# AN EFFICIENT TREE-BASED PROCESSOR ALLOCATION SCHEME FOR MESH CONNECTED SYSTEMS

*Shuen-Fu Tsai, Chin-Laung Lei, and Alexander I-Chi Lai*

Department of Electrical Engineering,
National Taiwan University, Taipei, Taiwan, R.O.C.
Email: {afu, alex}@fractal.ee.ntu.edu.tw,
lei@cc.ee.ntu.edu.tw.

## ABSTRACT

Processor allocation is one of the key issues in the management of mesh connected systems. In this paper, we propose an efficient tree-based processor allocation scheme, called Tree-Allocation (TA), and an Earliest-Available-First-Reserved (EAFR) strategy for mesh connected systems. The TA scheme uses a binary tree to keep the processor allocation information, and its time complexity is $O(N_B)$ where $N_B$ is the number of busy meshes. The EAFR strategy reserves an earliest available submesh for the incoming task which cannot be executed immediately. Our simulation results indicate that the TA scheme outperforms other allocation schemes in the literature in terms of the average allocation time, system utilization and average waiting time.

## 1 INTRODUCTION

In a mesh connected system, the job scheduler uses a scheduling policy to select a task for execution, and the processor allocator uses an allocation scheme to find a *free submesh* for the selected task. Most of the existing allocation schemes for mesh connected systems, including 2-dimensional buddy system (2DB) [5], frame slide (FS) [2], first-fit (FF) and best-fit (BF) [12], adaptive scan (AS) [3], adjacency strategy [8], quick allocation (QA) [11], and Free-Submesh-List (FSL) [4], allocate physically *contiguous submeshes* instead of *non-contiguous* ones [6], because non-contiguous submeshes induce non-atomic (i.e. larger) communication overhead between logically adjacent processors. However, contiguous allocation schemes suffer from the *fragmentation* problem of unusable or wasted chunks of free processors. Specifically, fragmentation can be further subdivided into two categories: *internal* fragmentation in which there are unused processors in a busy submesh; and *external* fragmentation in which an incoming task size (in number of required processors) is smaller than the total number of free processors yet larger than any free submesh. Both kinds of fragmentation severely reduce the processor utilization and hamper the throughput of mesh connected systems. The above allocation schemes record the mesh partitioning status to alleviate the impact of fragmentation; unfortunately, most of them only maintain the information of *busy* submeshes and need additional computation to derive the information of free submeshes, which is time-consuming. The FSL scheme [4] does use a list to directly keep track of the free submeshes, yet the expense of searching the list for a best fit is proportional to the square of the number of active tasks, substantially decreasing the performance especially for heavy-loading systems.

In this paper, we propose an efficient processor allocation scheme, called *Tree-Allocation* (TA), for mesh connected systems. The TA scheme uses a binary *allocation tree* to keep the complete allocation status of the whole mesh. Each leaf of the allocation tree represents a submesh partition, either free or busy; and the internal nodes represent the adjacency relation of the submeshes. For an incoming task, our scheme selects a free submesh from the leaf nodes. If the selected free submesh is larger than the task, then we partition that submesh, both *vertically* and *horizontally*, to fit exactly the height and width of the incoming task to eliminate internal fragmentation totally. In order to suppress external fragmentation, we propose an *Earliest-Available-First-Reserved* (EAFR) strategy to pre-allocate currently busy submeshes for pending tasks. Each node in the allocation tree has a field called *ready-time* (rtime) which indicates the time that this node will become available. By examining *rtime*, we can reserve the earliest available submesh for the task and go on the next incoming task. Searching and managing available submeshes in the binary allocation tree is more efficient than other mechanisms used in previous approaches, and the application of EAFR to our TA scheme further improves the system utilization.

The rest of the paper is organized as follows. In Section 2, we give a brief review of previous processor allocation schemes for mesh connected systems. In section 3, we present our TA allocation scheme and EAFR strategy. Section 4 shows the simulation results and performance comparison with other schemes. Finally, conclusions are drawn in section 5.

## 2 PREVIOUS APPROACHES

The 2-Dimensional Buddy (2DB) scheme [5] is a generalization of the famous buddy system. It can only be applied to allocate a square submesh with side length $2^n$ for some integer $n$. For an incoming task of size $w \times h$, this scheme allocates a submesh of size $u \times u$ where $u = 2^{\lceil \log_2 \max(w,h) \rceil}$. The disadvantage of the scheme is that it cannot recognize rectangular submeshes with arbitrary side length and may cause large internal fragmentation.

The Frame Sliding (FS) scheme [2] is applicable to non-square meshes with arbitrary side length. For an $w \times h$ incoming task, the scheme uses a frame of size $w \times h$ to find a free submesh. The scheme slides the frame from the lower left node along the horizontal and vertical direction where its horizontal and vertical strides are equivalent to $w$ and $h$. If the current node cannot serve as the lower left node of the frame, it moves to the next frame along the horizontal direction. If it cannot find a free submesh in the current row, then it moves to the next frame along the vertical direction. The searching is stopped when a free submesh is found or all candidate frames are checked. This scheme can reduce the internal fragmentation and yield a better performance than 2DB. However, the searching process may result in allocation miss.

In order to solve the allocation miss problem, the First-Fit (FF) and Best-Fit (BF) allocation schemes are proposed in [12]. These schemes use bit arrays to indicate which nodes have enough free neighbors to serve the lower left node of the task. The FF scheme searches these bit arrays for the first free submesh, and the BF scheme searches all the free regions and chooses one with the largest number of busy neighbors. However, these schemes still suffer from the external fragmentation.

The Adaptive Scan (AS) scheme is proposed in [3]. The scheme uses the scanning operation instead of the sliding operation used in FS. It starts from the lower left node of the mesh. If the current node cannot serve as the lower left node for the task, it moves to the next column. If it still cannot find a free submesh in the current row, it moves to the left-most node of the next row. A strategy called *address translation* is used to change the orientation of the incoming tasks. If the searching fails to find a free submesh, then it rotates the orientation of the task from $w \times h$ to $h \times w$ and redo this scheme again. This scheme improves the FS scheme by the scanning operation and the rotation of the task.

The Quick Allocation (QA) scheme is purposed in [11]. Like AS, the address translation strategy is adopted. It uses a one-dimensional array called *last_covered* to keep the x-coordinate of the right-most covered node of each row. By using the last_covered, this scheme can find a free submesh without scanning the entire mesh, and reduces the allocation overhead.

The Free-Submesh-List (FSL) scheme is purposed in [4]. It is based on maintaining a list of free submeshes, all of which dominate other free submeshes. For an allocation request, it chooses the best-fit submesh which causes the least amount of potential external fragmentation. However, the allocation/deallocaton overhead of this scheme is quite large.

## 3 THE TREE ALLOCATION SCHEME

A two-dimensional rectangular mesh with $a \times b$ nodes is denoted by $M(a,b)$ where $a$ and $b$ represent the width and height of the mesh, respectively. Each node in the mesh represents a processor, and its address is identified by its coordinate $< x, y >$ where $1 \leq x \leq a$ and $1 \leq y \leq b$. A submesh $s$ is identified by the quadruple $< x, y, x', y' >$ where $< x, y >$ is the lower left corner of the submesh and $< x', y' >$ is its upper right corner. A submesh $s = < x, y, x', y' >$ is *free*, if all nodes in it are not allocated to any task. A submesh is *busy*, if all nodes in it are allocated to a task.

An incoming task is denoted by $t(w, h, arr, sev)$ where $w$ and $h$ are the width and height of the task, and $arr$ and $sev$ denote the arrival and service time of the task, respectively. Each task is assigned to a submesh of the same size in $M(a,b)$. The rotation of changing task from $t(w, h, arr, sev)$ to $t(h, w, arr, sev)$ is called *address translation* [3]. With this mechanism, we can find more free submeshes for allocation [3, 4, 11].

In our allocation scheme, we use a binary *allocation tree* $T = (V, E)$ to record the allocation status. Each node $q \in V$ is represented by a 7-tuple $(x, y, w, h, stat, resvd, rtime)$ where $q$ is a submesh of size $w \times h$ whose lower left corner is $< x, y >$. Depending on whether $q$ is *free* or *busy*, *stat* is set accordingly. If $stat = internal$, it indicates that $q$ is an internal node of $T$. *resvd* is a flag which indicates whether $q$ has been reserved for a task or not. If $q$ has been reserved, then $resvd = 1$; otherwise $resvd = 0$. *rtime* is an integer value which indicates the time that $q$ will become available. In particular, if $rtime = 0$, then $q$ is *free.*

**Definition 1.** *The reservation set of an allocation tree $T = (V, E)$, denoted by $R$, is a set of nodes in $T$ which have been reserved for waiting tasks. Initially, $R$ is empty.*

**Definition 2.** *Let $V_R = \{v \in V \mid v \notin R$ and $v$ is a descendent of some node in $R$ $\}$, contain all nodes in any subtree of $T$ rooted by a node in $R$.*

Let function $parent_rtime(q)$ return $q'.rtime$ where $q'$ is the nearest ascendent of $q$ and $q' \in R$ and $q \in V_R$.

$R$, $V_R$, and $parent_rtime()$ can be simply maintained by the flag *resvd* of each node. If $rsevd = 1$, it indicates that we insert this node into $R$. When *rsevd* is set from 1 to 0, it indicates that we remove this node from $R$. When we traverse $T$, we check whether the

current node has been reserved or not. If this node has been reserved, then we know that it's descendent nodes are in $V_R$ and the $parent_rtime()$ value. We also use a waiting queue, called $r_queue$, to keep the reserved tasks in order.

In order to reduce the internal fragmentation, we use two partition methods to divide a submesh into 2 or 3 submeshes where one of these submesh is used for the requesting task. One is the horizontal partition, it is used when the selected free node (submesh) $q$'s height is larger than that of the requesting task. The other is the vertical partition method which is used when the selected free node $q$'s width is larger than that of the requesting task. By performing the horizontal and vertical partitions, a free submesh matches the size of the requesting task is generated for allocation. The order of applying these two partition methods is determined by which method will result in a larger free submesh. The details of the partition methods are shown as follows.

```
Procedure Partition_Submesh(q, t_i)
/* partition a free node q for the task t_i(w_i, h_i, arr_i, sev_i) */
/* output a node whose size is equal to t_i */
{
    if (q.w = w_i and q.h = h_i) {
        q_f = q;
    } else if (q.w = w_i and q.h > h_i) {
        q_f = Horizontal_Partition(q, t_i);
    } else if (q.w > w_i and q.h = h_i) {
        q_f = Vertical_Partition(q, t_i);
    } else if (q.w * (q.h − h_i) > (q.w − w_i) * q.h) {
        q' = Horizontal_Partition(q, t_i);
        q_f = Vertical_Partition(q', t_i);
    } else {
        q'' = Vertical_Partition(q, t_i);
        q_f = Horizontal_Partition(q'', t_i);
    }
    return q_f;
}
```

```
Procedure Horizontal_Partition(q, t_i)
/* partition a free node q into q_l and q_r for the task
    t_i(w_i, h_i, arr_i, sev_i) along the horizontal direction */
/* output a node whose height is equal to h_i */
{
    generate q's left child q_l with q_l.w = q.w, q_l.h = h_i,
        q_l.x = q.x, q_l.y = q.y;
    generate q's right child q_r with q_r.w = q.w,
        q_r.h = q.h − h_i, q_r.x = q.x, q_r.y = q.y + h_i;
    q.stat = internal, q_l.stat = free, q_r.stat = free;
    q_l.resvd = 0, q_r.resvd = 0, q_l.rtime = 0, q_r.rtime = 0;
    return q_l;
}
```

```
Procedure Vertical_Partition(q, t_i)
/* partition a free node q into q_l and q_r for the task
    t_i(w_i, h_i, arr_i, sev_i) along the vertical direction */
/* output a node whose width is equal to w_i */
{
    generate q's left child q_l with q_l.w = w_i, q_l.h = q.h,
        q_l.x = q.x and q_l.y = q.y;
    generate q's right child q_r with q_r.w = q.w − w_i,
        q_r.h = q.h, q_r.x = q.x + w_i and q_r.y = q.y;
    q.stat = internal, q_l.stat = free, q_r.stat = free;
    q_l.resvd = 0, q_r.resvd = 0, q_l.rtime = 0, q_r.rtime = 0;
    return q_l;
}
```

## 3.1 Tree-Allocation scheme

The search strategy used in our TA scheme is Best-Fit. We choose a free node $q$ that can accommodate task $t_i$ with the smallest area for allocation. When there are several candidates, we choose the one with the smallest depth. If $q \in V_R$, we also need to check the finish time of $t_i$ and $parent_rtime(q)$. The depth of $q$ and $parent_rtime(q)$ can be simply derived by using the Breadth-First-Search (BFS) traversal scheme.
The TA allocation scheme is shown as follows. In step 1, for an incoming task $t_i$, we use BFS to find a free node $q$ in $T$ that can accommodate $t_i$ with the smallest area. If such a node cannot be found, then we use the EAFR strategy to reserve a node for the task and go on processing the next incoming task. In step 2, if a free node $q$ is found, then we invoke the procedure of partition submesh on $q$, and return a free node $q_f$ of the same size as $t_i$ for allocation. In step 3, after we allocated this task, we need to update the $rtime$ field of nodes from $q_f$ backward to the root of $T$. When $q_f$'s $rtime$ is greater than its parent's, then we update the $rtime$ of its parent, and continue this procedure until the $rtime$ of the current node is less than its parent or the current node is the root of $T$.

```
Allocation(t_i, T)
/* allocate a free node q_f for the incoming task
    t_i(w_i, h_i, arr_i, sev_i) */
/* T is the allocation tree, R is the reservation set, and V_R is a
    set of nodes rooted by R */
/* current_time is the current time of the system */
/* parent_rtime() return the reserved time of this node */
{
    /* Step 1 */
    find a free node q in T that can accommodate t_i
        with the smallest area using the BFS traversal order;
    if (q is found and ((q ∉ V_R) or ((q ∈ V_R) and
        (current_time + sev_i < parent_rtime(q))))) {
        /* check whether t_i needs rotation or not */
        if t_i needs rotation, then swap t_i(w_i, h_i, arr_i, sev_i) into
            t_i(h_i, w_i, arr_i, sev_i);
    } else {
        /*perform the reservation (EAFR) strategy */
        EAFR(t_i, T);
        exit;
    }
    /* Step 2 */
    q_f = Partition_Submesh(q, t_i);
    allocate q_f for t_i;
    q_f.stat = busy, q_f.rtime = current_time + sev_i;
    /* Step 3 */
    let q_p be the parent node of q_f;
    while ((q_f is not the root of T) and
            (q_f.rtime > q_p.rtime)) {
        q_p.rtime = q_f.rtime;
        q_f = q_p;
        let q_p be the parent node of q_f;
    }
}
```

The detail deallocation scheme is shown as follows. In step 1, the busy node $q$ will be set as a free node. If $q$ has not been reserved and $q$'s sibling node is also a free node, then we need to delete $q$ and $q$'s sibling node, and set $q$'s parent node as a free node. We continue this procedure from $q$ backward to the root of $T$. In

Table 1: The tuples of arriving tasks.

| $t_i$ | $w_i$ | $h_i$ | $arr_i$ | $sev_i$ |
|---|---|---|---|---|
| $t_1$ | 2 | 1 | 1 | 6 |
| $t_2$ | 1 | 3 | 2 | 6 |
| $t_3$ | 1 | 1 | 3 | 6 |
| $t_4$ | 2 | 2 | 4 | 9 |
| $t_5$ | 1 | 4 | 5 | 6 |
| $t_6$ | 1 | 2 | 6 | 6 |
| $t_7$ | 1 | 1 | 7 | 7 |

step 2, if the current node $q$ has been reserved, then we need to remove $q$ from $R$ and remove $t_i$ which has reserved $q$ from *r_queue*, and allocate $q$ for $t_i$.

```
Deallocation(q, T)
/* deallocate a busy node q and allocate a reserved task */
/* T is the allocation tree */
/* current_time is the current time of the system */
/* r_queue is a waiting queue used by the reserved tasks */
{
    /* Step 1 */
    let q_s be the sibling node of q;
    while ((q is not the root node of T) and
            (q.resvd = 0) and (q_s.stat = free)) {
        let q_p be the parent node of q;
        delete q and q's sibling node;
        q = q_p;
        let q_s be the sibling node of q;
    }
    q.stat = free, q.rtime = 0;
    /* Step 2 */
    if (q.resvd = 1) {
        /* remove the reserved node q from R */
        q.resvd = 0;
        remove the task t_i which has reserved q from r_queue;
        if t_i needs rotation, then swap t_i(w_i, h_i, arr_i, sev_i) into
            t_i(h_i, w_i, arr_i, sev_i);
        q_f = Partition_Submesh(q, t_i);
        allocate q_f for t_i;
        q_f.stat = busy, q_f.rtime = current_time + sev_i;
        let q_p be the parent node of q_f;
        while ((q_f is not the root of T) and
                (q_f.rtime > q_p.rtime)) {
            q_p.rtime = q_f.rtime;
            q_f = q_p;
            let q_p be the parent node of q_f;
        }
    }
}
```

### 3.2 The Earliest-Available-First-Reserved strategy

In this subsection, we describe the Earliest-Available-First-Reserved (EAFR) strategy. The EAFR strategy is applied only when a task cannot be allocated immediately. Each node in the allocation tree has a *rtime* field which indicates the time that this node will become available. By *rtime*, we can exactly reserve the earliest available node for the task. Just like the TA scheme, the search strategy used in EAFR strategy is Best-Fit. When a task is coming for reservation, we try to find a node in $T$ and not in $R$ and $V_R$ that can accommodate the task with the smallest *rtime* by using the Breadth-First-Search (BFS) traversal order. If a node for reservation is found, then we add this node to the reservation set $R$ and append this task into *r_queue*, and update the *rtime* fields from this node backward to the root of $T$, and go on processing the next incoming task. If such a node cannot be found, then we simply insert this task into the waiting queue and wait for some executing jobs to be released. The details of EAFR strategy are described as follows.

```
EAFR(t_i, T)
/* reserve an earliest available node q for the task
    t_i(w_i, h_i, arr_i, sev_i) */
/* T is the allocation tree, R is the reservation set, and V_R is a
    set of nodes rooted by R */
/* current_time is the current time of the system */
/* r_queue is a waiting queue used by the reserved tasks */
{
    find a node q in T where q ∉ R and q ∉ V_R that can
        accommodate t_i with the smallest rtime using the BFS
        traversal order;
    if (q is found) {
        /* add the earliest available node q to R */
        q.resvd = 1, q.rtime = current_time + sev_i;
        append t_i into r_queue;
        /* update the rtime fields from q backward to the root
            of T */
        let q_p be the parent node of q;
        while ((q is not the root of T) and
                (q.rtime > q_p.rtime)) {
            q_p.rtime = q.rtime;
            q = q_p;
            let q_p be the parent node of q;
        }
    } else {
        insert t_i into the waiting queue;
    }
}
```

In general, all the exist processor allocation schemes use the First-Come-First-Serve (FCFS) scheduling policy. The drawback of the FCFS scheduling policy is "blocking" in nature. A request for a submesh if higher dimensions may block subsequent requests that my be serviceable [9]. However, the EAFR strategy can reserve submeshes for the waiting tasks and go on processing the next tasks, i.e., we can let the next incoming task be executing first and without occurring starvation.

**Theorem 1.** *The TA scheme with EAFR strategy is a starvation-free processor allocation scheme.*

*Proof.* The jobs arrive along the FCFS order. If an incoming task cannot be executing immediately, then we try to reserve an earliest available submesh for it. If such a submesh is found, then we reserve this submesh for the task and insert it into the reservation set $R$. The next incoming task may go on selecting a free node in $T$ and not in $R$ and $V_R$ for allocation. If there is no submesh for reservation, then we simply insert this task to the waiting queue and wait for the executing jobs and reserved jobs to be released. After a certain time, the entire mesh is empty, and the tasks in the waiting queue will begin to allocation. This will not occur the

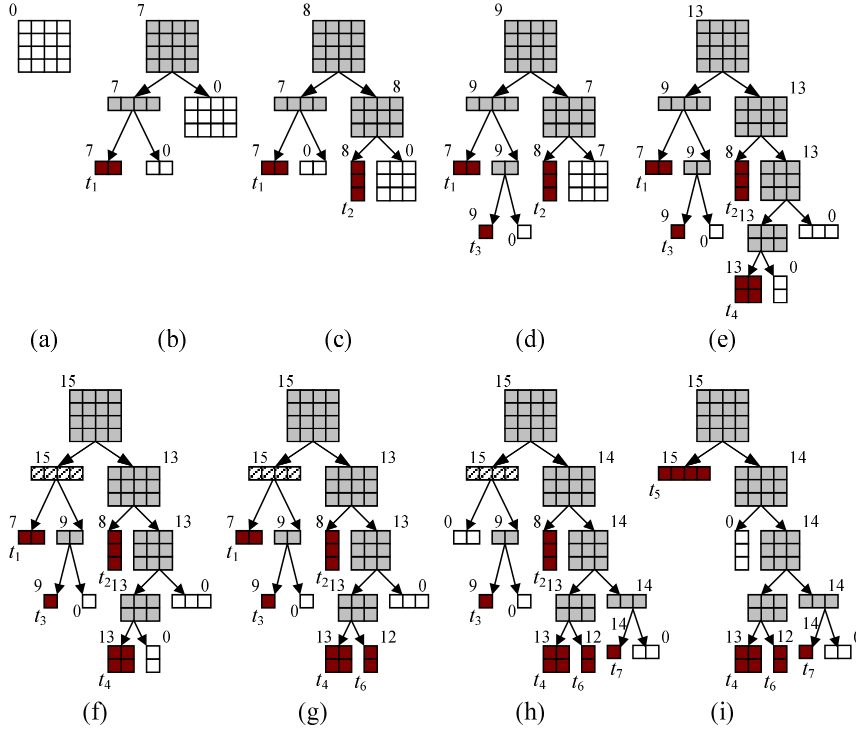


Figure 1: The allocation trees of example 1.

starvation even when we have changed the execution sequence of the tasks. Thus, the EAFR strategy makes the TA scheme be a starvation-free allocation scheme. □

Example 1. There are 7 tasks in Table 1 arriving in the $4 \times 4$ mesh with each time unit per task arriving. The first-come-first-serve (FCFS) schedule is used for these tasks, and our EAFR strategy is applied here to improve the performance. The allocation results of each task are shown in Fig 1.

In Fig 1, the *free* nodes are colored in white, the *busy* nodes are colored in black, the *internal* nodes are colored in gray and the *reserve* nodes are drawn with diagonal line. The number above each node is the value of *rtime*. In (b), the task $t_1(2, 1, 1, 6)$ is assigned to the submesh $< 1, 1, 2, 1 >$ and update the *rtime* from this node backward to the root node. In (c), we assign $t_2$ to $< 1, 2, 1, 4 >$. In (d) and (e), $t_3$ and $t_4$ are allocated. However, $t_5$ cannot be allocated immediately (at the system time $= 5$), then we use the EAFR strategy to reserve the earliest available submesh $< 1, 1, 4, 1 >$ ($rtime = 9$) for $t_5$, and update the *rtime* field of nodes in (f). In (g), we allocate the free submesh $< 4, 2, 4, 3 >$ for $t_6$. In (h), we allocate the free submesh $< 2, 4, 2, 4 >$ for $t_7$. In (i), until we deallocated $t_3$ (at system time $=$ 9), $t_5$ (need rotation) is assigned to the free submesh $< 1, 1, 4, 1 >$. Finally, the system will become idle after $t_5$ is deallocated.

The number in the brackets indicates the system time. The '+' indicates the allocation, and the '-' indicates the deallocation. The $t_i(x, y, w, h, r, arr, sev, wait)$ indicates the tuples of task $t_i$ where $x$ and $y$ denote the lower left corner coordinate, $w$ and $h$ denote the width and height, $r_i$ is flag which indicates whether address translation is used or not. If $t_i$ has been rotated then $r = 1$, otherwise $r = 0$. *arr* and *sev* refer as the arrival and service time of $t_i$, and *wait* indicates the time of $t_i$ in the waiting queue.

| time | +/- | $t_i$(x,y,w,h,r,arr,sev,wait) |
|---|---|---|
| [ 1] | + | $t_1$(1,1,2,1,0,1,6,0) |
| [ 2] | + | $t_2$(1,2,1,3,0,2,6,0) |
| [ 3] | + | $t_3$(3,1,1,1,0,3,6,0) |
| [ 4] | + | $t_4$(2,2,2,2,0,4,9,0) |
| [ 6] | + | $t_6$(4,2,1,2,0,6,6,0) |
| [ 7] | - | $t_1$ |
| [ 7] | + | $t_7$(2,4,1,1,0,7,7,0) |
| [ 8] | - | $t_2$ |
| [ 9] | - | $t_3$ |
| [ 9] | + | $t_5$(1,1,4,1,1,5,6,4) |
| [12] | - | $t_6$ |
| [13] | - | $t_4$ |
| [14] | - | $t_7$ |
| [15] | - | $t_5$ |

# 4 SIMULATION RESULTS AND ANALYSIS

## 4.1 Complexity analysis

Let $N_B$ be the number of busy submehes in an $a \times b$ mesh where $a$ and $b$ are the width and height of the mesh. The 2DB scheme applies only to the square meshes, i.e., $a = b$. Table 2 [4, 8, 11] compares the complexities of allocation, deallocation and searching space of our TA scheme against other schemes. The complete

Table 2: Comparisons of various tasks allocation schemes.

| Scheme | Allocation | Deallocation | Memory | Complete recognition | Internal fragmentation |
|---|---|---|---|---|---|
| 2DB | $\Theta(\log_2 a)$ | $O(ab)$ | $O(ab)$ | No | Yes |
| FS | $O(abN_B)$ | $\Theta(1)$ | $\Theta(N_B)$ | No | No |
| FF | $O(ab)$ | $O(ab)$ | $\Theta(ab)$ | No | No |
| BF | $O(ab)$ | $O(ab)$ | $\Theta(ab)$ | No | No |
| AS | $O(abN_B)$ | $\Theta(1)$ | $\Theta(N_B)$ | Yes | No |
| QA | $O(bN_B)$ | $\Theta(1)$ | $\Theta(N_B)$ | Yes | No |
| FSL | $O(N_B^2)$ | $O(N_B^3)$ | $\Theta(N_B)$ | Yes | No |
| TA | $O(N_B)$ | $O(N_B)$ | $\Theta(N_B)$ | No | No |

recognition [11] means that the allocation scheme can recognize an existing free submesh for an incoming task when one is available. The proofs of our TA scheme are listed in the following theorems.

**Lemma 1.** *Total number of nodes in the allocation tree $T$ ranges between $2N_B - 1$ and $4N_B + 1$.*

*Proof.* Let $N_B$ be the total number of busy nodes, $N_F$ be the total number of free nodes, $N_I$ be the total number of internal nodes and $N$ be the total number of nodes in $T$, i.e., $N = N_B + N_F + N_I$. Initially, there is only a entire free node, so $N_B = 0$, $N_F = 1$, and $N_I = 0$.
Case 1. We both perform the horizontal and vertical partition methods on a free node, i.e., this node will be partitioned into 2 free nodes and 1 busy node. After we perform the partition methods, $N_B = N_B + 1$ and $N_F = N_F + 1$. If we perform the partition methods $N_{B_1}$ times, then $N_B = N_B + N_{B_1}$ and $N_F = N_F + N_{B_1}$.
Case 2. We only perform the horizontal (vertical) partition method to partition a free node, i.e., this node will be partitioned into 1 free nodes and 1 busy node. After we perform the horizontal (vertical) partition method, $N_B = N_B + 1$ and $N_F = N_F$. If we perform the partition method $N_{B_2}$ times, then $N_B = N_B + N_{B_2}$ and $N_F = N_F$.
Case 3. We do not need the horizontal and vertical partition methods to partition a free node, i.e., the width and height length of this node are equal to the requested task. After we have allocated this node, $N_B = N_B + 1$ and $N_F = N_F - 1$. If we repeat $N_{B_3}$ times, then $N_B = N_B + N_{B_3}$ and $N_F = N_F - N_{B_3}$.
From case 1, 2, 3, we know that when $N_B = N_{B_1} + N_{B_2} + N_{B_3}$, then $N_F = N_{B_1} - N_{B_3} + 1$. We know that $0 \leq N_{B_3} \leq N_{B_1} + 1$ because of $0 \leq N_{B_3}$ and $0 \leq N_F$. And $T$ is a binary tree, so $N_I = N_B + N_F - 1$. Therefore, $N = N_B + N_F + N_I = 2(N_B + N_F) - 1 = 2(2N_{B_1} + N_{B_2}) + 1 = 2(N_B + N_{B_1} - N_{B_3}) + 1$. If only case 1 is used, i.e, $N_B = N_{B_1}$ and $N_{B_2} = N_{B_3} = 0$, then we get $N = 4N_B + 1$. If $N_{B_3} = N_{B_1} + 1$, then we get $N = 2N_B - 1$. Thus, the range of $N$ is $2N_B - 1 \leq N \leq 4N_B + 1$. □

**Lemma 2.** *The depth of the allocation tree $T$ ranges between $\lceil \log_2(2N_B - 1) \rceil$ and $2N_B + 1$.*

*Proof.* Case 1. We both perform the horizontal and vertical partition methods on a free node which has the maximal depth. Initially, the depth of $T$ is 1. Performing the partition methods, the depth of $T$ is $1 + 2 = 3$. After performing the second time of partition methods, the depth is $3 + 2 = 5$, and so on. If the number of busy nodes is $N_B$, then the maximal depth is $2N_B + 1$.
Case 2. From the lemma 1, we know that the minimal number of nodes in $T$ is $2N_B - 1$. If $T$ is a complete binary tree, then the minimal depth is $\lceil \log_2(2N_B - 1) \rceil$. Thus, the depth of $T$ is between $\lceil \log_2(2N_B - 1) \rceil$ and $2N_B + 1$. □

**Theorem 2.** *The EAFR strategy has a time complexity of $O(N_B)$.*

*Proof.* In the EAFR strategy, the procedure of reservation uses BFS to find an earliest available node in $T$. Form the lemma 1, we know that the total number of nodes in $T$ is less or equal than $4N_B + 1$. Thus, the time complexity of the EAFR strategy is $O(N_B)$. □

**Theorem 3.** *The TA allocation scheme has a time complexity of $O(N_B)$.*

*Proof.* In the TA allocation scheme, the search procedure uses BFS to find a free node in $T$. Form the lemma 1, we know that the total number of nodes in $T$ is less than $4N_B + 1$. The procedure of updating the *rtime* fields is processing from the busy node backward to the root. From the lemma 2, we know that the maximal depth of $T$ is $2N_B + 1$. Thus, the time complexity of TA allocation scheme is $O(N_B)$. □

**Theorem 4.** *The TA deallocation scheme has a time complexity of $O(N_B)$*

*Proof.* Deallocating a busy node and checking its sibling node are processing from the busy node up to the root node. From the lemma 2, we know that the maximal depth of $T$ is $2N_B + 1$. If the current busy node has been reserved for the task, then we need to reallocate this node for the task. From the theorem 3, we know allocating a reserved task need $O(N_B)$. Thus, the time complexity of TA deallocation scheme is $O(N_B)$. □

**Theorem 5.** *The memory space of TA allocation scheme has a complexity of* $\Theta(N_B)$.

*Proof.* Form the lemma 1, we know that the total number of nodes in $T$ is between $2N_B - 1$ and $4N_B + 1$. Thus, the memory space complexity of TA allocation scheme is $\Theta(N_B)$. □

### 4.2 Simulation results

The performance of the TA allocation scheme is compared with 2DB, FS, FF, BF, AS and QA allocation schemes using the computer simulation. The simulation is event-driven with the events being the allocation and deallocation of jobs. A separate host processor is used for processor allocation/deallocation and task dispatching. There are several measurements used in our simulation:

- *System Utilization* — the percentage of processors that are utilized over time.
- *Average Waiting Time* — the time from a job arrives in the waiting queue until it begins executing.
- *Average Allocation Time* — the time required to allocate a submesh for a task.

CASE 1:
Simulations are conducted for the meshes ranging from $8 \times 8$ to $128 \times 128$. The simulator was developed in C language running on Linux (Pentium II 350). All the simulation use 95% confidence level with the error range of 3%. We employ the same simulation model used in [1, 2, 3, 8, 11, 12]. Initially, the entire mesh is free, and 3000 tasks are generated and queued at the task dispatcher. The residence time of each task is assumed to be uniformly distributed between 5 and 10 time units (second). The tasks are assumed to arrive at each time unit. The side lengths (width and height) of tasks are assumed to be either uniformly or exponentially distributed. For the uniform distribution, the side lengths of the tasks are uniformly distributed between 1 and the side length of the mesh ($L$). For the exponential distribution, the mean is selected as a half of $L$. And those values exceeding the range $[1, L+1)$ were discarded.
The First-Come-First-Serve (FCFS) scheduling policy is used in our simulation model. The task dispatcher always tries to find a free submesh for the first task in the waiting queue. If it fails to find a free submesh, the dispatcher simply waits for a deallocation and then try to allocate again. The TA that does not contain the EAFR strategy is denoted as $\text{TA}^-$, and the TA that contains the EAFR strategy is denoted as $\text{TA}^*$. The system utilization for various size meshes are showed in Fig. 2 (a) and (b), the average waiting time are in (c) and (d), and the average allocation time are in (e) and (f).
Form the Fig. 2 (a)∼(d), we can see that the system utilization and average waiting time of TA are near to the AS and QA. In the Fig. 2 (e) and (f), we can see the average allocation time of TA is the smallest. The EAFR strategy does not gain a lot of performance improvement in the simulation of case 1, it only improves the 3%∼5% utilization.

CASE 2:
We try to use the same simulation model as case 1 except the residence time of each task. We assume that the residence time is uniformly distributed between 5 and 10 time units for the larger tasks, and it is uniformly distributed between 2 and 5 time units for the smaller tasks. For a task $t_i(w_i, h_i, arr_i, sev_i)$, the larger task is defined as $w_i * h_i \geq (a * b)/2$ and the smaller task is defined as $w_i * h_i < (a * b)/2$ where $a$ and $b$ are the width and height of the entire mesh.
The system utilization for various size meshes are showed in Fig. 2 (g) and (h), and the average allocation time is in (i). We can see that the system utilization of TA has a little improvement and is better than the above schemes. And the performance gain of the exponential distribution is not manifest as the uniform distribution. The average allocation time of TA is still the smallest.

## 5 CONCLUSION

In this paper, we have proposed an efficient tree-based allocation scheme and an earliest-available-first-reserved strategy for mesh connected systems. The time complexity of the allocation/deallocation is $\text{O}(N_B)$, the space complexity is $\Theta(N_B)$ and the EAFR strategy is $\text{O}(N_B)$ where $N_B$ is the number of busy submeshes.
From the simulation results, the system utilization and the average waiting time of our scheme are about the same as those of AS and QA, and our average allocation time is the smallest. Our scheme yields better performance when the residence times of the larger tasks are longer than those of the smaller tasks.

## REFERENCES


[1] D. Babbar and P. Krueger, "A Performance Comparison of Processor Allocation and Job Scheduling Algorithms for Mesh-Connected Multiprocessors," Proc. Symp. Parallel and Distributed Processing, pp. 46-53, Oct. 1994.

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

[3] J. Ding and L. N. Bhuyan, "An Adaptive Submesh Allocation Strategy for Two-Dimensional Mesh Connected Parallel Systems," Proc. Int'l Conf. Parallel Processing, pp. II-193-200, Aug. 1993.

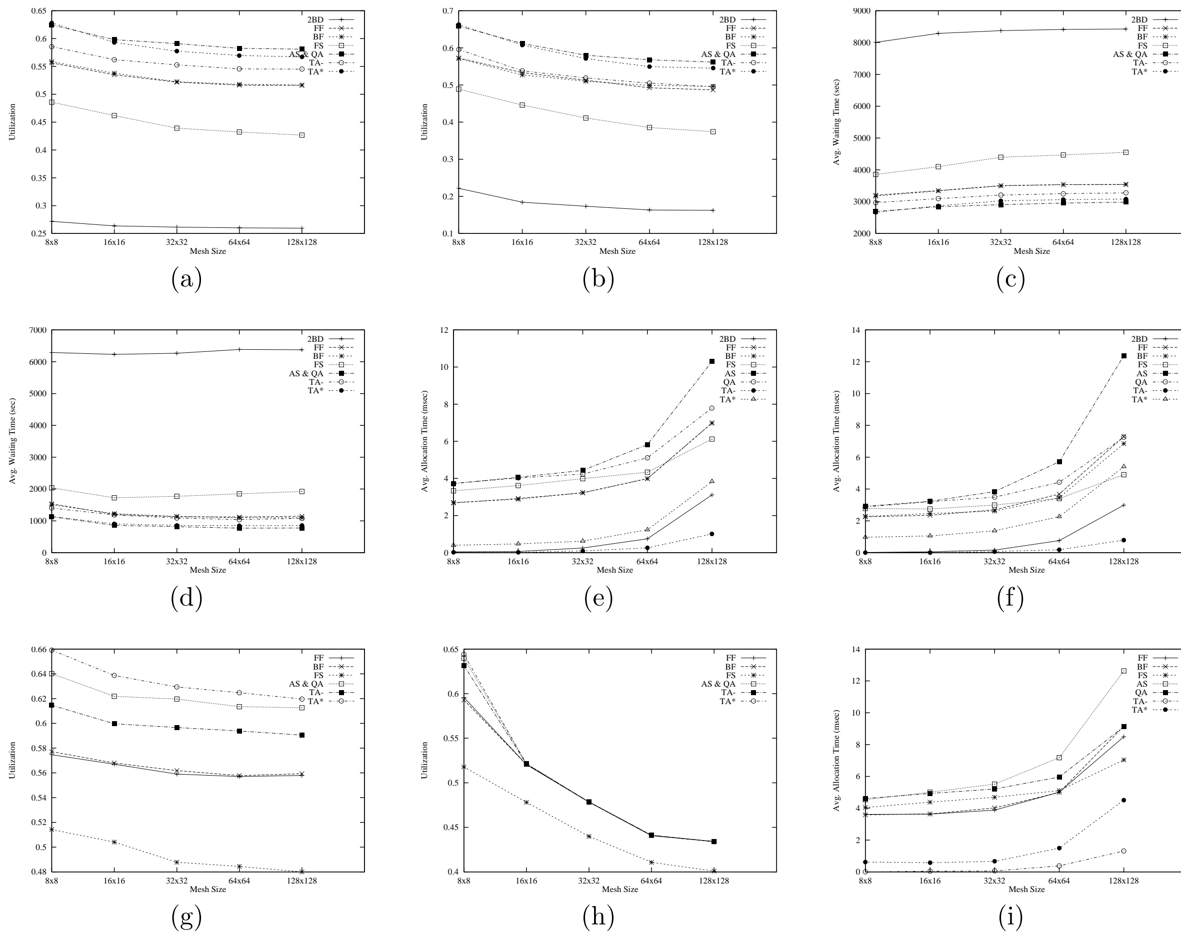


Figure 2: Simulation results. (a) uniform dist./utilization, (b) exponential dist./utilization, (c) uniform dist./avg. waiting time, (d) exponential dist./avg. waiting time, (e) uniform dist./avg. allocation time, (f) exponential dist./avg. allocation Time, (g) uniform dist./utilization, (h) exponential dist./utilization, (i) uniform dist./avg. allocation time.

[4] G. Kim and H. Yoon, "On Submesh Allocation for mesh Multicomputers: A Best-Fit Allocation and a Virtual Submesh Allocation for Faulty Meshes," IEEE Trans. Parallel and Distributed Systems, vol. 9, no. 2, pp. 175-185, Feb. 1998.

[5] K. Li and K. H. Cheng, "A Two-Dimensional Buddy System for Dynamic Resource Allocation in a Partitionable Mesh Connected System," J. Parallel and Distributed Computing, vol. 12, pp. 79-83, May 1991.

[6] V. Lo and K. J. Windisch and W. Liu and B. Nitzberg, "Noncontiguous Processor Allocation Algorithms for Mesh-Connected Multicomputers," IEEE Trans. Parallel and Distributed Systems, vol. 8, no. 7, pp. 712-725, Jul. 1997.

[7] D. Min and M. W. Mutka, "Efficient Job Scheduling in a Mesh Multicomputer Without Discrimination Against Large Jobs", Proc. Symp. Parallel and Distributed Processing, pp. 52-59. Oct. 1995.

[8] D. D. Sharma and D. K. Pradhan, "A Fast and Efficient Strategy for Submesh Allocation in Mesh-Connected Parallel Computers," Proc. Symp. Parallel and Distributed Processing, pp. 682-689, Dec. 1993.

[9] D. D. Sharma and D. K. Pradhan, "Job Scheduling in Mesh Multicomputers," IEEE Trans. Parallel and Distributed Systems, vol. 9, no. 1, pp. 57-70, Jan. 1998.

[10] S. M. Yoo and H. Y. Youn, "Largest-Job-First-Scan-All Scheduling Policy for 2D Mesh-Connected Systems", Proc. Symp. Frontiers of Massively Parallel Computation, pp. 118-125, Oct. 1996.

[11] S. M. Yoo and H. Y. Youn and B. Shirazi, "An Efficient Task Allocation Scheme for 2D Mesh Architectures," IEEE Trans. Parallel and Distributed Systems, vol. 8, no. 9, pp. 934-942, Sep. 1997.

[12] Y. Zhu, "Efficient Processor Allocation Strategies for Mesh-Connected Parallel Computers," J. Parallel and Distributed Computing, vol. 16, pp. 328-337, Dec. 1992.