Vol.2020-OS-148 No.5
2020/2/27

マルチコア *Tender* における コア間遠隔手続呼出制御処理の高速化と機能拡充

藤戸 宏洋[1]　山内 利宏[1]　谷口 秀夫[1]

概要：マルチコア環境におけるオペレーティングシステム（以降，OS）の並列処理性能の向上には，排他制御オーバヘッドの削減が重要である．そこで，***Tender*** では，OS が制御する資源をコアごとに管理し，資源操作の排他制御を撤廃する方式を実現している．この方式では，他のコアが管理する資源を操作するとき，コア間遠隔手続呼出制御により，コア間で資源操作を依頼する．コア間遠隔手続呼出制御は，1 回の資源操作依頼ごとに代行処理を実行する代行プロセスの生成処理を削減することで，高速化できる．コア間遠隔手続呼出制御の課題として，他のコアへの資源生成時に操作権を設定する機能や，他のコアに生成したプロセスを実行する機能が実現されていないことがある．本稿では，遠隔手続呼出制御の機能拡充による資源操作権制御への対応，および他のコアが管理するプロセスを実行する機能の実現について述べる．評価では，コア間遠隔手続呼出制御の処理時間を測定し，高速化の効果，資源操作権制御のオーバヘッド，およびプロセス分散処理の性能を示す．

Hiromi Fujito[1]　Toshihiro Yamauchi[1]　Hideo Taniguchi[1]

1. はじめに

マルチコアプロセッサが普及し，計算機に搭載されるコア数は増加傾向にある．OS が制御するデータを共有するコア数が増加すると，排他制御オーバヘッドも増加する [1]．マルチコア計算機における分散処理による性能向上のためには，排他制御オーバヘッドの削減が重要となる．そこで，分散指向永続オペレーティングシステム ***Tender***[2]（以降，***Tender***）では，OS が制御する資源をコアごとに管理することで，資源操作の排他制御を撤廃するマルチコア対応方式を実現している [3]．この方式では，他コアが管理する資源の操作において，コア間遠隔手続呼出制御（RPCC : Remote Procedure Call Controller）を利用する．

コア間 RPCC では，自コアが管理する資源と同じインタフェースで他コアが管理する資源を操作し，他コアにプロセスを生成や移動することができる．しかし，他コアが管理する資源の操作ごとに，代行で資源操作を実行する代行プロセス生成処理と消滅処理が発生することにより，処理時間に与える影響が大きい [4]．また，コア間 RPCC には，自コアが管理する資源の操作において可能な資源操作権の制御，および他コアが管理するよう生成したプロセスを実行できない課題がある．

文献 [5] では，代行プロセス生成処理回数を削減することによる高速化に加え，RPCC における資源操作権の制御の概要を述べた．本稿では，RPCC における資源操作権の制御の詳細とプロセス実行機能の RPCC 対応について述べる．また，コア間 RPCC の高速化の効果，RPCC において資源操作権制御を利用することによるオーバヘッド，および他のコアのプロセスを RPCC で実行することによるプロセス分散の処理時間について評価する．

2. *Tender* オペレーティングシステム

2.1 資源の分離と独立化

Tender では，OS が制御する対象を資源と呼び，分離と独立化を行っている．たとえば，既存 OS のプロセスを資源「プロセス」として扱い，プロセスに対するプロセッサの割り当てを資源「演算」として扱っている．

資源操作のインタフェースは，資源インタフェース制御（RIC : Resource Interface Controller）によって提供され，open（生成），close（削除），read（入力），write（出力），および control（制御）の 5 種類に分類される．プロセスを生成する場合，資源「プロセス」の open 操作を RIC に依

[1] 岡山大学大学院自然科学研究科
Graduate School of Natural Science and Technology, Okayama University

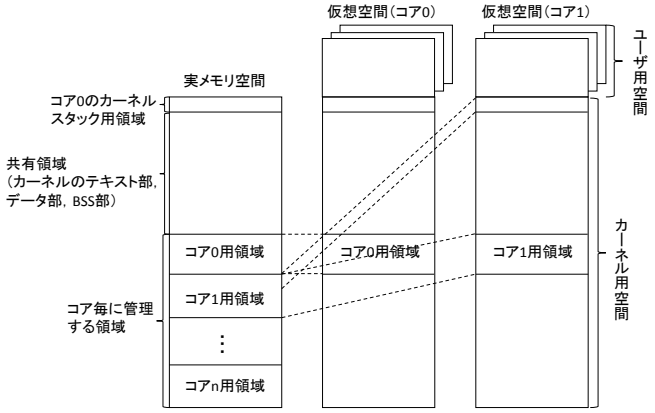

図 1 個別型 ***Tender*** の実メモリ割り当て

頼することにより可能である．

資源は，資源名と資源識別子により識別される．資源名は，場所，資源の種類，および固有名で構成される文字列である．資源識別子は，場所，種類，および同一種類内の通番で構成される数値である．RIC に資源操作を依頼するとき，生成時は資源名を指定し，生成後の操作時は資源識別子を指定する．RIC は，依頼された資源の場所が他計算機である場合，計算機間 RPCC により，指定された計算機において代行の資源操作を実行する．これにより，RIC に対する資源の操作依頼について，資源の存在する計算機に関わらず同じインタフェースとしている．

資源の操作権は，資源操作権制御（RCC : Resource Capability Controller）により設定できる [6]．この操作権設定では，資源を生成したユーザとその他のユーザに対し，それぞれ open を除く 4 種類の資源操作を，資源操作種類ごとに許可するか設定できる．また，RIC が提供する資源生成インタフェースでは，生成する資源に設定する操作権を指定できる．ユーザの識別には，流れ識別子 flowid を利用する．flowid はカーネル処理開始時に付与される数値であり，カーネル処理の実行中は同じ flowid を利用し続ける．

また，RPCC により他計算機の資源を操作するとき，依頼元プロセスのカーネル処理の flowid を依頼先計算機に送信し，この flowid に依頼元の計算機番号を付与した数値を代行処理の flowid とする．これにより，代行処理において，依頼元のカーネル処理の flowid を識別できる．

2.2 資源をコアごとに管理するマルチコア対応方式

Tender では，マルチコア計算機における排他制御オーバヘッドの削減のため，資源をコアごとに管理するマルチコア対応方式として，個別型 ***Tender*** を実現している．個別型 ***Tender*** の実メモリ割り当てを**図 1** に示す．個別型 ***Tender*** では，各コアが資源を個別に管理することで，資源操作におけるコア間の排他制御を撤廃している．各コアが個別に持つ必要があるデータは，コアごとに管理する領域に格納される．

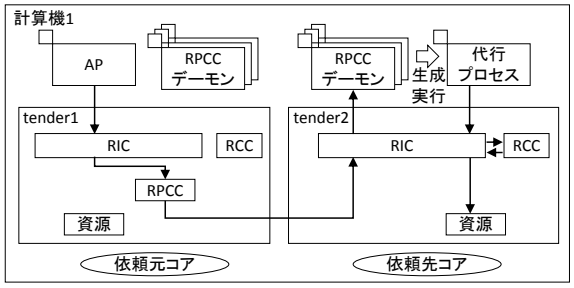

図 2 コア間 RPCC による資源操作の様子

2.3 コア間遠隔手続呼出制御

2.3.1 基本方式

個別型 ***Tender*** では，コア間 RPCC により他コアが管理する資源の操作を実現している．コア間 RPCC による資源操作の様子を**図 2** に示す．図 2 において，RPCC デーモンはそれぞれ 1 つのコアから RPCC のパケットを受信するプロセスであり，各コアに受信元コアの数だけ存在する．

コア間 RPCC は，計算機間 RPCC の拡張であり，通信処理を除いて計算機間 RPCC と同じ処理流れである．個別型 ***Tender*** において RIC に資源操作を依頼したとき，他コアが管理する資源であれば，RPCC に代行処理を依頼する．RPCC は，資源を管理するコアに代行処理を依頼するパケットを送信後，依頼元プロセスを休眠させる．RPCC が送信したパケットは依頼先コアの RPCC デーモンによって受信され，RPCC デーモンが代行プロセスを生成実行する．代行プロセスは，代行処理の実行後，代行処理結果のパケットを依頼元コアに送信し，消滅する．その後，依頼元コアの RPCC デーモンが依頼元プロセスを起床させ，依頼元プロセスにおいて RIC から資源操作結果を返却する．

コア間でのパケットの送受信は，カーネルのデータ部を介したデータの送受信とプロセッサ間割り込み（IPI）により行われる．送信では，送信元コアが管理する領域からカーネルのデータ部に送信するデータをコピーし，依頼先コアに IPI を送信する．IPI の割り込みハンドラは，送信先コアにおいて，受信を要求して休眠中の RPCC デーモンを起床させる．起床した RPCC デーモンは，カーネルのデータ部から送信先コアが管理する領域に，送信されたデータをコピーする．

2.3.2 課題

個別型 ***Tender*** におけるコア間 RPCC の課題として，以下の 3 つが存在する．

（**課題** 1）代行プロセス生成消滅処理のオーバヘッド

個別型 ***Tender*** は，コア数の増加に対する性能向上率の低下の削減を目的として，資源操作におけるコア間の排他制御を撤廃している．この目的を達成するために，他コアが管理する資源を操作する手段であるコア間 RPCC についても，処理の高速化が望まれる．しかし，コア間 RPCC では，代行処理ごとに代行プロセス生成消滅処理が発生す

ることによるオーバヘッドが問題となっている．

（課題 2）RCC への未対応

Tender では，RIC に資源操作を依頼するとき，資源操作を要求する計算機やコア（以降，ローカル）に生成する資源に対し設定する操作権を指定することや，設定した操作権に従った資源操作のみ許可することができる．しかし，これらの機能は RPCC に対応しておらず，ローカルでない資源を管理する計算機やコア（以降，リモート）の資源操作には適用されない．

（課題 3）リモートプロセス実行機能の未実現

個別型 ***Tender*** におけるコア間でのプロセスの分散には，コア間の RPCC が必要である．すでに，RPCC ではリモートへのプロセスの生成機能やプロセスの移動機能が実現済みである [7]．しかし，これらの機能によりリモートに生成や移動したプロセスは停止した状態である．プロセスの分散において利用するためには，プロセスの実行開始に必要な資源操作を RPCC に対応させることにより，リモートプロセスの実行開始機能を実現する必要がある．

3. コア間遠隔手続呼出制御の高速化と機能拡充

3.1 高速化

3.1.1 概要

コア間 RPCC における性能上の問題として，代行プロセス生成処理がある．これは，代行処理ごとに代行プロセス生成消滅処理が発生することにより，代行の資源操作処理時間やコア間の通信時間に対し，代行プロセスの生成時間の割合が比較的大きいことによる．

そこで，代行処理完了後に代行プロセスを消滅させず，次回の代行処理において代行プロセスを再度利用することにより，代行プロセス生成消滅処理を削減する [5]．コア間 RPCC の高速化後の処理流れを**図 3** に示す．代行処理完了後には，休眠中の代行プロセスとして情報を登録し，代行プロセスを休眠させる．次の代行処理開始時に，休眠中の代行プロセスが存在する場合，その代行プロセスを起床させて利用する．これにより，コア間 RPCC 処理を高速化できる．

3.1.2 実現

高速化により，RPCC デーモンが生成した代行プロセスは，消滅するまでに複数回の代行処理を実行する．このため，RPCC デーモンによる代行プロセスの生成時に指定する以下の引数について，代行処理ごとに同じ引数を利用する必要がある．

(1) RPCC デーモンの流れ識別子 `flowid`

引数 `flowid` は，RPCC デーモンの flowid である．代行プロセスは，初期化処理や終了処理において，RPCC デーモンの flowid を利用することにより RPCC デーモンと同じ資源操作権を持つ．

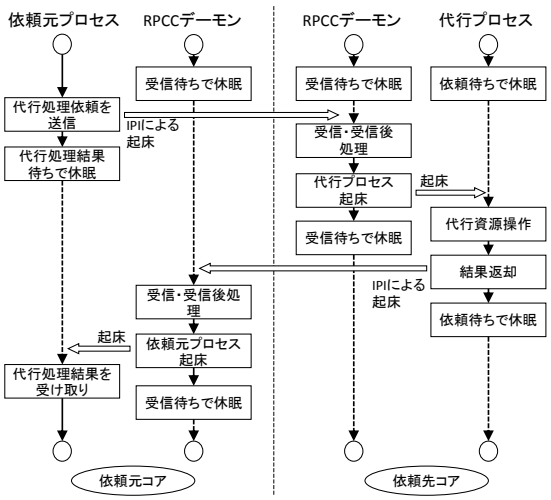

図 3 コア間 RPCC の高速化後の処理流れ

(2) 依頼元の計算機番号 `machine_num`

引数 `machine_num` は依頼元の計算機番号であり，代行プロセスにおいて RPCC が管理するデータを識別するために利用している．

(3) RPCC デーモンの受信回数から算出される数値 `agent_num`

引数 `agent_num` は RPCC デーモンの受信回数から算出される数値であり，RPCC デーモンが依頼のパケットを保管する場所を示す．

RPCC デーモンは，各コアや各計算機において，通信先ごとに存在する．そこで，各 RPCC デーモンが起床させて利用できる代行プロセスは，同じ RPCC デーモンが生成した休眠中の代行プロセスに限定する．これにより，1 つの代行プロセスが実行する複数回の代行処理において通信先が同じとなり，引数 `flowid` と引数 `agent_num` を連続して利用することができる．

また，RPCC デーモンが依頼内容のパケットを保管する場所について，受信回数から算出する方式の場合，複数回の代行処理によってパケットの保管場所が変更されている可能性が高い．そこで，パケットの保管場所を，パケット受信時に空いている RPCC の管理領域とし，代行プロセスのプロセス識別子と関連付けて管理するよう変更する．これにより，代行プロセスは引数 `agent_num` を利用せず，自身のプロセス識別子からパケットの保管場所を取得可能になる．

3.2 代行処理の資源操作権の対応

3.2.1 概要

コア間 RPCC による他コアが管理する資源の操作において，資源生成時の操作権設定や操作権に基づいた操作は有効でない．これは，操作権の制御に必要な情報が依頼元のコアにのみ存在し，依頼先のコアの RPCC や RCC から直接参照できないためである．そこで，この情報を資源操

作を依頼するパケットに含めて送信することで，他コアが管理する資源の操作権を制御することを可能にする．以降の項では，資源生成時の操作権設定とユーザの識別について，それぞれ対処を説明する．

3.2.2 資源生成時の操作権設定

RIC に他コアが管理する資源の open 操作を依頼した場合，open で指定した資源操作権は RPCC に渡されない．RPCC による代行処理では，生成したユーザに対してはすべての資源操作を許可し，その他のユーザに対してはすべての資源操作を禁止するよう設定される．

そこで，資源の open インタフェースで指定した資源操作権の設定値を，依頼する資源操作の情報とともに依頼先に送信し，代行プロセスが RIC に代行資源操作を依頼するときに利用する．これにより，他コアが管理する資源を生成するとき，自コアが管理する資源と同様に，資源操作権を設定できる．設定した資源操作権は，対象の資源を管理するコアにおいて管理される．

3.2.3 ユーザの識別

資源の open 操作では，資源を生成するユーザを識別し，資源操作権とユーザを関連付けて登録する．また，open を除く資源操作では，資源を操作するユーザを識別し，対象の資源操作が資源を操作するユーザに許可されている場合のみ資源操作を実行する．

このため，代行処理時の flowid から，資源操作を依頼したユーザを識別する必要がある．代行処理時の flowid は，資源操作依頼時の flowid に依頼元の計算機番号を付与した数値となっており，依頼元のカーネル処理と対応している．しかし，flowid からユーザを識別するためには，RCC による flowid とユーザの関連付けが必要であり，この関連付けは資源操作を依頼するプロセスが生成されたコアにのみ存在する．

そこで，RPCC が依頼のパケットを送信するとき，依頼元コアにおいて flowid とユーザの関連付けを取得し，依頼のパケットに含めて送信する．また，RPCC による代行処理の前に，依頼元の計算機番号を付与した flowid とユーザを関連付けて登録する．これにより，代行処理において flowid からユーザを識別できる．代行処理の終了後，登録した関連付けは解除する．

3.3 リモートプロセス実行処理の実現

3.3.1 目的

コア間 RPCC では，RPCC のプロセス生成機能を利用して，他コアが管理するプロセスとしてプロセスを生成できる．また，プロセス移動機能を利用して，自コアが管理するプロセスを他コアが管理するプロセスとして移動できる．しかし，RPCC の機能により生成や移動したプロセスは停止した状態であり，コア間 RPCC により実行開始することができない．これは，プロセスの実行開始に必要な資源操作が RPCC に対応していないためである．

表 1 プロセスの実行開始に必要な資源操作

	実行内容	資源操作
(1)	プロセスのコンテキスト読み取り	資源「プロセス」の read
(2)	演算の生成	資源「演算」の open
(3)	プロセスと演算の関連付け	資源「演算」の read

表 2 プロセスの実行停止に必要な資源操作

	実行内容	資源操作
(1)	プロセスと演算の関連付け解除	資源「演算」の write
(2)	演算の削除	資源「演算」の close

表 3 プロセスを実行するカーネルコールのインタフェース

形式	機能
procgetexec(pid, mips)	演算の程度 mips を持つ演算を 1 つ生成し，pid で指すプロセスに生成した演算を関連付ける．戻り値は，生成した演算の資源識別子である．
procgetexec2(pid, mips, core)	演算の程度 mips を持つ演算をコア番号 core に 1 つ生成し，pid で指すプロセスに生成した演算を関連付ける．戻り値は，生成した演算の資源識別子である．

本稿では，プロセス実行開始機能の RPCC 対応を実現し，RPCC によるプロセスの分散を可能にする．RPCC によるプロセス実行開始について，必要な資源操作とカーネルコールの RPCC 対応により実現することが検討されている [7]．また，プロセスの移動や削除を行うときは，事前に対象のプロセスを停止させる必要がある．そこで，プロセス実行停止機能についても RPCC に対応する．

3.3.2 実現

プロセスの実行開始に必要な資源操作を**表 1** に示す．表 1 の資源操作について，RPCC において引数や結果を渡す処理を実現することで，他コアが管理するプロセスを実行可能になる．ただし，個別型 ***Tender*** では資源をコアごとに管理するため，プロセスに関連付ける資源「演算」は，資源「プロセス」と同じコアに生成する必要がある．

プロセスの実行停止に必要な資源操作を**表 2** に示す．表 2 の資源操作について，プロセス実行機能と同様に RPCC に対応させることで，他コアが管理するプロセスを停止させることが可能になる．これにより，他コアが管理するプロセスの削除が可能になる．ただし，他コアが管理するプロセスの移動においては，RPCC の依頼先から RPCC を呼び出す必要がある．RPCC ではこの操作に対応していないため，他コアが管理するプロセスの移動については残された課題である．

プロセスを実行するカーネルコールのインタフェースを**表 3** に示す．プロセスを実行するカーネルコールは，

表 4　評価環境

OS	個別型 ***Tender***
CPU	Intel Core i7-2600 (3.4GHz，4 コア)
RAM	8192MB

procgetexec() と procgetexec2() の 2 つが存在する．これらのカーネルコールにおいて，リモートのプロセスを実行するよう指定した場合，プロセスと同じ場所に資源「演算」を生成し，資源「演算」によりプロセスを実行する．

4. 評価

4.1 評価方法

コア間 RPCC の性能について，コア間 RPCC を呼び出す処理の処理時間により評価した．処理時間の測定には RDTSC 命令を利用した．評価環境を**表 4** に示す．処理時間について，以下の観点により評価した．

（観点 1） 代行プロセス処理高速化の効果
（観点 2） RCC のオーバヘッド
（観点 3） コア間のプロセス分散処理の性能

それぞれの評価では，資源「仮想領域」の生成操作と削除操作により評価を行った．資源「仮想領域」は，実メモリあるいは外部記憶装置のデータ格納域情報を仮想化した資源である．メモリの確保は，資源「仮想領域」を生成し，仮想アドレス空間である資源「仮想空間」に貼り付けることにより可能である．「貼り付け」は，仮想アドレスと実アドレスを対応付けすることである．

RPCC では，分散共有メモリの実現のため，リモートに資源「仮想領域」を生成削除する機能を実現している [8]．これらの生成処理と削除処理では，それぞれ RPCC を 1 回のみ呼び出す．また，（観点 1）と（観点 3）の評価において，RCC を無効化して評価する．以上より，各評価観点において，RPCC を 1 回呼び出すことによる処理時間のオーバヘッドを明確化する．

4.2 代行プロセス処理高速化の効果

4.2.1 概要

本評価では，代行プロセス処理高速化による，RPCC の呼び出し 1 回あたりの処理時間の短縮について評価した．評価用プロセスは，コア 0 で動作し，4KB 分の資源「仮想領域」をコア 0（ローカル）もしくはコア 1（リモート）に生成する処理を 70 回繰り返す．その後，生成した資源「仮想領域」を 1 つずつ削除する処理を 70 回繰り返す．このうち，生成と削除それぞれの処理で最初と最後の 10 回を除いた 50 回の処理について，処理時間の最大値，最小値，および四分位数を評価した．評価においては，以下の内容を変更した．

(1) 操作対象の資源「仮想領域」の場所
(2) 代行プロセスが消滅するまでに代行処理を実行可能な回数

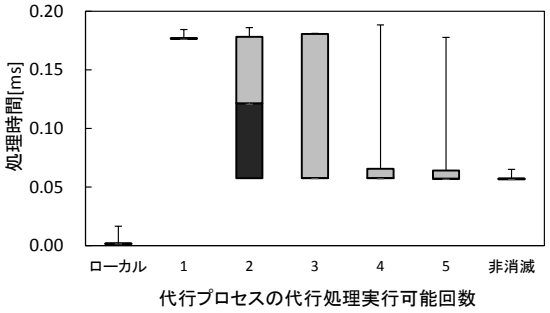

図 4　仮想領域生成処理における高速化の効果

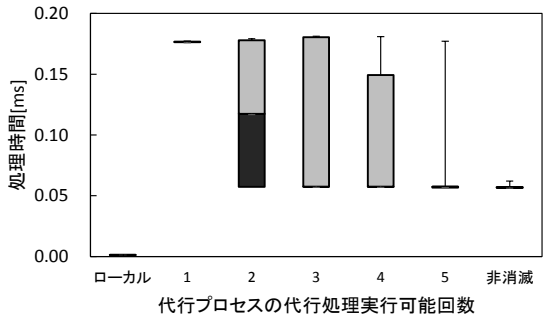

図 5　仮想領域削除処理における高速化の効果

従来のコア間 RPCC では，代行プロセスは代行処理を 1 回のみ実行し，消滅していた．高速化により，代行プロセスは消滅せず，代行処理を何回でも実行できる．本評価では，高速化の効果を明確化するため，代行プロセスが実行可能な代行処理回数を 2 回から 5 回までの範囲で増やした場合についても評価した．2 回から 5 回までの範囲の場合，代行プロセスはその回数だけ代行処理を実行後消滅し，次の代行処理では新しく代行プロセスが生成される．

4.2.2 評価結果

高速化の効果について評価した結果を**図 4** と**図 5** に示す．横軸について，“ローカル” はコア 0 の資源「仮想領域」を操作したときの評価結果であり，その他はコア 1 の資源「仮想領域」を操作したときの評価結果である．また，“非消滅” は，代行プロセス処理高速化により，代行プロセスが消滅しない場合の評価結果である．

従来のコア間 RPCC の通り，代行プロセスが代行処理を 1 回のみ実行可能な場合，ローカル処理に比べ，リモート処理は約 0.17ms だけ長い．代行プロセスが代行処理を実行可能な回数を増やすにつれ，最低値，第 1 四分位数，中央値という順に，処理時間が短くなっていく．これは，代行プロセス生成消滅処理が実行される回数を減らすことによる効果である．また，高速化により代行プロセスが消滅しない場合，ローカル処理に比べリモート処理は約 0.05ms だけ長い．以上により，代行プロセス処理高速化によって，RPCC を呼び出す回数あたり約 0.12ms だけ RPCC のオーバヘッドを削減できたといえる．

表 5　RCC のオーバヘッド [μs]

	RCC 無効	RCC 有効	オーバヘッド
コア 0 への open	2.285	4.045	1.759
コア 1 への open	57.693	62.190	4.497
コア 0 への close	1.192	1.306	0.114
コア 1 への close	56.885	60.180	3.295

しかし，ローカルの仮想領域の生成操作は約 0.002ms，削除操作は約 0.001ms であり，これに比べるとリモート操作のオーバヘッドの割合は大きい．このため，コア間の通信回数を減らすことなどによるリモート操作のオーバヘッド削減は，残された課題である．

4.3 RCC のオーバヘッド

4.3.1 概要

本評価では，資源「仮想領域」の open 操作と close 操作について，RPCC で RCC を利用することによるオーバヘッドを評価する．評価用プロセスは，6.2 節の評価用プロセスと同様に，コア 0 で動作して資源「仮想領域」を生成削除する．このうち，生成と削除それぞれの処理で最初と最後の 10 回を除いた 50 回の処理について，処理時間の平均値を評価した．評価において，以下の内容を変更した．

(1) 操作対象の資源「仮想領域」の場所
(2) RCC の無効化と有効化

RPCC による代行処理において RCC を有効化するため，RCC に必要なデータを依頼先コアに送信し利用している．送信したデータの利用方法が open 操作と open でない操作で異なることにより，オーバヘッドに影響が生じる可能性がある．

4.3.2 評価結果

RCC のオーバヘッドを**表 5** に示す．表 5 において，オーバヘッドは，RCC 有効時と RCC 無効時の処理時間の差である．コア 0（ローカル）における RCC のオーバヘッドについて，close 操作に比べて open 操作が $1.645\mu s(= 1.759-0.114)$ だけ長い．これは，生成する資源「仮想領域」に資源操作権を設定するオーバヘッドによる．

コア 0（ローカル）とコア 1（リモート）において RCC のオーバヘッドを比較した場合，open 操作は $2.738\mu s(= 4.497-1.759)$ だけ長く，close 操作は $3.181\mu s(= 3.295-0.114)$ だけ長い．これは，RCC に必要な情報を取得して依頼先に送信し，依頼先で利用するオーバヘッドによる．

RCC 無効時の処理時間に対する RCC のオーバヘッドの割合について，ローカルでは，open 操作において $78.0\%(= 1.759/2.285 * 100)$ であり，close 操作において $9.5\%(= 0.114/1.192 * 100)$ である．これに対し，リモートでは，open 操作において $7.8\%(= 4.497/57.693 * 100)$ であり，close 操作において $5.8\%(= 3.295/56.885 * 100)$ である．リモート操作における RCC オーバヘッドの割合は，ローカルに比べて小さい．

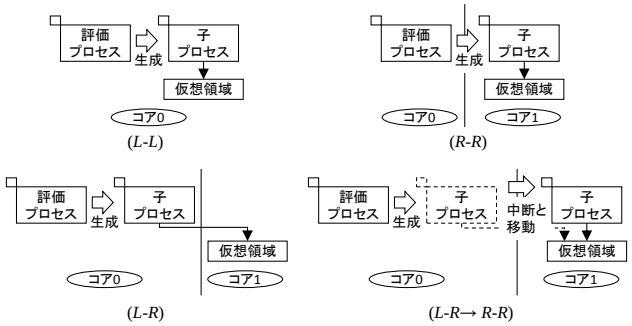

図 6　プロセス分散の評価項目

4.4 プロセス分散の性能

4.4.1 概要

本評価では，資源「仮想領域」の生成と削除を繰り返すプロセスを生成実行し，コア間でプロセスや資源「仮想領域」を分散させることによるオーバヘッドを評価する．評価用プロセスは，コア 0 で動作し，子プロセスを生成実行する．子プロセスは，資源「仮想領域」を生成して削除する操作を繰り返す．測定区間は，子プロセス生成前から，資源「仮想領域」の操作が完了するまでである．評価において，以下の内容を変更した．

(1) 子プロセスの配置先と操作対象の資源「仮想領域」の場所
(2) 子プロセスが実行する資源操作回数
(3) 子プロセスのテキスト部の大きさ

(1) について，評価項目を**図 6** に示す．（*L-L*）では，RPCC を呼び出さない場合について評価した．（*R-R*）では，プロセスを他コアに配置し，そのコア上の資源「仮想領域」を操作した場合について評価した．（*L-R*）では，プロセスが他コアの資源「仮想領域」を操作した場合について評価した．（*L-R* → *R-R*）では，（*L-R*）の評価において資源「仮想領域」を半分の回数だけ操作した後で，プロセスを資源「仮想領域」と同じ場所に移動させ，残りの半分の回数操作した場合について評価した．

(2) について，資源操作回数を 10 回から 80 回の範囲で変更した．また，(3) について，子プロセスのテキスト部の大きさを 4KB から 32KB の範囲で変更し，データ部と BSS 部の大きさを 4KB 固定とした．

各評価項目の RPCC 呼び出し回数を**表 6** に示す．プロセス生成処理や移動処理では，リモートの資源操作が複数回必要であり，それぞれの資源操作で RPCC を呼び出す．

4.4.2 評価結果

資源操作回数ごとのプロセス分散処理時間を**図 7** に示す．（*L-L*）は，4 つの評価項目の中で最も処理時間が短い．これは，RPCC 呼び出しのオーバヘッドが存在しないため

表 6　プロセス分散における各評価項目の RPCC 呼び出し回数

評価項目	プロセスの配置と実行	資源「仮想領域」の操作
（*L*-*L*）	0 回	0 回
（*R*-*R*）	5 回	0 回
（*L*-*R*）	0 回	操作回数
（*L*-*R* → *R*-*R*）	5 回	操作回数の半分

である．

（*R*-*R*）は，（*L*-*L*）に続いて処理時間が短い．また，資源操作回数の増加に従い，（*L*-*L*）と同程度処理時間が増加している．これは，プロセスの配置時のみ RPCC を呼び出し，資源「仮想領域」の操作では RPCC を呼び出さないためである．

（*L*-*R*）は，4 つの評価項目の中で最も処理時間が長く，資源操作回数の増加による処理時間増加の割合が高い．これは，資源操作回数だけ RPCC を呼び出すことによるオーバヘッドが存在するためである．

（*L*-*R* → *R*-*R*）は，（*L*-*R*）に比べて，資源操作回数の増加による処理時間増加の割合が低い．これは，プロセスを資源と同じ場所に移動することにより，RPCC の呼び出し回数が減ったためである．このとき，プロセスを移動するオーバヘッドについては，処理時間への影響が小さい．

子プロセスのテキスト部の大きさとプロセス分散処理時間を図 **8** に示す．テキスト部の大きさが増加した場合，（*L*-*L*），（*R*-*R*），および（*L*-*R*）では処理時間が同程度増加する．これは，テキスト部の大きさの増加に対し，リモートでプロセスを生成する（*R*-*R*）の処理時間とローカルでプロセスを生成する（*L*-*L*）と（*L*-*R*）の処理時間は，同程度増加していることを示す．

（*L*-*R* → *R*-*R*）では，（*L*-*L*），（*R*-*R*），および（*L*-*R*）に比べ，テキスト部の大きさの増加に対する処理時間の増加の割合が大きい．これは，プロセス移動処理において，テキスト部，データ部，BSS 部，およびユーザスタックの内容を全て，コア間 RPCC により移動先のコアにコピーすることが原因であると推察する．

以上より，コア間 RPCC によるプロセス分散の性能は，RPCC の呼び出し回数に大きく依存している．このため，プロセス分散の性能の向上のためには，RPCC の呼び出し回数を減らすことや，RPCC の呼び出しオーバヘッドを更に減らすことが必要となる．

5. 関連研究

ネットワークで接続されたハードウェアを分割して管理する OS として，LegoOS[9] がある．LegoOS は，ハードウェアの構成要素の間でネットワークを介した RPC により通信し，ハードウェアの使用率を改善する．RPC のサーバは，遅延を少なくするため，ポーリングスレッドにより受信を確認する．

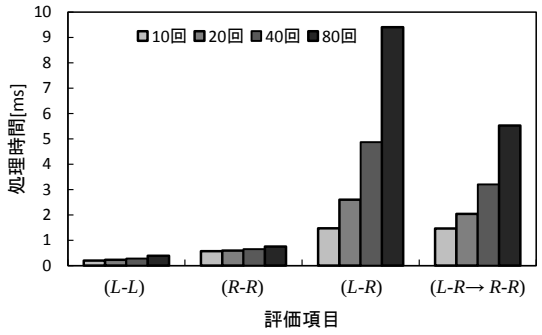

図 7　資源操作回数ごとのプロセス分散処理時間

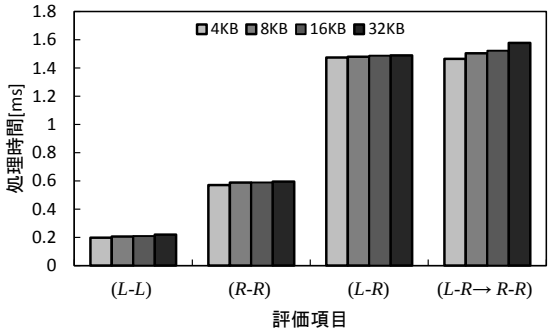

図 8　子プロセスのテキスト部の大きさとプロセス分散処理時間

LegoOS に対し，個別型 ***Tender*** は，コアごとに利用するメモリを分割することで，排他制御オーバヘッドを削減している．また，コア間 RPCC においては，RPCC デーモンが受信を要求して休眠し，IPI により起床して受信する．コア間 RPCC では，各コア上で RPCC デーモンを実行するため，ポーリングにより遅延低下を行うことは難しい．しかし，コア間の資源操作依頼が少ない場合，各コアを効率よく利用できる．

個別型 ***Tender*** と同様に，各コアで個別にメモリを扱う OS として，Barrelfish[10] がある．Barrelfish は，コア間通信を明示的に扱い，通信をモニターすることや最適化することを可能にしている．これに対し，個別型 ***Tender*** のコア間 RPCC は，他コアのプロセス生成や分散共有メモリのために必要なコア間通信を暗黙的に行い，ユーザが他コアの資源を簡単に扱うことを可能にしている．

文献 [11] では，分散システムにおいて RPC のスケーラビリティを向上するため，NIC キャッシュや CPU キャッシュに着目している．***Tender*** の RPCC においても，これらのキャッシュに着目することで，高速化できる可能性がある．

6. おわりに

RPCC において RCC に対応することにより，他コアへの資源生成時の資源操作権設定，および資源操作時の設定した操作権の利用を可能にする実現方式について述べた．また，RPCC によるプロセス実行処理を実現することに

より，他コアに生成したプロセスや移動したプロセスを実行可能にする方式について述べた．最後に，RPCC を呼び出す処理の処理時間を評価した．RPCC の高速化により，RPCC 呼び出しによるオーバヘッドを約 0.12ms だけ削減できたことについて述べた．リモート操作における RCC のオーバヘッドは，ローカル操作と比べ，時間としては長く，割合としては小さいことについて述べた．また，プロセス分散処理の性能は，RPCC の呼び出し回数に依存することについて述べた．

コア間 RPCC は，計算機間 RPCC の拡張であり，通信処理を除いて同じ処理流れである．このため，RPCC の高速化，RCC の RPCC 対応，およびリモートプロセス実行処理について，計算機間 RPCC についても適用できる．

残された課題として，RPCC の更なる高速化と計算機間 RPCC における評価がある．

参考文献

[1] Kashyap, S., Calciu, I., Cheng, X., et al.. : Scalable and Pratical Locking with Shuffling，Proceedings of the 27th ACM symposium on Operating Systems Principles (SOSP '19)，pp.586 – 599 (2019).
[2] 谷口秀夫，青木義則，後藤真孝，村上大介，田端利宏：資源の独立化機構による ***Tender*** オペレーティングシステム，情報処理学会論文誌， Vol. 41, No. 12, pp. 3363–3374 (2000).
[3] 堀井基史，山内利宏，谷口秀夫：***Tender*** におけるコアごとに資源を用意し個別に管理する OS 構造の設計，情報処理学会研究報告，Vol. 2014-OS-131, No. 7, pp. 1 – 8 (2014).
[4] 藤戸宏洋，山内利宏，谷口秀夫：マルチコア ***Tender*** におけるメモリを介した遠隔手続呼出制御方式，情報処理学会研究報告，vol. 2019-OS-145，No. 3，pp. 1 – 8 (2018).
[5] 藤戸宏洋，山内利宏，谷口秀夫：マルチコア ***Tender*** のコア間遠隔手続呼出における代行プロセス処理の高速化，第 18 回情報科学技術フォーラム（FIT2019） 講演論文集，第 1 分冊，pp.153 – 154 (2019).
[6] 山本淳，谷口秀夫：***Tender*** における統一的な資源操作権制御方式，情報処理学会第 63 回全国大会講演論文集，Vol. 2001，No.1，pp. 81 – 82 (2001).
[7] 石井陽介，谷口秀夫：位置透過な資源操作方式によるプロセス生成機構，情報処理学会論文誌コンピューティングシステム (ACS)，Vol. 44，No. SIG 10(ACS 2)，pp. 62 – 75 (2003).
[8] 下崎誠，谷口秀夫：***Tender*** オペレーティングシステムにおける分散共有メモリの実現と評価，情報処理学会コンピュータシステムシンポジウム論文集，Vol.2002，No.13，pp. 125 – 132 (2002).
[9] Shan, Y., Huang, Y., Chen, Y. and Zhang, Y. : LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation，13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)，pp.69 – 87 (2018).
[10] Baumann, A., Barham,P., Dagand, P., et al. : The Multikernel: A new OS architecture for scalable multicore systems，Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)，pp.29 – 44 (2012).
[11] Chen, Y., Lu, Y., Shu, J., et al. : Scalable RDMA RPC on Reliable Connection with Efficient Resource Sharing，Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)，pp.1 – 14 (2019).