時分割空間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[†]

パーティション分割可能な並列マシンにおける時分割空間分割ジョブスケジューリングの具体例としてDistributed Queue Tree (DQT) を提案する．DQTはジョブスケジューリングを分散協調的に行い，プロセッサ資源の利用率を高めると同時に，対話処理を可能とする．本稿では，DQTの基本的な特性の解明を目的とし，その挙動の解析とシミュレーションによる評価を行う．その結果，タスクサイズの分布に関する独立性，高負荷時におけるプロセッサ利用率に関しては十分な性能を確認できた．しかしながら，大きいタスクほどスケジューリング上不利になる傾向があることが示された．

Time Space Sharing Scheduling

ATSUSHI HORI,[†] YUTAKA ISHIKAWA,[†] HIROKI KONAKA,[†]
MUNENORI MAEDA[†] and TAKASHI TOMOKIYO[†]

We propose a new job scheduling technique, Distributed Queue Tree (DQT) as an instance of Time Space Sharing Scheduling (TSSS) for partitionable parallel machines. DQT is a distributed and cooperated job scheduling process to achieve high processor utilization and to realize an interactive programming environment. In this paper, the basic characteristics of DQT are analyzed and evaluated by simulations. The simulation results show that DQT exhibits good processor utilization in high-load situations independently of task size distribution. In scheduling fairness, however, larger tasks tend to have less oportunity than smaller tasks.

1. はじめに

並列マシン上で動作する並列プログラムは，そこでのアルゴリズムや与えられたデータセットにより，台数効果の現れ方が異なる．より多くのプロセッサを与えても，計算速度の伸びが頭打ちになったり，通信のオーバヘッドにより，かえって速度が低下する場合もある．オペレーティングシステムの立場からは，システム全体の資源管理という視点で，タスク（プログラム＋データ）に与えるプロセッサ資源を適切に管理しなければならない．逐次マシンのオペレーティングシステムにおいては，プロセッサは基本的に1つであるが，複数の，それも 10^3 といった規模のプロセッサ資源を管理するという意味で，並列マシンにおけるジョブスケジューリングは，逐次マシンのそれとは異なる方式が必要となる．

Thinking Machines社のCM-5/CMOSTでは，システム起動時のパラメータとしてプロセッサ空間の分割が可能である[1]．また，Intel社のParagon/OSF-1では，実行時に2次元メッシュのネットワーク上に任意のパーティションを指定できる[2]．しかしながら，一様なプロセッサを持つ並列マシンの場合，パーティションはシステム全体の負荷を均衡させるように選定されるべきである．ユーザはプログラムの実行に際し必要と思われるプロセッサの数を指定し，オペレーティングシステムが最適なパーティションを選定する方式が望まれる．

一方，並列マシンがより一般的に使われるようになるであろう将来，バッチ処理よりも時分割スケジューリングによる対話処理が望まれる場合も増えるであろう．計算時間が数十分～数時間といったバッチ処理で十分であったプログラムが，並列マシンによる高速化で対話処理が可能となるような場合も考えられるからである[3]．また，並列プログラムのデバッグにおいては，別な意味で対話処理が重要と考えられる．

本稿においては，タスクの時分割スケジューリングと，プロセッサ空間を分割する空間分割スケジューリングを組み合わせた時分割空間分割スケジューリング（Time Space Sharing Scheduling：TSSS）の一方

† 新情報処理開発機構つくば研究センタ
Tsukuba Research Center, Real World Computing Partnership

式である Distirbuted Queue Tree（DQT）を提案する[4)~8)]．また，DQT の特性の解析およびシミュレーションによる性能評価の結果について報告する．

DQT は大規模な並列マシンにおいてもボトルネックを回避できるよう，その処理やデータ構造の分散が容易となっているだけでなく，プロセッサ資源を全体的な視野から有効に管理できるよう，分散された処理が協調的に動作する．

2. 用語の定義と想定

ここで対象とする並列マシンは MIMD 型であり，以下のようにパーティション分割可能とする．

- パーティションとは，ある並列マシンにおけるプロセッサ空間の部分空間のことである．
- 同じ大きさの異なるパーティションは，ユーザにとって等価である．
- パーティション分割そのものは静的に決定され，タスクの実行状況に応じて動的にパーティションの大きさ/形状を変化させるようなことは考えない．
- それぞれのパーティションはオーバラップ可能であるが，オーバラップしたパーティションは時間的に排他的であり，任意の時刻においてあるプロセッサは 1 つのパーティションにしか属さない．
- パーティション分割は任意の時点で変更可能な variable partitioning[3)] とする．

また，タスクについては以下のように想定する．

– タスクとは並列プログラム実行の実体である．

– タスクは 1 つのパーティションにのみ属し，実行の途中で他のパーティションに移動すること（マイグレーション）は考えない．

– タスクは任意の時点でプリエンプション（タスク切替）可能である．

– ある時刻において，2 つ以上のタスクが同じパーティションで走行することはない．

– 複数のタスクは，それらがオーバラップしないパーティションに属するとき，同時に走行可能である．

– タスクは起動時にそのタスク（並列プログラム + データセット）が要求するプロセッサ数を満たすに十分な大きさのパーティションが割り当てられる．

図 1 に時分割空間分割スケジューリングの例を示す．各タスクに割り当てられたあるタイムスロット時間におけるパーティションは「仮想化された並列マシン」を意味する．タスクの切替は，その時点で走行しているすべてのスレッドが，短時間のうちに一斉に切り替わるギャングスケジューリングを仮定している．複数のタスクがスレッド単位で各プロセッサ独立にスケジューリングする方式も考えられるが，この方式ではワーキングセットが大きくなりやすいこと，スケジューリングタイミングのずれに起因するバリア同期や通信に要する遅延時間が大きくなるといった processor thrashing の問題が生じる[9),10)]．

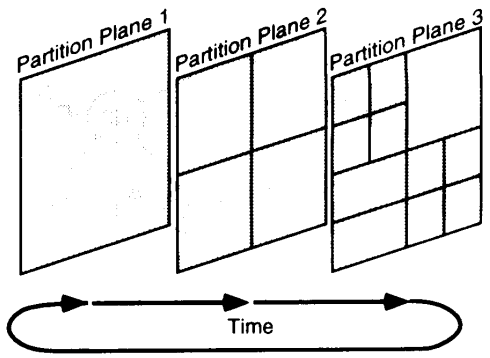


図 1 時分割/空間分割スケジューリングの例
Fig. 1 Example of Time Space Sharing Scheduling.

時分割空間分割スケジューリング方式の設計にあたっては，以下の点に注意しなければならない．

＊ 時分割空間分割スケジューラは，システムの状況に応じて適切なパーティションを選定するものとする．また，プロセッサ資源の有効活用という視点から，空間分割により生じる断片化（fragmentation）を最小限におさえる必要がある．

＊ 実践的なスケジューリングとするために，与えられたタスクのスケジューリングに際し，スケジューラは事前にタスクの走行時間を知る術はないものとする．

＊ タスクは，たとえば I/O 待ちなどにより，しばしばその状態が変化する．このため，従来のオペレーティングシステムに見られる 1 つのプロセッサに集中された待ち行列の実装は，ボトルネックとなる可能性がある．このため，スケジューリングの処理，およびスケジューリングに必要なデータ構造は適切に分散させ，スケーラビリティを損なわないようにする必要がある．

＊ スケジューリングのオーバヘッドは必要最低限におさえ，同一条件のタスクは公平にスケジューリングされることが望まれる．これら 2 つの要件は，逐次マシンにおけるスケジューリングと同じである．

＊ スケジューリングに用いるアルゴリズムの一般性および可搬性を高めるという意味において，特定のネットワークトポロジーに特化しないことが望まれる．

3. Distributed Queue Tree

図 2 に DQT の構造の例を，**表 1** に図 2 の DQT における時分割空間分割スケジューリングの例を示す．

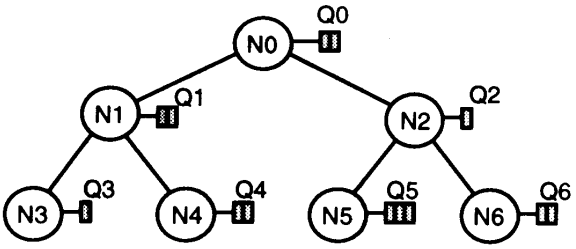


図2 DQTの例
Fig. 2 Example of DQT.

表1 DQTスケジューリングの例
Table 1 Example of DQT scheduling.

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	$Q_0(0)$			
:	:			

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 番目のタスク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タスクの走行待ち行列を持つ(図2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．しかしながら，実際に何をどのように分散させるかは具体的なシステムのパラメータにより決定されるべきである．ここでは，スケーラビリティを損なわないことを優先させ，できるだけ処理および情報を分散させる方向で考える．

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ニングの次数から決定される．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[11]が増大するため，注意が必要である．$w=2$ の場合，パーティションの割当はBinary-Buddy[12]と同等となる．DQTでは，ネットワークのトポロジー的に不連続なパーティション(「飛び地」の集合)は考慮していない．これは，i) タスク内での平均通信距離(ホップ数)が増大する，ii) タスク間に干渉が生じる可能性がある，iii) タスク切替のためのハードウェア支援機能[13)~15)]の実現が困難になる，といった理由による．

3.1 DQTスケジューリング

DQTの各ノードは，**図3**に示した状態遷移と，これから説明するノード間のメッセージにより，DQT全体のラウンドロビンスケジューリングを分散協調的に実現する．

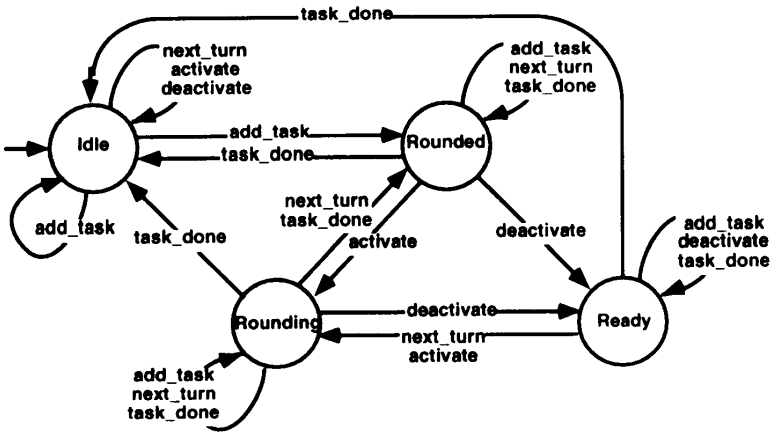
以下のメッセージの説明において，↓は根ノードから葉ノード(下位)に向かって流れるメッセージを示し，↑は葉ノードから根ノードに向かって流れるメッセージを示す．矢印のないものはそのノード内で発生したメッセージである．また，子ノードを持たない葉ノードは，永久にタスクが投入されない空の子ノードを持つと考える．

add_task ↓ 新規タスクの投入．タスクサイズが自ノードのパーティションの大きさと等しい場合は，自ノードの待ち行列に接続し，そうでない場合は，タスク割当ポリシー(後述)に従って下位ノードに転送する．

activate ↓ ノードを活性化する．その結果，待ち行列が空でなければRounding状態になる．そうでなければRounded状態になり **activate** メッセージをすべての下位ノードに転送する．このメッセージは根ノードが直下のすべてのノードより **rounded** メッセージを受け取った場合(ラウンドロビンでいうところの新しい周回に入ることを意味する)，および，下位の子ノードの **rounded** メッセージの同期待ちのときに，より早く **rounded** を報告した子ノードを再度周回させる場合に発生する．

deactivate ↓ ノードを不活性化する．その結果，待ち行列が空でなければReady状態になり，そうでない場合はIdle状態になる．このメッセージはすべての下位ノードに転送される．activateメッセージを受けて活性化したノードが，すべての下位ノードを不活性化する際に発生する．

next_turn ↓ ラウンドロビンにおけるタスク切替を指示する．このメッセージを受け，自ノードの待ち行列の次のタスクをスケジューリングする．もし，



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3 DQT ノードの状態遷移図
Fig. 3 State transition of DQT node.

待ち行列が空であったり，そのノードにおける最後のタスクを実行中であった場合は，自ノードの状態を *Rounded* とし，**next_turn** メッセージを下位ノードに転送する．このメッセージ送出の結果，以下のメッセージが返される．

rounding↑ 下位の部分 DQT がラウンドロビンでいうところの周回中であることを示す．

rounded↑ 下位の部分 DQT がラウンドロビンでいうところの周回が完了したことを示す．

すべての子ノードから1回以上 **rounded** メッセージを受け取った場合，すべての子ノードに **deactivate** メッセージを送出し，親ノードに対し **rounded** メッセージを返す．そうでない場合は，先に **rounded** メッセージを返したノードに対して再度 **activate** メッセージを送出し，親ノードに対し **rounding** メッセージを返す．

task_done タスクの終了あるいはタスクが中断された場合に発生する．この結果，そのノードのすべてのタスクがスケジューリングされた場合，自ノードの状態を *Rounded* にし，すべての子ノードに対し **next_turn** メッセージを送出する．

図3において，タスクの終了を示す task_done メッセージが，タスクを走らせていないはずの状態である *Rounded* や *Ready* 状態でも発生していることに注意を要する．これは，ノードの状態遷移がタスクの状態遷移と非同期に行われるからである．

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4** に前線移動の例を示す．図中，ノードを表す円の中の長方形は待ち行列のエントリーを示し，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すべて *Rounded* であり，前線より下のノードの状態はすべて *Ready* である．前線はつねに根から葉に向かう．

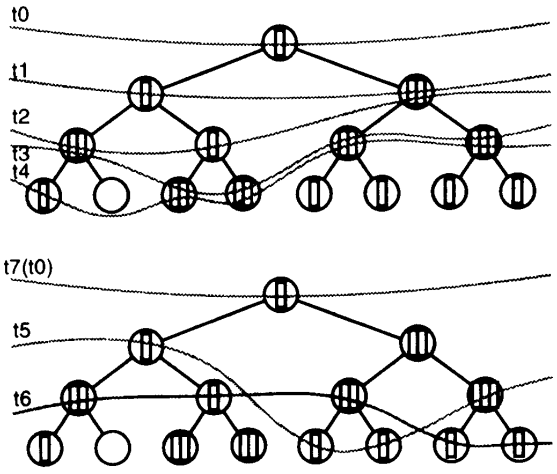


図4 前線移動の様子の例
Fig. 4 Example of front movement.

DQT の枝がすべて同じ負荷であるような場合，前線は水平線となる（図中の t0，t1 の前線）．しかし，図4に示すように，同じレベルの負荷が不揃いの場合は，負荷が軽い方の枝上の前線が先に進む．したがって，負荷の軽い枝の前線が先に前線が葉に到着することになる．この場合には先に到着した前線の部分は，負荷の不均衡が生じたノードまでさかのぼり，そこから再度，前線が葉に向かって進む．こうして，すべての枝が少なくともラウンドロビンにおける「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）．

結果的に，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．この点に関するシミュレーションの結果は，4.2 節および 4.3 節に示す．

3.2 DQT の性質

Total Queue Length of a Branch（TQLB）

TQLB とは DQT のある葉ノードから根ノードに至るすべてのノードの待ち行列長の総和である．TQLB は以下の式で再帰的に計算される値である．

$$B_i = \begin{cases} L_0 & \text{for } i = 0 \\ L_i + B_{\lfloor (i-1) \div w \rfloor} & \text{otherwise} \end{cases}$$

ここで，B_i は根ノードから i 番目の葉ノードに至る 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，幅優先の順序で番号付けられているものとする．**図 5** に例を示す．

DQT スケジューリングの性質から，DQT 内のすべてのタスクは 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 の負荷状況を示す 1 つの指標と考えることができる．

Balanced DQT

空でない DQT において，すべての TQLB が同じ長さであった場合，その DQT は “Balanced DQT” と呼ばれる（**図 6**）．Balanced DQT においては 100%のプロセッサ利用率が達成されると同時に，すべてのタスクが同等のスケジューリング機会を与えられる（これは DQT のスケジューリング方式と TQLB の定義から明らかである）．

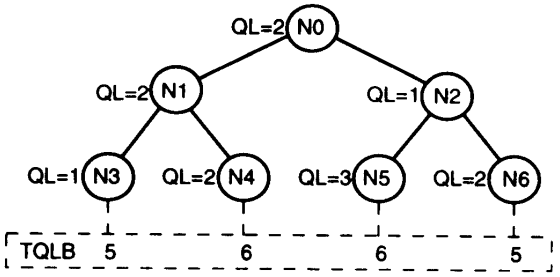


図 5 TQLB の例
Fig. 5 Example of TQLB.

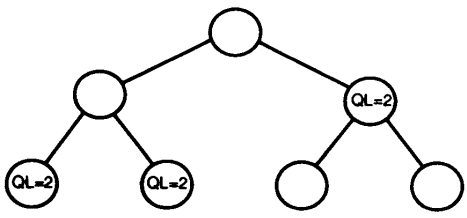


図 6 Balanced DQT の例
Fig. 6 Example of balanced DQT.

3.3 Task Allocation Policy

投入されたジョブにパーティションを割り当てる方式の例を**図 7** に示す．図中 **add_task** メッセージが DQT の根ノードに投げられている．各 DQT ノードを示す ○ の中の数字は，あるポリシーにより決まる負荷を代表する数値を示している．このポリシーでは，各 DQT ノードにおいて子ノードの中から最も数値の低いノードに対し，**add_task** メッセージを転送している．**add_task** メッセージ引数の数値は，ここで投入されたタスクが必要とするパーティションの大きさを示しており（この場合は 1），**add_task** メッセージの転送は，タスクが必要とするプロセッサ数に十分な大きさのパーティションを担当する DQT ノードに到達するまで続けられる．このタスクをパーティションに割り当てるポリシーを我々は Task Allocation Policy（TAP）と呼んでいる[6]．

ここではタスクのマイグレーションは考えていないため，タスクの割当は負荷を平衡させると同時にプロセッサ空間の断片を埋める唯一の機会である．負荷の偏りは，不公平なスケジューリングの原因になる．

以下に我々が提案した TAP の中から特に有効と思われるものについてのみ説明する．

APA ポリシー

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

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

ここで，S_i は i 番目の DQT ノードが担当するパーティションの大きさ（プロセッサ数）である．あるノードの APA の値は，そのノードを根とする部分 DQT に含まれるタスクが必要とする仮想的なプロセッサの量を示す．このポリシーに基づく TAP では，子ノー

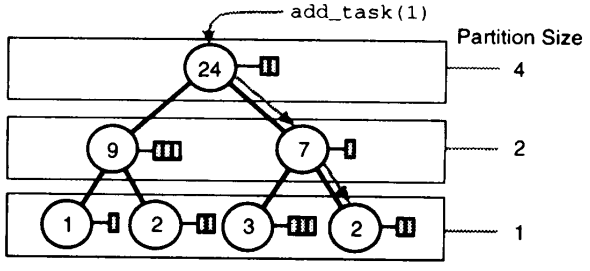


図 7 ポリシーの例（APA）
Fig. 7 Example of task allocation polisy (APA).

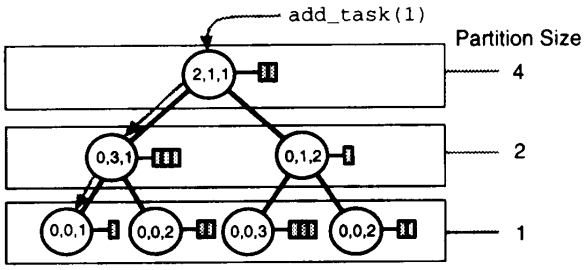


図8 ポリシーの例 (FF)
Fig. 8 Example of TAP (FF).

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．先ほどの図7はAPAポリシーによるタスク割当の例である．DQTノードを示す○の中の数字はAPAの値を示す．○の右にはそれぞれのノードの待ち行列の長さを表している．この例では図中最も下位の右端のノードに割り当てられている．

FFポリシー☆

各DQTノードは，そのノードを根とする部分DQTのそれぞれのパーティションサイズにおける待ち行列の最小値を保持する．図8はFFポリシーによるタスク割当の例である．DQTノードを示す○の中にある数列は，左から，パーティションサイズ4，2，1に相当する待ち行列の最小値を示している．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．この例では図中最も左端のノードに割り当てられている．FFポリシーではTQLBのバランスに対する配慮がまったく欠如しているため，公平さを欠く傾向にある．

FF-APAポリシー

いくつかのTAPを組み合わせることも考えられる．たとえば，上記のAPAポリシーとFFポリシーの組合せである．この場合，最初FFポリシーにより子ノードの選択を試み，判断できなかった場合にAPAポリシーで判定する．これをここでは，FF-APAポリシーと呼ぶことにする．現時点でこのFF-APAポリシーが最良と考えられており[6]，本稿におけるシミュレーションではすべてこのポリシーを用いた．

4. シミュレーションによる評価

4.1 シミュレーション条件

シミュレーションはすべてバイナリDQTを対象とした．特に明記されていない場合，シミュレーション時間は10^6単位時間，すべてのタスクはポアソン到着とし，タスクのサービス時間は平均10^3単位時間の指数分布とした．投入するタスクのサイズは，一様分布，タスクサイズに比例した分布，および，タスクサイズに逆比例した分布の3種類を行った．正規分布あるいはそれに類似した分布を用いたシミュレーションを用いなかった理由は，実際の並列マシンの運用を考えたとき，中間的な大きさのタスクが集中するような場合は考え難いためである．文献16)では，実測の結果，タスクサイズの分布はほぼ一様であったことが示されている．

すべての場合において，サービス時間の分布はタスクサイズの分布と独立である．ただし，最大パーティションを要求するタスクは，単一待ち行列のTSSと同じことを意味するため，すべての場合から除いてある．タスクは他のタスクと独立にスケジューリングが可能であり，タスクの実行の途中でサスペンドすることなく最後まで走り切るものとした．また時分割の量子時間は1単位時間とした．スケジューリングのオーバヘッドは無視されている．タスクが要求するプロセッサの数はすべて2のべき乗とした．シミュレーション結果にinternal fragmentation[11]は反映されていないため，プロセッサ利用率の数値は楽観的な値となっている．

ここで平均タスク到着時間間隔（$\widetilde{T_{interval}}$）は，システムの目標負荷率（W_{target}，$0 < W_{target} < 1$）に応じて以下の式で求めた値を用いた．

$$\widetilde{T_{interval}} = \frac{\widetilde{task\_size} \times \widetilde{task\_length}}{P \times W_{target}} \quad (1)$$

ここで，$\widetilde{task_size}$はタスクが要求するプロセッサの数の平均（タスクサイズの頻度分布から求めた値），$\widetilde{task_length}$はタスクサービス時間の平均，Pはシステムのプロセッサ数である．

タスクサイズ，タスク到着時間およびサービス時間は，それぞれ分布をもった乱数となっているため，必ずしもW_{target}で設定したとおりの負荷になるとは限らない．そこで，シミュレーションの結果から得られる実績負荷率（W_{actual}）を以下のように定義する．

$$W_{actual} = \frac{\sum_{l=0}^{L-1}(task\_size_l \times task\_length_l)}{P \times T}$$

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番目のタスクのサービス時間，Lは投入したタスクの総数，Tはシミュレーション時間である．

一方，時刻tにおけるプロセッサ利用率（U_t）は，P_t^*を走行している可能性のあるプロセッサの数（ある

☆ 文献5)，7)ではBF（Best Fit）ポリシーを呼んでいたが，最良とはいえないことからここでFF（First Fit）ポリシーと改名する．

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）とすると $U_t = P_t^*/P$ 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．

$$\tilde{U} = \frac{\sum_{t=t_1}^{t_2} P_t^*}{P \times (t_2 - t_1 + 1)}$$

さらに，実実行時間比☆（Real Execution Time Ratio: RETR）を定義する．RETR は個々のタスクのサービス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，

$$R_l^{RET} = \frac{t_l^{task\_exit} - t_l^{task\_entry}}{task\_length_l}$$

と定義される．ここで，$t_l^{task\_entry}$ はタスク l の投入時刻，$t_l^{task\_exit}$ は終了時刻である．

4.2 DQT の基本動作

図 9 にプロセッサ数が 128 の DQT に，逆比例のタスクサイズの分布で負荷を与えた場合の，各タスクの終了時刻におけるその RETR を点でプロットしたものである（横軸：シミュレーション時間，縦軸：RETR）．シミュレーション時間は 1.2×10^6 で 10^6 単位時間でタスク投入を打ち切っている．RETR はそのタスク実行時間中の最大 TQLB の平均になる．図には判読が難しくなるため示さなかったが，実際の最大 TQLB の線はこのグラフにおける RETR の包絡線に近い．タスクサイズとタスク到着間隔を互いに関連のない乱数として生成していることから，シミュレーション期間中の負荷が大きく変動していることが分かる．

RETR の各点は，最大 TQLB の近くに多く分布しているのが分かる．しかしながら分布の最大幅（ある時間間隔における最大の RETR と最小の RETR）は，特に負荷が高い場合において，かなり広がって分布している．これは，3.1 節で説明したように，DQT の負荷の低い部分により多くのスケジューリング機会を与えるというスケジューリング方式の結果と考えられる．

図 10 は RETR の図 9 のデータをタスクサイズごとに RETR の頻度分布を示したものである（縦軸：頻度）．いずれのタスクサイズにおいても RETR の分布はなだらかである．これは，前述したようにシステムの負荷がシミュレーションの期間中に大きく変動することと，DQT のスケジューリング方式の特性からくるものと考えられる．このグラフからは RETR の分布とタスクサイズの関係は明らかではない．この点に関しての統計的な解析に関しては 4.3 節にゆずる．

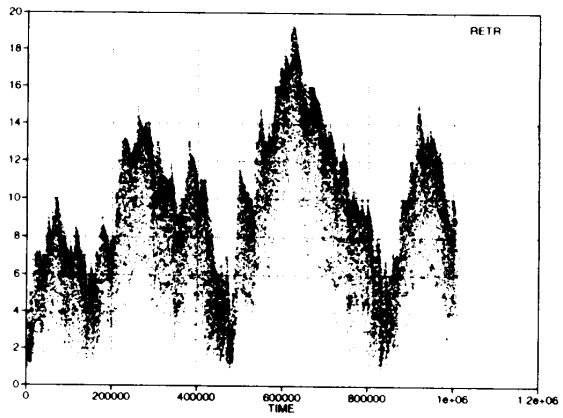


図 9 実実行時間比（プロセッサ数：2^7，タスクサイズの分布：逆比例，目標負荷率：99%）
Fig. 9 Real Execution Time Ratio (2^7 processors).

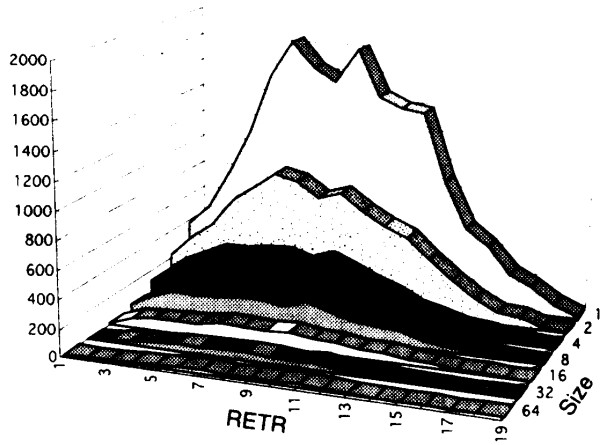


図 10 実実行時間比のヒストグラム（プロセッサ数：2^7，タスクサイズの分布：逆比例，目標負荷率：99%）
Fig. 10 Histogram of Real Execution Time Ratio (2^7 processors).

4.3 シミュレーション結果とその考察

図 11 にシミュレーション結果を示す．左列の 3 つのグラフは，負荷率（横軸）を変化させたときの，シミュレーション全体を通した平均プロセッサ利用率をプロットしたものである．上段のグラフはタスクサイズに正比例した分布，中段が一様分布，下段のグラフはタスクサイズに逆比例した分布の場合の結果である．高負荷時の違いを強調するため，低負荷時の結果はクリップしてある．また，参考のため，理想的な値を示す直線（"Ideal"）もプロットした．右列のグラフは，同じ列の左側のグラフと同じ条件でシミュレーションを行った場合の，RETR の平均を縦軸にプロットしたものである．これらのシミュレーションにおいて，システムの規模（式 (1) 中の P）を 128, 256, 512, 1024,

☆「実実行時間」（Real Execution Time）とは，あるタスクが走行キューに滞在する時間を指す．したがって，I/O 待ちなどタスクがサスペンドしている時間は含まれない．この意味で「経過時間（Elapsed Time）」とは異なる．

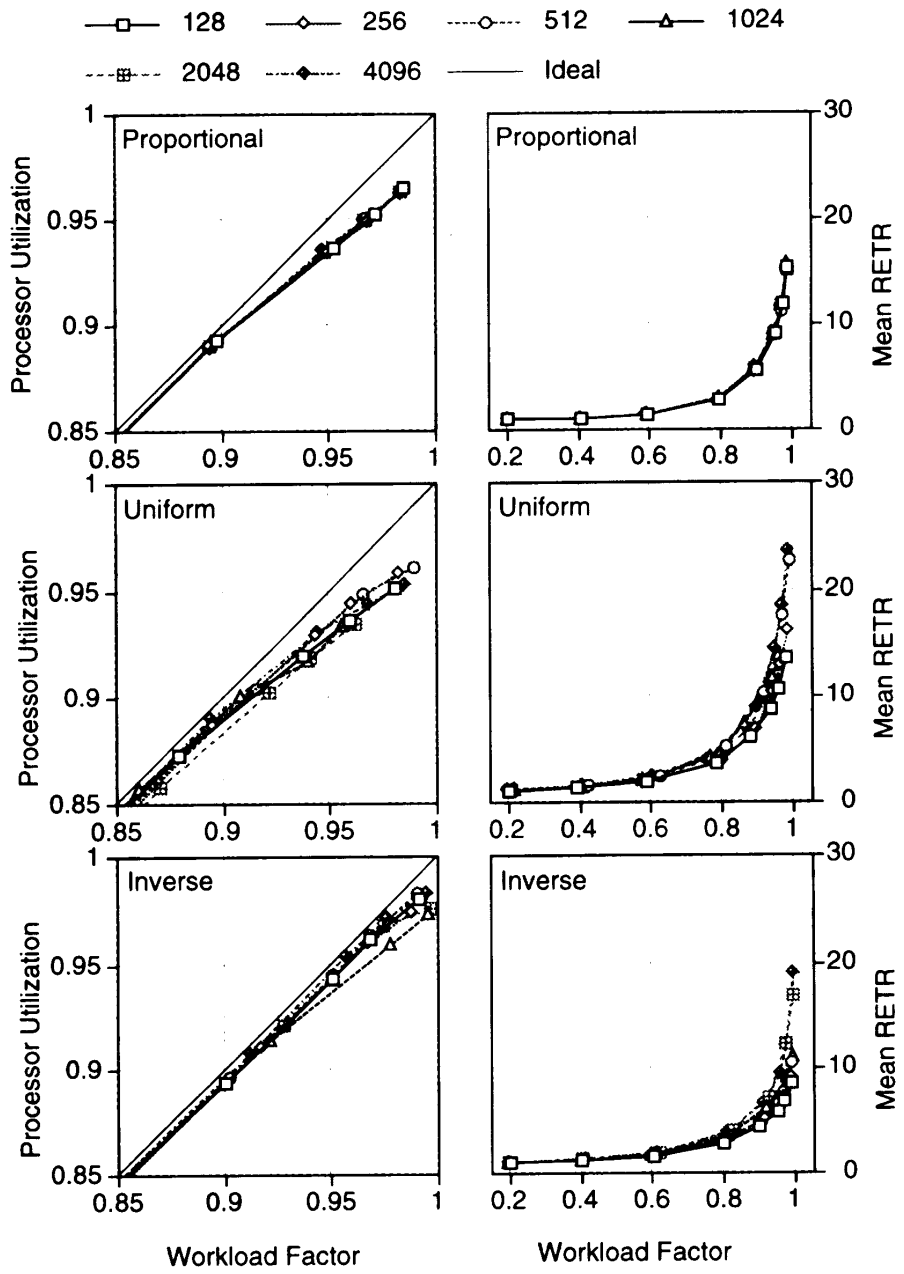


図 11 シミュレーション結果
Fig. 11 DQT simulation.

2048, 4096 としたときのそれぞれについて，目標負荷率（式 (1) 中の W_{target}）を 0.2, 0.4, 0.6, 0.8, 0.9, 0.95, 0.97, 0.99 と変化させた．プロットした負荷率の値は実績負荷率を用いた．乱数の偏りにより実績負荷率が 1 を超えたケースはグラフより除いてある．

図 11 の左列のグラフにおいて，理想状態（プロセッサ利用率が設定された負荷率と同じ）に近いほど良いスケジューリングであるということができる．これらのグラフから，DQT はシステムの規模の影響を受けずに，低負荷から高負荷に至るまで，理想に近い挙動を示した．

プロセッサ利用率のグラフ（図 11 の左列のグラフ）からは，3 種類のタスクサイズの分布の中では，サイズに逆比例する分布がベストであり，次いで正比例の分布，最悪が一様分布であると考えられる．DQT の性質から，あるレベルで生じた負荷のアンバランスはより小さいサイズのタスクでのみ解消することができる．逆比例分布の場合，小さなタスクが十分にあるため良好なプロセッサ利用率を示すものと考えられる．逆に正比例分布では，i) 大きなタスクが多数を占めているためにプロセッサ利用率が下がらなかった，ii) 負荷の不均衡が DQT のスケジューリングにより解消

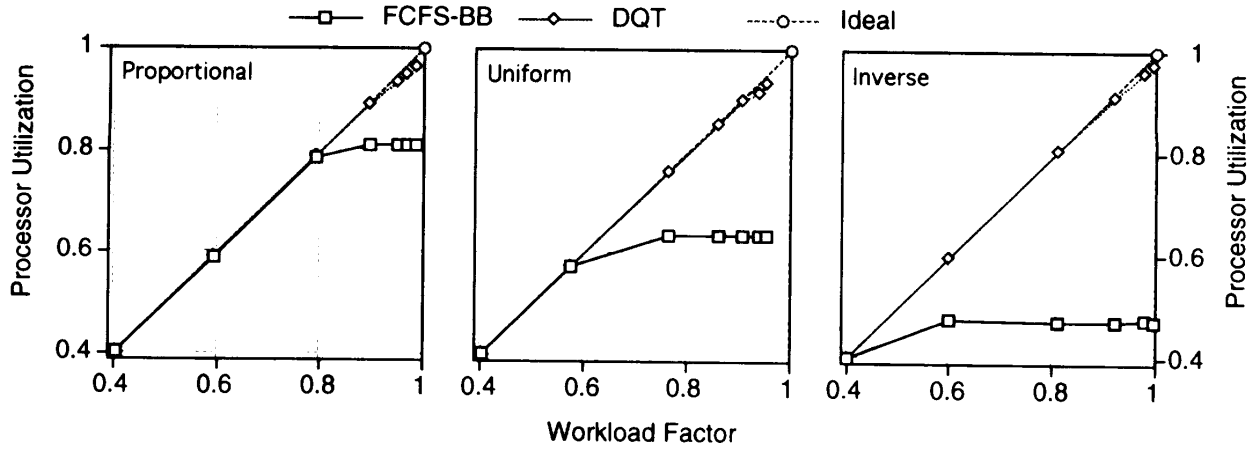


図12 バッチスケジューリングとのプロセッサ利用率の比較（プロセッサ数：2^{10}）
Fig. 12 DQT vs. Batch Scheduling (2^{10} processors).

された，と推測される．

バッチスケジューリングとの比較

図12は，Binary-Buddy方式[12]によるパーティションを割り当て，First-Come-First-Serve (FCFS) の順にバッチスケジューリングを行う場合 (FCFS-BB) と，DQTによるスケジューリングを同じシミュレーション条件で，プロセッサ利用率に関して比較したものである（プロセッサ数：1024）．

FCFS-BBでは，特にタスクサイズに逆比例のパターンでのプロセッサ利用率が悪い．これは小さなタスクが多数投入されることでプロセッサ空間に断片化が多く発生したためと考えられる．一方，DQTの場合では，プロセッサ空間の断片化が後から投入されたタスクでおさえられるため，プロセッサ利用率を高く維持できるものと考えられる．

公 平 さ

図13に，それぞれの分布についてタスクサイズの次数（$log_2(task\_size)$）とRETRの相関を縦軸に，負荷率を横軸にプロットしたグラフを示す（プロセッサ数1024）．この図から分かるように，全般的にタスクサイズとRETRは正の相関があることが分かる．DQTは負荷のアンバランスが生じた場合，下位のレベル（より小さいタスクサイズ）の負荷の軽い方を余計にスケジューリングすることでシステム全体のプロセッサ利用率の向上を目指している．このために小さいサイズのタスクがより多くのスケジューリングの機会を与えられるためと考えられる．一様分布あるいは逆比例分布において，特に低負荷時に相関が高い傾向が見られる．低負荷時では投入されるタスクが少なく，負荷の不均衡を埋めるに十分なタスクがなかったためと推測される．

正の相関は，大きなタスクほどスケジューリング上

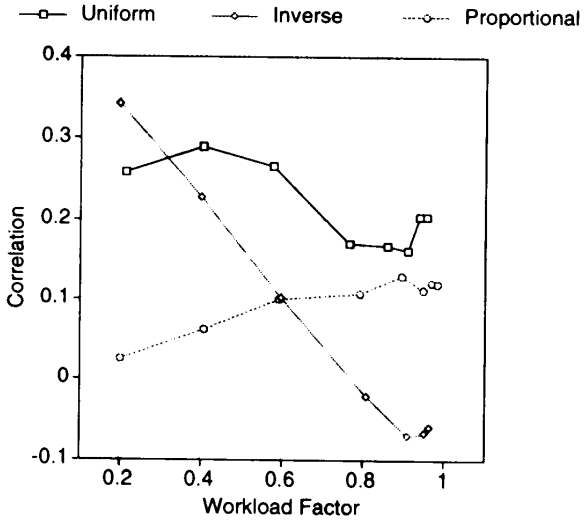


図13 タスクサイズと実実行時間比の相関（プロセッサ数：2^{10}）
Fig. 13 Task size correlation (2^{10} processors).

不利になることを意味する．これは，運用という観点から見れば，ユーザがむやみに大きなタスクを投入することを阻止する効果があると考えられ，どちらかといえば，望ましい傾向と考えられる．

ここで，公平なスケジューリングとは，同じシステムの負荷において，タスクのRETRが同じ（仮想並列マシンの時間割当率が等しいこと）であることを指す．図9に示されるようにDQTでのスケジューリングでは，同じ条件のタスクでも実行時間のバラツキが大きいという問題があり，公平さに問題があることを示す．DQTのスケジューリングでは負荷の大小によりスケジューリングの機会を変化させているためである．文献8)では，プロセッサ利用率をある程度譲性にし，より公平なスケジューリングを実現するFair-DQTを提案し，DQTとの比較を行っている．

5. 関連研究

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17)，18)などでいくつかの方式が提案されている．特に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さかんである（たとえば，文献19)）．一方，Krueger ら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[20]．文献8)において，Krueger らが提案する Scan バッチスケジューリング方式と DQT スケジューリングの負荷の違いによるプロセッサ利用率の変化を比較した．その結果，DQT スケジューリングは Scan スケジューリングと同等のプロセッサ利用率を示すことが確認された．

Tilborg と Wittie は，並列マシンのスケジューリングにおいて，耐故障性と処理のボトルネックの回避という観点から，Wave Scheduling と呼ばれる木構造のスケジューリング機構を提案している[21]．Wave Scheduling は時分割を考慮していないが，木構造のスケジューリング機構という意味で DQT と共通点がある．

一方，Feitelson と Rudolph は Distributed Hierarchical Control（DHC）と呼ばれる TSSS の一方式を提案している[9),22]．DQT の管理構造が木構造であるのに対し，DHC のそれは X-tree 構造である．X-tree 構造としたのは，タスクがフォークされることを前提にしているからである．フォークの要求は，最初，必要な大きさまで木を上下に移動し，そこから負荷分散のために水平に移動する．この水平移動のために X-tree 構造となっている．

DHC と DQT の最も大きな違いは，DHC ではスレッド（単一プロセッサでの実行）あるいはスレッド群を，DQT ではより粒度の粗いタスクをスケジューリングの単位として想定していることである．DHC で X-tree 構造を採用したのは，親子タスク間の距離がなるべく近くなるようにと意識されているからである．DQT では，タスクは与えられたパーティション内において，比較的低いオーバヘッドでスレッド群をフォークできる一方，パーティションサイズより少ないスレッド群しかフォークされなかった場合には，プロセッサ利用率が低下する．DHC ではスレッド群単位でスケジューリングするため，DQT で問題となるようなプロセッサ利用率低下を生じないが，生存期間の短いスレッドをフォークする場合にスケジューリングオーバヘッドが問題となる可能性がある．DQT ではタスク間に密な関係がないものとし，親子タスク間の距離は考慮されていない．タスクの投入はすべて木構造の根から始まる．これは，粒度の粗い（ライフタイムの長い）タスクであればボトルネックは生じないであろう，という仮定に基づいたものである．また，根からタスクを投入した方が，システム全体の負荷の均一化が比較的容易であるという理由にもよる．

DHC では木の葉に位置するコントローラ（DQT のノードに相当する）は，最終的に個々のスレッド（プロセッサ）をコントロールする必要があるが，DQT では個々のノードが「仮想並列マシン」に相当しているため，木の高さが必ずしもプロセッサの数に見合う必要がない．木の高さはシステムが提供するパーティションサイズの最大と最小から決まるシステムパラメータである．DQT においてスレッドの制御とタスクのスケジューリングは直交している．

おわりに

本稿では，時分割空間分割スケジューリングの方式のひとつである DQT を提案し，そのいくつかの特性の解析とシミュレーションによる評価を試みた．その結果，低負荷から高負荷に至るまで良好な性能を示すことが確認された．また，タスクサイズの分布による影響も比較的わずかなものであることが判明した．タスクサイズの分布の違いにおける傾向としては，タスクサイズに逆比例した場合が最も良い結果を示し，一様分布の場合が最も悪い結果を示した．スケジューリング上の公平さという面では，タスクサイズと実実行時間比に正の相関が見られ，大きいタスクほど不利なスケジューリングとなることが判明した．

今後は，本稿で述べた DQT を実装するとともに，I/O，ページングおよびデバッガとジョブスケジューリングの関係について研究を進め，並列マシン上での実用的なジョブスケジューリングの研究に貢献するつもりである．

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いただいた．RWC つくば研究所の超並列アーキテクチャ研究室の方々には多くの議論に参加していただいた．ここに感謝の意を表す．

参考文献

1) Thinking Machines Corporation: Connection Machine CM-5 Technical Summary (1992).
2) Intel Corporation: PARAGON OSF/1 User's

Guide (1993).
3) Feitelson, D.G. and Rudolph, L.: Parallel Job Scheduling: Issues and Approaches, *Job Scheduling Strategies for Parallel Processing*, Feitelson, D.G. and Rudolph, L. (Eds.), *Lecture Notes in Computer Science*, Vol.949, pp.1–18, Springer-Verlag (1995).
4) 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, 情報処理学会, pp.25–32 (1994).
5) 堀 敦史, 石川 裕, 小中裕喜, 前田宗則, 友清孝志：超並列マシンにおける時分割スケジューリング方式, 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-65, pp.33–40, 情報処理学会 (1994).
6) Hori, A., Ishikawa, Y., Konaka, H., Maeda, M. and Tomokiyo, T.: A Scalable Time-Sharing Scheduling for Partitionable, Distributed Memory Parallel Machines, *Proc. Twenty-Eighth Annual Hawaii International Conference on System Sciences*, Vol.II, pp.173–182, IEEE Computer Society Press (1995).
7) 堀 敦史, 石川 裕, Nolte, J., 小中裕喜, 前田宗則, 友清孝志：Distributed Queue Tree のシミュレーションによる解析, 並列処理シンポジウム JSPP '95, pp.313–320 (1995).
8) Hori, A., Ishikawa, Y., Nolte, J., Konaka, H., Maeda, M. and Tomokiyo, T.: Time Space Sharing Scheduling: A Simulation Analysis, Euro-Par '95 Parallel Processing, Haridi, S., Ali, K. and Magnusson, P. (Eds.), *Lecture Notes in Computer Science*, Vol.966, pp.623–634, Springer-Verlag (1995).
9) Feitelson, D.G. and Rudolph, L.: Distributed Hierarchical Control for Parallel Processing, *Computer*, Vol.23, No.5, pp.65–77 (1990).
10) Ousterhout, J.K.: Scheduling Techniques for Concurrent Systems, *Proc. of Third International Conference on Distributed Computing Systems*, pp.22–30 (1982).
11) Peterson, J.L. and Norman, T.A.: Buddy System, *Comm. ACM*, Vol.20, No.6, pp.421–431 (1977).
12) Knuth, D.E.: *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison-Wesley (1968).
13) 堀 敦史, 石川 裕, 小中裕喜, 前田宗則, 友清孝志：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 (1993).
14) 堀 敦史, 石川 裕, 坂井修一, 小中裕喜, 前田宗則, 友清孝志, 松岡浩司, 岡本一晃, 廣野英雄, 横田隆史：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 (1993).
15) Hori, A., Yokota, T., Ishikawa, Y., Sakai, S., Konaka, H., Maeda, M., Tomokiyo, T., Nolte, J., Matsuoka, H., Okamoto, K. and Hirono, H.: Time Space Sharing Scheduling and Architectural Support, *Job Scheduling Strategies for Parallel Processing*, Feitelson, D.G. and Rudolph, L. (Eds.), *Lecture Notes in Computer Science*, Vol.949, pp.92–105, Springer-Verlag (1995).
16) Feitelson, D.G. and Nitzberg, B.: Job Characteristics of a Production Parallel Scientific Workload on the NASA Ames iPSC/860, *Job Scheduling Strategies for Parallel Processing*, Feitelson, D.G. and Rudolph, L. (Eds.), *Lecture Notes in Computer Science*, Vol.949, pp.337–360, Springer-Verlag (1995).
17) Li, K. and Cheng, K.-H.: 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 (1991).
18) Zhu, Y.: Efficient Processor Allocation Strategies for Mesh-Connected Parallel Computers, *Journal of Parallel and Distributed Computing*, Vol.16, pp.328–337 (1992).
19) Chen, M.-S. and Shin, K.G.: Subcube Allocation and Task Migration in Hypercube Multiprocessors, *IEEE Trans. Comput.*, Vol.39, No.9, pp.1146–1155 (1990).
20) Krueger, P., Lai, T.-H. and Dixit-Radiya, V.A.: Job Scheduling Is More Important than Processor Allocation for Hypercube Computers, *IEEE Trans. on Parallel and Distributed Systems*, Vol.5, No.5, pp.488–497 (1994).
21) Tilborg, A.M.V. and Wittie, L.D.: Wave Scheduling — Decentralized Scheduling of Task Forces in Multicomputers, *IEEE Trans. on Comput.*, Vol.c-33, No.9, pp.835–844 (1984).
22) Feitelson, D.G. and Rudolph, L.: Mapping and Scheduling in a Shared Parallel Environment Using Distributed Hierarchical Control, *International Conference on Parallel Processing*, Vol.I, pp.1–8 (1990).

(平成 7 年 8 月 29 日受付)
(平成 8 年 2 月 7 日採録)

堀　敦史（正会員）

1979年早稲田大学電気工学科卒業．1981年同大学院理工学研究科計測制御工学専攻修士課程修了．同年（株）三菱総合研究所入社．1992年より技術研究組合新情報処理開発機構に出向．現在に至る．並列オペレーティングシステムの研究に従事．並列プログラミング言語，並列アーキテクチャなどに興味を持つ．

石川　裕

1987年慶應義塾大学理工学部電気工学科博士課程修了．工学博士．同年電子技術総合研究所入所．1988～1989年カーネギー・メロン大学客員研究員．1990年日本ソフトウエア科学会高橋奨励賞を授賞．1993年から新情報処理開発機構に出向．並列・分散システム，適応可能並列プログラミング言語/環境/処理系，リアルタイム処理，等に興味を持つ．ソフトウエア科学会，ACM，IEEE各会員．

小中　裕喜（正会員）

1987年東京大学工学部電子工学科卒業．1989年同大学院工学系研究科電気工学専攻修士課程修了．同年三菱電機（株）入社．1992年より技術研究組合新情報処理開発機構に出向．現在に至る．並列プログラミング言語の研究に従事．並列・分散処理，計算機アーキテクチャ，プログラミング環境などに興味を持つ．

前田　宗則（正会員）

1989年大阪大学大学院基礎工学研究科物理系専攻前期課程修了．同年富士通（株）入社．1992年より技術研究組合新情報処理開発機構に出向．現在に至る．並列ガーベジコレクション等の研究に従事．

友清　孝志（正会員）

1989年九州大学工学部電気工学科卒業．1991年同大学院総合理工学研究科修士課程（情報システム学専攻）修了．同年（株）東芝入社．1992～1995年技術研究組合新情報処理開発機構に出向．現在（株）東芝研究開発センター勤務．並列・分散システム，並列オブジェクト指向言語，拡張可能な並列プログラミング言語，関数型言語などに興味を持つ．