ネットワークプリエンプションによるギャングスケジューリングの実現

堀 敦史[†] 手塚宏史[†] 石川 裕[†]

我々はシステムコールのオーバヘッドを削除したユーザレベル通信と，並列処理に有効とされるギャングスケジューリングに着目し，時分割多重並列プログラミング環境を構築した．本稿では，そのような並列プログラミング環境を構築する際の問題点を明らかにし，ユーザレベル通信とギャングスケジューリングという両者の利点を最大限に活かす手法として，「ネットワークプリエンプション」を提案する．ネットワークプリエンプションとは，並列プロセス切替え時に，プロセスのコンテキストのみならずネットワークのコンテキストをも退避/復帰しようとするものである．PC クラスタ上の評価結果では，500 msec の時分割間隔において約 2% のスケジューリングオーバヘッドであることが判明した．本稿で提案されたネットワークプリエンプションは，ギャングスケジューリングだけでなく，分散プロセスの大域状態検出などへの応用が考えられる．

An Implementation of Gang Scheduling by Network Preemption

Atsushi Hori,[†] Hiroshi Tezuka[†] and Yutaka Ishikawa[†]

The goal of this research is the implementation of high-performance and easy-to-use parallel programming environment. We focus on the user-level communication technique and gang scheduling. In this paper, first we clarify some problems when implementing the user-level communication and gang scheduling, and then we propose *network preemption* that can extract the both benefits of the user-level communication and gang-scheduling. The network preemption is to save and restore network context when switching parallel processes. The proposed scheme is implemented on our PC cluster. On our evaluation on the PC cluster, gang-scheduling overhead is about 2% when the time slice interval is 500 msec. The proposed network preemption can be applied for not only gang scheduling, but also global state detection.

1. はじめに

ギャングスケジューリングは，並列計算機上で通信の頻度が高い（細粒度）並列プログラムのスケジューリング手法として有効であるとされている[6),8),16)]．また，ギャングスケジューリングは，並列ジョブの時分割スケジューリングを可能にする．時分割スケジューリング環境では，ユーザはいつでも自分のプログラムの実行を開始できるだけでなく，バッチスケジューリングに比べ短い平均応答時間，対話処理が可能といったことが期待できる．独立なローカルスケジューリングによる時分割スケジューリングも可能であるが，processor thrashing[5)]により性能が低下する可能性が指摘されている[6),8)]．しかしながら，ギャングスケジューリングの実装例は少なく，ギャングスケジューリングの実装上どのような問題があるか，ギャングスケジューリングに起因するオーバヘッドがどの程度で，何がオーバヘッドの主要因なのか，といった研究はほとんどない．

並列処理における通信の頻度は，UNIX の IP 通信で当初想定されていたものよりも桁違いに高い．またネットワークハードウェア技術の進歩により，ギガビット毎秒クラスの通信が可能になり，物理層レベルの通信遅延時間もマイクロ秒のオーダーになりつつある．このような状況において，通信のたびにシステムコールを発行することのオーバヘッドが問題視されるようになった．この問題を回避する手法として，ユーザプロセスが直接ネットワークインタフェースハードウェアを制御し，システムコールのオーバヘッドを削除することで，より高性能な通信を実現する手法（ユーザレベル通信）がいくつか提案されている[17),21)]．しかしながら，この手法を用いた並列プログラミング環境上にギャングスケジューリングによるマルチプロセス

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

環境を実現しようとした場合，ユーザプロセスがネットワークハードウェアを占有するため，プロセス切替えのつどネットワークハードウェアの状態の退避および復帰が必要となる．また，プロセスを切り替えた際，直前に走行していたプロセスの通信メッセージが，プロセス切替えにより新たにスケジューリングされたプロセスに配送される可能性がある．

我々はユーザレベル通信とギャングスケジューリングを同時に実現するために，「ネットワークのプリエンプション」と呼ばれる機能を実現した．ネットワークプリエンプションとは，個々のプロセッサに付随するネットワークインタフェースのハードウェア状態および通信ソフトウェアの状態に加え，ネットワークの状態をも退避/復帰することである．ギャングスケジューリングによるプロセスのいっせい切替え時にネットワーク全体をプリエンプションすることにより，上記の問題を解決することが可能にある．このようにして退避されたネットワークの状態は，その時点でのネットワークのスナップショットでもある．退避されたネットワークの状態を調べることで，従来実装が難しいとされているいくつかの問題，分散プロセスの大域終了検出などへの応用も考えられる．このため，ネットワークプリエンプションは，ギャングスケジューリングの実装に必要な機能としてだけでなく，並列分散処理における基本的ないくつかの問題に対する解決策を提供するものと考えられる．

ギャングスケジューリングを実装するために，我々はワークステーションクラスタおよびPCクラスタ上にPMと呼ばれるユーザレベルの低レベル通信ライブラリを開発した[19),22)]．PMは非同期かつreliableな通信モデルを提供する．低レイテンシ，高バンド幅の通信性能を確保するために，ネットワークハードウェアをユーザレベルで操作をすることを許す．さらに，PMではギャングスケジューリングを実装する際に生じる上記の問題点をすべて回避するために必要な機能が支援されている．我々は，このPMを用いてワークステーションならびにPCクラスタ上でのマルチスレッド実行時ライブラリを開発した．さらに，それらを用いたSCore-Dと呼ばれるギャングスケジューラを開発した[25)]．その結果，ユーザの並列プログラムの実行に対してユーザレベル通信の高性能を犠牲にすることなく，ギャングスケジューリングによるマルチプロセス環境の利点をユーザに提供することが可能になった．

本稿は，分散メモリ型並列計算機において，ユーザレベル通信とギャングスケジューリングの利点を両立させるためのネットワークプリエンプションを提案し，それに基づいて開発されたSCore-Dのギャングスケジューリングの基本性能の評価結果について報告するものである．

2. 用語の定義

ここでは，本稿で用いられる用語について定義する．本文中では別途，説明の順を追って適宜用語が定義されている．

(ワークステーションあるいはPC) クラスタ ワークステーションあるいはPCを高速なネットワークで接続したもの．ワークステーションクラスタとPCクラスタを特に区別しない場合，本稿では，単に「クラスタ」と表記する．以下，PCクラスタを“PCC”と略記する．

並列プロセスと(要素)プロセス 本稿ではSPMD型の並列プログラムのみを仮定している．同じSPMD並列プログラムから生成され，ノードプロセッサを指定することで互いに通信が可能なプロセス群全体を「並列プロセス」と呼ぶことにする．並列プロセスは普通のプロセスの集合であり，その要素は「要素プロセス」と呼ぶ．

3. SCore-Dの概要

我々がギャングスケジューラとして開発したSCore-D[11),25)]は，我々が構築したPCC（詳細は6.1節を参照）で動作している．PCCの各プロセッサ上ではUNIXオペレーティングシステムが動作しているものと仮定されている．SCore-Dの各要素プロセスはUNIX上でデーモンプロセスとして実現されている．SCore-Dは並列言語であるMPC++言語[14),15)]で記述されている．MPC++は，C++をベースに，遠隔関数呼び出し，待合せのための同期機構など，並列マルチスレッドのための機能が拡張されている．MPC++のスレッドは実行時ライブラリによりユーザレベルスレッドとして実現されており，高速なスレッド切替えが実現されている[24)]．

図1 にSCore-Dのプロセス構造を示す．SCore-Dはクラスタ上で動作する並列プロセスである．SCore-Dでは受信メッセージを`select()`システムコールで待つため，SCore-Dとユーザ並列プロセスはスレッドレベルでインタリーブされて並行動作することが可能になる．

図中“SCore-D Server”とあるのはユーザからのジョブの投入要求を受け付ける場所を示している．ユーザのジョブ投入要求は，UNIXのソケットにより実現さ

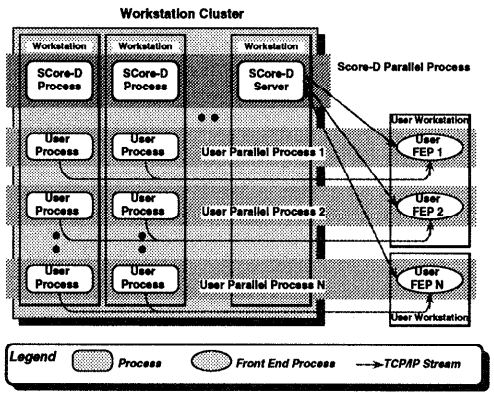


図1 SCore-D のプロセス構造
Fig. 1 SCore-D process structure.

れているため，ジョブの投入はサーバの TCP ポートに接続可能な任意の場所から可能である．ジョブ投入要求を受け付けると，並列プロセスが生成される．ジョブ投入要求を発したプロセスは "Front End Process (FEP)" と呼ばれる．起動された各要素プロセスの標準出力は FEP の標準出力に導かれる．また，FEP に対するジョブ制御は SCore-D に伝えられ，FEP のジョブの状態とクラスタ上の並列プロセスの状態はつねに同期するように制御される．

SCore-D におけるユーザ並列プロセスを管理する構造体は，並列プロセス構造体と，要素プロセス構造体，および，並列プロセス構造体と要素プロセスを結ぶ制御構造体から構成される．ユーザ並列プロセスは，その要求時にプロセッサノード数を指定することができる．要求されたプロセッサノード数がクラスタに存在するプロセッサノード数よりも小さい場合，クラスタのどのプロセッサノードを割り振るかはスケジューラの仕事になっている☆．

並列プロセス構造体は，スケジューラと同じプロセッサノードに置かれ，要素プロセス構造体は，ユーザ並列プロセスに割り当てられた各プロセッサノードに生成される．制御構造体は，ユーザ並列プロセスの制御が効率的に行えるよう，また，要素プロセスの状態変化を並列プロセス構造体に伝えるために分散されたツリー構造になっている．この分散ツリー型の制御構造体は，一種のブロードキャストおよびバリア同期の経路となっているが，要素プロセスの状態変化が同時に多発した場合，途中のノードで状態変化の通知は必要に応じてマージされ，並列プロセス構造体のあるプロセッサノードに処理が集中しないようにしている．分散ツリー構造とすることで，ユーザ並列プロセスの制御のためのブロードキャストやバリア同期は，原理的にプロセッサノード数 P に対し $\log(P)$ の時間オーダになり，スケーラビリティを確保できる．

割り当てられたプロセッサノード上にこれらの構造体を生成された後，ユーザの要素並列プロセスはそれぞれ `fork()&exec()` により SCore-D の子プロセスとして生成される．したがって，SCore-D はすべてのユーザ要素プロセスの親プロセスであり，UNIX のシグナルによりユーザプロセスを制御することが可能である．また，ユーザ要素プロセスで発生した例外シグナルによる異常終了や正常終了などのプロセスの状態変化は，対応する要素プロセス構造体に反映され，制御構造を通じて並列プロセス構造体に並列プロセスの状態変化となる．

4. ネットワークプリエンプションの実現

1 章で述べたように，ユーザレベル通信を用いてギャングスケジューリングを実現しようとした場合，通信メッセージの取扱いが問題となる．我々は高性能なユーザレベル通信とギャングスケジューリングを両立させることを目的に，低レベルの通信ソフトウェア PM[19),22)]と，PM を用いたギャングスケジューラ SCore-D[11),25)]を開発した．PM と SCore-D は互いに機能を補完することでネットワークプリエンプションを実現している．本章では，PM の機能においてネットワークプリエンプションに関連する機能について述べ，その機能を用いて SCore-D でどのようにネットワークプリエンプションを実装したかについて述べる．

4.1 PM の概要

PM は Myrinet[3)]用に開発された低レベルな通信ソフトウェアである[19),22)]．Myrinet のネットワークインタフェースには LANai と呼ばれる特殊なプロセッサとその作業領域のための SRAM が載っている[3)]．正確には，PM は LANai ファームウェア，デバイスドライバ，ライブラリから構成されている．PM は送信用の FIFO バッファと受信用の FIFO バッファおよびそれらの管理情報から構成される複数の「チャネル」を提供する．送受信用の FIFO バッファは PM のチャネルをオープンすることで，オープンしたプロセスにメモリマップされる．この結果，通信に必要なコピーの回数を減らし，それぞれのチャネルは他のプロセスによる不正アクセスから保護される．また，あるチャネルからの送信は，受信側で同じチャネルに配送される．チャネルをまたがった通信は認められていない．送信

☆ このようなスケジューラとして，たとえば "Distributed Queue Tree (DQT)" [9)]があり，すでに SCore-D に実装されている．

可能なプロセッサノードを限定することでサブネットワーク（パーティション）を設定することも可能である．SCore-D はPMの1つのチャネルを占有し，ユーザ並列プロセスは他のチャネルを使う．

ギャングスケジューラの実装では，各プロセッサで走るそれぞれのプロセスを制御かつ同期する必要から通信が必要である．このため，ギャングスケジューリングの対象となるユーザの並列プロセス内の通信とギャングスケジューリングのための通信が混在することになり，メッセージを弁別するためのなんらかの工夫が必要となる．この問題は2系統のネットワークを持つことで解決することができる．ギャングスケジューラが1系統のネットワークを用い，ユーザプロセスは別の系統のネットワークを用いればよいことになる．しかしながら，ギャングスケジューラは，ユーザのプロセス群を切り替える際にしか動作しないし，スケジューラが動作しているときにユーザプロセスは動作しない．このように双方のプロセスがネットワークを利用するパターンは排他的であるため，2系統のネットワークを用いる解決策は無駄なハードウェア投資を強いることになる．PMのチャネルはこの問題を解決する．

Myrinetではハードウェアレベルで流量制御が支援されており，メッセージの配送が保証されている．しかしながら，ハードウェアにまかせた流量制御はデッドロックの危険性があるので，PMレベルで流量制御を別途行っている．このためのプロトコルはModified Ack/Nackと呼ばれている[19),22)]．このプロトコルは，後述するギャングスケジューリングの実装に密接に関わっているので，ここでその概要について説明する．受信側でメッセージを受信したとき，そのメッセージが受信FIFOバッファに収まれば送信側にAckを返し，そうでない場合はそのメッセージを捨て，送信側にNackを返す．一方，送信側では，Ackが戻ってくるまで送信FIFOバッファ中の当該メッセージが占有するメモリ領域を解放しない．Nackが返ってきた場合に再送する必要があるからである．Ackはメッセージ領域の解放にのみ用いられる．ユーザプログラムから見た場合の送信は送信バッファにメッセージを書き込んだ時点で終了しているように見える．したがって，PMにおけるメッセージ送信はAPIレベルでは非同期である．

このプロトコルには以下に述べるような性質がある．

チャネルの独立性 なんらかの理由により受信側のFIFOバッファが満杯になった場合，受信側はつねにNackを送信側に返す．Nackを受けとった送信側は通信メッセージを再送する．この手順は受信側が通信メッセージを消費し，受信バッファに十分な空きができるまで繰り返される．この結果，送信バッファが満杯になり，ユーザは通信メッセージを送ることができなくなる可能性がある．このような状況は，一種の閉塞状態に見えるが，実際にはネットワーク中には再送メッセージが流れている．このため，あるチャネルがこのような状態に陥っても，他のチャネルの通信メッセージはネットワーク中を流れることができ，正常に通信可能である．

SCore-D はユーザレベルで実現されたオペレーティングシステムと見なすこともできる[11)]．その意味でPMのチャネル独立性は重要である．なぜなら，ユーザプログラムのバグによりネットワークが閉塞してもSCore-D並列プロセス内の通信は可能であり，そのような状態に陥ったユーザ並列プロセスをkillするなどの処理が可能であるからである．しかしながら，現在のPMではチャネル間の保護は完全でないため，SCore-D の健全性は完全ではない．

チャネルの定常状態 あるチャネルについて，そのチャネルから送出された通信メッセージに対応するすべてのAckあるいはNackを受けとった時点で，そのプロセッサノードから送信されたAck/Nackを含むすべてのメッセージはネットワーク中に存在しない．この状態を「そのプロセッサノードの指定されたチャネルが定常状態にある」と呼び，PMはチャネルが定常状態にあるか否かを検出する機構を提供している．もし，あるユーザ並列プロセスに割り当てられた（サブ）ネットワークのすべてのプロセッサノードにおいてこの条件が満たされるならば，そのユーザ並列プロセスから送信されたいかなるメッセージも（サブ）ネットワーク中に存在しないことになる（証明は自明なので省略する）．このような状態をここでは「（サブ）ネットワークの定常状態」と呼ぶことにする．ただし，この状態の検出にはPMより上位の機構が必要である．

この他にPMでは，i) 可能な限り低レイテンシ，高スループットであること，ii) PMがネットワークハードウェアを操作中であっても，任意の時点でプロセスが停止し，PMの状態が退避されても，後に状態が復帰されることで，通信を含むプロセスの実行は正常に再開されること，という点にも注意して設計されている．PMの詳細については文献19), 22)を参照され

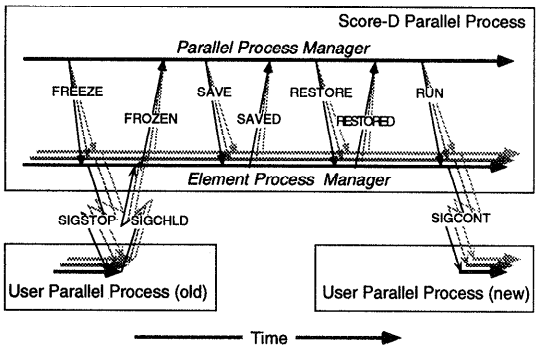


図2 並列プロセス切替え
Fig. 2 Parallel process switching.

たい.

我々がMyrinetを用いた理由はクラスタの通信性能をできる限り高性能にするためである. Myrinetの使用は以下に提案するギャングスケジューリングにおいて必要条件ではない.

4.2 ネットワークのプリエンプション

SCore-Dでは, **図2** に示した手順で並列プロセスの切替えを行う. この図の横軸方向は時間軸になっており, イベントの発生順序が示されている. 並列プロセス構造体を管理する並列プロセスマネジャ(図中, Parallel Process Manager)と, 個々の要素プロセス構造体を管理する要素プロセスマネジャ(図中, Element Process Manager)がある. これらは, 3章で説明したようにツリー型制御構造により関連付けられているが, 図中ツリー構造は省かれている. また, 図には記述されていないが, 実際にはスケジューラも存在している.

まず, 走行しているユーザ並列プロセスを止める場合の手順について説明する(図2左半分).

(1) スケジューラが走行中のプロセスの停止を決断すると, 並列プロセスマネジャが, すべてのユーザ要素プロセスに対し`SIGSTOP`シグナルを送るよう指示する(図2のFREEZEメッセージ).

(2) SCore-Dの各要素プロセスマネジャは`SIGSTOP`シグナル送出後, `SIGCHLD`によりユーザプロセスの停止を知る. この後, ユーザプロセスが使用しているPMのチャネルが定常状態に陥るのを待つ.

(3) 定常状態になった旨, サーバプロセスに通知する(図2のFROZENメッセージ). このメッセージは, 制御構造においてバリア同期がとられる. その結果, 並列プロセスマネジャに通知された時点で, ユーザ並列プロセスに関与するすべてのプロセッサノードで定常状態, つまりネットワーク中にユーザ並列プロセスのメッセージが存在せず, いつでもネットワークの状態が退避可能になったことを示す.

(4) 並列プロセスマネジャはすべての要素プロセスマネジャに対し, ユーザが使用していたチャネルの状態を退避するよう指示する(図2のSAVEメッセージ).

(5) 各要素プロセスマネジャはユーザのチャネルの状態をメモリに退避する. 退避完了は並列プロセスマネジャに通知される(図2のSAVEDメッセージ).

(6) ユーザ並列プロセスマネジャですべてのプロセッサノードでのチャネルの状態退避を確認して, ユーザ並列プロセスの停止を完了する.

逆に, 停止している並列プロセスの実行を再開する場合は以下の手順となる(図2右半分).

(1) スケジューラがユーザプロセスの再開を決断し, 並列プロセスマネジャが各プロセッサノードに対しチャネルの状態を復帰するよう指示を出す(図2のRESTOREメッセージ).

(2) 各要素プロセスマネジャは各々チャネルの状態を復帰し, 復帰完了通知はバリア同期される. 同期の完了は並列プロセスマネジャに通知される(図2のRESTOREDメッセージ).

(3) 並列プロセスマネジャはユーザ並列プロセスの走行開始を指示する(図2のRUNメッセージ).

(4) 各プロセッサノードでは, ユーザプロセスに対し`SIGCONT`シグナルを発する.

以上の操作, ネットワークプリエンプションとすべての要素プロセスのコンテキスト切替えにはFREEZE, SAVE, RESTOREそしてRUNメッセージのブロードキャストを4回, FROZEN, SAVEDそしてRESTOREDのバリア同期を3回行うことになる. これらのブロードキャストおよびバリア同期は前述した分散ツリー構造に沿って行われるため, 原理的にプロセッサ数 P に対し $\log(P)$ の時間オーダで済む. 実際のfreeze, save, restoreの処理時間は, PMのチャネルの状態に含まれるメッセージ数と総量, つまりユーザ並列プロセスの通信パターンに依存する.

このギャングスケジューリングの手順において, すべてのプロセッサノードで定常状態になるということ(FROZEN)は, ネットワークの状態が各プロセッサノードにおけるPMのチャネルの状態に反映されることを意味する. また, FROZEN状態に続く各プロセッサノードでのネットワークハードウェアの状態お

よびそれに付随するソフトウェア的な状態の保存は，ネットワークそのものの状態を保存していることになる．したがって，我々はこれを「ネットワークのプリエンプション」あるいは「ネットワークコンテキストの退避/復帰」と呼んでいる．

5. 関連研究

Thinking Machines 社の CM-5 ではギャングスケジューリングが実装されている．CM-5 では，NI と呼ばれるネットワークハードウェアをユーザのアドレス空間にマップすることで，ユーザが直接ネットワークハードウェアを操作することを許している[20]．通信メッセージの問題を回避するために，“All-Fall-Down”と呼ばれるハードウェア機構がネットワークに備わっている．特殊レジスタを操作することでネットワークは all-fall-down モードになり，その時点でネットワーク中に存在する通信メッセージはすべて，その宛先に関係なく，最寄りのプロセッサに転送される．プロセス切替え時には，最初にネットワークを all-fall-down モードにし，次に各プロセッサ上では all-fall-down の結果として到着した通信メッセージをメモリに退避する．新しいプロセスが起動される直前に，退避された通信メッセージは再びネットワークに注入される．この方法では，通信メッセージの順序を保存することができないが，CM-5 のネットワークでは動的ルーティングを行っているので，通信メッセージ順序が保存されないことは問題にはならない．換言すれば，通信メッセージの順序に意味はないことになり，all-fall-down はネットワークプリエンプションのためのハードウェア機構であるということができる．しかしながら，多くのアプリケーションで通信メッセージ順序の保存を必要とする場合が多く，また，メッセージの最大長も 5 語と短いため，packetizing 処理の負担がかかる．All-fall-down のようなハードウェア支援があるにもかかわらず，CM-5 のギャングスケジューリングのオーバヘッドについては “CM-5 incurrs a minimum overhead of 4 ms, with typical times closer 10 ms.” という報告がある[4]．

Franke らは，IBM 社の SP-2 上に SHARE と呼ばれるギャングスケジューラを実装している[7]．SHARE においてギャングスケジューリング直後で通信メッセージが誤って配送されるという問題は，SP-2 が unreliable な同期送信通信モデルを提供しているため，比較的容易に回避できる．すべての通信メッセージには通信ハードウェアにより並列プロセス ID に対応するタグが付けられているため，プロセス切替えにより誤ったプロセスに配送されたメッセージを検出することが可能である．プロセス切替え時には，通信ハードウェアのコンテキストは退避/復帰するが，その時点でネットワーク中に存在するメッセージはメッセージを受けとった側で失敗を返すことで対処している．このため，上位のプロトコル階層において reliable なプロトコルを実装しなければならず，その分のオーバヘッドが並列処理効率の低下を招く．この方式は，並列プロセス切替え時に，ネットワークの状態を完全には退避しないが，それにより失われた情報は復帰時の再実行により回復するというやり方である．一方，SCore-D の実現方式は，並列プロセス切替え時にネットワークの状態を完全に退避/復帰しようとするものである．SCore-D で実装されたネットワークプリエンプションは，ハードウェアが保証する reliable な通信という性質をそのまま保存し，ユーザ並列プロセス内の通信のオーバヘッドをできる限り低減させることを目的として設計されている．この結果，ネットワークの状態を完全に退避/復帰するための余分な処理が必要となる．

また，Franke らの SHARE の実装において，ギャングスケジューリングのために同期機構を特に設けず，各プロセッサの時計が NTP プロトコルにより同期していることを前提に，スケジューリングをタイマで同期させることでギャングスケジューリングを実現している[7]．NTP による時刻合わせは基本的に誤差を許容している．このため，各プロセッサ上のウォールクロックの時刻のずれがもたらすギャングスケジューリングのずれ，co-scheduling skew[1]，が問題になると思われる．このためか，文献 7) では時分割スケジューリングのための時分割間隔は分のオーダが想定されている．

Intel 社製の Paragon では，PM と同様，非同期メッセージ送信における通信メッセージの送達確認機能が提供されている[13]．しかしながら，この機能だけではギャングスケジューリングを正しく実装することはできない．ここであるユーザ並列プロセスのあるプロセスが計算等に忙しく，その結果受信メッセージの処理が滞り，受信メッセージバッファが満杯な状況を想定する．ネットワークのプリエンプションのために，各プロセッサ上で送信メッセージの送達完了を待つとする．受信バッファが満杯のノードに対する通信メッセージの送達確認は，受信メッセージが消費されないので永遠に終了しない．SCore-D では，メッセージの送達確認ではなく，最下位プロトコル層（PM）でネットワーク中のメッセージの有無を確認することにより，この問題を回避している．

表1 PCクラスタの仕様
Table 1 PC cluster specification.

System	Number of Nodes	32
Node	Processor	Pentium
	Clock [MHz]	166
	Memory [MB]	64
	I/O Bus	PCI
	OS	NetBSD
Network	Bandwidth [MB/s]	160

一方，クラスタを並列計算専用とせずに，ワークステーションと分散メモリ型並列計算機の中間に位置付け，その上にギャングスケジューリングを実現しようとするものもある[18]．この場合のギャングスケジューリングは，単一の並列プログラムの実行プロセスとワークステーションで通常実行される普通のプロセスを対象とする．我々はワークステーションクラスタを分散メモリ型並列計算機の一形態と位置付けている．このため我々は普通の逐次プロセスの存在を仮定せず，複数の並列プロセスのみを対象としている．

SiliconGraphics社のIRIXオペレーティングシステムにおいてもギャングスケジューリングが実装されている[2]．しかしながら，対象としている並列計算機は共有メモリ型であるため，我々が想定している分散メモリ型並列計算機とでは通信の形態が異なる．本稿ではワークステーションクラスタを含む分散メモリ型の並列計算機のみを対象とする．

6. 評　価

6.1 計測環境

表1に我々が構築したPCC[23]（**図3**）についての主要な諸元を示す．以下の計測はすべてPCCを用いた．ネットワークはMyrinetを用いている．

6.2 PMの基本性能

次に，これらのクラスタにおけるPMの通信性能を**表2**に示す．表1に示したネットワークハードウェアのバンド幅よりもPMのバンド幅が小さい値になっているのは，PCIバスのバンド幅の制約による．

表3は，PM単体での1つのチャネルコンテキストの退避および復帰に要する時間を示したものである．チャネルコンテキストの退避とは，具体的には，LANaiのメモリの状態はPCIバス経由で，送受信FIFOバッファの内容はそのまま，退避領域のメモリ空間にコピーすることである．現在のPMのバッファの大きさは，送信FIFOバッファには最大512メッセージ，メッセージ総量の最大48 KB，受信FIFOバッファには最大4,096メッセージ，メッセージ総量の最大64 KBとなっている．退避の時間の方が長いのは，プロセッサから見たI/Oデバイスからの読み込み速度が書き速度よりも遅いためである．

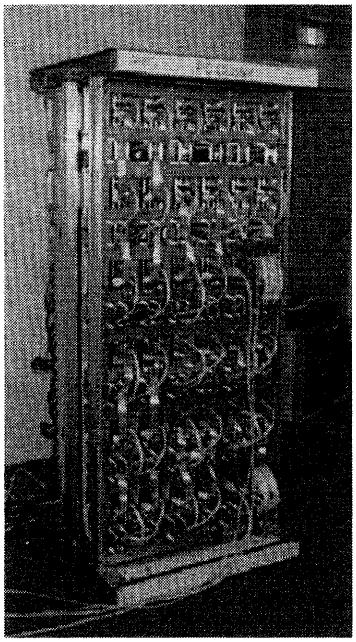


図3 PCクラスタ
Fig. 3 PC cluster.

表2 PMの基本性能
Table 2 PM performance.

Latency [μs]	Bandwidth [MB/s]
7.2	117.6

表3 チャネルコンテキストの退避/復帰時間
Table 3 Channel context saving and restoring times.

Buffer Status		Save	Restore
Send	Receive	[msec]	[msec]
Empty	Empty	0.94	0.37
Full	Empty	2.40	1.84
Empty	Full	4.66	2.96
Full	Full	6.12	4.43

退避/復帰の時間は，コンテキストに含まれる通信メッセージの数および通信メッセージの総量に依存する．通信メッセージの数が多いほど，また，通信メッセージの総量が多いほど，退避/復帰に要する時間は長くなる．通信メッセージ総量の最大値は，PMのチャネルに割り当てられたバッファの大きさで規定される．より大きなバッファは通信性能に貢献するが，ネットワークコンテキストの退避/復帰により時間がかかることになる．この関係はプロセッサのレジスタファイルの大きさとプロセスの切替え速度の関係によく似ている．より大きなレジスタファイルは，個々のプロセ

スの計算速度に貢献するが，プロセスコンテキストの退避/復帰が遅くなる．

6.3 ギャングスケジューリングの性能

以下，SCore-Dにおけるギャングスケジューリングの性能（スケジューリングオーバヘッド）について計測した結果を示す．ここでギャングスケジューリングのオーバヘッドには，並列プロセスの個々の要素プロセスのコンテキスト切替え時間と，ネットワークのコンテキスト切替え時間が含まれる．実際のSCore-Dのギャングスケジューリングでは，図2に示すように，この両者の境が明確でなく，また，UNIXにおいてはシグナル `SIGCONT` 配送結果の確認ができないため，以下では両者を含んだギャングスケジューリングのオーバヘッドの計測結果について述べる．計測に用いたテストプログラムは2種類用意した．1つは，個々のプロセッサノードが独立に計算し（実際には空ループを回る）通信はまったく行わないものと，もう1つはbinary shuffle型のバリア同期のみを繰り返すものである．前者の計測プログラムをEPと呼び，後者のプログラムをBARと呼ぶことにする．これらのプログラムは，計測中にページングを引き起こさない程度に十分小さい．また，特に影響の大きいと思われる `update` デーモンはすべてのプロセッサノード上で停止させて計測した．

ギャングスケジューリングのオーバヘッドは，図2に基づき，i) ユーザ並列プロセスを `SIGSTOP` で止め，ネットワークが定常状態になるまでの時間，ii) ネットワークコンテキストを退避する時間，iii) 次のユーザ並列プロセスをスケジューリングする時間，iv) 新しいユーザ並列プロセスのネットワークコンテキストを復帰する時間，v) `SIGCONT` でユーザ並列プロセスが実際に走行を開始するまでの時間，に分解することができる．ネットワークが定常状態になる時間はネットワークに存在する通信メッセージの数に依存する．また，表3に示したように，ネットワークコンテキストの退避および復帰の時間はコンテキスト切替え時にネットワークに割り当てられたバッファ中に存在する通信メッセージの数，量と，ネットワーク中に存在する通信メッセージの数，量に依存する．EP計測プログラムは通信をまったく行わないことから，ギャングスケジューリングオーバヘッドの通信メッセージの数，量に依存しない部分のオーバヘッドをEPにより計測することができる．オーバヘッド以外のギャングスケジューリングの性能を示す重要な指標としてco-scheduling skewがある．このco-scheduling skewの時間はBAR計測プログラムを用いることでオーバヘッドに組み入れることが可能である．また，BAR計測プログラムでは通信が頻発するため，EP計測プログラムと比較することで通信メッセージの有無がチャネルの定常状態の待ち時間，状態退避/復帰へ与える影響を調べることができる．

ギャングスケジューリングのオーバヘッド（O）は，ギャングスケジューリングなしの場合の計測プログラムの実行時間（T_{NoGang}）とギャングスケジューリング下での実行時間（T_{Gang}）から次式により求めた．

$$O = (T_{Gang} - T_{NoGang})/T_{NoGang}$$

ここで計算された値は T_{NoGang} と T_{Gang} の時間差が少ない場合，計測誤差の影響を受けやすいことに注意を要する．T_{NoGang} の時間は，スケジューリングをまったく行わないスタンドアローンなMPC++実行時ライブラリをリンクした実行ファイルをクラスタ上で実行した結果である．実行時間はすべて計測に用いたプログラムの内部で計測した経過時間（Elapsed Time）である．このとき，SCore-D並列プロセスは走行していないため，SCore-D並列プロセスに関係するいかなる影響も T_{NoGang} に含まれていない．T_{Gang} の計測においては，計測プログラムはジョブとしてSCore-Dに1つだけ投入される．SCore-Dは次にスケジューリングすべきプロセスが，プロセス切替え前に走行しているプロセスと同じかどうかというチェックをしていない．このため1つしかユーザ並列プロセスがないにもかかわらずSCore-Dは時分割間隔時間が経過するたびに並列プロセス切替えを行う．

図4は，EP計測プログラムのオーバヘッドをプロセッサノードの数を変化させて計測したものである．通信メッセージが存在しないこと，co-scheduling skew

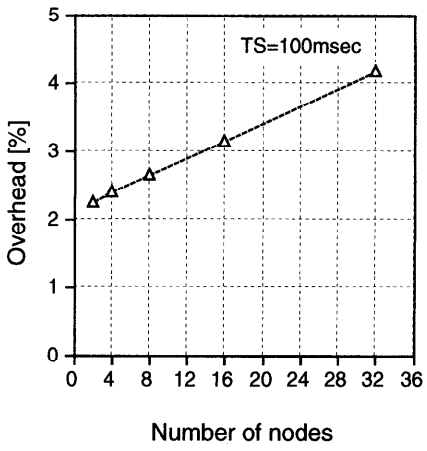


図4 ギャングスケジューリングオーバヘッド（EP）
Fig. 4 Gang scheduling overhead (EP).

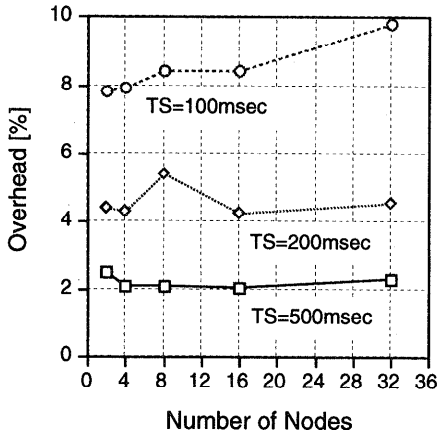


図5 ギャングスケジューリングオーバヘッド（BAR）
Fig. 5 Gang scheduling overhead (BAR).

の効果が現れないこと，という理由により図4で示されるオーバヘッドの大半がSCore-Dの並列プロセスに対する制御に要する時間からくるものと考えることができる．ここでは，誤差の影響を少なくするために，時分割間隔は最小の100 msecに設定した．ここで時分割間隔とは，時分割スケジューリングサイクルの間隔である．一方，**図5**にはBAR計測プログラムについて，時分割間隔（TS）を100，200，500 msec，プロセッサノードの数を2，4，8，16，32と変化させて計測した結果を示す．縦軸はオーバヘッド，横軸はプロセッサノードの数である．

図4に比べ図5のグラフが凸凹しているのは，より長い時分割間隔でのオーバヘッドが小さいことによる計測誤差の影響と思われる．図4では，ほぼプロセッサノード数に比例してオーバヘッドが増加しているが，図5では時分割間隔が長い場合に増加の割合が少なくなっている．図4でプロセッサノード数が2の場合と32の場合のオーバヘッドの差は約2%であり，この差は図5における時分割間隔100 msecの場合とほぼ同じである．

プロセッサノード数に依存すると考えられるオーバヘッドは，主にユーザ並列プロセスを制御するためのブロードキャストとバリア同期のための処理時間である．制御に関しては計測プログラムの通信パターンの影響を受けない．一方，通信パターンは実際のfreeze，saveおよびrestoreの処理時間に影響を及ぼす．通信メッセージの量に対する影響は無視できない．表3にあるように，送受信FIFOバッファともに満杯のときは，バッファが空のときに比べ，チャネルコンテキストの退避/復帰時間の合計約9 msec増え，時分割間隔100 msecの場合で合計12 msecになる可能性がある

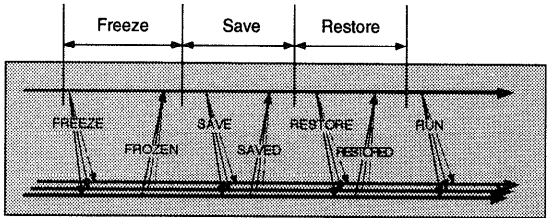


図6 オーバヘッド内訳の計測箇所
Fig. 6 Overhead measurement.

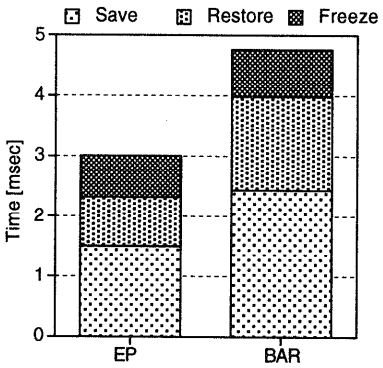


図7 オーバヘッドの内訳
Fig. 7 Overhead breakdown.

ことを示唆している．

6.4 オーバヘッドの内訳

ここではギャングスケジューリングオーバヘッドの内訳を計測した結果について述べる．計測は，図2に対応した**図6**の各時点に相当する箇所（並列プロセスマネジャ）に計測ルーチンを埋め込んで行った，この図にはないが，実際にはスケジューリング時間，saveの終了からrestoreの開始までの時間も計測した．しかしながら，今回の計測の場合，ユーザ並列プロセスが1つしかなく，単純なラウンドロビンのスケジューリングであるため，他の処理時間に比べ十分に無視できる値（数 μsec）であったので省略した．

図7はその計測結果である（32プロセッサノード）．図の左側にネットワーク退避（図中，Save）と復帰（図中，Restore）に要する時間を，図の右側にはユーザ並列プロセスをシグナルにより止め，ネットワークが定常状態になるまで（図中，Freeze）の時間をそれぞれ示す．計測プログラムはEPとBARを用い，プロセッサノードの数は32，時分割間隔は200 msecである．その結果，freezeに要する時間は1 msec以下であり，EPとBARのfreeze処理時間の差は約80 μsecであった．ことから，ネットワーク中のメッセージの存在がfreeze処理時間に与える影響は，コンテキスト

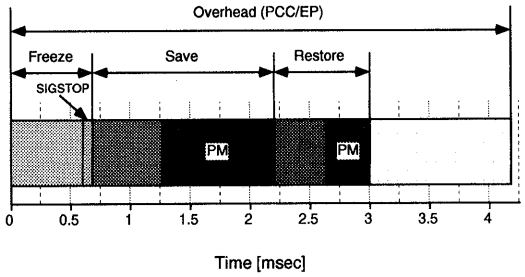


図8 オーバヘッドの詳細内訳（EP）
Fig. 8 Detailed overhead breakdown (EP).

の退避/復帰に要する時間に比べ，十分小さいことが分かる．

これまでの計測結果をまとめる意味で，EP 計測プログラムにより計測した各種のオーバヘッドを 1 つのグラフにしたものが**図 8** である．この棒グラフは，32 プロセッサノード，時分割間隔 100 msec の計測条件における図 7，図 4，および表 3 のそれぞれの値を重ね，別途計測された SIGSTOP 配送の平均時間（0.08 msec）を組み入れたものである．棒グラフの中に “PM” とあるのは，表 3 における送受信バッファが両方とも Empty のときの値に相当する部分である．

図 7 の各処理時間の合計から推測されるギャングスケジューリングのオーバヘッドは，図 4，図 5 で計測された値よりも小さい．これは，SCore-D のスレッドが走る時間，SCore-D が与えるキャッシュへの影響，`SIGCONT` の配送時間，などが考えられる．図 8 から，freeze，save，restore の処理時間の合計におよそ 3 msec 要することが分かるが，アプリケーション側で計測したオーバヘッド（図 4）ではおよそ 4 msec（100 msec の時分割間隔で 4% のオーバヘッド）となっていることから，図 7 に現れないオーバヘッドが約 1 msec 程度であると推定される．

この隠れたオーバヘッド（1 msec）がアプリケーションによらず一定であると仮定して，図 5 に含まれる co-scheduling skew がどの程度かを推測してみる．図 7 における BAR 計測プログラムの並列プロセス切替え処理時間はおよそ 5 msec である．一方，図 5 のオーバヘッドは約 10 msec であるから，隠れたオーバヘッドの 1 msec を除いた残り約 4 msec が co-scheduling skew により生じた時間である可能性がある．

実際のアプリケーションにおいて BAR プログラムほど頻繁に同期をとるものはないことを考え，また BAR プログラムで計測されたオーバヘッドに co-scheduling skew が含まれているとするならば，他の実際的なアプリケーションにおいても BAR 程度のスケジューリングオーバヘッドであろうと予測される．

7. 考　　察

ギャングスケジューリングオーバヘッドの大半がチャネルコンテキストの退避/復帰時間であることに対しては，単純には PM の送受信バッファのサイズを減らせば改善されることは明らかである．PM の現在の実装において，Myrinct の LANai のメモリが許す限り最大のバッファを持ち，通信性能を極限まで引き上げることが第 1 目標になっている．この結果，チャネルコンテキストの退避/復帰の性能は犠牲になっている．どの程度の大きさの送受信バッファが最適なのかは難しい問題である．PCC のネットワークプリエンプションでは，I/O バスを経由したネットワークハードウェアの状態のメモリへの退避と，送受信バッファの退避領域へのコピー時間が大きな割合を占めている．最近のアーキテクチャではメモリバンド幅が重視される傾向にあり，そのようなハードウェアを用いることで改善が期待される．SCore-D では基本的にすべてソフトウェアでギャングスケジューリングを実現しているが，文献 12) にはネットワークプリエンプションをハードウェアで支援する実現方法が提案されており，このようなハードウェア支援によりいっそうの高速化が期待できる．

PM が一度に走行する並列プロセスの数だけのチャネルを提供すればネットワークプリエンプションはまったく不要である，という議論は正確さを欠く．なぜならば，並列プロセスが終了したり kill された場合，その並列プロセスのメッセージがネットワーク中に存在してしないことが確認されない限り，そのチャネルは再利用できないからである．少なくとも SCore-D における freeze と同様の処理が必要である．また，PM のチャネルは Myrinct インタフェース上の貴重な SRAM（256 KB）の大部分を占有している．より多くのチャネルを実装することはそれぞれのチャネルに割り振られるバッファ領域が小さくなることを意味し，その分通信性能が低下する．

評価では 100～500 msec 程度を時分割スケジューリングの時分割間隔として用いた．これらの値は UNIX で設定されている時分割間隔よりも大きい．UNIX では対話処理プログラムが多く，ユーザからの 1 文字入力に対してどの程度の応答時間が確保されるかが重要視されている．しかしながら，1 文字をエコーバックするのに何十台ものプロセッサを割り当てるのは無意味である．我々は並列対話処理における「対話処理の粒度」は逐次の場合よりも大きいと仮定し，100～

1,000 msec 程度の時分割間隔でも対話処理は可能と考えている．時分割間隔 500 msec のときの BAR 計測プログラムにおけるオーバヘッドは 2% 程度であり，応答速度を含めて十分実用になる範囲と考える．実際に我々は対話並列プログラムを構築し，SCore-D 上で走らせているが，対話処理の応答性に問題は見つけられなかった．

本稿で提案しているネットワークのプリエンプションの応用範囲は広い．この応用の 1 つとして並列プロセスの大域的状態検出がある[10]．互いに通信し合う分散プロセスの大域状態の検出は "Distributed Termination Problem" としてよく知られる問題である．ネットワークコンテキストはネットワークのスナップショットでもあり，コンテキストを調べることで並列プロセスの大域状態が検知可能である[10]．また，ギャングスケジューリングによる対話的な並列プログラムの場合，大域的なアイドルや終了検出機構は必須であると考えられる．この機能はすでに SCore-D に実装されており[10]，6.3 節に示した値には，並列プロセスの大域状態検出機構に起因するオーバヘッドも含まれている．

8. まとめ

本稿では，高性能なユーザレベル通信においてギャングスケジューリングを実現する際の問題点を明らかにし，その解決策としてネットワークプリエンプションを提案し，クラスタ上での実装を通じてその検証と評価を行った．

ギャングスケジューリングの実現は通信方式に大きく依存する．本稿で提案されたネットワークプリエンプションは，ユーザ並列プロセスに高性能な通信を提供すると同時に，ギャングスケジューリングを実現する．ネットワークプリエンプションは，CM-5 における all-fall-down ですでに実現されている方式であるが，本稿での提案は，特殊なハードウェア支援のないソフトウェアだけの実現であること（Myrinet という通信ハードウェアを用いてはいるが，これは性能を確保するためであって，ギャングスケジューリング実現のために必須ではない），通信メッセージの順序をも保存するという意味でより一般化されていること，という点で CM-5 の all-fall-down と異なっている．

評価の結果として，PCC では時分割スケジューリングの時分割間隔が 500 msec の場合，ギャングスケジューリングがユーザ並列プロセスに与えるスケジューリングオーバヘッドは 32 プロセッサノードの場合で 2% 程度であることが判明した．またそのオーバヘッドの多くがネットワークの状態の退避/復帰の時間であることを示した．

このオーバヘッドの値が，クラスタ上で必要十分に小さい値であるかどうかを客観的に示すことは難しい問題である．7 章で述べたようにプロセッサなどのハードウェアの性能向上が，オーバヘッドをさらに低減させることは期待できる．我々は本稿で示したギャングスケジューリングの実現手法が有効であり，十分実用的になりうると確信している．

現在，より大規模な PC クラスタの上で，より実用的なアプリケーションを用いて SCore-D のギャングスケジューリングの性能を評価を進めている．さらに，クラスタ上のギャングスケジューラとしての SCore-D をより機能拡充し，ユーザレベルの並列オペレーティングシステムとする方向で研究を発展させるつもりである[11]．

参考文献

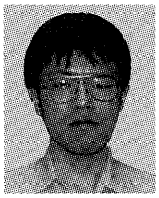
1) Arpaci, R.H., Dusseau, A.C., Vahdat, A.M., Liu, L.T., Anderson, T.E. and Patterson, D.A.: The Interaction of Parallel and Sequential Workloads on a Network of Workstations, UC Berkeley Technical Report, CS-94-838, Computer Science Division, University of California, Berkeley (1994).
2) Barton, J.M.: A Scalable Multi-Dicipline, Multiple-Processor Scheduling Framework for IRIX, *Job Scheduling Strategies for Parallel Processing*, Feitelson, D.G. and Rudolph, L. (Eds.), LNCS, Vol.949, pp.45–69, Springer-Verlag (1995).
3) Boden, N.J., Cohen, D., Felderman, R.E., Kulawik, A.E., Seitz, C.L., Seizovic, J.N. and Su, W.-K.: Myrinet: A Gigabit-per-Second Local Area Network, *IEEE Micro*, Vol.15, No.1, pp.29–36 (1995).
4) Burger, D.C., Hyder, R.S., Miller, B.P. and Wood, D.A.: Paging Tradeoffs in Distributed-Shared-Memory Multiprocessors, *Supercomputing'94*, pp.590–599 (1994).
5) Feitelson, D.G. and Rudolph, L.: Distributed Hierarchical Control for Parallel Processing, *COMPUTER*, Vol.23, No.5, pp.65–77 (1990).
6) Feitelson, D.G. and Rudolph, L.: Gang Scheduling Performance Benefits for Fine-Grain Synchronization, *Journal of Parallel and Distributed Computing*, Vol.16, No.4, pp.306–318 (1992).
7) Franke, H., Pattnaik, P. and Rudolph, L.: Gang Scheduling for Highly Efficient Distributed Multiprocessor Systems, *Frontier'96*,

pp.1–9 (1996).
8) Gupta, A., Tucker, A. and Urushibara, S.: The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications, *ACM SIGMETRICS*, pp.120–132 (1991).
9) Hori, A., Ishikawa, Y., Konaka, H., Maeda, M. and Tomokiyo, T.: A Scalable Time-Sharing Scheduling for Partitionable, Distributed Memory Parallel Machines, *Proc. 28 Annual Hawaii International Conference on System Sciences*, Vol.II, pp.173–182, IEEE Computer Society Press (1995).
10) Hori, A., Tezuka, H. and Ishikawa, Y.: Global State Detection using Network Preemption, *IPPS'97 Workshop on Job Scheduling Strategies for Parallel Processing*, Feitelson, D.G. and Rudolph, L. (Eds.), LNCS, Vol.1291, pp.262–276, Springer-Verlag (1997).
11) Hori, A., Tezuka, H. and Ishikawa, Y.: User-level Parallel Operating System for Clustered Commodity Computers, *Proc. Cluster Computing Conference '97* (1997).
12) 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.), LNCS, Vol.949, pp.92–105, Springer-Verlag (1995).
13) Intel Corporation: *PARAGON OSF/1 USER'S GUIDE* (Apr. 1993).
14) Ishikawa, Y.: Multi Thread Template Library – MPC++ Version 2.0 Level 0 Document, Technical Report, TR-96012, RWC (1996).
15) Ishikawa, Y., Hori, A., Tezuka, H., Matsuda, M,. Konaka, H., Maeda, M., Tomokiyo, T. and Nolte, J.: MPC++, *Parallel Programming Using C++*, Wilson, G.V. and Lu, P. (Eds.), pp.429–464, MIT Press (1996).
16) Ousterhout, J.K.: Scheduling Techniques for Concurrent Systems, *Proc. 3rd International Conference on Distributed Computing Systems*, pp.22–30 (1982).
17) Pakin, S., Lauria, M. and Chien, A.: High Performance Messaging on Workstations: Illinoi Fast Messages (FM) for Myrinet, *Supercomputing'95* (1995).
18) Sobalvarro, P.G. and Weihl, W.E.: Demand-Based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors, *Job Scheduling Strategies for Parallel Processing*, Feitelson, D.G. and Rudolph, L. (Eds.), LNCS, Vol.949, pp.106–126, Springer-Verlag (1995).
19) Tezuka, H., Hori, A., Ishikawa, Y. and Sato, M.: PM: An Operating System Coordinated High Performance Communication Library, *High-Performance Computing and Networking*, Hertzberger, P.S.B. (Ed.), LNCS, Vol.1225, pp.708–717, Springer-Verlag (1997).
20) Thinking Machines Corporation: *NI Systems Programming*, Version 7.1. (Oct. 1992).
21) von Eicken, T., Basu, A. and Vogels, W.: U-Net: A User Level Network Interface for Parallel and Distributed Computing, *15th ACM Sumposium on Operating Systems Principles*, pp.40–53 (1995).
22) 手塚，堀，石川：ワークステーションクラスタ用通信ライブラリ PM の設計と実装，並列処理シンポジウム JSPP'96，pp.41–48，情報処理学会 (1996).
23) 手塚，堀，石川，曽田，原田，古田，山田：PC とギガビット LAN による PC クラスタの構築，計算機アーキテクチャ研究会資料，96-ARC-119，pp.37–42，情報処理学会 (1996).
24) 堀，手塚，石川，高橋，曽田，堀川，小中，前田：マルチスレッド言語のための実行時ライブラリの実装，計算機アーキテクチャ研究会資料，96-ARC-117，pp.37–42，情報処理学会 (1996).
25) 堀，手塚，石川，曽田，小中，前田：並列プログラム実行環境のワークステーションクラスタ上での実装，並列処理シンポジウム JSPP'96，pp.49–56，情報処理学会 (1996).

(平成 9 年 4 月 25 日受付)
(平成 10 年 7 月 3 日採録)

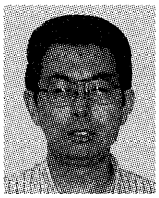
堀　敦史（正会員）

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．JSPP'98 最優秀論文賞受賞．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．



手塚 宏史（正会員）

1980年東京大学教養課程中退．1981年（株）生活構造研究所入社．1985年ソニー（株）入社．1988年（株）ソニーコンピュータサイエンス研究所入社．1990年ソニー（株）入社．1993年北陸先端科学技術大学院大学研究生．1995年より技術研究組合新情報処理開発機構研究員．現在に至る．オペレーティングシステム，リアルタイム処理，マルチメディア処理などに興味を持つ．日本ソフトウェア科学会会員．



石川 裕（正会員）

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年日本ソフトウェア科学会高橋奨励賞を受賞．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．日本ソフトウェア科学会，ACM，IEEE各会員．