システムソフトウェアと
オペレーティング・システム 65－5
（１９９４．７．２１）

超並列マシンにおける時分割スケジューリング

堀 敦史, 石川 裕, 小中 裕喜, 前田 宗則, 友清 孝志
技術研究組合 新情報処理開発機構（RWC） つくば研究センタ
e-mail:{hori,ishikawa,konaka,m-maeda,tomokiyo}@trc.rwcp.or.jp

我々は, パーティション分割可能な分散メモリ型並列マシンをターゲットに, 分散樹状待ち行列と呼ばれる時分割スケジューリング方式を既に提案した. 本論文では, 分散樹状待ち行列の特性のいくつかについて明らかにすると同時に, タスク割当ポリシーを提案する. 最適なスケジューリングを得るためには, DQT の負荷分散が重要である. うまく負荷分散された DQT においては, 公平なスケジューリングと高いプロセッサ利用率が達成される. ここで提案されたタスク割当ポリシーはシミュレーションにより比較された. また, バッチ処理との比較において, DQT を用いたスケジューリングの方が高負荷時により高いプロセッサ利用率を達成することが判明した.

A Time-Sharing Scheduling for Massively Parallel Machines

Atsushi HORI, Yutaka ISHIKAWA, Hiroki KONAKA,
Munenori MAEDA and Takashi TOMOKIYO

Real World Computing Partnership, Tsukuba Research Center

We have proposed a new process scheduling queue system called the **Distributed Queue Tree** (**DQT**) for a distributed memory, dynamically partitionable parallel machine. In this paper, the fundamental characteristics of DQT are analyzed. We also propose several task allocation policies. To achieve the best scheduling policy, DQT load balancing is very important. Failing to do so results in unfairness in process scheduling and/or low processor utilization. Several task allocation policies have been evaluated by software simulations. The simulation results show that time-sharing with DQT results in better processor utilization than that available from batch scheduling in high-load situations.

1 はじめに

並列マシン上での時分割スケジューリングの基本的アイデアは Ousterhout により提案されている [4]. しかしながらこの提案ではプロセス走行待ち行列を集中管理しているため, 大規模システムではボトルネックを生じる可能性があると同時に, ネットワークトポロジーからくるパーティショニングの制約を反映していない.

タスクをどのパーティションに割り当てるかという問題に対しては, 文献 [3][7] などでいくつかの方式が提案されている. ハイパーキューブ結合のマシンにおいて, この問題に対する研究は比較的盛んである (例えば, 文献 [1]). 一方, 文献 [2] ではパーティション選定アルゴリズムよりも (バッチ) スケジューリング方式そのものを見直した方が, より高いプロセッサ利用効率が得られることを示している.

我々は時分割スケジューリングと動的パーティショニングを組み合わせた新しいスケジューリング方式分散樹状待ち行列 (Distributed Queue Tree : DQT) を提案した [10]. DQT の特徴を以下に示す.

- DQT は並列マシンにおける横取り可能なスケジューリング方式である.
- プロセス走行待ち行列を分散させているため, ボトルネックを生じ難い.
- バッチスケジューリング方式に比べ, より高いプロセッサ利用率を得ることができる.
- 高負荷時においても DQT は安定かつ実用上十分な性能を発揮できる.

論文 [10] では DQT の基本的なアイデアのみの説明にとどまっていた. 本論文では, その後の研究で明らかになった DQT の性質, 挙動について述べ, 同時にいくつかの Task Allocation Policy (TAP) を提案する.

構成

まず最初に DQT の概要について触れ, 次に DQT に関する性質を用語の定義とともに説明する. また, いくつかの TAP を提案する. 最後にシミュレーションの結果を示す. 提案されたポリシーはシミュレーションにより比較される. また, DQT の特性をより明らかにするために, DQT のパラメータを変えたシミュレーション結果と, バッチ方式との比較結果も示す.

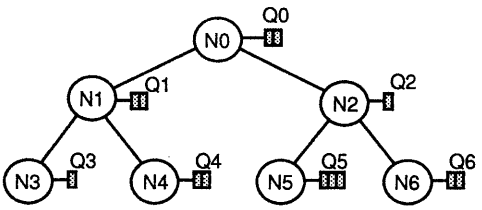


図 1: DQT の例

表 1: DQT スケジューリングの例

Time Slot	PE0	PE1	PE2	PE3
0	$Q_0(0)$			
1	$Q_0(1)$			
2	$Q_1(0)$		$Q_2(0)$	
3	$Q_1(1)$		$Q_5(0)$	$Q_6(0)$
4	$Q_3(0)$	$Q_4(0)$	$Q_5(1)$	$Q_6(1)$
5	$Q_3(0)$	$Q_4(1)$	$Q_5(2)$	$Q_6(0)$
6	$Q_0(0)$			
7	$Q_0(1)$			
8	$Q_1(0)$		$Q_2(0)$	
9	$Q_1(1)$		$Q_5(0)$	$Q_6(1)$
10	$Q_3(0)$	$Q_4(0)$	$Q_5(1)$	$Q_6(0)$
11	$Q_3(0)$	$Q_4(1)$	$Q_5(2)$	$Q_6(1)$
12	$Q_0(0)$			
:	:			

2 分散樹状待ち行列の概要

本章では, DQT の概略について述べる. 詳細については文献 [10] を参照されたい. 図 1に DQT の例を, 表 1 に図 1 の DQT における時分割スケジューリングの例を示す. 表中, $Q_i(j)$ とあるのは i 番目の DQT のノードの待ち行列中の j 番目のプロセスが走行していることを示す.

DQT は動的パーティションの入れ子構造を反映した木構造を成す. DQT ノードはパーティションに 1 対 1 で対応している. DQT ノードにはそのパーティションに割り当てられたプロセスの走行待ち行列を持つ (図 1 で, DQT ノードを示す丸の右側にあるのが待ち行列を表す). 負荷を分散させる意味から, DQT ノードは対応するパーティション内の適当なプロセッサに分散配置される.

DQT の木構造は 2 分木に限らない. 一般に N 分木構造を持つことができる. 以下 DQT の構造を表すのに w^h と表記する. ここで w はノードから派生する子ノードの数, h は DQT の木の高さである. 多くの場合, w はネットワークトポロジーとパーティショ

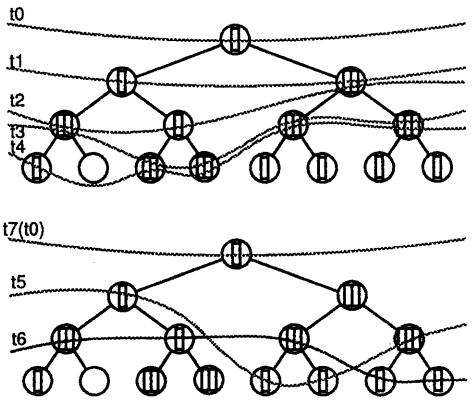


図 3: 前線移動の様子の例

ニングから規定される. h はシステムが提供するパーティションサイズの範囲を指定するパラメータとなる. 一般に w が大きくなると internal fragmentation [5] が増大するため, 注意が必要である.

DQT の各ノードは, 親ノードおよび子ノード間の通信により DQT 全体のラウンドロビンスケジューリングを実現する. 図 2 に DQT ノードの状態遷移図を示す.

DQT スケジューリングについて, ここでは直観的な説明にとどめる. DQT の木構造において, アクティブなノード (状態遷移で "Rounding" 状態にあるノードのこと) を結んだ曲線を「前線 (front)」と呼ぶことにする. 図 3に前線移動の例を示す. 図中の○中の長方形は待ち行列のエントリを示し, 前線は木を横断し左右に伸びる曲線で示した. 前線より上のノードの状態は全て "Rounded" であり, 前線より下のノードの状態は全て "Ready" である. 前線は葉に到達するまでは, 常に根から葉に向かう.

同じレベルのノードが全て同じ部分待ち行列長であるような場合, 前線は水平線となり (図中の t0, t1 の前線), 全ての葉に 1 回以上到達した時点でラウンドロビンが 1 周したことになる. しかし, 図 3に示すように, 部分待ち行列長が不揃いの場合は, 部分木がラウンドロビンにおける「1 周」するまで, 先に周回した方の枝を繰り返し周回させる. こうして, 全ての枝が少なくとも 1 周するまで, 前線は部分的に上下に波を打つ挙動を見せる (図中の t3〜t6 の前線). 負荷の低い DQT の部分木は負荷の高い部分木よりも多くスケジューリングされる. この方針はプロセッサ利用効率を高める一方, スケジューリングの不公平さを招く要因になる.

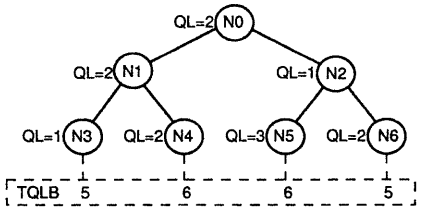


図 4: TQLB の例

3 DQT の性質

以下に DQT にまつわる用語の定義をしながら, DQT の挙動について説明する.

部分 DQT

DQT の部分木を部分 DQT (sub-DQT) と呼ぶ.

空 DQT

(Sub-) DQT 内にプロセスが存在しないものを空の (sub-) DQT と呼ぶ.

Total Queue Length of a Branch (TQLB)

TQLB は以下の式で計算される値である.

$$B_i = \begin{cases} 0 & \text{for } i = 0 \\ L_i + B_{\lfloor (i-1) \div w \rfloor} & \text{otherwise} \end{cases}$$

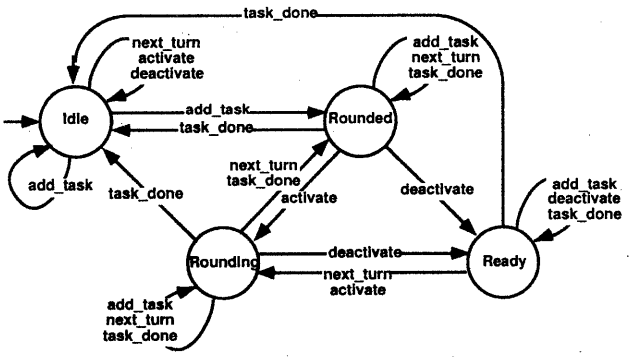
ここで, B_i は i 番目の DQT ノードにおける TQLB であり, L_i はそのノードにおける待ち行列の長さである. 各 DQT ノードは根ノードをゼロとし, 幅優先で番号付けられているものとする. 図 4 に例を示す.

TQLB は DQT の各枝上の全てのノードの待ち行列長の総和である. ある DQT における最大の TQLB は重要な意味を持つ. DQT スケジューリングの性質から, DQT 内の全てのプロセスは, TQLB の最大値分のタイムスロット数の時間内に少なくとも 1 回はスケジューリングされることが保証されている. TQLB の最大値は DQT の負荷状況を知る指標にもなる.

プロセッサ利用率

時刻 t におけるプロセッサ利用率 (U_t) は, $U_t = \frac{P_t^*}{P}$ で定義され, 時間区間 t_1 から t_2 における平均プロセッサ利用率 ($\tilde{U}$) は以下の式で定義される.

$$\tilde{U} = \frac{\sum_{t=t_1}^{t_2} P_t^*}{P \times (t_2 - t_1)}$$



状態の説明

Idle：自ノードの待ち行列が空である状態。

Rounded：タスクが投入されてまだスケジューリングされていない状態、または自ノードが管理する待ち行列のラウンドロビンスケジューリングが1周した状態を示す。この状態における next_turnメッセージはそのまま下位ノードに転送される。

Rounding：ラウンドロビンの最中である状態。

Ready：スケジューリング待ちの状態。

図 2: DQT ノードの状態遷移図

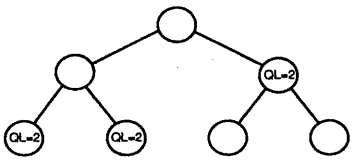


図 5: Balanced DQT の例

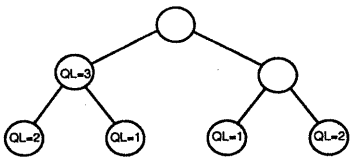


図 6: Fulfilled DQT の例

ここで, P_t^* は走行している可能性のあるプロセッサの数（あるいは, アクティブなパーティションにおけるプロセッサ数の総和）であり, P はシステム全体のプロセッサ数である.

このプロセッサ利用率は理想的な上限を示すものである. 実際のプロセッサ利用率は, パーティション内部で生じる internal fragmentation や, 通信や同期あるいは I/O などの待ちで生じるアイドルなプロセッサなどにより, ここでいうプロセッサ利用率よりも低い値となる. 本論文では, 理想的な状態のみに着目し, internal fragmentation やアイドルプロセッサによるシステム全体のプロセッサ利用率の問題は, 本論文の対象外とする.

Balanced DQT

空でない DQT において, 全ての TQLB が同じ長さであった場合, その DQT は "Balanced DQT" と呼ばれる（図5）. Balanced DQT において 100% のプロセッサ利用率が達成されると同時に, 全てのプロセスが全く公平なスケジューリングの機会を与えられる.

Fulfilled DQT

全ての DQT ノードについて, 子ノードを根とする部分 DQT の全てが空である, あるいは全てが空でない場合, その DQT は "Fulfilled" である, という（図6）. Fulfilled DQT ではプロセッサ利用率が 100% であることが保証されるが, スケジューリングの公平さは保証されない.

4 Task Allocation Policy

Task Allocation Policy（TAP）は, 投入されたタスクをどのパーティションに割り当てるかを決定するものである. この決定においてタスクが要求するプロセッサの数（これを「タスクサイズ」と呼ぶ）の情報しか得られないものとする. ただし, タスクサイズやタスクの実行時間の分布に関する知識を予めシステムに与えておくことは可能とする.

タスクがシステムに投入されると, add_task メッセージが根ノードに送られる. DQT ノードはこのメッセージを受けとると TAP に基づいて判断された子ノードに add_task メッセージを転送する. この手順は add_task メッセージが持つタスクの要求を満た

す最小の大きさのパーティションに到達するまで, 再帰的に繰り返される.

TAP の性能は, プロセッサ利用率とスケジューリングの公平さの2点から測ることができる. これは結局, DQT をいかにバランスさせるか (TQLB の差を少なくするか) ということと同じである. TAP に必要な情報は, 子ノードから根方向に向かって伝搬される. 詳細な情報を得ることは TAP の性能の向上に寄与する一方, 情報収集のためのオーバヘッドの増加を招く.

DQT が低負荷時, つまり DQT が fulfilled でないような状態では, タスクに十分な空きを見つけることが TAP の大きな関心である. 一方, 高負荷時, DQT がほとんど fulfilled であるような状態では, TQLB をバランスさせることが主な関心事となる. タスクは投入された時点で即座にキューに登録されなければならないため, タスクの投入により TQLB のバランスが崩れる場合も考えられる. TAP にはこのように矛盾する側面があり, 常に TQLB をバランスさせることは難しい面がある.

以下に, 我々が提案するいくつかの TAP について説明する.

Round-Robin Policy

もっとも単純な TAP である. 各ノードでは前回 add_task メッセージを転送した子ノードを覚えておき, 順次違う子ノードに add_task メッセージを転送する. この TAP において子ノードからの情報を必要としないため, 情報収集のためのオーバヘッドは存在しない. シミュレーションではこのポリシーの性能が非常に低いことが判明した. この結果から, TAP が非常に重要であることが判明したといえる.

Max-TQLB (MAX) Policy

DQT ノードがそれぞれの子ノードに対し下位の部分 DQT における最大の TQLB (Max-TQLB) の値を持つことにする. i 番目のノードの Max-TQLB (M_i) は以下の式で定義される.

$$M_i = \begin{cases} 0 & \text{for } i \geq w^h \\ L_i + \max\limits_{0 \leq k < w} M_{i \times w + k + 1} & \text{for } 0 \leq i < w^h \end{cases}$$

add_task メッセージを受けとった各ノードでは, 子ノードの中から最小の Max-TQLB を持つ子ノードに対しメッセージを転送するものとする. この TAP では Max-TQLB に着目し, 兄弟ノード間での Max-TQLB をバランスさせようとするものである.

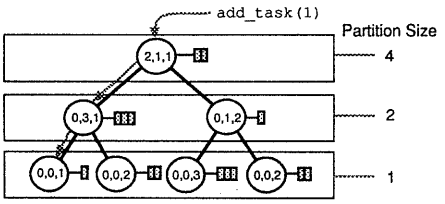


図 7: Best Fit ポリシーの例

Min-TQLB (MIN) Policy

Max-TQLB ポリシーに対し Min-TQLB ポリシーでは, TQLB の最大ではなく, 最小に着目して TQLB のバランスをはかるものである. i 番目のノードの Min-TQLB (N_i) は以下のように定義される.

$$N_i = \begin{cases} 0 & \text{for } i \geq w^h \\ L_i + \min\limits_{0 \leq k < w} N_{i \times w + k + 1} & \text{for } 0 \leq i < w^h \end{cases}$$

Assigned Processor Amount (APA) Policy

i 番目の DQT ノードの Assigned Processor Amount (APA) とは以下の式で定義される値 (A_i) である.

$$A_i = \begin{cases} 0 & \text{for } i \geq w^h \\ L_i \times S_i + \sum\limits_{k=0}^{w-1} A_{i \times w + k + 1} & \text{for } 0 \leq i < w^h \end{cases}$$

あるノードの APA の値は, そのノードを根とする部分 DQT に含まれるプロセスが必要とするプロセッサ量である. このポリシーに基づく TAP では, 子ノードのそれぞれの APA の値を保持しておき, add_task メッセージは最も APA の値が小さいノードに転送する.

APA の値は, その (部分) DQT に存在する全てのプロセスを一度にスケジューリングするのに必要なプロセッサ数を示す.

Best Fit (BF) Policy

各 DQT ノードは下位の部分 DQT のそれぞれのパーティションサイズのにおける待ち行列の最小値を保持する (図 7). add_task メッセージは対象となるパーティションサイズのうち, 最も短い待ち行列のノードを含む方に転送される. このポリシーでは TQLB のバランスに対する配慮が全く欠如しているため, 公平さを欠く傾向にある.

5 シミュレーション

システムの負荷率（Workload Ratio）を以下のように定義する（R_W）.

$$R_W = \frac{\sum_{l=0}^{L-1} task\_size_l \times task\_length_l}{P \times T}$$

ここで, $task\_size_l$ は l 番目のタスクが要求するプロセッサの数, $task\_length_l$ は同じく l 番目のタスクの理想環境下での処理時間, L は投入したタスクの総数である.

以下のシミュレーションでは, 投入したタスクのサイズは2のベキ乗に丸め, 丸められたタスクサイズに逆比例の分布とした. タスクの長さは 500〜19,999 単位時間の範囲で一様分布とした. 負荷率を変化させるため, タスクの投入間隔を調整した. 1回のシミュレーションでタスク投入間隔は一定である. シミュレーション時間は 1,000,000 単位時間である. 時分割スケジューリングのための量子時間は1単位時間に設定した.

タスクは I/O 待ちなど実行途中で中断しないものとし, 全てのタスクは独立, つまり, あるタスクの実行が他のタスクの影響を受けることはないものとした. あるポリシーにおいて, どこにタスクを割り当てるか判定できなかった場合には, 固定の子ノードにタスクを割り当てるようにした.

5.1 ポリシーの比較

一般的にプロセッサ利用率が高いほど, また, TQLB の最大値が小さいほど良い TAP であると言うことができる. 前述したように TAP の挙動は低負荷時と高負荷時で異なることが多いため, 負荷率 0.368 と 0.793 (ともに実測値) のそれぞれについてシミュレーションをおこなった.

表 2, 3 に TAP の違いによるシミュレーション結果を示す. DQT の構成は 2^7 である. TAP の性能を比較するために, プロセッサ利用率 (表中 "Proc. Util") および TQLB の最大値 (表中 "Max. TQLB") を載せた. 表の最後にある BF&APA ポリシーは, 最初 BF ポリシーで子ノードを選択し, 判断できなかった場合に APA ポリシーで判定するという, 複合ポリシーである.

MAX ポリシーは他のポリシーと比較してあまり良い結果ではない. TQLB の最大値を取る操作が TQLB の「穴」を隠してしまうために生じるものと考えられる. これを回避した MIN ポリシーでは, 高負荷時には比較的良い結果をもたらしているが, 低負荷時に問題が残る. これは, 低負荷時には最小の TQLB がほとんどの場合ゼロであるため, ポリシーの効果が出ないためと考えられる. 他の単体ポリシーに比較して APA ポリシーは低負荷時, 高負荷時とも良い結果を示した. BF ポリシーは, そのポリシーの性格上 TQLB のバランスを考慮していないため, TQLB の最大値が低負荷時に DQT の高さ分の倍数になる傾向がある. これはタスクの終了を考えない場合, 同一レベルの待ち行列長の差が高々1に抑えられるためである. この結果, BF ポリシーでは特に低負荷時において性能が悪い.

表 2: タスク割当ポリシーの比較 負荷率=0.368

Policy	Proc. Util.	Max. TQLB
MAX	0.366	4
MIN	0.366	5
APA	0.366	3
BF	0.363	7
BF & APA	0.366	3

表 3: タスク割当ポリシーの比較 負荷率=0.793

Policy	Proc. Util.	Max. TQLB
MAX	0.768	10
MIN	0.775	8
APA	0.777	8
BF	0.768	8
BF & APA	0.776	7

BF と APA ポリシーの組合せでは, 双方のポリシーが互いに補完し合い, 低負荷時, 高負荷時ともに表中で最も良い性能を示した.

5.2 プロセッサ利用率

ここでは, 先のポリシー比較シミュレーションで最も良い結果を示した BF&APA 複合ポリシーを用いて, DQT のより詳細なシミュレーションをおこなった結果を示す. 図 8 は, DQT の規模を 2^7, 2^8, 2^9, 2^{10}, 2^{11} とし, それぞれの場合について負荷率を変化させた時のプロセッサ利用率の変化をプロットしたものである. この図から, DQT の規模（高さ）とプロセッサ利用率に相関はないと考えられる. また, 負荷率の変化に対しプロセッサ利用率もほぼ比例して変化している. このことから高負荷時においても DQT が安定した挙動を示すことが分かる.

図 9 は, 図 8 と同様に DQT の規模や負荷率を変えた場合の実実行時間比（平均）をプロットしたものである. 実実行時間比（Real Execution Time Ratio:RETR）は, 理想環境下でのタスクの実行時間に対する実実行時間の比を表すもので, タスク l の実実行時間比を R_l^{RET} とすると, $R_l^{RET} = \frac{t_l^{task\_end} - t_l^{task\_entry}}{task\_length_l}$ と定義される. ここで, $t_l^{task\_entry}$ はタスク l の投入

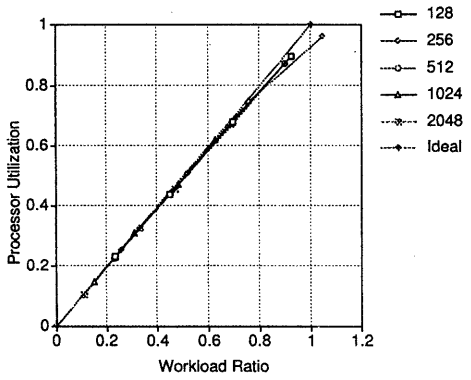


図 8: 負荷率によるプロセッサ利用率の変化

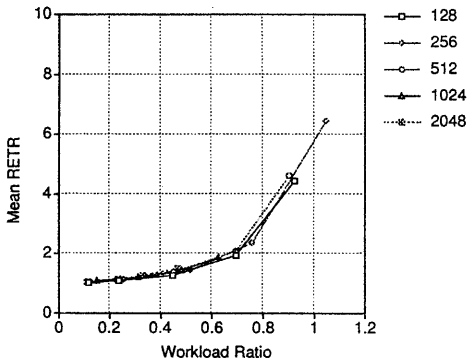


図 9: 負荷率による実実行時間比の変化

時刻, $t_i^{task\_end}$ は終了時刻である. この図から, 負荷率が 1.0 付近でも平均でタスクの実行時間が 6 倍程度に抑えられることが分かる. 理論的に RETR の上限はタスクの実行時間間隔における TQLB の最大値の積分した値で抑えられる.

5.3 Batch vs. DQT

DQT のスケジューリング方式と比較するために FCFS (First Come First Serve) スケジューリングポリシーに基づくバッチ方式のシミュレーションを行なった. タスクの割当には binary-Buddy 方式を採用した. この方式はメモリ割当アルゴリズムで良く知られる Buddy 方式をプロセッサ (パーティション) 割当に応用したもので, 比較的単純な方式でありながら性能的には他の割当アルゴリズムと遜色がない [2].

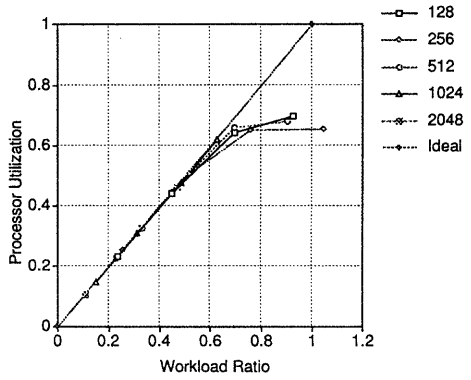


図 10: バッチ処理におけるプロセッサ利用率の変化

表 4 は表 2および 3 と全く同じタスク投入パターンでシミュレーションした場合の結果である. また, 図 10 は図 8 と全く同じ条件でシミュレーションした場合の結果である.

これらの図表からは判別し難いが, 低負荷時では FCFS バッチ処理の方がほんのわずかプロセッサ利用率が DQT を上回っていた. しかしながら表 4 の高負荷時の最大待ち行列長 (表中 "Max. Queue Length") が 361 と, 明らかに過負荷の状態を示していることが分かる. このことは, 図 10 において負荷率が 0.6 を越えた辺りからプロセッサ利用率が頭打ちとなっていることから, FCFS バッチ方式の限界が負荷率 0.6 付近であることが分かる.

一般的に, バッチ処理における頭打ちとなる負荷率はタスクの投入パターンやタスクサイズの分布などで変化するので, この値はあくまで参考値であることに注意を要する. しかしながら, DQT が理論的に 100% のプロセッサ利用率を達成可能であるのに対し, バッチ処理では 100% の利用率の達成はほとんど不可能である. このため, DQT は特に高負荷時においてバッチ処理よりも有利となる.

表 4: FCFS バッチスケジューリング

負荷率	Proc. Util.	Max. Queue Length
0.368	0.366	13
0.793	0.687	361

まとめ

本論文では, 分散樹状待ち行列（DQT）の特性のいくつかについて明らかにし, タスク割当ポリシー（TAP）を提案した. DQT がその特徴を発揮するには TQLB をバランスさせるように TAP を考える必要がある. 今回提案した TAP の中では, BF ポリシーと APA ポリシーの組合せが最適であった. 一般的には, タスクの分布, 想定されるシステムの負荷など考慮し TAP を設計すべきである.

シミュレーションでは, DQT がその規模によらず安定して高いプロセッサ利用率を実現することが示された. 一方, バッチ処理においては, 理論的に 100% のプロセッサ利用率の実現が困難であり, 高負荷時においてシステムが破綻することがシミュレーションでも確認された.

一般にバッチスケジューリングと時分割スケジューリングのどちらが効率的かの判断を下すことは難しい. 時分割スケジューリングの場合, プロセス切替のオーバヘッドをどの程度に見積もるかで実効のプロセッサ利用率が異なるからである. しかしながら並列マシンにおいても, RWC-1[6] のようにハードウェアによる支援があれば逐次マシンと同じ程度にまでオーバヘッドが低減されるものと考えられる [8]. 超並列の時分割スケジューリングは, 対話的なプログラミング環境を実現するだけでなく, より効率的なシステムの運用を可能にする.

今後の研究の方向としては, i) 統計的的手法による DQT の詳細な挙動の解析, ii) 実用的な時分割スケジューリングを実現するために優先度の導入, などを考えている.

DQT は我々のプロジェクトで開発が進められている超並列マシン RWC-1 上のオペレーティングシステムカーネル *SCore* [9] 上に実装される予定である.

謝辞

RWC 超並列ソフトウェアワークショップに参加の各位からは数々の貴重なアドバイスを頂いた. ここに感謝の意を表す.

参考文献

[1] Ming-Syan Chen and Kang G. Shin. Subcube Allocation and Task Migration in Hypercube Multiprocessors. *IEEE Transactions on Computers*, Vol. 39, No. 9, pp. 1146–1155, 1990.

[2] Phillip Krueger, Ten-Hwang Lai, and Vibha A. Dixit-Radiya. Job Scheduling Is More Important than Processor Allocation for Hypercube Computers. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No. 5, pp. 488–497, 1994.

[3] Keqin 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*, Vol. 12, No. 5, pp. 79–83, May 1991.

[4] John K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of Third International Conference on Distributed Computing Systems*, pp. 22–30, 1982.

[5] James L. Peterson and Theodore A. Norman. Buddy system. *Communication of the ACM*, Vol. 20, No. 6, pp. 421–431, June 1977.

[6] Shuichi Sakai, Kazuaki Okamoto, Hiroshi Matsuoka, Hideo Hirono, Yuetsu Kodama, and Mitsuhisa Sato. Super-threading: Architectural and software mechanisms for optimizing parallel computation. In *Proceedings of 1993 International Conference on Supercomputing*, pp. 251–260, 1993.

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

[8] 堀, 石川, 坂井, 小中, 前田, 友清, 松岡, 岡本, 廣野, 横田. 並列計算機オペレーティングシステムカーネル SCore におけるプロセス管理とハードウェア支援機能. コンピュータシステム・シンポジウム論文集, pp. 59–66. 情報処理学会, October 1993.

[9] 堀, 石川, 小中, 前田, 友清. 超並列システムカーネル SCore の構想. システムソフトウェアとオペレーティング・システム研究会資料, pp. 57–64. 情報処理学会, August 1993.

[10] 堀, 石川, 小中, 前田, 友清. 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 方式の提案. システムソフトウェアとオペレーティング・システム研究会資料, 94-OS-63, pp. 25–32. 情報処理学会, March 1994.