PAPER NO.599

SUBCUBE MIGRATION IN MULTISTAGE INTERCONNECTION NETWORKS (PART-II)

BY

Muhammad Naeem Ayyaz

SUBCUBE MIGRATION IN MULTISTAGE INTERCONNECTION NETWORKS (PART-II)

Muhammad Naeem Ayyaz[1]

ABSTRACT

It is now possible to cost-effectively build small-scale to large-scale multiprocessors based on Multistage Interconnection Networks (MINs). Most efficient use of such systems would be to make them available in multi-user and multitasking environments. Scalable and near-optimal schemes for subcube migration have been designed and implemented in this part of this two-part study for MIN-based multiprocessors. This part depends upon concepts developed in part-I in [mna1]. Performance results for combined subcube compaction and migration schemes are presented. The schemes greatly enhance system throughput and utilization.

Keywords

Multistage interconnection networks, multiprocessors; network partitioning, network reconfiguration, subcube allocation, task migration.

INTRODUCTION

The multiprocessors that we are concerned with here are again based on the particular class of Multistage Interconnection Networks (MINs) called fully partitionable MINs. To ensure efficient system management and high system throughput, two resource management facilities can be provided within the multiprocessor. Firstly, the idle processors can be mutually combined to form the largest possible free subcubes by re-partitioning the underlying network; this is called Subcube Compaction. We gave a scalable and near-optimal solution for this problem in [mna1]. Secondly, some current task(s) can be migrated, from time to time, to create larger free subcubes; this is called Subcube Migration. The same notations and definitions used in [mna1] apply to this paper.

T. Schwederski, H.J. Siegel and T. L Thomas [tht88][tht89] recently showed that if the position of a source subcube and a target subcube is already known, then how a task can be actually migrated between the subcubes. We, however, address this problem in the context of a real multi-user environment, in as comprehensive and practical sense as possible. Accordingly, we give a sophisticated scheme for subcube migration, which is not only scalable, over different sizes of the multiprocessor, but also near-optimal. Whenever migration is attempted, we always create the largest free subcubes out of the current idle processors. Most importantly, we achieve the above by migrating tasks from a near-minimal number of source PEs to the near-minimal number of target PEs. Some of the notations used in [mna1] are repeated below for convenience.

[1] Department of Electrical Engineering, University of Engg. & Tech. Lahore, Pakistan.
Mnayyaz@ee.uet.edu.pk

SOME NOTATIONS

Let $S_{F,n}$ denote the set of all current free subcubes in the system, and $P_{F,n}$ the set of all free PEs contained in $S_{F,n}$. Let a set called the ultimate set $U_{F,n}$ contain a free i-cube for every ith '1' bit in the binary representation of $M = |P_{F,n}|$. This is that set of free subcubes that would form because of subcube migration operation. This is set is always guaranteed by the migration operation. Some more notations can be found in [mna1].

PROBLEM FORMULATION AND GOALS

- Migration problem is formulated around the following goals.
- Whenever migration is attempted, create the ultimate solution $U_{F,n}$ from the given $S_{F,n}$.
- Achieve the above by migrating tasks from a near-minimal number of source PEs to the same number of target PEs.
- Precisely identify all subcubes that would belong to the set of source subcubes, henceforth called set U. That is, determine the set of subcubes that form the above-mentioned near-minimal number of source PEs which should be migrated?
- Precisely identify the set of target subcubes (which contain the above-mentioned near-minimal number of target PEs), henceforth called set V.
- Also determine, for each given source subcube, the target subcube to which its task should be migrated?
- Create as many members of the set $U_{F,n}$ as possible without any migration at all.
- Accomplish these goals in as reasonable time as possible, for all possible sizes of the multiprocessor. This was the prime reason of introducing scalability in the solution.
- Integrate migration scheme into the overall allocation scheme in such a way that it will be attempted as infrequently as possible. Accordingly, migration would be attempted only when it is necessary and, most importantly, worthwhile to apply.

EXHIBITING SOME POWER OF THE MIGRATION SCHEME

Let us analyze the migration problem in a real practical situation. Suppose a free k-cube must be provided to satisfy some current request. In addition, although 2^k or more free PEs are currently available, yet no free i-cube, $i >= k$, exists there. In addition, suppose even a compaction operation wouldn't be able to create a free k-cube and as such migration is the only sure way left to create the k-cube.

The following simple example, which illustrates the above problem, was picked from actual simulation. It demonstrates some of the power of our migration scheme, and shows how it is possible to create a complete $U_{F,n}$ from the given $S_{F,n}$.

Example 1: Fig. 1 shows a D2Tree with $S_{F,5}$ = {1X1XX, 000XX, 100XX, 1100X, 0111X, 11011}. That is, $|P_{F,n}|$ = 21. The required $U_{F,5}$ must have one 4-cube, one 2-cube and one 0-cube.

(a) This $U_{F,5}$ can be formed in two possible ways. (i) The first one is to migrate 001XX into 000XX or 100XX, and then migrate 0110X into 1100X. This requires six one-PE migrations. (ii) The second possible way is to decompose 1X1XX into two 2-cubes and migrate 010XX into one of them, and then decompose 0111X into two 0-cubes and migrate 11010 into one of the 0-cubes. This would require five one-PE migrations.

(b) Our migration scheme, however, forms a preliminary D2Tree of Fig. 2 (a), in its first phase, and declares 11010 as one source subcube. The set {1XXXX, 000XX, 0111X} on the D2Tree represents the free subcube set out of which $U_{F,5}$ and a target subcube would be extracted. In its second phase, the scheme would identify 01110 as the required target subcube. After migration, a new D2Tree will emerge as shown in Fig. 2 (b).

Therefore, the power of our scheme is that with the help of only one one-PE migration, a whole $U_{F,5}$ = {1XXXX, 000XX, 01111} can be created. Compare this with five/six one-PE migrations needed in (i). Note that $U_{F,5}$ has three less free subcubes than the original $U_{F,5}$.

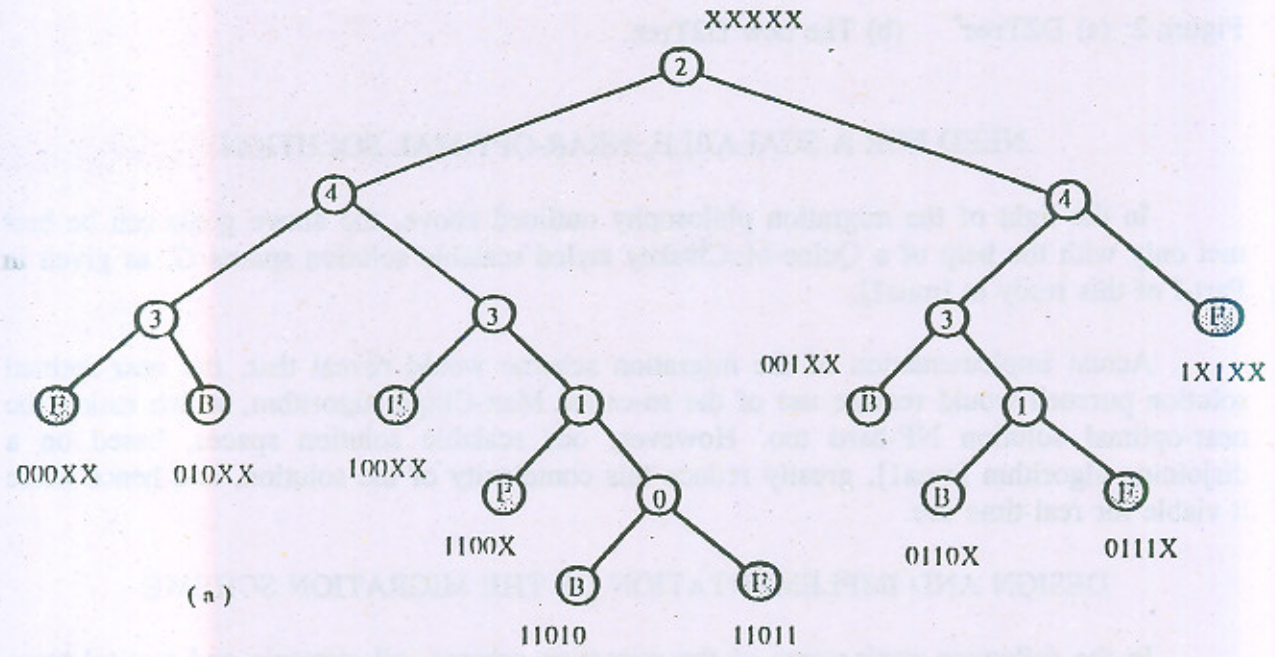


Figure 1: The original D2Tree, before migration.

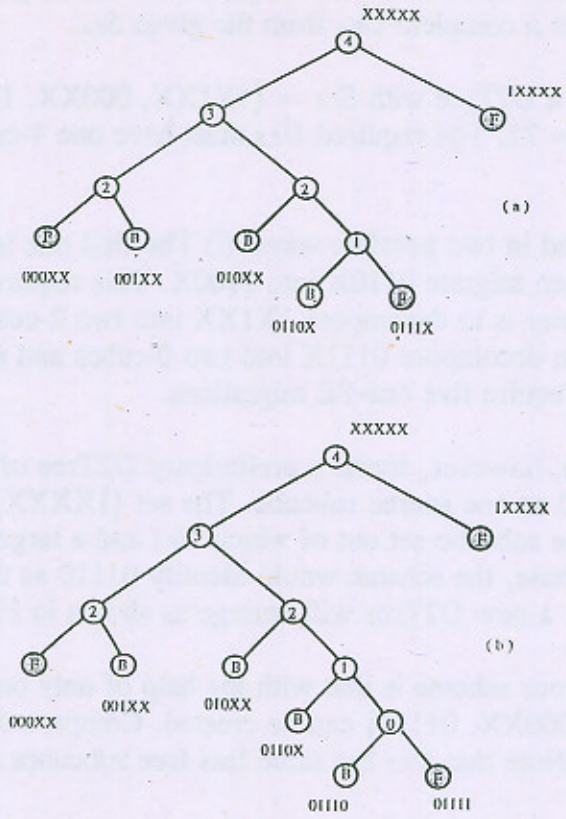


Figure 2: (a) D2Tree^P (b) The new D2Tree.

NEED FOR A SCALABLE, NEAR-OPTIMAL SOLUTION

In the light of the migration philosophy outlined above, the above goals can be best met only with the help of a Quine-McCluskey styled scalable solution spaces *G*, as given in Part-I of this study in [mna1].

Actual implementation of the migration scheme would reveal that, the near-optimal solution pursued would require use of the so-called Max-Clique algorithm, which makes the near-optimal solution NP-hard too. However, our scalable solution spaces, based on a disjoining algorithm [mna1], greatly reduce this complexity of the solution, and hence make it viable for real-time use.

DESIGN AND IMPLEMENTATION OF THE MIGRATION SCHEME

In the following explanation of the migration scheme, all strategic and tactical steps that contribute to accomplishing the problem goals mentioned-above will be highlighted as and when necessary. The migration scheme is divided into two phases, phase I and phase II.

Phase I:

In phase I of the solution, a so-called preliminary D2Tree (henceforth called D2Tree^P) is carved, level by level, out of the solution space G. The D2Tree^P is defined to represent a partitionally-disjoint (PD) subcube set which is a union of the non-migrating busy subcubes belonging to $S_{B,n}$-U and the free i-cubes, where $0 <= i <= floor(\log_2|P_{F,n}|)$, which would accumulate in the set *UltimatOrTrgt*. The set UltimatOrTrgt = $U_{F,n}$ U V would comprise those subcubes which make up the sets $U_{F,n}$ and V. As such, the subcubes belonging to *UltimatOrTrgt* are supposed to act (1) as targets of some primary/secondary migrations and (2) as members of the set $U_{F,n}$. Subcubes of the set U are also identified during this phase.

Phase II:

At the beginning of phase II, the target subcubes belonging to V are virtually unknown. They will be extracted one by one out of the set *UltimatOrTrgt*, now showing on D2Tree^P. For each source k-cube belonging to U its target subcube belonging to V will be identified by finding a nearest higher dimension free i-cube on D2Tree^P and decomposing it (if necessary) to create a target k-cube. This is equivalent to assigning a free subcube to the source subcube, and may be done using regular allocation algorithm. This process gives all source-to-target subcube pairs, indicated below by (U, V). The leftover set *UltimatOrTrgt* would represent the desired $U_{F,n}$. At this point, a new D2Tree has emerged out of D2Tree^P. Next all U subcube(s) to V subcube(s) migrations are actually performed, in the manner suggested in [tht89]. When the migration is over, network switches must be reset according to the new D2Tree.

The migration scheme is implemented with the help of algorithm *migration*(), and the accompanying procedures *Carve-D2Tree*^{P}(). More details can be found in [mna96].

SOME MORE NOTATIONS AND DEFINITIONS

First, some necessary notations and definitions are introduced which help explain general design of the migration scheme. Let F_{Gi}, $0 <= i <= n$-1, keep record of the dimension and number of the leaf/non-leaf free i-cubes that appear within subspace G_i belonging to G. As pointed out earlier, a D2Tree^P will be built, level by level, during phase I of the scheme. The migration scheme therefore can be explained, more naturally, around three abstract levels, namely level ℓ^{prev}, level ℓ^{curr} and level ℓ^{next}. The purpose of this notation is to be able to concisely describe the process of how different levels of the D2Tree^P will be actually created. One other purpose is to show how the source and target subcubes would actually come into being as the solution graph is traversed top to bottom, level by level.

(a) Level ℓ^{prev}: Suppose that a D2Tree^P was previously successfully carved, out of the solution space G, up to level ℓ^{prev}, where $\ell^{prev} = n$ - *DimAtLastPDLevel*. *DimAtLastPDLevel* represents dimension of the subcubes that correspond to the lowest level of the yet partially complete D2Tree^P. In the special case when ℓ^{prev} represents the root node level, *DimAtLastPDLevel* = n. Level ℓ^{prev} is thus defined to represent the currently lowest level of the yet incomplete D2Tree^P. Level ℓ^{prev} would also represent the last D2Tree^P level (with the exception of the root node level) at which one or more free subcubes will have been

previously formed. Similarly, let *NonleafAtLastPDLevel* represent the number of nonleaf (n- l^{prev})-cubes, not belonging to $S_{B,n}$- U and *UltimatOrTrgt,* that currently reside at level l^{prev}.

(b) Level l^{curr}: Suppose the D2Tree^P will be currently in the process of being extended, at the point of the non-leaf subcubes of level l^{prev}, to a new level l^{curr}. That is, l^{curr} would represent the next in line or the new lowest level up to which the D2Tree^P will be currently being extended. (n- l^{curr}), which is represented by
DimToAct in the algorithm, would also indicate dimension of the solution subspace which will be currently under consideration, and out of which subcubes of level l^{curr} will be extracted. In general, $(l^{curr} - l^{prev}) >= 1$.

(c) Level l^{next}: Suppose that after successfully extending the D2Tree^P up to the current level l^{curr}, it will have to be subsequently further extended (if necessary) up to another lowest level l^{next}, where $l^{next} = n$- *NextDimToAct*. *NextDimToAct* would thus also represent dimension of the subspace out of which the subcubes of level l^{next} will be picked. $(l^{next} - l^{curr}) >= 1$, in general.

SOME CRITICAL SUPPORTING ARRAYS USED

In the following, a few supporting (bookkeeping) arrays are introduced, which play an important role in actual overall design of the migration scheme. They are also an integral part of the scheme.

(a) We first define an array named *RequirdFSCs* [i], $0 <= i <= m$, $m = floor$ ($\log_2 |P_{F,n}|$), which, in the beginning, contains binary representation of $|P_{F,n}|$. Accordingly, each i belonging to {0,..., m}, such that *RequirdFSCs* [i] is not 0, would represent the dimension of a free i-cube of the sought-after solution $U_{F,n}$. In other words, it specifies that a free i-cube must be eventually available at level (n-i) of the new D2Tree, when the phase II will be finally over. Besides, as the D2Tree^P will be built, level by level, and as each new source busy j-cube will be discovered during the course of phase I, *RequirdFSCs* [j] will be incremented accordingly to reflect that one more target j-cube must be provided at level (n-j) of the D2Tree^P, so as to accommodate the new source j-cube. Hence, *RequirdFSCs* [i], i belonging to {0,..., m}, would in general collectively represent the number of free i-cubes that must be eventually provided on level (n-i) of the D2Tree^P, before the end of phase I.

(b) In the subsequent discussion, let *UltimatOrTargetFSCs* = *RequirdFSCs*[i] specify the number of required free i-cubes, $i = n$- l^{curr}, that must, at all cost, be created at some level l^{curr} of the D2Tree^P. And, if some or all of these i-cubes will have descendent busy g-cubes (under their corresponding nodes on the solution graph), $g < i$, those g-cubes must be migrated. These free i-cubes have to act as targets for some primary/secondary migrations, and/or serve as members of the ultimate set $U_{F,n}$. Thus, a related second supporting array is *ExcessFSCs*[]. At the start, *ExcessFSCs* [b] = 0, for all b, $0 <= b <= m$. If for some level l^{curr} of the D2Tree^P, the number of free subcubes found (from the solution space) would actually turn out to be greater than the required number *UltimatOrTargetFSCs*, then the value of the difference (the excess) will be added up into *ExcessFSCs* [n- l^{curr}]. This array in

fact would act as a dynamic run-time pool or reservoir indicating dimension and number of the current excess free subcubes available at the D2Tree^P levels l^{curr} and above. In other words, the array indicates that there exist excess free subcube(s) that are over and beyond the need of all D2Tree^P levels, from l^{curr} up to the root node level. Such excess subcubes are supposed to provide target(s) for accommodating some source busy g-cubes, $g < (n - l^{curr})$, or serve as possible members of the set $U_{F,n}$ having dimensions less than $(n - l^{curr})$.

When D2Tree^P will be subsequently considered for extension to a new level $l^{curr} > l^{curr}$, the *UltimatOrTargetFSCs* value corresponding to that level, *UltimatOrTargetFSCs* = *RequirdFSCs* $[n - l^{next}]$, would, initially, indicate the number of free subcubes that must at all cost be made available for that level. However, the final value of this number would be brought down by first tapping as many $(n - l^{next})$-cubes as possible from the pool *ExcessFSCs*[j], $j > (n - l^{next})$. This would minimize the number of free $(n - l^{next})$-cubes that must be finally created or found from the solution graph. Consequently, this would contribute to minimize the number of migrations that (otherwise) would have been caused because of he existence of descendent busy subcubes under non-leaf $(n - l^{next})$-cubes.

Algorithm *migration*()

```
BEGIN

  create the solution graph G<V,E>;

  Phase I:
  m = floor(log2(M));
  DimToAct = m;
  DimAtLastPDLevel = n;
  NonleafAtLastPDLevel = 1;

  FOR i = 0 TO m-1 DO
     ExcessFSCs [i] = 0;
  let S = n-cube;  //  subcube corresponding to root node
  U  = Carve-D2Tree^P();

  Build a D2Tree^P from the non-migrating busy subcubes belonging to
  S_B,n - U and all i-cubes in UltimatOrTrgt,  0 <=  i <= m;

  Phase II:
  FOR (each source k-cube Q belonging to U, m-1 >= k >= 0) DO
      identify a nearest higher dimension free i-cube Q' on D2Tree^P;
      carve, out of Q', a k-cube to act as target of the source Q;
  ENDFOR

  Migrate each source subcube belonging to U to corresponding target subcube belonging to
V;
```

Reset the network switches, as per new D2Tree;
END

Example 2: Consider D2Tree of Fig. 3 (a), where $S_{F,6}$ = {11XXXX, 101XXX, 0000XX, 0011XX, 00101X} and $S_{B,6}$ = {01XXXX, 100XXX, 0001XX, 00100X}. That is, $|P_{F,6}|$ = 34. Normally, at least 14 1-PE migrations must be performed to create a $U_{F,6}$. This can be done by migrating busy subcubes 100XXX, 0001XX and 00100X into the free subcube 11XXXX. However, when our migration scheme is applied, as described below, a complete $U_{F,6}$ can be created with the help of only 10 1-PE migrations. Let, initially, *UltimatOrTrgt* and U are only empty sets.

Phase I -

DimToAct = 5:

index: 5 4 3 2 1 0
RequirdFSCs: 1 0 0 0 1 0
F_G: 0 2 8 19 25 13
UltimatOrTargetFSCs = *RequirdFSCs* [5] = 1, F_G [5] = 0.
D2Tree^P comprises level 0 to 1, in Fig 3 (b).
UltimatOrTrgt = *UltimatOrTrgt* U {1XXXXX}
$U = U$ U {100XXX}
ExcessFSCs: 0 0 0 0 0 0

DimToAct = 3:

index: 5 4 3 2 1 0
RequirdFSCs: 1 0 1 0 1 0
F_G: 0 0 0 2 6 5
UltimatOrTargetFSCs = *RequirdFSCs* [3] = 1, F_G [2] = 0.
D2Tree^P comprises level 0 to 3, in Fig. 3 (b).
UltimatOrTrgt = *UltimatOrTrgt* U {00X0XX}
$U = U$ U {00100X}
ExcessFSCs: 0 0 0 0 0 0

DimToAct = 2:

index: 5 4 3 2 1 0
RequirdFSCs: 1 0 1 0 2 0
F_G: 0 0 0 1 2 2

UltimatOrTargetFSCs = *RequirdFSCs* [2] = 0, F_G [2] = 1.
ActualFSCsCreated = 1.
D2Tree^P comprises level 0 to 4, in Fig. 3 (b).
UltimatOrTrgt = *UltimatOrTrgt* U {0011XX}
ExcessFSCs: 0 0 0 1 0 0

At this point *RequirdFSCs*[1] = 2 can be supplied from *ExcessFSCs*[2] = 1, therefore phase I ends here. Finally, *UltimateOrTrgt* = {1XXXXX, 00X0XX, 0011XX} and $S_{B,6}$ - U = {01XXXX, 0001XX} would form the D2Tree^P of Fig. 3 (b).

Phase II -

(U, V) = {(100XXX, 00X0XX), (00100X, 00110X)}. Finally, new $S_{B,6}$ = {01XXXX, 00X0XX, 0001XX, 00110X} and $U_{F,6}$ = {1XXXXX, 00111X} would form the new D2Tree, as shown in Fig. 3 (c).

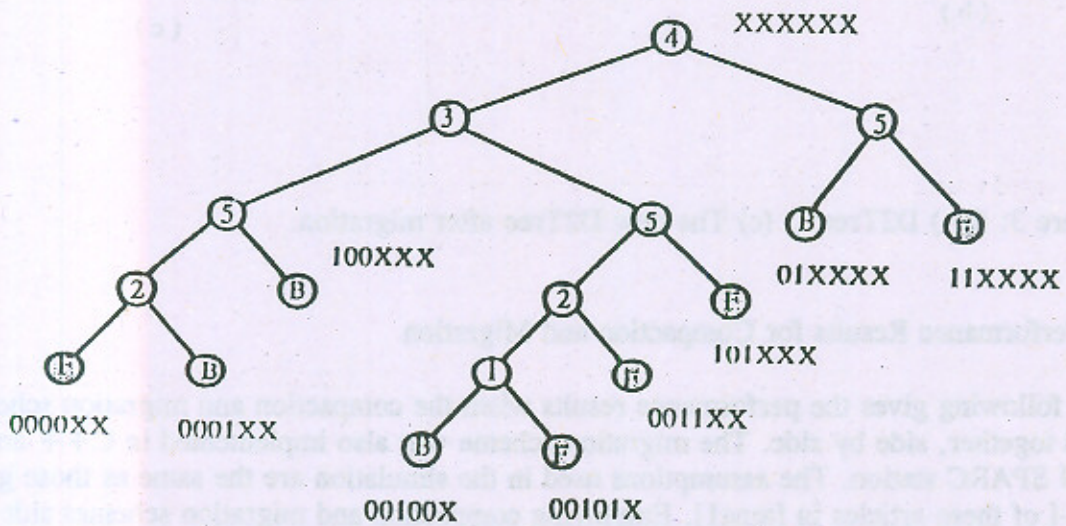


Figure 3: (a) Original D2Tree for Example 2.

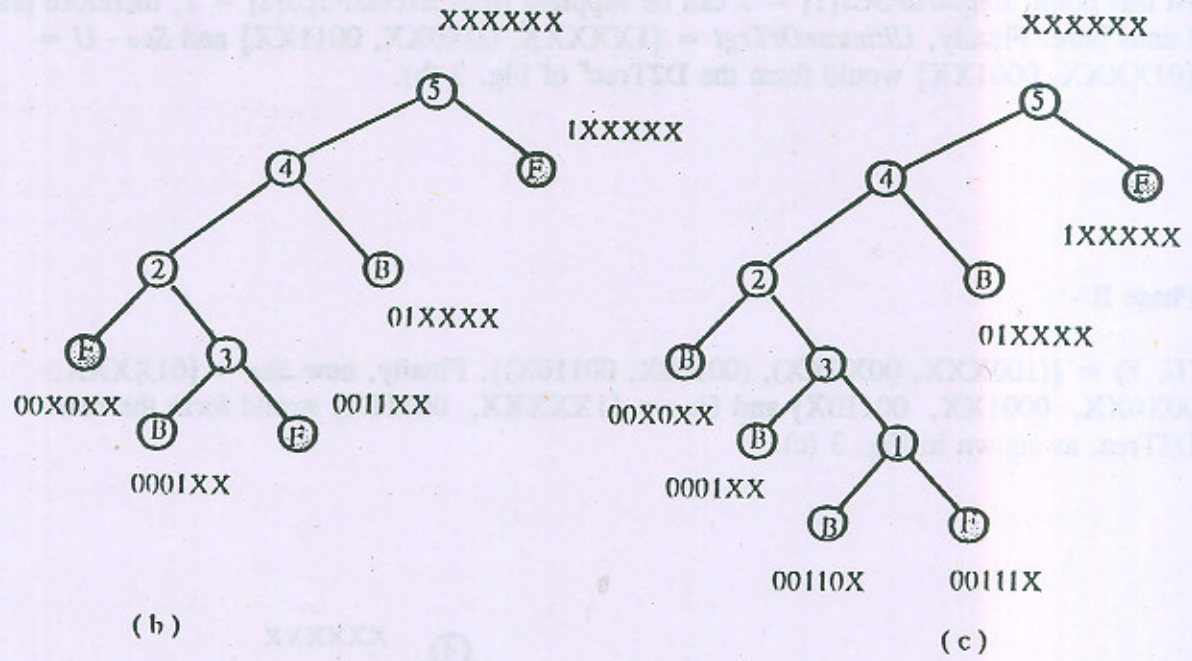


Figure 3: (b)) D2Tree^{P}, (c) The new D2Tree after migration.

9. Performance Results for Compaction and Migration

The following gives the performance results when the compaction and migration schemes are used together, side by side. The migration scheme was also implemented in C++ and run on SUN SPARC station. The assumptions used in the simulation are the same as those given in Part-I of these articles in [mna1]. Employing compaction and migration schemes side by side creates the following four-fold improvement in system performance:

(a) Up to 12% more jobs can be allocated in a given time.

(b) Overall, system utilization improves by as low as 3% to as high as 10%.

(c) The average waiting delay for each job is decreased by as low as 10% to as high as 50%.

(d) Higher dimension jobs, most importantly, now wait much less.

This is achievable with about five compactions and five migrations only for each 100 of the incoming requests. Most importantly, the average number of PEs, from which tasks have to be migrated during each migration, is also extremely small compared to the size of the system. For instance, for 16-PE, 32-PE, 64-PE and 128-PE systems, this number is only about 2, 3, 5 and 9 respectively. Consequently, migrations would incur only a minimal overhead and, as such, cause minimal negative effects on the system operation.

Table-I, for example, gives performance results, for medium-scale systems, when the compaction and migration schemes are used side by side.

Table-I Performance Results for Small-scale System

N	T	Resid dist	Compctns performd	Jobs assignd	Delay (d_h)	Utilztn (%)
16	100	$[2.5,5.5]^1$	None	77.5	8.71	83.38
			3.6	78.1	8.53	83.92
		$[2.5,6.5]^2$	None	78.9	7.87	85.01
			5.25	80.15	7.70	86.43
		$[2.5,8.0]^3$	No	79.34	7.82	81.96
			9.15	80.30	7.61	83.10
32	100	$[3,7]^1$	No	71.65	9.66	82.14
			4.25	72.05	9.50	82.80
		$[3.83,7.83]^2$	No	75.90	8.78	82.50
			7.6	77.45	8.60	83.56
		$[8,12]^3$	No	77.66	7.02	75.04
			19.72	79.56	6.85	76.51

REFERENCES

[cs90] M. S. Chen and K. G. Shin, Subcube Allocation and Task Migration in hypercube multiprocessors, . IEEE Trans. On Computers, C-39: 1146-1155, Sep. 1990.

[tht88] H. J. Siegel, T. Schwederski, and T. L. Casavant. A Model of Task Migration in Partitionable Parallel Processing Systems. In FMPC, pages 211-214, 1988.

[tht89] H. J. Siegel, T. Schwederski, and T. L. Casavant. Task Migration Transfers in Multistage Cube Based Parallel Systems. In Proc. 1989 Int. Conf. Parallel Processing, 1989.

[mna1] M.N. Ayyaz. Subcube Compaction in Multistage Interconnection Networks, Part-I, this issue.

[mna96] M. N. Ayyaz. Resource Management Schemes for Multiprocessors Based on Multistage Interconnection Networks. PhD thesis, Syracuse University, 1996.

Table 2: Performance Results for Medium-Scale Systems

N	T	Resid dist	Compctns/ Migratns	Avg PEs Migratd	Jobs assignd	delay (d_h)	Utilztn (%)
64	200	[3,8][1]	None	None	144.00	19.29	80.57
			2.42/2.43	6.26	149.87	18.21	83.26
		[4,11][2]	None	None	156.73	16.46	77.16
			5.61/6.51	5.30	171.60	11.89	84.45
		[8,14.5][3]	None	None	149.33	17.34	73.76
			8.01/6.16	4.05	167.20	13.06	83.10
128	200	[3.3,8.7][4]	None	None	147.60	18.11	79.20
			2.82/2.55	11.14	153.60	17.33	81.92
		[4.5, 14.2][5]	None	None	153.13	17.29	75.45
			6.85/6.8	8.75	170.17	11.98	84.10
		[9,16.5][6]	None	None	161.66	14.85	70.77
			10.03/5.80	7.73	181.3	9.23	79.51