Vol.2019-OS-145 No.3
2019/2/28

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

藤戸 宏洋[1]　山内 利宏[1]　谷口 秀夫[1]

概要：コア数の増加による計算機の性能向上において，OS 資源（以降，資源）の排他制御オーバヘッドの削減は重要である．***Tender*** では，各コア上の OS ごとに資源を管理することで，コア間の資源操作における排他制御を撤廃したマルチコア対応方式を実現している．各コア上の OS が計算機全体の資源を同じインタフェースで利用することを可能にするため，メモリを介したコア間の遠隔手続呼出制御による代行資源操作を可能にする方式を設計した．本稿では，改造箇所を通信資源の処理部分に局所化することで，通信処理の流用を可能にした実現方式について述べる．また，改造量，および提案手法におけるプロセスの生成と削除の実行時間についての評価結果を述べる．

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している．マルチコア計算機では，各コアが並列処理を行うことにより，計算機の性能を向上できる．しかし，並列処理において共有する資源を扱う場合，データの不整合を防ぐために排他制御を行う必要がある．このため，計算機の性能向上において排他制御オーバヘッドの削減は重要である [1]．

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，各コア上の OS ごとに資源を管理するマルチコア対応方式（以降，個別型 ***Tender***）を実現している．個別型 ***Tender*** では，各コア上の OS が個別に資源の管理表群を保持することにより，コア間で資源を共有しない．これにより，資源操作におけるコア間の排他制御を撤廃している．

Tender は，分散 OS であり，ローカルの資源とリモートの資源を同じインタフェースで操作可能である．同一のインタフェースは，資源インタフェース制御（以降，RIC: Resource Interface Controller) と遠隔手続呼出制御（以降，RPCC: Remote Procedure Call Controller）により実現している．RIC は資源操作のインタフェースを提供しており，リモート計算機の資源操作は RIC から計算機間の RPCC を利用することにより実行される．

個別型 ***Tender*** において，各コア上の OS が計算機全体の資源を操作するには，資源を管理するコア上の OS による資源の代行操作機能が必要である [3]．そこで，コア間の RPCC を設計した [4]．コア間の RPCC では，共有するメモリを介してリモートコアに資源操作を依頼する．

本稿では，コア間の RPCC による資源の代行操作機能の実現方式について述べる．改造箇所を通信資源の処理部分に局所化することにより，通信資源を利用した処理の流用を可能にし，改造量を小さくしている．また，プロセスの生成と削除処理について，コア間の RPCC を利用した場合の性能の評価結果を示す．

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 が制御し管理する対象を資源と呼び，資源の分離と独立化を行っている．たとえば，既存 OS におけるプロセスを，プログラムの実行単位である資源「プロセス」や，テキスト部の大きさとプログラムの開始アドレスなどを提供する資源「プログラム」などに分離している．また，資源を操作するプログラムは資源の種類と操作内容ごとに独立して存在する．資源を操作するプログラムは RIC によって管理され，資源操作は RIC を介して実行される．

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

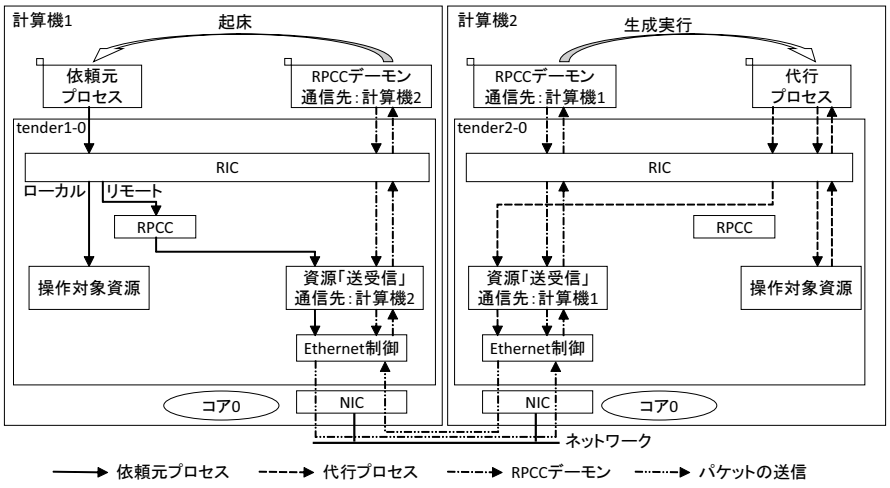

図 2 計算機間の RPCC によるリモート計算機の資源利用の様子

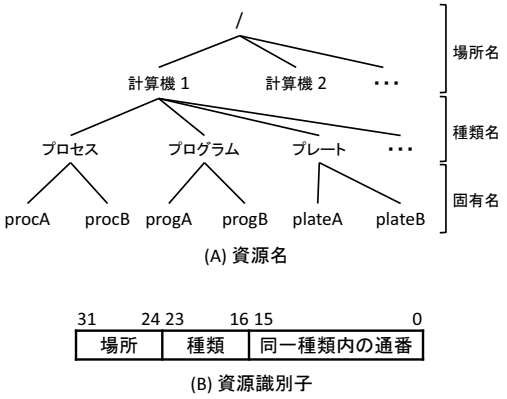

図 1 資源名と資源識別子

資源は資源名と資源識別子により識別される．RIC に資源の生成を依頼するときは資源名を利用し，生成した資源の操作を依頼するときは資源識別子を利用する．資源名と資源識別子を**図 1** に示す．資源名は，たとえば “/tender/process/procA”のような文字列であり，場所名，資源の種類名，および固有名により構成される．資源識別子は，たとえば 0x000a0001 のような数値であり，場所，資源の種類，および同一種類内の通番により構成される．資源名の場所名と資源識別子の場所は，陽にローカルに指定できる．陽にローカルである資源名と資源識別子は，どの計算機においてもローカル計算機の資源として扱われる．

2.2 計算機間の遠隔手続呼出制御

Tender では，計算機間の RPCC によりリモート計算機の資源の利用が可能である．計算機間の RPCC によるリモート計算機の資源利用の様子を**図 2** に示す．図 2 において，RPCC デーモンとは RPCC により送信されたパケットの受信と受信内容に応じた処理を実行するデーモンプロセスであり，代行プロセスとはリモート計算機において代行で資源操作を実行するプロセスである．

Tender において RIC に資源操作が依頼されたとき，資源の場所が陽にローカルでなければ，RPCC に資源操作が依頼される．RPCC では，各計算機の情報が登録されている通信表に基づいて資源の場所を判断する．資源の場所がローカルであれば，資源の場所を陽にローカルに変更し，再度 RIC に資源操作を依頼する．資源の場所がリモートであれば，資源操作依頼の内容を手続き呼び出しパケットとして資源が存在する計算機に送信し，資源操作結果が返却されるまで休眠する．送信された手続き呼び出しパケットは依頼先計算機の RPCC デーモンにより受信され，RPCC デーモンは代行プロセスを生成実行する．代行プロセスは依頼された資源操作を代行で実行した後，資源操作結果の内容を呼び出し結果パケットとして依頼元計算機に送信し，終了する．送信された呼び出し結果パケットは依頼元計算機の RPCC デーモンにより受信され，RPCC デーモンは依頼元プロセスを起床させる．依頼元プロセスでは，RPCC から RIC に資源操作結果が返却され，RIC から資源操作結果が返却される．

RPCC により，依頼元プロセスではローカル計算機に存在する資源とリモート計算機に存在する資源を RIC に対する同じインタフェースで利用可能である．また，資源の場所を陽にローカルにして RIC に資源操作を依頼することで，RPCC への依頼や通信表を利用した判定処理をせず資源操作が可能である．

計算機間の RPCC では，リモート計算機との通信に ***Tender*** における通信資源である資源「送受信」を利用する．資源「送受信」は，Myrinet 制御や Ethernet 制御といった入出力制御への依頼により送信や受信を実行する．送信や受信時に利用する入出力制御は，資源「送受信」の生成時に指定する．図 2 では利用する入出力制御に Ethernet 制御を指定している．また，データの送受信に用いる送信領域と受信領域は，資源「送受信」の生成後，送信や受信をする前に登録する．

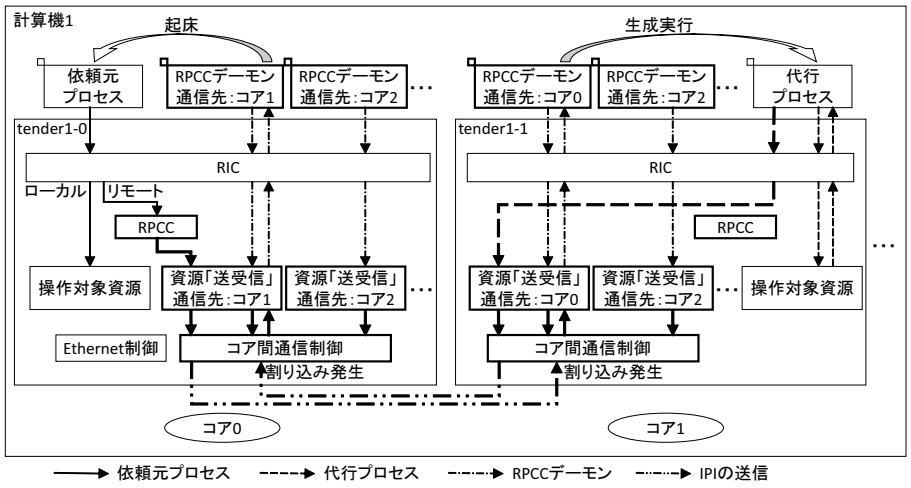

図 4　コア間の RPCC によるリモートコアの資源利用の様子

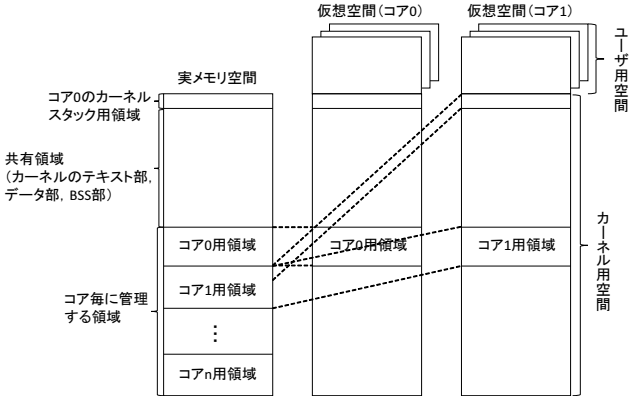

図 3　個別型 ***Tender*** の実メモリの割り当て

2.3　個別型 *Tender*

本節では，マルチコアプロセッサ環境において性能向上を妨げている排他制御オーバヘッドに対処するために，***Tender*** のマルチコア対応方式の 1 つとして提案した個別型 ***Tender*** について述べる [3]．個別型 ***Tender*** では，各コア上の OS がコアごとに個別の資源管理表群を持つ．これにより，各コアで管理する資源が個別になるため，コア間の排他制御無しで資源操作が可能である．また，個別型 ***Tender*** では，実メモリの一部をコアごとに管理する領域として利用している．個別型 ***Tender*** における実メモリの割り当てを図 **3** に示す．これにより，コアごとに管理する領域についてはコア間でメモリの排他制御をせずに扱うことができる．

個別型 ***Tender*** では，リモートコアが管理する資源の操作は，計算機間の RPCC を拡張し，リモートコアでの代行操作を行えば実現できる．しかし，本機能は実現されていない．

3.　遠隔手続呼出制御方式

3.1　設計

個別型 ***Tender*** においてローカルコアの資源とリモートコアの資源を同じインタフェースで利用可能にするため，計算機間の RPCC を拡張することで，コア間の RPCC を設計した [4]．本節では，文献 [4] で述べた設計について述べる．設計方針は，***Tender*** における既存の RPCC の変更量の削減である．設計したコア間の RPCC によるリモートコアの資源利用の様子を図 **4** に示す．

コア間の RPCC を実現するためには，個別型 ***Tender*** におけるコア間の通信方法と通信先コア番号の特定手段が必要である．コア間の通信方法について，コア間通信制御を実現し，資源「送受信」の生成時に入出力制御の一つとして指定可能にすることで，資源「送受信」のインタフェースを変更せず拡張して実現する．また，通信表にコア番号と資源「送受信」の資源識別子の項目を追加し，状態の項目にリモートコアを追加することで，通信先コア番号を特定する．これらの局所化により，設計方針に従い，コア間の RPCC の実現において，リモートと通信する処理の流用を可能にする．

また，資源「送受信」は生成時に通信先を指定する必要がある．このため，コア間の RPCC では，各コア上の OS で通信先コア数だけの資源「送受信」を生成し，利用する必要がある．資源「送受信」は受信要求においてデータが到着していなければプロセスを休眠して待つため，1 つの RPCC デーモンから複数の資源「送受信」を利用することはできない．このため，各コアで通信先コア数だけの RPCC デーモンを生成し実行する．

コア間通信制御の設計としては，コア間でのデータの送受信方法としてコア間での共有領域を利用し，他コアへのデータの送信通知方法としてプロセッサ間割り込み (以降,

IPI) を利用する．共有領域について，コア間の排他制御を避けるため，送信元コアと送信先コアごとに別の領域を通信に利用する．IPI について，発生させる割り込みハンドラにおいて送信元コア番号を求めるため，送信元コアごとに個別の割り込みベクタ番号を利用する．

3.2 実現方式

3.2.1 コア間の遠隔手続呼出制御の実現方式

計算機間の RPCC を拡張し，コア間の RPCC を実現した．コア間の RPCC の実現においては，RPCC と RPCC デーモンに対し，以下の 3 つの改変を行った．

（改変 1）RPCC と RPCC デーモンを各コアで個別化

各コアで個別に RPCC を利用可能にするため，RPCC で利用している管理表や通信表を，各コアで個別に確保して利用する必要がある．このため，RPCC の管理表や通信表の確保と初期化を実行する RPCC の初期化処理を，***Tender*** の起動時に各コアで呼び出すように改変した．

RPCC デーモンについて，各コアで利用可能にする必要がある．このため，個別型 ***Tender*** の起動時に，各コアで最初に起動するユーザプロセスからカーネルコールを呼び出すことで，RPCC デーモンの生成実行を行うように改変した．

（改変 2）複数の通信先への対応

各コアが他のすべてのコアに対して資源操作依頼を可能にするため，通信先ごとに送信領域や受信領域を管理する必要がある．このため，RPCC が送信領域や受信領域を管理するために利用している管理表を，通信先ごとに個別に確保し利用するように改変した．

RPCC デーモンについて，3.1 節で述べた設計の通り，通信先計算機と通信先コアに対し個別に生成実行し，各 RPCC デーモンにおいて 1 つずつ資源「送受信」を利用する．それぞれの RPCC デーモンにおいては，通信先を元に RPCC の管理表を選択する必要がある．このため，RPCC デーモンの生成時と代行プロセスの生成時の引数を追加し，通信先が把握できるように改変した．

また，RPCC と RPCC デーモンでは，RPCC パケットの送信時に，通信先をもとに資源「送受信」を選択して利用する必要がある．このため，資源「送受信」の資源識別子を生成時に通信表に登録し，生成後の操作では通信表を利用して資源「送受信」を選択するよう改変した．

（改変 3）リモートコアに対する資源操作依頼への対応

資源の場所がリモートコアの場合に資源操作依頼先を特定するため，資源名の場所名や資源識別子の場所からコア番号を取得する必要がある．このため，通信表を拡張し，コアの情報を登録可能にした．また，RPCC の初期化処理において，通信表に計算機上のすべてのコアの情報を登録する処理を追加した．

RPCC デーモンについて，通信先がリモートコアの場合に対応した処理を実行する必要がある．このため，RPCC デーモンによる資源「送受信」の生成時，通信先がリモートコアの場合は通信種類としてコア間通信制御を設定するよう改変した．

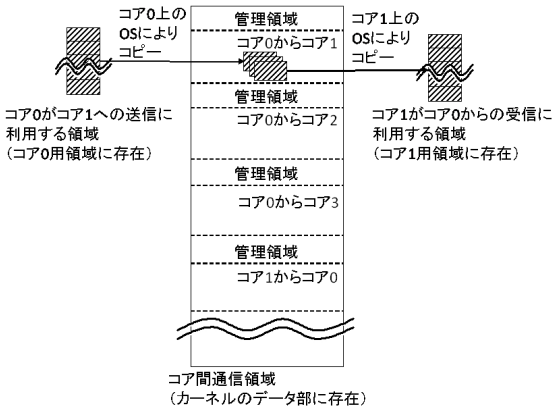

図 5 共有領域を利用したコア間でのデータ送受信の様子

3.2.2 コア間通信制御の実現方式

コア間通信制御について，コア間でのデータ送信手段として共有領域を利用し，データ送信通知として IPI を利用することで実現した．共有領域を利用したコア間でのデータ送受信の様子を**図 5** に示す．コア間での通信に利用する共有領域については，各コアで共有しているカーネルのデータ部に確保することで，個別型 ***Tender*** においてコア間での共有を可能にする．また，共有領域の一部を，共有領域に存在するデータの管理領域として利用する．管理領域では，存在するデータのサイズなどの情報のほかに，それぞれの領域に対する受信待ちで休眠状態のプロセスの識別子を保存している．IPI については，個別型 ***Tender*** において利用されていなかった割り込みベクタ番号をコア数分利用し，送信元のコア番号によって IPI で発生させる割り込みベクタ番号を変更する．

コア間通信制御の初期化について，個別型 ***Tender*** の各コアの起動処理において呼び出すこととした．これは，個別型 ***Tender*** における資源の扱いと同様に，管理表をコアごとに個別に保持することでコア間の排他制御を避けるためである．ただし，割り込みベクタの登録については，最初に起動するコア 0 上の OS のみで行っている．これは，***Tender*** ではすべてのコアで割り込みベクタテーブルを共有しているためである．

4. 評価

4.1 評価の目的と評価内容

実現したコア間の RPCC について，改造量と処理時間の 2 つの観点から評価した．改造量の評価の目的は，コア間の RPCC の実現の局所化ができたかについて確認するためである．このため，資源「送受信」の操作プログラムの改造量と RIC に資源「送受信」の操作を依頼する処理の

表 1　評価環境

OS	個別型 ***Tender***，シングルコア ***Tender***
CPU	Intel Core i3-2100 (3.1GHz，2 コア)
RAM	4096MB
NIC	Intel 82540EM Gigabit Ethernet Controller

表 2　評価項目に対する RPCC の呼び出し発生回数

評価項目	プロセス生成	プロセス削除
LC-LC，*LM-LM*	0	0
LC-RC，*LM-RM*	6	1
RC-RC，*RM-RM*	2	1

改造量をそれぞれ論理行数で調査し，RPCC において資源「送受信」の処理を流用できているか評価する．

処理時間の評価の目的は，コア間の RPCC を利用した資源操作の処理時間のオーバヘッドがどの程度か調査し，実用性を確認するためである．このため，まず，個別型 ***Tender*** におけるリモートコアの資源操作の処理時間をローカルコアの資源操作の処理時間と比較し，リモートコアの資源利用にどの程度処理時間がかかるか評価する．次に，個別型 ***Tender*** におけるリモートコアの資源操作の処理時間をシングルコア ***Tender*** におけるリモート計算機の資源操作の処理時間と比較し，リモートコアの資源利用とリモート計算機の資源利用は，どの程度処理時間の差があるか評価する．

資源操作の処理時間の評価には，プロセスを生成するカーネルコールとプロセスを削除するカーネルコールを利用する．評価で実行するユーザプログラムは，カーネルコールによりプロセスを生成し，その後，別のカーネルコールにより生成したプロセスを削除する．このとき，それぞれのカーネルコールの前後で rdtsc 命令によりクロックを測定し，処理時間を求める．

4.2　評価環境

評価環境を**表 1** に示す．個別型 ***Tender*** における評価では，他計算機と接続せず，2 コアを利用して ***Tender*** を起動する．シングルコア ***Tender*** における評価では，2 台の計算機をハブに LAN ケーブルで接続し，各計算機で 1 コアを利用して ***Tender*** を起動する．

各評価では，生成削除するプロセスとプロセスが実行するプログラムについて，資源の場所を変更して評価を行った．評価項目名について，「*LC-RC*」のように表す．先に記述しているのがプロセスの生成先であり，後に記述しているのがプロセスが実行するプログラムの場所である．個別型 ***Tender*** における計算機間の RPCC の評価では，ローカルコアの資源を *LC*，リモートコアの資源を *RC* と表記する．シングルコア ***Tender*** におけるコア間の RPCC の評価では，ローカル計算機の資源を *LM*，リモート計算機の資源を *RM* と表記する．ここで，ローカルとリモートは，RIC に依頼した OS からみてローカルかどうかを表す．**表 2** に示すように，6 つの組み合わせで評価した．ただし，*RC-RC* と *RM-RM* において，プロセスの生成先とプロセスが実行するプログラムの場所は同じである．

プロセスの生成と削除では，資源の場所の組み合わせにより，RPCC の呼び出しが発生する回数は異なる．評価項目に対する RPCC の呼び出し発生回数を表 2 に示す．また，プロセスの生成削除処理における RPCC への依頼内容を**図 6** に示す．

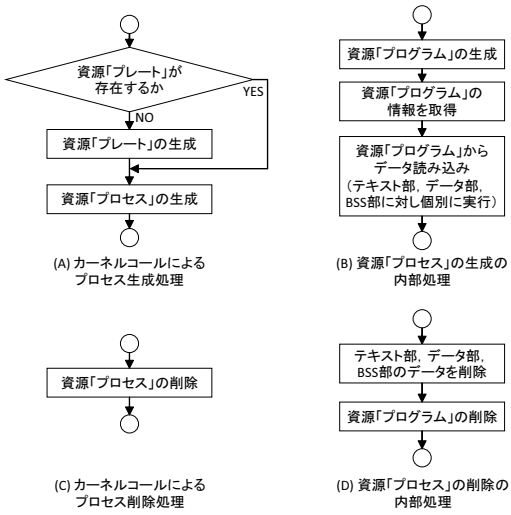

図 6　プロセス生成削除処理における RPCC への依頼内容

図 6 に示した資源操作において，それぞれの資源が資源操作を依頼した OS からみてリモートである場合，RPCC に資源操作が依頼される．また，図 6 における資源「プレート」とは，永続的な記憶をメモリ上に提供する資源であり，既存 OS のファイルに相当する．

プロセスの生成について，*LM-RM* もしくは *LC-RC* のとき，図 6(A) 資源「プレート」の生成，図 6(B) 資源「プログラム」の生成，情報取得，およびデータの読み込みにおいて，計 6 回 RPCC に資源操作が依頼される．*RM-RM* もしくは *RC-RC* のとき，図 6(A) 資源「プレート」の生成と資源「プロセス」の生成において，計 2 回 RPCC に資源操作が依頼される．

プロセスの削除について，*LM-RM* もしくは *LC-RC* のとき，図 6(D) 資源「プログラム」の削除で 1 回 RPCC に資源操作が依頼される．*RM-RM* もしくは *RC-RC* のとき，図 6(C) 資源「プロセス」の削除で 1 回 RPCC に資源操作が依頼される．

4.3　改造量についての評価

コア間の RPCC の実現を局所化し，資源「送受信」の処理が流用できたか確認するため，通信に関する処理の改造量について評価する．コア間の RPCC における通信に関する処理を以下に示す．

表 3　実現における通信に関する改造量 [LOC]

	生成	送信・受信	領域登録
資源の操作プログラム	0(0)	91(2)	60(1)
資源操作依頼処理	7(1)	2(1)	1(1)

(1) RPCC デーモンによる資源「送受信」の生成
(2) RPCC デーモンによる資源「送受信」への送信領域と受信領域の登録
(3) RPCC デーモンによる資源「送受信」への受信要求
(4) 依頼元プロセスと代行プロセスによる資源「送受信」への送信要求

これらの処理について，(1) を「生成」，(2) を「領域登録」，(3) と (4) を「送信・受信」とし，それぞれの改造量について評価する．実現における通信に関する改造量を**表 3** に示す．表 3 において，() の内部は変更ファイル数を示す．

「生成」について，資源「送受信」の操作プログラムは改造していない．これは，資源「送受信」の生成においてコア間通信制御を登録するための改変が必要ないことによる．資源「送受信」に対する操作依頼処理では，7 行の改造を行っている．このうち 5 行は資源名と資源識別子が重複しないための改造であり，この改造は複数の通信先に対応したことによる．また，2 行はコア間の通信を依頼するための改造であり，この改造は通信先がリモートコアの場合に対応したことによる．

「送信・受信」について，資源「送受信」の操作プログラムでは 91 行の改造を行っている．これは，生成時にコア間通信が指定されている場合，送信と受信をコア間通信制御に依頼するための改造である．資源「送受信」に対する操作依頼処理では，2 行の改造を行っている．これは，通信先によって別の RPCC 管理表を利用するための改造である．

「領域登録」について，資源「送受信」の操作プログラムでは 60 行の改造を行っている．これは，「送信・受信」と同様に，コア間通信制御に依頼するための改造である．資源「送受信」に対する操作依頼処理についても，「送信・受信」と同様に RPCC 管理表を利用するため 1 行の改造を行っている．

以上より，資源「送受信」に対する操作依頼処理について，改変箇所は複数の通信先への対応とコア間通信への対応のみであり，計 10 行となっている．このため，設計方針を満たし，コア間の RPCC の実現を局所化したことにより，通信資源を利用した処理の流用を可能にしたといえる．

4.4　個別型 *Tender* における評価

個別型 ***Tender*** におけるプロセスの生成処理時間を**図 7** に示し，プロセスの削除処理時間を**図 8** に示す．プロセスの生成処理時間は，*LC-LC* では 0.21ms から 0.36ms，*LC-RC* では 4.55ms から 4.89ms，*RC-RC* では 2.08ms から 2.86ms であり，*LC-RC* > *RC-RC* > *LC-LC* である．プロセスの削除処理時間は，*LC-LC* は 0.03ms，*LC-RC* では 0.74ms から 0.77ms，*RC-RC* では 0.73ms から 0.75ms であり，*LC-RC* ≒ *RC-RC* > *LC-LC* である．この処理時間の差は，RPCC への依頼によるオーバヘッドによると考えられる．表 2 より，1 回の RPCC において約 0.70ms の処理時間がかかっていることがわかる．

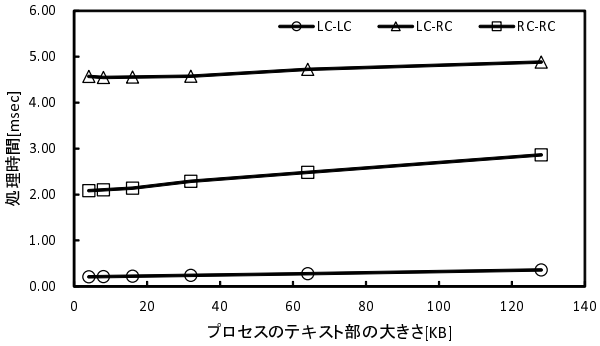

図 7　個別型 ***Tender*** におけるプロセスの生成処理時間の評価

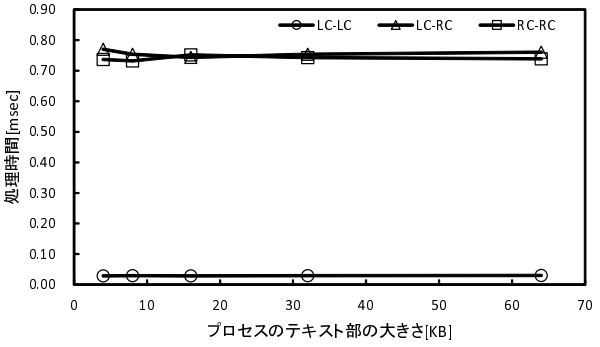

図 8　個別型 ***Tender*** におけるプロセスの削除処理時間の評価

RPCC のオーバヘッドの主な原因は，RPCC デーモンによる代行プロセスの生成処理とプロセッサ割り当て処理である．これらの処理時間の合計は，1 回の RPCC への依頼につき 0.62 ミリ秒程度である．表 2 により，プロセスのテキスト部が 4KB のときの代行プロセスの生成処理時間とプロセッサ割り当て処理時間の合計は，*LC-RC* における生成処理 4.57 ミリ秒のうち 3.72 ミリ秒程度であり，*RC-RC* における生成処理 2.08 ミリ秒のうち 1.24 ミリ秒程度である．以上より，RPCC の処理時間においてコア間の通信によるオーバヘッドの割合は比較的小さいことがわかる．

4.5　リモートコアの資源操作とリモート計算機の資源操作の処理時間の比較

プロセスのテキスト部を 4KB としたときのプロセスの生成削除処理時間を**図 9** に示す．プロセスの生成処理時間について，*LC-RC* は *LM-RM* より 11.67ms 短く，28%の

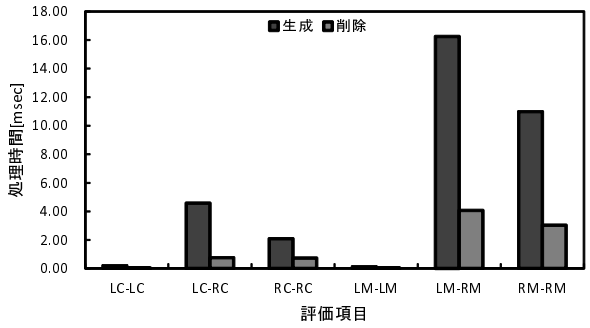

図 9 プロセスのテキスト部が 4KB のときのプロセスの生成削除処理時間

処理時間である．また，*RC-RC* は *RM-RM* より 8.91ms 短く，19%の処理時間である．プロセスの削除処理時間について，*LC-RC* は *LM-RM* より 3.29ms 短く，19%の処理時間である．また，*RC-RC* は *RM-RM* より 2.30ms 短く，24%の処理時間である．よって，プロセスの生成と削除の両方において，処理時間は *LC-RC* < *LM-RM* かつ *RC-RC* < *RM-RM* である．これは，コア間での通信が計算機間での通信より高速であることによると考えられる．それぞれの処理時間の差は，RPCC への依頼回数が多いほど大きくなっている．

以上より，コア間の RPCC によるリモートコアの資源利用は，計算機間の RPCC によるリモート計算機の資源利用より，処理時間が高速であるといえる．

5. 関連研究

コア間の遠隔手続き呼び出しを利用し排他制御を撤廃している例として，Remote Core Locking[6]（以降，RCL）がある．RCL では，クリティカルセクション専用のコアを用意し，このコアに対する遠隔手続き呼び出しに排他制御を置き換える．これに対し，本稿で述べている個別型 ***Tender*** とコア間の RPCC では，専用のコアは用意せず各コアで資源を管理し，他のコアの資源操作を RPCC により行う．

ZygOS[7] では，本稿で述べたコア間の RPCC とは異なる目的で，遠隔手続き呼び出しにおいて IPI を利用している．ZygOS は，遠隔手続き呼び出しの処理を高速化するため，遠隔手続き呼び出しの依頼を受けた計算機上でコア間で作業を調整するために IPI を利用している．これに対し，本稿で述べたコア間の RPCC では，リモートコアに対して遠隔手続き呼び出しの依頼をするために IPI を利用している．

文献 [8] では，***Tender*** の RPCC と同じく，リモートの資源に対し使いやすいインタフェースを提供している．文献 [8] における remote_regions は，RDMA などのリモートメモリを利用する技術に対してのインタフェースとして，ファイルインタフェースを実現している．これに対し，***Tender*** の RPCC では，リモート計算機上のメモリを含めた様々な資源をローカル計算機上の資源と同じインタフェースで利用可能である．また，本稿で述べたコア間の RPCC は ***Tender*** の RPCC の拡張により実現しているため，リモートコア上の資源をローカルコア上の資源と同じインタフェースで利用可能である．

6. おわりに

個別型 ***Tender*** におけるコア間の RPCC の実現について述べた．通信処理の改造箇所は，***Tender*** の通信資源である資源「送受信」の処理部分に局所化し，その改造量は 10 行程度であることを述べた．これは複数の通信先に対応するための改造とリモートコアに資源操作を依頼するために必要な改造によるものであり，コア間の RPCC の実現を局所化したことにより，通信処理の流用を可能にしている．

また，コア間の RPCC を利用したプロセスの生成と削除の処理時間の評価について述べた．コア間の RPCC を利用した資源操作では，RPCC の利用回数 1 回ごとに約 0.70ms の処理時間が増加する．この処理時間の大半は代行プロセスの 1 回の生成処理時間とプロセッサの割り当て処理時間 (合計 0.62ms) であり，コア間の通信処理時間の割合は小さいといえる．次に，シングルコア ***Tender*** における計算機間の RPCC と個別型 ***Tender*** におけるコア間の RPCC を比較した評価について述べた．コア間の RPCC は，計算機間の RPCC に比べ処理時間が 1/4 程度であり，高速である．したがって，コア間の RPCC により複数のコアで資源の分散をすることは，資源操作処理の高速化に有効であるといえる．

残された課題として，コア間の RPCC の処理時間の短縮とリモート計算機に対するコアを指定した資源操作依頼についての検討がある．

参考文献

[1] Boyd-Wickizer，S.，Kaashoek，M. F.，Morris，R. and Zeldovich，N.: Non-scalable locks are dangerous，Proceedings of the Linux Symposium (2012).
[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の方式の設計，vol. 2018-OS-144，No. 6，pp. 1–8 (2018)
[5] 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).
[6] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller : Fast and Portable Locking for Multicore Architectures, ACM Transactions on Computer Systems, Vol.33, No. 4, pp. 1–62 (2016).
[7] G. Prekas, M. Kogias, and E. Bugnion : ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks, Proceedings of the 26th Symposium on Operating System Principles, pp. 325–341 (2017).
[8] M. K. Aguilera, N. Amit et al. : Remote regions: a simple abstraction for remote memory, 2018 USENIX Annual Technical Conference, pp. 757–787 (2018).