システムソフトウェアと
オペレーティング・システム 65−18
（１９９４．７．２２）

メッシュ結合並列計算機のパーティショニングアルゴリズム 2D Buddy または Adaptive Scan を使った時分割処理

須崎 有康　田沼 均　平野 聡　一杉裕志　塚本亨治

suzaki@etl.go.jp

電子技術総合研究所

メッシュ結合並列計算機用のパーティショニングアルゴリズムとして提案されている 2D Buddy と Adaptive Scan を時間分割処理 (TSS) に拡張した。これらのアルゴリズムはタスクのためのサブメッシュを計算機上に効率よく割り当てる。本 TSS では仮想並列計算機を用意し、これらのアルゴリズムを仮想並列計算機上でのサブメッシュの割り当てに用いる。ある仮想並列計算機でタスクが占有している領域を別の仮想並列計算機で未使用の場合、そのタスクは複数の仮想並列計算機に存在することができる。本論文ではシミュレーションにより、TSS が応答性とプロセッサの利用効率を上げることを示す。

A TSS using partitioning algorithm "2D Buddy" or "Adaptive Scan" for mesh-connected parallel computers

Kuniyasu SUZAKI, Hitoshi TANUMA, Satoshi HIRANO,
Yuuji ICHISUGI and Michiharu TSUKAMOTO

Electrotechnical Laboratory
1-1-4 Umezono,Tsukuba-city,Ibaraki,305,Japan

This paper presents a TSS which uses partitioning algorithm "2D Buddy" or "Adaptive Scan" for mesh-connected parallel computers. These algorithms partition a sub-mesh for an incoming task. The TSS has virtual parallel computers. The virtual parallel computers are executed by a real parallel computer alternately. Each virtual parallel computer is partitioned into sub-meshes by these algorithms and if a area of a sub-mesh of a task on a virtual parallel computer is free on other virtual parallel computers, the task can sit on virtual parallel computers. The TSS can shorten response time of tasks and increase utilization of computers.

1 はじめに

現在、並列計算機の効率的利用を目指してパーティショニングアルゴリズムの研究が行なわれている。パーティショニングアルゴリズムは投入されるタスクが要求するプロセッサ数や形状に合わせて、FCFS(First Come First Serve)で物理プロセッサに効率良く割り当てるアルゴリズムである。これらアルゴリズムによって複数のタスクを同時に実行できるようになり、また、全体のプロセッサのうち稼働しているプロセッサの比率(プロセッサ利用率)を高めることができる。

しかし、パーティショニングアルゴリズムで処理してもプロセッサ空間の大部分を占有するタスクが到着すると、その前後のタスクを組み合わせてプロセッサ空間に割り当てることができない閉塞状態になる。図1を例にとると、タスク 1,2,4,5 のみならばパーティショニングアルゴリズムを使って4つのタスクを同時に実行することができる。しかしタスク 3 が間に入るため、タスク 1,2 の処理後、タスク 3 を処理し、タスク 4,5 を処理することになる。このような状況ではプロセッサの利用率が低下し、効率的利用ができない。さらに、個々のタスクの応答性(レスポンス)も閉塞状態のために悪化する。

本論文では、このような状態を回避する手法としてパーティショニングアルゴリズムを拡張した時間分割処理(TSS: Time Sharing System)を提案する。提案するTSSでは、実並列計算機と同一形状の仮想並列計算機を用意し、タスクの割り当ては各仮想並列計算機にパーティショニングアルゴリズムを用いて行なう。一台の仮想並列計算機に割り当てられなかったタスクは別の仮想並列計算機を新たに用意し、そこに割り当てる。TSSでは、これらの仮想並列計算機に実並列計算機を一定時間ごとに割り当てる。このTSS化によって、一つの仮想並列計算機でタスク 1,2,4,5 を処理し、もう一つの仮想並列計算機でタスク 3 を処理することで閉塞状態を回避することができる。

現在TSSを採用している商用並列計算機としてCM5がある。CM5ではシステムの立ち上げ時にパーティションされた領域個々に対してTSSが実行可能である。しかし、立ち上げ時に決められたパーティションを変えることはできないため、立ち上げ時にパーティションした数以上のタスクを同時実行することができない。新情報処理開発機構(RWCP)で考案されているOSのTSS[7]は、分散樹状待ち行列によるTSSを提案している。ここではタスクの要求がプロセッサの個数のみで行なっており、ネットワーク構成を特に考慮していない。

本論文では二次元メッシュ結合並列計算機を対象とし、パーティショニングアルゴリズム 2D Buddy[3],[4] と Adaptive Scan[2] を拡張したTSSを提案する。投入されるタスクの形状は長方形であるものを対象とする。以下 2章で本TSSが想定するTSSの環境について概観する。3章ではパーティショニングアルゴリズムを紹介し、4章でTSSへの拡張方法を説明する。5章では提案するTSSの実行効率をシミュレーション結果で示す。6章で今後の課題を議論し、7章で結論を述べる。

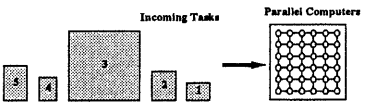

図 1: 閉塞状態になるタスクの投入

2 想定するTSSの環境

提案するTSSは二次元メッシュ結合の並列計算機を対象とする。並列計算機はサブメッシュを動的に切り出すことができ、タスクはサブメッシュに割り当てられる。タスクのX方向とY方向を転置した形状のサブメッシュ上にタスクの割り当ても可能であり、性能は変わらないものとする。また一定時間単位で全空間のプロセスを切替えられる機構を有するものとする。

投入されるタスクは並列計算機が提供する物理形状を越えない任意の長方形のプロセッサを要求するものとする。このタスクは終了するまで最初に要求したプロセッサ以上のプロセッサを要求したり、解放したりすることはしない。

3 パーティショニングアルゴリズム

メッシュ結合並列計算機を対象とするパーティショニングアルゴリズムにはFrame Slide[1]、First Fit[6]、Best Fit[6]、Adaptive Scan[2]、2D Buddy[3],[4] などがある。これらのうち本TSSで採用した2D Buddy と Adaptive Sacn について説明する。

3.1 2D Buddy

2D Buddy(2DB)[3],[4] では、メッシュ結合しているプロセッサが2の巾乗の正方形であることを仮定されている。2DBではサブメッシュの形状を並列計算機より小さい2の巾乗の正方形とする。この正方形のサブメッシュは、メッシュ結合しているプロセッサを四分割(縦方向二分割、横方向二分割)を繰り返して得られる正方形の位置に割り当てられる。投入された任意の長方形のタスクは、長辺 l が $2^n - 1 < l \leq 2^n$ であるサブメッシュに割り当てられる。

サブメッシュの空き状態は個々の大きさごとFBL(Free Block List)で管理し、該当する大きさのFBLが空の場合はより大きいサブメッシュを分割してタスクに割り当てる。管理されている状態を図2に示す。

投入されるタスクの大きさをX方向w、Y方向hとしたときの2DBによるサブメッシュの割り当てと解放の手続きを次に示す。

Allocation Procedure

1. Set i $\leftarrow min(\lceil \log w \rceil, \lceil \log h \rceil)$。
2. j $\geq$ i のうち最小のjとなる FBL_j から buddy 要素を探す。なかったら、"not found" をリターン。
3. While j $>$ i, FBL_j から buddy 要素の一つ k をとり、それを崩して $(k \times 4, k \times 4 + 1, k \times 4 + 2, k \times 4 + 3)$ を FBL_{j-1} に加える。Set j $\leftarrow$ j−1。

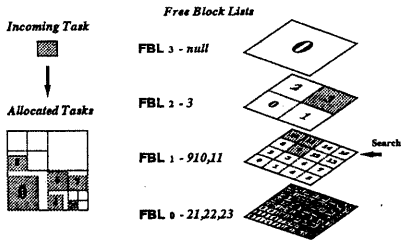

図 2: 2D Buddy の管理

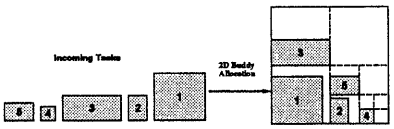

図 3: 2D Buddy によるサブメッシュの割り当て

4. i レベルの buddy 要素一つとりだし、それを返す。

Deallocation Procedure

1. 解放する Buddy 要素 k が該当するサイズの FBL_i に登録する。
2. k が属する Buddy 要素 4 つが FBL_i になかったらリターン、あったら
   (a) FBL_i から buddy 要素 4 つの要素を削除。
   (b) k ← (削除したうち最小の buddy 要素 /4)。
   (c) i ← i+1 として、ステップ 1 へ。

上記の手続きに従った割り当ての一例を図を 3に示す。

投入されるタスクの大きさを (w,h) とすると、一辺が $2^{max(\lceil \log w \rceil, \lceil \log h \rceil)}$ の正方形がサブメッシュとして取られる。このうちタスクが使用するのは $w \times h$ であるので、残りの $2^{max(\lceil \log w \rceil, \lceil \log h \rceil)} - w \times h$ は未使用であるが他のサブメッシュとして使用することはできない。この領域を内部フラグメンテーションと呼ぶ。割り当てが行なわれていない領域を外部フラグメンテーションと呼ぶ。

3.2 Adaptive Scan

Adaptive Scan(AS)[2] では投入されたタスクと同じ長方形のサブメッシュを、メッシュの下端 (0,0) から X 方向、Y 方向とサーチし、他のサブメッシュと重ならない最初の位置に割り当てる。また AS では投入されたタスクの長方形でサブメッシュが見つからないとき、タスクの X 方向の大きさと Y 方向の大きさを転置してもう一度サーチする。

AS のサブメッシュのサーチは一つの bit が一台のプロセッサを表す bit map を用いる。ここではサブメッシュを割り当てられない領域を示す coverage set と reject set がある。メッシュの大きさを (M,N)、投入されるタスクの大きさを (w,h) とすると、coverage set は、投入されたタスクのサブメッシュの左下端を割り当てることができない X 方向の領域 ($M-w+1$ から $M-1$) と Y 方向の

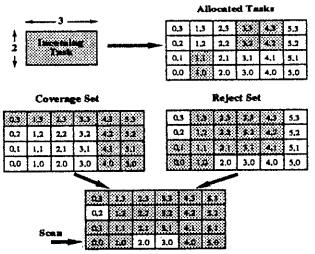

図 4: Adaptive Scan の管理

領域 ($N-h+1$ から $N-1$) を示す。reject set は、既にサブメッシュ (左下端が (p,q)、占有する大きさが (s,t)) が割り当てられているために投入されたタスクのサブメッシュの左下端を割り当てられない領域 ($(p-w+1, q-h+1)$ と $(p+s-1, q+t-1)$ を囲む長方形部分) を示す。AS のサーチでは、coverage set と reject set をマージした bit map から coverage set でなく reject set でない部分を探す。図 4に AS にサブメッシュサーチの一例を示す。

AS によるサブメッシュの割り当てと解放の手続きを次に示す。

Allocation Procedure

flag = FALSE として呼び出す。

1. If flag = FALSE then T ← T(w,h), a ← min(0,M−w+1), b ← min(0,N−h+1).
   otherwise T ← T(h,w), a ← min(0,N−h+1), b ← min(0,M−w+1)。
2. T と割り当てられているサブメッシュに基づいて coverage set を作る。x ← 0, y ← 0。
3. x,y が coverage set でなければ、ステップ 4へ。
   (a) If x < a − 1 then x ← d + 1(d は coverage set の x 方向の最大値), ステップ 3へ。
   (b) If x = a − 1 and y < b − 1 then x ← 0, y ← y + 1, ステップ 3へ。
   (c) If x = a − 1 and y = b − .1 and flag = FALSE, then flag ← TRUE, ステップ 1へ。そうでなかったら "not found" を返す。
4. if flag = FLASE then S ← (x,y,w−1,h−1); otherwise S ← (x,y,h−1,w−1), リターン。

Deallocation Procedure

1. サブメッシュを解放。リターン。

AS ではタスクの長方形と同じ大きさの長方形のサブメッシュを割り当てるので、2DB で発生した内部フラグメンテーションは起こらない。上記の手続きに従った割り当ての動作の一例を図 5に示す。

4 並列計算機の TSS

逐次計算機ではタスクに一定時間ごと計算機資源を割り当てていたが、本論文で提案する TSS では実並列計算

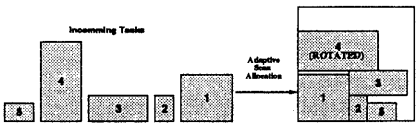

図 5: Adaptive Scan によるサブメッシュの割り当て

機と同一の構成をする仮想並列計算機を用意し、この仮想並列計算機を一定時間ごとに実並列計算機に割り当てる。この仮想並列計算機を本論文ではスライスと呼ぶ。タスクはスライスにパーティショニングアルゴリズムに従って割り当てられ、スライス数は投入されるタスクの数によって変化する。スライスへの時間割り当てはラウンドロビンで行なわれる。この状況を図6に示す。

並列計算機の TSS では、逐次計算機では問題にならなかったプロセッサ利用率の問題がある。一つのスライスにおいてはパーティショニングアルゴリズムによって利用率を上げることができる。本 TSS では更に、複数のスライスの同一領域にサブメッシュを割り当てることのできるタスクはプロセッサ利用率を上げるため、複数のスライスに存在する。ラウンドロビンが一周する間に複数のタスクを実行させる手法は、論文 [7] では樹状待ち行列を使って実現されている。本論文では一つのスライスにのみ存在するタスクを *single*、他のスライスに存在することができるタスクを *multiple* と呼ぶ。この状況を図6に示す。

4.1 TSS アルゴリズム

single と multiple のタスクの管理は図7に示すデータ構造で管理される。各スライスは、single と multipe のタスクを管理するリストを持ち、リストにはタスクの割り当てられている領域情報が格納されている。multiple リストでは、さらに multiple 元となっているスライスの情報を保持する。

本TSS はラウンドロビンでスライスに時間を割り当てているので、レスポンスを良くするにはスライス数を少なくする必要がある。スライスの削除はそのスライスに single タスクがなくなったときに行なわれる。もし mltiple タスクが存在すれば、multiple 元の multiple 情報を書き換え、削除される。ここでは、この動作を multiple タスクの縮退と呼ぶ。縮退されたタスクが一つのスライスのみで存在するタスクとなれば、その multiple タスク

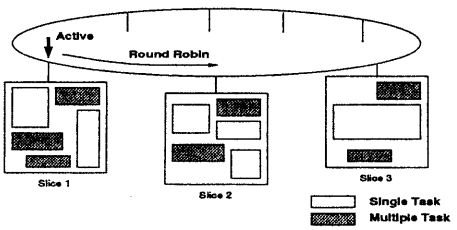

図 6: スライスに基づくタスクの割り当て

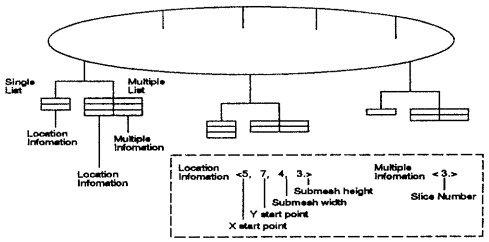

図 7: スライスのデータ構造

は single タスクに変わる。投入されたタスクと終了したタスクの管理手続きを次に示す。

Task Incoming Procedure

1. スライスを順に single タスクのみがあるものとしてパーティショニングアルゴリズムでサブメッシュの割り当てを行なう。割り当てられなければ、ステップ4へ。
2. 別のスライスにも同じサブメッシュが割り当てられれば multiple として登録、割り当てられなければ single として登録する。
3. 割り当たった領域に multiple タスクが存在していればその multiple タスクを縮退し、リターン。
4. 新しいスライスを作成し、パーティショニングアルゴリズムでサブメッシュを割り当てる。
5. 新しいスライスに他のスライスの single タスクが割り当てられるか探し、割り当てられたら multiple タスクにして割り当てる。新しいスライスに他のスライスの multiple タスクが割り当てられるか探し、割り当てられたら割り当て、リターン。

Task Outgoing Procedure

1. 終了したタスクが multiple なら、そのタスクを multiple リストから削除。ステップ2へ。終了したタスクが single であり、そのスライスから single タスクが無くなるなら、multiple タスクを縮退し、スライスを削除。そうでないなら、タスクを single リストから削除し、スライスを vacant-list に登録する。ステップ3へ。
2. 終了したタスクの multiple 元のスライスから、そのタスクを削除する。multiple を持っていた個々のスライスを vacant-list に登録。
3. vacant-list に登録されているスライスに割り当てられる single タスクを他のスライスから探し、割り当てられれば、multiple タスクにして割り当てる。次に割り当てられる multiple タスクを調べ、割り当てられれば割り当てる。
4. single タスクを multiple に変えたスライスから single タスクがなくなれば、そのスライスは multiple タスクを縮退した後、スライスを削除して、リターン。

4.2 重なり判定

タスクが他のスライスに割り当てられるか割り当てられないかは、タスクが割り当てられたサブメッシュが各々のスライスで重ならないかの判定に基づく。サブメッシュの重なり判定は 2DB、AS それぞれの特徴を生かして次の手法で行なっている。

2DB のサブメッシュの割り当て位置は X 方向、Y 方向とも 2 の巾ごとの四分割に従うので、四分木による管理を行なことができる。重なり判定はこの四分木を検索すること実現できる。プロセッサ数を n としたときの検索は $log_4 n$ で行なえる。

AS では、長方形のタスクを任意に投入するので 2DB の四分木のような効率的管理ができない。しかし、長方形の重なり問題は計算幾何学の分野でよく研究されており、長方形の数が N、交差の数が I のとき、$N \log N + I$ である効率的な判定アルゴリズム [5] がある。

4.3 TSS の動作

タスクの投入時、終了時のスライス上のタスクの独特な動作について説明する。

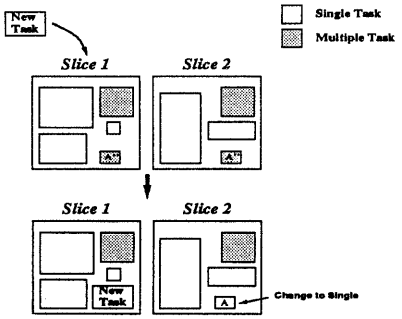

図 8: タスクの投入

タスクが投入され multipe であった既存のタスクが single に変わる例を図 8に示す。TSS では、タスクのためのサブメッシュの割り当てを single タスクのみを対象として行なう。割り当てが成功の場合、その領域が multiple として使われていないか調べ、使われていたら multiple のタスクを他のスライスに移す。図 8の multiple タスクは他に multiple がないため、single に変化している。

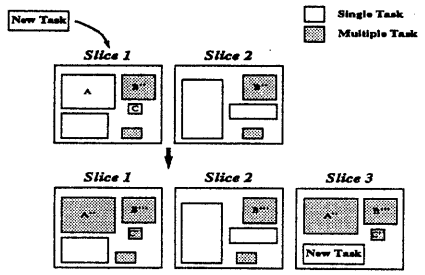

図 9: 新しいスライスの作成

新しいスライスが作成されるときの動作を図 9に示す。タスクが投入され既存のスライスに割り当てられないとき、新しいスライスが作成される。このとき新しいスライスは投入されたタスクのみではなく、他のスライスにあるタスクで新しいスライスにも割り当てられるものををできるだけ multiple とする。ここでは single であったタスクを優先とし、できるだけ multiple であるタスクを増やす。single タスクにできるだけ減らすことで、スライスを削除できる可能性を大きくする。

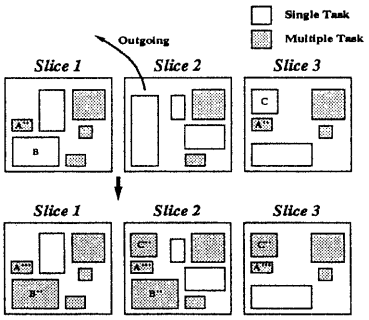

図 10: タスクの終了

タスクが終了したとき、そのタスクが占有していた領域を解放することで他のスライスのタスクが multiple として移る動作を図 10に示す。

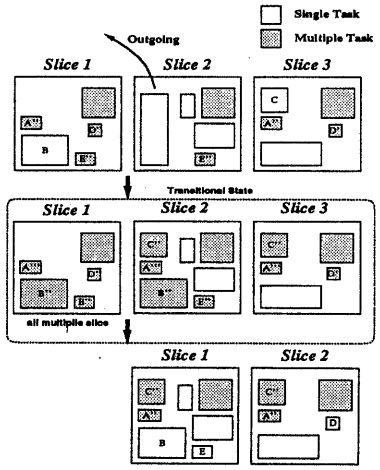

図 11: multiple タスクの縮退

図 11ではタスクが終了したとき、そのスライスに他のスライスからタスクが multiple として移り、あるスライスの single タスクが無くなった例を示す。このとき single タスクがなくなったスライスは削除され、そのスライスに multiple として存在していたタスクは縮退し、他のスライスに移る。

5 性能評価

TSS の性能評価には、タスクの到着時間 (arrival)、処理時間 (service) を M/M/1 型待ち行列に従うと仮定してシミュレーションを行なった。ここで用いる M/M/1 型待ち行列とは、arrival、service それぞれ指数分布に従い、タスクを受け付ける窓口が一つのモデルである。

本論文では、TSS 化した 2DB、AS と通常の 2DB、AS それぞれでタスクを割り当てた場合について比較し

た。対象となる計算機は、メッシュで 32 × 32 台構成とした。時間は仮想的時間を用い、タスクの投入は service (平均時間 100) / arrival (平均時間 25) = 4 となる状態、タスクの形状は X 方向、Y 方向とも 1 から 32 までの一様乱数に従うものとした。この状態のタスクを 100 個投入し、全タスクの終了までシミュレートした。

5.1 レスポンス

投入したタクスの到着時間 (arrival)、処理開始時間 (start)、終了時間 (finish) を示したグラフを図 12に示す。このグラフの X 軸最大値は全体の処理時間を示している。これより性能は TSS 化した AS、通常の AS、TSS 化した 2DB、通常の 2DB の順であることがわかった。正確な値はそれぞれ 4845, 5855, 8213, 9207 であった。

個々の処理を見ると TSS 化されていない計算機では、ほとんどのタスクの終了時間が到着順であったのに対し、TSS 化されている計算機では、到着順に依存していないことがわかる。これは、TSS 化しないとタスクの詰め込みで閉塞状態ができ、それがボトルネックとなり処理順が逐次化されるためである。

各タスクのレスポンス ($(finish-arrival)/service$) を図 13に示す。グラフより TSS 化されている計算機では、service に対するレスポンスが良いことがわかった。TSS 化されていない計算機では、service が短いタスクではレスポンスが非常に悪くなることがあった。このことはタスクの到着から処理開始までがタスクの大きさに関係なく、計算機の負荷状態によることが図 12より読みとれる。

service が長いタスクでは、TSS 化されていない計算機の方がレスポンスが良くなる傾向あった。最大 service (699) であったタスクのレスポンスを比較すると、通常の AS、TSS 化した AS、通常の 2DB、TSS 化した 2DB それぞれ 2.466, 5.63, 4.16, 10.45 であり、TSS 化されていない計算機が良い結果を示した。この理由は TSS 化していない計算機では一度タスクが割り当てられると CPU を終了まで占有できるためである。このため、service が長いタスクが投入されると TSS 化されていない計算機では、他のタスクが割り当てが困難になり、これらのタスクがレスポンスが悪く、犠牲となっていた。service が短いものほどこの影響が大きく、service が長いものとのレスポンスの差が大きい。TSS では service に関係なく計算時間を均等に割り当てるのでレスポンスがほぼ均一となった。

5.2 スライス数

図 14は TSS 化された計算機における経過時間対するスライスの変化を示す。図 13と比べるとレスポンスの上限が最大スライス数で抑えられていることがわかる。これはラウンドロビンでタスクを実行しているためである。また、TSS でも service が長いタスクがレスポンスが良くなる傾向があるが、これは service が長いタスクではスライスが多くなる期間に存在してもこの期間は一時期のため、平均化されると影響が薄くなるためである。最大スライス数である期間に投入された service の短いタスクは、スライス数の影響を直接受け、レスポンスが悪くなる。

2D と AS を比較すると AS の方がスライス数が少ないことがわかる。これはパーティショニングアルゴリズムの性能であり、次に示プロセッサ利用率より明らかになる。

表 1: プロセッサ利用率 (%)

	utilization		fragmentation	
	single	multiplie	internal	external
TSS 2DB	37.4		62.6	
	37.1	0.3	58.3	4.4
TSS AS	63.3		36.7	
	59.1	4.3	-	36.7
2DB	33.3		66.7	
	33.3	-	52.0	14.7
AS	52.4		47.6	
	52.4	-	-	47.6

5.3 プロセッサ利用率

表 1は時間平均のプロセッサ利用率とフラグメンテーションを示す。single、multiple は single タスク、multiple タスクとして実行された割合、internal、external は内部、外部フラグメンテーションの割合を示す。この表より、AS のほうが 2DB より利用率がよく、ともに TSS 化した方がさらに利用率がよかった。TSS 化による利用率の向上は multiple による効果より、TSS 化で多くのタスクを受け付ける効果が大きかったことがこの表からわかる。

フラグメンテーションは利用率と逆の結果になる。TSS 化した 2DB と通常の 2DB では、内部フラグメンテーションがフラグメンテーションのほとんどを占めていることがわかった。TSS 化して外部フラグメンテーションが減って、内部フラグメンテーションが増えている。この理由は、TSS によって通常の 2DB では外部フラグメンテーションであった領域にサブメッシュを割り当てることができたためである。この新たに割り当てたサブメッシュで、内部フラグメンテーションが発生し、通常の 2DB より内部フラグメンテーションが増える。

AS におけるタスクの転置の効果は、100 個のタスクに対し TSS 化した場合 13 回、通常の場合 15 回であった。このことより、約 1 割のタスクが転置することで直ちに割り当て可能だったことがわかった。

図 15は TSS 化した並列計算機の各プロセッサが処理したタスクの時間総和を示している。グラフの Z 軸の最大値は全タスクが終了するまでの時間を示している。このグラフより、AS も 2DB も X 軸、Y 軸の小さい方にタスクが多く割り当てられ、常時使用されていたことがわかる。また、AS の方が 2DB より均等にタスクをプロセッサに割り当て、負荷の均衡がはかられていたことがわかる。

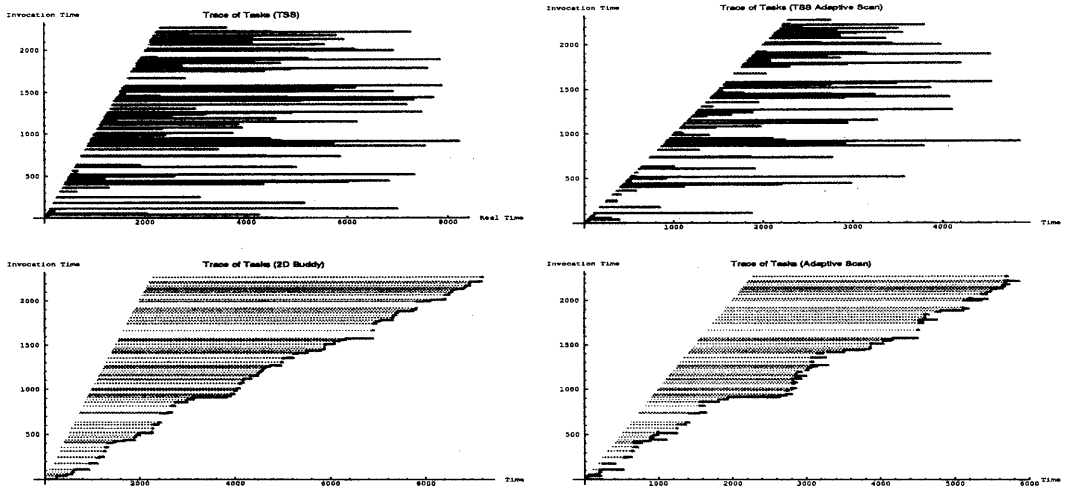

図 12: タスクが処理される過程：点線はタスク到着時間から処理開始までを示し、太線は処理開始から終了までを示す。

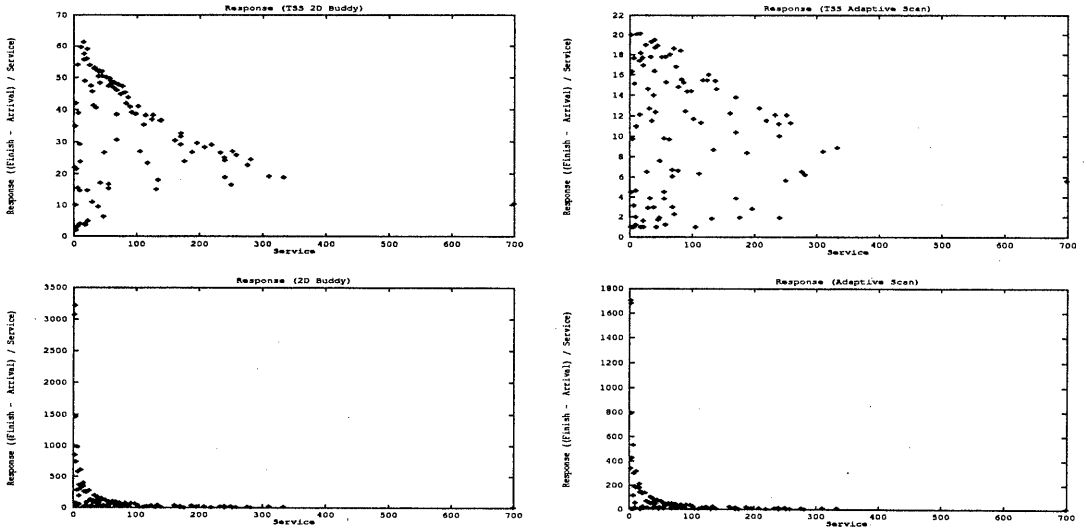

図 13: レスポンス

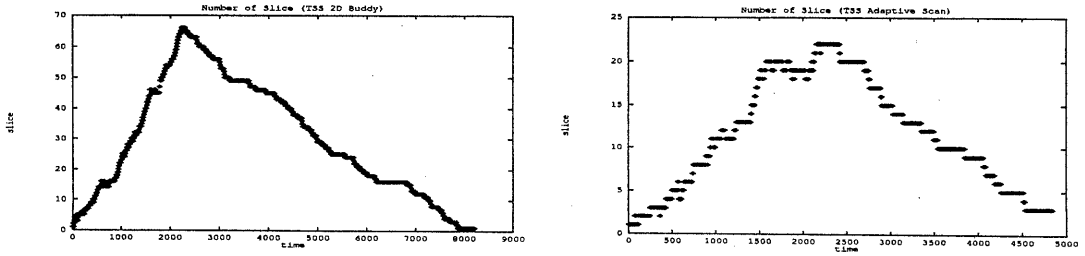

図 14: スライス数の時間変化

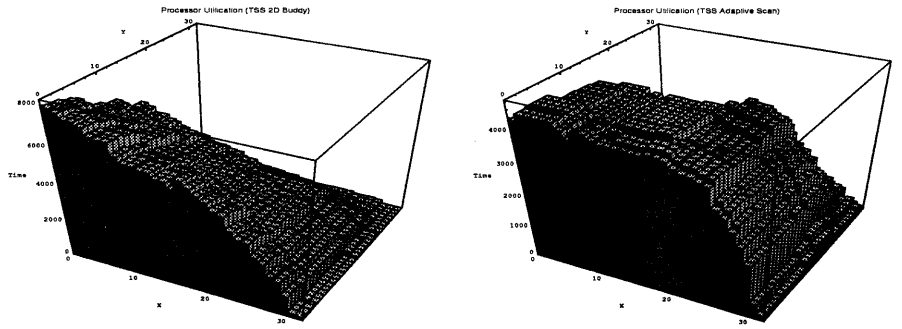

図 15: プロセッサ利用率

6 検討課題

提案したTSSには現在プライオリティの概念がないが、スライスごとにプライオリティを付け、管理することを考案中である。ここでは、投入されるタスクにプライオリティを付け、スライスが保持するタスクのプライオリティの総和をスライスのプライオリティとして管理する。ここで問題となるのは、スライスが形状の小さいタスクを多く保持する場合と形状の大きいタスクをいくつか保持する場合のプライオリティの計算やmultipleとなったタスクのプライオリティをどのようにするかながある。今後はこれらを解決し、プライオリティを導入した並列のTSSにする予定である。

本論文では、パーティショニングアルゴリズムとして2DBとASを採用したが他のパーティショニングアルゴリズムも容易にTSSに拡張できる。また、提案したTSSはメッシュ結合並列計算機を仮定しているが、他の結合形態の並列計算機にも容易に拡張可能である。他の結合形態に適用する場合、その結合形態に合ったパーティショニングアルゴリズムと重なり判定アルゴリズムを必要とする。これらは既にいくつか提案されているので、他の結合形態にもTSSを拡張していく予定である。

7 おわりに

本論文では、メッシュ結合計算機のパーティショニングアルゴリズム2D BuddyとAdaptive Scanを拡張したTSSを提案した。提案したTSSでは仮想並列計算機を用意し、これらのアルゴリズムを仮想並列計算機上でのサブメッシュの割り当てに用いた。仮想並列計算機は一定時間ごとに実並列計算機に割り当てられる。本TSSでは、ある仮想並列計算機でタスクが占有している領域を別の仮想並列計算機で未使用の場合、そのタスクは複数の仮想並列計算機に存在することでプロセッサ利用率を向上させた。

本論文では、提案したTSSの効率をシミュレーションにより示した。シミュレーションは、プロセッサは32 × 32台の計算機を対象にタスクをM/M/1モデルでタスクのサービス時間に対してタスクの到着間隔を4、大きさをX方向、Y方向とも1から32の一様乱数に従う100個で行なった。この結果より、タスクの応答性とプロセッサ利用率がパーティショニングアルゴリズムのみを使った場合より良くなることを示した。また、タスク全体の処理時間はTSS化により2D Buddyで1.12倍、Adaptive Scanで1.21倍であった。

謝辞

本研究の一部はRWC計画の一環として「超並列システムアーキテクチャに関する研究」で行なわれたものである。関係各位に感謝いたします。

参考文献

[1] Po-Jen Chuang and Nian-Feng Tzeng. An Efficient Submesh Allocation Strategy for Mesh Computer Systems. *Proc. on 11th International Conference on Distributed Computing Systems*, pp. 259–263, 1991.

[2] Jianxun Ding and Laxmi N. Bhuyan. An Adaptive Submesh Allocation Strategy for Two-Dimensional Mesh Connected Systems. *Proceedings of International Conference on Parallel Processing*, pp. 193–200, 1993.

[3] Kequin Li and Kam-Hoi Cheng. Job Scheduling in a Partitionable Mesh Using a Two-Dimensional Buddy System Partitioning Scheme. *IEEE Trans. on PARALLEL AND DISTRIBUTED SYSTEMS*, Vol. 2, No. 4, pp. 413–422, 1991.

[4] Kequin Li and Kam-Hoi Cheng. A Two Dimensional Buddy System for Dynamic Resource Allocation in a Partitionable Mesh Connected System. *Journal of Parallel and Distributed Computing*, No. 12, pp. 79–83, 1991.

[5] Robert Sedgewick. *ALGORITHMS*. Addison Wesley, 1988.

[6] Yahui Zhu. Efficient Processor Allocation Strategies for Mesh-Connected Parallel Computers. *Journal of Parallel and Distributed Computing*, Vol. 16, pp. 328–337, 1992.

[7] 堀敦史, 石川裕, 小中裕喜, 前田宗則, 友清孝志. 超並列オペレーティングシステムにおけるスケジューリングの提案. 情報処理学会研究会報告 *94-OS-63*, pp. 25–32, 1994.