論文2003-40CI-6-2

메쉬 구조 다중컴퓨터 시스템을 위한 효율적인 서브메쉬 할당방법

(An Efficient Submesh Allocation Scheme for Mesh-Connected Multicomputer Systems)

李 元 柱 *, 全 昌 浩 **

(Won Joo Lee and Chang Ho Jeon)

요 약

본 논문에서는 메쉬 구조 다중컴퓨터 시스템의 성능을 향상시킬 수 있는 새로운 서브메쉬 할당방법을 제안한다. 이 할당방법의 특징은 가용 서브메쉬의 탐색시간과 외적단편화로 인한 서브메쉬의 할당지연을 최소화함으로써 태스크의 대기시간을 줄이는 것이다. 이 할당방법은 가용 서브메쉬를 유형(정방형, 가로 직사각형, 세로 직사각형)에 따라 분류하고, 태스크와 동일한 유형별 가용 서브메쉬 리스트에서 최적의 서브메쉬를 찾아 할당함으로써 서브메쉬 탐색시간을 줄인다. 또한 외적단편화로 인해 서브메쉬의 할당지연이 발생하면 할당 서브메쉬에서 수행중인 태스크를 다른 가용 서브메쉬에 재배치하고, 프로세서 단편을 통합하여 할당함으로써 서브메쉬의 할당지연을 최소화한다. 시뮬레이션을 통하여 서브메쉬 탐색시간을 줄이는 방법보다 외적단편화로 인한 서브메쉬의 할당지연을 줄이는 방법이 태스크의 대기시간을 단축하는데 더 효과적임을 보인다. 그리고 제안한 할당방법이 시스템의 성능 향상 면에서 기존의 할당방법들보다 우수함을 보인다.

Abstract

In this paper, we propose a new submesh allocation scheme which improves the performance of multicomputer systems. The key idea of this allocation scheme is to reduce waiting time of task by minimizing the submesh search time and the submesh allocation delay caused by external fragmentation. This scheme reduces the submesh search time by classifying independent free submeshes according to their types (square, horizontal rectangle, vertical rectangle) and searching a best-fit submesh from the classified free submesh list. If a submesh allocation delay occurs due to external fragmentation, the proposed scheme relocates tasks, executing in allocated submeshes, to another free submeshes and compacts processor fragmentation. This results in reducing the submesh allocation delay. Through simulation, we show that it is more effective to reduce the submesh allocation delay due to external fragmentation than reducing the submesh search time with respect to the waiting time of task. We also show that the proposed strategy improves the performance compared to previous strategies.

Keywords : 메쉬(Mesh), 독립 가용 서브메쉬(IFS: Independent Free Submesh), 유형별 가용 서브메쉬 리스트(CFSL: Classified Free Submesh List), 확장지수(Expansion index), 태스크 재배치(Task Relocation)

* 正會員, 斗源工科大學 인터넷프로그래밍과

(Department of Internet Programming, Doowon Technical College)

** 正會員, 漢陽大學校 電子컴퓨터工學部

(School of Electrical and Computer Engineering, Hanyang University)

接受日字:2003年6月30日, 수정완료일:2003年10月20日

I. 서 론

메쉬 구조는 단순하고, 규칙적이며 확장성이 뛰어나기 때문에 Intel Touchstone Delta[1]와 Intel Paragon XP/S[2], Tera Computer System[3] 등과 같은 상업용 또는 프로토타입의 다중컴퓨터 시스템에 널리 사용되고 있다. 이러한 시스템의 성능은 프로세서 활용도, 메모리 사용량, 상호연결망의 대역폭 등과 같은 여러 요인에 따라 달라진다[4]. 프로세서 활용도를 높이는 하나의 방법은 효율적인 태스크 스케줄링과 프로세서 할당을 통하여 태스크 대기시간과 외적단편화를 줄이는 것이다. 특히 프로세서 할당방법에서는 전체 메쉬 구조에서 태스크에 최적인 가용 서브메쉬를 탐색하여 할당하는 서브메쉬 탐색시간과 외적단편화로 인한 할당지연을 최소화함으로써 태스크 대기시간을 줄일 수 있다.

기존의 서브메쉬 할당방법들은 가용 서브메쉬 탐색시간과 외적단편화를 최소화하여 시스템의 성능을 향상시키는 연구를 진행해 왔으며, 그 결과 여러 서브메쉬 할당방법들이 제안되었다[5-13]. 기존의 할당방법들은 서브메쉬를 할당하는 방법에 따라 최초할당과 최적할당으로 분류된다. 최초 할당방법[6,9]은 태스크에 할당 가능한 가용 서브메쉬를 찾으면 바로 할당하는 방법으로 할당 알고리즘이 단순하고 서브메쉬 탐색시간이 짧다. 하지만 최적할당에 비해 외적단편화가 심하다. 최적할당방법[5,8-13]은 가용 서브메쉬 리스트에서 최적의 서브메쉬를 찾아 할당하는 방법으로 최초할당에 비해 외적단편화가 적고 시스템의 성능 향상에 유리하다. 그러나 이 할당방법은 할당 알고리즘이 복잡하고 가용 서브메쉬 리스트내의 서브메쉬 수가 증가하면 서브메쉬 탐색시간이 길어지는 문제점이 있다. 기존의 최적 할당방법들은 최초 할당방법에 비해 외적단편화를 많이 줄였지만 2차원 메쉬 구조의 구조적 한계로 인한 외적단편화는 줄이지 못하였다. 2차원 메쉬 구조에서는 할당 서브메쉬를 중심으로 상하, 좌우로 양분된 프로세서 단편들을 연결하여 더 큰 가용 서브메쉬를 형성할 수 없기 때문에 서브메쉬의 할당지연이 발생한다. 이러한 서브메쉬의 할당지연은 태스크 대기시간을 증가시키기 때문에 시스템의 성능을 저하시키는 요인이 된다.

본 논문에서는 메쉬 구조 다중컴퓨터 시스템의 성능을 향상시킬 수 있는 새로운 서브메쉬 할당방법을 제안한다. 이 할당방법의 특징은 가용 서브메쉬의 탐색시간과 외적단편화로 인한 서브메쉬의 할당지연을 최소화하여 태스크 대기시간을 줄이는 것이다. 이 할당방법은 먼저 메쉬 구조에서 탐색한 가용 서브메쉬를 유형별로 분류하여 유형별 가용 서브메쉬 리스트를 생성한다. 그리고 태스크 유형과 동일한 유형별 가용 서브메쉬 리스트에서 최적의 서브메쉬를 찾아 할당함으로써 서브메쉬 탐색시간을 줄인다. 하지만 서브메쉬의 할당과 할당 해제를 반복하는 과정에서 외적단편화로 인해 서브메쉬의 할당지연이 증가하면 태스크 대기시간도 함께 증가하는 문제점이 있다. 이러한 문제점을 해결하기 위해 할당 서브메쉬에서 수행중인 태스크들을 다른 가용 서브메쉬에 재배치하고 프로세서 단편들을 통합하여 할당한다. 본 논문에서는 시뮬레이션을 통하여 제안한 할당방법이 시스템의 성능 향상 면에서 기존의 할당방법들보다 우수함을 보인다. 또한 태스크의 대기시간을 단축하는 방법으로는 서브메쉬 탐색시간을 줄이는 방법보다 외적단편화로 인한 서브메쉬의 할당지연을 줄이는 방법이 더 효과적임을 보인다.

본 논문의 II절에서는 메쉬 구조에 대하여 설명하고, III절에서는 기존의 서브메쉬 할당방법에 대하여 설명한다. IV절에서는 메쉬 구조를 위한 새로운 서브메쉬 할당방법을 제안한다. V절에서는 시뮬레이션 환경을 설명하고 성능 평가를 통하여 본 논문에서 제안한 서브메쉬 할당방법이 기존의 할당방법에 비해 우수함을 보인다. 그리고 VI절에서 결론을 맺는다.

II. 2차원 메쉬 구조

2차원 메쉬 M(W, H)는 너비와 높이가 W와 H인 사각형 격자 구조이다. M(W, H)는 W x H개의 노드로 구성되며 각 노드는 하나의 프로세서를 나타낸다. 본 논문에서는 각 노드의 주소를 <x, y> 형식으로 표현한다. 노드 <x, y>는 좌측하단의 기본 노드 <0, 0>를 기준으로 너비는 x만큼, 높이는 y만큼 떨어진 좌표에 위치한 노드를 가리킨다. 따라서 x, y값은 $0 \leq x \leq W-1$와 $0 \leq y \leq H-1$ 조건을 만족한다. 태스크를 할당 받아 실행중인 노드를 할당 노드라고 하며 유휴 상태의 노드를 가용 노드라 한다. <그림 1>에서 할당 노드는 검정색으로, 가용 노드는 흰색으로 표현되어 있다. M(W, H) 내의 서브메쉬는 *크기* 또는 주소를 이용하여 표현할 수 있다. 사각형 격자 구조로 w x h개의 노드로 구성된 서브메쉬는 크기를 이용하여 S(w, h)로 표현한다.

w와 h는 너비와 높이를 의미하며 $1 \le w \le W$와 $1 \le h \le H$ 조건을 만족한다. 또한 좌측하단 노드 주소 $<x_1, y_1>$와 우측상단의 노드 주소 $<x_2, y_2>$를 갖는 서브메쉬는 $S(<x_1, y_1>, <x_2, y_2>)$로 표현한다. 할당 서브메쉬는 할당 노드들로 구성한 서브메쉬이고 가용 서브메쉬는 가용 노드들로 구성한 서브메쉬이다. 가용 서브메쉬와 할당 서브메쉬는 각각 $S_i(w, h)$와 $A_j(w, h)$ 또는 $S_i(<x_1, y_1>, <x_2, y_2>)$와 $A_j(<x_1, y_1>, <x_2, y_2>)$로 표현한다.

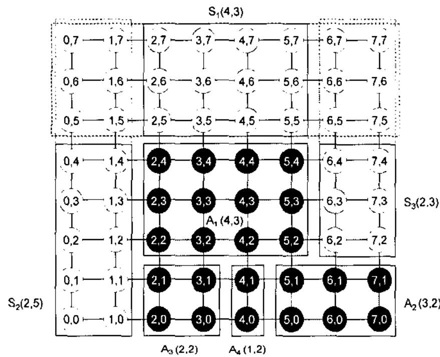


그림 1. 2차원 메쉬 M(8, 8)
Fig. 1. Two-dimensional Mesh M(8, 8).

<그림 1>에서 할당 서브메쉬는 $A_1(<2, 2>, <5, 4>)$, $A_2(<5, 0>, <7, 1>)$, $A_3(<2, 0>, <3, 1>)$, $A_4(<4, 0>, <4, 1>)$이고 가용 서브메쉬는 $S_1(<2, 5>, <5, 7>)$, $S_2(<0, 0>, <1, 4>)$, $S_3(<6, 2>, <7, 4>)$ 이다. 가용 서브메쉬와 할당 서브메쉬는 각각 크기와 할당 순서에 따라 내림차순으로 정렬한다.

Ⅲ. 기존의 서브메쉬 할당방법

메쉬 구조를 위한 기존의 서브메쉬 할당방법들은 서브메쉬를 할당하는 방법에 따라 최초할당과 최적할당으로 분류 할 수 있다.

1. 최초 할당방법

최초 할당방법은 전체 메쉬 구조에서 할당 가능한 가용 서브메쉬를 찾으면 바로 할당하는 방법이다.

FS(Frame Sliding) 할당방법[6]은 정방형이 아닌 메쉬 구조와 태스크의 크기가 2^n이 아닌 경우에도 적용할 수 있도록 기존의 2DB 할당방법[5]을 개선한 할당방법이다. 이 할당방법은 태스크 크기에 알맞은 가용 서브메쉬를 찾아 할당함으로써 내적단편화 문제를 해결하였지만, 가용 서브메쉬를 완벽하게 인식하지 못하는 단점이 있다. AS(Adaptive Scan) 할당방법[7]은 태스크 T(w, h)의 방향을 T(h, w)로 회전하여 할당 할 수 있도록 FS 할당방법을 개선함으로써 외적단편화를 줄인 할당방법이다.

2. 최적 할당방법

최적 할당방법은 먼저 전체 메쉬 구조에서 가용 서브메쉬를 탐색하여 가용 서브메쉬 리스트를 생성한다. 그리고 가용 서브메쉬 리스트에서 태스크에 최적인 가용 서브메쉬를 찾아 할당하는 방법이다.

2DB(Dimensional Buddy) 할당방법[5]은 다양한 크기를 가진 태스크에 2^n 크기의 정방형 가용 서브메쉬를 할당하는 방법이다. 이 할당방법은 태스크가 요청한 프로세서 수보다 많은 프로세서가 할당되기 때문에 내적단편화가 심하다는 단점이 있다. FF(First-Fit)/BF(Best-Fit) 할당방법[8]은 내적단편화 문제를 해결하기 위해 태스크 크기와 일치하는 가용 서브메쉬를 찾아 할당하는 방법이다. 이 할당방법은 할당 배열과 범위 배열의 관리로 인한 오버헤드가 크며, 또한 주어진 태스크의 방향을 고정하여 탐색하기 때문에 서브메쉬 인식률이 낮다는 단점이 있다. ADJ(ADJacency) 할당방법[9]은 할당 서브메쉬의 4개 모서리에 인접한 가용 서브메쉬만을 대상으로 경계값을 구한 후, 최대 경계값을 갖는 서브메쉬를 할당하는 방법이다. 이 할당방법은 메쉬 크기에 관계없이 할당시간이 일정하고 외적단편화를 줄일 수 있는 장점이 있다. LKH(Look ahead) 할당방법[10]은 주어진 태스크에 가용 서브메쉬를 할당할 때, 큐에 대기 중인 최대 크기의 태스크에 할당 가능한 다른 가용 서브메쉬가 존재하는지 여부를 예측하여 할당하는 방법이다. FL(Free List) 할당방법[11]은 가용 서브메쉬 리스트에서 주어진 태스크의 크기보다 크거나 같은 가용 서브메쉬를 찾아 할당하는 방법이다. 이 할당방법은 가용 서브메쉬 리스트내의 가용 서브메쉬 수가 증가하면 서브메쉬 탐색시간이 길어지는 단점이 있다. QA(Quick Allocation) 할당방법[12]은 메쉬의 각 행 상태를 관리하는 배열을 사용하여 전체 메쉬 구조를 검색하지 않고 가용 서브메쉬를 찾음으로써 서브메쉬 탐색시간을 줄이는 할당방법이다. 이 할당방법은 AS 할당방법을 개선하여 가용 서브메쉬의 인식률을 높이고, 할당 알고리즘의 시간복잡도를 낮추었지만 메쉬 크기에 따라 할당방법의 성능이 달라진다. FSL(Free

Submesh List) 할당방법[13]은 가용 서브메쉬 리스트(FSL)에서 태스크에 최적인 서브메쉬를 찾아 할당하는 방법이다. 이 할당방법은 먼저 FSL에서 태스크에 할당 가능한 서브메쉬들을 후보 가용 서브메쉬로 선정한다. 그리고 각 후보 가용 서브메쉬에 대한 예약지수를 구하여 그 값이 최대인 서브메쉬를 할당한다. 이 할당방법은 가용 서브메쉬의 인식률은 높지만 후보 가용 서브메쉬 수가 증가하면 예약지수를 구하는 횟수도 함께 증가하기 때문에 할당시간이 길어진다. 또한, 가용 서브메쉬는 서로 중첩하기 때문에 서브메쉬를 할당할 때마다 다른 가용 서브메쉬의 크기에 영향을 미친다. 따라서 서브메쉬의 할당 및 할당 해제 후에는 반드시 FSL을 재생성하며 그에 따른 오버헤드로 인해 할당시간이 길어진다.

위에서 설명한 기존의 최적 할당방법들도 2차원 메쉬 구조의 구조적 한계로 인한 외적단편화는 줄이지 못하였다. 2차원 메쉬 구조에서는 할당 서브메쉬를 중심으로 상하, 좌우로 양분된 프로세서 단편들을 연결하여 더 큰 가용 서브메쉬를 형성할 수 없기 때문에 서브메쉬의 할당지연이 발생한다. 즉, <그림 1>에서 S_2(<0, 0>, <1, 4>)와 S_3(<6, 2>, <7, 4>)는 할당 서브메쉬에 의해 좌우로 양분되었기 때문에 더 큰 가용 서브메쉬를 형성하기 위해 연결 할 수 없다. 이때 태스크 T(4, 6)이 주어진다면 할당 가능한 가용 서브메쉬를 찾을 수 없으므로 서브메쉬의 할당은 지연된다. 이러한 서브메쉬의 할당지연을 최소화하기 위해 본 논문에서는 프로세서 단편들을 통합하여 할당하는 새로운 서브메쉬 할당방법을 제안한다.

Ⅳ. 제안한 서브메쉬 할당방법

1. 용어 정의

본 논문에서 제안하는 서브메쉬 할당방법을 설명하기에 앞서 기본적인 용어들을 정의한다.

정의 1. 독립 가용 서브메쉬(IFS: Independent Free Submesh)는 다른 가용 서브메쉬와 중첩되지 않게 형성 할 수 있는 최대 크기의 가용 서브메쉬이다.

<그림 1>에서 실선으로 구분한 S_1(<2, 5>, <5, 7>), S_2(<0, 0>, <1, 4>), S_3(<6, 2>, <7, 4>)는 IFS이다. 서브메쉬 S(<0, 0>, <1, 7>), S(<0, 5>, <7, 7>), S(<6, 2>, <7, 7>)은 점선으로 구분한 서브메쉬 S(<0, 5>, <1, 7>), S(<6, 5>, <7, 7>)에서 서로 중첩된다. 이때 서로 중첩되는 서브메쉬 S(<0, 5>, <1, 7>)과 S(<6, 5>, <7, 7>)을 제외하고 S_1(<2, 5>, <5, 7>), S_2(<0, 0>, <1, 4>), S_3(<6, 2>, <7, 4>)를 IFS로 지정한다. IFS들은 서로 중첩되지 않기 때문에 특정 IFS를 태스크에 할당하더라도 다른 IFS의 크기에 영향을 주지 않는다. 따라서 IFS를 태스크에 할당하는 경우에는 유형별 가용 서브메쉬 리스트를 재생성할 필요가 없다. 유형별 가용 서브메쉬 리스트는 다음과 같이 정의한다.

정의 2. 유형별 가용 서브메쉬 리스트(CFSL: Classified Free Submesh List)는 IFS의 너비와 높이에 따라 정방형, 가로 직사각형, 세로 직사각형으로 분류한 IFS들의 집합이다.

본 논문에서는 정방형, 가로 직사각형, 세로 직사각형 IFS의 집합을 각각 SQ-flist, HR-flist, VR-flist로 표현한다. 또한 할당 서브메쉬의 집합은 Alloc-list로 표현한다. CFSL내의 IFS는 크기에 따라 내림차순으로 정렬되며, SQ-flist ∩ HR-flist ∩ VR-flist = ∅ 조건을 만족한다. <그림 1>에서 S_1(4, 3)은 가로 직사각형으로 분류되어 HR-flist에 삽입된다. 그리고 S_2(2, 5)와 S_3(2, 3)은 세로 직사각형으로 분류되어 VR-flist에 삽입된다. 또한 Alloc-list는 A_1(<2, 2>, <5, 4>), A_2(<5, 0>, <7, 1>), A_3(<2, 0>, <3, 1>), A_4(<4, 0>, <4, 1>)를 포함한다.

주어진 태스크는 T(w, h)로 표현한다. w와 h는 각각 태스크의 너비와 높이를 의미한다. T(w, h)를 처리하기 위해서는 최소 w x h개의 노드를 가진 가용 서브메쉬가 필요하다. 본 논문에서는 T(w, h)를 w와 h에 따라 정방형, 가로 직사각형, 세로 직사각형으로 분류하여 서브메쉬 탐색과정에서 활용한다. 즉, 태스크의 유형이 가로 직사각형일 경우, 태스크 유형과 동일한 HR-flist에서 먼저 최적의 서브메쉬를 찾는다. 전체 서브메쉬를 대상으로 최적의 서브메쉬를 찾는 것보다 태스크의 유형과 동일한 CFSL에서 먼저 찾음으로써 서브메쉬의 탐색시간을 줄인다.

정의 3. IFS의 확장지수(EI: Expansion Index)는 다른 가용 서브메쉬와 중첩을 허용하면서 IFS를 최대 크기로 확장할 수 있는 변위값이다.

$S_i(<x_1, y_1>, <x_2, y_2>)$의 확장지수는 $EI(S_i)=(<\alpha, \beta>, <\alpha', \beta'>)$ 형식으로 표현한다. 확장지수는 전체 메쉬 구조를 탐색하여 IFS를 형성하는 과정에서 구하며, 각 IFS의 속성으로 저장된다. $EI(S_i)=(<\alpha, \beta>, <\alpha', \beta'>)$는 $S_i(<x_1, y_1>, <x_2, y_2>)$와 $S_i'(<x_1', y_1'>, <x_2', y_2'>)$의 좌측하단 노드와 우측상단 노드의 좌표값으로 구한다. 즉, $<\alpha, \beta>$는 각각 $<x_1'-x_1, y_1'-y_1>$이며 $<\alpha', \beta'>$는 각각 $<x_2'-x_2, y_2'-y_2>$로 구한다. 여기서 $S_i'(<x_1', y_1'>, <x_2', y_2'>)$는 $S_i(<x_1, y_1>, <x_2, y_2>)$를 다른 가용 서브메쉬와 중첩을 허용하면서 형성한 최대 크기의 가용 서브메쉬이다.

예를 들어 <그림 1>에서 IFS인 $S_1(<2, 5>, <5, 7>)$은 다른 가용 서브메쉬와 중첩하는 서브메쉬 $S(<0, 5>, <1, 7>)$과 $S(<6, 5>, <7, 7>)$를 포함하여 $S_1'(<0, 5>, <7, 7>)$를 형성할 수 있다. 확장지수는 $S_1(<2, 5>, <5, 7>)$과 $S_1'(<0, 5>, <7, 7>)$의 좌측하단 노드와 우측상단 노드의 좌표값을 이용하여 $<\alpha, \beta> = <0-2, 5-5>$와 $<\alpha', \beta'> = <7-5, 7-7>$로 구한다. 따라서 $EI[S_1(<2, 5>, <5, 7>)] = (<-2, 0>, <2, 0>)$ 이다.

2. 프로세서 단편 통합(Compaction)

메쉬 구조에서는 외적단편화로 인해 할당지연이 발생하면 태스크는 할당 서브메쉬들이 할당 해제되어 더 큰 가용 서브메쉬를 형성할 때까지 대기해야 하기 때문에 태스크 대기시간이 증가한다. 하지만 본 논문에서는 할당 서브메쉬에서 수행중인 태스크를 다른 가용 서브메쉬로 재배치한 후 프로세서 단편들을 통합하여 할당함으로써 태스크 대기시간을 단축한다. 태스크를 재배치하기 위해 메쉬 M(H, W)의 각 노드는 서로 다른 링크를 통하여 하나 이상의 프로세스를 동시에 양방향으로 전송 할 수 있다고 가정한다. 즉, 노드$<x, y>$는 노드$<x-1, y>$ 또는 노드$<x, y-1>$로 프로세스 P_1을 보낼 때 노드$<x+1, y>$ 또는 노드$<x, y+1>$로부터 프로세스 P_2를 받을 수 있다. 또한, 재배치 대상의 할당 서브메쉬가 하나 이상일 경우에는 각 할당 서브메쉬의 좌측하단 노드 좌표값이 최소인 할당 서브메쉬를 먼저 재배치한다. 본 논문에서는 외적단편화를 줄이기 위해 $A_i(<x_1, y_1>, <x_2, y_2>)$에서 실행중인 태스크를 x축의 좌측 방향과 y축의 하단 방향으로 이동하여 기본 노드 $<0, 0>$에 근접하도록 재배치한다.

태스크 재배치과정에서는 $A_i(<x_1, y_1>, <x_2, y_2>)$와 기본 노드 $<0, 0>$ 사이의 영역을 <그림 2>와 같이 세

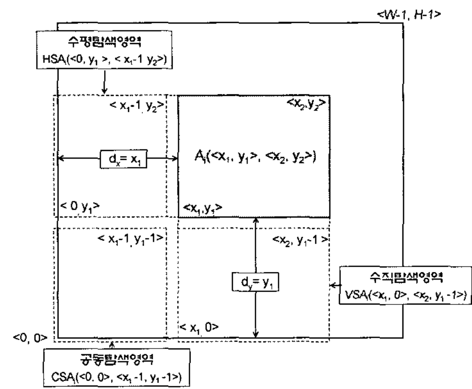


그림 2. 할당 서브메쉬의 탐색영역
Fig. 2. Search area of allocated submesh.

영역으로 분할하여 각 영역별로 다른 할당 서브메쉬가 존재하는지 탐색한다. 각 탐색영역은 다음과 같이 정의한다.

정의 4. 탐색영역(SA : Search Area)은 기본 노드 $\langle 0, 0\rangle$와 $A_i(\langle x_1, y_1\rangle, \langle x_2, y_2\rangle)$ 사이에 위치한 서브메쉬이다.

탐색영역은 공동(Common), 수평(Horizontal), 수직(Vertical) 탐색영역으로 구분하며, 각각 $CSA_i(<0, 0>, <x_1-1, y_1-1>)$, $HSA_i(<0, y_1>, <x_1-1, y_2>)$, $VSA_i(<x_1, 0>, <x_2, y_1-1>)$로 표현한다. 각 탐색영역에 다른 할당 서브메쉬가 존재하지 않으면 $A_i(<x_1, y_1>, <x_2, y_2>)$에서 실행중인 태스크는 x축의 좌측 방향과 y축의 하단 방향으로 각각 x_1과 y_1만큼 이동하여 $S(<0, 0>, <x_2-x_1, y_2-y_1>)$에 재배치된다. 그러나 다른 할당 서브메쉬들이 존재하면 각 탐색영역내의 할당 노드들을 탐색하여 할당범위를 생성한다. 할당범위는 다음과 같이 정의한다.

정의 5. 할당범위(AD: Allocation Domain)는 각 탐색영역에 위치한 할당 노드들로 구성한 서브메쉬이다.

할당범위는 공동(Common), 수평(Horizontal), 수직(Vertical) 할당범위로 구분하며 각각 $CAD_i(<x_{c1}, y_{c1}>, <x_{c2}, y_{c2}>)$, $HAD_i(<x_{h1}, y_{h1}>, <x_{h2}, y_{h2}>)$, $VAD_i(<x_{v1}, y_{v1}>, <x_{v2}, y_{v2}>)$로 표현한다. 할당범위는 이미 다른 태스크를 실행중인 할당 노드들로 구성하기 때문에 태스크를 재배치 할 수 없는 영역이다. 만약 $HSA_i(<0,$

$y_1>$, $<x_1-1, y_2>$)에 하나 이상의 할당 서브메쉬가 존재한다면 $HSA_i(<0, y_1>, <x_1-1, y_2>)$내의 각 할당 노드들을 탐색하여 $HAD_i(<x_{h1}, y_{h1}>, <x_{h2}, y_{h2}>)=(<min(x_i), min(y_i)>, <max(x_i), max(y_i)>)$를 생성한다. 이때 x_i와 y_i는 각각 $0 \le x_i \le x_1-1$, $y_1 \le y_i \le y_2$ 조건을 만족한다. 그리고 $CSA_i(<0, 0>, <x_1-1, y_1-1>)$와 $VSA_i(<x_1, 0>, <x_2, y_1-1>)$도 $HSA_i(<0, y_1>, <x_1-1, y_2>)$와 같이 각각 $CAD_i(<x_{c1}, y_{c1}>, <x_{c2}, y_{c2}>)$와 $VAD_i(<x_{v1}, y_{v1}>, <x_{v2}, y_{v2}>)$를 생성한다. 할당범위를 생성한 후에는 태스크를 재배치하기 위한 이동거리를 구한다. 태스크의 이동거리는 다음과 같이 정의한다.

정의 6. 할당 서브메쉬 $A_i(\langle x_1, y_1\rangle, \langle x_2, y_2\rangle)$에서 수행중인 태스크의 이동거리(D)는 수평이동거리(horizontal migration distance) d_x와 수직이동거리(vertical migration distance) d_y의 합이다.

d_x와 d_y는 $A_i(<x_1, y_1>, <x_2, y_2>)$에서 수행중인 태스크를 각각 x축과 y축 방향으로 이동할 수 있는 거리이며, 각각 $0 \le d_x \le x_1$, $0 \le d_y \le y_1$ 조건을 만족한다. <그림 3>과 같이 d_x와 d_y는 $A_i(<x_1, y_1>, <x_2, y_2>)$와 각 할당범위의 위치에 따라 달라진다.

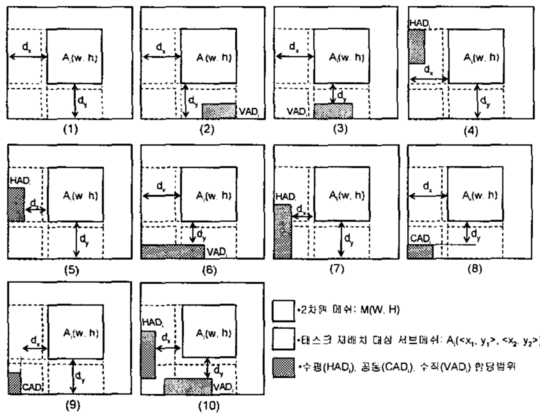


그림 3. 할당 서브메쉬의 재배치 유형
Fig. 3. Relocation type of allocated submesh.

<그림 3>의 유형 (1)에서는 각 탐색영역에 할당 서브메쉬가 존재하지 않기 때문에 $d_x=x_1$, $d_y=y_1$ 이다. 그러나 유형 (2)~(10)에서는 각 할당범위의 위치에 따라 d_x와 d_y가 달라진다. <그림 3>의 할당 서브메쉬 재배치 유형에 따른 d_x와 d_y는 <표 1>과 같다. <표 1>의 CSA, HSA, VSA는 각 탐색영역에서 각각 할당범위를 생성하면 T(True) 이고 그 반대의 경우에는 F(False) 이다. <표 1>에 따라 d_x와 d_y가 결정되면 $A_i(<x_1, y_1>, <x_2, y_2>)$에서 실행중인 태스크는 다른 위치의 $S(<x_1-d_x, y_1-d_y>, <x_2-d_x, y_2-d_y>)$로 재배치된다. 이때 $A_i(<x_1, y_1>, <x_2, y_2>)$의 모든 노드들은 교착상태(deadlock)를 방지하기 위해 이동거리($D=d_x+d_y$)만큼 같은 방향으로 함께 이동한다.

표 1. $A_i(<x_1, y_1>, <x_2, y_2>)$의 수평이동거리와 수직이동거리
Table 1. Horizontal and vertical migration distance of $A_i(<x_1, y_1>, <x_2, y_2>)$.

Type	CSA	HSA	VSA	Condition	d_x	d_y
(1)	F	F	F	-	x_1	y_1
(2)	F	F	T	$(x_1<x_{v1})$ AND $(w \le x_{v1})$	x_1	y_1
(3)	F	F	T	$(x_1<x_{v1})$ AND $(w>x_{v1})$	x_1	y_1-y_{v2}
(4)	F	T	F	$(y_1<y_{h1})$ AND $(h \le y_{h1})$	x_1	y_1
(5)	F	T	F	$(y_1<y_{h1})$ AND $(h>y_{h1})$	x_1-x_{h2}	y_1
(6)		F	T	$(x_1<x_{v1})$ AND $(w>x_{v1})$	x_1	y_1-y_{v2}
(7)		T	F	$(y_1<y_{h1})$ AND $(h>y_{h1})$	x_1-x_{h2}	y_1
(8)	T	F	F	$(x_1>x_{c2})$ AND $(y_1>y_{c2})$ AND $(w_c>h_c)$	x_1	y_1-y_{v2}
(9)		F	F	$(x_1>x_{c2})$ AND $(y_1>y_{c2})$ AND $(w_c<h_c)$	x_1-x_{c2}	y_1
(10)		T	T	$((x_1>x_{h2})$ AND $(y_1 \le y_{h2}))$ AND $((x_1 x_{v2})$ AND $(y_1>y_{v2}))$	x_1-x_{h2}	y_1-y_{v2}

태스크를 재배치하는데 소요되는 시간은 다음과 같이 정의한다.

정의 7. 태스크 $T_i(w, h)$의 재배치 시간(RT_i: Relocation Time)은 $A_i(\langle x_1, y_1\rangle, \langle x_2, y_2\rangle)$에서 수행중인 태스크를 이동거리($D=d_x+d_y$)만큼 이동하여 $S(\langle x_1-d_x, y_1-d_y\rangle, \langle x_2-d_x, y_2-d_y\rangle)$로 재배치하는데 소요되는 시간이다.

$T_i(w, h)$의 재배치 시간은 메시지 전송 준비, 태스크의 수행중지/재실행, 동기화, 프로세스 이동, 문맥 교환(context switch)에 소요되는 시간으로 구성된다. 태스크의 수행중지/재실행 과정에서 태스크의 상태 정보와 레지스터 정보를 저장하는 시간과 동기화, 문맥 교환에 소요되는 시간은 실제 프로세스 이동시간에 비해 그 값이 매우 작기 때문에 무시된다[14-16]. 본 논문에서는 $T_i(w, h)$의 재배치시간을 식 (1)과 같이 구한다.

$$RT_i = D \cdot \tau \qquad (1)$$

식 (1)에서 D는 태스크의 이동거리이고 τ는 태스크 단위이동시간으로 태스크를 이동거리 1만큼 이동하는데 소요되는 시간이다. 할당 서브메쉬의 각 할당 노드에서 수행중인 프로세스를 인접한 유휴 노드로 이동할 때 소요되는 프로세스 이동시간들 중에 최대값을 τ로 지정한다. 각 할당 노드의 프로세스 이동시간은 패킷 길이 또는 플릿(flit), 네트워크 대역폭 등에 의해 결정되며, 그 값은 라우팅 방법에 따라 달라진다[4].

예를 들어, 최대 단방향 네트워크 대역폭이175MB/s인 Intel Paragon XP/S[2] 시스템에서 175 바이트의 데이터를 가진 프로세스를 이동하기 위한 시간은 1 μs이다. 따라서 이 시스템의 각 노드들이 동일한 크기의 데이터를 가진 프로세스를 수행한다고 가정한다면, <그림 1>의 A_1(<2, 2>, <5, 4>)을 이동거리 2만큼 이동시켜 S(<0, 2>, <3, 4>)에 재배치하는데 소요되는 재배치시간(RT_1)은 2 μs 이다. 본 논문의 시뮬레이션에서는 태스크 재배치시간을 평균대기시간에 합산하여 할당방법의 성능을 평가할 때 고려한다.

3. CFSL-TR 할당 방법

본 논문에서는 제안하는 서브메쉬 할당방법을 CFSL-TR(Classified Free Submesh List-Task Relocation) 할당방법이라 한다. 이 할당방법의 서브메쉬 할당 알고리즘은 <그림 4>와 같다. <그림 4>의 Make_CFSL() 모듈에서는 전체 메쉬 구조에서 탐색한 IFS를 유형에 따라 분류하여 CFSL을 생성한다. 그리고 Select_Submesh() 모듈에서는 태스크의 유형과 동일한 CFSL에서 최적의 가용 서브메쉬를 찾는다. 최적의 가용 서브메쉬가 존재하면 Do_Allocate() 모듈에서는 그 가용 서브메쉬를 태스크에 할당하고 CFSL에서 삭제한다. 그러나 최적의 가용 서브메쉬를 찾지 못하면 외적단편화 여부를 판별한다. 만약 외적단편화로 인해 가용 서브메쉬를 찾지 못한 경우에는 Processor_Fragmentation_Compaction() 모듈에서 수행중인 태스크를 재배치하고, 프로세서 단편들을 통합하여 할당한다. 하지만 외적단편화가 아닌 경우에는 할당 서브메쉬들이 할당 해제되어 더 큰 가용 서브메쉬를 형성할 때까지 대기한다. 본 논문에서는 세 가지 서브메쉬 할당을 예로 들어 <그림 4>의 서브메쉬 할당 알고리즘을 상세히 설명한다.

```
CFSL-TR_Allocation() //Request(Ti<w, h>)
{
  Ti=Dispatch_Qi(Ti<w, h>); //대기 큐에서 태스크 dispatch
  Make_CFSL();          // 메쉬 구조를 탐색하여 CFSL 생성
  Select_Submesh(); //CFSL에서 최적의 가용 서브메쉬 선택
  if(최적의 가용 서브메쉬 = ∅)
  {
    if (External fragmentation) // 외적단편화
    {   //프로세서 단편 통합
        Processor_Fragmentation_Compaction();
        Goto Make_CFSL(); // Make_CFSL() 모듈로 이동
     }else{
        Waiting(); //할당 가능한 가용 서브메쉬가 생성될 때까지 대기
    }
  }
  Do_Allocate() //서브메쉬 할당 과정
  {
     Allocate(); // 최적의 가용 서브메쉬를 할당하고 Alloc-list에 삽입
     Delete();   // 가용 서브메쉬 S를 CFSL에서 삭제
  }
}
Processor_Fragmentation_Compaction()
{ //프로세서 단편 통합 과정
  for(int i=0; i <= N; i++) // N: 할당 서브메쉬의 수
  {
    HSA_CSA_VSA_Search();
     if( (dx != 0) or (dy !=0) ) { Task_Relocation(dx, dy)}
  }
}
HSA_CSA_VSA_Search()
{
   태스크 재배치 유형에 따라 수평(dx) /수직(dy) 이동거리 반환
}
Task_Relocation(dx, dy)
{
  // Ai(<x1, y1>, <x2, y2>)내의 태스크 재배치
  <x1, y1> = <x1-dx, y1-dy > // 좌측하단 노드(<x1, y1>)
  <x2, y2> = <x2-dx, y2-dy > // 우측상단 노드(<x2, y2>)
}
```

그림 4. 서브메쉬 할당 알고리즘
Fig. 4. Submesh allocation algorithm.

첫 번째, 할당 예는 T_5(4, 3)와 동일한 유형의 CFSL에서 최적의 가용 서브메쉬를 찾아 할당하는 예이다. 먼저 Make_CFSL() 모듈에서는 <그림 1>의 전체 메쉬를 탐색하여 IFS의 유형에 따라 CFSL을 생성한다. CFSL과 Alloc-list는 다음과 같다.

- Alloc-list={A_1(<2, 2>,<5, 4>), A_2(<5, 0>, <7, 1>), A_3(<2, 0>,<3, 1>), A_4(<4, 0>, <4, 1>)},
- SQ-flist={∅}, HR-flist={ S_1(<2, 5>, <5, 7>)},
- VR-flist={S_2(<0, 0>, <1, 4>), S_3(<6, 2>, <7, 4>)}.

Select_Submesh()에서는 태스크의 유형과 동일한 CFSL에서 최적의 가용 서브메쉬를 찾는다. T_5(4, 3)는 가로 직사각형이므로 HR-flist에서 S_1(<2, 5>, <5, 7>)을 최적의 가용 서브메쉬로 선택한다. 그리고 Do_Allocate() 모듈에서 T_5(4, 3)를 S_1(<2, 5>, <5, 7>)에

할당한다. <그림 5>는 T_5(4, 3)의 할당 결과이다. <그림 5>에서 생성된 Alloc-list와 CFSL은 다음과 같다.

- Alloc-list={A_1(<2, 2>,<5, 4>), A_2(<5, 0>, <7, 1>), A_3(<2, 0>, <3, 1>), A_4(<4, 0>, <4, 1>), A_5(<2, 5>, <5, 7>)},
- SQ-flist={∅}, HR-flist = {∅},
- VR-flist={S_1(<0, 0>, <1, 7>), S_2(<6, 2>, <7, 7>)}

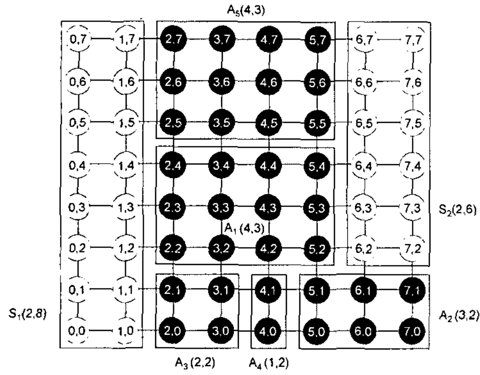


그림 5. 태스크 T_5(4, 3)의 할당 결과
Fig. 5. Result of Task T_5(4, 3) allocation.

두 번째, 할당 예는 외적단편화로 인해 할당 가능한 서브메쉬를 찾지 못함에 따라 할당 서브메쉬에서 수행중인 태스크를 재배치하고 프로세서 단편들을 통합하여 할당하는 예이다. <그림 5>에서 S_1(2, 8)과 S_2(2, 6)는 할당 서브메쉬들에 의해 좌우로 양분되었다. 이때 T_6(4, 6)이 주어진다면 전체 메쉬의 유휴 프로세서 수가 T_6(4, 6)이 요청한 수보다 많지만 할당 가능한 서브메쉬를 찾을 수 없다. 이러한 외적단편화로 인해 할당 지연이 발생하면 할당 서브메쉬에서 수행중인 태스크를 다른 가용 서브메쉬로 재배치하고 프로세서 단편들을 통합하여 할당한다. <그림 5>의 할당 서브메쉬들은 A_3(<2, 0>, <3, 1>), A_1(<2, 2>, <5, 4>), A_5(<2, 5>, <5, 7>), A_4(<4, 0>, <4, 1>), A_2(<5, 0>, <7, 1>)의 순서로 재배치된다. 먼저 A_3(<2, 0>, <3, 1>)는 <그림 3>의 재배치 유형 (1)에 해당되기 때문에 d_x=2와 d_y=0만큼 이동하여 S(<0, 0>, <1, 1>)에 재배치된다. 그리고 A_1(<2, 2>, <5, 4>)은 재배치 유형 (3)에 해당된다. A_1(<2, 2>, <5, 4>)의 HSA_1(<0, 0>, <1, 1>)에는 할당 서브메쉬가 존재하지 않기 때문에 수평할당범위를 생성할 수 없다. 따라서 수평이동거리는 d_x=2 이다. 하지만 CSA_1(<0, 0>, <1, 1>)에는 A_3(<2, 0>, <3, 1>)가 S(<0, 0>, <1, 1>)에 이미 재배치되어 존재하기 때문에 CAD_1(<0, 0>, <1, 1>)을 생성한다. 그리고 VSA_1(<2, 0>, <5, 1>)에는 A_2(<5, 0>, <7, 1>)와 A_4(<4, 0>, <4, 1>)가 존재하기 때문에 각 할당 서브메쉬의 할당 노드들을 탐색하여 VAD_1(<4, 0>, <5, 1>)를 생성한다. A_1(<2, 2>, <5, 4>)과 CAD_1(<0, 0>, <1, 1>), VAD_1(<2, 0>, <5, 1>)의 위치에 따라 수직이동거리를 구하면 d_y=2-2=0 이다. 따라서 A_1(<2, 2>, <5, 4>)는 d_x=2 와 d_y=0만큼 이동하여 S(<0, 2>, 3, 4>)에 재배치된다. A_5(<2, 5>, <5, 7>), A_4(<4, 0>, <4, 1>), A_2(<5, 0>, <7, 1>)에서 수행중인 태스크도 동일한 과정으로 재배치되면 <그림 6>과 같은 프로세서 단편 통합 결과를 얻을 수 있다.

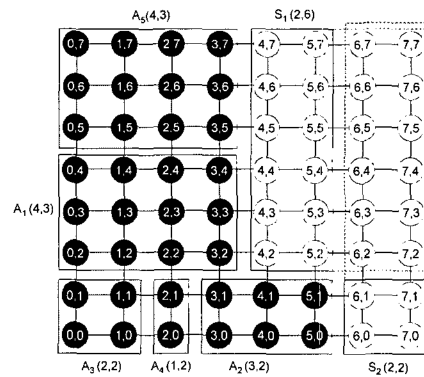


그림 6. 프로세서 단편 통합 결과
Fig. 6. Result of processor fragmentation compaction.

<그림 6>에서 생성한 Alloc-list와 CFSL은 다음과 같다.

- Alloc-list={A_1(<0, 2>,<3, 4>), A_2(<3, 0>, <5, 1>), A_3(<0, 0>, <1, 1>), A_4(<2, 0>, <2, 1>), A_5(<0, 5>, <3, 7>)},
- SQ-flist={S_2(<6, 0>, <7, 1>)}, HR-flist={∅},
- VR-flist={S_1(<4, 2>, <5, 7>)}.

<그림 6>에서 생성한 CFSL에서도 T_6(4, 6)에 할당 가능한 서브메쉬를 찾을 수 없다. 하지만 VR-flist의 S_1(<4, 2>, <5, 7>)은 확장지수 EI(S_1)=(<0, 0>, <2, 0>)를 사용하여 서브메쉬 S_1'(<4, 2>, <7, 7>)로 확장하면 할당이 가능하다. 따라서 VR-flist의 S_1(<4, 2>, <5, 7>)을 최적의 가용 서브메쉬로 선택하고 확장하여 할당함으로써 T_6(4, 6)의 대기시간을 줄인다. T_6(4, 6)의 할당 결과는 <그림 7>과 같다.

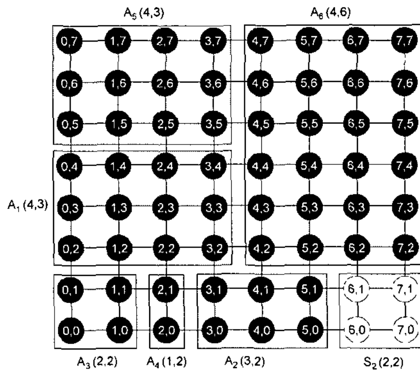


그림 7. 태스크 $T_6(4, 6)$의 할당 결과
Fig. 7. Result of Task $T_6(4, 6)$ allocation.

세 번째, 할당 예는 전체 메쉬 구조에 존재하는 유휴 노드의 수가 태스크가 요청하는 노드 수보다 적을 경우, 할당 가능한 가용 서브메쉬를 찾지 못해 할당이 지연되는 예이다. <그림 7>에서 $T_7(3, 3)$이 주어진다면 CFSL에서 할당 가능한 가용 서브메쉬를 찾을 수 없기 때문에 서브메쉬의 할당이 지연된다. 이때 $T_7(3, 3)$은 Alloc-list의 할당 서브메쉬들이 할당 해제되어 더 큰 가용 서브메쉬를 형성할 때까지 대기한다. 따라서 $T_7(3, 3)$의 대기시간이 증가한다.

4. 할당 해제 알고리즘

할당 서브메쉬에서 수행중인 태스크를 완료하면 할당 서브메쉬를 가용 서브메쉬로 환원하는 할당 해제 알고리즘은 <그림 8>과 같다.

<그림 8>의 할당 해제 알고리즘에서는 할당 서브메쉬가 IFS 인지 아닌지를 먼저 판별한다. IFS가 아니면 Deallocation() 모듈에서 할당 서브메쉬를 할당 해제한다. 그리고 Make_CFSL()모듈에서는 전체 메쉬 구조를 탐색하여 CFSL을 재생성 한다. 그러나 IFS 이면 Insert_CFSL() 모듈에서 서브메쉬의 유형에 해당하는 CFSL에 삽입함으로써 할당 서브메쉬를 해제한다. <그림 7>에서 $A_4(<2, 0>, <2, 1>)$는 할당 해제시에 다른 가용 서브메쉬의 크기에 영향을 미치지 않는 세로 직사각형 IFS이기 때문에 그 유형에 따라 VR-flist에 삽입함으로써 할당 해제한다. $A_4(<2, 0>, <2, 1>)$의 할당 해제 결과는 <그림 9>와 같다.

```
CFSL-TR_Deallocation ( ) //Deallocation(A<w, h>)
{
    if( Alloc-list = ∅){ // Alloc-list: 할당 서브메쉬 리스트
        SQ-flist={ M(W, H) }; HR-flist = VR-flist = ∅;
    } else {
        if (A(w, h) == IFS){
            Insert_CFSL(); // IFS 유형에 해당하는 CFSL 삽입
        } else{
            Deallocation(); //서브메쉬 할당해제
            Make_CFSL(); //가용_서브메쉬를 탐색하여 CFSL 재생성
        } //end if (A(w, h) == IFS)
    } //end if( Alloc-list =∅)
} //end
```

그림 8. 할당 해제 알고리즘
Fig. 8. Deallocation algorithm.

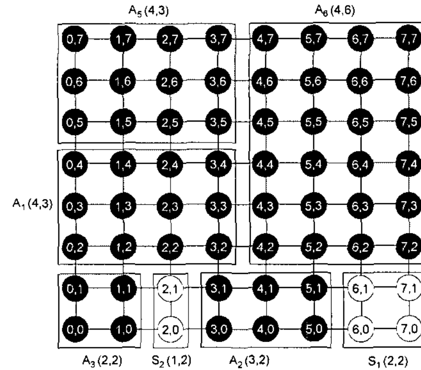


그림 9. $A_4(2, 1)$의 할당 해제 결과
Fig. 9. Result of $A_4(2, 1)$ deallocation.

V. 시뮬레이션

본 논문에서는 시뮬레이션을 통하여 제안한 CFSL-TR 할당방법이 기존의 FSL 할당방법[13]에 비해 태스크 대기시간을 줄이는 면에서 우수함을 보인다. Kim and Yoon[13]은 FSL 할당방법이 ADJ 할당방법[8]과 FL 할당방법[10]에 비해 우수함을 검증하였다. 그러므로 본 논문에서는 CFSL-TR 할당방법과 두 할당방법[8, 10]과의 성능 비교는 생략한다.

1. 시뮬레이션 환경

시뮬레이션에서는 객체지향형 프로그래밍 언어인 C#으로 개발한 시뮬레이터를 사용한다. 시뮬레이터에서는 다양한 속성을 가진 서브메쉬와 태스크를 객체로 구현한다. 작업부하는 태스크의 도착 분포, 크기 분포, 수행시간 분포에 따라 결정된다. 태스크는 도착율인 포아송 분포[17]에 따라 입력되고, FCFS 스케줄링에 따라 할당 순서가 결정된다고 가정한다. 태스크 도착율(λ)은 식 (2)와 같이 구한다[18].

$$태스크도착율\ (\lambda) = \frac{\rho \cdot N}{m \cdot r} \qquad (2)$$

식 (2)에서 ρ는 시스템 부하($0<\rho\le1$)이며, N은 메쉬 M(W, H)를 구성하는 전체 프로세서의 수이다. 또한 m은 태스크가 요청하는 평균 프로세서 수이며 r은 평균 수행시간이다. 각 태스크는 크기와 수행시간을 속성으로 가진다. 태스크의 크기는 지수분포, 균일분포, 정규분포에 따라 주어지고, 수행시간은 평균값 10인 지수분포에 따라 주어진다. 정규분포에서 태스크 크기의 평균은 (H+1)/2(W+1)/2 이고, 표준편차는 (H+1)/4(W+1)/4로 가정한다. 여기서 W와 H는 전체 메쉬 구조의 너비와 높이를 의미한다. 지수분포에서도 정규분포와 동일한 평균값을 가진다고 가정한다. 프로세서 단편 통합과정에서 태스크 재배치시간을 계산하기 위해 사용하는 태스크 단위이동시간(τ)은 시스템에 따라 달라지기 때문에 시뮬레이터의 입력 파라미터로 주어진다.

2. 시뮬레이션 결과 분석

시뮬레이션에서는 지수분포(Exp), 균일분포(Uni), 정규분포(Nor)에서 각각 300,000개의 태스크를 생성하여 할당하면서 시스템 부하와 메쉬 크기에 따른 평균탐색시간과 평균대기시간을 측정한다. 본 논문에서는 사용하는 성능 평가 척도는 다음과 같이 정의한다.

- 평균탐색시간(mean search time): 전체 메쉬 구조에서 탐색한 가용 서브메쉬를 유형에 따라 분류하여 CFSL을 형성하고 태스크에 최적인 가용 서브메쉬를 찾아 할당하는데 소요되는 평균시간이다.
- 평균대기시간(mean waiting time): 태스크가 큐에 처음 도착한 시점에서 할당되는 시점까지의 평균소요시간이다. 또한 프로세서 단편 통합과정에서 오버헤드로 발생하는 태스크재배치시간이 평균대기시간에 합산된다. 본 논문에서는 CFSL을 사용하여 가용 서브메쉬 탐색시간을 줄이는 방법보다 외적단편화로 인한 서브메쉬의 할당지연을 줄이는 CFSL-TR 할당방법이 태스크 대기시간을 단축하는 면에 있어서 더 효과적임을 검증하기 위해 평균대기시간을 사용한다.

첫 번째, 시뮬레이션에서는 CFSL을 사용하면 기존의 FSL할당방법에 비해 가용 서브메쉬의 탐색시간을 단축하는데 효과가 있음을 검증한다. 또한 가용 서브메쉬의 탐색시간을 단축함으로써 태스크의 대기시간을 줄일 수 있음을 검증한다. 이 시뮬레이션은 메쉬 크기(16×16 ~512×512)에서 다양한 시스템 부하($0<\rho\le1$)와 태스크 단위이동시간($0.01\le\tau\le0.05$)에 따른 평균탐색시간과 평균대기시간을 측정한다.

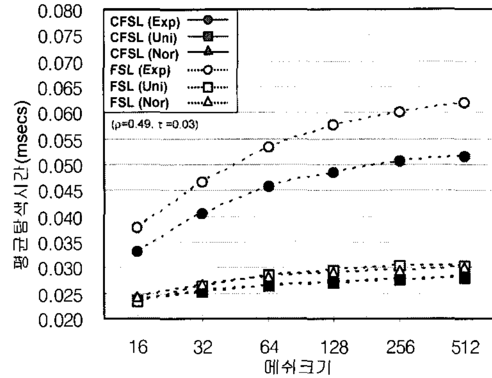


그림 10. 메쉬 크기에 따른 평균탐색시간 (ρ=0.49, τ= 0.03)

Fig. 10. Mean search time vs. size of the mesh (ρ=0.49, τ=0.03)

<그림 10>은 ρ=0.49, τ=0.03일 경우 지수분포, 균일분포, 정규분포에서 다양한 메쉬 크기에 따른 평균탐색시간을 측정한 결과이다. <그림 10>을 살펴보면 CFSL 할당방법은 지수분포, 균일분포, 정규분포에서 FSL 할당방법에 비해 각각 12.1~16.9%, 4.5~7.2%, 2.1~7.3%의 평균탐색시간을 줄였다. CFSL 할당방법은 전체 메쉬 구조에서 탐색한 가용 서브메쉬를 유형에 따라 분류하여 CFSL을 생성하고, 주어진 태스크의 유형과 동일한 CFSL에서 최적의 서브메쉬를 찾아 할당함으로써 서브메쉬 탐색시간을 줄인다. <그림 11>은 ρ=0.49, τ=0.03일 경우 지수분포, 균일분포, 정규분포에서 다양한 메쉬 크기에 따른 평균대기시간을 측정한 결과이다. <그림 11>을 살펴보면 균일분포, 정규분포에서 CFSL 할당방법의 평균대기시간은 FSL 할당방법에 비해 우수하지만 지수분포에서는 오히려 저하되었음을 볼 수 있다. <그림 10>에서 CFSL(Exp)는 FSL(Exp)에 비해 12.1~16.9%의 서브메쉬 탐색시간을 단축하였기 때문에 <그림 11>의 평균대기시간도 그 값에 비례하여 감소되어야 하지만 오히려 증가하였다. 지수분포에서는 태스크의 크기가 전체 메쉬 크기의 1/2 이상 되는 태스크의 비율이 3% 이지만 균일분포와 정규분포에서는 15%이다. 즉, 균일분포와 정규분포에 비해 지수분포에서는 크

기가 작은 태스크가 많다는 것이다. 태스크의 크기가 작으면 서브메쉬에 할당되어 동시에 처리될 가능성이 크고, 할당 서브메쉬에서 분할되는 가용 서브메쉬로 인해 외적단편화가 심해진다. 따라서 외적단편화로 인한 서브메쉬의 할당지연이 증가하기 때문에 태스크 대기시간도 함께 증가한다. CFSL 할당방법은 CFSL을 사용하여 서브메쉬의 탐색시간을 단축함으로써 태스크 대기시간을 줄였지만, 지수분포에서는 외적단편화로 인한 서브메쉬의 할당지연이 증가하면서 태스크 대기시간이 오히려 증가하는 문제점이 있다.

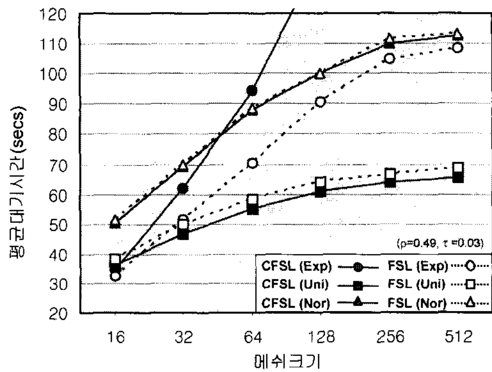


그림 11. 태스크 재배치를 적용하지 않을 경우 메쉬 크기에 따른 평균대기시간(ρ=0.49, τ=0.03)
Fig. 11. Mean waiting time without task relocation vs. size of the mesh(ρ=0.49, τ=0.03)

두 번째, 시뮬레이션에서는 태스크 재배치로 프로세서 단편들을 통합하여 할당하는 CFSL-TR 할당방법이 태스크 대기시간을 단축하는데 효과가 있음을 검증한다. 또한 CFSL-TR 할당방법이 CFSL 할당방법보다 태스크 대기시간을 단축하는 면에서 효과적임을 검증한다. 이 시뮬레이션은 메쉬 크기(16×16~512×512)에서 다양한 시스템 부하($0<\rho\leq1$)와 태스크 단위이동시간($0.01\leq\tau\leq0.05$)에 따른 CFSL-TR 할당방법의 평균대기시간을 측정한다. <그림 12>는 ρ=0.49, τ=0.03일 경우 다양한 메쉬 크기에 따른 평균대기시간을 측정한 결과이다. <그림 12>를 살펴보면 CFSL-TR 할당방법은 지수분포, 균일분포, 정규분포에서 FSL 할당방법에 비해 각각 23.2~38.1%, 12.3~15.6%, 9.4~16.9%의 평균대기시간을 줄였다. CFSL-TR 할당방법에서는 외적단편화로 인해 서브메쉬의 할당지연이 발생하면 할당 서브메쉬에서 수행중인 태스크를 재배치하고 프로세서 단편들을 통합하여 할당함으로써 태스크의 대기시간을 줄인다. 이때 태스크를 재배치하는데 소요되는 시간이 오버헤드로 발생한다. 메쉬 크기에 따른 평균 태스크 재배치시간을 측정한 결과는 <그림 13>과 같다.

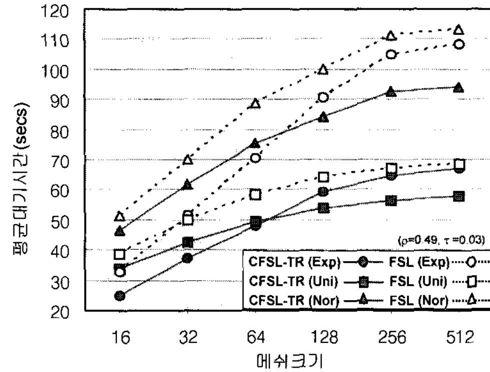


그림 12. 태스크 재배치를 적용한 경우 메쉬 크기에 따른 평균대기시간(ρ=0.49, τ=0.03)
Fig. 12. Mean waiting time with task relocation vs. size of the mesh(ρ=0.49, τ=0.03)

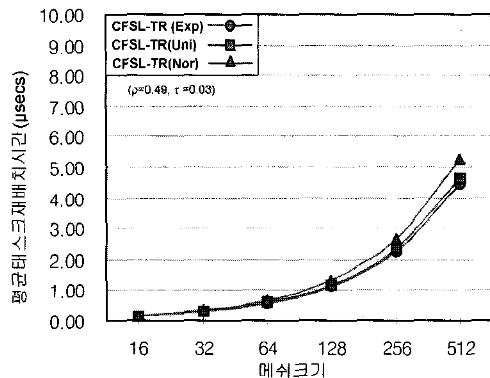


그림 13. 메쉬 크기에 따른 평균 태스크 재배치시간 (ρ= 0.49, τ=0.03)
Fig. 13. Mean task relocation time vs. size of the mesh(ρ=0.49, τ=0.03).

<그림 13>을 살펴보면 메쉬 크기가 증가하면서 평균 태스크 재배치시간도 함께 증가함을 볼 수 있다. 또한, 평균 태스크 재배치시간은 <그림 12>의 CFSL-TR 할당방법의 평균대기시간에 비해 무시해도 좋을 만큼 매우 작은 값임을 알 수 있다. 따라서 외적단편화로 인해 서브메쉬의 할당지연이 발생하면 태스크를 재배치하고 프로세서 단편들을 통합하여 할당하는 방법이 시스템의 성능 향상에 유리하다는 것을 알 수 있다.

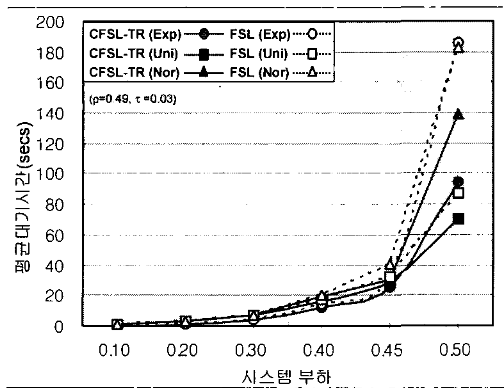


그림 14. 태스크 재배치를 적용한 경우 시스템 부하에 따른 평균대기시간 (N=256×256, τ=0.03)
Fig. 14. Mean waiting time with task relocation vs. System load (N=256×256, τ=0.03)

<그림 14>는 N=256×256, τ=0.03일 경우 다양한 시스템 부하에 따른 평균대기시간을 측정한 결과이다. <그림 14>를 살펴보면 시스템 부하가 0.1~0.3인 범위에서는 두 할당방법의 평균대기시간 차가 거의 없음을 볼 수 있다. 하지만 시스템 부하가 0.3~0.5인 범위에서는 시스템 부하가 증가할수록 CFSL-TR 할당방법의 평균대기시간이 FSL 할당방법보다 우수함을 볼 수 있다. 그러므로 CFSL-TR 할당방법은 시스템 부하가 큰 시스템의 성능 향상에 유리함을 알 수 있다.

Ⅶ. 결 론

본 논문에서는 메쉬 구조 다중컴퓨터 시스템의 성능을 향상시킬 수 있는 새로운 서브메쉬 할당방법으로 CFSL-TR 할당방법을 제안하였다. 이 할당방법은 먼저 태스크와 동일한 CFSL에서 최적의 가용 서브메쉬를 찾음으로써 평균탐색시간을 단축하였다. 하지만 외적단편화로 인한 할당지연이 증가하면 평균대기시간이 오히려 길어지는 문제점이 있었다. 이러한 문제점을 해결하기 위해 할당 서브메쉬에서 수행중인 태스크를 재배치하고 프로세서 단편들을 통합하여 할당하였다. 그 결과 지수분포, 균일분포, 정규분포에서 FSL 할당방법에 비해 각각 23.2~38.1%, 12.3~15.6%, 9.4~16.9%의 평균대기시간을 줄일 수 있었다. 따라서 CFSL-TR 할당방법이 기존의 할당방법에 비해 태스크 대기시간을 단축하는 면에서 우수함을 알 수 있었다.

또한 서브메쉬의 탐색시간을 줄이는 방법보다 외적단편화로 인한 서브메쉬의 할당지연을 줄이는 방법이 태스크 대기시간을 단축하는데 더 효과적임을 알 수 있었다. 그리고 프로세서 단편 통합과정에서 오버헤드로 발생하는 태스크 재배치시간은 감소하는 태스크 대기시간에 비해 매우 작은 값임을 알 수 있었다. 따라서 외적단편화로 인해 서브메쉬의 할당지연이 발생하면 태스크를 재배치하고 프로세서 단편들을 통합하여 할당하는 방법이 시스템의 성능 향상에 유리하다는 것을 알 수 있었다.

본 논문에서 제안한 CFSL-TR 할당방법은 3D 메쉬 구조 다중컴퓨터 시스템을 위한 새로운 서브메쉬 할당 방법 연구에 활용 할 수 있다. 또한, 시스템의 성능 향상을 위해 FCFS 스케줄링 외에 새로운 스케줄링 방법을 적용해 볼 수 있다.

참 고 문 헌

[1] Intel Corp., A Touchstone DELTA System Descripttion, 1991.
[2] Intel Corp., Paragon XP/S Product Overview, 1991.
[3] R. Alverson et al., "The Tera Computer System," Proc. 1990 Int'l Conf. Supercomputing, pp. 1-6, Nov. 1990.
[4] K. Hwang, Advanced computer Architecture, McGraw-Hill, 1996.
[5] K. Li and K.H. Cheng, "A Two-Dimensional Buddy System for Dynamic Resource Allocation in a Partitionable Mesh Connected system," IEEE Journal of Parallel and Distributed Computing, vol. 12, pp. 79-83, May 1991.
[6] 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, Aug. 1991.
[7] J. Ding and L.N. Bhuyan, "An Adaptive Submesh Allocation Strategy for Two-Dimensional Mesh Connected Systems," Proc. Int'l Conf. Parallel Processing, pp. II-193-200, Aug. 1993.
[8] Y. Zhu, "Efficient Processor Allocation Strategies for Mesh Connected Parallel Computers,"

IEEE Journal of Parallel and Distributed Computing, vol. 16, pp. 328-337, Dec. 1992.

[9] D.D. 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, Dec. 1993.

[10] S. Bhattacharya and W.T. Tsai, "Lookahead Processor Allocation in Mesh-Connected Massively Parallel Multicomputer," Proc. Int'l Parallel Processing Symp., pp. 868-875, Apr. 1994.

[11] T. Liu, W.K. Huang, F. Lombardi, and L.N. Bhuyan, "A Submesh Allocation Scheme for Mesh-Connected Multiprocessor Systems," Proc. Int'l Conf. Parallel Processing, pp. II-159-II-163, Aug. 1995.

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

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

[14] C.H. Chung and J.Y. Juang, "A Partial compaction scheme for processor allocation in hypercube multicomputers," Proc. Int'l Conf. Parallel Processing, pp. I-211-I-217, Aug. 1990.

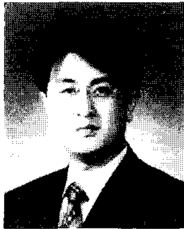
[15] N.F. Tzeng, H.L. Chen, "Fast Compaction in Hypercubes," IEEE Trans. on Parallel and Distributed Systems, vol. 9, no. 1, pp. 50-56, Jan. 1998.

[16] S.M. Yoo and H. Choo, H.Y. Youn, C. Yu, Y. Lee, "On Task Relocation in Two-Dimensional Meshes," IEEE Journal of Parallel and Distributed Computing, vol. 60, no 5, pp. 616-638, May 2000.

[17] S.M. Ross, Introduction to Probability Models, sixth edition, Academic Press, 1985.

[18] P. Krueger, T.H. Lai, and V.A. Radiya, "Processor Allocation vs. Job Scheduling on Hypercube Computers," Proc. 11th Int'l Conf. Distributed Computing Systems, pp. 394-401, Aug. 1991.

저 자 소 개



李 元 柱(正會員)

1989년 : 한양대학교 전자계산학과 학사. 1991년 : 한양대학교 전자계산학과 석사. 1997년 : 한양대학교 컴퓨터공학과 박사과정 수료. 1991년~1995년 : (주)큐닉스컴퓨터 응용연구소 선임연구원. 1999년~현재 : 두원공과대학 인터넷프로그래밍과 조교수. <주관심분야 : 병렬처리시스템, 인터넷 및 모바일 컴퓨팅, Grid 컴퓨팅>

全 昌 浩(正會員)

1977년 : 한양대학교 전자공학과 학사. 1982년 : Cornell University, 컴퓨터공학과 석사. 1986년 : Cornell University, 컴퓨터공학과 박사. 1977년~1979년 : 전자통신연구소 연구원. 1989년~현재 : 한양대학교 전자컴퓨터공학부 교수. <주관심분야 : 병렬처리시스템, 성능분석, Grid 컴퓨팅>