システムソフトウェアと
オペレーティング・システム 73－21
（1996．8．27）

ワークステーションクラスタにおける並列プログラミング環境の実現

堀　敦史[†]　手塚宏史[†]　石川　裕[†]
曽田哲之[††]　原田　浩[††]　古田　敦[††]
山田　務[††]　岡　靖裕[†††]

我々は並列マシンにおける時分割空間分割スケジューリング方式を提案し、ワークステーションクラスタ用のスケジューリングシステム SCore-D を UNIX のデーモンプロセスとして開発した．現在，対話的な並列プログラミング環境実現方式の研究の最初のステップとして，オンライン並列デバッガ SCDB の設計開発を行ってきている．一般に，デバッガプロセスはシステムコールによってデバッグ対象のプロセスの実行制御を行う．このような環境では、SCore-D はユーザプロセスの状態を制御できず，スケジューリングシステムが正常に動作しない．そこで，SCore-D がユーザプロセスに対して OS 機能をサービス可能とするための機構を，プロセス間共有メモリおよび UNIX のシグナルを用いて設計開発した．

Parallel Programming Environment on Workstation Clusters

Atsushi HORI ,[†] Hiroshi TEZUKA ,[†] Yutaka ISHIKAWA ,[†]
Noriyuki SODA ,[††] Hiroshi HARADA ,[††] Atsushi FURUTA ,[††]
Tsutomu YAMADA [††] and Yasuhiro OKA[†††]

We have been proposing Time-Space-Sharing Scheduline (TSSS) and developed a scheduling system, named SCore-D, as demon processes on UNIX. As a first step towards the research and development of an interactive parallel programming environment, we are designing a parallel online debugger, named SCDB. Generally debugger process has a control of debugee process. In the case of SCore-D, however, SCore-D can not control debugger and debugee processes, and the sceduling of SCore-D and the control of debugger can conflict. To avoid this situation, we design and develop a mechanism for SCore-D to support OS functions using inter-process shared memory and UNIX signals.

1. はじめに

我々は，並列マシン上で、性能、効率および使い易さの全てを満足させるという目的のもとに，並列ジョブの時分割空間分割スケジューリング方式を提案した[1)~3)]．これは時分割スケジューリングと空間分割スケジューリングを組み合わせたものであり，単純な FIFO バッチスケジューリングに比べ高いプロセッサ利用率とより短い応答時間を可能とする[4),5)]．ワークステーションクラスタを含む並列マシンはその性質上高価であり，複数人での共有利用が必然的である．時分割空間分割スケジューリングのような共有方策は並列マシンの運用上，非常に有効なものである．

現在，商用，研究用を含め多くの並列マシンが存在するが，その大半がバッチによる利用環境しか提供されておらず，時分割スケジューリングの利点のひとつである対話処理に基づく並列プログラムは少なく、実用とはいい難いシステムも見受けられる．

並列処理による高速化は，数時間の処理を要する大規模アプリケーションが数分で終了すると期待される．このことは，今まで無意味であった対話処理が，並列化による高速化の結果，実用的となる可能性を示している．また，アプリケーション開発に必須なデバッガは，別の意味で対話処理を必要とするアプリケーションのひとつである．

しかしながら，並列 OS の研究という意味において，並列プログラミングの対話処理は実績も少なく，

† 新情報処理開発機構つくば研究センタ
Tsukuba Research Center, Real World Computing Partnership
†† （株）SRA
Software Research Associates, Inc.
††† 三菱東部コンピュータシステム（株）
Mitsubishi Electric Computer Systems (Tokyo) Co.Ltd.

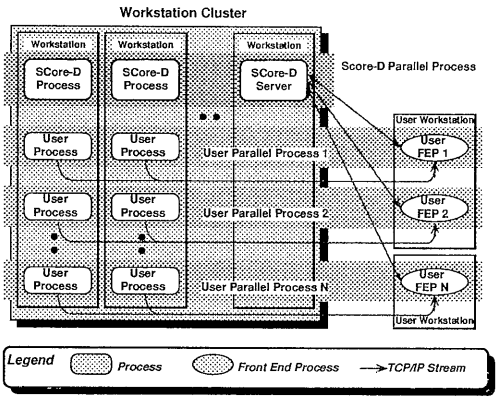

図1　SCore-D のプロセス構造

その性質はまだ十分に解明されてはいない．逐次 OS がディスクやネットワークなどの I/O 待ちにおいてプロセスを待ち状態にするが，マルチスレッドによる並列の場合スレッドが待ちになるだけでプロセスは依然として走り続けることが可能である．また，最も実用的な UNIX の TSS においては見かけの応答時間を短くするためにさまざまな技巧が施されているが，並列マシンの TSS においてどのような対話処理の特性があるのか，応答時間や効率を改善するために OS としてどのような技巧があり，どの程度有効か，といった事項は十分に解明されてはいない．

我々は，"SCore-D" と呼ばれるワークステーション上の時分割空間分割スケジューラを開発した[6]．また，SCore-D 上に対話型並列デバッガを設計開発した．最も重要な対話プログラミングのひとつという理由から，並列デバッガを選択した．本稿は，並列対話処理の研究を目的としたシステムコールの実装方式に関する検討結果について報告するものである．

次章では SCore-D の概要について，第 3 章では並列デバッガ SCDB についてそのぞれの概略について説明する．第 4 章ではスケジューラとしての SCore-D を拡張し，並列プログラミング環境を提供する広義の OS とするための方式について検討する．第 5 章では，第 4 章の結果を受けて，実際に SCDB サーバを例にとり，OS 機能のより具体的な説明を試みる．

2. 並列プログラミング環境 SCore-D

我々は，同じプログラムから同時に生成され，互いに通信しあうプロセス群を「並列プロセス」と呼んでいる．SCore-D[6] は，ワークステーションクラスタにおいて，ユーザの並列プロセスをギャングスケジューリングにより時分割空間分割スケジューリング[7]をサポートするデーモンプロセスの集合である（図 1）．

各ユーザプロセスは，ユーザからの要求に応じて SCore-D の子プロセスとして生成される．ユーザプロセスのスケジューリングは，SCore-D からユーザの子プロセスに対するシグナルで制御されている．図 2 にユーザプロセス生成の過程を示す．ユーザが SCore-D に並列プロセスを生成するには，ユーザは単に手元のワークステーションからプログラムを実行するだけでよい．実行時ライブラリが自動的に SCore-D サーバに接続し，必要な情報を送る．SCore-D では与えられた情報に基づいて対応するユーザプロセスをワークステーションクラスタの各マシン上に生成する．ユーザが手元で起動したプロセスは FEP（Front End Process）と呼ばれ，ワークステーションクラスタ上のユーザ並列プロセスからの標準出力は全てマージされて FEP の標準出力に出力される．また，FEP のプロセスが ^Z でサスペンドされたり，^C で殺された場合，ワークステーション上の並列プロセスもそれに追随するようになっている．

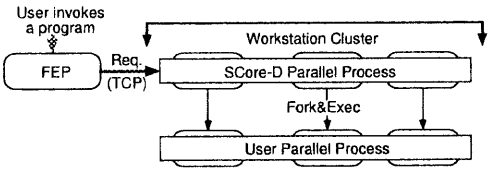

図2　SCore-D におけるユーザ並列プロセスの起動

SCore-D は MPC++[8] で記述されており，MPC++ の実行時ライブラリ[9] は C/C++ で記述されている．MPC++ の実行時ライブラリはユーザレベルスレッドを提供しているため，SCore-D はマルチスレッドで動作する．SCore-D は block-wait で通信メッセージを待つが，ユーザプロセスは busy-wait で通信メッセージを待つ．このため，SCore-D とユーザプロセスはスレッド単位でプロセス切替が発生する．SCore-D は基本的にユーザプロセスを記述している言語に独立である．

我々のワークステーションクラスタの各ワークステーションはギガビット毎秒の速度をもつ Myrinet[10] で接続されており，Myrinet のドライバとして PM[11] が用いられている．PM の特徴は，ユーザレベルで通信ハードウェアを直接制御していることと，複数の通信チャネルを持ち，そのそれぞれを異なるプロセスに割り当てていることである．複数の通信チャネルにより，ユーザレベルで直接ハードウェアを制御しているにも関わらず，複数のプロセスが同時にネットワークを利用することが可能になっている．SCore-D においては，SCore-D が常にひとつのチャネルを占有し，ユーザプロセスには別のチャネルを割り当てている（図 3）．

PM はいくつかの制約から 3 チャネル分しか提供していないため，同時にネットワークを利用するプロセスには制限がある．この制限を解除するため，ユーザプロセスをギャングスケジューリングする場合には，

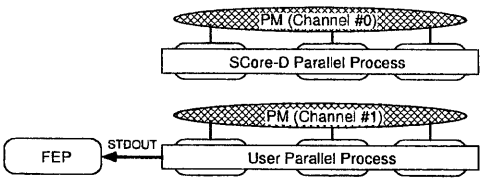

図3　SCore-D におけるユーザ並列プロセスの実行

プロセッサのコンテキスト切替えと同時に，PM チャネルのコンテキストもメモリに退避／復帰している[7]．

SCore-D が提供するギャングスケジューリングだけでは，実用的な対話処理を実現するには不十分である．次章で説明する並列デバッガ "SCDB" は，並列対話処理のアプリケーションとして，また，プログラミング環境ツールとして必要不可欠なデバッグ機能を提供するものとして開発された．

3. 並列デバッガ SCDB

並列デバッガはそれだけで研究テーマとなり得るものであるが，ここでは並列対話処理の研究を目的とし，必要最低限の機能を実現することとした．開発の手間を軽減させるため，GDB（GNU debugger）のリモートデバッグ機能を改造し，最低限の並列デバッグのための拡張を考えた．以下，ユーザがコマンドを直接入力したり，コマンドを解析したりするプロセスを単に "SCDB" と呼び，ワークステーションクラスタ上で実際に debugee（被デバッグプロセス）の状態を調べる並列プロセスを "SCDB サーバ" と呼ぶことにする．

3.1 基本設計

GDB によるプログラムのデバッグは，i)core ファイルによる post mortem 型デバッグ，ii) GDB からユーザプロセスを起動する方式，iii) 走行中のユーザプロセスに attach する方式，がある．SCDB ではより簡便な方式として，i) ユーザプログラムが例外やその実行時ライブラリが panic を検出した時点で SCore-D が自動的に SCDB を attach する方式，および ii) プログラム起動直後のランタイムの初期化が終った段階で SCDB を自動的に attach する方式，の 2 種類とした．SCDB の実装という面から見れば，この 2 つの起動方式に起因する違いはほとんどない．SCDB 起動のイベントがユーザプロセスの例外シグナルによるものか，ユーザプロセスのネットワーク初期化完了のシグナル（SCore-D がバリア同期をとり，ユーザプロセスの開始を指示するために用いられている）によるものかだけの違いである．図 4 は例外発生時における SCDB 起動の様子を示している．SCDB プロセスはユーザプロセスの FEP が子プロセスとして生成し，SCDB サーバは SCore-D が生成する．

SCore-D は全てのワークステーションでネットワークファイルシステムによりファイルシステムが同一視できるものと仮定している．このため，複数のプロセスがほぼ同時に core を生成すると，core ファイルが破壊されてしまう可能性がある．SCore-D がシグナルを捕まえ，core ファイルの生成を逐次化し，core ファイル生成後に名前を変更することも可能ではあるが，SCDB が複数の core ファイルを読み込む処理が別途必要となる．以上の理由から，core ファイルによる post mortem デバッグを実装しないこととした．

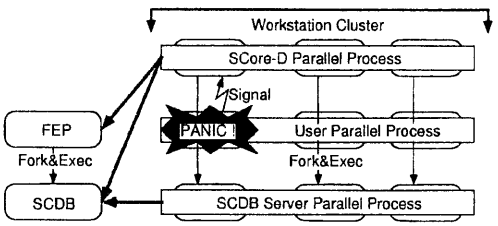

図4　例外発生時の SCDB の起動

3.2 サーバのプロセス構造の設計

SCore-D から見れば SCDB サーバは一般のユーザプロセスと同じであるため，ユーザが直接 SCDB サーバ並列プロセスを生成することも可能ではある．しかしながら，SCDB サーバが SCore-D と無関係に debugee の状態を調べたり，その状態を変化させることは大きな問題を生じる．SCore-D はシグナルを用いてユーザ並列プロセスの実行を制御しているが，SCDB サーバもまた debugee プロセスの実行を制御するため，SCore-D の知らないところで debugee 並列プロセスが動いたり止まったりするからである．これは単にギャングスケジューリングが阻害されるという問題に留まらず，ネットワークが横取りされないため，異なる並列プロセスのメッセージが交じり合い，ある並列プロセスが別の並列プロセスのメッセージを受けとってしまう可能性が生じる．これを避けるためには，SCDB サーバ専用に SCore-D，ユーザプロセスとは別の PM チャネルを割り当てるという回避策も考えられる．この策では SCDB サーバプロセスを特殊扱いにすることを意味し，SCDB はひとつのプロセッサ上で高々 1 つ（PM の通信チャネル数が 3 の場合）しか同時に走ることができない．

より一般的な方法としては，SCore-D を単なるスケジューラから，スケジューリングを含む OS へと発展させ，`ptrace()` などのシステムコールを SCore-D が間接的にサポートする方法が考えられる．次章ではこの方式について検討する．

4. OS としての SCore-D

SCDB サーバをユーザプロセスとして実現する場合，debugee の状態を調べるための `ptrace()` システムコールをどのように実現するかが問題となる．以下

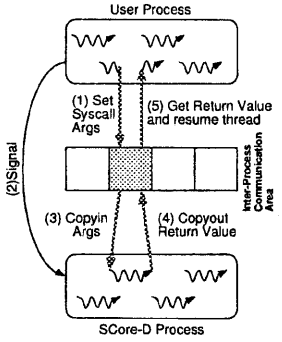

図 5 システムコール処理の流れ

に，システムコール実装の方式と，システムコールとスケジューリングの関係について検討を進める．

4.1 System Call 実装方式の検討

システムコールを OS プロセスとの通信に帰着させることを考える．こうすることでシステムコールの結果を待つためのスレッドサスペンドは，他のスレッドの結果を待つのと同じ枠組で扱えるからである．SCore-D とユーザプロセス間の通信は，プロセス間の共有メモリで実現する．

図 5 に，システムコールの詳細を示す．SCore-D プロセスはユーザプロセスを横取りするため，プロセス間で共有ざれる通信領域の管理には注意を要する．複雑さを避けるため，通信領域はユーザが同時に発することができるシステムコールの数だけ，ブロック化されているものとする．ユーザプロセスがシステムコールを発する際，このブロックにシステムコールに必要な情報（システムコール番号，引数など）を書き込み，シグナルを自分自身に発し，親プロセスである SCore-D に通知する．SCore-D ではこれを受けて通信領域から必要な情報を取り出す．この時，ユーザが誤って間違ったデータを書き込んでいるかもしれないので，チェックは十分に行なわなければならない．システムコールの結果は通信領域に書き込まれる．

ここではユーザプロセスはユーザレベルのマルチスレッドで動作するものとする．システムコールを発したユーザスレッドは，システムコールの結果が得られるまでサスペンド状態になる．システムコール発行の最終段階で発せられるシグナルは，SCore-D への通知であると同時に，ユーザプロセスから SCore-D プロセスへの切替も兼ねている．SCore-D のマルチスレッドランタイムは，ユーザプロセスが発したシグナルを `wait()` で知り，対応するシステムコール処理関数スレッドを生成し，enqueue する．システムコールの処理が完了した時点で，戻り値と，そのシステムコールの処理が完了した旨を通信領域に書き込む．ユーザプロセスの実行が再開され，ランタイムが適当な時期に通信領域を参照し，処理が終了しているシステムコールがあった場合は，そのシステムコールを待っているスレッドを enqueue し，実行を再開する．

ユーザプログラムが SCDB サーバであった場合，SCDB サーバは，i) debugee の状態を調べる，あるいはメモリやレジスタの内容を変更する（`ptrace()`），ii) debugee の状態変化を待つ，iii) SCDB からのコマンドを待つ（I/O），という最低限 3 つのシステムコールが必要となる．

4.2 スケジューリングのタイミング

一般に，個々のプロセスのサスペンドと並列プロセス全体のサスペンドは異なる．ギャングスケジューリングの立場から，個々のプロセスのサスペンドではなく，並列プロセス全体のサスペンドのタイミングが，その並列プロセス切替のタイミングとなる．

ユーザプロセスがマルチスレッドであり，特にユーザレベルスレッドで，busy-wait している場合には，外部からの並列プロセスのアイドルの検出は難しい．ある瞬間に全てのプロセスがアイドルであっても次の瞬間にメッセージが到着し，プロセスがアイドルでなくなる可能性がある．この問題に対し，我々はギャングスケジューリングで用いている「ネットワークプリエンプション」の機構を並列プロセスのアイドル検出に応用することを提案している[7]．全てのプロセスがアイドルであり，かつ退避された PM の通信チャネルにメッセージが存在しない場合，その並列プロセスはアイドルである．もし，アイドルが検出された時点で，SCore-D に対するシステムコールの待ちも存在しない場合には，そのプロセスは終了した（より正確には，これ以上の「計算」は見込めない、デッドロックの可能性がある）と見做すことができる．しかしこの方法では，並列プロセスのアイドル状態を（例えば量子時間間隔で）定期的に調べなければならない．依存関係にある並列プロセスのスケジューリングを考えた場合，資源を浪費する可能性がある．

従って，待ちが発生するシステムコールをどのように扱うかという問題に対処しなければならない．ディスクやプロセッサ間通信ネットワークの I/O 待ちは比較的短時間で解消されると予想されるので，スレッドレベルでの待ちで十分と考えられる．ユーザからのマウスやキーボードからの入力待ちは，比較的長い時間要する場合が多いため，並列プロセスをサスペンドさせた方が効率的であると考えられる．SCDB サーバの場合，SCDB からのコマンド待ち（これはつまり SCDB のユーザからの入力待ちに相当する）および debugee 並列プロセスの状態変化を待つ（トラップ，例外シグナルが挙がった場合や `exit` した場合）部分が SCDB サーバ並列プロセスをサスペンドさせるタイミングと考えることができる．

5. SCDB サーバ

前章で検討した結果を踏まえ，SCDB サーバの処

```
 1 void scdb_command_loop( void ) {
 2   char packet[PLENGTH];
 3
 4   while( 1 ) {
 5     // SCDB からのコマンドパケット待ち
 6     // 自分自身の並列プロセスは入力が
 7     // あるまでサスペンドする.
 8     sc_read( scdb_port, packet, PLENGTH );
 9     switch( packet[0] ) {
10       :
11     case 'c':   // continue コマンドの場合:
12       // 各 PE にコマンドを broadcast する.
13       // この結果, 全ての PE で slave()
14       // 関数スレッドが起動される.
15       broadcast_commad( 'c' );
16       // Debugee 並列プロセスの状態変化を待つ.
17       // このシステムコールにより自分自身の
18       // 並列プロセスはサスペンドする.
19       sc_wait( debugee_pid, ... );
20       // Debugee の個々のプロセスの
21       // 状態を SCDB に返す.
22       gather_resume_replys();
23       break;
24     }
25   }
26 }
```

図 6 マスタースレッドの処理概要

理の骨格を MPC++ で記述した例を以下に示す. 図 6 は SCDB との接続点であり, SCDB サーバ全体の動作を制御するマスタースレッドのコードであり, 図 7 は, マスターの指令を受け, 個々の debugee の状態を調べるスレーブスレッドのコードである. マスタースレッドは SCDB サーバにひとつしか存在せず, スレーブスレッドは各 PE にひとつだけ存在する. プログラム中, sc_ で始まる関数は, SCore-D に対するシステムコールである.

マススレッドは, SCDB からの通信パケットを受けとり, 指定されたプロセッサあるいは全てのプロセッサに対し, コマンドを投げ, その結果を SCDB に中継する. 一方, スレーブスレッドでは, マスターから送られてきたコマンドを実行し, その結果をマスターに返す処理をする.

並列プロセス切替は, i) SCDB からのパケット待ち, ii) debugee 並列プロセスの状態変化待ち, の 2 箇所で発生する可能性がある. スレーブスレッドにおいて, debugee プロセスの状態変化結果を待つ部分では, 当該スレッドのみのサスペンドとなる.

6. 関連研究

ワークステーションクラスタ上のプログラミング環境は, NOW[12], CARMI[13] などがある. SCore-D はワークステーションクラスタ上でギャングスケジューリングおよび時分割空間分割スケジューリング[7] を実装した最初のシステムである. NOW プロジェクトで開発が進められている GlUnix[12] は, OS 拡張された

```
 1 void slave( char command ) {
 2   switch( command ) {
 3     :
 4   case 'c':     // continue コマンドの場合:
 5     // Debugee プロセスの実行を再開させ,
 6     sc_ptrace( PTRACE_CONT, ... );
 7     // Debugee プロセスの状態変化を待つ.
 8     // このシステムコールによりこのスレッドは
 9     // サスペンドする.
10     sc_waitpid( debugee_pid, ... );
11     // 止まった状態を調べ, マスター経由で
12     // SCDB に状態を返す.
13     get_infereior_registers( ... );
14   }
15   return;
16 }
```

図 7 スレーブスレッドの処理概要

SCore-D と同じように, 既存の OS の上に並列 OS の機能を別途構築しようというものである. GlUnix では, 既存のバイナリプログラムでも同様の動作環境を提供しようとしているのに対し, SCore-D ではソースレベルでの互換性しか支援していない.

第 4.1 節で提案したシステムコールとスレッドの関係は, システムコールの待ちをスレッドレベルで実現できないというユーザレベルスレッドの問題点と, カーネルレベルスレッドの大きなオーバヘッドという問題点を解決するという意味で Scheduler Activation[14], Unstable Thread[15] と同じであるが, 本稿で提案した方式はシステムコールを通信と同じ扱いにしたという点が大きく異なっている. これは並列システムにおいてプロセッサ間通信をユーザレベルにし, 通信とスレッドを融合させた MPC++ 実行時ライブラリ[9] の場合に有効と考えられる方式である.

PRISM[16] は, CM-5 上のデバッガであり, CM-5 の OS である CMOST 下において debugee プロセスと時分割スケジューリングされる. 本稿で説明したように, SCDB (サーバ) も SCore-D 下において, debugee プロセスと時分割スケジューリングされる.

ま と め

本稿では, ワークステーションクラスタ上に的な並列プログラミング環境を実現するための検討結果について報告した. 具体的な対話処理プログラムとして, 最も重要なツールのひとつであるオンラインデバッガを選び, 開発した. SCDB は GDB のリモートデバッグ機能を改良し, 並列拡張したものである.

また, デーモンプロセスを OS と見做し, デーモンプロセスで OS 機能を実現するための具体的な検討結果について報告した. その結果, ユーザプロセスとデーモンプロセスとの通信は共有メモリを媒介させ, プロセッサ間通信と同じ枠組で扱うことで, カーネル (デーモン) スレッドとユーザスレッドの双方の利点を活かしつつ, 協調動作が可能になることを示した.

デーモンプロセスで並列 OS 機能を実現することで，効率はある程度犠牲になるが，既存の OS をそのまま利用できるという利点がある．

システムコールでは，並列プロセス全体が待ち（サスペンド）されるものと，システムコールを発したスレッドだけが待ちになるものと２種類用意することで，プロセッサ資源を無駄にすることなく，スケジューリングが可能であることが判明した．

本稿における SCDB は，SCore-D が例外を検知した時点で，例外が発生した並列プロセスに attach する．これは開発期間を短縮するための選択であり，本来はシェルなど，OS の外部に実現されるべき機能である．時分割空間分割スケジューラにおいてシェルをどのように実現するかは今後の検討課題である．また，ユーザプロセスの大域的なアイドル検出の詳細も，本稿に述べた SCore-D の実装を待って，検証および評価を開始する予定である．

参考文献

1) 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).

2) 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, 情報処理学会, pp. 33–40 (1994).

3) Hori, A., Ishikawa, Y., Konaka, H., Maeda, M. and Tomokiyo, T.: A Scalable Time-Sharing Scheduling for Partitionable, Distributed Memory Parallel Machines, *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences, Vol. II*, IEEE Computer Society Press, pp. 173–182 (1995).

4) 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).

5) 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, Springer-Verlag, pp. 623–634 (1995).

6) Hori, A., Tezuka, H., Ishikawa, Y., Soda, N., Konaka, H. and Maeda, M.: Implementation of Gang-Scheduling on Workstation Cluster, *IPPS'96 Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 76–83 (1996).

7) 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, Springer-Verlag, pp. 92–105 (1995).

8) 石川裕, 堀敦史, 小中裕喜, 前田宗則, 友清孝志: 並列プログラミング言語 MPC++の実現, 並列処理シンポジウム JSPP'94, pp. 105–112 (1994).

9) 堀敦史, 手塚宏史, 石川裕, 高橋俊行, 曽田哲之, 堀川勉, 小中裕喜, 前田宗則: 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, 情報処理学会, pp. 37–42 (1996).

10) 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).

11) 手塚宏史, 堀敦史, 石川裕: 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 (1996).

12) Anderson, T.E., Culler, D.E., Patterson, D.A. et al.: A Case for NOW (Networks of Workstations), *IEEE Micro*, Vol. 15, No. 1, pp. 54–64 (1995).

13) Pruyne, J. and Livny, M.: Parallel Processing on Dynamic Resources with CARMI, *Job Scheduling Strategies for Parallel Processing* (Feitelson, D. G. and Rudolph, L.(eds.)), Lecture Notes in Computer Science, Vol. 949, Springer-Verlag, pp. 259–278 (1995).

14) Anderson, T. E., Bershad, B. N., Lazowska, E. D. and Levy, H. M.: Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism, *ACM Transactions on Computer Systems*, Vol. 10, No. 1, pp. 53–79 (1992).

15) Inohara, S., Kato, K. and Masuda, T.: 'Unstable Threads' Kernel Interface for Minimizing the Overhead of Thread Switching, *International Parallel Processing Symposium*, pp. 149–155 (1993).

16) Sistare, S., Allen, D., Bowker, R., Jourdenais, K., Simons, J. and Title, R.: A Scalable Debugger for Massively Parallel Message-Passing Program, *Scalable High-Performance Computing Conference*, pp. 825–832 (1994).