI each entry is a pair of surrogate. A surrogate is system generated unchangeable unique tuple identifier. There is an index which maps each surrogate value in the JI to its tuple. The algorithm uses the JI and the index to form the join from the actual tuples. As in HH join, a hypercube processor organization and the same output buffer allocation scheme can be used.

The algorithm proceeds as follows:(Assume R and S as operand relations.)

Each P_i

1. Reads JI based on S and S index.

2. Reads an S tuple.
3. Stores the tuple, if it has to be joined locally with some R tuple otherwise sends the tuple together with its destination address to one of its buffers.
4. Repeats steps 2 and 3 for all the tuples of S residing in its storage element.
5. Reads JI based on R and R index.
6. Reads an R tuple.
7. Joins the R tuple with some S tuple if this tuple has to be joined locally, otherwise it sends it to an appropriate buffer.
8. Repeats 5, 6 and 7 till all the tuples of R residing in its storage element are done.
9. Sends tuples of one of its buffers to one of its neighbours and receives tuples form the same neighbour.
10. Performs join between the tuples of S on its hash table and the tuples of R addressed to it. It directs tuples addressed to other processors to an appropriate buffer.
11. Repeats steps 9 and 10 till all its buffers are empty.

2.2.3.2. JI Join drawbacks

1. When ever a new tuple is inserted into a relation, this information must be propagated to all the processor to update their join indices
2. All processors have to send some information about the matching surrogates to the processor that is storing the new tuple to update its join index.
3. Not only the join index table but each index table in a processor must be

updated.

4. It includes nearly all the drawbacks of HH.

2.2.4. Parallel Nested-loop

In the parallel nested-loop algorithm, the tuples of the smaller relation are sent to all processors containing tuples of the other, larger, relation. This is achieved by forming a ring of processors in the cube, as described in Section 2.2.4.3, and pipelining the data of the smaller relation around the ring.

The best performance is, obviously, achieved when R and S are uniformly distributed across the nodes. The algorithm presented here ensures such a distribution by performing tuple balancing as the first step of the algorithm. Once R and S are balanced, the degree of parallelism is increased by having multiple rings inside a cube, each performing the same join. To achieve this, the data related to the smaller relation are replicated in each ring. The second step of the join algorithm, called the merging step, packs the data of the smaller relation into the smallest possible sub-cube thus maximizing the number of sub-cubes, each performing the join in parallel. Data are exchanged by transmitting packets that are packed as "full" as possible in the merge step. For the nested-loop algorithm, this packing minimizes the CPU idle time between packet arrivals. Finally, in the join step, tuples from the smaller relation are pipelined around the ring and local joins are performed in parallel at each node.

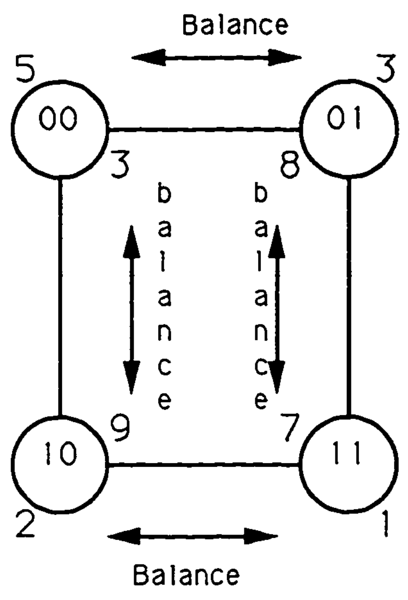


Figure 2.3 (a)
Tuple balancing step 1

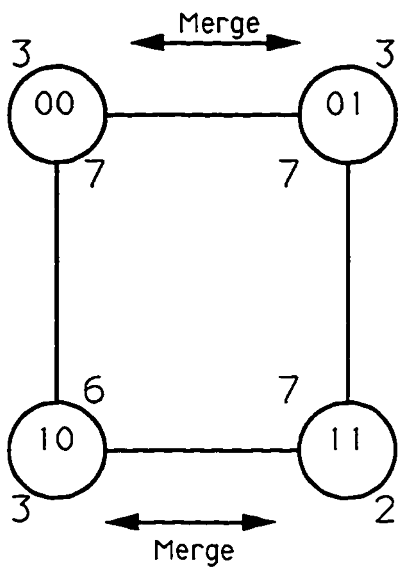


Figure 2.3 (C)
Merging

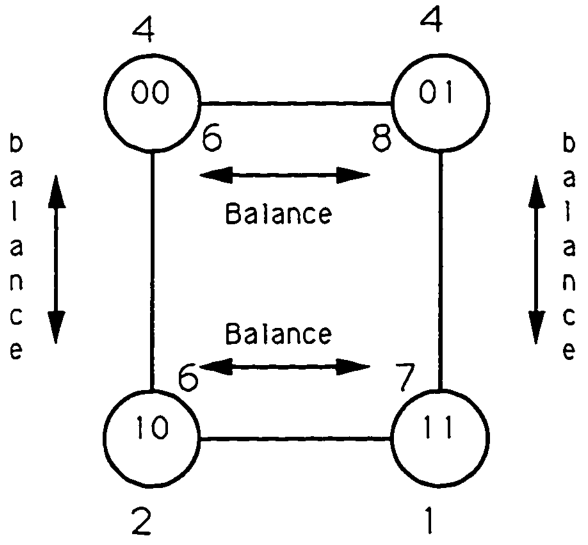


Figure 2.3 (b)
Tuple balancing step 2

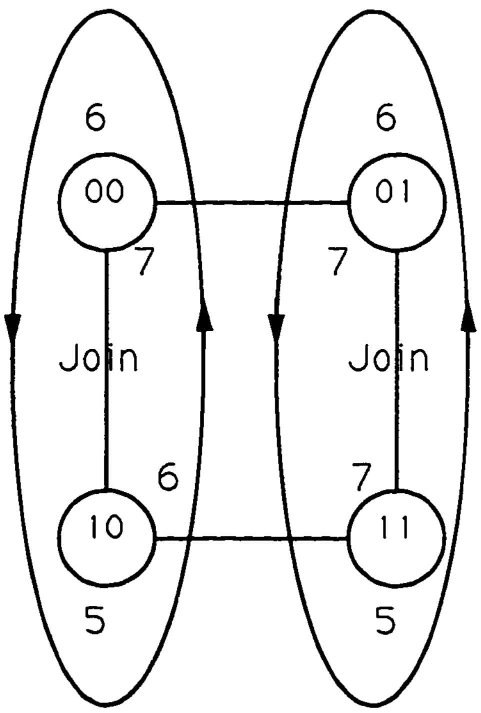


Figure 2.3 (d)
Cycle & join

2.2.4.1 Tuple balancing

Consider a 2-cube connecting 4 nodes, as shown in Figure 2.3(a) The nodes of the cube are associated with an *n-bit* ($n=\log_2 N$) addresses as shown. Let $\xi_{R,i}$ and $\xi_{S,i}$ denote the number of tuples of R and S at node i, respectively ($0 \leq i \leq 3$). The tuple distribution of the two relations is as follows, $\xi_{R,0}=5$, $\xi_{R,1}=3$, $\xi_{R,2}=2$ and $\xi_{R,3}=1$ and $\xi_{S,0}=3$, $\xi_{S,1}=8$, $\xi_{S,2}=9$ and $\xi_{S,3}=7$. Tuple balancing proceeds in j steps. ($1 \leq j \leq \log_2 N$) where nodes that differ in address in the j^{th} bit balance tuples of R1 while, simultaneously, node that differ in address in the $(n-j+1)^{th}$ bit balance tuples of S. In this example it is assumed that each fixed sized packet can hold a maximum of 6 tuples. Figure 2.3(a) shows the distribution of R and S before balancing. The $\xi_{R,i}$ and $\xi_{S,i}$ values are loaded into the tuple count registers, TCR's, of each node. In the first stage, one of the communication processors of node 0 reads TCR1 of node 1 and determines that it has to transfer 1 tuple of S to node 1 in order to make $\xi_{R,0} = \xi_{R,1} = 4$. Simultaneously the other communication processors reads TCR2 of node 2 and determines that it should receive 3 tuples of S from node 2 to make $\xi_{S,0} = \xi_{S,2} = 6$. The distribution of tuples after the first stage ($j=1$) is shown in Figure 2.3(b) and the distribution after the second stage ($j=2$) is shown in Figure 2.3(c).

If maximum that any node is allowed to transmit to any of its neighbours is one packet, then the entire balancing operation will take a total of n packet sends. The more nodes there are in the cube, the better the distribution becomes because of the increase in the number of neighbours to which a node can pass its excess tuples. If each node is permitted to transmit more than one packet to each neighbour then there will be a corresponding increase in the total number of packets transmitted.

2.2.4.2 Merging

After tuple balancing, each node has roughly the same number of tuples for R and S. Since all nodes have the same number of tuples for R and S, Each node can independently determine by either a software or a hardware mechanism as to which of R and S is smaller. In the software approach two aggregation operations are performed to compute the total number of tuples of R and S. The host node then knows which relation is smaller and broadcasts this information back into the cube.

Since tuple balancing is performed before merging, conflicts in decision can arise only when both relations are of about the same size. In this case either one of the two can be selected as the smaller relation.

Merging may be performed on both relations, R and S . In the case of the smaller relation, R, There are two reasons for merging. First, to ensure that packets carrying the tuples of R as full as possible. Second, to duplicate tuples of R in multiple sub-cubes so that a ring can be established in every sub-cube to increase the parallelism of the last, costly phase of the join operation. For the larger relation, S, an optimum number of tuples per node can be specified based on system parameter values in order to balance computation and communication. Merging of S may some times be required to reach this optimum value. Merging occurs between pairs of nodes at a time. Upon completion, in the case of the smaller relation, both nodes contain the union of the original tuples stored at each nodes. A variation of this can be employed for the larger relation such that only one of the two nodes need contain the result.

2.2.4.3 Join

Join is performed by forming rings of processors and sending tuples of the smaller relation around the ring. This is referred as *cycling* . Cycling is also used to collect tuples following a merge. Each node has the value of k, which indicates the point at which merging stopped in the previous phase. Rings are formed in the cube based on the n-k most significant bits of address. Thus, there are 2^k rings of 2^{n-k} nodes each. If no merging was possible, then $j=k=0$ and only a single ring containing all N nodes is formed. If $j=k=n$, then each node has all the tuples of R1 and maximum parallelism is achieved in the join phase. A ring is formed by sequencing nodes such that their addresses form a Gray code.

If each node can send all its R1 tuples in one packet, then it takes 2^{n-k} packet send time units to send all packets around the ring and return each to its sender. If each node is able to join a packet of R1 tuples with its local segment of R2 in one time unit, then the join is also performed in 2^{n-k} time units. In Figure 2.3(d), one step of merging was performed. Nodes 0 and 1 merged their tuples resulting in each node having 6 tuples (1 packet) of R1. Similarly, nodes 2 and 3 merge tuples resulting in 5 tuples in each node. Thus, in this example, $j=k=1$ and n-$k=2$-$1=1$. Therefore, in the join phase there are $2^k=2$ rings of $2^{2-k}=2$ nodes each, with each ring performing the join in parallel.

CHAPTER 3

INTERPOLATION BASED GRID FILE (IBGF)

3.1 Data file

The physical data organization of an Interpolation Based Grid File can be invisioned as a file of cardinality N whose elements are records each consisting of d-dimensional tuple $k = (k_0,k_1,k_2,....k_{d-1})$ of values which correspond to attributes $A_0,A_1......A_{d-1}$ respectively, where d is an integer number greater than 0. The file is assumed to reside on a direct access device such as magnetic disk. The storage space is divided into fixed-size blocks called buckets or pages. A bucket is a unit of transfer between secondary and primary memory. The retrieval time of a particular record is measured in terms of buckets access to get that record.

Each component value of a record $k = (k_0,k_1,......k_{d-1})$ can be mapped to a real number in the half open interval $[0, 1)$. Since the domains are considered bounded, a conversion to this form can simply be accomplished by scaling. Let $k^0 = (k^0_0, ..., k^0_{d-1})$ be the result of the mapping. The superscript of k^0 indicates the level in the hierarchy of directories to which k^0 belong. Directory hierarchies will be elaborated later in this chapter. Each record can now be viewed as a point in the d-dimensional space $U^d = [0, 1)^d$. For example, consider the two dimensional case where $D_0 = (0,50000)$ and $D_1 = (0,80)$, where D_i is the domain of attribute A_i. A Record $K=(37000,10)$ will be mapped to $K^0 = (\frac{37500}{50000}, \frac{10}{80})$

or (0.110, 0.001) in binary of $U^2 = [0,1) \times [0,1)$. Before discussing the interpolation based grid file, let us first define some of the terminology used in the next paragraphs.

Definition 1: Given a set of intervals along the i^{th} axis of U^d defined as follows:

$I(l_{R,i}) = [\ I(l_{R,i,k}) \mid I(l_{R,i,k}) = |\frac{k}{2^{l_{R,i,k}}},\ \frac{(k+1)}{2^{l_{R,i,k}}}|\ ]$ where $0 \le k \le\ 2^{l_{R,i}} - 1$ and R is any relation. $l_{R,i}$ is called the interval partition level along axis i of relation R. The number of intervals in $I(l_{R,i})$ is $2^{l_{R,i}}$ [*OUK*85*a*].[]

Definition 2: The search space partition level l_R is defined as the summation of interval partition levels along the d axes forming the search space, that is $l_R = \sum_{i=0}^{d-1} l_{R,i}$. So, the number of partition in the search space is 2^{l_R}. We shall denote by l_R for a relation R based on interpolation-based grid file of search partition level l [*OUK*85*a*].[]

Given an interval $I_{R,i,k}$, its corresponding interval in domain D_i can be determined in a straight forward manner. For example if $D_i\,[\min D_i, \max D_i)$ then $[\frac{k}{2^{l_{R,i}}} \cdot q_i + \min D_i,\ \frac{(k+1)}{2^{l_{R,i}}} \times q_i)$ *where* $q_i = |\max D_i - \min D_i|$ is the interval in D_i corresponding to $I_{R,i,k}$. Without loss of generality, we shall assume $\min D_i = 0$. The interpolation based-grid file representation is best illustrated by tracing an example. To simplify the discussion, only the two-dimensional case is presented. The data search space is thus viewed as the rectangle delimited by the cartesian product $D_0 \times D_1$ of attributes A_0 and A_1. The vector k = (k_0, k_1) refers to the coordinates of a record in the search space, where as k^0 = $(k^0{}_0 , k^0{}_1)$ refers to its mapping $U^2 = [0,1)^2$.

In the example, we shall first examine how the search space is partitioned under repeated insertions. An insertion of a record to the database will be represented by exhibiting a dot in the graph. We shall assume that the data file bucket capacity $b_0 = 2$ and each partition may thus contain at most 2 dots.

Initially, a single partition of the search space exists since as for now the data space contains only 2 records. (See Figure 3.1.a). The Figure shows the correspondence between the data search space and its mapping to U^2. Note that the $D_0 \times D_1$ is $I_{R,0,0} \times I_{R,1,0}$. The search space partition level is 0.

An additional insertion causes partition delimited by $I_{R,0,1} \times I_{R,1,0}$ to split. We shall adopt the policy of cyclicly partitioning the search space along the various axes, thus the overflowing partition must be split along axis 1 as illustrated in Figure 3.1.c.

Several remarks can be made at this point concerning the relationship between the coordinates and the insertion process.

1) The last insertion caused the implicit partitioning of the search space into four possible partitioning $I_{R,0,0} \times I_{R,1,0}$, $I_{R,0,0} \times I_{R,1,1}$, $I_{R,0,1} \times I_{R,1,0}$ and $I_{R,0,1} \times I_{R,1,1}$.

2) A subspace delimited by $I_{R,0,i} \times I_{R,1,i}$ can simply be represented by its

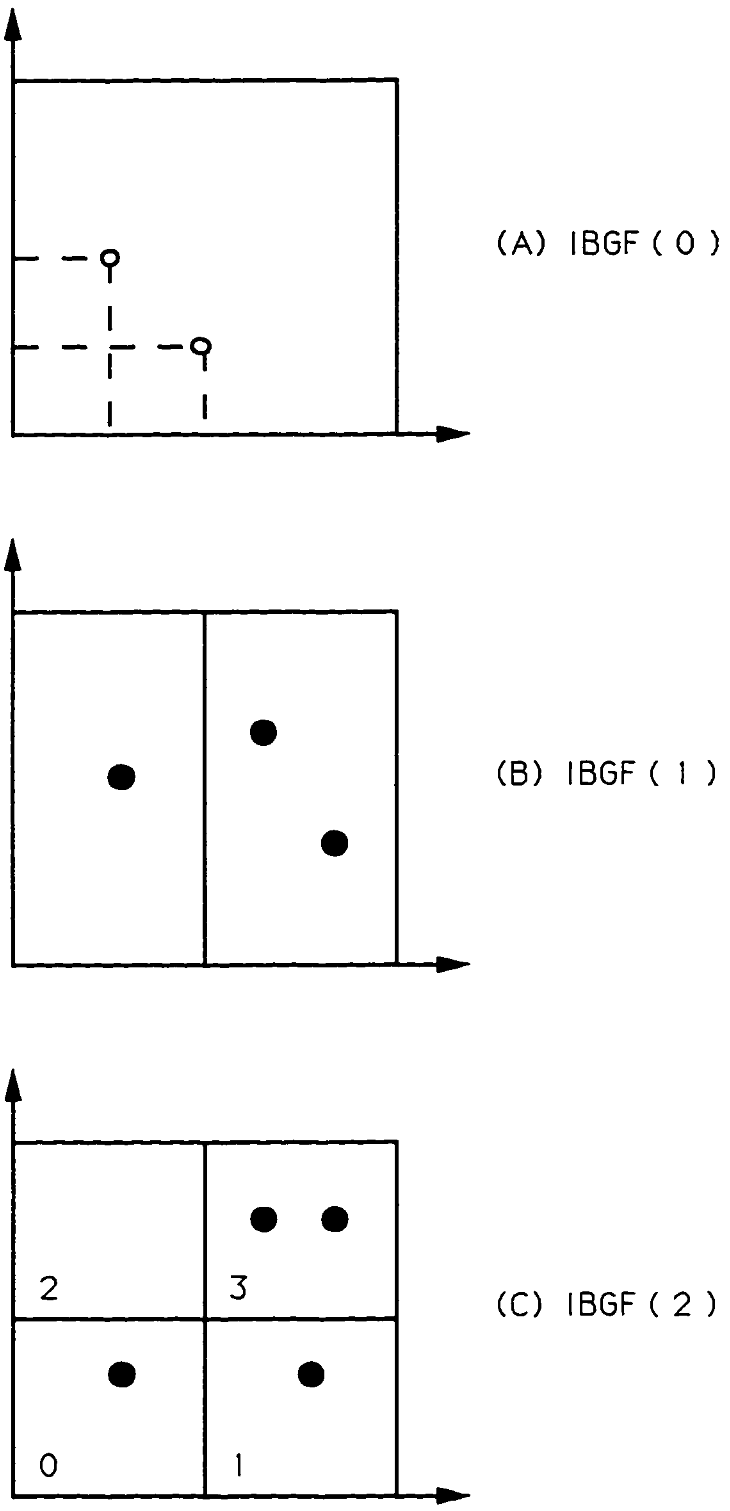


Figure 3.1 : IBGF insertion process
n in IBGF(n) is the partition
level. Bucket size = 2

coordinates (i,j). So, in the example, the possible pairs are (0,0), (0,1), (1,0), (1,1).

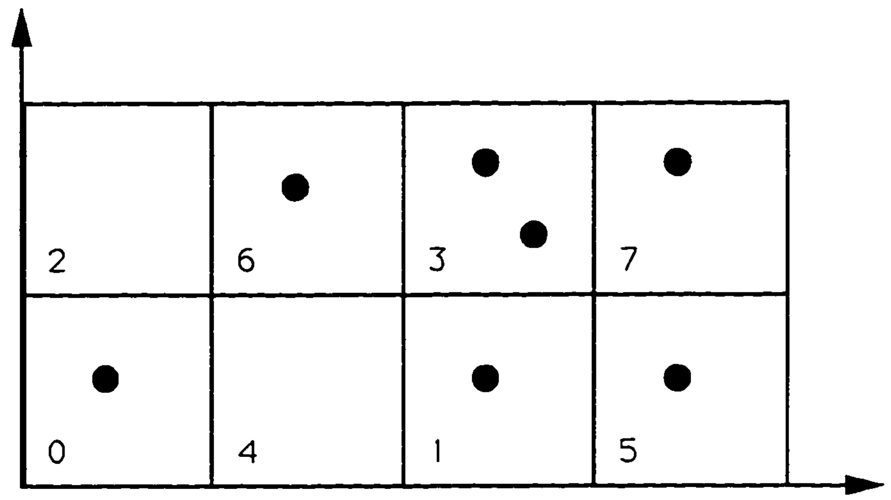
3) A pair of coordinates represent the leading binary digits of the fraction part of k_0^0 and k_0^1 respectively. In other words, coordinates are simply prefixes of elements located in the subspace they determine. The length of this prefixes is exactly the interval partition level along the corresponding axis.

4) Both partition (0,0) and (0,1) should still be represented by partition (0,0) since this later partition contains less than two elements (the bucket size) and therefore, its splitting is not necessary.

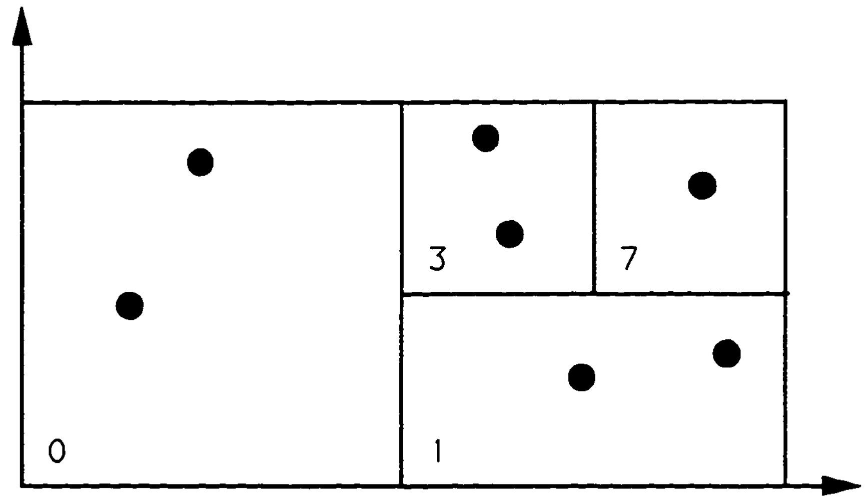
The followings take these remarks in consideration to device a mechanism that will work in both uniform and nonuniform data distribution.

3.2 Storage mapping

Given a record $k = (k_0,k_1.....k_{d-1})$ the record is mapped to $k^0 = (k_0^0,k_1^0......k_{d-1}^0)$ where $k^0_i = \dfrac{k_i - \min D_i}{q_i}$ and $q_i = |\max D_i - \min D_i|$ and D_i is the domain of attribute A_i. Let $c^0 = (c_0^0,c_1^0......c_{d-1}^0)$ denote the coordinates of the partition table where k is contained. Then each c_i^0 can be determined from k_i^0 using the interval partition level $l_{R,i}$: $c^0_i = \lfloor k^0_i \times 2^{l_{R,i}} \rfloor$ for $0 \leq i < d$. For instance, let us consider the case discussed at the beginning of this chapter where k = (375000,10). This record was mapped to $k^0 = (0.110, 0.001)$ in binary. Assume the search partition level is $l_R = 3$ as in Figure 3.2, then the



(A) STORAGE MAPPING



(B) IBGF (3)

Figure 3.2 : Partitions after splittig twice along axis-0 and axis-1

coordinates of the partition where k is stored are $c^0 = (11, 0)$ in Binary since $l_{R,0} = 2$ and $l_{R,1} = 1$. Let also c^0_{ij} indicate the j^{th} binary digit of c_i starting from the left. Then, assuming a cyclic splitting policy, the storage mapping is completely characterized by the following theorem.

Theorem I: The number of partition in which a record $k = (k_0, k_1, ..., k_{d-1})$ may be contained is given by [*OUK*85*a*]:

$$M(k,l) = \sum_{i=0}^{d-1} 2^i \sum_{j=0}^{l_{R,i}} 2^{d(l_{R,i}-1-j)} c^0_{ij} \qquad \square$$

This mapping is illustrated in Figure 3.2.a where partition numbers are indicated at the bottom left corner. Note that all the numbers are consecutive in the range $0, 1, ..., 2^{l_R} - 1$ If the records in the search space are not uniformly distributed some of the partition in the range $0, 1, ..., 2^{l_R} - 1$ might contain no record at all. So assigning space to such empty partitions will result in poor space utilization. The number of empty partitions will increase exponentially with each round of splits along a given axis. The section below discusses the solution proposed to this problem.

3.3 Mapping partitions to buckets

As discussed above in a one-to-one mapping between 2^{l_R} partitions forming the search space (assume the search partition level is l_R) and the buckets will result in an extremely poor utilization of storage space. This can be further worsened by skewed data distribution. Clearly a better approach will be to merge several of those partitions into the largest possible region of the several spaces so long as the number of records does not exceed the bucket size. The objective is to minimize the number of regions thus formed and thereby the number of buckets. The rules governing the formation of regions can be stated as follows :

Rule 1: Let p_i be the initial partition number forming a region . Then the region can be expanded by adding any partition obtainable through a split of p_i that is partition p_i+2^j where $\log p_i < j < l$ as long as the number of records in the region does not exceed the bucket size [*OUK*85*a*].[]

Rule 2: If a partition p_j is merged into the same region as p_i, then all partitions obtainable through a split of p_j are also merged into the same region [*OUK*85*a*].[]

The first rule defines the initialization of the region forming process, while the second specifies the constraints on any partition merged into the same region. Inherent in this rule is the recursive merging procedure. This is particularly obvious if we assume $p_i=0$ Then all partitions in the search space can possibly collapse into a single region; namely the whole search space.

Definition 3 : A partition p_j is said to be embedded in partition p_i if it is merged into the same region p_i by applying the rules specified above. Partition p_i is referred to as the embedding partition and p_i is the region identifier [*OUK*85*a*].[]

Figure 3.3 illustrates the partitioning of the search space into regions. Observe that partition 2 and 6 are embedded in partition 0. Partition 5 is embedded in 1. Partition 2 is obtainable from 0 through split and partition 6 is

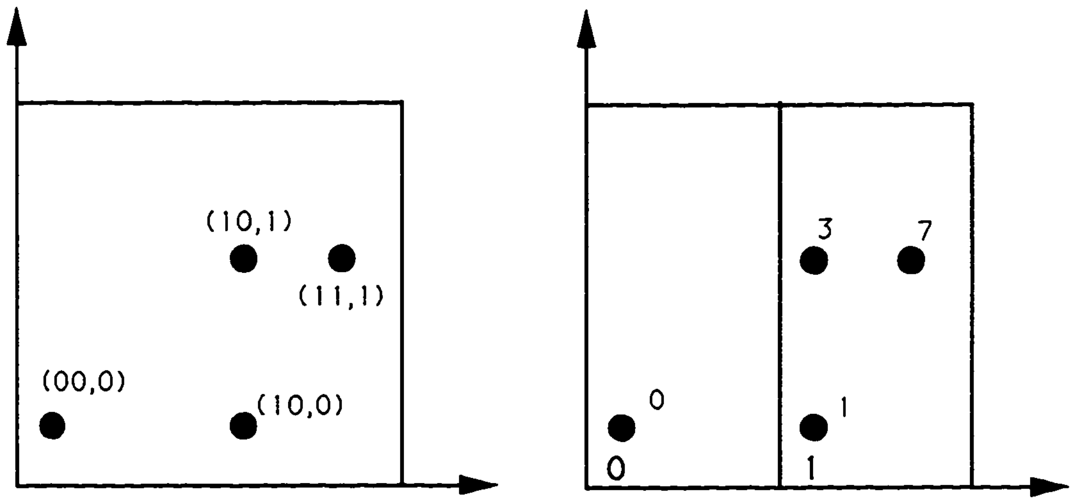


Figure 3.3 : Directory after first split

obtainable from partition 2 .

3.4 Directory

Figure 3.2.b represents a more natural partitioning of the search space than Figure 3.2.a since it is obtainable by splitting overflowing partitions only. Several partitions of Figure 3.2.a have coalesced to form a single region. For instance, partitions 0, 2, 4, and 6 form the single region 0. The major problem now is the fact that the region number are no longer consecutive. Specifically the search space is composed of regions 0, 1, 3 and 7 . Therefore a directory is necessary to hold the mapping of these regions into the physical storage.

Each item stored in the index will consist of two parts : a region number and the address of the bucket assigned to this region. The region number may be considered as the logical address of a bucket. Note that each partition is also uniquely identified by its coordinates c^0_j, $0 \leq j < d$. The organization of the directory is identical to that of the data space.

Let $k^1 = (k^1_0, k^1_1, \ldots, k^1_{d-1})$ be the vector obtainable from c^0 (defined as above) as follows: $k^1_i = |c^0_i \times 2^{2^{R,j}}|$ for $0 \leq i < d$. Clearly k^1 can be deduced from k^0 by truncation. The directory may now be regarded as one of the storing record k^1 whose components are directly obtainable from the coordinates of the data file partitions. Note that the components of records k^1 have their values in $U = [0, 1)$ therefore the same methodology applied to organize the data records are used to structure the directory. In other words, the data file and the directory file have identical structures.

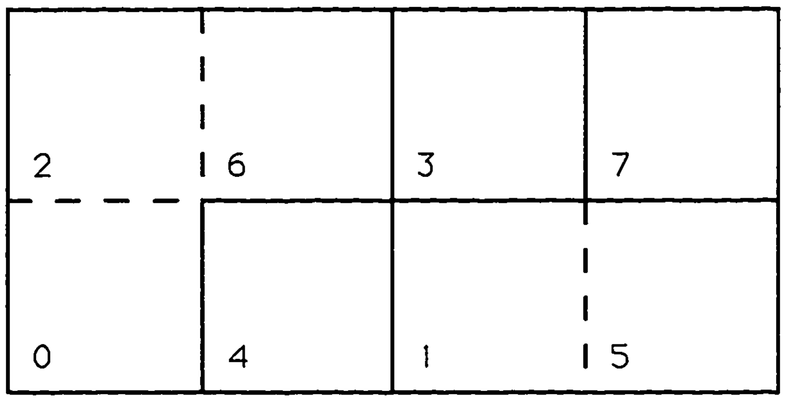


Figure 3.4 : Partitioning the search space into regions. Dotted lines indicate nonexistent boundries.

Let us assume that the directory bucket capacity b_1 = 3. Figure 3.4 shows the directory for the data file illustrated in Figure 3.2.b.The directory partition numbers are indicated at the bottom left corner. Let the directory constructed in Figure 3.4 be labelled as level 1 index. Suppose we want to insert a new record which maps to partition 0 of the data file. First, the level 1 index must be accessed to determine the bucket address for data file partition 0 . Because of the methodology utilized, this bucket address will be directly found in the level index partition 0 . Finally the new record can be inserted. The methodology used to build the directory systematically builds a hierarchy of directories as illustrated in Figure 3.5. For example if the partition numbers in the level 1 directory are not consecutive, a second level might be necessary. It would consist of storing records $k^2 = (k^2_0, k^2_1, ..., k^2_{d-1})$ where each k^2_i is obtained from k^1_i by truncation. The length of k^2_i is determined by the interval partition level $l_{R,i}$ of the first level directory. Each additional level will incur one more access. Fortunately, the number of records must be extremely high to require a second level. Moreover if the number of partitions in this second level directory is small enough, then the bucket containing them can be kept in the main memory.

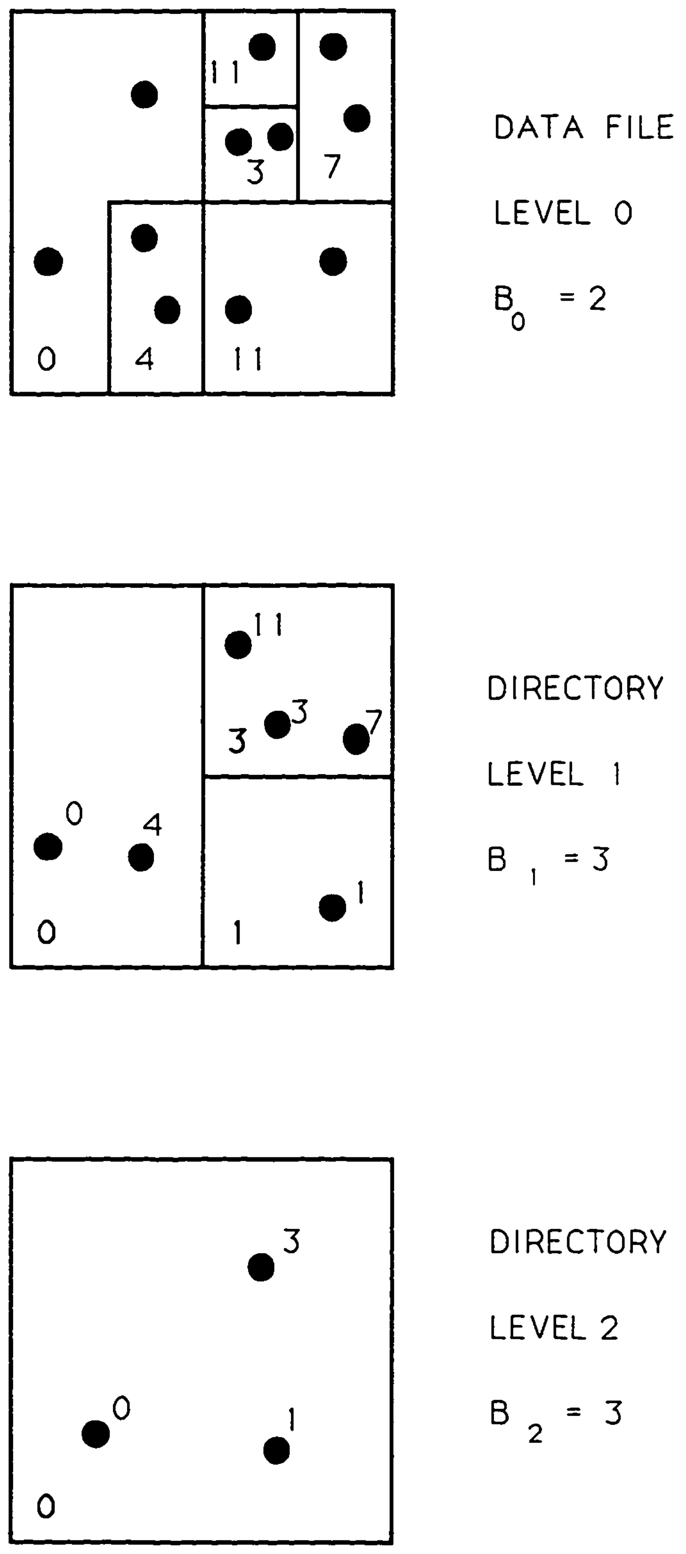


Figure 3.5 : IBGF with 2 directory levels

CHAPTER 4

HARDWARE MODEL

Based on the reasons that will be presented in Chapters 5 and 6, the network selected for this research was a mesh of $m \times n$ processors, where m and n are powers of 2. Each processor has a local CPU, memory, 4 dedicated communication processors, and 4 buffers one for each communication processor. There is a dedicated link between neighbouring processors. Each processor is labeled as P_{ij} where $1 \leq i \leq m$ and $1 \leq j \leq n$, i & j for the row and the column numbers of the processor in the mesh. A processor P_{ij} is called a boundary processor if either i or j is equal to 1 or $i=m$ or $j=n$ otherwise it is a nonboundary processor. Any nonboundary processor in the mesh has 4 neighbours, while boundary processors all have 3 neighbours except $P_{1,1}$, $P_{1,n}$, P_{mn} and $P_{m,1}$ with only 2 neighbours each. There are S storage media, where S is any positive integer, connected to processors $P_{i,1}$ for $1 \leq i \leq m$ and $P_{1,j}$ for $1 \leq j \leq n$ via a crossbar (Figure 4.1). Since the number of connections, B, is always less than the number of processors directly connected to it more than one processor shares the same connection. The following equation shows to which connection each of the processors are connected.
(Note: All operations are integer operations.)

Let each connection be labeled by b_k for $1 \leq k \leq B$. Let $r = \frac{m}{B}$ and $q = \frac{n}{B}$

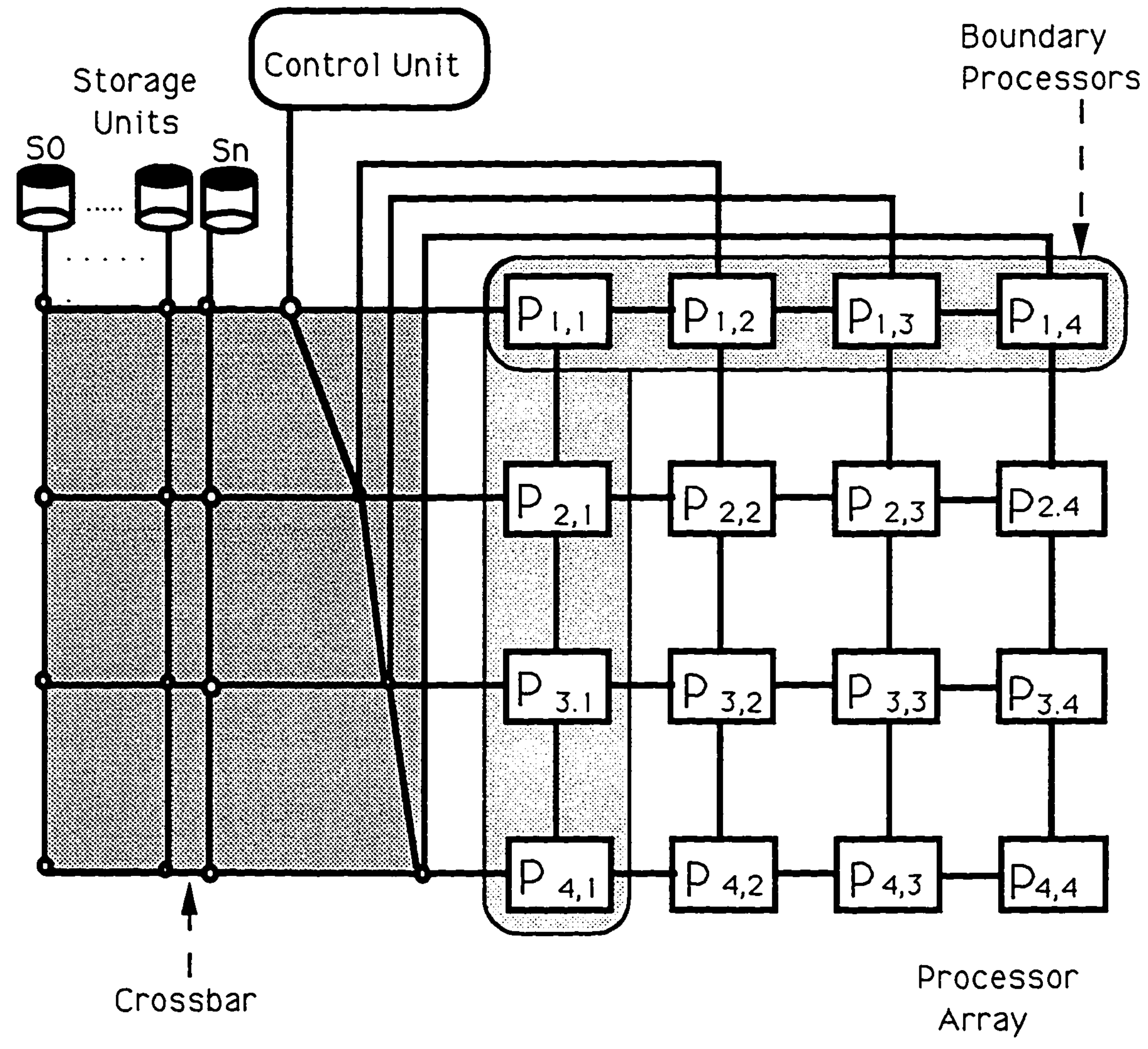


Figure 4.1 : Mesh of 4 by 4 .

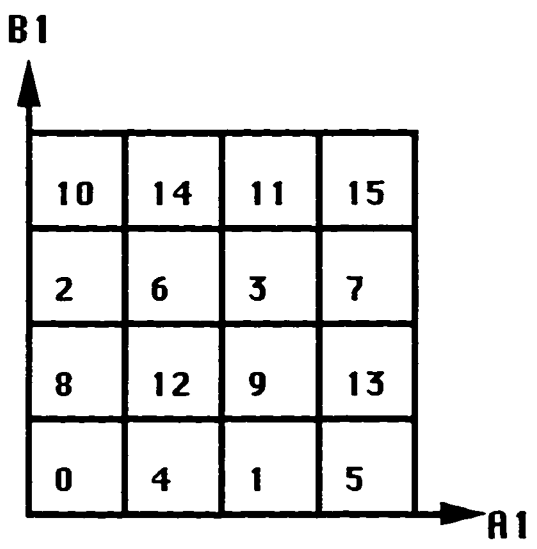
Therefore $P_{i,1}$ is connected to b_l where $l = \frac{i+r-1}{r}$ and $P_{1,j}$ is connected to b_w where $w = \frac{j+q-1}{q}$. There are B switches one for each connection, which permits a single processor to use a connection a time. The switches are controlled by a control unit. The control unit receives the request of each processor requesting access through the connection to the storage media and if there are k processor requesting for the same connection at the same time the control unit turn off the switch from the k-1 processors and it will turn on for 1. The selection of one out of the k requesting processors is done according to predefined priority scheme. In this thesis the priority scheme used is first come first serve.

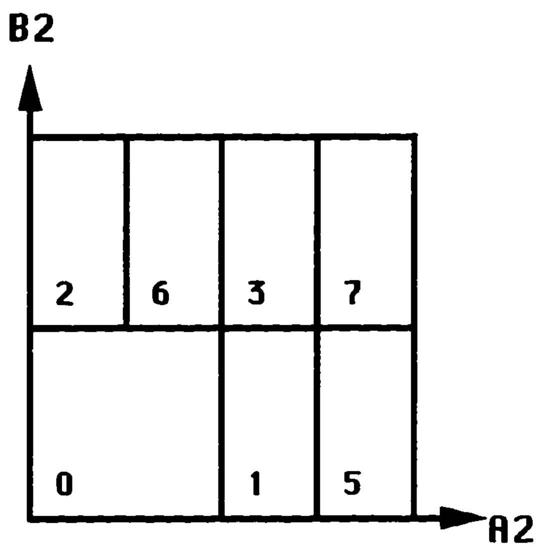
CHAPTER 5

JOIN ALGORITHM FOR RELATIONS WITH UNIFORM DATA DISTRIBUTION

The only parameter known about the physical structure of a given relation, before actually accessing the data is its partition level. In particular it is not known whether the record distribution is uniform or not. The preprocessing of relational queries using the IBGF cannot be done on any other assumption beside the partition level. But if an extra parameter, data partition counter, was added to the IBGF parameters during the insertion of tuples, then this parameter together with the partition level can be used to determine whether the records are uniformly distributed or not.

Suppose relation R is one of the relations assumed by a query which is going to be processed. Let l_R and dc_R be the partition level and the data partition counter of relation R respectively. Given l_R and dc_R, if the logical expression $dc_R = 2^{l_R}$ is true, the records in relation R are uniformly distributed. Thus partitions $0,1,...,2^{l_R}-1$ physically exist in the data search space of R. But if the above logical expression is false, the records in R are not uniformly distributed, so some of the $0,1,...,2^{l_R}-1$ partition are not physically present in the data search space of R. Example 5.1 uses relations R1 and R2 of Figure 5.1 to demonstrate how the partition level and the data partition counter can determine relations of uniform and nonuniform data record distribution.





RELATION (R1) RELATION (R2)

FIGURE 5.1 : R1 is uniform, R2 is non-uniform

Example 5.1

$l_{R1} = 3.$

$l_{R2} = 3.$

$dc_{R1} = 2^{l_{R1}} = 2^3 = 8.$ Records of R1 are uniformly distributed.

$dc_{R2} < 2^3.$ Records of R2 are not uniformly distributed.

***

In Chapter 3 it was discussed that, given any arbitrary attribute A of an arbitrary relation R, there are $2^{l_{R,A}}$ intervals along the axis corresponding to A. There is a mapping which maps each partition in R to one of the $2^{l_{R,A}}$ intervals. Partitions which map to the same interval form a join-class. Since there are $2^{l_{R,A}}$ intervals there are $2^{l_{R,A}}$ join-classes. Like the axis intervals the join-classes are labeled by integer numbers between 0 and $2^{l_{R,A}} - 1$ inclusive. Let the symbol $C_{R,A}{}^{i}$ be a symbol for the i^{th} join-class when attribute A of relation R is taken as the join-attribute.

The mapping of a partition P of a relation R to a join-class when A is the join attribute is as follows:

Let $b_0 b_1 b_2 ... b_{l_R - 1}$ be the binary representation of P, d be the number of attributes (axes) in R, i be the axis number of attribute A and m be equal to $\frac{(l_R + i + 1)}{d} - 1$, then the join-class of P is: $C^{q}_{R,A}$, where

$$q = \sum_{j=0}^{m}(b_{i+d\cdot j} \times 2^{m-j}) \qquad (5.1)$$

Since the records of R are uniformly distributed each partition will fall into a single class and all the classes will have the same number of partitions. Since there are $2^{l_{R,A}}$ join-classes, the number of partitions in each class is $2^{l_R - l_{R,A}}$. Let the symbol $C_{R,A}^{i,k}$ be the k^{th} partition of $C_{R,A}^{i}$. Example 5.2 shows the partition number of R3 (Figure 5.2) and the join-classes when A3 and B3 are assumed as the join attributes. The example also shows the axis partition levels of A3 and B3.

Example 5.2

$l_{R3,A3} = 2.$

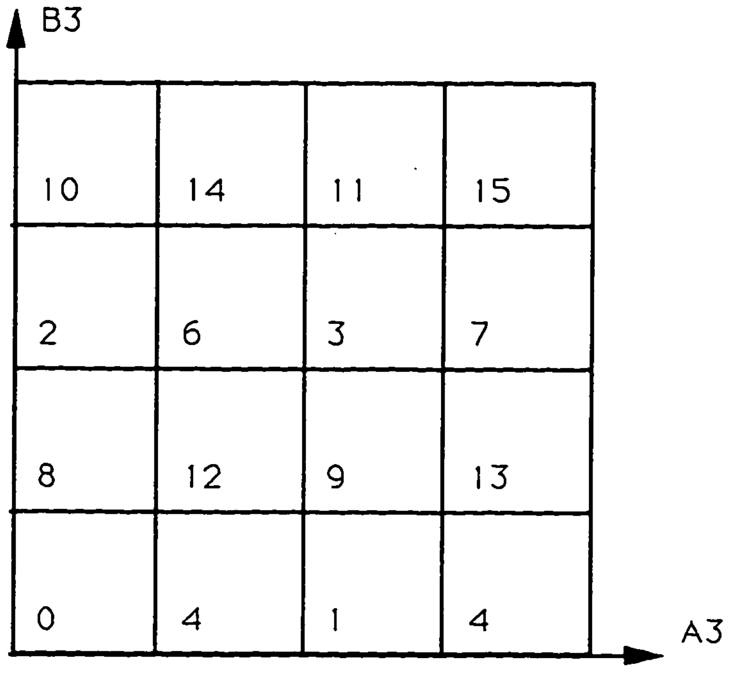
$l_{R3,B3} = 2.$

$l_{R3} = l_{R3,A3} + l_{R3,B3} = 4.$

The join-classes in R3 are:

$C^0_{R3,A3} = [0,8,2,10]$	$C^1_{R3,A3} = [4,12,6,14]$	$C^2_{R3,A3} = [1,9,3,11]$
$C^3_{R3,A3} = [5,13,7,15]$	$C^0_{R3,B3} = [0,4,1,5]$	$C^1_{R3,B3} = [8,12,9,13]$
$C^2_{R3,B3} = [2,6,3,7]$	$C^3_{R3,B3} = [10,14,11,15]$	

***



RELATION (R3)

Figure 5.2 : Uniform relation with partition level of 4. Axis patrition level of A3 is 2 and B3 is 2

5.1 Join of relations with equal number of join-axis intervals

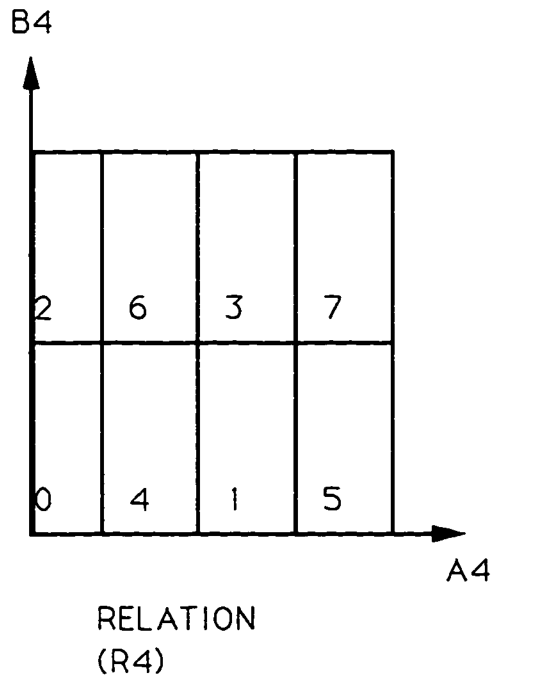
Let R and S be any 2 relations assumed by a join query. Assume that records of R and S are uniformly distributed and attributes A and B are the two join attributes in R and S respectively. The join between R and S on attributes A and B is represented as follows:

$$R \Xi_{A-B} S = C^{i}_{R,A} \Xi C^{j}_{S,B} \quad \text{for } 0 \le i,j \le 2^{L_{R,A}} - 1 \text{ and for } i=j. \qquad \textbf{(5.2)}$$

$C^{i}_{R,A}$ and $C^{j}_{S,B}$ are join compatible if $i = j$. In other words each class from R has one join-compatible class from S.

5.1.1 Join in a single processor

Let R and S be assumed by a join query to be joint on their attributes A and B. Since A and B are of equal axis intervals, $C^{0}_{R,A}$ will have to be joint with $C^{0}_{S,B}$ in P, Where P is a processor. The processing of $C^{0}_{S,B}$ and $C^{0}_{R,A}$ will start by the read of $C^{0,1}_{R,A}$ followed by the read of $C^{0,1}_{S,B}$. These two partitions are then processed to join. The join of these two partitions is then followed by the read of $C^{0,2}_{S,B}$. $C^{0,2}_{S,B}$ is then processed to join with $C^{0,1}_{R,A}$. This join again is followed by the read of $C^{0,3}_{S,B}$, which is processed to join with $C^{0,1}_{R,A}$. P will repeat this process of read and join with the rest of the $C^{0}_{S,B}$ partitions, $C^{0,4}_{S,B}$, $C^{0,5}_{S,B}$, ..., $C^{0,q}_{S,B}$ where $q = 2^{l_S - l_{S,B}}$. Since $C^{0,1}_{R,A}$ has been processed with all the partitions of $C^{0}_{S,B}$, it is no longer needed in P. It is replaced by $C^{0,2}_{R,A}$. $C^{0,2}_{R,A}$ again joins with all the partitions of $C^{0}_{S,B}$, which by now they are all in P. $C^{0,3}_{R,A}$



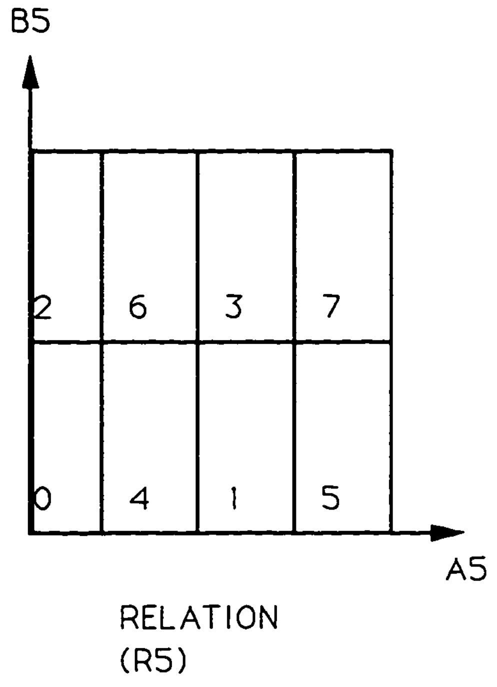


Figure 5.3 : Uniform relations

will then replace $C^{0,2}_{R,A}$ to be processed in the same way as $C^{0,1}_{R,A}$ and $C^{0,2}_{R,A}$. When all the $2^{l_R - l_{R,A}}$ partitions of $C^0_{R,A}$ are processed with all the partitions of $C^0_{S,B}$ the join of these two classes is complete. But For the complete join of R and S, all the classes in R and their join-compatible-classes in S must be processed the same way as $C^0_{R,A}$ and $C^0_{S,B}$. Algorithm 5.1 is an algorithm for a join of two relation with uniformly distributed data records. The join is done in a single processor. *Read* is a procedure which reads from a secondary storage the partition given as its parameter. *Join* is another procedure of two arguments. These two arguments are partitions. It performs the join operation between them. Example 5.3 uses R4 and R5 of Figure 5.3 to show how the join of R4 with R5 is conducted in a single processor.

Example 5.3

Let the join attributes of R4 and R5 be A4 and A5 respectively.

The partition levels are:

$l_{R4} = 3$	$l_{R5} = 3$
$l_{R4,A4} = 2$	$l_{R5,A5} = 2$

The join-classes are:

$C^0_{R4,A4} = [0,2]$	$C^0_{R5,A5} = [0,2]$
$C^1_{R4,A4} = [4,6]$	$C^1_{R5,A5} = [4,6]$
$C^2_{R4,A4} = [1,3]$	$C^2_{R5,A5} = [1,3]$

```
FOR $i = 0$ to $2^{l_{R,A}} - 1$ DO
BEGIN
  FOR $k = 1$ to $2^{l_R - l_{R,A}}$ DO
  BEGIN
    read( $C_{R,A}{}^{i,k}$ )
    FOR $j = 1$ to $2^{l_S - l_{S,B}}$ DO
    BEGIN
    IF ( $k = 1$ ) THEN
      BEGIN
        read( $C_{S,B}{}^{i,j}$ )
      END
      join( $C_{R,A}{}^{i,k} C_{S,B}{}^{i,j}$ )
    END
  END
END
```

Algorithm 5.1 : Join algorithm for a single processor.

$C^{3}_{R4,A4} = [5,7]$ $\qquad$ $C^{3}_{R5,A5} = [5,7]$

The compatible-classes are:

$C^{0}_{R4,A4}$ & $C^{0}_{R5,A5}$, $C^{1}_{R4,A4}$ & $C^{1}_{R5,A5}$, $C^{2}_{R4,A4}$ & $C^{2}_{R5,A5}$ and $C^{3}_{R4,A4}$ & $C^{3}_{R5,A5}$.

The join of R4 and R5 proceeds as follows:

P, the processor, reads $R4_0$ followed by the read of $R5_0$. These two partitions are then joint. Next $R5_2$ is read and is joined with $R4_0$. The join of $R4_0$ and $R5_2$ is followed by the read of $R4_2$. $R4_2$ is then joint with $R5_0$ and $R5_2$. This ends the join of $C^{0}_{R4,A4}$ with $C^{0}_{R5,A5}$. The processing of $C^{1}_{R4,A4}$ and $C^{1}_{R5,A5}$ starts with the reading of $R4_4$ followed by the reading of $R5_4$. After these two partitions join $R5_6$ is read to join with $R4_4$. Next $R4_6$ is read and is joint with both $R5_4$ and $R5_6$ ending the join of $C^{1}_{R4,A4}$ and $C^{1}_{R5,A5}$. The join of $C^{2}_{R4,A4}$ and $C^{2}_{R5,A5}$ starts with the reading of $R4_1$ followed by the reading of $R5_1$. After their join $R5_3$ is read to join with $R4_1$. The join of $C^{2}_{R4,A4}$ with $C^{2}_{R5,A5}$ is complete with the reading of $R4_3$ and its join with $R5_1$ and $R5_3$. At last the join of $C^{3}_{R4,A4}$ with $C^{3}_{R5,A5}$ starts with the reading of $R4_5$ followed by the reading of $R5_5$. The join of $R4_5$ with $R5_5$ is followed by the reading of $R5_7$. The reading of of the last partition $R4_7$ follows the join of $R4_5$ with $R5_7$. The join of $R4_7$ with both $R5_5$ and $R5_7$ marks the end of the join of R4 with R5 .

***

5.1.2 Join in vector of processors

To shorten the time taken to process the join of R and S a vector of n processors can be employed. Each processor in the vector is labeled P_i for $0 \le i \le n$-1. In processing the join of R and S the vector starts by the partitions of $C^0{}_{R,A}$ and $C^0{}_{S,B}$. Each P_i reads a partition from $C^0{}_{S,B}$. P_0 again reads partition $C^{0,1}{}_{R,A}$ from $C^0{}_{R,A}$. $C^{0,1}{}_{R,A}$ will then be pipelined to P_2, P_3, ... , $P_{k(1)}$, where $1 \le k(1) < n$. Each P_i will then perform the join of $C^{0,1}{}_{R,A}$ with the partition of $C^0{}_{S,B}$ that it is holding. Since each P_i is having one partition of $C^0{}_{S,B}$, $C^{0,1}{}_{R,A}$ by now has been joint with k(1) partitions of $C^0{}_{S,B}$. If there are still more unprocessed partitions of $C^0{}_{S,B}$ each P_i, for $1 \le i < k(2)$, where $1 \le k(2) < n$, will again read another partition form $C^0{}_{S,B}$. These new $k(2)$ partitions will then be joint with $C^{0,1}{}_{R,A}$ which is already in the vector. This process of read and join will continue till all the partitions of $C^0{}_{S,B}$ are joint with $C^{0,1}{}_{R,A}$. By now $C^{0,1}{}_{R,A}$ will be through joining with all the partitions it was supposed to join. P_0 now reads the second partition in $C^0{}_{R,A}$, which is $C^{0,2}{}_{R,A}$. $C^{0,2}{}_{R,A}$ is then pipelined to P_2, P_3, ... , $P_{k,}$ where k the highest labeled processor having at least a partition of $C^0{}_{S,B}$. $C^{0,2}{}_{R,A}$ is directly joint with all the partitions of $C^0{}_{S,B}$, which by now all of them are in the vector. $C^{0,2}{}_{R,A}$ is then replaced by $C^{0,3}{}_{R,A}$ which again goes through the same process as $C^{0,2}{}_{R,A}$ & $C^{0,2}{}_{S,B}$. In this way all the $2^{l_R - l_{R,A}}$ partitions of $C^0{}_{R,A}$ are joint with the $2^{l_S - l_{S,B}}$ partitions of $C^0{}_{S,B}$. When the join of these two classes is over, the join of $C^1{}_{R,A}$ with $C^1{}_{S,B}$ follows . The complete join of R and S is done when each $C^i_{R,A}$ is joint with each $C^i_{S,B}$ for $1 \le i \le 2^{l_{R,A}}$. Algorithm 5.2 is an algorithm for a join operation done in a vector of processors. S and R in the algorithm are two arbitrary relations to be joint on their join attributes, A of R and B of S. Let n be the number of processors in the vector.

In Algorithm 5.2, *accept* is a procedure which receives an R partition from a neighbouring processor. In case of a vector of processors *accept* procedure running in P_i will receive an R partition from P_{i-1}. A *send* procedure in P_i will send an R partition to processor P_{i+1}. Each R partition is read by processor, P_1, and is pipelined to processor P_2 and then to processor P_3 till it reaches the last processor in the vector. As for the partitions of S, in each join-class, are divided into equal chunks among the processors of the vector and each one of the processors reads and processes its chunk. Example 5.4 uses a vector of 2 processors of Figure 5.4 to join R4 with R5 of Figure 5.3. The partition levels and the join-classes are all discussed in Example 5.3.

Example 5.4

The join of R4 and R5 (Figure 5.3) in a vector of 2 processors proceeds as follows:

P_0 read $R5_0$ and P_1 read $R5_2$. P_0 again reads $R4_0$ which is pipelined to P_1. P_0 then performs the join of $R4_0$ with $R5_0$ while P_1 performs the join of $R4_0$ with $R5_2$. Next P_0 read $R4_2$ and pipelines it to P_1. The join of $R4_2$ with both $R5_0$ and $R5_2$ takes place in P_0 and P_1 respectively. This ends the join of $C^0_{R4,A4}$ with

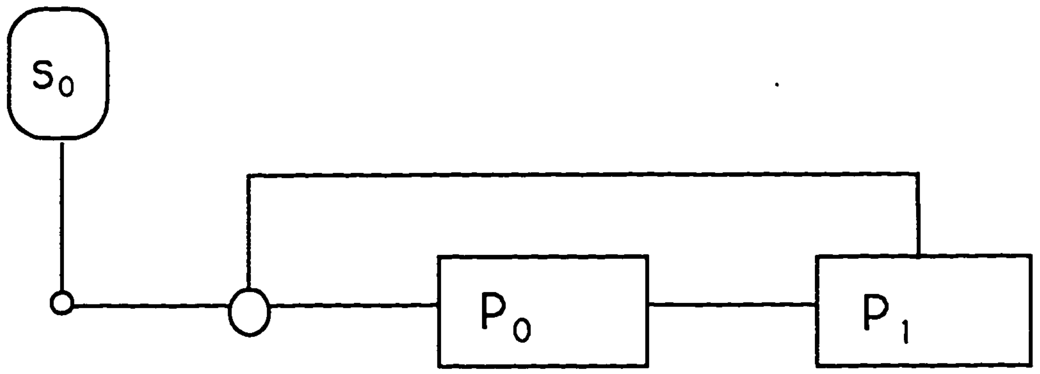


Figure 5.4 : A vector of 2 processors

```
* At processor $P_i$  FOR $i = 0$ TO $n$-1

    $m = \frac{2^{l_S - l_{S,B}}}{n}$
    FOR $t = 0$ to $2^{l_{R,A}} - 1$ DO
    BEGIN
      FOR $k = 1$ to $2^{l_R - l_{R,A}}$ DO
      BEGIN
          IF ( $i <> 1$) then
          BEGIN
            accept( $C^{t,k}_{R,A}$)
          ELSE
            read( $C^{t,k}_{R,A}$)
          END
          IF ( $i < n$) then
          BEGIN
            send( $C^{t,k}_{R,A}$)
          END
          FOR $j = (i-1)$ times $m$ to $i$ times $m$ DO
          BEGIN
            IF ( $k = 1$ ) THEN
            BEGIN
              read( $C_{S,B}^{t,j}$)
            END
            join( $C_{R,A}^{t,k}$ , $C_{S,B}^{t,j}$)
        END
      END
    END
```

Algorithm 5.2 : Join algorithm for a vector n of processors.

$C^0_{R5,A5}$. The processing of $C^1_{R4,A4}$ and $C^1_{R5,A5}$ starts with the reading of P_0 to $R5_4$ and P_1 to $R5_6$. P_0 again reads $R4_4$ and pipelines it to P_1. The join of $R4_4$ with $R5_4$ takes place in P_0 and the join of $R4_4$ with $R5_6$ takes place in P_1. The join of $R4_4$ with $R5_4$ in P_0 is followed by the read of $R4_6$ by P_0. This page is then pipelined to P_1. $R4_6$ then joins with $R4_4$ in P_0 and with $R5_6$ in P_1. This ends the processing of $C^1_{R4,A4}$ and $C^1_{R5,A5}$. and starts the processing of $C^2_{R4,A4}$ and $C^2_{R5,A5}$ with the reading of $R5_1$ by P_0 and the reading of $R5_3$ by P_1. P_0 again reads $R4_1$ which it pipelines it to P_1. The join of $R5_1$ with $R4_1$ takes place in P_0 while the join of $R4_1$ with $R5_3$ takes place in P_1. The last partition in $C^2_{R4,A4}$, $R4_3$, is read by P_0 and it is pipelined to P_1. It is then joint with $R4_1$ in P_0 and with $R4_3$ in P_1. The processing of the last two join compatible-classes, $C^3_{R4,A4}$ and $C^3_{R5,A5}$, starts with the reading of $R5_5$ by P_0 and $R5_7$ by P_1. Again P_0 reads $R4_5$ and pipelines it to P_1. The join of $R5_5$ with $R4_5$ takes place in P_0 while the join of $R5_7$ with $R4_5$ takes place in P_1. The last remaining partition, $R4_7$ is read next by P_0 and is pipelined to P_1. The join of $R5_5$ with $R4_7$ takes place in P_0 while the join of $R5_7$ with $R4_7$ takes place in P_1. This ends the processing of $C^3_{R4,A4}$ and $C^3_{R5,A5}$ ending the join of R4 with R5.

***

5.1.3 Join in a mesh

When a vector of n processors was used to join R and S, only the partitions of R were pipelined. If a mesh of n by m is employed to perform the join, both the partitions of R and S are pipelined. In the vector the reading of the R partitions was sequential since P_0 was the only processor employed in reading the partitions of R. In a mesh, processors $P_{i,1}$ for $1 \leq i \leq m$ can simultaneously read partitions from R. The join of R and S in a mesh starts by each $P_{i,1}$ reading a partition from $C^0_{R,A}$ and each $P_{1,j}$ for $1 \leq j \leq n$ reading a partition from $C^0_{S,B}$. Each $P_{i,1}$ for $1 \leq i \leq m$ then pipelines the partition it reads from $C^0_{R,A}$, to all the processors in row i, while each $P_{1,j}$ pipelines the partition it reads from $C^0_{S,B}$, to all the processors in column j of the mesh. So a $C^0_{R,A}$ partition read by $P_{i,1}$ and a $C^0_{S,B}$ partition read by $P_{1,j}$ will only meet and join in processor P_{ij}. Each $P_{1,j}$ when it is done processing the join of the two partitions it is holding, it reads another partition of $C^0_{S,B}$ which has never been read before. This page is pipelined to all the processors in the same column as that of the processor which read it. This new partition is then joint to the same $C^0_{R,A}$ partition in each processor it is pipelined. Each $P_{1,j}$ repeats this process till it joins the $\frac{2^{l_S - l_{S,B}}}{n}$ partitions of $C^0_{S,B}$ with the partition of R already held by it. The first set of $C^0_{R,A}$ partitions which was read before is now replaced by another set of $C^0_{R,A}$ partitions. This set is also processes in the same way as the previous sets. The later set will be processed faster because the partitions of $C^0_{S,B}$

with which they have to join are already in the mesh. The second set will be followed by the third one and so on till join of $C^0_{R,A}$ with $C^0_{S,B}$ is complete. When $C^0_{R,A}$ and $C^0_{S,B}$ are done they will be replaced by $C^1_{R,A}$ and $C^1_{S,B}$ in the mesh. Again these two classes will be replaced by another two classes and so on till the join of R and S is complete. Example 5.5 shows how the join of two relations can be carried out in a mesh.

Example 5.5

In This example a mesh of 2 by 2 is used to perform the join of R4 and R5 of Figure 5.3. The partition levels and the join-classes of these two relations has been discussed in Example 5.3. The join of R4 with R5 in the mesh is as follows:

The join process starts with the reading of, $R4_0$ by $P_{1,1}$, $R4_2$ by $P_{2,1}$ and $R5_2$ by $P_{1,2}$. $P_{1,1}$ again reads $R5_0$. Then $P_{2,2}$ receives $R4_2$ from $P_{2,1}$ and $R5_2$ from $P_{1,2}$. $P_{1,1}$ sends $R4_0$ to $P_{1,2}$ and $R5_0$ to $P_{2,1}$. The join of, $R4_0$ with $R5_0$ takes place in $P_{1,1}$, $R4_0$ with $R5_2$ takes place in $P_{1,2}$, $R4_2$ with $R5_0$ takes place in $P_{2,1}$ and $R4_2$ with $R5_2$ takes place in $P_{2,2}$ ending the join of $C^0_{R4,A4}$ with $C^0_{R5,A5}$. The join of $C^1_{R4,A4}$ with $C^1_{R5,A5}$ starts with the reading of, $R4_4$ by $P_{1,1}$, $R4_6$ by $P_{2,1}$ and $R5_6$ by $P_{1,2}$. $P_{1,1}$ again reads $R5_4$. Then $P_{2,2}$ receives $R4_6$ from $P_{2,1}$ and $R5_6$ from $P_{1,2}$. $P_{1,1}$ sends $R4_4$ to $P_{1,2}$ and $R5_4$ to $P_{2,1}$. The join of, $R4_4$ with $R5_4$ takes place in $P_{1,1}$, $R4_4$ with $R5_6$ takes place in $P_{1,2}$, $R4_6$ with $R5_4$ takes place in $P_{2,1}$ and $R4_6$ with $R5_6$ takes place in $P_{2,2}$ ending the join of $C^1_{R4,A4}$ with $C^1_{R5,A5}$. The join of $C^2_{R4,A4}$

with $C^2_{R5,A5}$ starts with the reading of, $R4_1$ by $P_{1,1}$, $R4_3$ by $P_{2,1}$ and $R5_3$ by $P_{1,2}$. $P_{1,1}$ again reads $R5_1$. Then $P_{2,2}$ receives $R4_3$ from $P_{2,1}$ and $R5_3$ from $P_{1,2}$. $P_{1,1}$ sends $R4_1$ to $P_{1,2}$ and $R5_1$ to $P_{2,1}$. The join of, $R4_1$ with $R5_1$ takes place in $P_{1,1}$, $R4_1$ with $R5_3$ takes place in $P_{1,2}$, $R4_3$ with $R5_1$ takes place in $P_{2,1}$ and $R4_3$ with $R5_3$ takes place in $P_{2,2}$ ending the join of $C^2_{R4,A4}$ with $C^2_{R5,A5}$. The join of the last two join-compatible classes namely $C^3_{R4,A4}$ and $C^3_{R5,A5}$ starts with the reading of, $R4_5$ by $P_{1,1}$, $R4_7$ by $P_{2,1}$ and $R5_7$ by $P_{1,2}$. $P_{1,1}$ again reads $R5_5$. Then $P_{2,2}$ receives $R4_7$ from $P_{2,1}$ and $R5_7$ from $P_{1,2}$. $P_{1,1}$ sends $R4_5$ to $P_{1,2}$ and $R5_5$ to $P_{2,1}$. The join of, $R4_5$ with $R5_5$ takes place in $P_{1,1}$, $R4_5$ with $R5_7$ takes place in $P_{1,2}$, $R4_7$ with $R5_5$ takes place in $P_{2,1}$ and $R4_7$ with $R5_7$ takes place in $P_{2,2}$ ending the join of R4 and R5.

***

5.2 Join of relations with unequal join-axis intervals

If the join axis-intervals of R are less than the join axis-intervals of S, each join-class in R will have to be joint to $2^{l_{S,B}-l_{R,A}}$ join-classes from S. Let $AC^i_{S,B}$ be a symbol representing all the $2^{l_{S,B}-l_{R,A}}$ join compatible-classes of $C^i_{R,A}$ when R and S are joint on attributes A and B. So $AC^i_{S,B}$ is an aggregate-class of $2^{l_{S,B}-l_{R,A}}$ join-classes of S. The number of partitions in $AC^i_{S,B}$ are equal to $2^{l_S-l_{R,A}}$. The classes forming $AC^i_{S,B}$ are:

$C^j_{S,B}$ for $j = (i\text{-}1)q,\ (i\text{-}1)(q+1),\ \ (i\text{-}1)(q+2),\ \ldots.,\ i(q\text{-}1)$ where $q = 2^{l_{S,B}-l_{R,A}}$. Equation 5.2 can now be modified to :

$$R\,\Xi_{A-B}\,S = C^i_{R,A}\,\Xi\,AC^i_{S,B} \quad \text{for } 0 \le i \le 2^{L_{R,A}}-1 \tag{5.3}$$

Equation 5.3 implies that one join-class from R has only one join-compatible aggregate-class form S. The join algorithms which were introduced in the previous section can be used to join relations with unequal number of join axis intervals by simply replacing a group of $C^j_{S,B}s$ by their corresponding $AC^i_{S,B}$ in the algorithms. Example 5.6 demonstrates how two relations of unequal number of join-axis intervals can be joined.

Example 5.6

This example uses a 2 by 2 mesh in the join of relation R6 with relation R7 (Figure 5.5). The join attribute of R6, A6, has less partition intervals than the join attribute of R7, A7. The following are the partition levels of each relation and each join-attribute :

$l_{R6} = 2$	$l_{R7} = 3$
$l_{R6,A6} = 1$	$l_{R7,A7} = 2$

The join-classes are as follows:

$C^0_{R6,A6} = [0,2]$	$C^1_{R6,A6} = [1,3]$
$C^0_{R7,A7} = [0,2]$	$C^1_{R7,A7} = [4,6]$
$C^2_{R7,A7} = [1,3]$	$C^3_{R7,A7} = [5,7]$

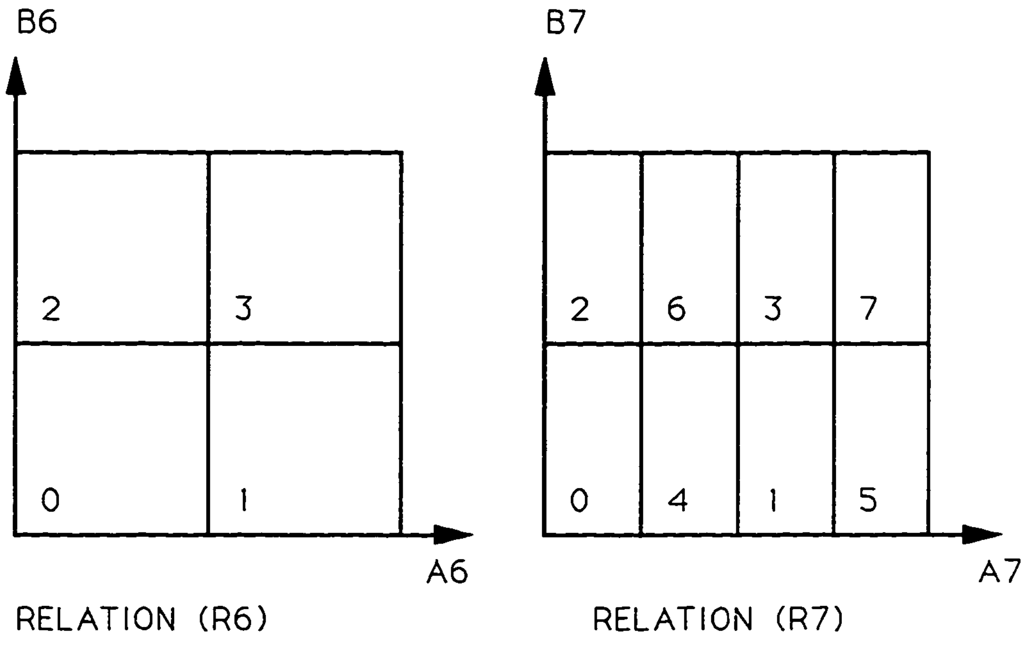


Figure 5.5 : Uniform relation of different axis partition levels

The number of join-classes in each aggregate-class are $2^{l_{R7,A7}-l_{R6,A6}} = 2$.

The aggregate-classes are as follows:

$$AC^0_{R7,A7} = C^0_{R7,A7} \text{ and } C^1_{R7,A7} = [0, 2, 4, 6]$$
$$AC^1_{R7,A7} = C^2_{R7,A7} \text{ and } C^3_{R7,A7} = [1, 3, 5, 7]$$

The join process is as follows:

The join of R6 and R7 starts with the join of $C^0_{R6,A6}$ and $AC^0_{R7,A7}$. The process starts with the reading of, $R6_0$ by $P_{1,1}$, $R6_2$ by $P_{2,1}$ and $R7_2$ by $P_{1,2}$. $P_{1,1}$ again reads $R7_0$. $P_{2,2}$ receives $R6_2$ from $P_{2,1}$ and $R7_2$ from $P_{1,2}$. $P_{1,1}$ sends $R6_0$ to $P_{1,2}$ and $R7_0$ to $P_{2,1}$. Next the join of, $R6_0$ with $R7_0$ takes place in $P_{1,1}$, $R6_0$ with $R7_2$ takes place in $P_{1,2}$, $R6_2$ with $R7_0$ takes place in $P_{2,1}$, $R6_2$ with $R7_2$ takes place in $P_{2,2}$. Since there are still 2 more partition unprocessed partitions in $AC^0_{R7,A7}$, $P_{1,1}$ reads one of them, $R7_4$ and $P_{1,2}$ reads the second one, $R7_6$. $P_{1,1}$ sends $R7_4$ to $P_{2,1}$ and $P_{1,2}$ sends $R7_6$ to $P_{2,2}$. Since $R6_0$ and $R6_2$ are already in the mesh The join of, $R6_0$ with $R7_4$ in $P_{1,1}$, $R6_0$ with $R7_6$ in $P_{1,2}$, $R6_2$ with $R7_4$ in $P_{2,1}$, and $R6_2$ with $R7_6$ in $P_{2,2}$, takes place as soon as $R7_4$ and $R7_6$ are available in the processor. This end the join of $C^0_{R6,A6}$ with $AC^0_{R7,A7}$. Next the processing of $C^1_{R6,A6}$ and $AC^1_{R7,A7}$ starts with the reading of, $R6_1$ by $P_{1,1}$, $R6_3$ by $P_{2,1}$ and $R7_3$ by $P_{1,2}$. $P_{1,1}$ again reads $R7_1$. $P_{2,2}$ receives $R6_3$ from $P_{2,1}$ and $R7_3$ from $P_{1,2}$. $P_{1,1}$ sends $R6_1$ to $P_{1,2}$ and $R7_1$ to $P_{2,1}$. Next the join of, $R6_1$ with $R7_1$ takes place in $P_{1,1}$, $R6_1$ with $R7_3$ takes place in $P_{1,2}$, $R6_3$ with $R7_1$ takes place in $P_{2,1}$, $R6_3$ with

$R7_3$ takes place in $P_{2,2}$. Since there are still two more unprocessed partitions in $AC^1_{R7,A7}$, $P_{1,1}$ reads one of them, $R7_5$ and $P_{1,2}$ reads the second one, $R7_7$. $P_{1,1}$ sends $R7_5$ to $P_{2,1}$ and $P_{1,2}$ sends $R7_7$ to $P_{2,2}$. Since $R6_1$ and $R6_3$ are already in the mesh The join of, $R6_1$ with $R7_5$ in $P_{1,1}$, $R6_1$ with $R7_7$ in $P_{1,2}$, $R6_3$ with $R7_5$ in $P_{2,1}$, and $R6_1$ with $R7_7$ in $P_{2,2}$, takes place as soon as $R7_5$ and $R7_7$ are available in the processor. This end the join of R6 with R7.

***

5.3 Load balancing

For a better efficiency equal number of joining partitions must be assigned to each processor in the mesh. This can be achieved by equally dividing partitions in the same aggregate-class to the processors in the first row or the first column of the mesh. To be able to do this each processor must be able to know which aggregate-class is currently in use and how many partitions of each aggregate-class it should assume and process. The number of partitions in an aggregate-class of relation R taking A as the join-attribute is $2^{l_R - l_{R,A}}$. If the number of processors in one column of the mesh is m then each processor will process $\frac{2^{l_R - l_{R,A}}}{m}$ partitions from each join-class. A one to one mapping between $0 \ldots 2^{l_R - l_{R,A}} - 1$ and partition numbers of each class will help to assign $\frac{2^{l_R - l_{R,A}}}{m}$ partitions of each class to each of the m processors. The mapping is as follows:

Let d be the number of axes in R, i be the axis number of A, k be equal to $\frac{(l_R+i-1)}{d} - 1$, p be a partition number in R and $b_{l_R-1}b_{l_R-2}...b_1b_0$ be the binary representation of p, where b_0 is the least significant bit. Then the mapping of p written in binary is:

$$b_{l_R-1}b_{l_R-2} \ \cdots \ b_{l_R-(d-1)}b_{l_R-(d+1)}b_{l_R-(d+2)} \ \cdots \ b_{l_R-(k*d-1)}b_{l_R-(k*d+1)}b_{l_R-(k*d+2)} \ \cdots \ b_1b_0$$

In other words the mapping of p was obtained by ignoring bits $b_i, b_{i+d}, b_{i+2*d} \cdots b_{i+k*d}$ (b_{i+j*d} for $0 \leq j \leq k$) from its binary representation and evaluating the decimal equivalence for the rest of the bits. Since the number of the bits in the mapping is $l_R - l_{R,A}$ the decimal number evaluated will be in the range 0 ... $2^{l_R-l_{R,A}} - 1$ inclusive. Processor $P_{i,1}$ for $1 \leq i \leq m$ will be assigned to partitions of a join-class which will map to:-

$(i\text{-}1) \times \frac{q}{m}$, $(i\text{-}1) \times \frac{q}{m} + 1$, $(i\text{-}1) \times \frac{q}{m} + 2$ ... $i \times \frac{q}{m} - 1$ of 0 ... q-1,

where $q = 2^{l_R-l_{R,A}}$. The load balancing is more elaborated by Example 5.7.

Example 5.7

This example uses R3 (Figure 5.2) to clarify load balancing. Each aggregate-class has $2^{l_R-l_{R3,A3}} = 2^{4-2} = 4$ partitions when A3 is considered as the join attribute. The following are the join-classes (Assume A3 as the join attribute).

$C^0_{R3,A3}$ = [0,8,2,10] $C^1_{R3,A3}$ = [4,12,6,14]

$C^2_{R3,A3}$ = [1,9,3,11] $C^3_{R3,A3}$ = [5,13,7,15]

By ignoring bits 3 and 1 and reading the rest of the bits from right to left will map partitions of $C^0_{R3,A3}$ to the following integers.

0000 -----> 00 --> 0

1000 -----> 10 --> 1

0010 -----> 01 --> 2

1010 -----> 11 --> 3

Let the number of processors in one column of the mesh used be 2. Each processor will be assigned $\frac{2^{4-2}}{2}=2$ partitions from each aggregate-class . The partitions that are going to be assigned to $P_{1,1}$ from $C^0_{R3,A3}$ are those partitions which map to the numbers in the range $(1\text{-}1)\times\frac{2^{4-2}}{2}=0$ and $(1)\times\frac{2^{4-2}}{2}-1=1$, and those which are going to be assigned by $P_{2,1}$ from $AC^0_{R3,A3}$ are those which map to the numbers in the range $(2\text{-}1)\times\frac{2^{4-2}}{2}=2$ and $2\times\frac{2^{4-2}}{2}-1=3$. Hence from $C^0_{R3,A3}$, $P_{1,1}$ will be assigned partitions 0 and 8 and $P_{2,1}$ will be assigned partitions 2 and 10.

***

Algorithm 5.3 is a join algorithm for relations R an S carried out in a mesh of *m* by n . Let A and B be 2 join-attributes of R and S respectively.

```
FOR $0 \le i \le m$ AND FOR $0 \le j \le n$ DO
      * With $P_{i,j}$
        $\gamma_R = \beta^0_{R,A}/m$
        $\gamma_S = \beta^0_{S,B}/n$
        FOR $t = 0$ to $2^{l_{R,A}} - 1$ DO
        BEGIN
          FOR $h = (i-1) \times \gamma_R + 1$ TO $i \times \gamma_R$ DO
          BEGIN
            IF ($j = 1$) THEN
              read( $C^{t,h}_{R,A}$)
            ELSE
            BEGIN
              accept( $C^{t,h}_{R,A}$)
            END
           send( $C^{t,h}_{R,A}$)
            FOR $g = (j-1) \times \gamma_S + 1$ TO $j \times \gamma_S$ DO
            BEGIN
              IF ($j = 1$) THEN
              BEGIN
                IF ( $h = (i-1) \times \gamma_R + 1$ ) THEN
                BEGIN
                   read( $AC^{h,g}_{S,B}$)
                   send( $AC^{h,g}_{S,B}$)
                END
                join( $C^{t,h}_{R,A}$, $AC^{h,g}_{S,B}$)
              END
              ELSE
              BEGIN
                IF ( $h = (i-1) \times \gamma_R + 1$ ) THEN
                BEGIN
                  accept( $AC^{h,g}_{S,B}$)
                  IF ( $j < n$ ) send( $AC^{h,g}_{S,B}$)
                END
                join( $C^{t,h}_{R,A}$, $AC^{h,g}_{S,B}$ )
              END
            END
          END
        END
```

Algorithm 5.3 : A join algorithm for nonuniform relations in a mesh of $m \times n$.

CHAPTER 6

JOIN ALGORITHM FOR RELATIONS WITH NONUNIFORM RECORD DISTRIBUTION

When the data partition counter of relation R, dc_R, is less than 2^{l_R} the records of R are not uniformly distributed in the data search space of R. Unlike relations with a uniform data distribution, some data partitions of R will be embedded in others.

In nonuniform case, just from l_R and dc_R it is impossible to know which partitions are physically present in the data search space of R. Unless all the directory records of R are accessed there is no way to know which data pages are physically present in R.

6.1 Join-classes

The idea of a join-class which was introduced in Chapter 5 for relations with uniform record distributions, also works in the join of relations with nonuniformly distributed data records. However, the join-class of a nonuniform relation can be embedded in another join-class, a partition can be a member of more than one join-class and the number of partitions in each join-class can vary. $C^{i}_{R,A}$ embeds $C^{i+1}_{R,A}$, $C^{i+2}_{R,A}$, . . . , $C^{i+n}_{R,A}$ if the partitions of each one of them are identical. Such join-classes exist because a partition can be a member of more than one join-class.

In classifying the partitions of a relation R in to join-classes, assuming A as the join-attribute, the knowledge of l_R and the axis number of A in the schema of R is essential. In representing the label of R_i, where $0 \le i < 2^{l_R}$, as a binary number the minimum number of the binary digits needed is l_R. Let the number of attributes in R be d, the axis number of A in the schema of R be k, the binary representation of i be $b_{l_R-1}, b_{l_R-2}, \ldots, b_0$, the axis partition level of A when R_i was last divided or created be m then the join-class of R_i is $C^q_{R,A}$, where

$$q = \sum_{j=m-1}^{0} 2^j \times b_{k+(m-(j+1)(d)} \qquad \textbf{(6.1)}$$

So in collecting the partitions of a specific join-class from the data search space of a relation, Equation 6.1 is used. Algorithm 6.1 does the collection of partitions of a specific join-class from the data search space of a relation. To collect all the partitions of a join-class from the data search space of any relation R, the processors in the first row or the first column of the mesh selectively read through the directories of R gathering partition numbers satisfying Equation 6.1. The algorithm use the symbol *stk* as a stack variable, *root* as the address of the first directory page of a relation in an IBGF file, *i* as the label of a join-class and its compatible-class currently in use. A logical function *Equal* takes 2 arguments and it returns true if the partition (the first argument corresponding to a join attribute) belongs to the join-class (the second argument) otherwise it returns false. Algorithm 6.1 uses the stack, *stk* , to go through the directories searching for partitions satisfying equation 6.1 for the current join-class label. To join two relations R and S Algorithm 6.1 is used to collect all the data partitions of a join-class *i* of R in array $C^i_{R,A1}$ and that of S which is compatible to $C^{i}_{R,A1}$ in $C^{i}_{S,B1}$. Example 6.1 uses Equation 6.1 in classifying the partitions of R1 into join-classes.

Example 6.1

This example uses R1 (Figure 6.1) to demonstrate the mapping of a partition to a join-class using Equation 6.1.

The minimum number of binary digits needed to represent a partition label in R1 is $l_{R1,A1} = 4$.

The partitions of R1 as written in binary are :

0000, 0010, 1010, 0110, 0001, 0011 .

The number of join-classes at most are $2^{l_{R1,A1}} = 4$.

Using Equation 6.1, the join-classes of R1 are:

$C^{0}_{R1,A1} = [0, 2, 10]$

$C^{1}_{R1,A1} = [0, 6]$

$C^{2}_{R1,A1} = [1, 3]$

$C^{3}_{R1,A1} = [1, 3]$

$R1_0$, $R1_1$ & $R1_3$ are partitions in more than one join-class. $C^{0}_{R1,A1}$ has 3 partitions while the other classes have 2 each.

For join class i

```
BEGIN
    x := root
    push(x, stc)
    k := 0
    WHILE (not empty(stk)) DO
    BEGIN
      pop(x, stk)
      FOR j=0 TO reclimit - 1 DO
     BEGIN
         IF ( Equal(x ⌐ (j).prt, i) then
          BEGIN
           IF (x ⌐ (j).type = data) then
          BEGIN
             k := k + 1
            $C^{i,k}_{R,A}$ = x.prt;
          END
           ELSE
             push(x ⌐ (j).pntr, stk)
       END
     END
   END
 END
```

Algorithm 6.1: Collects partitions of join-class i from a nonuniform relation R. A is the join attribute of R.

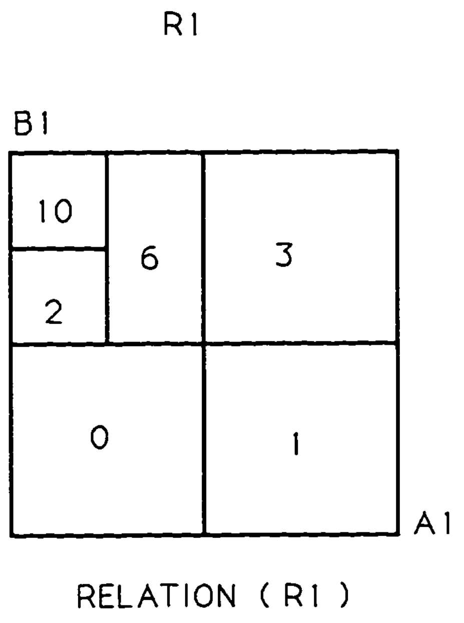


Figure 6.1 : A nonuniform relation

Since the partitions of $C^2_{R1,A1}$ and $C^3_{R1,A1}$ are identical and 2 is less than 3, $C^3_{R1,A1}$ is embedded in $C^2_{R1,A1}$.

***

6.2 Aggregate-classes

With minor modification, the idea of aggregate-classes which was introduced in the previous chapter is also used in the join of two relations with nonuniformly distributed data records. When joining two relations with uniformly distributed data records each aggregate-class will have equal number of join-classes. And this number is fixed by the knowledge of the axis numbers of the join attributes in each relation and the partition level of each relation. But in the join of two relations with nonuniform data record distribution, each class of a relation not necessarily join with equal number of join-classes. And the number of the join-classes for each aggregate-class is determined after accessing the directory records of both the relations which are to be joined.

In partitioning join-classes to form aggregate-classes, the labels of the intervals, on the join-axes, are used. The label of the highest labeled interval, from those intervals covered by a particular partition, can be computed using the following 5 parameters:

1. The number(id) of the particular partition.
2. The number of the attributes in the relation.
3. The axis number where these intervals lie.

4. The axis partition level now.
5. The value of the axis partition level when this particular partition was last divided.

Let the number of attributes in R be d, the axis number of A in the schema of R be k, the binary representation of i be b_{l_R-1}, b_{l_R-2}, $\ldots$, b_0, the axis partition level of A when R_i was last divided or created be m then the label of the highest labeled interval , from those intervals covered by R_i is:

$$\sum_{j=m-1}^{0} 2^j \times b_{k+(2^m-(j+1))(d)} \quad + \quad 2^{l_{R,A}-m} - 1 \qquad \textbf{(6.2)}$$

Partitions of a uniform relation cover equal number of axis intervals. Two partitions of a nonuniform relation not necessarily cover the same number of axis intervals. Partitions of a join-class, of a uniform relation, cover identical axis intervals, while those of nonuniform relation have at least one axis-interval in common. The common axis intervals for partitions of a join-class are used in the mapping of the join-class to one aggregate-class during the join of nonuniform relations. Let the label, of the highest labeled common axis interval, covered by the partitions of $C^i_{R,A}$ be $HCIL^i_{R,A}$. See Example 6.2 for more clarification of the points discussed so far in this section.

Example 6.2

This example uses R4 (Figure 6.2) to clarify the finding of the highest common interval label (HCIL), from the axis interval labels covered by all the partitions of a join-class. The axis considered here is that of A4.

$C^0_{R4,A4} \;=\; [0,\ 8,\ 2]$

The intervals covered by each partition of $C^0_{R4,A4}$ are :

$R4_0$ covers 0

$R4_8$ covers 0, 1

$R4_2$ covers 0, 1, 2, 3

Axis interval common to all the partitions of $C^0_{R4,A4}$ is 0. $HCIL^0_{R4,A4} \;=\; 0$.

$C^1_{R4,A4} \;=\; [16,\ 8,\ 2]$

The intervals covered by each partition of $C^1_{R4,A4}$ are :

$R4_{16}$ covers 1

$R4_8$ covers 0, 1

$R4_2$ covers 0, 1, 2, 3

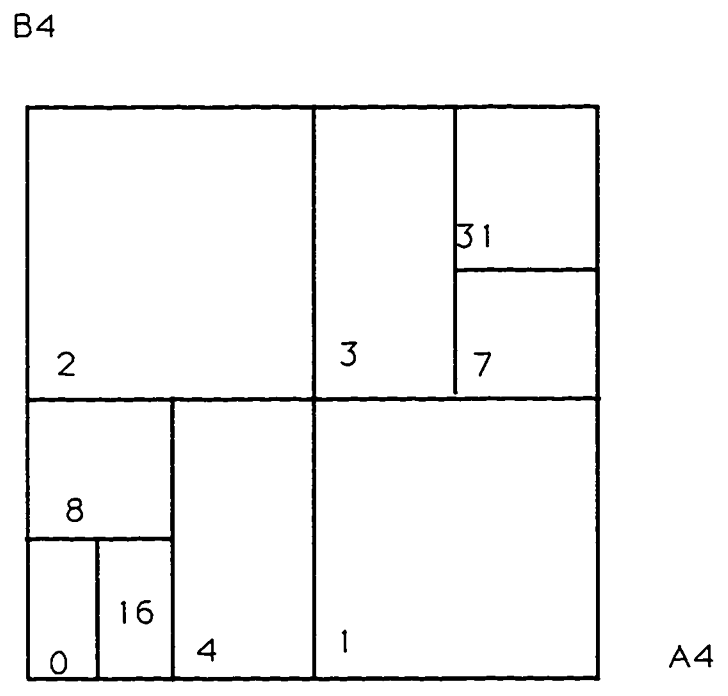
Axis interval common to all the partitions of $C^1_{R4,A4}$ is 1. $HCIL^1_{R4,A4} \;=\; 1$.

$C^2_{R4,A4} \;=\; [4,\ 2]$

The intervals covered by each partition of $C^2_{R4,A4}$ are :

$R4_4$ covers 2, 3

$R4_2$ covers 0, 1, 2, 3



RELATION (R4)

Figure 6.2 : A nonuniform relation

Axis interval common to all the partitions of $C^2_{R4,A4}$ are 2, 3. $HCIL^2_{R4,A4} = 3$.

$C^4_{R4,A4} = [1, 3]$

The intervals covered by each partition of $C^4_{R4,A4}$ are :

$R4_1$	covers	4, 5, 6, 7
$R4_3$	covers	4, 5

Axis interval common to all the partitions of $C^4_{R4,A4}$ are 4, 5. $HCIL^4_{R4,A4} = 5$.

$C^6_{R4,A4} = [1, 7, 15]$

The intervals covered by each partition of $C^6_{R4,A4}$ are :

$R4_1$	covers	4, 5, 6, 7
$R4_7$	covers	6, 7
$R4_{15}$	covers	6, 7

Axis interval common to all the partitions of $C^6_{R4,A4}$ are 6, 7. $HCIL^6_{R4,A4} = 7$.

***

Formation of aggregate-classes out of join-classes is as follows:

In the join of R and S, to start with, there is one aggregate-class of R, $AC^0_{R,A}$, and its compatible aggregate-class from S , $AC^0_{S,B}$. In the beginning both these classes are empty. Then $C^0_{R,A}$ is added to $AC^0_{R,A}$ and $C^0_{S,B}$ is added to $AC^0_{S,B}$. Now if $HCIL^0_{R,A}$ is equal to $HCIL^0_{S,B}$, $AC^0_{R,A}$ and $AC^0_{S,B}$ are done. But if

$HCIL^0_{R,A}$ less than $HCIL^0_{S,B}$, the second join-class (in the order of increasing join-class labels) is added to the join-classes of $AC^0_{R,A}$. Let this join-class be $C^{k(1)}_{S,B}$. If $HCIL^{k(1)}_{R,A}$ is equal to $HCIL^0_{S,B}$ these two classes are done other wise more join-classes are added till the $HCIL^{k(n)}_{S,B}$, where $k(n)$ is the label of the last join-class added to $AC^0_{R,A}$, is equal to $HCIL^0_{S,B}$. If the case were that $HCIL^0_{S,B}$ less than $HCIL^0_{R,A}$ more join-classes would have been added to $AC^0_{S,B}$ till $HCIL^{k(m)}_{S,B}$, where $k(m)$ is the label of the last join-class added to $HCIL^0_{S,B}$, is equal to $HCIL^0_{R,A}$. If there are some join-classes not yet included in $AC^0_{S,B}$ and $AC^0_{R,A}$, $AC^1_{S,B}$ or $AC^1_{R,A}$ are created to ingulf all or some of them . New aggregate-classes will be built till all the join-classes of R and S are ingulfed. Unlike the labels of the join-classes the labels of aggregate-classes are always sequential and by convention they start from 0. The number of aggregate-classes in both the join operand relations are always equal. From now on let the number of aggregate-classes in each of the join operand relations be represented as $\alpha^{S,B}_{R,A}$, where R and S are the join operand relations and A and B the join attributes. Similarly let the number of partitions in $AC^i_{R,A}$ for $0 \le i \le \alpha^{S,B}_{R,A}$, be $\beta^i_{R,A}$. An aggregate-class in R will always have one and only one join compatible aggregate-class from S. Now Equation 5.3 can be modified to include the join of nonuniform relations as follows:

$$R \Xi_{A-B} S = AC^i_{R,A} \Xi AC^i_{S,B} \quad \text{for } 0 \le i \le \alpha^{S,B}_{R,A} - 1 \qquad \textbf{(6.2)}$$

5.3 is special case of Equation 6.2 in which $C^i_{R,A}$ is $AC^i_{R,A}$ and $\alpha^{S,B}_{R,A} = 2^{l_{R,A}}$.

At processor $P_{1,1}$

```
BEGIN
  $i = 0$
  $j = 0$
  $k = 0$
  $u(j) = 0$
  $v(k) = 0$
  $AC^{i}_{R,A} = [\ ]$
  $AC^{i}_{S,B} = [\ ]$
  $done = false$
  $WHILE\ (\ NOT\ done\ )\ DO$
   BEGIN
    $AC^{0}_{R,A} = AC^{i}_{R,A} + C^{u(j)}_{R,A}$
    $AC^{i}_{S,B} = AC^{i}_{S,B} + C^{v(k)}_{S,B}$
    $hi\_com\_lbl(\ HCIL^{u(j)}_{R,A},\ C^{u(j)}_{R,A},\ l_{R,A})$
    $hi\_com\_lbl(\ HCIL^{v(k)}_{S,B},\ C^{v(k)}_{S,B},\ l_{S,B})$
    $IF\ (\ HCIL^{u(j)}_{R,A}) < HCIL^{v(K)}_{S,B})$ THEN
      $k = k + 1$
    $ELSEIF\ (\ HCIL^{u(j)}_{R,A}) > HCIL^{V(K)}_{S,B})$ THEN
      $j = j + 1$
    $ELSEIF\ (\ j = \alpha^{S,B}_{R,A}\ )$ THEN
     BEGIN
      $i = i + 1$
      $AC^{i}_{R,A} = [\ ]$
      $AC^{i}_{S,B} = [\ ]$
      $j = j + 1$
      $k = k + 1$
     END
    $ELSE$
     $done = TRUE$
  END
END
```

Algorithm 6.2 : Aggregate-classes builder.

Algorithm 6.2 is aggregate-classes builder. In the algorithm u & v are integer arrays containing join-class labels of the two join operand relations. The labels in each array are kept sorted in increasing order. *Hi-com-lbl* is a subprogram which finds the highest common axis interval label of a join-class partitions. Example 6.3 shows step by step the formation of aggregate-classes.

Example 6.3

This example uses R2 & R3 of (Figure 6.3) in showing the formation of aggregate-classes. In the join of R2 and R3, let A2 of R2 and A3 of R3 be the join attributes.

The join-classes in R2 are :

$C^0_{R2,A2} = [0, 2]$

$C^2_{R2,A2} = [1, 3]$

$C^3_{R2,A2} = [1, 7]$

The join-classes in R3 are :

$C^0_{R3,A3} = [0, 2]$

$C^1_{R3,A3} = [0, 6]$

$C^2_{R3,A3} = [1, 3]$

The formation process of the aggregate-classes is :

$AC^0_{R2,A2} = [C^0_{R2,A2}]$

$AC^0_{R3,A3} = [C^0_{R3,A3}]$

Since $(HCIL^0_{R2,A2} = 1) < (HCIL^0_{R3,A3} = 0)$ so $C^1_{R3,A3}$. has to be added to $AC^0_{R3,A3}$.

$AC^0_{R3,A3} = [C^0_{R3,A3}, C^1_{R3,A3}]$

$HCIL^0_{R2,A2} = 1 = HCIL^1_{R3,A3} = 1.$

This ends the formation of $AC^0_{R2,A2}$ & $AC^0_{R3,A3}$. But there are still some join-classes not yet ingulfed in either $AC^0_{R3,A3}$ or $AC^0_{R2,A2}$.

$AC^1_{R2,A2} = [C^2_{R2,A2}]$

$AC^1_{R3,A3} = [C^2_{R3,A3}]$

Since $(HCIL^2_{R2,A2} = 2) < (HCIL^2_{R3,A3} = 3)$ so $AC^3_{R2,A2}$. has to be added to $AC^1_{R2,A2}$.

$AC^1_{R2,A2} = [C^2_{R2,A2}, C^3_{R2,A2}]$

$HCIL^3_{R2,A2} = 3 = HCIL^2_{R3,A3} = 3.$

This ends the formation of $AC^1_{R2,A2}$ & $AC^1_{R3,A3}$.

Since now all the join-classes are ingulfed, the process of aggregate-class formation ends. $\alpha^{S,B}_{R,A} = 1.$

***

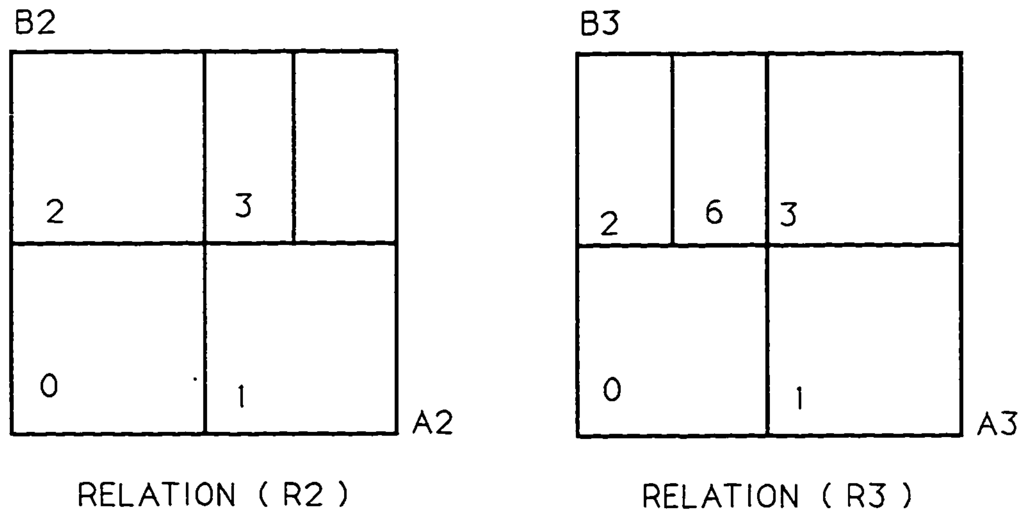


Figure 6.3 : Nonuniform relation

6.3 Join in a single processor

In the join of two relations, R and S, on attributes A of R and B of S, a processor P starts collecting partitions of $AC^0_{R,A}$ and $AC^0_{S,B}$ form the data and directory search space of R and S respectively. The join starts by P first reading $AC^{0,1}_{R,A}$ and $AC^{0,1}_{S,B}$. After the join of these two partitions P again reads $AC^{0,2}_{S,B}$ and joins it with $AC^{0,1}_{R,A}$. P next reads $AC^{0,3}_{S,B}$ and joins it with $AC^{0,1}_{R,A}$. P keeps reading the rest of $AC^0_{S,B}$ partitions , $AC^{0,4}_{S,B}$, $AC^{0,5}_{S,B}$, . . . , $AC^{0,q}_{S,B}$, where $q = \beta^0_{S,B}$, and joining them with $AC^{0,1}_{R,A}$, which is already in P. Since $AC^{0,1}_{R,A}$ is through joining with all the partitions of $AC^0_{S,B}$ it is replaced in P by $AC^{0,2}_{R,A}$. $AC^{0,2}_{R,A}$ starts joining with all the partitions of $AC^0_{S,B}$, which by now are all in P. The same process, of reading a partition from $AC^0_{R,A}$ and the joining of it with all the partitions of $AC^0_{S,B}$ continues till the join of $AC^0_{R,A}$ with $AC^0_{S,B}$ is done. Next the processing of $AC^1_{R,A}$ and $AC^1_{S,B}$ start and it will proceed as the processing of $AC^0_{R,A}$ and $AC^0_{S,B}$. The processing of $AC^1_{R,A}$ and $AC^1_{S,B}$ will be followed by the processing of the next two compatible aggregate-classes which are again followed by the processing of the next aggregate-classes and so on till the last two aggregate-classes are processed. This ends the join of R and S. Algorithm 6.3 is a join algorithm for two nonuniform relations in a single processor. Example 6.4 is an example of a join operation on two nonuniform relations, in a single processor.

Example 6.4

This is an example of a join operation on relations R2 and R3 (Figure 6.4). A2 of R2 and A3 of R3 are the join attributes. The join is performed in a single processor, P.

The aggregate-classes, their partitions and their sizes are:

$AC^0_{R2,A2}$ = [0, 2]

$AC^1_{R2,A2}$ = [1, 3, 7]

$AC^0_{R3,A3}$ = [0, 2, 6]

$AC^1_{R3,A3}$ = [1, 3]

$\beta^0_{R2,A2} = 2$

$\beta^1_{R2,A2} = 3$

$\beta^0_{R3,A3} = 3$

$\beta^1_{R3,A3} = 2$

The join process is as follows:

P, first reads $R2_0$ followed by the read of $R3_0$. After these two partitions join, P again reads $R3_2$ and joins it with $R2_0$. The last partition in $AC^0_{R3,A3}$, $R3_6$, is read and is joint with $R2_0$ finishing the join of $R2_0$ with all the partitions of $AC^0_{R3,A3}$. The second partition in $AC^0_{R2,A2}$ is read next and it is directly joined with all the 3 partitions of $AC^0_{R3,A3}$ finishing the join of $AC^0_{R2,A2}$ with $AC^0_{R3,A3}$. The join of $AC^1_{R2,A2}$ with $AC^1_{R3,A3}$ starts with the read of $R2_1$ followed by the read of $R3_1$.

```
FOR $i = 0$ to $\alpha^{S,B}{}_{R,A} - 1$ DO
BEGIN
  FOR $k = 1$ to $\beta^{i}{}_{R,A}$ DO
  BEGIN
    read( $AC_{R,A}{}^{i,k}$)
    FOR $j = 1$ to $\beta^{i}{}_{S,B}$ DO
    BEGIN
      IF ( $k = 1$) THEN
      BEGIN
        read( $AC_{S,B}{}^{i,j}$)
      END
      join( $AC_{R,A}{}^{i,k} AC_{S,B}{}^{i,j}$)
    END
  END
END
```

Algorithm 6.3 : Join algorithm in a single processor.

After the join of these two partitions P again reads the last partition of $AC^1_{R3,A3}$, $R3_3$. The join of these partition with $R2_1$ finishes the join of $R2_1$ with all the partitions of $AC^1_{R3,A3}$. $R2_3$ is read next and it joins with all the partitions of $AC^1_{R3,A3}$ for they are all in P. P at last reads the last partition in $AC^1_{R2,A2}$ and joins it with all the partitions of $AC^1_{R3,A3}$ ending the join of R2 and R3.

***

6.4 Join in a vector of processors

If a vector of n processors is employed to perform a join of relations R and S on join attributes A of R and B of S, each P_i for $0 \leq i < n$, must know the following 3 parameters :

1. The partition levels of R and S.
2. The partition in each aggregate-class.
3. The join-axis-intervals covered by each partition.

All these 3 parameters are processed by P_0. P_0 extracts these parameters by searching through the directory and data search space of both R and S. Then P_0 pipelines the first parameter to the rest of the processors in the vector before starting to process the join of the first two join compatible aggregate-classes. It pipelines the last 2 parameters just before their corresponding aggregate-classes are processed. The join starts by the reading of the first $k(1)$ partitions of

$AC^0_{S,B}$, where $0 \le k(1) < n$, by processors P_i for $0 \le i \le k(1)$. Each P_i reads only one partition. P_0 again reads $AC^{0,1}_{R,A}$. This page is pipelined from P_0 to processors P_i for $1 \le i \le k(1)$. Then $AC^{0,1}_{R,A}$ is joint with all the $k(1)$ partitions in the vector . If there are more unprocessed partitions of $AC^0_{S,B}$ another $k(2)$, where $0 \le k(2) \le n$, partitions are read by the first $k(2)$ processors of the vector . Again these $k(2)$ partitions are joint with $AC^{0,1}_{R,A}$. This process of, reading more partitions of $AC^0_{S,B}$ and joining them to $AC^{0,1}_{R,A}$ continues till all the $\beta^0_{S,B}$ partitions are processed with $AC^{0,1}_{R,A}$. P_0 replaces $AC^{0,1}_{R,A}$ by reading $AC^{0,2}_{R,A}$. This partition is then pipelined to all the processors holding at least one partition of $AC^0_{S,B}$. Since all the partitions of $AC^0_{S,B}$ are in the vector $AC^{0,2}_{R,A}$ starts joining as soon as it arrives in each processor. In such a way all the partitions of $AC^0_{R,A}$ are joint with all the partitions of $AC^0_{S,B}$. This process again is repeated for the next join compatible aggregate-classes in sequence, till all the the compatible aggregate-classes are joined, ending the join of R with S. See Algorithm 6.4 for the join of nonuniform relations in a vector of processors. Example 6.5 shows the steps of how two relations join in a vector of 4 processors.

Example 6.5

In this example relations R5 and R6 (Figure 6.4) are used to join in a vector of 4 processors, P_i for $0 \le i \le 3$. The join Attributes used are A5 of R5 and A6 of R6.

```
FOR $h$ = 0 to n-1 DO
   with $P_h$
   FOR $i$ = 0 to $\alpha^{S,B}_{A,R} - 1$ DO
   BEGIN
     FOR $k$ = 1 to $\beta^{i}_{A,R}$ DO
     BEGIN
       BEGIN
         IF ( $h <> 1$) then
         BEGIN
           accept( $AC^{i,k}_{R,A}$)
         ELSE
           read( $AC^{i,k}_{R,A}$)
         END
         IF ( $h < n$) then
         BEGIN
           send( $C^{i,k}_{R,A}$)
         END
         FOR $j$ = $f(h-1,n)$ TO $f(n,h)$ DO
         BEGIN
           IF ( $k=1$) THEN
           BEGIN
             read( $AC_{S,B}^{i,j}$)
           END
           join( $AC_{R,A}^{i,k}$,  $AC_{S,B}^{i,j}$)
         END
     END
   END
```

Algorithm 6.4 : Join algorithm for a vector of processors.

First P_0 will read the partition levels of R5 and R6 and will pipeline them to the rest of the processors in the vector. Then P_0 will build $AC^0_{R5,A5}$ and $AC^0_{R6,A6}$. It will compute $\beta^0_{R5,A5}$ and $\beta^0_{R6,A6}$. It will also pipeline these to the rest of the processors. Now Each of the 4 processors of the vector will carry out the rest of the processing as follows:

P_0 will read $R6_0$, P_1 will read $R6_8$, P_2 will read $R6_2$ and P_3 will read $R6_{10}$. P_0 will again read a partition from $AC^0_{R5,A5}$, $R5_0$. This page will be pipelined from P_0 to all the processors in the vector. $R5_0$ will then join with each of the 4 partitions of $AC^0_{R6,A6}$ in its corresponding processor. Since $R6_0$ joined with all the partitions it was supposed to join P_0 replaces it by reading $R6_2$. This page is again pipelined to the other processors and it will start to join with same partitions that $R6_0$ did join, ending the join of $AC^0_{R5,A5}$ with $AC^0_{R6,A6}$. Next the 2 partitions of $AC^1_{R6,A6}$ are read, $R6_4$ by P_0 and $R6_6$ by P_1. P_0 again reads $R5_0$ and pipelines it to P_1. Then in P_1, $R5_0$ will join with $R6_6$ and in P_0 it will join with $R6_4$. P_0 will then read $R5_6$. This partition will be processed in the same way as $R6_4$. The last partition in $AC^1_{R5,A5}$, $R5_{14}$, will also be treated as $R6_4$ ending the join of $AC^1_{R5,A5}$ with $AC^1_{R6,A6}$. In the same way as that of $AC^1_{R5,A5}$ and $AC^1_{R6,A6}$, the partitions of $AC^2_{R5,A5}$ and $AC^2_{R6,A6}$ will be processed but this time three processors will be involved instead of two. This is because $B^2_{R6,A6} = 3$. Again in the same way $AC^3_{R5,A5}$ will be joint with $AC^3_{R6,A6}$ ending the join of R5 with R6.

***

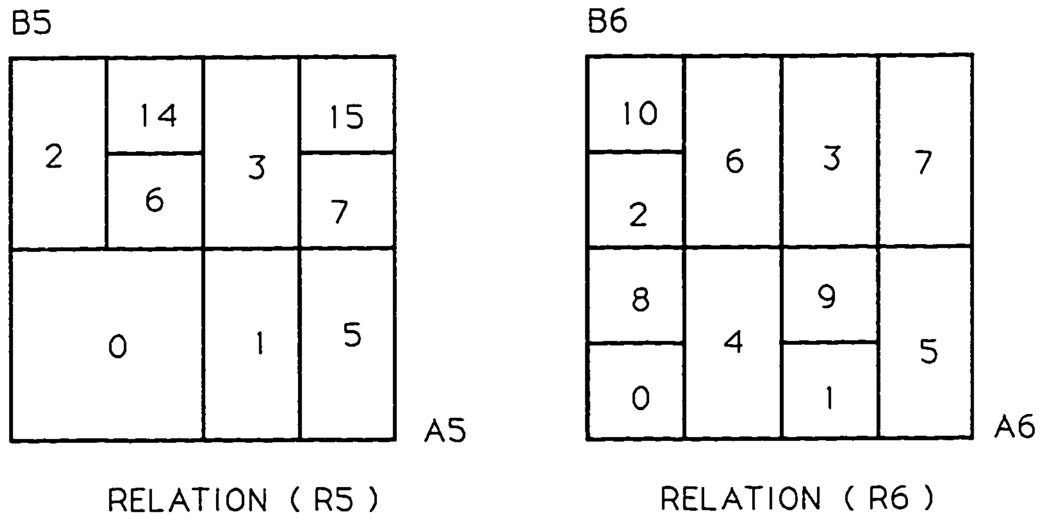


Figure 6.4 : Nonuniform relations

6.5 Join in a mesh

If a mesh of m by n is used to join relations R and S on attributes A of R and B of S, the knowledge of the following parameters by each processor in the mesh is essential.

1. The partition levels of R and S.
2. The partition numbers in each aggregate-class.
3. The join-axis-intervals covered by each partition.

All These parameters are extracted from the data and directory search space of both R and S by $P_{1,1}$. $P_{1,1}$ then pipelines the partition levels of R and S before the processing of the first two compatible aggregate-classes, namely $AC^0_{R,A}$ and $AC^0_{S,B}$, start. The partition numbers in each aggregate-class and the join-axis intervals covered by each partition in an aggregate-class are pipelined just before their corresponding aggregate-classes start to be processed. when $P_{1,1}$ pipelines the values of all the above mentioned parameters, the join starts by each $P_{i,1}$ reading a partition from $AC^0_{R,A}$ and each $P_{1,j}$, for $1 \leq j \leq n$, reading a partition from $AC^0_{S,B}$. Each $P_{i,1}$, for $1 \leq i \leq m$, then pipelines the partition it reads from $AC^0_{R,A}$, to all the processors in row i, while each $P_{1,j}$ pipelines the partition it reads from $AC^0_{S,B}$, to all the processors in column j of the mesh. So a $AC^0_{R,A}$ partition read by $P_{i,1}$ and a $AC^0_{S,B}$ partition read by $P_{1,j}$ will only meet and join in processor P_{ij}. Each $P_{1,j}$ when it is done processing the join of the two partitions it is holding it reads another partition of $AC^0_{S,B}$ which has never been read before. This page is pipelined to all the processors in the same column as that of the processor which read it. This new partition is then joint to the same $AC^0_{R,A}$ partition in all the processor it is pipelined to. Each $P_{1,j}$ repeats this process till it joins the $\frac{\beta^0_{S,B}}{n}$ partitions of $AC^0_{S,B}$ with the partition of R already held by it. The first set of $AC^0_{R,A}$ partitions which was read before, is now replaced by another set of $AC^0_{R,A}$ partitions. This set again goes through the same process as the first set. The later set will be processed faster because the partitions of $AC^0_{S,B}$ are already in the mesh. The second set will be followed by the third one and so on till join of $AC^0_{R,A}$ with $AC^0_{S,B}$ is complete. When $AC^0_{R,A}$ and $AC^0_{S,B}$ are done they will be replaced by $AC^1_{R,A}$ and $AC^1_{S,B}$ in the mesh. Again these two classes will be replaced by another two join compatible aggregate-classes till all the $\alpha^{S,B}_{R,A}$ join compatible aggregate-classes are processed, ending the join of R with S. Algorithm 6.5 is an algorithm for the join of 2 nonuniform relations in a mesh. Example 6.6 shows how the join of two relations can be carried out in a mesh.

Example 6.6

In this example relations R5 and R6 (Figure 6.4) are to be joined in a mesh of 2 by 2 processors. The join Attributes are A5 of R5 and A6 of R6.

First $P_{1,1}$ will read the partition levels of R5 and R6 and will pipeline them to the rest of the processors in the mesh. Then $P_{1,1}$ will build $AC^0_{R5,A5}$ and $AC^0_{R6,A6}$. It will also compute $\beta^0_{R5,A5}$ and $\beta^0_{R6,A6}$. It will then pipeline these parameters to the rest of the processors. Now $P_{1,1}$, $P_{1,2}$ and $P_{2,1}$ know which partitions to read from those of $AC^0_{R5,A5}$ and $AC^0_{R6,A6}$. $P_{1,1}$ will read $R6_0$ followed by $R5_0$, $P_{1,2}$ will read $R6_2$ and $P_{2,1}$ will read $R5_2$. $P_{2,2}$ will receive $R6_2$ from $P_{1,2}$ and $R5_2$ from $P_{2,1}$. $P_{1,1}$ will send $R5_0$ to $P_{1,2}$ and $R6_0$ to $P_{2,1}$. Then the join of , $R5_0$ with $R6_0$ will take place in $P_{1,1}$, $R5_0$ with $R6_2$ will take place in $P_{1,2}$, $R5_2$ with $R6_0$ will take place in $P_{2,1}$ and $R5_2$ with $R6_2$ will take place in $P_{2,2}$. Since there are still two more partitions of $AC^0_{R6,A6}$, $R6_8$ and $R6_{10}$, not yet joint with $R5_0$ with $R5_2$. $P_{1,1}$ will read $R6_8$ and send it to $P_{2,1}$ while $P_{1,2}$ will read $R6_{10}$ and send it to $P_{2,2}$. Then the join of , $R5_0$ with $R6_8$ will take place in $P_{1,1}$, $R5_0$ with $R6_{10}$ will take place in $P_{1,2}$, $R5_2$ with $R6_8$ will take place in $P_{2,1}$ and $R5_2$ with $R6_{10}$ will take place in $P_{2,2}$. This will end the join of $AC^0_{R5,A5}$ with $AC^0_{R6,A6}$. Again $P_{1,1}$ will build $AC^1_{R5,A5}$ and $AC^1_{R6,A6}$. It will also compute $\beta^1_{R5,A5}$ and $\beta^1_{R6,A6}$. It will again pipeline these parameter to the rest of the processors. Now $P_{1,1}$, $P_{1,2}$ and $P_{2,1}$ know which partitions to read from those of $AC^1_{R5,A5}$ and $AC^1_{R6,A6}$. $P_{1,1}$ will read $R6_4$ followed by $R5_0$, $P_{1,2}$ will read $R6_6$ and $P_{2,1}$ will read $R5_6$. $P_{2,2}$ will receive $R6_6$ from $P_{1,2}$ and $R5_6$ from $P_{2,1}$. $P_{1,1}$ will send $R5_0$ to $P_{1,2}$ and $R6_4$ to $P_{2,1}$. Then the join of , $R5_0$ with $R6_4$ will take place in $P_{1,1}$, $R5_0$ with $R6_6$ will take place in $P_{1,2}$. $R5_6$ with $R6_4$ will take place in $P_{2,1}$ and $R5_6$ with $R6_6$ will take place in $P_{2,2}$. Since there is still one more partition of $AC^1_{R5,A5}$, $R5_{14}$, not yet joint with $R6_4$ and $R6_6$. $P_{1,1}$ will be read it and sent to $P_{1,2}$. later $R5_{14}$ will be joint with $R6_4$ in $P_{1,1}$ and with $R6_6$ in $P_{1,2}$. This will end the join of $AC^1_{R5,A5}$ with $AC^1_{R6,A6}$. The remaining two pairs of join compatible aggregate-classes will be processed in the same way as that of the first two, ending the join of R5 with R6.

***

6.6 Load balancing

To gain a better efficiency each processor in the mesh must be kept as busy as possible. If some processors in the mesh are kept idle during the processing time, the efficiency thus the speed of processing becomes lower than when they are kept busy. During processing, If idleness can not be totally avoided it must be reduced to the minimum. This reduction can not be achieved unless the load that must be carried out by each processor is balanced. In joining two nonuniform relations R and S, a way to balance the load among the processors is by dividing the elements of $AC^{i}_{R,A1}$ among the m rows and that of $AC^{i}_{S,B1}$ among the n columns of an m by n mesh. The following two equations , 6.3 and 6.4, were used as load balancers in Example 6.6. In the join of $AC^{q}_{R,A1}$ with $AC^{q}_{S,B1}$ in a mesh of m by n, let $k_R = \beta^{q}_{R,A}$ MOD m and $k_S = \beta^{q}_{S,B}$ MOD n. Each P_{ij} will be assigned $AC^{q,s}_{R,A}$ for

$$s = (i\text{-}1) \times \frac{\beta^{q}_{R,A}}{m}, \quad (i\text{-}1) \times \frac{\beta^{q}_{R,A}}{m} + 1 \quad , \; . \; . \; . \; . \; , \quad i \times \frac{\beta^{q}_{R,A}}{m} - 1 \qquad \text{and}$$

$$(i) + \beta^{q}_{R,A} - k_R \text{ if } (k_R > i) \qquad \textbf{(6.3)}$$

and $AC^{q,t}_{R,A}$ for

$$t = (j\text{-}1) \times \frac{\beta^q_{S,B}}{n},\ (j\text{-}1) \times \frac{\beta^q_{S,B}}{n} + 1, \ldots, j \times \frac{\beta^q_{S,B}}{n} - 1 \text{ and } (j) + \beta^q_{S,B} - k_S$$

if $(k_S > j)$ **(6.4)**

Equations 6.3 and 6.4 some times are not fair enough in balancing load among processors. For example, in the join of $AC^q_{A,R}$ and $AC^q_{S,B}$ in a mesh of m by n let $\beta^i_{A,R} = \frac{3 \times m}{2}$ and $\beta^i_{S,B} = k \times n$. In the processing the first m partitions of $AC^q_{A,R}$ with the $k \times n$ partitions of $AC^q_{S,B}$ all the processors of the mesh will be involved. While in the processing the rest $\frac{m}{2}$ partitions of $AC^q_{A,R}$ with the $k \times n$ partitions of $AC^q_{S,B}$ only processors P_{ij} for $1 \le i \le \frac{m}{2}$ and for $1 \le j \le n$ will be involved. In other words half the processors of the mesh will be idle during the processing of the $\frac{m}{2}$ partitions of $AC^q_{A,R}$ with the $k \times n$ partitions of $AC^q_{S,B}$. Half the processors in the mesh will process $2 \times k$ partitions from both the aggregate-classes while the other half will process k partitions only. If the k, in the $k \times n$, is a large integer the idle processors will be kept idle for a longer time, thus affecting the efficiency badly. To minimize the unfair load balancing of Equation 6.3 the following improvement over Equation 6.3 was suggested. The problem is with the k_R partitions of Equation 6.3, which were assigned to the first k_R rows of the mesh. This left many processors uninvolved. But if each of the k_R partitions were assigned to $\frac{m}{k_R}$ rows nearly all the processors will be kept busy. According to this modification in Equation 6.3, partitions of $AC^{q,l}_{R,A}$ will be assigned to the following processors:

$P_{l-m,\, j}$, $P_{l-m + \frac{m}{k_R},\, j}$, $P_{l-m + 2\times\frac{m}{k_R},\, j}$, . . ., $P_{l-m + (k_R - 1)\cdot\frac{m}{k_R},\, j}$ for $m < l \leq \beta^{q}_{R,A}$

and for $1 \leq j \leq n$ and for $0 \leq q \leq \alpha^{S,B}_{A,R} - 1$.

With this modification in Equation 6.3, if the previous two aggregate-classes were processed again each processor in the mesh will process $\frac{3\times k}{2}$ partitions.

```
FOR $0 \le i \le m$ AND FOR $0 \le j \le n$ DO
    * With $P_{i,j}$
        FOR $t = 0$ to $\alpha^{S,B}_{R,A} - 1$ DO
        BEGIN
        $\gamma_R = \beta^t_{R,A}/m$
        $\gamma_S = \beta^t_{S,B}/n$
          FOR $h = (i-1) \times \gamma_R + 1$ TO $i \times \gamma_R$ DO
          BEGIN
            IF ($j = 1$) THEN
              read( $AC^{t,h}_{R,A}$)
            ELSE
            BEGIN
              accept( $AC^{t,h}_{R,A}$)
            END
           send( $AC^{t,h}_{R,A}$)
            FOR $g = (j-1) \times \gamma_S + 1$ TO $j \times \gamma_S$ DO
            BEGIN
              IF ($j = 1$) THEN
              BEGIN
                IF ( $h = (i-1) \times \gamma_R + 1$ ) THEN
                BEGIN
                    read( $AC^{h,g}_{S,B}$)
                    send( $AC^{h,g}_{S,B}$)
                END
                join( $AC^{t,h}_{R,A}$, $AC^{h,g}_{S,B}$)
              END
              ELSE
              BEGIN
                IF ( $h = (i-1) \times \gamma_R + 1$ ) THEN
                BEGIN
                  accept( $AC^{h,g}_{S,B}$)
                  IF ( $j < n$ ) send( $AC^{h,g}_{S,B}$)
                END
                join( $AC^{t,h}_{R,A}$, $AC^{h,g}_{S,B}$ )
              END
            END
          END
        END
```

Algorithm 6.5 : A join algorithm for nonuniform relations in a mesh of $m \times n$.

CHAPTER 7

SIMULATION

The simulation program consists of 3 main modules. Module 1 simulates Interpolation Based Grid File (IBGF). Module 2 simulates join algorithm for relations based on Interpolation Based Grid File and of uniform data distribution. Module 3 simulates join algorithm for relations based on Interpolation Based Grid File and with nonuniformly distributed data tuples.

7.1 Module 1

This module does 3 main functions. It simulates the Interpolation Based Grid File, distributes data equally among available storage elements and dumps to a file each join-class in a relation if the data distribution of the relation is nonuniform. In this module each directory partition is simulated by an array of records (tuples). Each directory tuple has 3 fields. Field 1 is a pointer field which points to another directory partition or it is nil if it is pointing to an implied data partition. Field 2 is a data tuple counter. It counts the number of data tuples of the implied data partition if field 2 is nil. Field 3 is partition level counter which counts the number of partition levels that an implied data or directory partition pointed to by field 2 has undergone. No structure simulates the data partitions or data tuples at all. Instead in each directory tuple, field 3 (data tuple counter) keeps the the count of the data tuples mapping to the index of a directory tuple. The mapping of each data tuple to an index of a directory tuple is as follows:

Each data tuple is a series of zeroes and ones, each generated randomly by a random number generator which produces binary numbers. To obtain more generation of zeroes than ones or more ones than zeroes there is a probability factor associated with the parameters of the random number generator subprogram. The binary number thus generated is then converted to a decimal number which is then used as an index to address a directory tuple in a directory partition. Field 2, data tuple counter, of the addressed directory tuple is then incremented by one. The number of data tuples that can be accomodated in a data partition is limited. If after, the increment of the value in field 2 of a directory tuple exceeds the limit, a new directory tuple index is computed from the index of the one whose data tuple counter just exceeded the limit and its partition level. Since the directory tuples which can be accomodated by a directory partition is limited, the new index can exceed the range of indices available in a directory partition. If the new index is in the range of directory tuple indices, the value of the data tuple counter of the directory tuple whose data tuple counter exceeded the limit is subtracted by D and the data tuple counter of the new directory tuple is assigned to D. D is a an integer value computed as follows:

Let b_0 be the size of data tuples that can be accomodated by a data partition. Let f be the probability factor that was associated with the random number generator in producing the binary numbers used to address the overflowed directory tuple. Then $D = (1-f) \times (b_0 + 1)$. But if the new address is out of the range of the directory indices a new directory partition is created. Field 1 of the overflowed tuple is assigned the address of this new directory partition. The data tuple counter of tuple 1 of the new directory partition is assigned to D and that of tuple 0 of the same directory partition is assigned to $b_0 - D + 1$. The data tuples counter of the overflowed directory tuple is useless from now on.

If a binary number generated by the random number generator maps to a directory tuple whose field 1 is not nil or in other words it is pointing to another directory partition , this directory partition which is pointed to by this directory tuple is made the current directory partition and a new set of binary numbers are generated again to address one of the directory tuples out of the current directory partition. This process continues till a directory tuple whose field 1 is nil is encountered. Obviously in the beginning, the root directory is the current directory. Whenever a data tuples counter exceeds the limit, the partition level (field 3) of the overflowed directory tuple is incremented by one and this incremented value is also assigned to field 3 of the directory tuple(s) generated from the overflowed directory tuple. The new index which is computed from the overflown directory tuple index and its partition level is computed as the following:

Let m be the partition level of the overflown directory tuple, n be the index of the overflown directory tuple then the new index is $= n + 2^m$.

Function 2 of this module is the data distribution of the relations to a number of available storage elements. There are 2 different distribution algorithms in the simulation. Distribution algorithm 1 is used with relations of uniform data distribution while distribution algorithm 2 is used for relations with nonuniform data distribution. In distribution algorithm 1 a partition p is stored in storage s, where s is an integer indexing a storage element, computed as follows:

Let S be the number of available storage elements, L_R be the partition level of relation R, $bit(i)$ the value of the i^{th} binary digit when P is written in binary, where $bit(0)$ is the most significant bit and $bit(l_R - 1)$ is the least significant bit. Then

$$s = \sum_{i=0}^{l_R/2} 2^i \times bit(2 \times i) + \sum_{i=0}^{l_R - 1/2} 2^i \times bit(2 \times i + 1) \text{ MOD } S.$$

In distribution algorithm 2 post order traversal is used on the Interpolation Based Grid File tree. If there are n data tuples in a relation and if there are S storage elements available to store these n data tuples, each storage element is made to store nearly $\frac{n}{S}$ data tuples. During a post order traversal a data tuple counter is used to reach a value of up to $\frac{n}{S}$ data tuples. If this number is reached when the post order traversal is still is in the middle of a partition, then all the tuples in this partition are added to $\frac{n}{S}$ and all this partitions and their tuples are stored in one storage element. A table of S entries is kept to show the range of partitions stored by each storage.

Function 3 of this module collects the partitions of each join-class of a relation and sequentially dumps each join-class to an external file starting from join-class 0. The first record of the file contains the partition level of a relation and the number of storage elements used. Since in nonuniform relations some join-classes can be embedded in others, an integer number which is the range of join-classes impeded in the current join-class is written in the file just before writing each partition of the current join-class in the sequence of join-classes.Since a partition can be a member of more than one join-class in a nonuniform relation, with each partition the range of join-classes it covers, the storage address in which it resides and a flag to indicate whether this partition is a data or a directory partition is written in the file. To signal the end of the partitions of a join-class a series of -1s are written in the line following the last partition of each join-class in the file. To join two nonuniform relations two such files, file 1 and file 2, are created by this module.

Before discussing the rest of the modules let us discuss how the hardware used in this thesis is simulated.

The operations done by the processors in the mesh are as follows:

1. Read
2. Receive
3. Send
4. Compute
5. Join
6. Wait

Fixing the time units that each of the above 5 operations use, excluding operation 6 which is wait, the processors, storage elements and storage-to-processor connections can be simulated as time unit counters. Thus the mesh of processors are represented in the simulation by a two dimensional array of counters. Each processor is represented by a single entry in the two dimensional array. For simplicity the processor in row i and column j in the mesh is represented by the entry in row i and column j in the two dimensional array of counters. The S storage elements are also represented by a one dimensional array of time unit counters. Storage labeled i where $0 \le i < S$ is represented by the i^{th} entry of the one dimensional array. The C connections are also represented by a one dimensional array of time unit counters. Storage to processor connection labeled i where $0 \le i < C$ is represented by the i^{th} entry of the one dimensional array.

To minimize the time consumed by wait it is better to make some balance between the processing time and the read time. This is done by appropriately choosing the right page size. Since seek time is quite big compared to transmission or computation time, to minimize the number of seek operations, bigger pages are preferred. The choice of the size of the data pages in this simulation was done as follows:

The size of a data tuple was fixed to 56 bytes. The seek time was fixed to 8 milliseconds. The transmission time was fixed to 16 mega bits per second. The computation time together with the join time was fixed to 4 micro second for a pair of tuples. So the number of tuples, t, in each data page was computed as follows: Let Tc be time needed to transmit 1 data tuple and Tj be time needed to join two tuples. The the time needed to join two pages, $4 \times t \times t$ must be nearly equal to the time needed to read two pages which is equal to $2 \times (28 \times t + 8000)$. Computing t form the above equation resulted in t to be 73. And the size of the data page with t of 73 was 4k.

7.2 Module 2 & Module 3

Module 2 is the join algorithm for relations with uniformly distributed data partitions while module 3 is the join algorithm for relations with nonuniformly distributed data partitions. The procedures for the join were thoroughly discussed in Chapters 5 and 6. In this section, only the simulation details are discussed.

Initially all the counters are set to zeroes. External information needed by module 1 are the partition level of each relation to be joined and the number of storage elements each relation is using. No directory pages are read in module 1. A processor requesting to read a data page checks the time counter of the storage element, the time counter of the connection that connects it with the storage element and its own time counter. Then it synchronizes its time with the time of the connection and the storage by assigning the value of the highest entry , out of these 3 counters. Then the read is complete by incrementing these 3 counters by the time units needed to read and transmit a page. In sending a page to a neighbour, the times of both the sending and the receiving processors is synchronized by letting the time of the receiving processor more than that of the sending by the time needed to transmit a page. Then after the transmission the times of both the receiving and sending processors are incremented by the time that was needed to transmit the page. The join of two data pages is done when both pages reside into the same processor. So join is simulated by incrementing the processor counter by the time needed to join 2 pages. When joining relations with nonuniform data distribution 2 join-compatible pages residing in the same processor simultaneously do not necessarily join. This is because some pages can be a member of more than one join-class. So some pages can appear again and again with subsequent join-classes. So if such two pages are once met for the first time and made to join, they must not be made to join again. This is implemented by associating a flag with each data partition. When this data partition comes for its first time in the mesh its flag is set to false. By the end of the join of the current join-classes, its flag is set to true. Two join compatible pages residing in the same processor are allowed to join if at least the flag of one of them is false. In other words two join compatible partitions with both true flags meeting in a processor will not join this time because they have already joined once; they appeared as members of some previously done join-classes.The programs for modules 1,2 and 3 are in appendices 1, 2 and 3 respectively.

CHAPTER 8

SIMULATION RESULTS

In this Chapter, the term efficiency which was discussed in Chapter 2 is used as the main performance factor for the analysis of the join algorithm for relations with uniform data distribution and for the join algorithm for relations with nonuniform data distribution. Section 8.1 discusses the efficiencies of the join algorithm for relations with uniformly distributed data partitions and Section 8.2 discusses the efficiencies obtained by the join algorithm for relations with nonuniform data distribution. The effect of, number of processors, number of storage elements, number of storage to processor connections (s-p-connections), number of aggregate-classes and number of partitions per aggregate-class, on the efficiency of the algorithms of Chapters 5 and 6 is also discussed in both the Sections, 8.1 and 8.2. Section 8.3 discusses the efficiency results obtained when the join algorithm for relations with uniform data distribution was used to join relations with nonuniformly distributed data partitions.

Another two factors, affecting the efficiency, inherent with any pipeline system, are as below.

Factor 1 : When starting a process in a pipeline environment some time is needed for the processors at the rear of the pipeline to start processing.

Factor 2 : Near the end of a process in a pipeline environment, the processors at the front of the pipeline will be kept idle while those in the rear are still active.

These two factors contribute to the reduction of the overall efficiency of a pipelined system. In a process of a very short duration these two situations will contribute proportionately more, thus reducing the overall efficiency. The time of each factor increases as the number of the processors in the pipeline increases.

8.1 Uniform relations

8.1.1 Number of partitions

In the join of R and S let $k = \beta^q_{R,A}$ and $l = \beta^q_{S,B}$, where $0 \le q \le \alpha^{S,B}_{R,A} - 1$. Every partition in $AC^q_{R,A}$ must be processed with all the l partitions of $AC^q_{S,B}$. If a mesh of $m \times n$ is used to do the join of R and S, the aggregate-classes formed will be one of the following 4 types:
(Let the partitions of R be read by the processors that are in the first column of the mesh.)

Type 1: when $k < m$ and $l < n$.

Relations of this type of aggregate-classes will only be processed by $k \times l$ processors, while $(m\text{-}k) \times l$ processors will remain idle. This type of classes will contribute to low efficiency. The smaller the value of $k \times l$ the lower is the efficiency.

Type 2: when $k < m$ and $l \geq n$.

Relations of this type of classes will be processed by $k \times n$ processors only. The rest will remain idle resulting in low efficiency. Lower values of k will increase the number of idle processors thus lowering the efficiency more. Increasing the value of l will not increase or decrease the number of idle processors but will keep the active processors active for much longer time thus contributing to minor efficiency improvement.

Type 3: when $k \geq m$ and $l < n$.

Relations of this type of classes will be processed by $m \times l$ processors only. The rest will remain idle resulting in low efficiency. Lower values of l will increase the number of idle processors thus lowering the efficiency more. Increasing the value of k will not increase or decrease the number of idle processors but will keep the active processors active for much longer time thus contributing to minor efficiency improvement.

Type 4: when $k \geq m$ and $l \geq n$.

Relations of this type of aggregate classes will be processed by all the processors in the mesh resulting in a high efficiency. Higher values of l or k will keep the processors continuously busy for longer periods of time thus resulting in even better efficiency.

8.1.2 Number of aggregate-classes

Increasing the number of aggregate-classes while keeping the number of tuples in a relation unchanged will result in poor efficiencies for the following reasons :

1. Dividing the partitions amongst many aggregate-classes means fewer number of partitions per class, thus increasing the probability of aggregate-classes of types 1,2 and 3.

2. When the join of an aggregate-class with its join-compatible aggregate-class is over, both of these classes must be sent out of the mesh and be replaced by another aggregate-class and its join compatible one. This transition period will force many processors to stay idle due to *factors* 1 and 2. As the aggregate-classes increase the transition periods also increase, lowering the efficiency.

Table 8.1 shows the effect of increasing the number of partitions or the number of aggregate-classes on the efficiency. As the number of partitions within each aggregate-class was doubled efficiency improved significantly. When the number of aggregate-classes was doubled with out decreasing the number of partitions within each aggregate-class the improvement in the efficiency was insignificant. Doubling the number of aggregate-classes but decreasing the number of partitions in each aggregate-class resulted in poor efficiency.

8.1.3 Number of processors

Increasing the number of processors might increase the speed of processing but it decreases efficiency due to the following reasons:

1. Increased overall transition time due to the increased time of factors 1 and 2.

Number of partitions per join-class	Number of join-classes	Efficiency
8	8	40.8
8	16	44.5
16	16	61.5
16	32	62.5
32	32	77.1
32	64	77.2
64	64	87.3

Table 8.1 : Effect of partition and join-class size on efficiency. A mesh of 2 by 2 is used for table.

2. Increased the probability of an aggregate-class to be of types 1,2 or 3.

3. Shortened the time of continuous processing of each aggregate-class. This is because each processor has to process fewer partitions from each aggregate-class.

Table 8.2 shows how the efficiency is affected when the number of processors is increased and the sizes of the joining relations are kept unchanged. Table 8.3 shows increasing both the number of processors and the sizes of the joining relations by the same proportion results in nearly the same efficiency.

8.1.4 Number of storage elements

Increasing the number of storage elements is decreasing the number of partitions in each element. As a result the number of simultaneous requests for the same element will decrease. Efficiency will increase because processors requesting access to a storage will get them sooner. Table 8.4 shows how the increment in the number of storage elements affects the efficiency. One can see from the table, as the number of storage devices was doubled the efficiency did not improve significantly. This is because the whole join process is more towards computation (CPU) bound than towards I/O bound.

8.1.5 Number of s-p-connections

Increased number of s-p-connections means less number of processors connected to each s-p-connection. So a processor requesting a free storage element wouldn't wait longer due to high competition for the same s-p-connection. Table 8.5 shows how the efficiency was insignificantly affected each

Number of processors	Efficiency	Speedup
64	77.3	50.5
32	81.5	26.1
16	87.5	14.0
8	90.1	7.2
4	93.4	3.7

Table 8.2 : Effect of number of processors on the efficiency. 1024 sized relations used.

Number of processors	Size of R	Size of S	Efficiency
4	256	256	87.7
16	1024	1024	87.5
64	4096	4096	87.4
256	16384	16384	87.1

Table 8.3 : Effect of ratio of relation size to processor size on the efficiency.

Number of storage elements	**Efficiency**	**Speedup**
1	49.0	31.4
2	68.0	43.5
4	72.5	46.4
8	77.1	49.3
16	78.3	50.1

Table 8.4 : Effect of number of storage elements on efficiency. Relations of 1024 tuples and mesh of 64 is used.

Number of s-p-connections	Efficiency	Speedup
1	49.0	31.4
2	68.9	43.7
4	74.5	47.5
8	77.3	49.5

Table 8.5 : Effect of number of s-p-connections on efficiency. 16 storages & 64 processors were used.

time the number of s-p-connections is decreased by half. This is because, as was mentioned in subsection 8.1.4, the whole join process is more towards computation (CPU) bound than towards I/O bound.

8.2 Nonuniform relations

8.2.1 Number of partitions

In the uniform case the number of partitions in each aggregate-class is the same. So in the join of two uniform relations, processors which are idle during the processing of the first join compatible aggregate-classes will remain idle and processors which were active will remain active during the processing of the rest of the aggregate-classes, until the join of these two relations is over. But in the join of two nonuniform relations this is not the case. Processors active in the processing of one aggregate-class might participate in the processing of the next aggregate-class. And processors active in the processing of one aggregate class might stay quite during the processing of the next one. This is because the number of partitions in the aggregate-classes of nonuniform relations can vary from one Aggregate-classes class to another. Join-compatible Aggregate-classes of uniform relation are of only one type, while those of nonuniform relations can vary in type from one join compatible aggregate-class to another. In other words, in the join of R and S, where R and S are uniform relations, if $AC^0_{R,A}$ and $AC^0_{S,B}$ are of type 1, all the other join-compatible aggregate-classes of S and R are also of type 1. But if R and S are nonuniform relations, if $AC^0_{R,A}$ and $AC^0_{S,B}$ are of type 1, $AC^1_{R,A}$ and $AC^1_{S,B}$ might be of type 1 or 2 or 3 or 4.

8.2.2 Number of aggregate-classes

For the same number of data tuples highly skewed relations have more join-classes than less skewed ones. As a result many join-classes of a highly skewed relation will have few number of partitions. In the join of two highly skewed relations, the number of aggregate-classes formed is high. Many aggregate-classes with few partitions will result in a poor efficiency due to the following reasons:

1- High number of transition periods due to the many aggregate-classes involved.

2- Most of the aggregate-classes contain few partitions. This might force many of them to be of type 1 or 2 or 3. And we know that having too many of these types is bad news.

Table 8.6 shows how the efficiency is extremely low when a R of 0.1 probability factor(Pf) was joint with S of 0.9 or 0.1 Pf. The join of R (0.1 Pf) with S (0.2 Pf) or (0.8 Pf) will result in a better efficiency than S (0.1 Pf) or (0.9 Pf). This is because S of 0.8 Pf or 0.2 Pf have fewer join-classes than S of 0.1 or 0.9. These fewer join-classes will form fewer aggregate-classes thus better efficiency. As the Pf of S approaches 0.5 the number of aggregate-classes formed decreases in number resulting in higher number of partitions per aggregate-class. Many join-classes of R will be mapped into the same aggregate-class. The number of aggregate-classes so formed will be less than or equal to the number of join-classes of the relation with the Pf nearest to 0.5. So ,when R of 0.1 Pf and S of 0.4 Pf are to be joint the number of aggregate-classes formed will

Pf of S

	S / R	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
	0.1	29.0	40.9	58.6	62.2	65.0	48.0	32.4	26.1	18.6
	0.2	41.4	34.8	45.0	53.0	64.4	42.6	34.9	29.3	26.4
Pf	0.3	58.7	46.4	40.9	46.2	63.1	43.9	37.0	34.5	36.4
of	0.4	61.8	53.2	47.6	43.8	62.4	44.1	42.2	40.2	45.5
R	0.5	67.9	60.9	59.6	60.3	70.2	61.5	61.0	61.5	66.1
	0.6	45.5	42.0	40.4	41.1	62.7	40.8	45.9	50.8	62.1
	0.7	33.4	35.9	36.9	40.3	63.7	44.6	35.8	40.9	58.1
	0.8	26.1	29.5	33.4	41.5	65.8	51.2	42.0	27.8	32.1
	0.9	18.7	26.4	33.4	47.3	70.9	62.2	59.9	31.6	22.2

Table 8.6: Effect of probability factors on efficiency.

Note: Pf stands for probability factor.

be less than or equal to the number of the join-classes of S. That is why when R of 0.1 or 0.9 Pf is joint to S of 0.5, the efficiency was high. See table 8.6. For the same number of tuples, S of 0.3 Pf and S of 0.7 Pf, will have the same number of join-classes. But when S of 0.7 Pf was joint with R of 0.1 Pf resulted in a lower efficiency than when S of 0.3 Pf was joint with R of 0.1 Pf. This is because S of 0.3 Pf and R of 0.1 Pf have more join-compatible aggregate-classes of type 4, Than S of 0.7 Pf and R of 0.1 Pf. As the Pf of R and Pf of S approaches 0.5, the number of aggregate-classes diminishes and the number of join compatible aggregate-classes of type 4 increases, thus the efficiency too increases.

In the join of 2 uniform relations, the efficiency increases if the size of the relations is increased. In the join of nonuniform this might not be always true. This is because increasing the size of relations by adding more partitions might result in more join-compatible aggregate-classes of type 1 or 2 or 3. Table 8.7 shows how the efficiency was decreased as the size of the nonuniform relations was increased.

8.2.3 Number of storage elements

Reducing the number of storage elements where the join operand relations reside, result in a poor efficiency. Processors requesting pages from the same storage will increase as the number of storage elements decrease. Thus processors have to wait idle for a longer time to get a page from a storage. Each time reducing the number of storage elements by half did not result in a proportional efficiency reduction. This is because, as mentioned previously, the

Size of R	Size of S	Efficiency	Speedup
25000	25000	58.7	37.5
32000	32000	52.3	33.5
45000	45000	70.1	44.9
64000	64000	54.7	34.6

Table 8.7 : Effect of relation size on efficiency.

Number of storage elements	Efficiency	Speedup
1	49.0	31.4
2	57.6	36.9
4	60.3	38.6
8	70.1	44.9
16	71.3	45.6

Table 8.8 : Effect of number of storage elements on efficiency. Relations of 1024 tuples and mesh of 64 are used.

whole join process is more towards computation bound than towards input output. Table 8.8 demonstrates this fact.

8.2.4 Number of s-p-connections

Reducing the number of s-p-connections also reduces the efficiency. This is because more processors will be connected to the same connection thus increasing the competition between processors for the same connection. Because the join process is competition bound reducing the number of such connections did not result in a decrease of efficiency by the same proportion. Table 8.9 demonstrates how varying the number of connections affect the efficiency of the join algorithm.

8.3 Uniform algorithm for nonuniform relations

In Chapter 5 it was discussed that the join algorithm for relations of uniform data distribution takes as an input the partition levels of each operand relations and the number of storage elements each of the operand relations are residing on. Having this information a processor can compute the page number it wants to read and it can also compute the address of the storage where this page is residing. Because the relations involved were all of uniform data distribution the processor knows which data partitions are physically present in the data search space of a relation. But if the relations involved are of nonuniformly distributed data partitions, having the information of their partition level will never be useful to determine what data partitions are physically present in the data search space of such relations. The only information that the partition level of such relations can tell is the maximum

Number of s-p-connections	Efficiency	Speedup
1	48.6	31.1
2	66.7	42.7
4	69.5	44.9
8	70.2	44.9

Table 8.9: Effect of number of s-p-connections on efficiency. 16 storages & 64 processors were used.

range of partition number that this relation can assume. In a very skewed relation the range is quite high. In fact as the skewedness of a relation increases, its range of partitions increases exponentially. So relations of probability factor of 0.9 and 0.1 have much higher ranges of partition than relations of 0.5 probability factor. Figure 8.1 shows the possible range of partitions of R, whose probability factor is 0.5, and that of S whose probability factor is 0.9. The range of partition numbers in R is from 0 to 7 while that of S is from 0 to 63. If S was an operand of our current join algorithm, this algorithm will try to read all the 64 possible pages, wasting a lot of read time for partitions that do not exist. But if R is used as an operand relation for this join algorithm, there are still some read statements which are attempted in reading non existent pages; but, they are fewer compared to that of the highly skewed S. Using this join algorithm for nonuniform relation will result in a very bad efficiency because a lot of these dummy reads will result in keeping many nonboundry processors in the mesh idle. Table 8.10 shows the comparison between using join algorithm for relations with uniform data distribution and using join algorithm for relation with nonuniform data distribution, both in the join of relations of nonuniform data distribution.

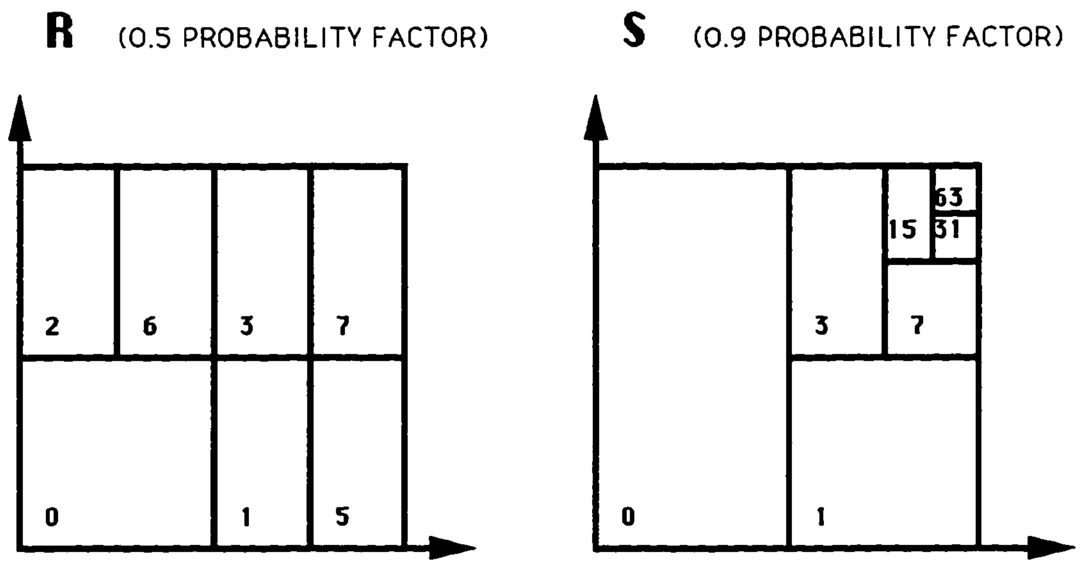


Figure 8.1: R is nearly uniform while S is highly skewed. The partition numbers in R range between 0 and 7 while that of S range between 0 and 63 though, both relations have 7 partitions each.

Probability factor	Ratio
0.3	731.4
0.4	8.7
0.5	0.9
0.6	8.3
0.7	719.3

Table 8.10 : Effect on the ratio of uniform algorithm efficiency over nonuniform algorithm efficiency when both are used to join nonuniform relations.

CHAPTER 9

CONCLUSION AND FUTURE WORK

The common need of join operation in many applications makes it important to find more efficient algorithms. In this thesis two new join algorithms for parallel computers are presented. Both are based on IBGF. They are join algorithms for uniform relations and for nonuniform relations. The main factors affecting the efficiencies were the number of processors, the number of secondary storage elements, the number of processors to storage elements connections, the number of aggregate-classes and the number of partitions in each aggregate-class.

Increase in the number of processors increases the speedup. However, the efficiency which is speedup divided by number of processors, decreases. The rate of decrease in efficiency is a subject of further analysis under say different architectures and communication protocols.

The increase in the number of aggregate-classes has shown decrease in efficiency. This is mainly due the relatively fixed sequential processing required per aggregate-class. Also high number of aggregate-class means aggregate-classes of smaller sizes. Which implies lower parallelism because some processors will remain idle. Further research is required to improve parallelism across the aggregate-classes.

The increase in the number of storage elements and their connections with the processors, has improved the efficiency as expected. This is due to higher parallelism in I/O.

The study needs to be enhanced further to include different architectures and the protocols. Our choice of mesh is not based on a particular yield of this topology. However mesh topology fits the topology of IBGF. Its performance should be compared with other architectures before any thing more can be said about it.

Finally, load balancing techniques imbedded into the distribution algorithms need further study.

REFERENCES

[*ALE*87] Alexander H., "The multiple Prime Random Number Generator," ACM Transaction on Mathematical Software, vol. 13, No. 4, December 1987, pp 368-381.

[*BAR*87] Baru, C. and Frieder, O., "Implementing relalational database operations in a cube-connected multicomputer," Third International Conference on Data Engineering, IEEE, 1987, pp 36-43.

[*BIT*83] Bitton, D.H., Boral, H. and Witt, D.J., "Parallel algorithms for execution of relational database operation," ACM Transaction on Database Systems, 8(3), September 1983.

[*CHA*87] Chaitanya K. B. and Ophir F., "Implementing relational database operations in a cube-connected multicomputer systems," IEEE, May 1987, pp 36-43.

[*DAT*83] C. J. Date, *An Introduction to Database Systems.*, Reading, Ma: Addison-Welsey, Vol II, 1983.

[*DEW*81] DeWitt, D. J. and Hawthrone, P. B., "A performance Evaluation of Database Machine Architectures," Int'l Conference on Very Large Databases, 1981.

[*DEW*84] Dewitt, D. Katz, R., Olken, F. Shapiro, D., Stonebraker, M. and Wood, D. , "Implementation techniques for main memory database systems," Proceeding of SIGMOD conference, 1984.

[*DEW*85] Dewitt, D. and Gerber, R., "Multiprocessor hash based join algorithms," Proceedings of VLDB (Stockholm), 1985.

[*DU*82] H. C. Du and J. S. Sobelewski, "Disk allocation for cartesian product files on multiple disk systems," ACM Transaction on Database Systems, vol. 7, No. 1, pp 157-168, March 1982.

[*FEN*81] T. Y. Feng, "A survey of interconnection networks," Computer, December 1981, pp12-27

[*GAJ*85] D. Gajski and J. K. Peir, "Essential issues in multiprocessor systems," Computer, June 1985, pp 9-27.

[*HAY*86] Hayes, J. P. et aL, "Architecture of a hypercube supercomputers," Proceedings Int'l Conference on Parallel Proc., 1986.

[*INT*85] Intel iPSC Data Sheet, Order No. 280101-001, 1985.

[*KIM*84] W. Kim, D. Gajski and D. Kuck, "A parallel pipelined relational query processor," ACM Transaction on Database Systems, vol. 9, No. 2, pp 214-242, June 1984.

[*KRI*88] Krishna P. M. and Stanley Y. W. Su, "An evaluation of relational join algorithms in a pipelined query processing environment," IEEE Transaction on Software Engineering, vol. 14, No. 6, June 1988.

[*MEN*83] Menon M. J. and Hsiqo D. K., *Design and analysis of join operations of database machines, in advanced database machine architecture*, Prentice-Hall Inc., Englewood Cliffs, N. J. 07632, 1983, pp203-255.

[*MER*81] Merrett T. H., "Why sort merge gives the best implementation of the natural join," SIG-MOD Rec. 13, 2, 1981, pp 39-51.

[*MIC*88] Michel D. and Christoph S., "Synchronization, Coherence and event ordering in multiprocessors," Computer, Feb. 1988 pp 9-21.

[*OUK*85*a*] Ouksel M., "The interpolation based grid file: a multidimensional dynamic order-preserving partitioning scheme," Proc. ACM SIGMOD Symp on Principles of Database Systems, Portland, Oregon, march 1985.

[*OUK*85*b*] Ouksel M. and Ozkarahan E., "Dynamic order-preserving partitioning for database machines," Proc. Eleventh Int. Conference on Very Large Database Systems, Aug. 1985.

[*OMI*88] E. Omiecinski and E. Tien, "Hash-based and index-based join algorithms for cube and ring connected multicomputers," Technical report, GIT-ICS, September 1988.

[*OZK*88] E. A. Ozkarahan and C. H. Bozsahin, "Join strategies using dataspace partitioning," New Generation Computing, June 1988, pp 19-39.

[*PAT*87] Patrick V., "Join indices," ACM Transaction on Database Systems, vol. 12, No. 2, June 1987, pp218-246.

[*PRE*81] Preparata F. P. and Vuillemin J., "The cube-connected cycles: a versatile network for parallel computation," Comm. of A.C.M., vol. 24, No. 5, May 1981.

[*RIC*87] Richardson J., La H. and Mikkilineni K., "Design and evaluation of parallel pipelined join algorithms," SIGMOD Management of Data Conference, ACM, 1987.

[*SAN*88] Sanjay R., Youngju W. and Sartaj S., "Programming a hypercube multicomputer," IEEE Software, Sep. 1988.

[*SEI*85] Seitz C., "The cosmic cube," Communications of ACM, 28, 1 Jan. 1985

[*SHA*86] Shapiro L. "Join processing in database systems with large main memories," ACM TODS, 11, 3, Sep. 1986, 239-264.

[*TIE*87] Tien E., Omieciniski E., "Join algorithms for a cube-connected multicomputer," GIT-ICS-87/37, 1987.

[*VAL*84] Valduricz P., Gardarin G., "Join and semi-join algorithms for a multiprocessor database machine," ACM TODS, 9, 1, Mar. 1984, pp 131-161.

APPENDIX A

Program listing for IBGF simulator

(MODULE 1)

```
(* ----------------------------------------------------
                  IBGF SIMULATOR
   ---------------------------------------------------- *)
PROGRAM IBGF(INPUT, IBGFIN,OUTPUT,IBGF);
CONST
  BITLIMIT  = 3000;
  RECLIMIT  = 31;
  DATALIMIT = 73;
  STKLIMIT  = 100;
  AXES      = 2;
  VEC1SIZE   = 3000;
  VEC2SIZE   = 10;
TYPE
  VECVALUE  =  0..1 ;
  DIRPNTR = @DIRBUK;
  DIRREC  = RECORD
     DONE : BOOLEAN;
     SIZE : INTEGER;
     NEXT : DIRPNTR;
     LVL  : INTEGER;
  END;
  DIRBUK = RECORD
     STO     : INTEGER;
     RECLVL  : INTEGER;
     MAXPRT  : INTEGER;
     BITSIZE : INTEGER;
     REC     : ARRAY(.0..RECLIMIT.) OF DIRREC;
  END;
  BITSARRAY = ARRAY(.1..BITLIMIT.) OF INTEGER;
  STKREC     = RECORD
    PNTR     : DIRPNTR;
    LVL      : INTEGER;
    DISK     : INTEGER;
  END;
  STKTYPE    = ARRAY(.1.. STKLIMIT.) OF STKREC;
  BINVECTOR1 = ARRAY(. 1 .. VEC1SIZE .) OF VECVALUE;
  BINVECTOR2 = ARRAY(. 1 .. VEC2SIZE .) OF VECVALUE;
VAR
  IBGF          : TEXT;
  IBGFIN        : TEXT;
  BINARRY       : BITSARRAY;
  ROOT          : DIRPNTR;
  N,SP          : INTEGER;
  MAXLVL        : INTEGER;
  IX,IY         : INTEGER;
  TOTDISK       : INTEGER;
  INTR          : INTEGER;
  RANGE         : INTEGER;
  BITSIZE       : INTEGER;
  P, YFL        : REAL;
(* ----------------------------------------------------
                  RANDOM NUMBERS GENERATOR
  ---------------------------------------------------- *)
  PROCEDURE RANDOM (VAR IX   :   INTEGER;
                        VAR IY    :   INTEGER;
                        VAR YFL   :   REAL);
 CONST
   MULT = 65539;
 BEGIN
   IY := IX * MULT;
   IF (IY < 0) THEN
     IY := IY + 2147483647 + 1;
   YFL := IY;
   YFL := YFL*0.4656613*0.00001*0.0001
 END;
(* ---------------------------------------------------
                REAL TO BINARY NUMBER CONVERTER
  -------------------------------------------------- *)
 PROCEDURE GETBINARRY( VAR BINARRY : BITSARRAY;
                           P       : REAL;
                           BITSIZE : INTEGER );
 VAR
   I : INTEGER;
   Q : REAL;
 BEGIN
   Q := 1 - P;
   FOR I := 1 TO BITSIZE DO
   BEGIN
     RANDOM(IX, IY, YFL);
     IX := IY;
     IF ( YFL >= Q ) THEN
       BINARRY(.I.) := 1
     ELSE
       BINARRY(.I.) := 0;
   END;
 END;
(* ---------------------------------------------------
              BINARY TO INTEGER NUMBER CONVERTER
  -------------------------------------------------- *)
 PROCEDURE BINTOINT(VAR PRT : INTEGER;
                    BITSIZE : INTEGER;
                    BINARRY : BITSARRAY);
 VAR
   N, I : INTEGER;
 BEGIN
   N := BITSIZE;
   IF (BITSIZE > RANGE) THEN
     N := RANGE;
   PRT := 0;
   FOR I := 1 TO N DO
     PRT := PRT + TRUNC(EXP((I-1)*LN(2))+0.5)*BINARRY(.I.);
 END;
(* ---------------------------------------------------
   FUNCTION COMPUTING THE MINIMUM NUMBER OF BITS
   THAT AN INTEGER NUMBER CAN ASSUME
  -------------------------------------------------- *)
 FUNCTION SIZEOFBITS(K : INTEGER) : INTEGER;
 VAR
```

```
      M, I : INTEGER;
    BEGIN
      M := 1;
      WHILE (K >= 2) DO
      BEGIN
        M := M + 1;
        K := K DIV 2;
      END;
      SIZEOFBITS := M;
    END;
(* ---------------------------------------------------
           THE NEXT EMBEDDED PARTITION NUMBER FINDER
   --------------------------------------------------- *)
    FUNCTION EMBDED(K,L,M : INTEGER): INTEGER;
    VAR
      I : INTEGER;
    BEGIN
      M := L - RANGE*M;
      EMBDED := K + TRUNC(EXP(M*LN(2))+0.5);
    END;
(* ---------------------------------------------------
           THE NEXT EMBEDDER PARTITION NUMBER FINDER
   --------------------------------------------------- *)
    FUNCTION EMBDER(K1 : INTEGER): INTEGER;
    VAR
     TMP, I, LOG : INTEGER;
    BEGIN
      TMP := K1;
      LOG := 0;
      WHILE(TMP >= 2) DO
      BEGIN
        TMP := TMP DIV 2;
        LOG := LOG + 1;
      END;
      TMP := 1;
      FOR I := 1 TO LOG DO
        TMP := TMP*2;
      EMBDER := K1 - TMP;
    END;
(* ---------------------------------------------------
         ROOT OF THE IBGF INITIALIZER.
   --------------------------------------------------- *)
    PROCEDURE INITIALIZE(VAR ROOT : DIRPNTR;
                         VAR RANGE, BITSIZE : INTEGER);
    VAR
      I : INTEGER;
    BEGIN
      NEW(ROOT);
      ROOT@.MAXPRT  := 0;
      ROOT@.BITSIZE := 1;
      FOR I := 0 TO RECLIMIT DO
      BEGIN
        ROOT@.REC(.I.).DONE := FALSE;
        ROOT@.REC(.I.).SIZE := 0;
      ROOT@.REC(.I.).NEXT := NIL;
      ROOT@.REC(.I.).LVL  := 0;
    END;
      ROOT@.REC(.0.).SIZE := 1;
    I := 1;
    RANGE := 0;
    WHILE ( I < (RECLIMIT + 1)) DO
    BEGIN
      I := I * 2;
      RANGE := RANGE + 1;
    END;
    BITSIZE := RANGE;
  END;
(* -----------------------------------------------------
       CREATOR OF A NEW DIRECTORY PARTITION
   ----------------------------------------------------- *)
  PROCEDURE MAKENEWBUK( VAR CURPNTR : DIRPNTR;
                            P       : REAL;
                            K       : INTEGER;
                            LEVEL   : INTEGER);
  VAR
    NXTPNTR : DIRPNTR;
    I, M1, M2 : INTEGER;
  BEGIN
    NEW(NXTPNTR);
    NXTPNTR@.MAXPRT  := 1;
    NXTPNTR@.BITSIZE := LEVEL + 1;
    NXTPNTR@.RECLVL  := CURPNTR@.RECLVL + 1;
    FOR I := 0 TO RECLIMIT DO
    BEGIN
      NXTPNTR@.REC(.I.).DONE := FALSE;
      NXTPNTR@.REC(.I.).SIZE := 0;
      NXTPNTR@.REC(.I.).NEXT := NIL;
      NXTPNTR@.REC(.I.).LVL  := 0;
    END;
    M1 := TRUNC((DATALIMIT + 1) * P);
    M2 := TRUNC((DATALIMIT + 1) * (1.0 - P));
    IF (M2 > M1) THEN
      M1 := M1 + 1
    ELSE IF(M1 > M2 ) THEN
      M2 := M2 + 1;
    NXTPNTR@.REC(.1.).SIZE := M1;
    NXTPNTR@.REC(.0.).SIZE := M2;
    NXTPNTR@.REC(.0.).LVL  := LEVEL + 1;
    NXTPNTR@.REC(.1.).LVL  := LEVEL + 1;
    CURPNTR@.REC(.K.).NEXT := NXTPNTR;
  END;
(* -----------------------------------------------------
       SPLITTER OF AN OVERFLEW DIRECTORY PARTITION
   ----------------------------------------------------- *)
  PROCEDURE DIVIDEREC( P : REAL;
                       VAR PNTR : DIRPNTR;
                       PRT1, PRT2 : INTEGER);
  VAR
```

```
    M1,M2 : INTEGER;
  BEGIN
(*  WRITELN('PPPPP', PRT1, PRT2) *)
    M1 := TRUNC((DATALIMIT + 1) * P);
    M2 := TRUNC((DATALIMIT + 1) * (1.0 - P));
    IF (M2 > M1) THEN
      M1 := M1 + 1
    ELSE IF(M1 > M2 ) THEN
      M2 := M2 + 1;
    PNTR@.REC(.PRT2.).SIZE := M1;
    PNTR@.REC(.PRT1.).SIZE := M2;
    M1 := PNTR@.REC(.PRT1.).LVL + 1;
    PNTR@.REC(.PRT1.).LVL  := M1;
    PNTR@.REC(.PRT2.).LVL  := M1;
    IF (PNTR@.MAXPRT < PRT2) THEN
    BEGIN
      PNTR@.MAXPRT :=  PRT2;
      PNTR@.BITSIZE :=  M1;
    END;
  END;
(* ------------------------------------------------------
       CALCULATOR OF THE POWER OF 2 OF AN INTEGER NUMBER.
   ------------------------------------------------------ *)
  FUNCTION POWEROF2(K:INTEGER) : INTEGER;
  VAR
    I,T : INTEGER;
  BEGIN
    T := 1;
    FOR I := 1 TO K DO
     T := T*2;
   POWEROF2 := T;
  END;
(* ---------------------------------------------------
     CALCULATOR OF THE NUMBER OF BITS THAT THE
     RANDOMLY GENERATTED DATA TUPLE ASSUMES.
   ---------------------------------------------------- *)
  PROCEDURE GETRANG( MAXLVL, LVL, INTR : INTEGER;
                     VAR K : INTEGER);
  VAR
    KK : INTEGER;
  BEGIN
    IF ((MAXLVL MOD 2 ) = 1) THEN
      K := (MAXLVL + AXES - 1 - LVL) DIV AXES
    ELSE
      K := (MAXLVL - LVL) DIV AXES;
  END;
(* ---------------------------------------------------
     INTEGER TO BINARY NUMBER CONVERTER
   ---------------------------------------------------- *)
   PROCEDURE INTTOBIN(INTNUM : INTEGER;
                      RANGE  : INTEGER;
                      VAR BINVECTOR : BINVECTOR2);
   VAR
     I : INTEGER;
   BEGIN
     FOR I := 1 TO RANGE DO
     BEGIN
       IF ( (INTNUM MOD 2 ) = 1 ) THEN
          BINVECTOR(.I.) := 1
       ELSE
          BINVECTOR(.I.) := 0;
       INTNUM := INTNUM DIV 2;
     END;
   END;
(* -----------------------------------------------------
     COMPARATOR OF A PARTITION NUMBER TO  A JOIN-CLASS
   ----------------------------------------------------- *)
    FUNCTION COMPARE(JCBITS : BINVECTOR1;
                     PRT    : INTEGER;
                     LVL    : INTEGER;
                     PNTR   : DIRPNTR) : BOOLEAN;
    VAR
      PRTBINVEC   : BINVECTOR2;
      TOTPRTSIZE : INTEGER;
      MBR         : BOOLEAN;
      I,K, M      : INTEGER;
    BEGIN
      TOTPRTSIZE := (LVL-1) * RANGE;
      M := PNTR@.REC(.PRT.).LVL - TOTPRTSIZE;
      INTTOBIN(PRT, M, PRTBINVEC);
      K := (TOTPRTSIZE + AXES - 1) DIV AXES;
      MBR := TRUE;
      IF ( (NOT ODD(LVL)) AND (ODD(RANGE))) THEN
        I := 1
      ELSE
        I := 0;
      WHILE (MBR AND (I < M) ) DO
      BEGIN
        I := I + 1;
        IF ( (I + TOTPRTSIZE) MOD AXES = 1) THEN
        BEGIN
        K := K + 1;
       IF (PRTBINVEC(.I.)<>JCBITS(.K.)) THEN
          MBR := FALSE;
        END;
      END;
      COMPARE := MBR;
    END;
(* ----------------------------------------------------- *)
   PROCEDURE POP( VAR STK   : STKTYPE;
                  VAR SP   : INTEGER;
                  VAR LVL : INTEGER;
                  VAR PNTR : DIRPNTR);
   BEGIN
   IF (SP > 0 ) THEN
   BEGIN
     PNTR := STK(.SP.).PNTR;
     LVL  := STK(.SP.).LVL;
```

```
      SP := SP - 1;
    END
    ELSE
      WRITELN(IBGF,'STACK IS EMPTY');
    END;
(* ---------------------------------------------------
      END OF IBGF FORMATION CHECKER
   --------------------------------------------------- *)
      PROCEDURE CHECK(JCBINVEC : BINVECTOR1; JCVECSIZE : INTEGER;
                      VAR FINISHED : BOOLEAN);
      VAR
        I : INTEGER;
      BEGIN
        I := JCVECSIZE;
        FINISHED := TRUE;
        WHILE ( FINISHED AND ( I > 0)) DO
          IF(JCBINVEC(.I.) = 1) THEN
            FINISHED := FALSE
          ELSE
            I := I - 1;
      END;
(* ---------------------------------------------------
                A JOIN-CLASS BUILDER
   --------------------------------------------------- *)
      PROCEDURE MAKEJC(VAR JCBINVEC:BINVECTOR1;
                         K, JCVECSIZE:INTEGER);
      VAR
        I : INTEGER;
        DONE : BOOLEAN;
      BEGIN
        DONE := FALSE;
        I := JCVECSIZE - K;
        WHILE ( (NOT DONE) AND ( I > 0)) DO
          IF (JCBINVEC(.I.) = 0) THEN
          BEGIN
            JCBINVEC(.I.) := 1;
            DONE := TRUE;
          END
          ELSE
          BEGIN
            JCBINVEC(.I.) := 0;
            I := I - 1;
          END;
        END;

(* --------------------------------------------------- *)
    PROCEDURE PUSH( VAR STK : STKTYPE;
                    VAR SP : INTEGER;
                    LVL     : INTEGER;
                    PNTR    : DIRPNTR);
    BEGIN
    IF (SP < STKLIMIT ) THEN
    BEGIN
      SP := SP + 1;
      STK(.SP.).PNTR := PNTR;
      STK(.SP.).LVL  := LVL;
    END
    ELSE
      WRITELN(IBGF,'STACK STK HAS OVERFLOWN');
    END;
(* ----------------------------------------------------
           STORAGE MEDIA ALLOCATOR AND FINDER
   ---------------------------------------------------- *)
  PROCEDURE GETSTORAGE;
  VAR
    SUMPRTS : INTEGER;
    PRTS    : INTEGER;
    M       : INTEGER;
    I       : INTEGER;
    STK     : STKTYPE;
    PNTR    : DIRPNTR;
    TMPPNTR : DIRPNTR;
    STOR    : INTEGER;
    LVL     : INTEGER;
    SP      : INTEGER;
    EPS     : INTEGER;
    MXBTSIZE: INTEGER;
    STRLVL  : INTEGER;
  BEGIN
    STOR := 0;
    PNTR := ROOT;
    STRLVL := 0;
    PRTS := 0;
    SUMPRTS := 0;
    LVL := 0;
    SP := 0;
    MAXLVL := 0;
    EPS := DATALIMIT DIV 3;
    M := N DIV (TOTDISK - STOR);
    PUSH(STK,SP, LVL+1,PNTR);
    WHILE(SP > 0) DO
    BEGIN
      POP(STK,SP,LVL,PNTR);
      IF (MAXLVL < LVL) THEN
      BEGIN
         MXBTSIZE := PNTR@.BITSIZE;
         TMPPNTR := PNTR;
         MAXLVL := LVL;
      END
      ELSE IF (MAXLVL = LVL) THEN
        IF (PNTR@.BITSIZE > MXBTSIZE ) THEN
          MXBTSIZE := PNTR@.BITSIZE;
      IF(PRTS >= (M-EPS)) THEN
      BEGIN
        STOR := STOR + 1;
        STOR := STOR MOD TOTDISK;
        STRLVL := 1;
        PNTR@.STO := STOR;
        SUMPRTS := SUMPRTS + PRTS;
        PRTS := 0;
        M := (N - SUMPRTS) DIV (TOTDISK - STOR);
        FOR I := 0 TO RECLIMIT DO
        BEGIN
  IF ((PNTR@.REC(.I.).NEXT = NIL) AND (PNTR@.REC(.I.).SIZE > 0)) THE
          BEGIN
            PRTS := PRTS + PNTR@.REC(.I.).SIZE;
          END
          ELSE IF( PNTR@.REC(.I.).NEXT <> NIL ) THEN
            PUSH(STK,SP,LVL+1,PNTR@.REC(.I.).NEXT);
        END;
      END
      ELSE
      BEGIN
        STRLVL := STRLVL + 1;
        PNTR@.STO := STOR;
        FOR I := 0 TO RECLIMIT DO
        BEGIN
 IF ((PNTR@.REC(.I.).NEXT = NIL) AND (PNTR@.REC(.I.).SIZE > 0)) THEN
      BEGIN
            PRTS := PRTS + PNTR@.REC(.I.).SIZE;
      END
          ELSE IF( PNTR@.REC(.I.).NEXT <> NIL ) THEN
            PUSH(STK,SP,LVL+1,PNTR@.REC(.I.).NEXT);
        END;
      END;
    END;
    MAXLVL := MXBTSIZE;
 END;
(* ----------------------------------------------------
       PARTITIONS OF ALL JOIN-CLASSES FINDER
   ---------------------------------------------------- *)
   PROCEDURE GETALLJOINCLASSES;
   TYPE
     DREC = RECORD
       PRT  : INTEGER;
       DLVL : INTEGER;
       STR  : INTEGER;
       RNG  : INTEGER;
     END;
   VAR
     NN,III : INTEGER;
     JCBINVEC  : BINVECTOR1;
     JCVECSIZE : INTEGER;
     JCCLASS : INTEGER;
     TOTAXISINTR : INTEGER;
     DIRTAB : ARRAY(. 1 .. 500 .) OF DREC;
     DATTAB : ARRAY(. 1 .. 1500 .) OF DREC;
     M1, M2 : INTEGER;
     DISK   : INTEGER;
     STK    : STKTYPE;
     PNTR : DIRPNTR;
     SP,L, I, LVL, STR,K,M : INTEGER;
    FINISHED, MBR : BOOLEAN;
  BEGIN
   III := 0;
   INTR := 6;
   JCVECSIZE := ( MAXLVL + AXES - 1) DIV AXES;
   TOTAXISINTR := POWEROF2(JCVECSIZE);
 WRITELN(IBGF,TOTAXISINTR:9,TOTDISK:4,N:9,P:7:2,RECLIMIT:4,MAXLVL:7
   FOR I:= (JCVECSIZE+1) DOWNTO 1 DO
     JCBINVEC(.I.) := 0;
   FINISHED := FALSE;
   WHILE ( NOT FINISHED ) DO
   BEGIN
    PNTR := ROOT;
    SP:= 0;
    LVL := 0;
    DISK := 0;
    M := 0;
    M1 := 0;
    M2 := 0;
    PUSH(STK,SP,LVL+1,PNTR);
    WHILE(SP <> 0) DO
    BEGIN
      POP(STK,SP,LVL,PNTR);
      DISK := PNTR@.STO;
      FOR I := 0 TO RECLIMIT DO
        IF (PNTR@.REC(.I.).SIZE <> 0) THEN
        BEGIN
          MBR := COMPARE(JCBINVEC, I, LVL,PNTR);
          IF (MBR) THEN
          BEGIN
            IF (PNTR@.REC(.I.).LVL > M) THEN
              M := PNTR@.REC(.I.).LVL;
            GETRANG( MAXLVL, PNTR@.REC(.I.).LVL,INTR,K);
            IF(PNTR@.REC(.I.).NEXT <> NIL) THEN
            BEGIN
              IF( NOT PNTR@.REC(.I.).DONE ) THEN
              BEGIN
                 PNTR@.REC(.I.).DONE := TRUE;
                 M1 := M1 + 1;
                 DIRTAB(.M1.).PRT := I;
                 DIRTAB(.M1.).DLVL := LVL;
                 DIRTAB(.M1.).STR := DISK;
                 NN := POWEROF2(K) + III;
                 DIRTAB(.M1.).RNG := NN;
              END;
              PUSH(STK,SP,LVL+1,PNTR@.REC(.I.).NEXT);
            END
            ELSE
              IF( NOT PNTR@.REC(.I.).DONE ) THEN
              BEGIN
                 PNTR@.REC(.I.).DONE := TRUE;
                 M2 := M2 + 1;
                 DATTAB(.M2.).PRT := I;
                 DATTAB(.M2.).DLVL := LVL;
                 DATTAB(.M2.).STR := DISK;
                 NN := POWEROF2(K) + III;
                 DATTAB(.M2.).RNG := NN;
              END;
        END;
      END;
    END;
    GETRANG(MAXLVL, M,INTR, K);
    NN := POWEROF2(K)+III;
    WRITELN(IBGF,III:12,'  ',NN:12);
    III := NN;
    FOR I := 1 TO M1 DO
    BEGIN
    WITH DIRTAB(.I.) DO
      WRITELN(IBGF,PRT:12, DLVL:4, STR:9, RNG:12, 1:4);
    END;
    FOR I := 1 TO M2 DO
    BEGIN
    WITH DATTAB(.I.) DO
      WRITELN(IBGF,PRT:12, DLVL:4, STR:9, RNG:12, 0:4);
    END;
    WRITELN(IBGF,-1:9,-1:9,-1:9,-1:9,-1:9);
    MAKEJC(JCBINVEC, K, JCVECSIZE);
    CHECK(JCBINVEC, JCVECSIZE, FINISHED);
   END;
  END;
(* -------------------------------------------------------
              IBGF BULILDER
  ---------------------------------------------------- *)
 PROCEDURE INSERT;
 VAR
   PNTR           : DIRPNTR;
   LEVEL, M, I    : INTEGER;
   BGPRT, BTSIZE  : INTEGER;
   PRT1, PRT2     : INTEGER;
   DONE           : BOOLEAN;
   Q              : REAL;
 BEGIN
   MAXLVL := 1;
   Q := 1-P;
   SP := 0;
   PNTR := ROOT;
   PNTR@.RECLVL := 1;
   FOR I := 2 TO N DO
   BEGIN
     PNTR := ROOT;
     DONE := FALSE;
     WHILE ( NOT DONE ) DO
     BEGIN
       BTSIZE := PNTR@.BITSIZE;
       BGPRT :=  PNTR@.MAXPRT;
       GETBINARRY(BINARRY, P, BTSIZE);
       BINTOINT(PRT1, BTSIZE, BINARRY);
       WHILE (NOT DONE) DO
```

```
        BEGIN
          IF ( PRT1 > BGPRT) THEN
            PRT1 := EMBDER(PRT1)
          ELSE IF(PNTR@.REC(.PRT1.).SIZE = 0 ) THEN
            PRT1 := EMBDER(PRT1)
          ELSE
            DONE := TRUE;
        END;
        DONE := FALSE;
        IF(PNTR@.REC(.PRT1.).NEXT = NIL) THEN
        BEGIN
          M := PNTR@.REC(.PRT1.).SIZE + 1;
          PNTR@.REC(.PRT1.).SIZE := M;
          IF ( M > DATALIMIT) THEN
          BEGIN
            LEVEL := PNTR@.REC(.PRT1.).LVL;
            PRT2 := EMBDED(PRT1, LEVEL,PNTR@.RECLVL - 1);
            IF ( PRT2 > RECLIMIT ) THEN
               MAKENEWBUK(PNTR, P, PRT1,LEVEL)
            ELSE
               DIVIDEREC(P, PNTR, PRT1, PRT2);
          END;
          DONE := TRUE;
        END
        ELSE
          PNTR := PNTR@.REC(.PRT1.).NEXT;
      END;
    END;
(* ---------------------------------------------------
               THE MAIN ROUTINE
   --------------------------------------------------- *)
BEGIN
  RESET(IBGFIN, "NIBG22 DATA");
  REWRITE(IBGF, "IBGF OUT2");
  INITIALIZE(ROOT,RANGE, BITSIZE);
  READLN(IBGFIN,N, P, TOTDISK, IX);
  INSERT;
  GETSTORAGE;
  GETALLJOINCLASSES;
  TRAVERS;
END.
```

APPENDIX B

Program listing for Join Algorithm for uniform relations based on IBGF

(MODULE 2)

```
C-----------------------------------------------------------
C   SIMULATOR OF A JOIN OPERATION FOR UNIFORM RELATIONS
C   BASED ON IBGF.
C-----------------------------------------------------------
C$ STAT=99999999999,O=2000000,TI=0,STAC=999999
      INTEGER DISKS, VPRS, HPRS
      INTEGER BUS(36), QTYBUS, OLDBUS
      INTEGER RPGS, RVPGS, RHPGS
      INTEGER SPGS, SVPGS, SHPGS
      INTEGER AXES, RATIO
      INTEGER FR(64), FS(64), RSHR, SSHR
      REAL TMPRS(16,16), TMBUS(36), RDTIME, MPTIME
      REAL TMSTR(0:63)
      LOGICAL FLAG

      OPEN(4, FILE = 'UNI INP', STATUS = 'OLD')
C     OPEN(7, FILE = 'IBGF OUT1', STATUS = 'OLD')
C     OPEN(8, FILE = 'IBGF OUT2', STATUS = 'OLD')
      OPEN(9, FILE = 'UNI OUT', STATUS = 'UNKNOWN')
      READ(4,*) VPRS, HPRS, QTYBUS, DISKS, RPGS, SPGS, TMNON, PROB
C     READ(4,*) VPRS, HPRS, QTYBUS, DISKS, RPGS, SPGS
      IF ( SPGS .LT. RPGS) CALL SWAPI(SPGS, RPGS)
      OLDBUS = QTYBUS
      T = 73
      TC = 28*T
      TR = 28*T + 8000
      TJ = 4*T*T
      II = 16
      JJ = 16
      IF (QTYBUS .GT. VPRS) QTYBUS = VPRS
      CALL COMP(RPGS, RVPGS, RHPGS)
      CALL COMP(SPGS, SVPGS, SHPGS)
      CALL DVD(FR, RVPGS, VPRS, RSHR)
      CALL DVD(FS, SVPGS, HPRS, SSHR)
      CALL ASNBUS(BUS, QTYBUS, HPRS, VPRS)
      CALL INIT1D(TMBUS,  1, QTYBUS, 1.0)
      CALL INIT1D(TMSTR, 0, DISKS, 1.0)
      CALL INIT2D(TMPRS, II, JJ, VPRS, HPRS, 0.0)
      IRV = RVPGS/DISKS
      IRH = RHPGS/DISKS
      ISV = SVPGS/DISKS
      ISH = SHPGS/DISKS
      RATIO = SHPGS/RHPGS
      NJ    = 0
      IRAT  = 0
      IS    = 0
      IR    = 0
      IHR   = 0
      IHS   = 0
 11   IF( IHR .LT. RHPGS) THEN
        IRAT  = 0
        IS    = 0
        IR    = 0
        CALL RDSR(TMPRS,II,JJ,FR,RSHR,IR,IHR,FS,SSHR,TMSTR,
     *  DISKS,IS,IHS,TMBUS,QTYBUS,BUS,VPRS,HPRS,TR,
     *  IRV, IRH,ISV,ISH)
        CALL GETFHR(TMPRS,II,JJ,IR,IHR,RSHR,VPRS,HPRS,FR,TC)
        CALL GETS(TMPRS,II,JJ,IS,IHS,RSHR,VPRS,HPRS,FS,TC)
        CALL GETSHR(TMPRS,II,JJ,IR,IHR,RSHR,VPRS,HPRS,FR,TC)
        CALL JOIN(TMPRS,II,JJ,IR,IHR,IS,IHS,RSHR,SSHR,VPRS,HPRS,
     *  FR, FS,NJ,TJ)
 22     IS = IS + 1
        IF ( IS .LT. SSHR) THEN
          CALL RDS(FS,SSHR,IS, IHS,BUS,TMBUS,QTYBUS,
     *    TMSTR, DISKS, TMPRS,II,JJ, VPRS, HPRS, TR,ISV, ISH)
          CALL GETS(TMPRS,II,JJ,IS,IHS,SSHR,VPRS,HPRS,FS,TC)
          CALL JOIN(TMPRS,II,JJ,IR,IHR,IS,IHS,RSHR,SSHR,VPRS,HPRS,
     *    FR, FS,NJ,TJ)
          GOTO 22
        ELSE
          IRAT = IRAT + 1
          IHS = IHS + 1
          FLAG = .FALSE.
          IF (IRAT .LT. RATIO) THEN
             IS = -1
             FLAG = .TRUE.
          ENDIF
          IF(FLAG) GOTO 22
        ENDIF
 33     IR = IR + 1
        IF( IR .LT. RSHR) THEN
          CALL RDR(FR,RSHR,IR,IHR,BUS,TMBUS,QTYBUS,
     *    TMSTR, DISKS, TMPRS,II,JJ, VPRS, HPRS,TR, IRV, IRH)
          CALL GETR(TMPRS,II,JJ,IR,IHR,RSHR,VPRS,HPRS,FR,TC)
          DO 44 I = 1, RATIO
          KK = IHS - (RATIO - I)
          DO 44 J = 0, SSHR-1
          CALL JOIN(TMPRS,II,JJ,IR,IHR,J,KK,RSHR,SSHR,VPRS,HPRS,
     *    FR, FS,NJ,TJ)
 44       CONTINUE
          GOTO 33
        ELSE
          IHR = IHR + 1
          GOTO 11
        ENDIF
      ELSE
        UPT  = (SPGS + RPGS)*TR + NJ*TJ
        TM   = MPTIME(TMPRS, II,JJ, VPRS, HPRS)
        S = UPT/TM
        EFF = UPT/(TM * VPRS * HPRS)*100.
        WRITE(9,2) VPRS, HPRS, QTYBUS, DISKS, PROB,
     +  TM/TMNON, UPT/TM, UPT/(TM * VPRS * HPRS)*100
      ENDIF
  2   FORMAT(' ',I4,I4,I4,I4,F5.2,F12.2,2X,F6.2,2X,F6.2)
C 2     FORMAT(' ',I6,I6,I6,I6,I6,I6,F8.2,2X,F8.2)
      END
C-----------------------------------------------------------
C     ASSIGNER OF S-P-CONNECTIONS TO A PROCESSOR
```

```
C-----------------------------------------------------------
      SUBROUTINE ASNBUS(BUS, QTYBUS, VPRS, HPRS)
      INTEGER QTYBUS , VPRS, HPRS
      INTEGER BUS(VPRS + HPRS)
      K1 = VPRS/QTYBUS
      K2 = HPRS/QTYBUS
      M = 1
      DO 10 I = 1, VPRS
        BUS(I) = M
        IF (MOD(I, K1) .EQ. 0) M = M + 1
 10   CONTINUE
      M = 1
      DO 20 J = 1, HPRS
        BUS(J + VPRS) = M
        IF (MOD(J, K2) .EQ. 0) M = M + 1
 20   CONTINUE
      END
C-----------------------------------------------------------
C      DIVIDER OF AGGREGATE-CLASS PARTITIONS AMONGEST PROCESSORS
C-----------------------------------------------------------
      SUBROUTINE DVD(F,CNT,DIM1,SHR)
      INTEGER DIM1
      INTEGER F(DIM1),CNT, SHR
      SHR = CNT/DIM1
      M = 0
      DO 10 I = 1, CNT, SHR
        M = M + 1
        F(M) = I
 10   CONTINUE
      END
C-----------------------------------------------------------
C      THE ACTUAL JOIN OPERATION SIMULATOR
C-----------------------------------------------------------
      SUBROUTINE JOIN(TMPRS,II,JJ,IR,IHR,IS,IHS,SHRR,SHRS,VPRS,HPRS
     * FR, FS,NJ,TJ)
      INTEGER VPRS, HPRS
      INTEGER FR(VPRS), FS(HPRS)
      REAL TMPRS(II,JJ)
      INTEGER SHRR, SHRS
      DO 10 I = 1, VPRS
      DO 10 J = 1, HPRS
         M =  FR(I)+IR
         N =  FS(J)+IS
         TMPRS(I,J) = TMPRS(I,J) + TJ
         NJ = NJ + 1
10    CONTINUE
 2    FORMAT(' ',2X,F7.0,2X,'P(',I1,',',I1,')',5X,'R',I4,A,'S',I4)
      END
C-----------------------------------------------------------
C     ACCEPTOR OF AN S PARTITION FROM A NEIGHBOURING PROCESSOR
C-----------------------------------------------------------
      SUBROUTINE GETS(TMPRS,II,JJ,IS,IHS,SHRS,VPRS,HPRS,FS,TC)
      INTEGER VPRS, HPRS,SHRS
      REAL TMPRS(II,JJ)
      INTEGER FS(HPRS)
      DO 10 I = 2, VPRS
      DO 10 J = 1, HPRS
         IF(TMPRS(I-1,J) .LT. TMPRS(I,J)-TC) THEN
           TMPRS(I-1,J) = TMPRS(I,J) - TC
         ELSE
           TMPRS(I,J) = TMPRS(I-1,J) + TC
         ENDIF
         M = FS(J) + IS
         TMPRS(I-1,J) = TMPRS(I,J)
         TMPRS(I,J) = TMPRS(I,J) + TC
 10   CONTINUE
 2    FORMAT(' ',2X, F7.0, 2X, 'P(',I1,',',I1,')', 5X,A,I4,1X,A)
      END
C-----------------------------------------------------------
C     ACCEPTOR OF AN R PARTITION FROM A NEIGHBOURING PROCESSOR
C-----------------------------------------------------------
      SUBROUTINE GETR(TMPRS,II,JJ,IR,IHR,SHRR,VPRS,HPRS,FR,TC)
      INTEGER VPRS, HPRS, SHRR
      INTEGER FR(VPRS)
      REAL TMPRS(II,JJ)
      DO 10 I = 1, VPRS
      DO 10 J = 2, HPRS
         IF(TMPRS(I,J-1) .LT. TMPRS(I,J)-TC) THEN
            TMPRS(I,J-1) = TMPRS(I,J) - TC
         ELSE
            TMPRS(I,J) = TMPRS(I,J-1) + TC
         ENDIF
         M = FR(I) + IR
         TMPRS(I,J-1) = TMPRS(I,J)
         TMPRS(I,J) = TMPRS(I,J) + TC
 10   CONTINUE
 2    FORMAT(' ',2X, F7.0, 2X, 'P(',I1,',',I1,')', 5X,A,I4,A)
      END
C--------------------------------------------------------------
C     ACCEPTOR OF AN R PARTITION FROM A NEIGHBOURING PROCESSOR.
C     ONLY PROCESSORS WITH THEIR ROW NUMBER > OR = TO THEIR
C     COLUMN NUMBER ARE INVOLVED.
C--------------------------------------------------------------
      SUBROUTINE GETFHR(TMPRS,II,JJ,IR,IHR,SHRR,VPRS,HPRS,FR,TC)
      INTEGER VPRS, HPRS, SHRR
      INTEGER FR(VPRS)
      REAL TMPRS(II,JJ)
      DO 10 I = 1, VPRS
      DO 10 J = 2, I
         IF(TMPRS(I,J-1) .LT. TMPRS(I,J)-TC) THEN
            TMPRS(I,J-1) = TMPRS(I,J) - TC
         ELSE
            TMPRS(I,J) = TMPRS(I,J-1) + TC
         ENDIF
         M = FR(I) + IR
         TMPRS(I,J-1) = TMPRS(I,J)
         TMPRS(I,J) = TMPRS(I,J) + TC
 10   CONTINUE
 2    FORMAT(' ',2X, F7.0, 2X, 'P(',I1,',',I1,')', 5X,A,I4,A)
      END
C-----------------------------------------------------------------
C     ACCEPTOR OF AN R PARTITION FROM A NEIGHBOURING PROCESSOR.
C     ONLY PROCESSORS WITH THEIR ROW NUMBER < THEIR COLUMN NUMBER
C     ARE INVOLVED.
C-----------------------------------------------------------------
      SUBROUTINE GETSHR(TMPRS,II,JJ,IR,IHR,SHRR,VPRS,HPRS,FR,TC)
      INTEGER VPRS, HPRS, SHRR
      INTEGER FR(VPRS)
      REAL TMPRS(II,JJ)
      DO 10 I = 1, VPRS
      DO 10 J = I+1, HPRS
         IF(TMPRS(I,J-1) .LT. TMPRS(I,J)-TC) THEN
            TMPRS(I,J-1) = TMPRS(I,J) - TC
         ELSE
            TMPRS(I,J) = TMPRS(I,J-1) + TC
         ENDIF
         M = FR(I) + IR
         TMPRS(I,J-1) = TMPRS(I,J)
         TMPRS(I,J) = TMPRS(I,J) + TC
 10   CONTINUE
 2    FORMAT(' ',2X, F7.0, 2X, 'P(',I1,',',I1,')', 5X,A,I4,A)
      END
C-----------------------------------------------------------------
C       READER OF AN S AND AN R PARTITION FROM AN IBGF FILE
C-----------------------------------------------------------------
        SUBROUTINE RDSR(TMPRS,II,JJ,FR,SHRR,IR,IHR,FS,SHRS,TMSTR,
      * DISKS,IS,IHS,TMBUS,QTYBUS,BUS,VPRS,HPRS,TR,
      * IRV, IRH,ISV,ISH)
        INTEGER DISKS, QTYBUS,VPRS, HPRS
        INTEGER FS(64), SHRS, FR(64), SHRR, BUS(VPRS+HPRS)
        INTEGER FNDX(128), SNDX(128), TOTLOC, GETSTR
        REAL TMBUS(QTYBUS), TMSTR(0:DISKS-1),TMPRS(II,JJ)
        REAL LOC(128)
        M = 1
        LOC(M) = TMPRS(1,1)
        FNDX(M) = 1
        SNDX(M) = 1
        DO 10 I = 2, VPRS
          M = M + 1
          LOC(M) = TMPRS(I,1)
          FNDX(M) = I
          SNDX(M) = 1
 10     CONTINUE
        DO 20 I = 2, HPRS
          M = M + 1
          LOC(M) = TMPRS(1,I)
          FNDX(M) = 1
          SNDX(M) = I
 20     CONTINUE
        TOTLOC = M
        CALL SORT(LOC, FNDX, SNDX, TOTLOC)
        DO 30 I = 1, TOTLOC
```

```
          IF (SNDX(I) .EQ. 1 ) THEN
            KI = FNDX(I)
            KBI = BUS(KI)
            KD1 = FR(KI) + IR
            KD2 = GETSTR(KD1, IHR, IRV, IRH, DISKS)
            CALL SYNC(TMBUS(KBI), TMSTR(KD2), LOC(I), TR)
            TMPRS(FNDX(I), SNDX(I)) = LOC(I)
          ENDIF
          IF (FNDX(I) .EQ. 1 ) THEN
            KI = SNDX(I)
            KBI = BUS(KI+ VPRS)
            KD1 = FS(KI) + IS
            KD2 = GETSTR(KD1, IHS, ISV, ISH, DISKS)
            CALL SYNC(TMBUS(KBI), TMSTR(KD2), LOC(I), TR)
          TMPRS(FNDX(I), SNDX(I)) = LOC(I)
          ENDIF
 30     CONTINUE
 2      FORMAT(' ',2X,F7.0,2X,'P(',I1,',',I1,')',5X,'R',I6)
 3      FORMAT(' ',2X,F7.0,2X,'P(',I1,',',I1,')',5X,'S',I6)
        END
C---------------------------------------------------------------
C       READER OF AN R PARTITION FROM AN IBGF FILE
C---------------------------------------------------------------
        SUBROUTINE RDR(FR,SHRR,IR,IHR,BUS,TMBUS,QTYBUS,
     *  TMSTR, DISKS, TMPRS,II,JJ, VPRS, HPRS,TR, IRV, IRH)
        INTEGER HPRS, VPRS, QTYBUS, DISKS
        INTEGER FR(64), SHRR, BUS(HPRS+ VPRS)
        INTEGER FNDX(128), SNDX(128), TOTLOC, GETSTR
        REAL TMBUS(QTYBUS),TMSTR(0:DISKS-1),LOC(128),TMPRS(II,JJ)
        M  = 0
        DO 10 I = 1, VPRS
           M = M + 1
           LOC(M) = TMPRS(I,1)
           FNDX(M) = I
           SNDX(M) = 1
 10     CONTINUE
        TOTLOC = M
        CALL SORT( LOC, FNDX, SNDX, TOTLOC)
        DO 20 I = 1, TOTLOC
           KI  = FNDX(I)
           KBI = BUS(KI)
           M  = FR(KI) + IR
           KD2 = GETSTR(M, IHR, IRV, IRH, DISKS)
           CALL SYNC(TMBUS(KBI),TMSTR(KD2), LOC(I),TR)
           TMPRS(FNDX(I), SNDX(I)) = LOC(I)
 20     CONTINUE
 2      FORMAT(' ',2X,F7.0,2X,'P(',I1,',',I1,')',5X,'R',I4)
        END
C---------------------------------------------------------------
C       READER OF AN S PARTITION FROM AN IBGF FILE
C---------------------------------------------------------------
        SUBROUTINE RDS(FS,SHRS,IS, IHS,BUS,TMBUS,QTYBUS,
     *  STMSTR, SDISKS, TMPRS,II,JJ, VPRS, HPRS, TR, ISV, ISH)
        INTEGER HPRS, VPRS, QTYBUS, SDISKS
```

```
       INTEGER FS(64), SHRS, IS
       INTEGER BUS(HPRS+VPRS)
       INTEGER FNDX(128), SNDX(128), TOTLOC, GETSTR
       REAL LOC(128), TMPRS(II,JJ), TMBUS(QTYBUS),STMSTR(0:SDISKS-1)
       M  = 0
       DO 10 J = 1, HPRS
           M = M + 1
           LOC(M) = TMPRS(1,J)
           FNDX(M) = 1
           SNDX(M) = J
 10    CONTINUE
       TOTLOC = M
       CALL SORT( LOC, FNDX, SNDX, TOTLOC)
       DO 20 I = 1, TOTLOC
           KI = SNDX(I)
           KBI = BUS(KI+ VPRS)
           M  = FS(KI) + IS
           KD2 = GETSTR(M, IHS, ISV, ISH, SDISKS)
           CALL SYNC(TMBUS(KBI),STMSTR(KD2), LOC(I), TR)
           TMPRS(FNDX(I), SNDX(I)) = LOC(I)
 20    CONTINUE
 2     FORMAT(' ',2X,F7.0,2X,'P(',I1,',',I1,')',5X,'S',I2)
       END
C-------------------------------------------------------------------
C      SYNCHRONIZER BETWEEN S-P-CONNECTOR TIME, STORAGE MEDIA
C      TIME AND PROCESSOR TIME.
C-------------------------------------------------------------------
       SUBROUTINE SYNC(A, B, C, TR)
       REAL A, B, C
       IF ( A .LT. B) THEN
          A = B
       ELSE
          B = A
       ENDIF
       IF ( C .LT. B) THEN
          C = B
       ELSE
          B = C
          A = C
       ENDIF
       A = A + TR
       B = A
       C = A
       END
C-------------------------------------------------------------------
C      ORGANIZER OF THE FIRST COME FIRST SERVE PROIRITY QUEUE.
C-------------------------------------------------------------------
      SUBROUTINE SORT(LOC, FNDX, SNDX, TOTLOC)
      INTEGER TOTLOC
      INTEGER FNDX(TOTLOC), SNDX(TOTLOC)
      REAL LOC(TOTLOC)
      LOGICAL DONE
      DONE = .FALSE.
 10   CONTINUE
```

```
      IF (.NOT. DONE ) THEN
        DONE = .TRUE.
        DO 20 I = 1, TOTLOC-1
          IF ( LOC(I) .GT. LOC(I+1)) THEN
             CALL SWAPR(LOC(I), LOC(I+1))
             CALL SWAPI(FNDX(I), FNDX(I+1))
             CALL SWAPI(SNDX(I), SNDX(I+1))
             DONE = .FALSE.
          ENDIF
 20     CONTINUE
        GOTO 10
      ENDIF
      END
C-------------------------------------------------------------
C      ELEMENTS OF ANY ONE DIMENSIONAL INTEGER ARRAY INITIALIZER
C-------------------------------------------------------------
       SUBROUTINE INIT1D(ARRY, DIM1, DIM2, VAL)
       INTEGER DIM1, DIM2
       REAL ARRY(DIM1:DIM2)
       DO 10 I = DIM1, DIM2
 10      ARRY(I) = VAL
       END
C-------------------------------------------------------------
C      ELEMENTS OF ANY TWO DIMENSIONAL REAL ARRAY INITIALIZER
C-------------------------------------------------------------
       SUBROUTINE INIT2D(TMPRS,II,JJ,VPRS,HPRS,VAL)
       REAL TMPRS(II,JJ), VAL
       INTEGER VPRS, HPRS
       DO 10 I = 1, VPRS
       DO 10 J = 1, HPRS
 10       TMPRS(I,J) = VAL
      END
C-------------------------------------------------------------
C      FINDER OF THE TOTAL TIME NEEDED BY THE JOIN OPERATION
C-------------------------------------------------------------
       REAL FUNCTION MPTIME(TMPRS, II,JJ, VPRS, HPRS)
       INTEGER HPRS, VPRS
       REAL TMPRS(II,JJ)
       CURMAX = TMPRS(1,1)
       DO 10 I = 1, VPRS
       DO 10 J = 1, HPRS
         IF (CURMAX .LT. TMPRS(I,J)) CURMAX = TMPRS(I,J)
 10    CONTINUE
       MPTIME = CURMAX
       END
C-------------------------------------------------------------
C      CALCULATOR OF THE NUMBER OF JOIN-CLASSES AND NUMBER
C      OF PARTITIONS IN EACH JOIN-CLASS
C-------------------------------------------------------------
      SUBROUTINE COMP(TOTPG , VPG, HPG)
      INTEGER TOTPG, VPG, HPG
      VPG = TOTPG/2
      HPG = TOTPG - VPG
      VPG = 2**VPG
      HPG = 2**HPG
      END
C-------------------------------------------------------------
C     FINDER AND ASSIGNER THE STORAGE MEDIA OF EACH DATA PARTITION
C-------------------------------------------------------------
      INTEGER FUNCTION GETSTR(VPRT, HPRT, IV, IH, IDISK)
      INTEGER VPRT, HPRT
      K1= VPRT/ IV
      K2= HPRT/ IH
      GETSTR = MOD(K1+K2, IDISK)
      END
C
C-----------------------------------------------------------------
C
```

APPENDIX C

Program listing for Join Algorithm for non-uniform relations based on IBGF

(MODULE 3)

```
C------------------------------------------------------------------
C     SIMULATOR OF A JOIN OPERATION FOR NON-UNIFORM RELATIONS
C     BASED ON IBGF
C------------------------------------------------------------------
C$ STAT=9999999999,O=2000000,TI=0,STAC=999999
       LOGICAL TST(500, 2000)
       INTEGER DIR1(6000,4), DIR2(6000,4)
       INTEGER DIR1C,      DIR2C
       INTEGER CDAT1(6000,4),CDAT2(6000,4),PDAT1(6000,4),PDAT2(6000,
       INTEGER CDAT1C,      CDAT2C,      PDAT1C,      PDAT2C
       INTEGER VS(64,64), OLDBUS, BUS(128)
       INTEGER VPRS, HPRS, AXES, INTNUM, RPRT, SPRT
       INTEGER FR(64), CR(64), CCR, CAP, RR(64)
       INTEGER FS(64), CS(64), CCS, SVD,NJ,RLVL,SLVL
       INTEGER RDISKS,SDISKS, QTYBUS, STRNUM, EMBDER,RPGS,SPGS
       REAL TMPRS(64,64),TMBUS(128),  RTMSTR(0:63), STMSTR(0:63)
       REAL JNTIME, RDTIME, UPTIME, RATIO
       REAL MPTIME, MP
       LOGICAL MMM,RSTAT(6000), SSTAT(6000)
       OPEN( UNIT = 7, FILE = 'IBGF OUT1', STATUS = 'OLD')
       OPEN( UNIT = 8, FILE = 'IBGF OUT2', STATUS = 'OLD')
       OPEN( UNIT = 9, FILE = 'NONUNI OUT', STATUS = 'UNKNOWN')
       OPEN( UNIT = 4, FILE = 'UNI INP', STATUS = 'UNKNOWN')
C  SEEK=8 MSEC, TRANSMIT=16 MEGA BITS, COMPUT=4 MICRO SEC, REC = 56
       T = 73
       TD = 32
       TC = 28*T
       VPRS   = 8
       HPRS   = 8
       QTYBUS = 8
       OLDBUS = QTYBUS
       IF (QTYBUS .GT. VPRS) QTYBUS = VPRS
       TR = 28*T + 8000
       TJ = 4*T*T
       TCD = 6*TD
       TRD = 6*TD + 8000
       II = 64
       JJ = 64
       CAP = 99999999
       RDTIME = 0
       AXES = 2
       CALL INITR(RLVL,RDISKS,RPGS,RPROB,RPRT)
       CALL INITS(RPRT, SLVL, RATIO,SDISKS,SPGS,SPROB,SPRT)
       CALL INIT(TMBUS, 1,QTYBUS, 1.0)
       CALL INIT(RTMSTR, 0, RDISKS-1, 1.0)
       CALL INIT(STMSTR, 0, SDISKS-1, 1.0)
       CALL INIT2(TMPRS,II,JJ,VPRS,HPRS,0.0)
       CALL INTTAB(PDAT1, PDAT1C, -1)
       CALL INTTAB(PDAT2, PDAT2C, -1)
       CALL ASNBUS(BUS, QTYBUS, VPRS, HPRS)
       NJ  = 0
       IIR = 0
       IIS = 0
       RLMT = 0
```

```
10     IF (IIR .LT. RLVL) THEN
         CALL INTCNT( CCR, CCS, SVD,DIR1C,DIR2C,CDAT1C,CDAT2C)
         LMTR = IIR
         LMTS = IIS
         CALL BDTABR(DIR1,DIR1C,CDAT1,CDAT1C,IIR,7,RDISKS,RLMT,.TRUE
         SLMT = IIR*RATIO
         CALL BDTABS(DIR2,DIR2C,CDAT2,CDAT2C,IIS,8,SDISKS,SLMT)
         RLMT = IIS / RATIO
         IF ( IIR .LT. RLMT) THEN
         CALL BDTABR(DIR1,DIR1C,CDAT1,CDAT1C,IIR,7,RDISKS,RLMT,.FALS
         ENDIF
         CALL UPDATA(CDAT1, CDAT1C, PDAT1, PDAT1C, LMTR,RSTAT,RDISKS
         CALL UPDATA(CDAT2, CDAT2C, PDAT2, PDAT2C, LMTS,SSTAT,SDISKS
         IF (CDAT1C .EQ. 0 .OR. CDAT2C .EQ. 0)THEN
          GOTO 10
         ENDIF
         CALL SORTAB(CDAT1, CDAT1C,RSTAT)
         CALL SORTAB(CDAT2, CDAT2C,SSTAT)
         CALL GETMMM( CDAT1C, M1, M2, M3,VPRS,MMM)
         RDTIME = RDTIME + TRD*(DIR1C + DIR2C) + TR*(CDAT1C + CDAT2C
         CALL RDDIR(DIR1,DIR1C,TMPRS,II,JJ,
     *        TMBUS, QTYBUS, BUS,RTMSTR, RDISKS,VPRS,HPRS,TRD,TCD)
         CALL RDDIR(DIR2,DIR2C,TMPRS,II,JJ,
     *        TMBUS, QTYBUS, BUS,STMSTR, SDISKS,VPRS,HPRS,TRD,TCD)
         CALL DVDR(FR,CR,RR,CDAT1C,VPRS,M1,M2,M3,MMM)
         CALL DVD(FS,CS,CDAT2C, HPRS)
         CALL RDSR(TMPRS,II,JJ,FR,CR,CCR,FS,CS,CCS,RTMSTR,RDISKS,
     *   CDAT1,CDAT2,STMSTR,TMBUS,QTYBUS,SDISKS,BUS,VPRS,HPRS,TR)
         CALL GETRFH(TMPRS, II, JJ, CCS, CCR, CS, CR, VPRS, HPRS,
     *   FR, CDAT1,TC)
         CALL GETS(TMPRS,II,JJ,CCS,CCR,CS,CR,VPRS,HPRS,SVD,CAP,
     *   FS, CDAT2,TC)
         CALL GETSHR(TMPRS, II, JJ, CCS, CCR, CS, CR, VPRS, HPRS,
     *   FR, CDAT1,TC)
 20      CALL JOIN(TMPRS, II,JJ,CCR, CCS,CR, CS, VPRS, HPRS,
     *   CDAT1, CDAT2,FR, FS,NJ,SSTAT,RSTAT,TJ)
         IF (CCS .LT. CS(1)) THEN
           CALL  RDS(CDAT2, FS,CS,CCS,BUS,TMBUS,QTYBUS,
     *     STMSTR, SDISKS, TMPRS,II,JJ, VPRS, HPRS,SVD, CAP,TR)
           CALL GETS(TMPRS,II,JJ,CCS,CCR,CS,CR,VPRS,HPRS,SVD,CAP,
     *     FS, CDAT2,TC)
           GOTO 20
         ELSEIF(CCR .LT. CR(1)-1 .OR. MMM .AND. CCR .LT. CR(1)) THEN
           CCS = 0
           CALL  RDR(CDAT1, FR,CR,CCR,BUS,TMBUS,QTYBUS,
     *     RTMSTR, RDISKS, TMPRS,II,JJ, VPRS, HPRS,TR)
           CALL GETR(TMPRS, II, JJ, CCS, CCR, CS, CR, VPRS, HPRS,
     *     FR, CDAT1,TC)
           CALL  RDS(CDAT2, FS,CS,CCS,BUS,TMBUS,QTYBUS,
     *     STMSTR, SDISKS, TMPRS,II,JJ, VPRS, HPRS,SVD, CAP,TR)
           CALL GETS(TMPRS,II,JJ,CCS,CCR,CS,CR,VPRS,HPRS,SVD,CAP,
     *     FS, CDAT2,TC)
           GOTO 20
         ELSEIF(.NOT. MMM .AND. CCR .LT. CR(1)) THEN
```

```
           CALL SPRDR(CDAT1,TMBUS,RTMSTR,TMPRS,BUS,RR,M2,M3,HPRS,
     *     VPRS, QTYBUS, RDISKS,II,JJ,TR)
           CALL SHFTRD(TMPRS,M2,M3,II,JJ,TC)
           CALL GETR(TMPRS, II, JJ, CCS, CCR, CS, CR, VPRS, HPRS,
     *     FR, CDAT1,TC)
           CALL GETVS(CS, M2,M3, HPRS, CAP, VS, II,JJ,VPRS)
           CALL SPJOIN(TMPRS,II,JJ,HPRS,M2,M3,VS,NJ,
     *     RR,FS,CDAT1C,CDAT2C,VPRS,CDAT1,CDAT2,SSTAT,RSTAT,TJ)
           SVD = 0
           CCS = 0
           DO  30 I = 1, M3
 30          CCS = CCS + VS(I,1)
 40        IF (CCS .LT. CS(1)) THEN
             CALL  RDS(CDAT2, FS,CS,CCS,BUS,TMBUS,QTYBUS,
     *       STMSTR, SDISKS, TMPRS,II,JJ, VPRS, HPRS,SVD, CAP, TR)
             CALL GETS(TMPRS,II,JJ,CCS,CCR,CS,CR,VPRS,HPRS,SVD,CAP,
     *       FS, CDAT2, TC)
             CALL JOIN2(TMPRS, II,JJ,CCS,CS, VPRS, HPRS,
     *       CDAT1, CDAT2,FS,RR,M2,M3,NJ,SSTAT,RSTAT, TJ)
             GOTO 40
           ENDIF
         ENDIF
         CCS = 0
         CCR = 0
         SVD = 0
         GOTO 10
       ENDIF
       MP = MPTIME(TMPRS, II,JJ, VPRS, HPRS)
       WRITE(4,4) VPRS, HPRS, QTYBUS,RDISKS+SDISKS,SPRT,RPRT,MP,
     + RPROB, SPROB
 4     FORMAT(' ',I3, I3, I3, I3, I4,I4, F15.2,F6.1, F5.1)
       UP = RDTIME + NJ*TJ
       IUP = UP
       SPEED = UP/MP
       EFF = UP/(MP*(VPRS*HPRS))*100
       WRITE(9,2) VPRS, HPRS, OLDBUS, RDISKS,SDISKS,RPGS,RPROB,
     * SPGS,SPROB, SPEED,EFF
 2     FORMAT(' ', I4,I4,I4,I4,I4,2(2X,I8,2X,F3.1), F9.2, F9.2)
 3     FORMAT(' ', F11.2, I8)
       END
C-------------------------------------------------------------
C      CALCULATOR OF TIME TAKEN BY THE JOIN OPERATION
C-------------------------------------------------------------
       REAL FUNCTION MPTIME(TMPRS, II,JJ, VPRS, HPRS)
       INTEGER HPRS, VPRS
       REAL TMPRS(II,JJ)
       CURMAX = TMPRS(1,1)
       DO 10 I = 1, VPRS
       DO 10 J = 1, HPRS
         IF (CURMAX .LT. TMPRS(I,J)) CURMAX = TMPRS(I,J)
 10    CONTINUE
       MPTIME = CURMAX
       END
C-------------------------------------------------------------
```

```
C      INITIALIZER OF COUNTERS
C-------------------------------------------------------------
       SUBROUTINE INTCNT( CCR, CCS, SVD,DIR1C,DIR2C,CDAT1C,CDAT2C)
       INTEGER CCR, CCS, SVD,DIR1C,DIR2C,CDAT1C,CDAT2C
       CCR    = 0
       CCS    = 0
       SVD    = 0
       DIR1C  = 0
       DIR2C  = 0
       CDAT1C = 0
       CDAT2C = 0
       END
C-------------------------------------------------------------
C      SYNCHRONIZER BETWEEN S-P-CONNECTOR TIME, STORAGE MEDIA
C      TIME AND PROCESSOR TIME.
C-------------------------------------------------------------
       SUBROUTINE SYNC(A, B, C, D)
       REAL A,B,C,D
       IF ( A .LT. B) THEN
          A = B
       ELSE
          B = A
       ENDIF
       IF ( C .LT. B) THEN
          C = B
       ELSE
          B = C
          A = C
       ENDIF
       A = A + D
       B = A
       C = A
       END
C-------------------------------------------------------------
C      FIRST COME FIRST SERVE PRIORITY QUEUE SIMULATOR
C-------------------------------------------------------------
      SUBROUTINE SORT(LOC, FNDX, SNDX, TOTLOC)
      INTEGER TOTLOC
      REAL LOC(TOTLOC)
      INTEGER FNDX(TOTLOC), SNDX(TOTLOC)
      LOGICAL DONE
      DONE = .FALSE.
 10   CONTINUE
      IF (.NOT. DONE ) THEN
        DONE = .TRUE.
        DO 20 I = 1, TOTLOC-1
          IF ( LOC(I) .GT. LOC(I+1)) THEN
             CALL SWAPR(LOC(I), LOC(I+1))
             CALL SWAPI(FNDX(I), FNDX(I+1))
             CALL SWAPI(SNDX(I), SNDX(I+1))
             DONE = .FALSE.
          ENDIF
 20     CONTINUE
        GOTO 10
      ENDIF
      END
C-----------------------------------------------------------------
C      READER OF THE NUMBER OF PARTITIONS IN RELATION R,
C      ITS SkEWEDNESS, THE LEVELS OF ITS DIRECTORY PARTITIONS
C      AND THE NUMBER OF DISKS IT IS RESIDING ON.
C-----------------------------------------------------------------
       SUBROUTINE  INITR(RLVL,RDISKS,RPGS,RPROB,RPRT)
       INTEGER   RLVL,RDISKS,RPGS, RPRT
       READ(7,*) RLVL, RDISKS, RPGS,RPROB,RPRT, RPRT
       END
C-----------------------------------------------------------------
C      READER OF THE NUMBER OF PARTITIONS IN RELATION S,
C      ITS SkEWEDNESS, THE LEVELS OF ITS DIRECTORY PARTITIONS
C      AND THE NUMBER OF DISKS IT IS RESIDING ON.
C-----------------------------------------------------------------
       SUBROUTINE  INITS(RPRT, SLVL, RATIO,SDISKS,SPGS,SPROB,SPRT)
       INTEGER RPRT, SLVL, SDISKS,SPGS,SPRT
       READ(8,*) SLVL, SDISKS,SPGS,SPROB,SPRT, SPRT
       TMP1 = (SPRT + 1)/ 2
       TMP2 = (RPRT + 1)/ 2
       TMP = TMP1 - TMP2
       RATIO = 2.0**TMP
       END
C-----------------------------------------------------------------
C      BUILDER OF AGGREGATE-CLASSES
C-----------------------------------------------------------------
       SUBROUTINE UPDATA(CDATA, CDATAC, PDATA, PDATAC, LMT,STAT,DISK
       INTEGER CDATA(6000,4), CDATAC, DISKS
       INTEGER LMT
       INTEGER PDATA(6000,4), PDATAC
       LOGICAL STAT(6000)
       M = CDATAC
       DO 44 I = 1, M
 44       STAT(I) = .FALSE.
       DO 10 I = 1, PDATAC
         IF (PDATA(I,4) .GT. LMT) THEN
           M = M + 1
           STAT(M) = .TRUE.
           DO 20 J = 1, 4
             CDATA(M,J) = PDATA(I,J)
 20        CONTINUE
         ENDIF
 10    CONTINUE
       DO 30 I = 1, M
       DO 30 J = 1, 4
         PDATA(I,J) = CDATA(I,J)
30     CONTINUE
       PDATAC = M
       CDATAC = M
       END
C-----------------------------------------------------------------
C     FINDER OF THE NEXT EMBEDDER PARTITION NUMBER.
C-----------------------------------------------------------------
```

```
      INTEGER FUNCTION EMBDER(PRT)
      INTEGER PRT, TMP, K
      TMP = PRT
      K = 0
 10   IF ( TMP .GE. 2) THEN
        K = K + 1
        TMP = TMP/2
        GOTO 10
      ENDIF
      TMP = 2**K
      EMBDER = PRT - TMP
      END
C--------------------------------------------------------------
C      BUILDER OF AN S JOIN-CLASS
C--------------------------------------------------------------
       SUBROUTINE BDTABS(DIR,DIRC,DATA,DATAC,IIS,KK,SDISKS,SLMT)
       INTEGER DIR(6000,4), DIRC, DATA(6000,4), DATAC, IIS
       INTEGER D(5), SDISKS, RNG
       LOGICAL DUP
       I = 0
       K = 0
 10    IF (IIS .LT. SLMT .AND. IIS .LT. 2000000000) THEN
         READ(KK,*) IIS, RNG
  5      READ(KK,*)(D(J),J=1,5)
         IF (D(1) .NE. -1) THEN
           IF (D(5) .GT. 0) THEN
               I = I + 1
               DO 30 J = 1, 4
                 DIR(I,J) = D(J)
  30           CONTINUE
           ELSE
               K = K + 1
               DO 20 J = 1, 4
                 DATA(K,J) = D(J)
  20           CONTINUE
           ENDIF
           GOTO 5
         ENDIF
         IIS = RNG
         GOTO 10
       ENDIF
       DIRC = I
       DATAC = K
       END
C--------------------------------------------------------------
C      BUILDER OF AN R JOIN_CLASS
C--------------------------------------------------------------
      SUBROUTINE BDTABR(DIR,DIRC,DATA,DATAC,IIR,KK,RDISKS,RLMT,INTIA
       INTEGER DIR(6000,4), DIRC, DATA(6000,4), DATAC, IIR
       INTEGER D(5), RDISKS, RNG
       LOGICAL INTIAL
       I = DIRC
       K = DATAC
 10    IF( (IIR .LT. RLMT .OR. INTIAL) .AND. IIR .LT. 2000000000) TH
```

```
        INTIAL = .FALSE.
        READ(KK,*) III, RNG
  5     READ(KK,*)(D(J),J=1,5)
        IF (D(1) .NE. -1) THEN
          IF (D(5) .GT. 0) THEN
              IF(D(3) .LT. 0) PRINT*, IIS, RNG
              I = I + 1
              DO 30 J = 1, 4
                DIR(I,J) = D(J)
  30          CONTINUE
          ELSE
              K = K + 1
              DO 20 J = 1, 4
                DATA(K,J) = D(J)
  20          CONTINUE
          ENDIF
          GOTO 5
        ENDIF
        IIR = RNG
        GOTO 10
      ENDIF
      DIRC = I
      DATAC = K
      END
C--------------------------------------------------------------
C    INITIALIZER OF A 2 DIMENSIONAL INTEGER ARRAY
C--------------------------------------------------------------
      SUBROUTINE INTTAB(TABLE, TABSIZ, VAL)
      INTEGER TABSIZ, TABLE(6000,4), VAL
      TABSIZ = 0
      DO 10 I = 1, TABSIZ
      DO 10 J = 1, 4
        TABLE(I,J) = VAL
10    CONTINUE
      END
C--------------------------------------------------------------
C    INITIALIZER OF A 2 DIMENSIONAL REAL ARRAY
C--------------------------------------------------------------
      SUBROUTINE INIT2(TMPRS,II,JJ,VPRS,HPRS,VAL)
      REAL TMPRS(II,JJ), VAL
      INTEGER VPRS, HPRS
      DO 10 I = 1, VPRS
      DO 10 J = 1, HPRS
 10      TMPRS(I,J) = VAL
     END
C--------------------------------------------------------------
C    INITIALIZER OF ANY 1 DIMENSIONAL REAL RARRAY
C--------------------------------------------------------------
      SUBROUTINE INIT(ARRY, DIM1, DIM2, VAL)
      INTEGER DIM1, DIM2
      REAL ARRY(DIM1:DIM2), VAL
      DO 10 I = DIM1, DIM2
 10     ARRY(I) = VAL
      END
```

```
C-------------------------------------------------------------
C    INITIALIZER OF ANY 1 DIMENSIONAL INTEGER ARRAY
C-------------------------------------------------------------
      SUBROUTINE INITI(ARRY, DIM1, DIM2, VAL)
      INTEGER DIM1, DIM2
      INTEGER ARRY(DIM1:DIM2), VAL
      DO 10 I = DIM1, DIM2
 10     ARRY(I) = VAL
      END
C-------------------------------------------------------------
C     ASSIGNER OF S-P-CONNECTOR TO A PROCESSOR
C-------------------------------------------------------------
      SUBROUTINE ASNBUS(BUS, QTYBUS, VPRS, HPRS)
      INTEGER QTYBUS , VPRS, HPRS
      INTEGER BUS(VPRS + HPRS)
      K1 = VPRS/QTYBUS
      K2 = HPRS/QTYBUS
      M = 1
      DO 10 I = 1, VPRS
        BUS(I) = M
        IF (MOD(I, K1) .EQ. 0) M = M + 1
 10   CONTINUE
      M = 1
      DO 20 J = 1, HPRS
        BUS(J + VPRS) = M
        IF (MOD(J, K2) .EQ. 0) M = M + 1
 20   CONTINUE
      END
C-------------------------------------------------------------
C     READER OF AN S AND AN R PARTITION
C-------------------------------------------------------------
      SUBROUTINE RDSR(TMPRS,II,JJ,FR,CR,CCR,FS,CS,CCS,RTMSTR,RDISKS
     * CDAT1, CDAT2,STMSTR, TMBUS,QTYBUS,SDISKS,BUS,VPRS,HPRS,TR)
      INTEGER FR(64), CR(64),  CCR, CAP
      INTEGER FS(64), CS(64),  CCS, SVD
      INTEGER RDISKS,SDISKS, QTYBUS,VPRS, HPRS
      INTEGER  BUS(VPRS+HPRS), CDAT1(6000,4), CDAT2(6000,4)
      INTEGER  FNDX(128), SNDX(128), TOTLOC
      REAL   TMBUS(QTYBUS), STMSTR(0:SDISKS-1),TMPRS(II,JJ)
      REAL   RTMSTR(0:RDISKS-1), LOC(128)
      M = 0
      CCR = CCR + 1
      CCS = CCS + 1
      IF (CCR .LE. CR(1) .OR. CCS .LE. CS(1)) THEN
        M = M + 1
        LOC(M) = TMPRS(1,1)
        FNDX(M) = 1
        SNDX(M) = 1
      ENDIF
      DO 10 I = 2, VPRS
      IF (CCR .LE. CR(I)) THEN
        M = M + 1
        LOC(M) = TMPRS(I,1)
        FNDX(M) = I
```

```
          SNDX(M) = 1
        ENDIF
 10     CONTINUE
        DO 20 I = 2, HPRS
        IF (CCS .LE. CS(I)) THEN
          M = M + 1
          LOC(M) = TMPRS(1,I)
          FNDX(M) = 1
          SNDX(M) = I
        ENDIF
 20     CONTINUE
        TOTLOC = M

        IF (M .GT. 0) CALL SORT(LOC, FNDX, SNDX, TOTLOC)
        DO 30 I = 1, TOTLOC
        IF (SNDX(I) .EQ. 1 ) THEN
          KI =  FNDX(I)
          KBI = BUS(KI)
          KD1 = CDAT1(FR(KI) + CCR - 1, 3)
          KD2 = CDAT1(FR(KI) + CCR - 1, 1)
          KD3 = CDAT1(FR(KI) + CCR - 1, 2)
C         PRINT*, KD2, KD3, KD1
          CALL SYNC(TMBUS(KBI), RTMSTR(KD1), LOC(I),TR)
C         WRITE(9,2) LOC(I)-TR,FNDX(I), SNDX(I),KD2
 2        FORMAT(' ',2X,F7.0,2X,'P(',I1,',',I1,')',5X,'R',I4)
          TMPRS(FNDX(I), SNDX(I)) = LOC(I)
        ENDIF
        IF (FNDX(I) .EQ. 1 ) THEN
          KI = SNDX(I)
          KBI = BUS(KI+ VPRS)
          KD1 = CDAT2(FS(KI) + CCS - 1, 3)
          KD2 = CDAT2(FS(KI) + CCS - 1, 1)
          KD3 = CDAT2(FS(KI) + CCS - 1, 2)
          CALL SYNC(TMBUS(KBI), STMSTR(KD1), LOC(I),TR)
C         WRITE(9,3) LOC(I)-TR,FNDX(I), SNDX(I),KD2
 3        FORMAT(' ',2X,F7.0,2X,'P(',I1,',',I1,')',5X,'S',I4)
          TMPRS(FNDX(I), SNDX(I)) = LOC(I)
        ENDIF
 30     CONTINUE
        END
C-----------------------------------------------------------
C     PARTITIONS OF AN AGGREGATE-CLASS DIVIDER AMONGEST PROCESSORS
C-----------------------------------------------------------
      SUBROUTINE DVD(F,C,CNT,DIM1)
      INTEGER DIM1
      INTEGER F(DIM1),C(DIM1), CNT
      INTEGER M, K, L
      K = CNT/DIM1
      L = MOD(CNT, DIM1)
      CALL INITI(F,1,DIM1,0)
      CALL INITI(C,1,DIM1,0)
      N = 0
      DO 10 I = 1, DIM1
        F(I) = N+1
```

```
          N = N + K
          C(I) = K
          IF ( I .LE. L ) THEN
            N = N + 1
            C(I) = C(I) + 1
          ENDIF
 10     CONTINUE
        END
C-------------------------------------------------------------
C       READER OF AN S PARTITION
C-------------------------------------------------------------
        SUBROUTINE RDS(PDAT2, FS,CS,CCS,BUS,TMBUS,QTYBUS,
     *  STMSTR, SDISKS, TMPRS,II,JJ, VPRS, HPRS,SVD, CAP, TR)
        INTEGER HPRS, VPRS, QTYBUS, SDISKS,SVD, CAP
        INTEGER PDAT2(6000,4),FS(64), CS(64), CCS
        INTEGER BUS(HPRS+VPRS)
        REAL TMBUS(QTYBUS), STMSTR(0:SDISKS-1), LOC(128),TMPRS(II,JJ)
        INTEGER  FNDX(128), SNDX(128), TOTLOC
        CCS = CCS + 1
        IF (CCS .GT. SVD) THEN
          M  = 0
          DO 10 J = 1, HPRS
          IF (CCS .LE. CS(J)) THEN
            M = M + 1
            LOC(M) = TMPRS(1,J)
            FNDX(M) = 1
            SNDX(M) = J
          ENDIF
 10       CONTINUE
          TOTLOC = M
          IF (TOTLOC .GT. 0) THEN
            CALL SORT( LOC, FNDX, SNDX, TOTLOC)
          ENDIF
          DO 20 I = 1, TOTLOC
            KI = SNDX(I)
            KBI = BUS(KI+ VPRS)
            KD1 = PDAT2(FS(KI) + CCS - 1, 3)
            KD2 = PDAT2(FS(KI) + CCS - 1, 1)
            KD3 = PDAT2(FS(KI) + CCS - 1, 2)
            CALL SYNC(TMBUS(KBI),STMSTR(KD1), LOC(I),TR)
C           WRITE(9,3) LOC(I)-TR,FNDX(I), SNDX(I),KD2
 3          FORMAT(' ',2X,F7.0,2X,'P(',I1,',',I1,')',5X,'S',I4)
            TMPRS(FNDX(I), SNDX(I)) = LOC(I)
 20       CONTINUE
        ENDIF
        END
C-------------------------------------------------------------
C       READER OF AN R PARTITION
C-------------------------------------------------------------
        SUBROUTINE RDR(PDAT1, FR,CR,CCR,BUS,TMBUS,QTYBUS,
     *  RTMSTR, RDISKS, TMPRS,II,JJ, VPRS, HPRS,TR)
        INTEGER HPRS, VPRS, QTYBUS, RDISKS
        INTEGER PDAT1(6000,4),FR(64), CR(64), CCR
        REAL TMPRS(II,JJ), TMBUS(QTYBUS),RTMSTR(0:RDISKS-1)
```

```
      INTEGER BUS(HPRS+ VPRS)
      REAL  LOC(128)
      INTEGER FNDX(128), SNDX(128), TOTLOC
        CCR = CCR + 1
        M  = 0
        DO 10 I = 1, VPRS
        IF (CCR .LE. CR(I)) THEN
          M = M + 1
          LOC(M) = TMPRS(I,1)
          FNDX(M) = I
          SNDX(M) = 1
        ENDIF
 10     CONTINUE
        TOTLOC = M
        IF (TOTLOC .GT. 0) THEN
          CALL SORT( LOC, FNDX, SNDX, TOTLOC)
        ENDIF
        DO 20 I = 1, TOTLOC
          KI  = FNDX(I)
          KBI = BUS(KI)
          KD1 = PDAT1(FR(KI) + CCR - 1, 3)
          KD2 = PDAT1(FR(KI) + CCR - 1, 1)
          KD3 = PDAT1(FR(KI) + CCR - 1, 2)
          CALL SYNC(TMBUS(KBI),RTMSTR(KD1), LOC(I),TR)
C         WRITE(9,2) LOC(I)-TR,FNDX(I), SNDX(I),KD2
 2        FORMAT(' ',2X,F7.0,2X,'P(',I1,',',I1,')',5X,'R',I4)
          TMPRS(FNDX(I), SNDX(I)) = LOC(I)
 20     CONTINUE
      END
C-----------------------------------------------------------------
C     ACCEPTOR OF AN R PARTITION FROM A NEIGHBOR. THIS ROUTINE
C     INVOLVES PROCESSORS WITH THEIR ROW NUMBER NOT LESS THAN
C     THEIR COLUMN NUMBER ONLY.
C-----------------------------------------------------------------
      SUBROUTINE GETRFH(TMPRS, II, JJ, CCS, CCR, CS, CR, VPRS, HPRS,
     * FR, PDAT1,TC)
      INTEGER VPRS, HPRS
      INTEGER CCS, CCR, CS(64), CR(64)
      REAL  TMPRS(II,JJ)
      INTEGER FR(VPRS), PDAT1(6000,4)
      DO 10 I = 1, VPRS
      DO 10 J = 2, I
        IF(CCS .LE. CS(J) .AND. CCR .LE. CR(I)) THEN
          IF(TMPRS(I,J-1) .LT. TMPRS(I,J)-TC) THEN
             TMPRS(I,J-1) = TMPRS(I,J) - TC
          ELSE
             TMPRS(I,J) = TMPRS(I,J-1) + TC
          ENDIF
          KD = PDAT1(FR(I) + CCR - 1, 1)
C         WRITE(9,2) TMPRS(I,J-1), I, J-1, 'R', KD,' ===>'
          TMPRS(I,J-1) = TMPRS(I,J)
C         WRITE(9,2) TMPRS(I,J), I, J, 'R', KD
          TMPRS(I,J) = TMPRS(I,J) + TC
        ENDIF
 10   CONTINUE
 2    FORMAT(' ',2X, F7.0, 2X, 'P(',I1,',',I1,')', 5X,A,I4,A)
      END
C-------------------------------------------------------------
C     ACCEPTOR OF AN R PARTITION
C-------------------------------------------------------------
      SUBROUTINE GETR(TMPRS, II, JJ, CCS, CCR, CS, CR, VPRS, HPRS,
     * FR, PDAT1,TC)
      INTEGER VPRS, HPRS
      INTEGER CCS, CCR, CS(64), CR(64)
      REAL TMPRS(II,JJ)
      INTEGER FR(VPRS), PDAT1(6000,4)
      DO 10 I = 1, VPRS
      DO 10 J = 2, HPRS
        IF(CCS .LE. CS(J) .AND. CCR .LE. CR(I)) THEN
          IF(TMPRS(I,J-1) .LT. TMPRS(I,J)-TC) THEN
             TMPRS(I,J-1) = TMPRS(I,J) - TC
          ELSE
             TMPRS(I,J) = TMPRS(I,J-1) + TC
          ENDIF
          KD = PDAT1(FR(I) + CCR - 1, 1)
C         WRITE(9,2) TMPRS(I,J-1), I, J-1, 'R', KD,' ===>'
          TMPRS(I,J-1) = TMPRS(I,J)
C         WRITE(9,2) TMPRS(I,J), I, J, 'R', KD
          TMPRS(I,J) = TMPRS(I,J) + TC
        ENDIF
 10   CONTINUE
 2    FORMAT(' ',2X, F7.0, 2X, 'P(',I1,',',I1,')', 5X,A,I4,A)
      END
C-------------------------------------------------------------
C     ACCEPTOR OF AN R PARTITION FROM A NEIGHBOR. THIS ROUTINE
C     INVOLVES PROCESSORS WITH THEIR ROW NUMBER LESS THAN
C     THEIR COLUMN NUMBER ONLY.
C-------------------------------------------------------------
      SUBROUTINE GETSHR(TMPRS, II, JJ, CCS, CCR, CS, CR, VPRS, HPRS,
     * FR, PDAT1,TC)
      INTEGER VPRS, HPRS
      INTEGER CCS, CCR, CS(64), CR(64)
      REAL TMPRS(II,JJ)
      INTEGER FR(VPRS), PDAT1(6000,4)
      DO 10 I = 1, VPRS
      DO 10 J = I+1, HPRS
        IF(CCS .LE. CS(J) .AND. CCR .LE. CR(I)) THEN
          IF(TMPRS(I,J-1) .LT. TMPRS(I,J)-TC) THEN
             TMPRS(I,J-1) = TMPRS(I,J) - TC
          ELSE
             TMPRS(I,J) = TMPRS(I,J-1) + TC
          ENDIF
          KD = PDAT1(FR(I) + CCR - 1, 1)
C         WRITE(9,2) TMPRS(I,J-1), I, J-1, 'R', KD,' ===>'
          TMPRS(I,J-1) = TMPRS(I,J)
C         WRITE(9,2) TMPRS(I,J), I, J, 'R', KD
          TMPRS(I,J) = TMPRS(I,J) + TC
        ENDIF
 10   CONTINUE
 2    FORMAT(' ',2X, F7.0, 2X, 'P(',I1,',',I1,')', 5X,A,I4,A)
      END
C-----------------------------------------------------------------
C     ACCEPTOR OF AN S PARTITION
C-----------------------------------------------------------------
      SUBROUTINE GETS(TMPRS,II,JJ,CCS,CCR,CS,CR,VPRS,HPRS,SVD,CAP,
     * FS, PDAT2, TC)
      INTEGER VPRS, HPRS, SVD, CAP
      INTEGER CCS, CCR, CS(64), CR(64)
      REAL TMPRS(II,JJ)
      INTEGER FS(HPRS), PDAT2(6000,4)
      IF (SVD .LT. CCS) THEN
        IF(SVD .LT. CAP) SVD = SVD + 1
        DO 10 I = 2, VPRS
        DO 10 J = 1, HPRS
          IF(CCS .LE. CS(J) .AND. CCR .LE. CR(I)) THEN
            IF(TMPRS(I-1,J) .LT. TMPRS(I,J)-TC) THEN
               TMPRS(I-1,J) = TMPRS(I,J) - TC
            ELSE
               TMPRS(I,J) = TMPRS(I-1,J) + TC
            ENDIF
            KD = PDAT2(FS(J) + CCS - 1, 1)
C           WRITE(9,2) TMPRS(I-1,J), I-1, J, 'S', KD,' ===>'
            TMPRS(I-1,J) = TMPRS(I,J)
C           WRITE(9,2) TMPRS(I,J), I, J, 'S', KD
            TMPRS(I,J) = TMPRS(I,J) + TC
          ENDIF
 10     CONTINUE
      ENDIF
 2    FORMAT(' ',2X, F7.0, 2X, 'P(',I1,',',I1,')', 5X,A,I4,A)
      END
C-----------------------------------------------------------------
C     JOIN OPERATION SIMULATOR.
C-----------------------------------------------------------------
       SUBROUTINE JOIN(TMPRS, II,JJ,CCR, CCS,CR, CS, VPRS, HPRS,
     * PDAT1, PDAT2,FR, FS,NJ,SSTAT,RSTAT, TJ)
       INTEGER VPRS, HPRS
       INTEGER PDAT1(6000,4), PDAT2(6000,4),FR(VPRS), FS(HPRS)
       REAL  TMPRS(II,JJ)
       INTEGER CCR, CCS,CR(VPRS), CS(HPRS)
       LOGICAL SSTAT(6000), RSTAT(6000)
       DO 10 I = 1, VPRS
       DO 10 J = 1, HPRS
         IF(CCR .LE. CR(I) .AND. CCS .LE. CS(J)) THEN
           M =  FR(I)+CCR-1
           N =  FS(J)+CCS-1
           KP1 = PDAT1( M,1)
           KP2 = PDAT2( N,1)
           IF( .NOT. (RSTAT(M) .AND. SSTAT(N))) THEN
C            WRITE(9,2) TMPRS(I,J), I, J, KP1,'|><|',KP2
             TMPRS(I,J) = TMPRS(I,J) + TJ
             NJ = NJ + 1
           ENDIF
         ENDIF
10     CONTINUE
 2     FORMAT(' ',2X,F7.0,2X,'P(',I1,',',I1,')',5X,'R',I4,A,'S',I4)
       END
C-----------------------------------------------------------------
C     CALCULATOR OF THE NUMBER OF ROWS THAT EACH R PARTITION
C     HAS TO BE READ TO.
C-----------------------------------------------------------------
      SUBROUTINE GETVS(CS, M2,M3, HPRS, CAP, VS, II,JJ,VPRS)
      INTEGER HPRS, CAP, VPRS
      INTEGER CS(HPRS), VS(II,JJ)
      DO 5 I = 1, VPRS
      DO 5 J = 1, HPRS
 5      VS(I,J) = 0
      DO 10 J = 1, HPRS
        K1 = CS(J)/M3
        K2 = MOD(CS(J), M3)
        DO 10 I = 1, M3
           VS(I,J) = K1
           IF (K1 .GE. CAP) THEN
             VS(I,J) = CAP
           ELSE
             IF(I .LE. K2) VS(I,J) = VS(I,J) + 1
           ENDIF
 10   CONTINUE
      DO 20 J = 2, M2
        K1 = (J-1)*M3
      DO 20 I = 1, M3
      DO 20 K = 1, HPRS
        VS(K1 + I, K) = VS(I,K)
 20   CONTINUE
      END
C-----------------------------------------------------------------
C     SENDER/ ACCEPTOR  OF R PARTITIONS TO PROCESSORS
C     IN THE SAME COLUMN
C-----------------------------------------------------------------
      SUBROUTINE SHFTRD(TMPRS,M2,M3,II,JJ,TC)
      REAL TMPRS(II,JJ)
      DO 10 I = 1, M2
      K = (I-1)*M3
      DO 10 J = 2, M3
        IF(TMPRS(K+J-1,1) .LT. TMPRS(K+J,1) - TC) THEN
          TMPRS(K+J-1,1) = TMPRS(K+J,1) - TC
        ELSE
          TMPRS(K+J,1) = TMPRS(K+J-1,1) + TC
        ENDIF
        TMPRS(K+J,1) = TMPRS(K+J,1) + TC
        TMPRS(K+J-1,1) = TMPRS(K+J-1,1) + TC
 10   CONTINUE
      END
C-----------------------------------------------------------------
C      READS AN R PARTITION TO MORE THAN ONE PROCESSOR.
C-----------------------------------------------------------------
       SUBROUTINE SPRDR(PDAT1,TMBUS,RTMSTR,TMPRS,BUS,RR,M2,M3,HPRS,
     * VPRS, QTYBUS, RDISKS,II,JJ, TR)
       INTEGER HPRS, VPRS, QTYBUS, RDISKS
       INTEGER PDAT1(6000,4),RR(VPRS)
       INTEGER BUS(HPRS+ VPRS)
       REAL LOC(128), TMBUS(QTYBUS),RTMSTR(0:RDISKS-1),TMPRS(II,JJ)
       INTEGER FNDX(128), SNDX(128), TOTLOC
       M = 0
       DO 10 I = 1, M2*M3,M3
         M = M + 1
         LOC(M) = TMPRS(I,1)
         FNDX(M) = I
 10      SNDX(M) = 1
       TOTLOC = M
       CALL SORT(LOC,FNDX,SNDX,TOTLOC)
       DO 20 I = 1, TOTLOC
         KI = FNDX(I)
         KD1 = PDAT1(RR(KI),3)
         KD2 = PDAT1(RR(KI), 1)
         KBI = BUS(KI)
         CALL SYNC(TMBUS(KBI),RTMSTR(KD1), LOC(I),TR)
C        WRITE(9,2) LOC(I)-TR,FNDX(I), SNDX(I),KD2
         TMPRS(FNDX(I), 1) = LOC(I)
 20    CONTINUE
 2     FORMAT(' ',2X,F7.0,2X,'P(',I1,',',I1,')',5X,'R',I4)
       END
C--------------------------------------------------------------
C      ALGORITHM TO CALCULATE THE  RATIO OF THE NUMBER OF
C      PARTITIONS IN AN AGGREGATE-CLASS TO THE NUMBER OF ROWS
C      IN A MESH.
C--------------------------------------------------------------
       SUBROUTINE GETMMM( PDATAC, M1, M2, M3,VPRS,MMM)
       INTEGER M1,M2,M3, PDATAC, VPRS
       LOGICAL MMM
       M1 = PDATAC/VPRS
       M2 = MOD(PDATAC, VPRS)
       IF (M2 .NE. 0) THEN
         M3 = VPRS/ M2
       ELSE
         M3 = 0
       ENDIF
       MMM = M1 .EQ. 0 .OR. M2 .EQ. 0 .OR. M3 .EQ. 0
       END
C--------------------------------------------------------------
C      DIVIDER AND ASSIGNER OF R PARTITIONS AMONGEST PROCESSORS.
C--------------------------------------------------------------
       SUBROUTINE DVDR(F,C,R,CNT,DIM,M1,M2,M3,MMM)
       INTEGER DIM,CNT
       INTEGER F(DIM), C(DIM), R(DIM)
       LOGICAL MMM
       CALL INITI(C,1,DIM,M1)
       IF(MMM) THEN
         N = 0
         DO 10 I = 1, DIM
           F(I) = N + 1
```

```
            N = N + M1
            IF ( I .LE. M2) THEN
              N = N + 1
              C(I) = C(I) + 1
            ENDIF
            R(I) = N
 10       CONTINUE
        ELSE
          DO 20 I = 1, M3 * M2
 20         C(I) = C(I) + 1
          M5 = M1 * M3 + 1
          M4 = 0
          K = 1
          DO 30 I = 1, DIM
            F(I) = M4 + 1
            M4 = M4 + M1
            IF (C(I) .GT. M1) THEN
              R(I) = M5 * K
              IF ( MOD(I,M3) .EQ. 0 ) THEN
                 M4 = M4 + 1
                 K = K + 1
              ENDIF
            ELSE
              R(I) = M4
            ENDIF
 30       CONTINUE
        ENDIF
        END
C--------------------------------------------------------------
C       READER OF DIRECTORY PARTITIONS
C--------------------------------------------------------------
        SUBROUTINE RDDIR(DIR1,DIR1C,TMPRS,II,JJ,
     *  TMBUS, QTYBUS, BUS, TMSTR, QTYSTR,VPRS,HPRS,TR,TC)
        INTEGER QTYBUS, VPRS, HPRS, QTYSTR
        INTEGER DIR1(6000,4), DIR1C
        INTEGER BUS(VPRS + HPRS)
        REAL TMPRS(II,JJ), TMBUS(QTYBUS)
        REAL TMSTR(0:QTYSTR-1)
        DO 30 I = 1, DIR1C
          KD1 = DIR1(I,3)
          KM1 = DIR1(I,1)
          KM2 = DIR1(I,2)
          CALL SYNC(TMBUS(1), TMSTR(KD1), TMPRS(1,1),TR)
C         WRITE(9,2) TMPRS(1,1)-TR,1,1,KM1
          CALL GETDIR(VPRS, HPRS, TMPRS, II, JJ, KM1,TC)
 30     CONTINUE
 2      FORMAT(' ',2X,F7.0,2X,'P(',I1,',',I1,')',5X,'D',I4)
        END
C--------------------------------------------------------------
C       ACCEPTOR AND SENDER OF DIRECTORY PARTITIONS
C--------------------------------------------------------------
          SUBROUTINE GETDIR(VPRS, HPRS, TMPRS, II, JJ, KM1,TC)
          INTEGER VPRS, HPRS, KM1
          REAL TMPRS(II,JJ)
```

```
         DO 10 I = 2, VPRS
           IF(TMPRS(I-1,1) .LT. TMPRS(I,1) - TC) THEN
              TMPRS(I-1,1) = TMPRS(I,1) - TC
           ELSE
              TMPRS(I,1) = TMPRS(I-1,1) + TC
           ENDIF
C          WRITE(9,2) TMPRS(I-1,1), I-1, 1, 'D', KM1, '===>'
           TMPRS(I-1,1) = TMPRS(I,1)
C          WRITE(9,2) TMPRS(I,1), I, 1, 'D', KM1
           TMPRS(I,1) = TMPRS(I,1) + TC
 10      CONTINUE
         DO 20 J = 2, HPRS
           IF(TMPRS(1,J-1) .LT. TMPRS(1,J) - TC ) THEN
              TMPRS(1,J-1) = TMPRS(1,J) - TC
           ELSE
              TMPRS(1,J) = TMPRS(1,J-1) + TC
           ENDIF
C           WRITE(9,2) TMPRS(1,J-1), 1, J-1, 'D', KM1, '===>'
           TMPRS(1,J-1) = TMPRS(1,J)
C          WRITE(9,2) TMPRS(1,J), 1, J, 'D', KM1
           TMPRS(1,J) = TMPRS(1,J) + TC
 20      CONTINUE
 2    FORMAT(' ',2X, F7.0, 2X, 'P(',I1,',',I1,')', 5X,A,I4,A)
        END
C-------------------------------------------------------------
```