ACM-SIGARCH Int'l Conf.
on Supercomputing (ICS 99)
June 20-25, 1999, Rhodes,
Greece

A New "Quad-Tree-Based" Sub-System Allocation Technique for Mesh-connected Parallel Machines

Jeeraporn Srisawat and Nikitas A. Alexandridis
Department of Electrical Engineering and Computer Science
The George Washington University
Washington DC 20052, U.S.A.
{jeera, alexan}@seas.gwu.edu

ABSTRACT

Massively mesh-connected parallel machines are an efficient and scalable class of computer systems used to execute supercomputing applications. In such systems, a number of independent smaller tasks (from the same or different applications) come in, each requiring at run time a separate subsystem (or partition) to execute. There is a need for the operating system to dynamically partition the computer system to allocate resources to incoming tasks, as well as to deallocate resources (and recombine partitions) as soon as they become available when a task completes. A number of processor allocation and deallocation techniques have been proposed in the past, with various degrees of time complexity and system performance.

In this paper, we present a new best-fit "quad-tree-based" sub-system allocation/deallocation technique, which has smaller time complexity than existing techniques, while yielding better system performance. We prove analytically that the time complexity of our approach is only $O(N_a)$ (where N_a is the number of allocated tasks, N the system size, and $N_a < N$) which is better than the best of the existing mesh-based strategies (such as the $O(N_a\sqrt{N})$ of "Quick Allocation", the $O(N_a^3)$ of "Busy List", and the $O(N_f^2)$ of "Free Sub-List", where N_f is the number of free sub-systems and $N_f < N$). Our simulation studies show that our "quad-tree based" allocation/deallocation technique also results in a 33% improvement in system performance (in terms of system utilization and average task completion time).

Keywords
Quad-Trees, parallel systems, system partitioning, resource allocation/deallocation, mesh-connected parallel machines.

1. INTRODUCTION

Massively mesh-connected configurations represent an efficient and scalable approach in constructing high-performance parallel computers. Examples of prototypes and commercial mesh-connected systems include the Intel/Sandia ASCI System [13], the Intel Touchstone system [7], the Intel Paragon XP/S [8], etc., which support a multi-user environment (to execute various independent applications in parallel.) These parallel systems are used for executing supercomputing applications, which usually require large sub-system sizes. In such environments, a sequence of incoming tasks (or applications) are required to be allocated at run time to a separate sub-system by the host's operating system (its processor allocator and task scheduler). A number of efficient processor allocation algorithms have been developed in the past, in order to allocate incoming tasks in efficient decision time, while at the same time maintaining the best (possible) performance for subsequent tasks (arriving at run time). For massively mesh-connected machines, many processor allocation strategies have been introduced and all of them are classified into four main classes: the "bit-map" approaches [14, 18], the "non-bit-map free list" approaches [9, 10-12], the "non-bit-map busy list" approaches [2-6, 17], and our "non-bit-map quad-tree-based" approach [15, 16]. Chronologically, some earlier mesh-based processor allocations used the **first fit** decision criterion. Examples include the 2-D Buddy [10, 11], the Frame Slide [2, 3], the First Fit bit-map [18], the Adaptive Scan [6], the First Fit with partition [14], and the first fit Q-Tree [15]. More recently, however, researchers are proposing the use of the more complicated **best fit** criterion, which yields improved system performance. Examples include the Quick Allocation [17], the Busy List [4, 5], the Free List [12], the Free Sub-List [9], and the best fit Q-Tree [16]. Usually, these higher system performance algorithms cost more in time complexity. In our previous work, we showed that our "first fit Q-Tree" [15] yields better time complexity and system performance than other first fit strategies; similarly, our "best fit Q-Tree" [16] yields better time complexity and system performance than the Best Fit bit-map and Free List techniques.

In this paper, we present a new and more efficient **best-fit** "Quad-Tree-based" approach for sub-system allocation and deallocation on massively mesh-connected multicomputers. In particular, we have modified the best-fit criterion as follows: first, it uses as the major criterion the maintaining of maximum free sub-systems (as does the Free Sub-List technique [9]); second, we have introduced two additional minor criteria (the "minimum different sizes factor" and the "minimum probability of combining"). The result of our approach is smaller time complexity and a higher system performance than other mesh-based strategies. We prove the improvement in time complexity by analytical methods, while the performance (in terms of system utilization, system fragmentation, and average task completion time) of our proposed Q-Tree-based approach is compared with other mesh-based techniques using simulation studies in which we vary several system parameters (such as system size, workloads of arriving tasks, task size, etc.).

ICS 99 Rhodes Greece
Copyright ACM 1999

The next section describes previous studies in processor allocation for mesh-connected multicomputers. Section 3 presents our modified criteria for the efficient best-fit and the new Quad-Tree-based sub-system allocation algorithm. Then in Section 4, the Quad-Tree-based approach is evaluated and compared to other techniques in terms of time complexity and system performance. Finally, conclusions are discussed in Section 5.

2. RELATED PROCESSOR ALLOCATION TECHNIQUES

Consider an initial partitionable mesh-connected system S(R,C) of size N = R x C (where R is the number of rows and C the number of columns). P(i, j) will be one of its processing elements (PEs) at coordinate address (i,j) (or i^{th} row and j^{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 (or bottom rightmost position) of the system. Any partitionable mesh-connected system will be identified by using two numbers: its **base address** or location (the Top Left point or TLpt) and its **last address** which is the Bottom Right point (BRpt) = (R, C) (computed using the base address and the system size). For the above initial partitionable system, (TLpt) = (1,1) and (BRpt) = (R, C) and, thus, the system is denoted as <(1,1), (R,C)>. A sub-system S(r, c) of size p = r X c will be allocated to an incoming task by specifying its two numbers: its base address TLpt = (x, y) and its last address BRpt = (x', y'), where x' = x+r-1 $\leq$ R and y'= y+c-1 $\leq$ C. Figure 1 shows an example of an initial 8 x 8 mesh system <(1,1), (8,8)> and a (4x4) task that has been allocated into its subsystem <(1,1), (4,4)>.

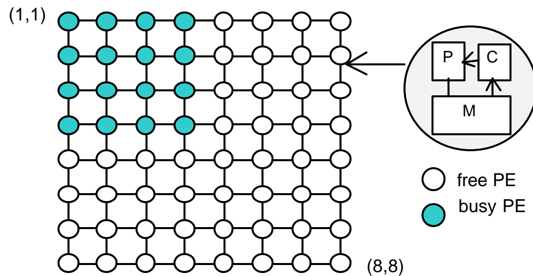


Figure 1: An example of a mesh-connected system (N = 8x8): the system status with an allocated task (4x4).

The rest of Section 2 presents short background material and definitions on a number of mesh-based processor allocation techniques by which the recently best-fit methods (i.e., the Busy List [5], the Quick Allocation [17], and the Free Sub-List [9]) will be compared to our technique proposed in this paper.

2.1 The Two-Dimensional Buddy Strategy

The first fit 2DB (2-Dimensional Buddy) strategy [10, 11], proposed in 1991, was used in an early study in processor allocation for a partitionable mesh parallel system (N = 2^n). In this strategy, an array of linked lists was used to store all available square sub-systems (2^p x 2^p, p = 1, 2, …, n/2). Therefore, searching to find a free sub-system for a request was done in O(log N) time. However, "sub-system combining" (executed in order to accommodate deallocation of system resources when a task completes) was done in O(N) time. Because this method was applied efficiently only for a square (power of 2) system and square-sized requested sub-systems , it caused high internal system fragmentation, especially for non-square requests.

2.2 The Frame Slide Strategy

In 1991, the first fit FS (Frame Slice) processor allocation [2, 3] was also proposed as a solution to the problem of internal fragmentation of the 2DB strategy. This method allowed any mesh system size (N = R x C) and any requesting task size (r X c). A linked list was used to store only the allocated (N_a) sub-systems. A candidate frame (starting at the left-most free PE) was created and compared with allocated frames in the list. If there was no intersection, then that frame was available; otherwise, that frame was slided horizontally by r rows (or vertically by c columns), to find a new candidate frame in O(N_a.N/(rXc)) time. The disadvantage however, was that sliding the frame by a factor of r (or c) might skip over some available sub-systems.

2.3 First Fit and Best Fit Bit Mappings

In 1992, the FF and BF bit-maps (First Fit and Best Fit bit-maps) [18] were introduced in order to solve the allocation miss problem in the FS strategy. In such bit-map approaches, a 2D-array system status (N = R x C) is used to store a free/busy status-bit for every processor. For an incoming request (r X c), all N bits have to be identified at least twice to find the corresponding available sub-system; therefore, the time complexity of these bit-map methods is O(N). The bit-map strategies gave better system utilization than the FS strategy. However, these strategies did not have complete recognition capability since they did not provide task rotation.

2.4 The Adaptive Scan Strategy

In 1993, the first fit AS (Adaptive Scan) strategy [6], the first strategy with complete recognition capability (i.e., including task rotation), was presented as another solution to the allocation miss problem of the FS strategy. It uses a busy list to store all (N_a) allocated tasks and task rotation to allocate either S(r, c) or S(c, r) for a requested task (r X c)). Time complexity for the AS to find the first free sub-system S(r, c) (or S(c, r)) for a task is O(N_a.N) and its performance results improved over those of the FS strategy.

2.5 The First Fit with Partition Strategy

In 1996, a combining approach (such as FSwp, FFwp, etc.) [14] was proposed to combine existing strategies (such as FS, FF, 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). This reduced the time complexity over existing strategies by a factor of log N. System fragmentation of the FFwp strategy was almost identical to that of the FF bit-map strategy.

2.6 The Busy List Strategy

The best fit BL (Busy List) strategy was proposed in 1993 [4] and improved in 1996 [5]. This strategy improved time complexity as well as system fragmentation over the BF bit-map. It uses "a busy list" to store allocated sub-systems and proposes the "maximum boundary value (BV)" as the best-fit criteria. For an incoming request task (r X c), all (up to 8) candidate sub-systems of size S(r, c) or S(c, r) are created from each of the 4 corners of a particular allocated sub-system. Then, the candidate sub-mesh with the maximum BV is stored. After all N_a allocated sub-systems are identified, the candidate sub-mesh with maximum BV is selected. This BL allocation process takes O(N_a^3) time.

2.7 The Free List Strategy

The best fit FL (Free List) strategy [12] was proposed in 1995 in order to improve time complexity and system fragmentation over the BF bit-map. The FL strategy used 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 (rXc), the first sub-system in the list(r) is considered. If it is larger than the request, then 2-8 candidate sub-systems of size S(r, c) or S(c, r) are created and the one with the maximum boundary value (similar definition of the term as in the BL strategy) is selected. The time complexity of the FL is $O(N_f^2)$ for both allocation and deallocation, where N_f is the number of free sub-systems ($N_f \leq N$).

2.8 The Quick Allocation Strategy

In 1997, the best fit QA (Quick Allocation) strategy [17] was proposed in order to improve time complexity. 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 list, and (3) reject areas. For an incoming task rXc, the searching process was to find an available sub-system (it began by computing the coverage sub-system list and the reject areas). Then, all coverage sub-systems were sorted in non-decreasing order. For each row (starting from 1 to R of N = R x C), find a free sub-system (that did not intersect with the coverage sub-systems and the rejected areas) and allocate it to the request. The time complexity of the QA allocation is $O(N_a\sqrt{N})$, and the performance improved over the AS strategy.

2.9 The Free Sub-List Strategy

In early 1998, the best fit FSL (Free Sub-List) strategy [9], was proposed to improve the average waiting time performance by trying to minimize the amount of potential system fragmentation and preserve as many large free sub-systems as possible for subsequent tasks. A list of free (N_f) sub-systems was used to store only free sub-systems (sorted in decreasing order of size values). Among all free sub-systems (N_f x (up to 8) sub-systems were computed from 4 corners of each free sub-system in the list), the one that yielded the minimum degree of fragmentation was to be selected. The time complexity of this FSL processor allocation strategy is $O(N_f^2)$ for an incoming task and the deallocation time complexity is $O(N_f^3)$ for computing all free sub-systems. Simulation results showed that the FSL strategy improves system performance over the FL and BL strategies.

3. THE QUAD-TREE-BASED APPROACH

3.1 System States Representation

In our approaches we use a "Q-Tree" data structure in order to represent system states (or store allocation information) of a given mesh-connected system. By using the Q-Tree (non-bit-map) approach, the number of nodes in the Q-Tree becomes dynamic, corresponding to the number of tasks allocated. At the start, the Q-Tree consists of only one node (called the root), used to store the system information (i.e., status, size, address, etc.). During execution time (when many jobs or tasks are allocated), each leaf node of the Q-Tree may be available/free (status = 0) or unavailable/ busy (status = 1) and each internal node is partially available (status = x). In order to allocate an incoming task (r X c), each larger available node (a X b) at level k in the Q-Tree can be dynamically created and partitioned into, at most, four sub-partitions (or Buddies): 1) TL (top left), 2) BL (bottom left), 3) TR (top right), or 4) BR (bottom right) and assigned one (of size r X c or c X r) at level k+1 to the request. Figure 2 illustrates an example of the Q-Tree and the status of a system that stores 3 tasks (4x4, 2x3, and, 2x4) on a given 8x10 system.

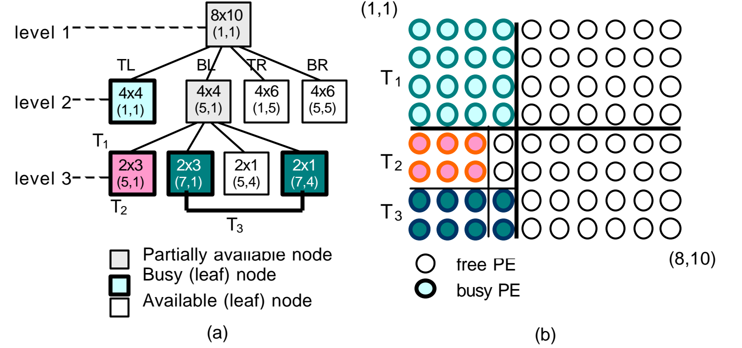
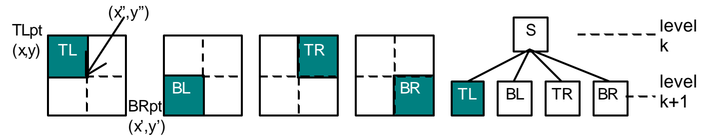


Figure 2: (a) A Q-Tree system state representation and (b) the corresponding 8x10 mesh system with 3 tasks allocated.

3.2 Sub-System Partitioning Procedure

DEFINITION 1: "4-Buddy partitioning" is applied to partition a larger free sub-system S(a, b) (at <TLpt, BRpt> = <(x,y), (x',y')>) for a task (r X c), where r ≤ a and c ≤ b. The S(a, b) at level k of the Q-Tree will be partitioned into 4 sub-partitions (called TL, BL, TR, and BR) such that there exists a sub-partition size = r X c at level k+1. These 4 Buddy nodes are not necessary equal.



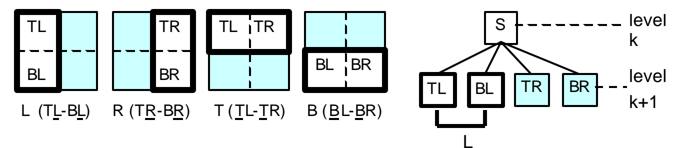
For a request task (r X c), suppose an allocated sub-partition is TL (size = r X c), then the address of TL and its corresponding 3 buddies with respect to the partition point = <x",y">, where x < x" < x' and y < y" < y', are TL sub-partition (size = r X c) at address <(x, y),(x", y")>, BL sub-partition (size = (a - r) X c) at address <(x"+1,y), (x', y")>, TR sub-partition (size = r X (b - c)) at address <(x, y"+1), (x", y')>, and BR sub-partition (size = (a - r) X (b - c)) at address <(x"+1, y"+1), (x', y')>, respectively.

3.3 Sub-System Combining Procedure

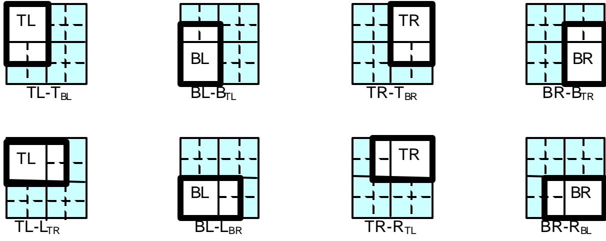
In the Q-Tree-based approach, all nodes in the Q-Tree at the same level k are stored as non-overlapping sub-systems. However, larger free sub-systems (at a lower level < k) may be overlapped with its corresponding nodes at a level ≥ k. Using the sub-system combining procedure, which is formally defined in Definition 2, can recognize these larger free sub-systems.

DEFINITION 2: "Sub-system Combining" is used to combine free buddies (at level k+1) or sub-buddies (at level k+2) of any internal node (at level k) in the Q-Tree, as in the following cases:

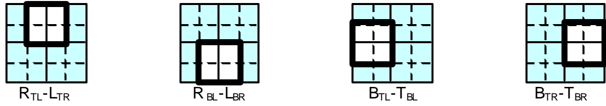
Case 1: "Buddy Combining": for any internal node (at level k), there are 4 possible combinations to combine larger free sub-systems from two to four free Buddies (at level k+1), as follows:



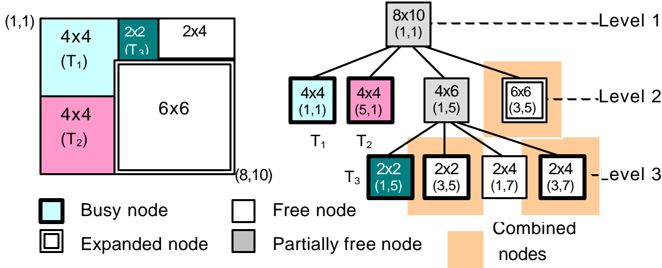
Case 2: "Buddy-SubBuddy Combining": for any internal node (at level k), there are 8 possible combinations to combine larger free sub-systems from a free buddy (at level k+1) and two free sub-buddy nodes (at level k+2) or expanding the row and expanding the column for each free buddy (TL, BL, TR, or BR), as follows:



(at level k), there are 4 possible combinations to combine more free sub-systems from 4 sub-buddies (at level k+2), as follows:



Note: for a combined sub-system, if its size is larger than the request, the "size expanding" process will be applied before the partitioning process. In this case, the old size and the old base-address of the expanded node are stored since they will be resumed later after this node is free. For instance, given a 8x10 system with 3 tasks allocated (4x4, 4x4 at level 2, 2x2 at level 3). Suppose there is a new incoming task of 5x6 (which is larger than each buddy node but less than a combined sub-system of 6x6). The expanded node of a combined 6x6 sub-system (at <(3,5), (8,10)>) is stored in the BR node at level 2 (with old size = 4x6 and base-address = (5,5)) before partitioning. The system status and the corresponding Q-tree (with size-expanding) are illustrated in the following figures:

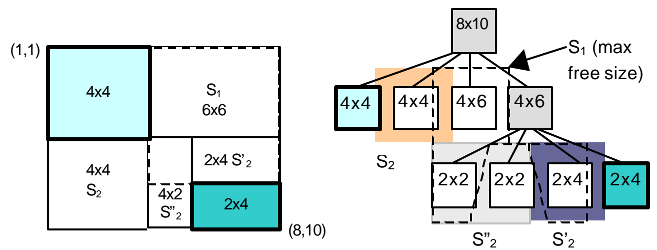


3.4 Best-Fit Criteria

The best-fit decision criteria is an improved performance policy as it tries to find the best sub-system that will tend to cause minimum future system fragmentation. In the Q-Tree-based approach, the best fit criteria is to find the best sub-system for an incoming task that can maintain the maximum free sub-system, give the minimum different size factor as well as the maximum free size after partitioning, be the smallest size, and yield the minimum combining factor. The priorities of these conditions are:

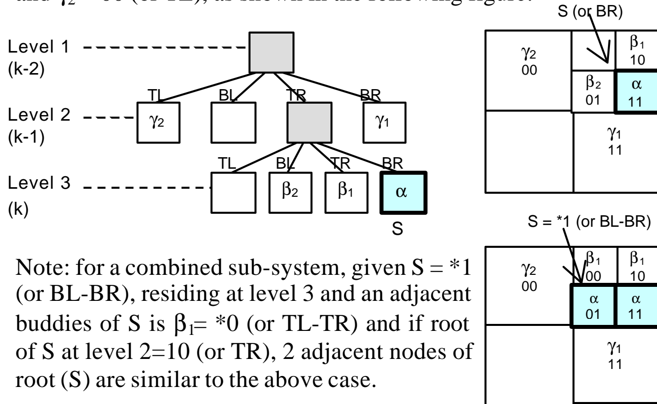
1) Find all free sub-systems (whose size ≥ the request) that can maintain the maximum free size as much as possible (i.e., disjoint first among a set of {disjoint, intersect, sub-set} overlap statuses (see Definition 3)).
2) If there are many candidates that have the same property in (1), then the candidate that gives the minimum different size factor and maximum free size after partitioning with the inclusion of task rotation (see Definition 4) is selected (i.e., diff SF = 0 if both sides of the request and the free sub-system are equal).
3) If there are many candidates that have the same property in (1) and (2), then the smallest candidate that yields the minimum combining factor (see Definition 5) is selected (or the minimum probability of combining with the closest adjacent node first). Otherwise, the first candidate or any one by random is selected.
4) After searching all nodes in the Q-Tree, if the best free sub-system is equal to the request, then it is allocated to the request; otherwise (if it is larger than the request), then it is partitioned and one of its buddies which yields properties similar to that given in steps 1 to 3 plus being the best buddy or sub-partition (see Definition 6) will be selected as the best sub-system and allocated to the request.

DEFINITION 3: "Overlap status": let S_1 be the maximum free sub-system in the Q-Tree (S_1's size $n_1 = n_{11}$ x n_{12} at <(a_{11}, a_{12}), (b_{11}, b_{12})>), computed once by using the combining procedure (see Section 3.3) in O(m) time, where m is the number of nodes in the Q-Tree. S_2 is any free node or combined sub-system in the Q-Tree (S_2's size $n_2 = n_{21}$ x n_{22} at <(a_{21}, a_{22}), (b_{21}, b_{22})>). S_2 is a subset of S_1 ($S_2 \subseteq S_1$) iff $\forall$i, i=1, 2 if ($a_{2i} \geq a_{1i}$ and $b_{2i} \leq b_{1i}$). S_1 and S_2 are disjoint iff $\exists$ i, i=1, 2 if ($a_{1i} > a_{2i}$ and $a_{1i} - a_{2i} \geq n_{2i}$) or if ($a_{2i} > a_{1i}$ and $a_{2i} - a_{1i} \geq n_{1i}$). Otherwise S_1 intersects S_2. Each of these statuses can be computed in O(1) time. For instance, the following figure illustrate various situations of overlap between S_1 "the maximum free size" (n_1= 6 x 6 at <(1,5), (6,10)>) and other free sub-systems S_2 (n_2 = 4 x 4 at <(5,1), (8,4)>), S'_2 (n'_2 = 2 x 4 at <(5,7), (6,10)>), and S''_2 (n''_2 = 4 x 2 at <(5,5), (8,6)>). Therefore, S_1 and S_2 are disjoint since for i=2 there exists (a_{12}-a_{22}) = (5-1 = 4) ≥ n_{22} (=4). S'_2 is subset of S_1 since S'_2 = <(5,7), (6,10)> ⊂ S_1 = <(1,5), (6,10)> (or $\forall$ i=1,2, $a_{2i} \geq a_{1i}$ and $b_{2i} \leq b_{1i}$). S''_2 intersects to S_1 since they are not disjoint and also neither is a subset of the other.



DEFINITION 4: "Task rotation": a process to find the suitable rotation of a task size (i.e., r X c or c X r) if it is allocated on a given free sub-system (i.e., the rotated size that yields minimum different size factor (diffSF) and maximum free size (maxFS) after partitioning is the best rotation of a task size (r X c)). For instance, given a free sub-system S of size 10x20 (or S(10,20)) for being allocated to an incoming task 5x10. In this case there are two possible rotations of the (5x10) task (they are 5x10 or 10x5) which can be allocated to the request. The second rotated size (10x5) is better than the first one because it yields a minimum different size factor (diffSF = 1) and hence maximum free size (maxFS=150) after partitioning and allocating, whereas the other size (5x10) yields diffSF = 2 and maxFS = 100.

DEFINITION 5: "Combining factor (CF)" of any sub-system S (assume at level k) is computed from its adjacent neighbor nodes as a summation of the probability of combining (PC) of combinable nodes of the same root as S. Let PC for each side or dimension of a sub-system S be defined as PC = 0 if that particular combined side is one of 4 system boundaries (since that side cannot be combined), PC = ¼ if its adjacent node of that particular side is busy (since that side can be combined after it becomes free); PC = ½ if its adjacent node is partially available, or PC = 1 if its adjacent node is free (or that side can be immediately combined). Let α (or S) be a considering node (at level k), β_1 and β_2 be buddy nodes of α with the same root R (at level k-1), and γ_1 and γ_2 be adjacent nodes of α with the same root R'(at level k-2). Therefore, CF(α) is the combining factor of α is the summation of $CF_1(\alpha,\beta) = PC(\alpha,\beta_1) + PC(\alpha,\beta_2)$ and $CF_2(\alpha,\gamma) = PC(\alpha,\gamma_1) + PC(\alpha,\gamma_2)$. As we can identify the β_1 and β_2 (at level k-1) and the γ_1 and γ_2 (at level k-2), then their PCs are computed, each of which is equal to either 0, ¼, ½, or 1 (if it is no combining, busy, partially free, or free). Therefore, the CF can be computed in O(1) time. For instance (finding combinable buddies of α), given an 8x8-mesh. Let $\alpha = b_1b_0 = 11$ (or BR), residing at level 3 and 2 adjacent buddies of α are $\beta_1 = 10$ (or TR) and $\beta_2 = 01$ (or BL). Assume root of α at level 2 = 10 (or TR), 2 adjacent nodes of root (α) are $\gamma_1 = 11$ (or BR) and $\gamma_2 = 00$ (or TL), as shown in the following figure:



Note: for a combined sub-system, given S = *1 (or BL-BR), residing at level 3 and an adjacent buddies of S is β_1= *0 (or TL-TR) and if root of S at level 2=10 (or TR), 2 adjacent nodes of root (S) are similar to the above case.

DEFINITION 6: "Best Buddy (or sub-partition) after partitioning" (for step 4 in the best fit criteria): assume the best free sub-system from step 1-3 is α (or S), whose size is larger than the request. It will then be partitioned and the best sub-partition (or one of 4 buddies (TL, BL, TR, BR)) is allocated to the request. Let $\beta_i = (\beta_{i1}, \beta_{i2})$ be a set of combinable buddies of any sub-partition α_i. Since the combining factor (Definition 5) CF (α,β) of α (at level k-1 and k-2) is the same for all α_i, i = 1, 2, 3, 4, the modified combining factor MCF (α_i,β_i) for each α_i is computed as CF (α_i,β_{ij}) of each α_i (at level k and k-2) and the buddy yielding the min MCF is selected as the best buddy (or sub-partition) after partitioning.

3.5 Quad-Tree-Based Sub-System Allocation/ Deallocation Algorithm

In dynamic Q-Tree-based sub-system allocation/deallocation, when there is an incoming task (or job), if the wait priority of the first task in the waiting queue is more than the threshold value, then the incoming task will be put in the waiting queue; otherwise, the processor "allocation" procedure will find for the request the best available sub-system by using the best bit criteria (defined in Section 3.4) and DFS (depth first search) to visit all nodes in the Q-Tree. If there exists an available sub-system, then the request will be allocated to the system. Otherwise (no free sub-system), the request will be put in the waiting queue. Note: the threshold value will be set to avoid starvation of large tasks in the waiting queue.

```
QTreeBestFitAllocation (QT, T (r, c))
{
  maxFS = InitizedMaxFreeSize();     /* i.e., masFS.r = 0, maxFS.c=0      */
   maxFS = DFSforMaxFreeSubSystem(QT->root, maxFS);
  bestS = InitilizedBestValue();      /* i.e., diffSF=2, freeS=-1,CF=5,etc.*/
   bestS = DFSforBestSub-system(QT->root, T, maxFS, bestS);
  if (bestS = InitilizedBestValue())   /*  No free sub-system → put that  */
        EnQueue(T, wait-Queue);      /*      task in the waiting queue    */
  else {   /* (step 4 in Section 3.4) operation for the best sub-system       */
     if (size(bestS) > size(T))          /* Partitioning & find best buddy   */
         bestSP = BestBuddyPartitioning(bestS); /*(see Definition 1&6)*/
     AllocateAndUpdateBusyQTree(QT, bestSP);
    /* Sub-system Combining (see Definition 2 )  and Size-Expanding       */
     ListCombSS = CombiningFreeSubSystems (bestS->Prev, NULL);
     ListCombSS = CombiningFreeSubSystems(bestS, ListCombSS);
     maxFreeSS  = InitizedMaxFreeSize();
     ComSS = ListCombinedS->Head;
     while (ComSS != NULL) {            /* There are combined sub-systems*/
         newFreeSS = ComputeDizeValue(ComSS, maxFreeSS);
         if (newFreeSS > maxFreeSS)   maxFreeSS=
             UpdateMsxFreeSubSystem(newFreeSS, maxFreeSS);
         ComSS = ComSS->Next;
     }   /* upto 16 iterations (see Definition 2 for #combined sub-systems)  */
     if (maxFreeSS != NULL)  ExpandNode(maxFreeSS) ;
  }
}
```

```
DFSforBestSub-system (QTnode, T, maxFS, bestS)
{
  if (IsLeaf(QTnode) & Status(QTnode)=Free) { /*(step1-3 in Section 3.4)*/
     if (BestRotate(QTnode, T) TR = Rotate(T);  /*(see Definition 4)        */
     if (size(QTnode) >= size(TR)) {              /*(see Definition3 &5)     */
         newBestValue = ComputeBestFitValue(QTnode, maxFS);
         if (newBestValue > bestS)
             bestS = UpdateBestSubSystem(newBestValue,bestS);
     }
  }
  else {      /* DFS on internal (partially free)  node */
     bestS = DFSforBestSub-system(QTnode->TL,T,maxFS,bestS);
     bestS = DFSforBestSub-system(QTnode->BL,T,maxFS,bestS);
     bestS = DFSforBestSub-system(QTnode->TR,T,maxFS,bestS);
     bestS = DFSforBestSub-system(QTnode->BR,T,maxFS,bestS);
  }
  return bestS;
}
```

In this paper, the proposed efficient best-fit criteria (described in Section 3.4) is applied to find the best sub-system (rXc or cXr) for an incoming task (rXc), by searching for the best sub-system (a node or combined node) that can maintain the maximum free sub-system, yield the minimum different size factor and maximum free size after partitioning, is the smallest node, and yield the minimum combining factor. At step 4 (in the best-fit criteria), for the best node that is partitioned, after finishing the allocation, the max. free size after partitioning of that corresponding node is computed by using the "sub-system combining" procedure (Definition 2) and the "size-expanding" procedure (applied to the max. free size) for subsequent task(s). This can improve system fragmentation. For example, after allocation of the first task (4x2) into an initial 8x10 system. The system status, the corresponding Q-Tree, and a special node with "size-expanding" are illustrated in the following figure:

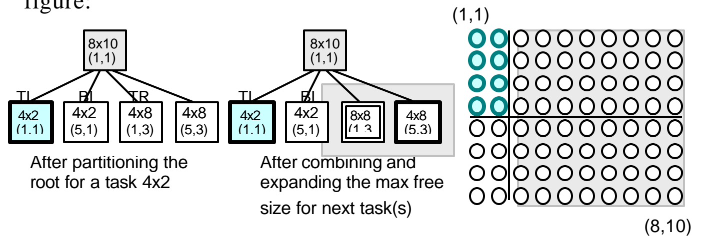


Figure 3 illustrates an example of "how to find the best sub-system, based on the proposed best fit criteria". Given a 10x10 system in which during run time there are two tasks allocated (4x4 and 3x5) at <(1,1), (4,4)> and <(8,6), (10,10)>. Suppose a new incoming task requests a 3x4 sub-system. After step 1-3 (in Section 3.4), the best sub-system is BL (6x4) at level 2 since it can maintain the maximum free sub-system (7x6) at <(1,5), (7,10)>. In step 4, the sub-system BL is then partitioned for the 3x4 request. Two candidates S_1 and S_2 have the same properties (i.e., diffSF = 1, maxFS = 12 (after partitioning), etc.) but different MCF (i.e., MCF(S_1) = 2¼, MCF(S_2) = 2). Figure 3.a shows the Q-Tree that contains α or S (the best sub-system from step 1-3) and its combinable buddies β_1 (or TL at level 2) and β_2 (or BR at level 2). Then, sub-buddies α_1 (or S_1) and α_2 (or S_2) of S after partitioning are described in Figure 3.b and 3.c. Finally, S_2 which yields the minimum MCF is selected as the best buddy or sub-partition after partitioning.

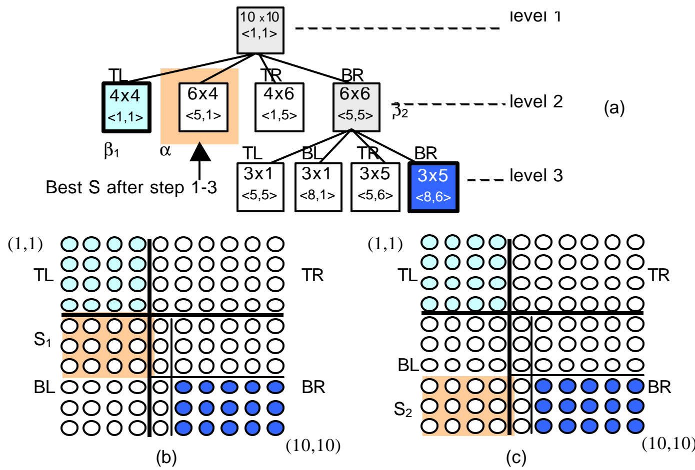


Figure 3: (a) The best sub-system after DFS into all nodes in the Q-Tree; (b) a candidate (S_1) after partitioning the best node in (a); and (c) the final best sub-partition (S_2).

When a task is finished, the processor "deallocation" will find the allocated position of that task by searching into the Q-Tree (or will directly go to that node if there is a pointer stored in the finished task). After freeing the node that stores information, the combining process is applied to combine all available sub-partitions from that node to the root (if possible). Note: "size-expanding" is applied (for a similar reason to that in the allocation process.) After that, if there are task(s) in the waiting queue, a FCFS scheduling will be applied to perform scheduling and allocation of these tasks.

```
QTree-Deallocation  (TF)        /* TF: finished task has a pointer to its       */
{                               /*     corresponding node in the Q-Tree)        */
   QTnode = TF->QTnode;  UpdateOrFreeQTree(QTnode, TF);
   ComN = QTnode;               /* Combined node (= NULL if no combine)  */
   While (ComN != NULL) {       /* Combining form the finished node to root */
      if (comStatus(QTnode = Expand))  /* Identify expanding status        */
          ComN = UnexpandFreeCombineNode(QTnode);
      else ComN = QTnode->Prev;        /* Regular (non-expand) node       */
      if (ComN != NULL)         /* Unexpanded size of other buddies (first)  */
          UnexpandEachof4Buddy(ComN) if (it is expanded and free);
      QTnode = ComN;            /* Store the last combined node            */
      if (All4Buddies(QTnode) = Free) ComN=CombineAllBuddy(ComN);
      else ComN = NULL;         /* No combined result (then stop combining)*/
   } /* #iterations ≤ depth of tree ≅ max(R,C) ≅ √N, N = RxC (system size)  */
   maxSS = CombineAndFindMaxFreeSubSystem (QTnode->Prev);
   if (maxSS = NULL) maxSS = CombineCase1ForMaxSS(QTnode);
   ExpandNode(QTnode);          /* Expand node size for sub-sequent task    */
}
```

Figure 4 illustrates an example of the "combining process" after a task is finished. Suppose the task (3x5) at <(8,6), (10,10)> of the example in Figure 3 is finished. Figure 4.a shows the Q-Tree after being freed of the finished node, Figure 4.b shows the Q-Tree after combining all 4 buddies, and Figure 4.c shows the Q-Tree after expanding the maximum size of the last combined node.

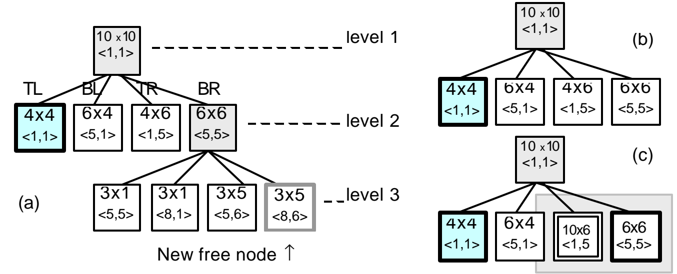


Figure 4: The Q-Tree: (a) after free a finished node, (b) after combining process, and (c) after expanding process.

3.6 Time Complexity Analysis

<u>THEOREM 1</u>: Time complexity of the Q-Tree-based allocation to find the best sub-system for each incoming task is $O(N_a)$ time, where N_a is the number of allocated tasks.

PROOF: The number of recursive iterations are at most the number of nodes in the Q-Tree since only free nodes, whose sizes are larger than or equal to the request, are visited. Since the number of leaf nodes in the Q-Tree is $N_a + N_f$ ($\leq N$), where N_a is the number of allocated tasks and N_f is the number of free nodes, the number of internal nodes is at most (#leaf nodes-1) divided by 3. Therefore the total number of nodes in the Q-Tree is at most $[(N_a+N_f)+(N_a+N_f-1)/3]$ nodes. For each leaf node, its operation is to compute the best fit value (i.e., overlap status, diffSF, maxFS, and CF). The "overlap status" best fit function can be computed in O(1) time (see Definition 3). The "diffSF and maxFS" best fit functions are also computed in O(1) time (see Definition 4). The "CF" best fit functions can also be computed from only neighbor nodes (at most 4 sides) which are also fixed (see Definition 5). Hence, the time complexity of DFS to visit all nodes is $O(N_a+N_f)$. Note: In DFS, finding the maximum free size (of the current system status) is computed only once in $O(N_a+N_f)$ time. The partitioning process is also computed only once after DFS to find the best free sub-system in O(1) time (see Definition 1 & 6). After partitioning, the combining process is applied to the corresponding (internal) node. All possible combined sub-systems of any internal node can be identified in O(1) time since the number of recognized sub-systems is fixed (see Definition 2). Therefore, time complexity of the best fit Q-Tree-based sub-system allocation is $O(N_a+N_f)$ or $O(N_a)$ if $N_a > N_f$ is assumed.

<u>THEOREM 2</u>: Time complexity of the Q-Tree-based deallocation to free the Q-Tree node(s) and to combine max free size of the corresponding QT's nodes is $O(\sqrt{N})$ time, where N is the system size (N = R x C).

PROOF: To go to the node that stores allocation information is O(1) time. Since the maximum depth of the Q-Tree is max(R,C) or $\cong \sqrt{N}$, the combining of all 4 Buddies from the finished sub-system (or node) to the root (if possible) is at most $\sqrt{N}$ steps. The final combining for the maximum free sub-system is O(1) time. Hence, time complexity of the Q-Tree-based deallocation is $O(\sqrt{N})$.

4. PERFORMANCE EVALUATION

4.1 Methodology and Complexity Comparison

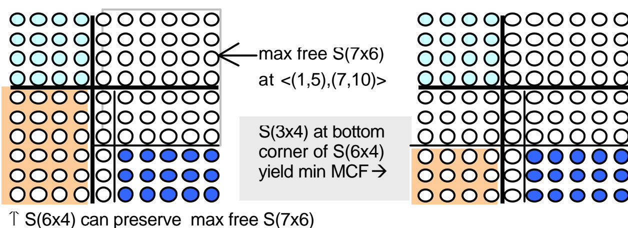
Existing mesh-based strategies were illustrated in Section 2 and summarized here in Table1. Early mesh-based allocations are **first fit** approaches (such as 2BD, FS, FFbit-map, AS, and FFwp) which yield allocation complexity in O(N) time, except those of 2BD, AS, and FFwp which are O(logN), $O(N_aN)$, and O(N/logN) respectively, where N is the system size (N = R x C) and N_a is the number of allocated tasks ($N_a \leq N$). The FFbit-map, AS, and FFwp strategies give better system performance than 2BD and FS. The FFwp gives similar results to the FFbit-map. Later allocation strategies are **best fit** approaches which use the maximum boundary value such as the BL in $O(N_a^3)$, and FL in $O(N_f^2)$, which is an improvement over the BF bit-map, where N_f is the number of free sub-systems. Recent best fit strategies use the more efficient best fit criteria; an example is the FSL (which uses fragmentation factor criteria to maintain a number of maximum free sizes) with time complexity $O(N_f^2)$. The FSL strategy gives better system performance results than both the BL and FL strategies.

Table 1: Mesh-based complexity and performance comparisons.

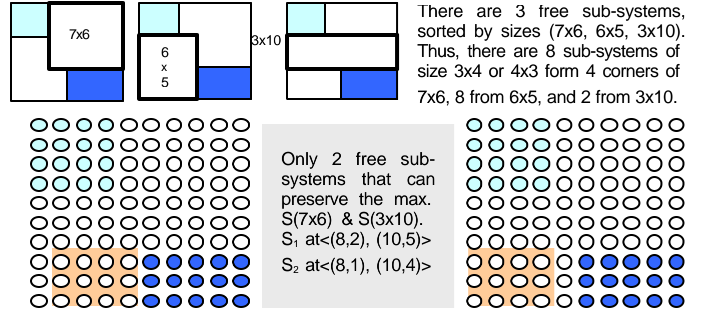
Processor allocation Strategies	Time complexity		Task rotation	System performance
	Allocation	Deallocation		
2-D Buddy: 2BD [10]	O(log N)	O(N)	No	No comparison
Frame Slide: FS [2]	O(N)	O(1)	No	Better than 2DB
FF & BFbit-map: [18]	O(N)	O(N)	No	Better than FS
Adaptive scan: AS[6]	$O(N_a N)$	O(1)	Yes	Better than FS
Busy List : BL [4, 5]	$O(N_a^3)$	O(1)	Yes	Better than BF
Free List : FL [12]	$O(N_f^2)$	$O(N_f^2)$	Yes	Better than BF
FF with partition: [14]	O(N/logN)	O(N/logN)	No	Close to FF
Quick Allocation: [17]	$O(N_a\sqrt{N})$	O(1)	Yes	Better than BL
Free Sub-List:FSL[9]	$O(N_f^2)$	$O(N_f^3)$	Yes	Better than FL
New Q-Tree	$O(N_a)$	$O(N_a)$	Yes	(see Section 4.2)

Our proposed "Q-Tree-based approach" improves system performance and yields similar results to (or better results than) the FSL, BL, and QA (shown later in Section 4.2) with better time complexity. The BL strategy [5] makes the best decision in $O(N_a^3)$, based on the maximum boundary value (BV). However, as mentioned in [9], this may cause the worst case selection when there are many sub-systems with the same BV. The FSL strategy [9] tends to cause minimum system fragmentation and also minimum average waiting time since it makes a decision in $O(N_f^2)$ using the more efficient condition (that of preserving a number of max. free sizes). The Q-Tree-based strategy makes the best decision in $O(N_a)$ based on the combination of various conditions and tends to yield similar results to the FSL as described below. Suppose there are two tasks allocated at <(1,1), (4,4)> and <(8,6), (10,10)>. For the next incoming task (3x4), the Q-Tree selects the best sub-system in a way similar to the FSL. In this case, the BL may cause the worst case selection and the QA [17] selects the worst sub-system due to the fewer conditions used.

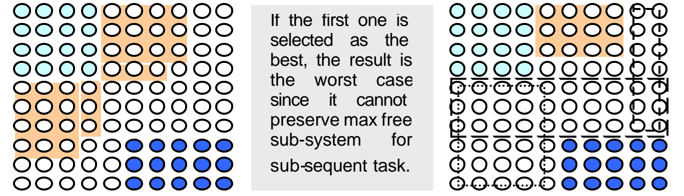
Q-Tree-based allocation (see Figure 3 in Section 3.5)



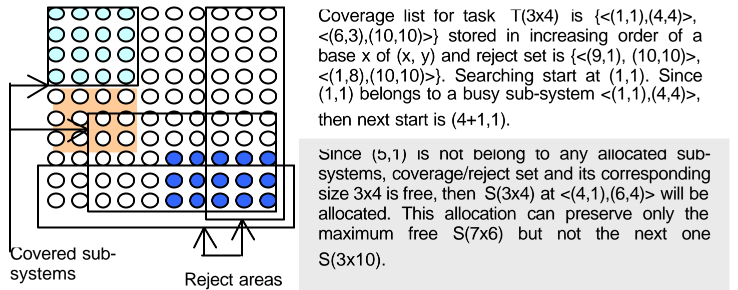
Free Sub-List (FSL) allocation (see Section 2.9)



Busy List (BL) allocation (see Section 2.6)
There are 11 sub-systems that have the same max BV = 7 and 4 of them are:



Quick Allocation (QA) allocation (see Section 2.8)



4.2 System Performance Evaluation

By simulation studies, system performance is evaluated in terms of **system utilization** (U_{sys} = #busyPEs/N), **system fragmentation** ($1-U_{sys}$), and **average task completion time** (= average allocation + execution + waiting time). In this study, the simulation is event-driven in which events are incoming tasks (or jobs) and finishing tasks. The system performance is measured at the steady-state operation of the system under various parameter settings in the experiments (i.e., various system sizes, task-size distributions, arrival rates, workloads, etc.). In each experiment, the number of simulation time units is around 5,000-50,000 units and the number of incoming tasks is approximately 1,000-10,000 tasks, according to the settings of system size, task-size distribution, and the steady state condition. For each evaluation, a number of different data (or task sizes) are generated and the algorithm is repeated until the average system performance value does not change. Experimental results of applying the Q-Tree-based strategy are represented for both **static system performance** (with concerning only processor allocation for incoming job(s)) and **dynamic system performance** (with taking into account of deallocation for some finished jobs). In each experiment, two task-size distributions are used: Uniform $U(\alpha,\beta)$ and Normal $N(\mu,\sigma)$. For each distribution, system sizes (N = R x C) are varied and task sizes [1x1-RxC] are generated, where $\alpha=1$, β=max (R, C) for uniform and $\mu=\sigma$=max (R, C)/2 for normal. Other parameters are fixed such as task arrival rate ~ Poisson (λ) (or inter-arrival time ~ Exp($1/\lambda=5$)), service time ~ Exp($\mu=10$).

$_{sys}$) (or system fragmentation = 1-U_{sys}) under static allocation. Task sizes were generated in a range of [1x1 – R x C]. In Table 2, for all test cases, the Q-Tree-based strategy performed the best or maximum (static) system utilization by about 54% for Uniform and 59% for Normal distribution, which improves up to 30% over the FSL, BL, and QA strategies.

Table 2: Effect of system sizes to (static) system utilization.

Task Size Dist.	System size (N = RxC)	Q-Tree	Free Sub-List (FSL)	Busy List (BL)	Quick Allocation
Uniform	128 x 128	54.70 %	54.04 %	51.32 %	48.40 %
U (α β)	256 x 256	53.21 %	53.01 %	50.72 %	47.82 %
[1x1–RxC]	512 x 512	52.87 %	52.08 %	50.71 %	47.52 %
Normal	128 x 128	59.58 %	57.46 %	57.46 %	49.64 %
N (μ, σ^2)	256 x 256	59.72 %	57.57 %	57.57 %	46.15 %
[1x1–RxC]	512 x 512	59.87 %	57.67 %	57.67 %	46.20 %

Experiment 2 was to investigate the "effect of system sizes" to average system utilization under dynamic allocation/deallocation. In Figure 3, for all test cases of the Uniform distribution, the Q-Tree yielded the best average system utilization ≰ 43%) and standard deviation (±12), which was an improvement of up to 20 %. For Normal distribution, the Q-Tree gives similar results to FSL (different < 5%) which are better than BL and QA by up to 33%.

Table 3: Effect of system sizes to (dynamic) average system utilization.

Task Size Dist.	System size (N = RxC)	Q-Tree	Free Sub-List (FSL)	Busy List (BL)	Quick Allocation
Uniform	128 x 128	43.90 ±12	42.68 ±18	43.15 ±18	36.47 ±15
U (α β)	256 x 256	43.50 ±12	42.28 ±18	41.72 ±18	35.05 ±12
[1x1–RxC]	512 x 512	43.21 ±12	42.03 ±18	42.47 ±18	35.05 ±12
Normal	128 x 128	44.96 ±19	46.75 ±19	41.69 ±18	29.95 ±16
N (μ, σ^2)	256 x 256	45.21 ±19	47.01 ±19	43.73 ±20	29.53 ±16
[1x1–RxC]	512 x 512	45.30 ±19	47.10 ±19	42.61 ±18	29.57 ±16

Experiment 3 was to investigate the "effect of workloads" to average task completion time (or average allocation + execution + waiting time) under dynamic allocation/deallocation on a fixed system size N=64x64, (uniform) task sizes [1x1–64x64], and incoming rate λ (or inter arrival time = Exp (5)). Since workload = $\lambda\mu p/N$, (p = average PEs/task), then task service time (Exp (μ)) was varied. The allocation time of each task was computed according to each strategy (i.e.,$O(N_a)$ in the Q-Tree) and normalized (by assume 1 simulation time unit ≅ 10 algorithm time unit) before being added to the task completion time. Figure 5 shows that the Q-Tree yields the best (or minimum) task completion time and the QA gives the worst task completion time since it varies according to the system size.

5. CONCLUSIONS

In this paper we introduced a new more efficient best fit Q-Tree-based sub-system allocation/deallocation technique. New and more powerful best-fit criteria are used (i.e., the "maintain maximum free sizes" criterion, the "minimum different size factor" criterion, the "maximum free size after partitioning" criterion, and the "minimum combining factor" criterion), and the time complexity has been reduced to $O(N_a)$, where N_a is the number of allocated tasks (N_a<N) and N is the system size. By simulation studies, a number of experiments were carried out to investigate and evaluate the system performance when applying the Q-Tree strategy and compare it to other existing strategies. System performance was measured in terms of system utilization, system fragmentation, and average task completion time. The simulation results showed that the Q-Tree approach yielded the best system utilization and the best average task completion time, representing an improvement of up to 33% over BL, FSL, and QA approaches.

Recently we have applied our Q-Tree-based approach to torus-based parallel system configurations resulting in more sub-systems recognition than other techniques. We have also applied our approach to multi-dimensional meshes and toruses with equally good results. These are not presented here.

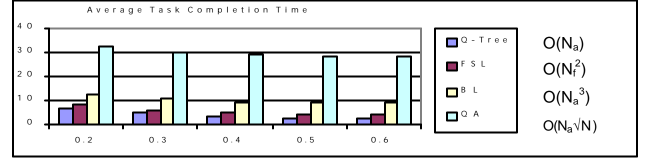


Figure 5: Effects of workload to average task completion time.

6. REFERENCES

[1] Alverson, R. et al. The Tera computer System. Proc. 1990 Int'l Conf. Supercomputing, 1990, 1-6.

[2] Chuang, P.J. and N.F. Tzeng, An Efficient Submesh Allocation Strategy for Mesh Computer Systems. Proc. Int'l Conf. on Distributed Computing Systems, May 1991, 256-263.

[3] Chuang, P.J. and N.F. Tzeng, Allocating Precise Submesh in Mesh-Connected Systems. IEEE Trans. on Parallel and Distributed Systems, v.5(2), 1994, 211-217.

[4] Das Sharma,D. and D.K. Pradhan, A Fast and Efficient Strategy for Submesh Allocation in Mesh-Connected Parallel Computers. IEEE Symp. Parallel and Distributed Processing, 1993, 682-689.

[5] Das Sharma, D. 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, 1996, 106-118.

[6] Ding, J. 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, 1993, 193-200.

[7] Intel, A Touchstone DELTA System Description. Supercomputer Systems Division, Intel Corporation, Beaverton, OR 97006, 1991.

[8] Intel, Paragon XP/S Product Overview. Supercomputer Systems Division, Intel Corporation, Beaverton, OR 97006, 1991.

[9] Kim, G. 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, 175-185.

[10] Li, K. 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), 1991, 413-422.

[11] Li, K. and K.H. Cheng, A Two-Dimensional Buddy system for Dynamic Resource Allocation in a Partitionable Mesh-connected System. J. of Parallel and Distributed Computing, v.12, 1991, 79-83.

[12] Liu, T. and et. al. A Submesh Allocation Scheme for Mesh-Connected Multiprocessor Systems. Proc. 1995 Int'l Conf. Parallel Processing, vol.II, 1995, 159-163.

[13] Mattson et al. Intel/Sandia ASCI system. Proc. of the 10th Int'l Parallel Processing Symp., 1996.

[14] Mohapatra, P. Processor Allocation Using Partitioning in Mesh Connected Parallel Computers. J. of Parallel and Distributed Computing, v.39, 1996, 181-190.

[15] Srisawat, J. and N. A. Alexandridis, Efficient Processor Allocation Scheme with Task Embedding for Partitionable Mesh Architectures. The 11th Intl. Conf. on Computer Applications in Industry and Engineering 98, Las Vegas, November 11-13, 1998, 309-312.

[16] Srisawat, J. 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, 241-244.

[17] Yoo, S. and et. al. An Efficient Task Allocation Scheme for 2D Mesh Architectures. IEEE Trans. on Parallel and Distributed systems, v.8(9), 1997, 934-942.

[18] Zhu, Y. Efficient Processor Allocation Strategies for Mesh-Connected Parallel Computers. J. of Parallel and Distributed Computing, v.16, 1992, 328-337.