

# A Unified Model for Sub-System Allocation on Product Networks

Jeeraporn Srisawat, Nikitas A. Alexandridis

[Jeera@seas.gwu.edu](mailto:Jeera@seas.gwu.edu), [alexan@seas.gwu.edu](mailto:alexan@seas.gwu.edu)

The George Washington University, Washington DC, U.S.A.

and

Tarek El-Ghazawi

[Tarek@science.gmu.edu](mailto:Tarek@science.gmu.edu)

George Mason University, Virginia, U.S.A.

## Abstract

Product networks are a class of interconnection networks that includes many useful networks such as multi-dimensional mesh, multi-dimensional torus, hypercube, and generalize hypercube. Most existing processor allocation methods are proposed for a specific interconnection network such as 2-D mesh or hypercube. In this research study, a “unified model” for processor allocation is proposed for partitionable product networks such as multi-dimensional mesh, multi-dimensional torus by using “a k-tree system state representation”. This unified model includes a general methodology of network partitioning, combining, searching, and best fit criteria, which can be scalable for any network belonging to the product networks class. It will be show that this model can be efficiently applied to existing useful product networks such as 2-D and 3-D meshes and toruses, yielding smaller time complexity and higher system performance than existing mesh-based algorithms.

**Keywords:** Unified model for sub-system allocation on product networks, k-Tree system state representation, partitionable product networks, and massively partitionable 2-D and 3-D mesh multicomputers.

## 1 Introduction

Product networks are a class of interconnection networks, producing many useful network topologies such as multi-dimensional mesh/torus, hypercube, and generalize hypercube. Recently, multi-dimensional mesh multicomputers trend to be the most efficient and scalable parallel architecture in constructing the high-performance parallel systems. Many existing prototypes and commercial k-D mesh systems have been built such as the 2-D Intel/Sandia ASCI System [17], the 2-D Intel Touchstone system [11], the 2-D Intel Paragon XP/S [12], the 3-D Tera Computer System [1], the 3-D Mosaic C [19], etc., for executing supercomputing applications (which usually require large sub-system sizes). Among those massively Mesh-connected multicomputers, the Intel products [11, 12, 17] support a multi-user environment (executing various independent applications in parallel), which an incoming task (or job) with a particular size is allocated on a sub-mesh by the operating system, residing in the host. In order to allocate those supercomputing applications (or jobs) in efficient decision time as well as to maintain the best performance for subsequence jobs (coming at run time) as possible, efficient processor allocation/deallocation algorithms have been developed to support those existing massively partitionable network systems.

In processor allocation research study area, most existing processor allocation strategies were developed for a specific network such as 2-D mesh or hypercube and were difficult to be efficiently modified for other networks. In 1990 a general idea of network partitioning on product networks [23] and in 1995 a generalized hypercycle-based processor allocation [8] were

introduced with some limitations. The general network partitioning study still needed to call an efficient existing topology such as mesh or hypercube when the partitioning process reached the basis step and the later hypercube-based study was limited in request sizes. For a specific 2-D mesh allocation, many strategies have been developed and all of them are classified into four main approaches: (1) “bit-map” approach i.e., First-Fit and Best-Fit bit-map (FF&BF) [24], FF with partition [18], etc., (2) “non-bit-map free list” approach i.e., 2-Dimensional Buddy (2DB) [14, 15], Free List (FL) [16], improved Free Sub-List (FSL) [13], etc., (3) “non-bit-map busy list” approach i.e. Frame Slide (FS) [2, 3], Adaptive Scan (AS) [9], Quick Allocation (QA) [22], Busy List (BL) [4, 5, 6], etc, and (4) our “non-bit-map quad-tree” approach i.e., first fit Q-Tree [20], the best fit Q-Tree [21]. Earlier mesh-based processor allocation studies were the first fit allocation (i.e., 2DB, FS, FF, AS, FFwp, QT, etc.) and lately studies were the best fit allocation for the better system performance (i.e., QA, BL, FL, FSL, BFQT, etc.).

In this research study, a “unified model” to processor allocation is proposed for product networks, which includes a k-tree system state representation, a general methodology of network partitioning, combining, searching, and best fit criteria. This model is also shown that it is efficiently applied on existing 2-D mesh network in terms of time complexity and system performance, compared with existing mesh-based algorithms and also expandable for other useful product networks such as 2-D toruses and 3D-meshes/toruses. In each application (i.e., 2-D or 3-D meshes), we try to accomplish following objectives: (1) complete recognition capability, (2) efficient searching time for finding the appropriate sub-system for each new incoming task in only  $O(N_a)$ , where  $N_a$  is the number of allocated sub-systems ( $N_a < N$ , where  $N$  is the system size), and (3) improved system performance. The system performance (i.e., system utilization, system fragmentation, and average waiting time) will be evaluated by using simulation studies. A number of experiments are done and their results are presented by investigating effects of applying our proposed algorithm and varying some parameters (i.e., system sizes, task sizes, workload, scheduling policy, etc.).

Next section describes existing studies in processor allocation on Meshes and a class of product networks. In Section 3, the unified model to processor allocation for product networks is proposed, which includes a k-tree system state representation, a general methodology of network partitioning, combining, searching, and efficient best fit criteria, and sub-system allocation/deallocation and task scheduling algorithm as well as its time complexity analysis. Section 4 presents efficient applications of the unified model to 2-D and 3-D meshes, compared to existing mesh-based strategies. Finally, conclusions are discussed in Section 5.

## 2 Related Studies in Processor Allocation

### 2.1 Processor Allocation on Meshes

The first fit 2DB (2-dim Buddy) strategy [14, 15], proposed in 1991, was an earlier research study in processor allocation for a square Mesh (of size  $N = 2^n$ ) by using an array of linked lists to store all available square sub-systems ( $2^p \times 2^p$ ,  $p \leq n/2$ ). Therefore, searching to find an available sub-system for a request was done in  $O(\log N)$  time and the sub-system combining process in order to accommodate deallocation was done in  $O(N)$  time. However, this strategy caused high internal fragmentation for any task sizes, especially non-(power of 2) square requests.

Later in 1991, the first fit FS (Frame Slice) processor allocation strategy [2, 3] was proposed as a solution of the internal fragmentation in the 2DB strategy. This method allowed any Mesh system ( $N = R \times C$ ) and any requested size ( $r \times c$ ) and a linked list was used to store only

assigned  $N_a$  sub-systems. In that approach, searching to find an available sub-system was starting at the left-most available processor, then a candidate frame was created and compared with allocated frames in the linked list. If it was not available, a new candidate frame was created by sliding horizontally by  $r$  (or vertically by  $c$ ) and the above comparing process was repeated until an available sub-system was found or it stopped when it could not slide to the next candidate frame. The time complexity of the FS was  $O(N_a \cdot N / (r \cdot Xc))$ . However, sliding the frame by a factor of  $r$  (or  $c$ ) might cause missing to find some available sub-systems.

In 1992, the efficient FF and BF (First Fit and Best Fit bit-map) strategies [24] was introduced in order to solve the allocation miss problem in the FS strategy. In such a bit-map approach, a 2D-array system status ( $N = R \times C$ ) was used to store free/busy bit-status of every processor. For an incoming request, all  $N$  bits had to be identified at least twice to find the corresponding available sub-system; therefore, the time complexity of this bit-map method was  $O(N)$ . In the BF bit-map, twice additional scanning were needed for finding the best sub-system. The bit-map strategy gave better system utilization than the FS up to 25%. However, this strategy did not have complete recognition capability since it did not provide task rotating.

In 1993, the first fit AS (Adaptive Scan) strategy [9] was presented as another solution for allocation miss in the FS strategy and claimed with complete recognition capability. It used a busy list of all allocated tasks (similar to the FS strategy) and included task rotating (which either  $S(r, c)$  or  $S(c, r)$  could be allocated for a requested task  $T(r \times c)$ ). The time complexity of the AS strategy to find the first available sub-system is  $O(N_a \cdot N)$  time and its performance results improved over the FS strategy.

In 1996, the combining approach (such as FSwp, FFwp, etc.) [18] was proposed to combine existing strategies (i.e., FF, FS, etc.) with predefined static partitions. This combining method improved time complexity as well as system performance by allocating requests with similar sizes close to each other (or into appropriate static partitions). If a system size was  $N = 2^j$ , the number of partitions were  $3\log(\sqrt{N})+1$ , and hence this reduced time complexity of existing strategies by a factor of  $\log N$  (or  $O(N/\log N)$ ). In simulation results, system fragmentation of the FFwp was almost identical to the FF bit-map strategy.

The best fit BL (Busy List) strategy was proposed in 1993 [4], improved in 1996 [5], and also in 1998 with reserved job scheduling [6]. This strategy improved time complexity as well as system performance over the BF (Best Fit bit-map) strategy by using “a busy list” to store only allocated sub-systems and the “maximum boundary value” best fit criteria. For an incoming task ( $r \times c$ ), all (up to 8) possible available candidate sub-system of size  $S(r, c)$  or  $S(c, r)$  were created from each 4 corners of each allocated sub-system, and then the candidate sub-mesh with maximum boundary value was stored. After all  $N_i$  allocated tasks were identified, the best candidate sub-mesh was obtained. The BL allocation was done in  $O(N_a^3)$  time but the deallocation time was  $O(1)$ . However, in the reserved job scheduling version, both allocation and deallocation complexity were  $O(N_a^3)$ , according to a number of iterations in the reservation process.

The best fit FL (Free List) strategy [16] was proposed in 1995 in order to improve time complexity and system performance over the BF strategy by using an array of linked lists to store all free sub-systems in increasing order by row (of all lists) and by column (in each list). For an incoming requested task ( $r \times c$ ), the first sub-system in list [ $r$ ] was considered. If it was larger than the request, then 2-8 candidate sub-systems of size  $S(r, c)$  or  $S(c, r)$  were created and one with maximum boundary value was selected. The FL time complexity is  $O(N_f^2)$  for both allocation and deallocation, where  $N_f$  was a number of free sub-systems.

In 1997, the best fit QA (Quick Allocation) strategy [22] was proposed in order to improve time complexity and average delay time. In that strategy, the following data structures were used: (1) a busy sub-system list (of allocated  $N_a$  tasks), (2) a coverage sub-system (CS) list, and (3) reject areas. For an incoming task  $r \times c$ , searching process was to find an available sub-system (started by computing the CS list and the reject areas), and for each row, the next process was to find a free sub-system (that did not intersect with the coverage sub-systems and the reject areas). The time complexity of the QA allocation and deallocation were  $O(N_a \sqrt{N})$  and  $O(1)$ . The average delay performance was improved over the AS.

In early 1998, the best fit FSL (Free Sub-List) strategy [13] was proposed to improve average waiting time by trying to perform the least amount of potential system fragmentation and preserve many large free sub-systems as possible for sub-sequence tasks. Among all free sub-systems, one that yielded the minimum degree of fragmentation will be selected for an incoming task. The time complexity of this FSL processor allocation/task was  $O(N_f^2)$ , deallocation time was  $O(N_f^3)$  for computing all free sub-systems, and task scheduling time for  $N_w$  tasks in a waiting queue was  $O(N_w N_f^2)$ . The simulation results showed that the FSL strategy improves performance 6-50% over the FL and BL strategies.

## 2.2 Processor Allocation on Product Networks

In 1990, product networks [23] were proposed as a unified theory, called “theory of Cartesian product networks”, which included common topological analysis (i.e., degree, diameter, and average distance), routing algorithms [10], embedding and network partitioning strategies. In that study, network partitioning was a process of dividing a network into sub-networks (or partitions) of various sizes for incoming tasks. In partition allocation, the “network manager” determined the smallest partition size that could allocate to the request and allocated if it was free. For a larger free partition, it was recursively partitioned until a partition of the appropriate size was obtained. If there was no free partition, then the request was queued according to a certain scheduling policy. When a partition was released, it was added into the pool (i.e., list structure) of free partitions and merging process was applied (if possible) to form larger free partitions to minimize fragmentation and improve system utilization. However, for a certain network (i.e., hypercube, mesh, etc.), this network partitioning strategy called any existing algorithms.

In 1995, hypercycle-based processor allocation [8] was proposed for hypercycles [7], a class of product networks, which included hypercube (binary n-cubes), multi-dimensional torus, etc. In this approach, the hypercycle was arranged in a linear number from 0 to  $N-1$ , where  $N$  was the number of processors. The first fit hypercycle-based allocation strategy utilized a list of allocation bits (numbered from 0 to  $N-1$ ), which was set to 1 if it is allocated; otherwise set to 0. The time complexity of a processor allocation was  $O(N)$  (or  $O(PN)$  if  $P$  permutations (or several lists) were used). Note that this approach was similar to the bit-map Gray code when applied on Hypercube; however it limited task sizes when applied on Torus and caused high internal fragmentation for any task sizes.

## 3 The Unified Model for Sub-system Allocation on Product Networks

Most existing algorithms were proposed for performing processor allocation on each network such as mesh and their methodologies were different based on data structures used to store allocation information, first fit/best fit criteria, and certain networks. Usually the algorithm yielding the best performance usually takes the longer time complexity. In this paper, a “unified

model” is proposed for sub-system allocation on product networks (i.e., multi-dimensional mesh and torus, hypercube, etc.) by using the same data structure, called a “k-tree”. When applied on a certain network (i.e., mesh, hypercube, etc.), this model is expected to improve both time complexity and system performance, compared with existing strategies.

**DEFINITION 1:** The Cartesian product [23] of  $k$  networks ( $G_i = (V_i, E_i)$ ,  $i = 1, 2, \dots, k$ , where  $V_i = \{1, 2, \dots, |V_i|\}$ ,  $E_i = \{e = \langle x, y \rangle \mid e \neq \emptyset; x, y \in V_i\}$ ) is defined as  $G = G_1 \times G_2 \times \dots \times G_k = (V, E)$ , where  $V = \{\alpha = (a_1, a_2, \dots, a_k) \mid a_1 \in V_1, a_2 \in V_2, \dots, a_k \in V_k\}$  and  $E = \{\langle \alpha, \beta \rangle, \alpha = (a_1, a_2, \dots, a_k), \beta = (b_1, b_2, \dots, b_k) \mid \text{there exists an } i \text{ such that } \langle a_i, b_i \rangle \in E_i \text{ (or } |a_i - b_i| = 1) \text{ and for } \forall j \neq i, a_j = b_j\}$ .

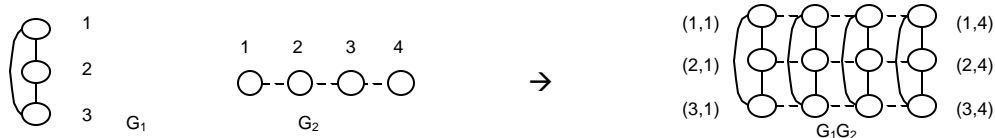


Figure 1: The product network of two networks:  $G_1$  is a ring of 3 nodes,  $G_2$  is a line of 2 nodes.

The product of two networks  $G_1 \times G_2$  (Figure 1) can be constructed either by taking  $|V_2|$  copies of  $G_1$  and connecting every set of the  $|V_1|$  corresponding nodes of these copies in  $G_2$  or by taking  $|V_1|$  copies of  $G_2$  and connecting every set of the  $|V_2|$  corresponding nodes of these copies in  $G_1$ . In this paper, assume a product network is constructed by using the maximum number of embedding network types first and for similar type networks, it is constructed by using the maximum size first.

### 3.1 System State Representation

A “k-tree” data structure is used to represent system states (or store allocation information) of a k-product network. A number of children/node, called buddies, will be derived by using criteria, based on the relation of the guest network and the host network. In this paper let 1) “host network” refers to a given k-product network ( $G_1 \times G_2 \times \dots \times G_k$ ); 2) “sub-network” refers to the j-product network ( $j < k$ ); 3) “system” refers to a certain well known network or product network (i.e., hypercube, mesh, torus, 3-d mesh, etc.); and 4) “sub-system” refers to the smaller system size (according to the guest network or the incoming request). In the unified model for a given product network, a number of buddies/k-tree’s node, are derived in general, as described next.

### 3.2 Network Partitioning Procedure

The partitioning procedure is used to partition a larger free network or system for a particular request. In the unified k-tree-based model, the proposed partitioning procedure on a given (available or free) product network for an incoming task with a certain network type and size, can be either 1) partitioning network type, 2) partitioning network size, or 3) combine 1 and 2, according to a particular request. Let define some data types used in general partitioning procedure, as follows:

```

Type tTNET = { line,ring,tree,complete,hypercube,mesh,mix }
                                                    /* free,busy,partially free */

Type tINFO = { k: int,                               /* k product networks */
              NetType: tTNET,                         /* (G = G1 x G2 x .. x Gk) */
              Gtype[i]:tTNET, i=1,2,..,k
              Gsize[i]:int, i=1,2,..,k }

Type tNODE = { Info: tINFO,                          /* info. of a k-tree's node */
              Status: tSTAT;
              ID: int;
              ParType: int;
              Base-Add: array(x1, x2,..,xk),
              ...

```

Let a given host network is a  $k$ -product network  $G_1 \times G_2 \times \dots \times G_k$  (of size  $N = n_1 \times n_2 \times \dots \times n_k$ ) with starting (base) address  $\alpha = (a_1, a_2, \dots, a_k)$  and ending (last cover) address  $\beta = (a_1+n_1-1, a_2+n_2-1, \dots, a_k+n_k-1)$ . In general, partitioning on a free  $k$ -tree's node (or host network) for a request task (or guest network) consists of two main steps, which is recursively partitioned from the host type ( $k$ ) into the request type ( $j < k$ ) and then from the sub-host size into the smaller request size.

**DEFINITION 2:** In buddy-based partitioning, a number of buddies/node (of a  $k$ -product network), each buddy's base-address, and each buddy's size can be defined as follows:

**CASE 1: Partitioning network type:** Let a guest network (or a requested type of an incoming task) is a  $j$ -product network, where  $j < k$ . If  $j = k-1$ , then the number of buddies/node in each partitioning is equal to the size of the  $G_k$  network ( $n_k$ ) and each buddy (or sub-network  $i$ ,  $i=1,2,\dots,n_k$ ) is a  $(k-1)$ -product network  $G_1 \times G_2 \times \dots \times G_{k-1}$  (of size  $n_1 \times n_2 \times \dots \times n_{k-1}$ ). The base address of each buddy  $i$  is  $\alpha_i = (a_1, a_2, \dots, a_{k-1}, i)$  which is different at the most significant digit and its last cover address  $\beta_i = (a_1+n_1-1, a_2+n_2-1, \dots, a_{k-1}+n_{k-1}-1, i)$ . Time complexity of the next level partitioning is  $O(n_i)$  and hence  $O(kn_i)$  for the  $k$ -level partitioning.

**CASE 2: Partitioning network size:** Let a guest network is a  $k$ -product network  $P_1 \times P_2 \times \dots \times P_k$  (of size  $p_1 \times p_2 \times \dots \times p_k$ ) where  $p_i \leq n_i$ ,  $i=1,2,\dots, k$ . The number of buddies/node in each partitioning is equal to  $2^k$ . For simplicity explanation, assume  $\forall j \ n_j = n$  (power of 2) and  $p_j = p$  (power of 2; and hence the base address of each buddy  $i$ ,  $i = 1, 2, \dots, 2^k$  is  $\alpha_i = (a_1', a_2', \dots, a_k')$ , where  $a_j' = a_j + (i/2^j * b_{j-1})$ ,  $j = 1, 2, \dots, k$  and  $(b_{k-1} \dots b_1 b_0) =$  bit-id of buddy  $i$ . For instance, given an  $8 \times 8 \times 8$ -mesh, stored in a node with a base address  $\alpha = (a_1, a_2, a_3) = (1, 1, 1)$ . Suppose it is partitioned into 8 buddies (of size  $4 \times 4 \times 4$ ). Then the bit-address of each buddy  $i$ ,  $i = 1, 2, \dots, 8$  is  $(b_2 \ b_1 \ b_0) = \{000, 001, 010, 011, \dots, 111\}$ , and hence the base-address is  $(a_1', a_2', a_3') = \{(1, 1, 1), (5, 1, 1), (1, 5, 1), (5, 5, 1), \dots, (5, 5, 5)\}$ . For any partitioning size (i.e., in 2-D mesh systems), if the first buddy size is  $p_1 \times p_2 \times \dots \times p_k$ , the general address conversion for each buddy  $i$  is  $\forall j = 1, 2, \dots, k$ ,  $a_j' = a_j + (p_j * b_{j-1})$  and each buddy's size  $\forall j \ n_j' = n_j - p_j$  if  $b_{j-1} = 1$ ; and  $n_j' = p_j$  if  $b_{j-1} = 0$ . Time complexity of next level partitioning is  $O(2^k)$  and  $O(2^k \log n)$  for the  $\log n$ -level equal-size partitioning.

### 3.3 Combining Procedure

In the unified  $k$ -tree model, the combining procedure is a process in any internal (partially free) node, which is used to combine smaller free nodes in to a larger sub-system.

**DEFINITION 3:** A sub-system ( $S$ ) in a given  $k$ -product network is represented by its start processor (or base address)  $\alpha = (a_1, a_2, \dots, a_k)$  and its size ( $n_1 \times n_2 \times \dots \times n_k$ ) and then the last processor (or cover address) in the sub-system is  $\beta = (b_1, b_2, \dots, b_k)$ , where  $b_j = a_j + n_j - 1$ . Any two sub-systems ( $S_1$  and  $S_2$ ) are adjacent along dimension  $i$  if  $|b_{1i} - a_{2i}| = 1$  or  $|a_{1i} - b_{2i}| = 1$ , where  $\alpha_1 = (a_{11}, a_{12}, \dots, a_{1k})$ ,  $\alpha_2 = (a_{21}, a_{22}, \dots, a_{2k})$ ,  $\beta_1 = (b_{11}, b_{12}, \dots, b_{1k})$ , and  $\beta_2 = (b_{21}, b_{22}, \dots, b_{2k})$ .

**DEFINITION 4:** According to buddy-based partitioning (defined in Definition 2), a number of combining sub-systems can be performed as follows:

**CASE 1: Combining network type:** For a partitioning network type, there are  $n_k$  buddies/node (at level  $L$ ) with some allocated buddies for tasks and possible combining of other free buddies are

- i) Combine all  $n_k$  buddies (of size  $n_1 \times n_2 \times \dots \times n_{k-1}$ ) at level  $L$  into a larger  $k$ -tree's node at level  $L-1$  (of size  $n_1 \times n_2 \times \dots \times n_{k-1} \times n_k$ ), if they are available (or free) in  $O(n_k)$  time, which is possible after deallocation of a finished task to maintain maximum free node in the  $k$ -tree (as possible).

- ii) Combine a number (2, 3, .., or  $n_k-1$ ) of adjacent free buddies (of size  $n_1 \times n_2 \times \dots \times n_{k-1} \times 1$ ) at level L into a larger sub-network in order to allocate for any request that larger than a buddy but equal to or less than the combined sub-network. Let  $n$  ( $1 < n < n_k$ ) be the number of combined buddies, and hence there are  $(n_k-n+1)$  possible results of each combining size ( $n_1 \times n_2 \times \dots \times n_{k-1} \times n$ ). Time complexity of finding the first free  $n$  adjacent buddies is  $O(n_k)$  time and hence  $O(n_k^2)$  for all possible combining.
- iii) For a complete recognition capability, a number of adjacent free nodes in other dimensions ( $1 \leq i < k$ ) or at higher level (L+1, L+2, ...) can be combined into a larger sub-system “(k-1)-product network”. At level L, each buddy node represents a (k-1)-product network (of size  $n_1 \times n_2 \times \dots \times n_{k-1} \times 1$ ) with partitioning along the network (or dimension) k. Each of other (k-1)-product networks (or  $n_i = 1, 1 \leq i < k$ ) can be recognized by combining free nodes at level L+k-i, which there are  $n_i$  possible combined sub-systems (of size  $n_1 \times \dots \times n_{i-1} \times 1 \times n_{i+1} \times \dots \times n_k$ ) by combining  $(n_{i+1} \times \dots \times n_k)$  free nodes (of size  $n_1 \times \dots \times n_{i-1}$ ), where  $n_0=1$ , and  $i=k-1, k-2, \dots, 1$ , for combining at level L+1, L+2, ..., L+k-1 respectively. This combining process should be computed only once after applying the combining in (ii) to all nodes in the k-tree and there is no free sub-system in  $O(m)$  time, where  $m$  is a number of node in the k-tree.

CASE 2: Combining network size: For a partitioning network size, there are  $2^k$  buddies/node (at level L) with some allocated buddies for tasks and possible combining of other free buddies are

- i) Combine all  $2^k$  buddies at level L into a larger k-tree’s node at level L-1, if they are free in  $O(2^k)$  time which is possible after deallocation of a finished task.
- ii) Combine a number (2, 4, ..., or  $2^{k-1}$ ) of adjacent free buddies at level L with the same root for a task size that is larger than a buddy but less than the combined sub-systems. There are  $k$  ( $2^{k-j}$ ) possible combined sub-systems to combine  $2^j$  adjacent free buddies,  $1 \leq j < k$  and by using the “adjacent address conversion rule”: a set of buddy-id =  $\{1, 2, \dots, 2^j\} \rightarrow$  a set of bit-address =  $\{0..00, 0..01, \dots, 1..11\} \rightarrow$  a set of base-address =  $\{(a_1', a_2', \dots, a_k') \mid \forall j = 1, 2, \dots, k, a_j' = a_j + (p_j * b_{j-1})\}$  & a set of size =  $\{(p_1', p_2', \dots, p_k') \mid \forall j, n_j' = n_j - p_j \text{ if } b_{j-1} = 1; n_j' = p_j \text{ if } b_{j-1} = 0\}$ , where  $(a_1, a_2, \dots, a_k) = \text{root's base address}$ ,  $(n_1 \times n_2 \times \dots \times n_k) = \text{root's size}$ , and  $(p_1 \times p_2 \times \dots \times p_k) = \text{first buddy's size}$ . Then, at level L, binary-ids (a permutation of  $j^*$ 's & (k-j) 0/1) of  $2^j$  buddies are identified, which can be converted into combinable buddy-ids (or integer  $(b_{k-1} \dots b_1 b_0) + 1$ ). Time complexity of the above combing process to combine  $2^j$  buddies is  $k(2^{k-j})$ . For instance, given a  $8 \times 8 \times 8$ -mesh (stored in the root at level 1) with 8-buddy partitioning. A set ( $3 \times 2^{3-2} = 6$  elements) of combining 4 adjacent buddies is  $\{0**, *0*, **0, 1**, *1**, **1\}$ , which is converted into a set of combinable buddy-id  $\{(1, 2, 3, 4), (1, 2, 5, 6), \dots, (2, 4, 6, 8)\}$ . Similarly, a set ( $3 \times 2^{3-1} = 12$ ) of combining 2 adjacent buddies is  $\{00*, 0*0, *00, \dots, *11\}$ , which is converted into a set of combinable buddy-id  $\{(1, 2), (1, 3), (1, 5), \dots, (4, 8)\}$ .
- iii) For complete recognition capability, a number of adjacent free nodes at level L+1 can be combined to a larger sub-system within  $k(2^{k-j})$  time. At level L+1, if any partitioning size is applied, there are  $K(2^{k-1})$  possible combined sub-systems by combining  $2^{k-1}$  nodes (at level L+1) per each of 2 adjacent partially free buddies if they are free. In this case, the methodology is similar to equal partitioning when  $J = 1$ . Under any size partitioning and combining, another possible combining whose size larger than each buddy is the combining of a free buddy (at level L) and  $2^{k-1}$  free sub-buddies (at level L+1) of its combinable partially free buddy. In this case, the “1-2-level conversion rule” is: let  $C = (c_{k-1} \dots c_1 c_0)$  is a possible combined binary-id. For each integer  $I = 1, 2, \dots, 2^j$ , let  $T = (t_{j-1} \dots t_1 t_0) = \text{ConvertIntegerIntoJ-Bit}(I, J); \forall k=0, 1, \dots, K-1$ , let Buddy bit-id  $B = (b_{k-1} \dots b_1 b_0) = \text{Replace* withBitT}(C, T, J)$ ; Sub-buddy bit-id  $S = (s_{k-1} \dots s_1 s_0) = \text{Replace* withBitT and 0/1 with*}(C, T, J)$ ; by let  $i = 0$ ; for  $k = 0, 1, \dots, K-1$ , if  $c_k = 0$  or 1, then  $b_{ik} = c_k$  and  $s_{ik} = *$ ; if  $c_k = *$ , then  $b_{ik} = t_i$  and  $s_{ik} = t_i + 1$ .

### 3.4 Best Fit Criteria and Procedure for Allocation

The best fit criteria is an improved performance policy by trying to find the best sub-system that trend to cause minimum future system fragmentation as possible.

DEFINITION 5: Best Fit Criteria for the unified k-tree model: To find the best sub-system (including task rotating) for an incoming task that can preserve the maximum free sub-system, gives the minimum different size factor, and yields the minimum combining factor (see Definition 6) since it trends to cause minimum system fragmentation and preserve the maximum free sub-system after an allocation. The priority of the best fit criteria is

- 1) Find all free sub-systems that can preserve the maximum free sub-system as possible (i.e., disjoint first among a set of {disjoint, intersect, sub-set} status results. For each candidate, only the best rotating size of k possible rotating sizes (for only partitioning by size) is stored.
- 2) If there are many candidates (whose size  $\geq$  the request) that have the same property in (1), then the candidate that gives the minimum different size factor (i.e., diff SF = 0 if all k sizes of the request and the free sub-system are equal) is selected.
- 3) If there are many candidates (whose size  $\geq$  the request) that have the same property in (1) and (2), then the smallest candidate that yields the minimum combining factor is selected (or the minimum probability of combining with the closet adjacent node first). Otherwise, select the first candidate or by random.
- 4) After searching on all nodes in the k-tree, if the best free sub-system is equal to the request, then allocate it to the request; otherwise (if it is larger than the request), then it will be partitioned and one of its buddies with yielding the properties similar to that in step 1 and 3 will be selected as the best sub-system and allocated to the request. Note: for partitioning by size the best selected buddy is the candidate with yielding the best task rotating at the best corner (of k locations).

DEFINITION 6: The combining factor (CF) of any sub-system S (assume at level L) is computed from its adjacent neighbor nodes as a summation of the probability of combining (PC) of combinable nodes of the same root with S of the same sub-trees at Level L-1, L-2, and L-3, respectively in only O(1) for partitioning/combining type or O(k<sup>2</sup>) for partitioning/combining size with task rotating. Let PC for each size or dimension of a sub-system S is defined as

$$\begin{aligned} PC &= 0, \text{ if the status of S's combinable (or adjacent) node is busy (or 1)} \\ &= 1, \text{ if the status of S's combinable (or adjacent) node is free (or 0)} \\ &= \frac{1}{2}, \text{ if the status of S's combinable (or adjacent) node is partially free (or x)} \end{aligned}$$

CF ( $\alpha$ ) is the combining factor of  $\alpha$  with adjacent nodes, such as

$$\begin{aligned} PC(\alpha, \beta) &= PC(\alpha, \beta_1) + PC(\alpha, \beta_2) + \dots + PC(\alpha, \beta_b) \text{ is the combining factor of } \alpha \text{ at level L-1} \\ PC(\alpha, \gamma) &= PC(\alpha, \gamma_1) + PC(\alpha, \gamma_2) + \dots + PC(\alpha, \gamma_c) \text{ is the combining factor of } \alpha \text{ at level L-2} \\ PC(\alpha, \delta) &= PC(\alpha, \delta_1) + PC(\alpha, \delta_2) + \dots + PC(\alpha, \delta_d) \text{ is the combining factor of } \alpha \text{ at level L-3} \end{aligned}$$

Let  $\alpha$  denotes a bit-id ( $b_{k-1} \dots b_1 b_0$ ) of a considering node at level L

$$\begin{aligned} \beta_1, \beta_2, \dots, \beta_b &\text{ denotes bit-ids of buddy node(s) of } \alpha \text{ with the same root of a sub-tree at level L-1} \\ \gamma_1, \gamma_2, \dots, \gamma_c &\text{ denotes bit-ids of adjacent node(s) of } \alpha \text{ with the same root of a sub-tree at level L-2} \\ \delta_1, \delta_2, \dots, \delta_d &\text{ denotes bit-ids of adjacent node(s) of } \alpha \text{ with the same root of a sub-tree at level L-3} \\ &\text{ where b, c, d are assigned according to the partitioning methodology.} \end{aligned}$$

For partitioning by type, the value of b, c, d are either = 2 if #processors/dimension (or network) > 2 (i.e., if integer ( $\alpha$ ) = i, then integer ( $\beta_1$ ) = i-1 and integer ( $\beta_2$ ) = i+1, etc.) or = 1 if #processors/dimension = 2 (i.e., if integer ( $\alpha$ ) = i, then integer ( $\beta_1$ ) = (i+1) mod 2).

For partitioning by size, the  $b, c, d = k$  (i.e., let  $\alpha = (b_{k-1} \dots b_1 b_0)$ , then  $\beta_1 = (b_{k-1} \dots b_1 b_0')$ ,  $\beta_2 = (b_{k-1} \dots b_1' b_0)$ , ...,  $\beta_k = (b_{k-1}' \dots b_1 b_0)$ ; let  $\text{root}(\alpha) = (r_{k-1} \dots r_1 r_0)$ , then  $\gamma_1 = (r_{k-1} \dots r_1 r_0')$ ,  $\gamma_2 = (r_{k-1} \dots r_1' r_0)$ , ...,  $\gamma_k = (r_{k-1}' \dots r_1 r_0)$ ; and let  $\text{root}(\text{root}(\alpha)) = (R_{k-1} \dots R_1 R_0)$ , then  $\delta_1 = (R_{k-1} \dots R_1 R_0')$ ,  $\delta_2 = (R_{k-1} \dots R_1' R_0)$ , ...,  $\delta_k = (R_{k-1}' \dots R_1 R_0)$ . For instance, given a 8x8-mesh. Let  $\alpha = (b_1 b_0) = (11)$ , residing at level 4 and 2 adjacent buddies of  $\alpha$  are  $\beta_1 = 10$  and  $\beta_2 = 01$ . Assume  $\text{root}$  of  $\alpha$  at level 3 = 10, 2 adjacent nodes of  $\text{root}(\alpha)$  are  $\gamma_1 = 11$  and  $\gamma_2 = 00$  and if previous root at level 2 = 00, 2 adjacent nodes of previous root ( $\alpha$ ) are  $\delta_1 = 1$  and  $\delta_2 = 10$ .

$\gamma_2$ 00		$\beta_1$ 10	$\delta_2$ 10
	$\beta_2$ 01	$\alpha$ 11	
	$\gamma_1$ 11		
$\delta_1$ 01			

### 3.5 Searching Procedure

In an “allocation”, the depth-first-search algorithm will be applied for finding the first (or the best free sub-system). If the operation in each node is  $O(1)$  time, then time complexity of the search procedure will be similar to that of the depth-first-search algorithm, which is  $O(m)$ , where  $m$  is the number of nodes in the  $k$ -tree. In particular, searching on a given product network, for an incoming task with a certain type and size, consists of two main steps that is the operation of a leave (free) node and the operation of a internal (partially free) node. The searching starts from the root node of the  $k$ -tree and then go to the first left most node (a leave node) and using depth first search algorithm to visit all nodes to find the best sub-system.

```
BestFitSearchingForAllocation (kTnode: tNODE, Task: tINFO)
{
    bestS = kTnode; /* initialize the best node */
    bestCT = InitializeBestValue(); /* the best value(criteria) */
    bestS = FindBestSubSystem (kTnode, Task, bestS, bestCF);
    if (bestCT != InitializeBestValue()) {
        if (Size(bestS) > Size(Task)) bestS = BestPartitioning (bestS, Task);
        AllocateAndUpdateKTree (bestS, Task);
        Return finishAllocate; /* allocate the best sub-system */
    }
    else {
        freeSystem = AllLevelCombineForLargerSystem(kTnode);
        if (Size(freeSystem) < Size(Task)) return NoAllocate;
        else if (Size(freeSystem) > Size(Task)) freeSystem = Partitioning(freeSystem, Task);
        AllocateAndUpdateKTree(freeSystem, Task);
    }
}
```

```
FindBestSubSystem (kTnode: tNODE, Task: tINFO, bestS: tNODE, bestCT: tBEST)
{
    k = kTnode.Info.k;
    /* ----- Leave node Operation (status = '0' or '1') ----- */
    if (IsLeave(kTnode)=true & kTnode.Status = '0') {
        subSystem = BestPartitioningSubSystemWithTaskRotating (kTnode, Task);
        if (subSystem != NULL) CF = ComputeBestFitValue(subSystem, Task);
        if (Better(bestCF, CF)) UpdateBestFitSubSystem(subSystem, bestS, bestCF, CF);
        return bestS;
    }
    /* ----- Depth First Search (go to next left most node) ----- */
    if (kTnode.Info.ParType = 1) nBuddy = kTnode.Info.Gsize[k];
    else nBuddy = 2k; /* #buddy = nk (CASE 1) or 2k (CASE 2) */
    for (i=1; i<=nBuddy; i++)
        FindBestSubSystem (kTnode->Buddy[i], Task, bestS, bestCF);
    /* ----- Internal node Operation (status='x'(partially free))-----*/
    ListSubSystem = Combining (kTnode, Task);
    subSystem = ListSubSystem->head;
    while (subSystem != NULL) {
        CF = ComputeBestFitValue(subSystem, Task);
        if (Better(bestCF, CF)) UpdateBestFitSubSystem(subSystem, bestS, bestCF, CF);
    }
    return bestS;
}
```

In the “deallocation” procedure, the intersection-buddy search algorithm will be simply applied for finding the location (or k-tree’s node) that store the finished task by recursively searching from the root to the finished node. After finding the node that store allocation information of the finished task, its status is updated from busy (1) to free (0) and then the combining process will be applied to combined all buddies if they are free. This combining process will be recursively applied from the finished node to the root (if possible).

```

LocationSearchingForDeallocation (kTnode: tNODE, finishT: tNODE)
{
    /* ----- Deallocate the finish node ----- */
    if (kTnode = finishT)
        FreeKTreeNodeAndUpdateKTree (kTnode, finishT);
    else /* ----- Searching into intersection path only ----- */
        if (kTnode.Info.ParType = 1) nBuddy = kTnode.Info.Gsize[k];
        else nBuddy = 2k; /* #buddy = nk (CASE 1) or 2k(CASE 2) */
        for (i=1; i<=nBuddy; i++) {
            if (Intersection(kTnode->Buddy[i], finishT) = true)
                LocationSearching (kTnode->Buddy[i], finishT);
        }
    /* ----- Combining processing ----- */
    if (AvailableAllBuddyNodes(kTnode, nBuddy) = true) {
        UpdateKTreeNode(kTnode);
        RemoveAllBuddyNodes(kTnode, nBuddy);
    }
}

```

### 3.6 Sub-System Allocation/Deallocation and Task Scheduling

In the dynamic processor allocation and task scheduling, when there is an incoming task, if the wait priority of the first task in the queue is high (more than a threshold value), then that task will be enqueued in the waiting queue; otherwise, the processor “allocation” procedure will find for the request the first or the best available sub-system by searching into the k-Tree. If there exists an available sub-system, then the request will be allocated on the system. Otherwise (no available sub-system), the request will be enqueued. When a task is finished, the processor “deallocation” procedure will find the assigned position of that task by searching into the k-Tree, in order to free the k-Tree’s node that stores information of that task and then combine available sub-partitions (if possible). At this time, if there are task(s) in the waiting queue, FCFS scheduling will be applied to perform scheduling and allocation these waiting tasks.

### 3.7 Time Complexity Analysis

**THEOREM 1:** The time complexity of the k-Tree-based allocation to find the best sub-system for each incoming task on a k-product network (of size  $N = n_1 \times n_2 \times \dots \times n_k$ ) is  $O(n_i^2(N_f + N_a))$  time for partitioning network type or  $O(k^2 2^{2k}(N_f + N_a))$  time for partitioning network size,  $n_i$  is the size of the network  $i$ ,  $N_a$  is the number of allocated tasks (or busy k-tree nodes), and  $N_f$  is the number of free nodes.

*PROOF:* A number of recursive iterations are at most a number of nodes in the k-tree since only nodes, whose types or sizes are larger than or equal to the request, are visited. In this non-bit map approach, the number of nodes in the k-Tree are proportion to the number of tasks allocated or busy nodes ( $N_a$ ) and the number of free nodes ( $N_f$ ) in the k-tree. Since the number of leaf nodes in the k-tree are at most  $N_a + N_f \leq N$  and the number of internal nodes are at most  $(\#leaves - 1) / (b - 1)$ , where  $b$  is the number of buddies/node; therefore the total nodes in the k-Tree are at most  $[(N_a + N_f) + (N_a + N_f - 1) / (b - 1)]$  nodes. For each leaf node in the k-tree, only an operation of the computing of the best fit value (or the combining factor (CF) of a free node) is needed, which can be computed in  $O(n_i)$  time for partitioning network type or in

$O(k^2 2^k)$  time for partitioning network size with task rotating (see Definition 2 and 6). For each internal node, major operations are the computing of a number of free combined sub-systems, which can be computed in  $O(n_i^2)$  time for partitioning network type or in  $O(k^2 2^{2k})$  time for partitioning network size (see Definition 4) and plus the computing of the best fit value, which is  $O(n_i)$  or  $O(k^2 2^k)$  time. Note: recursive partitioning will be applied only once for the best free sub-system that larger than the request in  $O(b)$  time. Also, the combine other dimensions (or all level combining) for larger free sub-systems under combining network type will be applied only once in  $O(N_a + N_f)$  time after cannot find a free sub-system by searching with regular combining. Hence, the time complexity to visit all nodes is  $O(n_i^2(N_a + N_f))$  for partitioning network type or  $O(k^2 2^{2k}(N_f + N_a))$  for partitioning network size, where  $N_a + N_f \leq N$ .

Note: For a 2-D mesh/torus application, all partitioning in the  $k$ -product network is based on the partitioning network size, and hence the allocation time complexity is  $O(N_a + N_f)$  since  $k = 2$  and  $2^k = 4$  (constants). Similarly for a 3-D mesh/torus, the allocation time complexity is also  $O(N_a + N_f)$  since  $k = 3$  and  $2^k = 8$ . If assume that  $N_a > N_f$ , then allocation time complexity is  $O(N_a)$  time for both 2-D and 3-D mesh applications.

**THEOREM 2:** The time complexity of the  $k$ -tree-based deallocation to free the  $k$ -Tree node(s) that store the finished task (sub-system) and to combine the maximum free size of the corresponding  $k$ -Tree's nodes on a  $k$ -product network (of size  $N = n_1 \times n_2 \times \dots \times n_k$ ) is  $O(b k n_i)$  time, where  $b$  is the number of buddies/ $k$ -tree's node ( $b = 2^k$  or  $n_i$ ) and  $n_i$  is the maximum size of a network of the  $k$  networks.

*PROOF:* Since the maximum depth of the  $k$ -tree is  $k$  ( $n_i$ ), the searching for the location of a finished sub-system from the root is at most  $k$  ( $n_i$ ) steps (in worst case) and also the combing  $b$  buddy nodes (if possible) from the finished sub-system to the root takes another  $b k$  ( $n_i$ ) steps. Therefore, the time complexity of the  $k$ -tree-based deallocation is  $O(b k n_i)$  time.

Note: For a 2-D mesh/torus application, the deallocation time complexity is  $O(n_i)$  since  $k = 2$  and  $2^k = 4$ . Similarly for a 3-D mesh/torus, the deallocation time complexity is also  $O(n_i)$  time since  $k = 3$  and  $2^k = 8$ .

## 4 Application of the Unified Model

The unified model is applicable to all product networks including many useful topologies such as 2-D and 3-D meshes and toruses. In this section, we design to have the unified model that can be applied by using the specific case of 2-D meshes and 3-D meshes.

### 4.1 Application of the Unified Model to 2D-Meshes

When applying the unified model to 2-D Mesh multicomputers, our approach is very efficient in time complexity and system performance effect. The  $k$ -tree system states representation is used for the faster allocation/deallocation; and hence time complexity of the sub-system allocation decision is only  $O(N_a + N_f)$  or  $O(N_a)$  if assume  $N_a > N_f$ , which is better than those of existing mesh-based strategies (i.e.,  $O(N_a \sqrt{N})$  in the quick allocation (QA),  $O(N_f^2)$  in the free sub-list (FSL),  $O(N_a^3)$  in the busy list (BL), etc.), where  $N_a$  is the number of allocated tasks,  $N_f$  is the number of free sub-meshes ( $N_a + N_f \leq N$ ), and  $N$  is the system size. In the application on a mesh of size  $N = R \times C$ , let  $P(i, j)$  denotes the PE at the coordinate address  $\langle i, j \rangle$  (or  $i^{\text{th}}$  row and  $j^{\text{th}}$  column), where  $P(1,1)$  denotes the PE at the Top Left (or upper leftmost position) and  $P(R,C)$  denotes the PE at the Bottom Right (bottom rightmost position) of the system. Hence, the

system address is represented by using two coordinates, which are Top Left point (TLpt) =  $\langle 1,1 \rangle$  (or base address) and Bottom Right point (BRpt) =  $\langle R,C \rangle$  (which is computed by using the base address and system size). Let a sub-system  $S(r, c)$  of size  $p = r \times c$  is allocated for a task at a base address TLpt =  $\langle x, y \rangle$  and BRpt =  $\langle x', y' \rangle$ , where  $x' = x+r-1 \leq R$ ,  $y' = y+c-1 \leq C$ . By using the k-tree non-bit-map approach, the number of nodes in the k-Tree is dynamic, which are corresponding to the number of tasks allocated. At the starting time, the k-Tree consists of only one node (called root), used to store the system information (i.e., status, size, address, and etc.). During executing time (many jobs or tasks allocated), each leave node of the k-Tree may be available/free (status = 0) or unavailable/busy (status = 1) and each internal node is partially available (status = x). For allocating an incoming task  $T(r, c)$ , each larger available node (a  $X$  b) at level  $k$  in the k-Tree can be dynamically created and partitioned into, at most, four sub-partitions (Figure 2.a) : 1) TL (top left), 2) BL (bottom left), 3) TR (top right), or 4) BR (bottom right) and assign one (of size  $r \times c$  or  $c \times r$ ) at level  $k+1$  for the requested task. An example of the k-Tree and system status that stores 3 incoming tasks (4x4, 2x3, and, 2x4 respectively) on a given 8x10 Mesh-connected system are depicted in Figure 2.b.

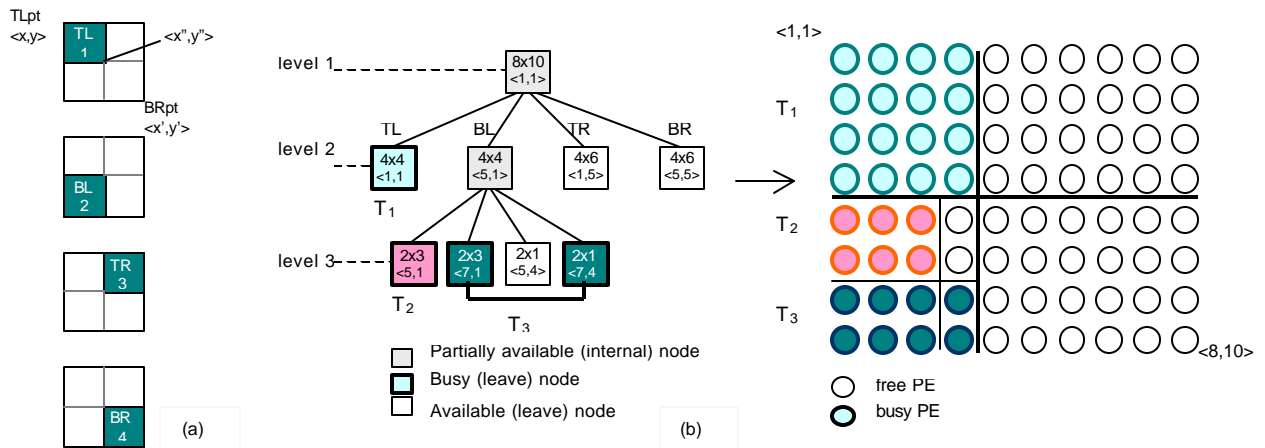
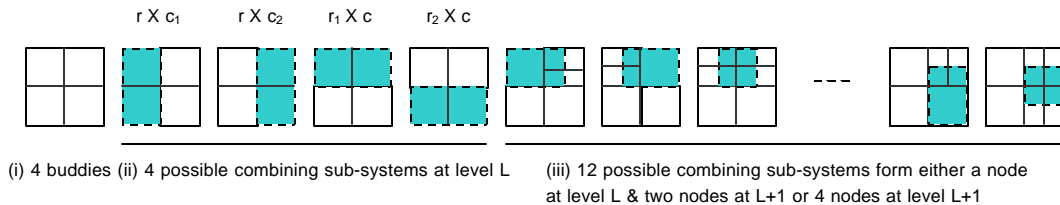


Figure 2: (a) 4-buddy partitioning and (b) The k-Tree of system states represents an 8x10 system status with 3 tasks allocated.

In combining for two-dimensional mesh or torus of size  $(r \times c)$  at level  $L-1$ , there are four buddies at level  $L$ , four possible sub-systems combined at level  $L$ , and 12 possible combined sub-systems (row, column, and middle) since guest network (mesh) size may not be necessary equal. The basic combining is



In this research study, system performances are evaluated by using an event-driven simulation, which events are incoming tasks (or jobs) and finishing tasks. The purpose is to evaluate the performance of the proposed allocation/deallocation strategy in terms of system utilization, system fragmentation, and average waiting time, which are measured at the steady-state operation of the system under various parameters setting during experiments (i.e., vary system sizes, task sizes, arrival rate, workload, etc.). The results of a number of experiments of applying the unified

model (k-Tree) to mesh-connected systems, compared with existing mesh-based processor allocation (i.e., free sub-system list (FSL), busy list (BL), quick allocation (QA), best fit bit-map (BF)), are described as follows: Experiment 1 is to investigate the effect of task sizes to the system utilization ( $U_{sys}$ ) and hence system fragmentation ( $1-U_{sys}$ ) under static processor allocation with a system size =  $30 \times 30$ . In most test cases of varying task sizes and fixing a system size (see Table 1), results of our k-tree-based approach are similar to the FSL and BL strategies.

Table 1: System utilization of static processor allocation of the k-tree and existing strategies.

Task sizes	k-Tree	FSL	BL	QA	BF
1x1- 5 x 5	92.5 %	88.9 %	97.8 %	91.4 %	89.9 %
1x1-10x10	88.8 %	81.6 %	89.6 %	78.7 %	77.6 %
1x1-15x15	78.1 %	74.2 %	79.2 %	69.7 %	67.9 %
1x1-20x20	66.3 %	65.1 %	67.6 %	60.1 %	57.1 %
1x1-25x25	59.0 %	59.4 %	58.3 %	57.0 %	49.4 %
1x1-30x30	58.5 %	58.4 %	58.0 %	57.5 %	51.9 %

Experiment 2 is to investigate the effect of task sizes to the system utilization ( $U_{sys}$ ) and hence system fragmentation ( $1-U_{sys}$ ) under dynamic processor allocation/deallocation with a system size =  $30 \times 30$ . Similar to the static allocation, in most dynamic test cases of varying task sizes (see Table 2), results of our k-tree-based approach are similar to the FSL and BL strategies.

Table 2: System utilization of dynamic processor allocation/deallocation of the k-tree and existing strategies.

Task sizes	k-Tree	FSL	BL	QA	BF
1x1- 5 x 5	79.4 ± 5.6 %	77.7 ± 5.3 %	80.4 ± 7.6 %	79.3 ± 6.6 %	79.0 ± 6.3 %
1x1-10x10	74.5 ± 3.6 %	71.7 ± 3.8 %	75.3 ± 4.7 %	71.3 ± 4.0 %	69.4 ± 3.8 %
1x1-15x15	70.5 ± 6.0 %	67.3 ± 5.7 %	70.9 ± 5.9 %	65.5 ± 5.2 %	62.0 ± 5.0 %
1x1-20x20	65.8 ± 8.6 %	64.1 ± 6.0 %	66.4 ± 7.7 %	62.0 ± 6.2 %	55.3 ± 7.8 %
1x1-25x25	65.5 ± 7.8 %	62.5 ± 7.2 %	64.3 ± 8.3 %	61.3 ± 7.1 %	54.9 ± 8.6 %
1x1-30x30	62.9 ± 8.2 %	62.3 ± 7.4 %	63.5 ± 8.6 %	60.5 ± 7.3 %	54.0 ± 8.3 %

Experiment 3 is to investigate the effect of system sizes to the average waiting time (under dynamic processor allocation/deallocation) with a workload = 0.5 and task sizes  $1 \times 1 - R \times C$ . In Table 3, our k-tree-based approach performs similar average waiting time to the FSL and BL in most test cases.

Table 3: Average waiting time of dynamic processor allocation/deallocation of the k-tree and existing strategies.

System sizes	k-Tree	FSL	BL	QA	BF
30 x 30	4.40	4.45	4.38	4.49	5.09
50 x 50	11.01	11.20	10.99	11.19	12.58
100x100	33.45	33.51	33.49	33.69	35.87
200x200	85.35	85.40	85.30	85.61	90.50

In summary, the time complexity and performance improvement are summarized as follows:

Table 4: Time complexity and system performance comparisons of our k-tree and existing strategies.

Year	Strategies	Time complexity			(space) Memory	Task rotating	Complete recognition	System performance
		Allocation/Deallocation/Scheduling						
1991	2DB (2 Dim Buddy)	$O(\log N)$	$O(N)$	$O(N_w \log N)$	$O(N)$	No	No	Not compare
1991	FS (Frame Slide)	$O(N)$	$O(1)$	$O(N_w \cdot N)$	$O(N)$	No	No	Improve > 2DB
1992	FF (First Fit bit-map)	$O(N)$	$O(N)$	$O(N_w \cdot N)$	$O(N)$	No	No	Improve > FS
1992	BF (Best Fit bit-map)	$O(N)$	$O(N)$	$O(N_w \cdot N)$	$O(N)$	No	No	Improve > FS
1993	AS (Adaptive scan)	$O(N_a \cdot N)$	$O(1)$	$O(N_w \cdot N_a \cdot N)$	$O(N_a)$	Yes	Yes	Improve > FS
1993.6	BL (Busy List)	$O(N_a^3)$	$O(1)$	$O(N_w \cdot N_a^3)$	$O(N_a)$	Yes	Yes	Improve > BF
1995	FL (Free List)	$O(N_r^2)$	$O(N_r^2)$	$O(N_w \cdot N_r^2)$	$O(N_r)$	Yes	Yes	Improve > BF
1996	FFwp (FF with partition)	$O(N/\log N)$	$O(N/\log N)$	$O(N_w \cdot N/\log N)$	$O(N)$	No	No	Improve $\equiv$ FF
1997	QA (Quick Allocation)	$O(N_a^3 \cdot N)$	$O(1)$	$O(N_w \cdot N_a^3 \cdot N)$	$O(N_a)$	Yes	Yes	Improve > AS
1998	FSL (Free Sub-List)	$O(N_r^2)$	$O(N_r^2)$	$O(N_w \cdot N_r^2)$	$O(N_r)$	Yes	Yes	Improve > FL
propose	best fit k-Tree	$O(N_a)$	$O(N)$	$O(N_w \cdot N_a)$	$O(N_a)$	Yes	Yes	Improve $\equiv$ FSL

Note:  $N_a$  = #allocated tasks,  $N_r$  = #free sub-systems,  $N_w$  = #tasks in the waiting queue, and  $N =$  system size ( $N_a + N_r \leq N$ ).

### 4.2 Application of the Unified Model to 3D-Meshes

When applying the unified model to 3-D Mesh multicomputers, our k-tree-based approach is very efficient in time complexity  $O(N_a)$  and system performance effect with similar trends to 2-D Meshes. In the application on a 3-D Mesh system of size  $N = R \times C \times H$ , let  $P(i, j, k)$  denotes PE at corner address  $\langle i, j, k \rangle$  ( $i^{\text{th}}$  row,  $j^{\text{th}}$  column,  $k^{\text{th}}$  height) such as  $P(1,1,1)$  PE at TL front (first/minimum address in the system) and  $P(R, C, H)$  PE at BR back (last/maximum address in the system). Figure 3 illustrates a 3-D mesh system of size  $8 \times 8 \times 8$  and its status and corresponding k-tree with 3 tasks ( $T_1 = 4 \times 4 \times 4$ ,  $T_2 = 2 \times 2 \times 2$ , and  $T_3 = 4 \times 4 \times 4$ ) allocated.

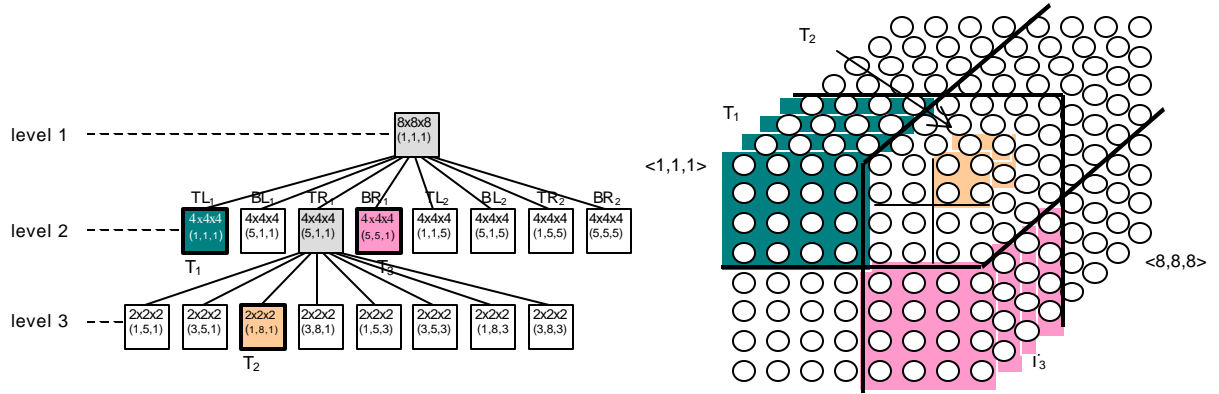
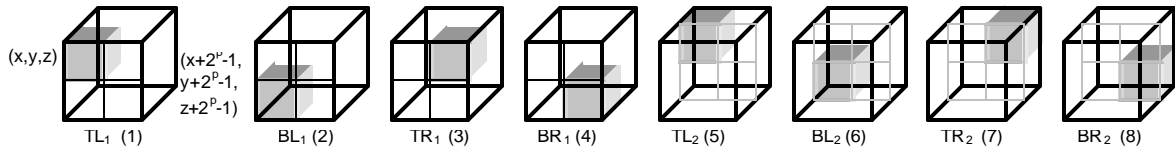
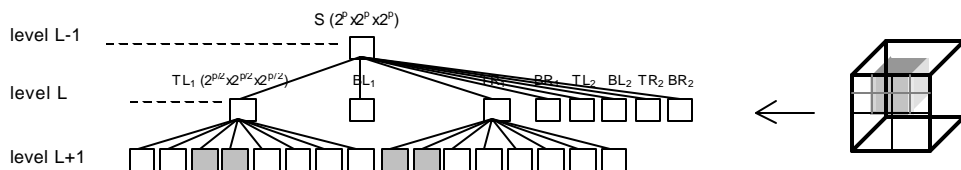


Figure 3: The k-tree system state representation for a 3D mesh (8x8x8) with 3 tasks allocated.

For simply explanation, assume given a system size  $N = 2^q \times 2^q \times 2^q$  and any task with all sizes =  $2^q$  ( $2^q \times 2^q \times 2^q$ ), where  $q = 0, 1, 2, \dots, n$ . In partitioning, a number of buddies/node =  $2^3 = 8$ , called  $TL_1, BL_1, TR_1, BR_1, TL_2, BL_2, TR_2,$  and  $BR_2$ , (or buddy-id 1, 2, ..., 8) respectively. For an incoming task ( $2^q \times 2^q \times 2^q$ ), suppose a free sub-system size is  $2^p \times 2^p \times 2^p$  (starting at a PE  $\langle x, y, z \rangle$ ) where  $p \geq q$ . If  $p = q$ , then this sub-system can be allocated to the request; otherwise it will be recursively partitioned each size into  $p/2, p/4, \dots$ , until equal to  $q$  and then allocated to the request. Note: for any size partitioning, a number of buddies/node =  $2^3 = 8$  (similar to above). For any task size ( $r \times c \times h$ ), suppose a free sub-system size is  $a \times b \times c$  (starting at a PE  $\langle x, y, z \rangle$ ) where  $p \geq q$ . If  $p = q$ , then this sub-system can be allocated to the request; otherwise it will be partitioned each size into  $r \times c \times h$  and then allocated to the request at the next level in the k-tree.



In combining under equal partitioning size, there are 18 more sub-systems of size  $(2^{p/2} \times 2^{p/2} \times 2^{p/2})$ , which are combined from free 8 nodes at level  $L+1$  and one of then is depicted as follows:



Under any task size partitioning, there are 8 buddies at level L, 4 possible combined sub-systems (with sizes > any buddy) plus 18 (or  $k \sum_{j=1}^{k-1} 2^{k-j}$ ,  $j=1, 2, \dots, k-1$ ) possible combined sub-systems at level L, and 36 possible combined sub-systems at level L and L+1 (or  $24 = k2^k$  for expanding each buddy at level L plus  $12 = k2^{k-1}$  for combining free nodes at level L+1 of two partially free buddies at level L).

In system performance evaluations, a number of experiments are done to investigate the effects of applying the unified k-tree model, as follows: Experiment 1 and 2 are to investigate the effect of task sizes ( $r \times c \times h$ ) to the system utilization ( $U_{sys}$ ) under static and dynamic processor allocation with a system size =  $32 \times 32 \times 32$ . The results in Table 5 show that the best fit k-tree yields better system utilization than the first fit k-tree in all test cases.

Table 5: System utilization of static processor allocation of the k-tree with first fit and best fit heuristics.

Task sizes:	Static		Dynamic	
	First Fit k-tree	Best Fit k-tree	First Fit k-tree	Best Fit k-tree
1x1x1- 8 x 8 x 8	75.23 %	85.89 %	54.96 %	74.16 %
1x1x1-16x16x16	60.45 %	77.55 %	36.47 %	59.89 %
1x1x1-32x32x32	35.26 %	50.54 %	10.62 %	29.20 %

Experiment 3 is to investigate the effect of system sizes to the system utilization (under dynamic processor allocation/deallocation) with a workload = 0.5 and task sizes  $1 \times 1 \times 1 - R_{1/2} \times C_{1/2} \times H_{1/2}$ . In Table 6, results show that the best fit k-tree yields better system utilization than the first fit k-tree in all test cases, which prove up to 50%.

Table 6: System utilization of dynamic processor allocation/deallocation of the k-tree and existing strategies.

System sizes:	16 x 16 x 16	32 x 32 x 32	64 x 64 x 64	128 x 128 x 128
First Fit k-tree	45.60 %	36.47 %	18.46 %	15.79 %
Best Fit k-tree	75.68 %	59.89 %	48.16 %	37.53 %

## 5 Conclusions and Future Studies

We propose the unified model for sub-system allocation on product networks (such as k-D Mesh/Torus, Hypercube, and generalize Hypercube) and its efficient application on 2-D and 3-D Mesh-connected multicomputers. For 2-D Meshes, our approach “best fit k-tree-based sub-system allocation/deallocation and priority task scheduling” is efficient in time complexity (which is  $O(N_a)$ ,  $N_a$  is the number of allocated tasks) and also system performance, compared to existing mesh-based strategies. Using simulation studies, a number of experiments are done to investigate and evaluate the system performance effects, compared with the existing strategies. Our approach performs similar system utilization, system fragmentation, and average waiting time to the FSL and BL strategies and better than the QA and BF strategies. For 3-D Meshes, the k-tree-based approach time complexity is also  $O(N_a)$  and the best fit k-tree approach yields better system performances than the first fit k-tree for both static and dynamic allocation up to 50%. In our future studies, we will apply our unified model to other useful interconnection networks such as 2-D and 3-D toruses and hypercube multicomputers.

## References

- [1] R. Alverson et al., “The Tera computer System,” Proc. 1990 Int’l Conf. Supercomputing, pp.1-6, 1990.
- [2] P.J. Chuang and N.F. Tzeng, “An Efficient Submesh Allocation Strategy for Mesh Computer Systems,” Proc. Int’l Conf. on Distributed Computing Systems, pp. 256-263, May 1991.
- [3] P.J. Chuang and N.F. Tzeng, “Allocating Precise Submesh in Mesh-Connected Systems,” IEEE Trans. on Parallel and Distributed Systems, v.5(2), pp. 211-217, 1994.

- [4] D. Das Sharma and D.K. Pradhan, "A Fast and Efficient Strategy for Submesh Allocation in Mesh-Connected Parallel Computers," IEEE Symp. Parallel and Distributed Processing, pp. 682-689, 1993.
- [5] D. Das Sharma and D.K. Pradhan, "Submesh Allocation in Mesh Multicomputers Using Busy-List: A Best-Fit Approach with Complete Recognition Capability," J. of Parallel and Distributed Computing, v36, pp106-118, 1996.
- [6] D. Das Sharma and D. K. Pradhan, "Job Scheduling in Mesh Multicomputers," IEEE Trans. in Parallel and Distributed Systems, v. 9(1), pp. 57-70, 1998.
- [7] N. J. Dimopoulos, and et al, "Routing and processor allocation on a hypercycle-based multiprocessor," in Proc. 1991 Int. Conf. Supercomputing., ACM, pp. 106-114, June 1991.
- [8] N. J. Dimopoulos and V. V. Dimakopoulos, "Optimal and suboptimal Processor Allocation for Hypercycle-based Multiprocessors," IEEE Trans. on Parallel and Distributed Systems, v.6(2), pp. 175-184, 1995.
- [9] J. Ding and L.N. Bhuyan, "An Adaptive Submesh Allocation Strategy for Two-Dimensional Mesh Connected Systems," Proc. 1993 Int'l Conf. on Parallel Processing, vol. II, pp. 193-200, 1993.
- [10]T. El-Ghazawi and A. Youssef, "A General Framework for Developing Adaptive Fault-Tolerant Routing Algorithms," IEEE Trans. on Reliability, v. 42(2), pp. 250-258, 1993.
- [11]Intel, "A Touchstone DELTA System Description," Supercomputer Systems Division, Intel Corporation, Beaverton, OR 97006, 1991.
- [12]Intel, "Paragon XP/S Product Overview," Supercomputer Systems Division, Intel Corporation, Beaverton, OR 97006, 1991.
- [13]G. Kim and H. Yoon, "On Submesh Allocation for Mesh Multicomputers: A Best-Fit Allocation and a Virtual Submesh Allocation for Faulty Meshes," IEEE Transactions on Parallel and Distributed Systems, v.9(2), 1998.
- [14]K. Li and K.H. Cheng, "Job Scheduling in a Partitionable Mesh Using a Two-Dimensional Buddy System Partitioning Scheme," IEEE Trans. on Parallel and Distributed Systems, v.2(4), pp.413-422, 1991.
- [15]K. Li and K.H. Cheng., "A Two-Dimensional Buddy system for Dynamic Resource Allocation in a Partitionable Mesh-connected System," Journal of Parallel and Distributed Computing, v.12, pp. 79-83, 1991.
- [16]T. Liu, and et. al., "A Submesh Allocation Scheme for Mesh-Connected Multiprocessor Systems," Proc. 1995 Int'l Conf. Parallel Processing, vol.II, pp. 159-163, 1995.
- [17]Mattson et al., "Intel/Sandia ASCI system," Proc. of the 10<sup>th</sup> Int'l Parallel Processing Symp., 1996.
- [18]P. Mohapatra, "Processor Allocation Using Partitioning in Mesh Connected Parallel Computers," J. of Parallel and Distributed Computing, v.39, pp. 181-190, 1996.
- [19]C.L. Seitz, "Mosaic C: An Experimental Fine-Grain Multicomputer," Technical report, California Institute of Technology, Pasadena, C91125, 1992.
- [20]J. Srisawat and N. A. Alexandridis, "Efficient Processor Allocation Scheme with Task Embedding for Partitionable Mesh Architectures," The 11<sup>th</sup> Intl. conference on Computer Applications in Industry and Engineering 98, pp. 309-312, Las Vegas, November 11-13, 1998.
- [21]J. Srisawat and N. A. Alexandridis, "Reducing System Fragmentation in Dynamically Partitionable Mesh-Connected Architectures" , Int'l Conf. on Parallel and Distributed Computing and Networks, Australia, December 14-16, 1998.
- [22]S. Yoo, and et. al., "An Efficient Task Allocation Scheme for 2D Mesh Architectures," IEEE Trans. on Parallel and Distributed systems, v.8(9), pp. 934-942, 1997.
- [23]A. Youssef, "Product Networks: A Unified Theory of Fixed Interconnection Networks," Technical report: GWU-IIST-90-38, the George Washington University, December, 1990.
- [24]Y. Zhu, "Efficient Processor Allocation Strategies for Mesh-Connected Parallel Computers," J. of Parallel and Distributed Computing, v.16, pp. 328-337, 1992.