OpenXM-RFC102 (Draft)

野呂 正行
(神戸大理)

1 server 間通信の導入

OpenXM-RFC-100, OpenXM-RFC-101 (以下 RFC-100,RFC-101) では, 計算は client (master) と server 間の RPC (remote procedure call) として行われる. この形態で行うことができる分散並列計算としては

- master が仕事を分割して、各 server に依頼する.
- 一つの計算にいくつか方法がある場合に, それぞれの方法を別々の server に依頼する.

などが考えられ, 実際に有効であることを実証した [1]. しかし, さらに複雑な分散並列計算アルゴリズムでは, server 間の通信が必要になる場合がある. 現状の RFC-100,101 でも, server が tree 状に結合して分散計算を行うことはできるが, この場合あくまで一方が client, 他方が server として動作している. 例えば, ScaLAPACK のように, 各 process が行列の一部を保持して, 同一のプログラムを実行しながら LU 分解をしていく, といった並列アルゴリズムを実装するのは難しい. この場合, pivot 選択のために, 複数 process 間で保持されている値の最大値を決定したり, 行の入れ換えを行ったりするために, process 間でのデータのやりとりが必要となる.

さらに単純な例としては broadcast がある. 例えば, ある多項式集合 G がグレブナー基底かどうかチェックするには, G から作られる全ての S-多項式が G により 0 に簡約されることを示せばよい。個々の簡約操作は独立なので, 適当に分割して並列計算できるが, まず G を各 server に送る必要がある. これは, RFC-100 のもとでは master が各 server に順に

送るので, server の数 N に比例した手間がかかるが, server 間通信が使えれば, $\log N$ に比例した手間で送ることもできる.

これらはごく限られた例であるが, 重要なことは, より高度な分散並列計算が実験できるような機能を提案し, 実装することである. 以下では MPI-2 [2] で定義された動的プロセス生成, プロセスグループ間での broadcast の仕様を参考に, OpenXM における server 間通信について述べる.

2 server の起動と server 間通信路の開設

server は RFC-100, 101 により起動されるとする. server は起動された時点では master との通信路のみを持つ. server-server 間の通信路は, master から SM コマンドにより開設される. RFC-100, 101 では, master-server 間の通信路は, server が起動した時点ですでに存在していたが, RFC-102 に対応するためには server が他のプロセスの通信路を開設するための仕組みを実装している必要がある.

1. `SM_set_rank_102`

   server 間の broadcast は, いくつかの server をグループ化して行うのが自然である. 簡単のため, 各 server は, ただ一つのグループに属するとする. master は, 各 server に, その server が属するグループのメンバーの総数 $nserver$ と, そのグループ内での識別子 $rank$ $(0 \leq rank \leq nserver - 1)$ を通知する.

   Request:

`int32 OX_COMMAND`	`int32 SM_set_rank_102`
`int32` $nserver$	`int32` $rank$

   Output: none.

2. `SM_tcp_accept_102`

   Request:

`int32 OX_COMMAND`	`int32 SM_accept_102`
`int32` $port$	`int32` $peer$

   Stack after the request:

`int32 OX_DATA`	`int32` $status$

   Output: none.

   次項の `SM_tcp_connect_102` とペアで, 既に存在している 2 つの server の間の TCP/IP による通信路を開設する. $port$ は, master

が (ランダムに) 選んだ TCP のポート番号である. このリクエストを受け取ると, server は, bind, listen, accept 動作を実行し, connect 待ち状態に入る. いずれかの動作においてエラーを生じた場合には *status* として −1, 成功した場合には 0 をスタックに置く. *peer* は, 相手の server のグループ内での識別子である.

3. `SM_tcp_connect_102`

Request:

`int32 OX_COMMAND`	`int32 SM_tcp_connect_102`
`int32 CMO_String`	*hostname*
`int32` *port*	`int32` *peer*

Stack after the request:

`int32 OX_DATA`	`int32` *status*

Output: none.

前項の `SM_tcp_accept_102` とペアで, 既に存在している 2 つの server の間の TCP/IP による通信路を開設する. *host* は相手の server が動作しているホスト名である. *host* 上, ポート番号 *port* で accept している server に対し, connect する. エラーを生じた場合には *status* として −1, 成功した場合には 0 をスタックに置く. *peer* は, 相手の server のグループ内での識別子である.

3 server 間の通信, broadcast および reduction

RFC-102 下でのプログラミングスタイルは, 基本的には RFC-100,101 と変わらない. すなわち, master であるプログラムが実行され, 必要に応じて server に仕事が依頼され, master はその結果を使って自らの仕事を続行する. これに加えて, RFC-102 では, server どうしが「自律的」にデータの送受信を行うことができる. このため, server は, server 間通信路に OX データを送信する機能, また、server 間通信路から OX データを受信する機能を提供しなければならない.

server 間通信を利用する最も典型的な例として broadcast がある.

1. グループ内 broadcast

   グループ内の broadcast は, いわゆる collective operation として実行される. すなわち, グループ内の各 server でそれぞれ独立に実行

されているプログラムにおいて, 一斉にある関数を呼び出すことにより, その関数から復帰したときに, broadcast されたデータが返される, という形をとる. この場合にデータの発信元 (root) の識別子は各 server があらかじめ知っておく必要がある.

2. master から server グループへの broadcast

   master から server グループへの broadcast は, グループ内の server がスタックコマンド待ち状態に行うことができるとする. この場合, master はある一つの server に data を push する. この server の識別子を $root$ とする. その後, グループ内の全 server に, $root$ を発信元とする broadcast を実行させるためのコマンドを逐次送信する.

以下では, グループ内で broadcast を行う手続きを, MPI での実装にしたがって説明する. 簡単のため $root$ が 0 であるとして, 識別子が $b2^k$ (b は奇数) である server の動作を説明する.

1. 識別子が $(b-1)2^k$ である server からデータを受信する.
2. 識別子が $b2^k + 2^i$ $(i = k-1, \ldots, 0)$ の server にデータを送信する.

この方法によれば, 下位により長く連続して 0 が現われる識別子を持つ server ほど先にデータを送信し始めるため, デッドロックにはならない. また, 独立なペアどうしの通信が同時に行えるとすれば, グループ内の server の総数を $nserver$ とするとき, 高々 $\lceil \log_2 nserver \rceil$ ステップ後には全ての server にデータが行き渡る.

以下に, `ox_asir` における実装を示す.

```
void ox_bcast_102(int root)
{
    Obj data;
    int r,mask,id,src,dst;

    r = myrank_102-root;
    if ( r == 0 )
        data = (Obj)asir_pop_one();
    if ( r < 0 ) r += nserver_102;
    for ( mask = 1; mask < nserver_102; mask <<= 1 )
```

```
        if ( r&mask ) {
            src = myrank_102-mask;
            if ( src < 0 ) src += nserver_102;
            ox_recv_102(src,&id,&data);
            break;
        }
    for ( mask >>= 1; mask > 0; mask >>= 1 )
        if ( (r+mask) < nserver_102 ) {
            dst = myrank_102+mask;
            if ( dst >= nserver_102 ) dst -= nserver_102;
            ox_send_data_102(dst,data);
        }
    asir_push_one(data);
}
```

同様の手続きに reduction がある. これは, 各 server にあるデータを, 2 項演算により処理していき, 最後に *root* に演算結果が集められる手続きである. この場合も, 簡単のため *root* が 0 であるとして, 識別子が識別子が b の server では, b を下位ビットから順に見て,

1. そのビットが 1 なら, そのビットを 0 にした識別子をもつ server (それは必ず存在する) にデータを送信して終了.
2. そのビットが 0 で, そのビットを 1 にした値が $nserver - 1$ 以下なら, そこからデータを受信する. そのデータと手持ちのデータの 2 項演算結果で手持ちデータを更新する.

この方法によれば, 最終結果は *root* にデータが集められる. この場合にも, server の総数を $nserver$ とするとき, 高々 $\lceil \log_2 nserver \rceil$ ステップ後には手続きが終了する.

```
void ox_reduce_102(int root,void (*func)())
{
    Obj data,data0,t;
    int r,mask,id,src,dst;

    r = myrank_102-root;
```

```
    if ( r < 0 ) r += nserver_102;
    data = (Obj)asir_pop_one();
    for ( mask = 1; mask < nserver_102; mask <<= 1 )
        if ( r&mask ) {
            dst = (r-mask)+root;
            if ( dst >= nserver_102 ) dst -= nserver_102;
            ox_send_data_102(dst,data);
            break;
        } else {
            src = r+mask;
            if ( src < nserver_102 ) {
                src += root;
                if ( src >= nserver_102 ) src -= nserver_102;
                ox_recv_102(src,&id,&data0);
                (*func)(CO,data,data0,&t); data = t;
            }
        }
    asir_push_one(r?0:data);
}
```

対応する SM コマンドは以下の通りである.

1. `SM_bcast_102`

   Request:

`int32 OX_COMMAND`	`int32 SM_bcast_102`	`int32` *root*

   Output: none.

   Stack after the request:

`int32 OX_DATA`	*CMObject*

2. `SM_reduce_102`

   Request:

`int32 OX_COMMAND`	`int32 SM_reduce_102`
`int32` *root*	`int32 CMO_String` *opname*

   Stack after the request:

`int32 OX_DATA`	*CMObject*

   Output: none.

4 エラー処理

server は RFC-100,101 の リセットプロトコルを実装していれば, master から server をリセットし, master-server 間の通信路をリセットすることはできる. これに加えて, グループ内の server 間通信路をリセットする必要がある. 識別子が i $(0 \leq i \leq nserver)$ の server の動作は次のようになる.

識別子が i $(0 \leq i \leq nserver)$ の server の動作
for $j = 0$ to $i - 1$ do
    do
        $data \leftarrow$ 識別子 j の server からの OX データ
    while $data \neq$ `OX_SYNC_BALL`
end for
for $j = i + 1$ to $nserver - 1$ do
    `OX_SYNC_BALL` を 識別子 j の server に送信
end for

この手順により, デッドロックなしに全ての通信路を空にすることができる. この操作は, master-server 通信路のリセット後に行われるため, 各 server は master からのデータ待ち状態にある. よって, 次の SM コマンドを各 server に一斉に送ることにより行う.

`SM_reset_102`

Request:

`int32 OX_COMMAND`	`int32 SM_reset_102`

Output: none.

5 API

以下, asir における RFC-102 関連の API を紹介する.

5.1 server 間通信路開設

以下の関数は, master 用に用意されたもので, SM コマンド送信用の wrapper である.

- ox_set_rank_102(*Server*,*Nserver*,*Rank*)

  Server が属するグループに属する server の総数 *Nserver* と, その server のグループ内識別子 *Rank* を通知する.

- ox_tcp_accept_102(*Server*,*Port*,*Rank*)

  Server に対し, ポート番号 *Port* で, 識別子 *Rank* の server からの connect 待ち状態に入るよう指示する. 通信が成立したら, 送受信バッファのセットアップ, 相手先テーブルへの登録などを行う.

- ox_tcp_connect_102(*Server*,*Host*,*Port*,*Rank*)

  Server に対し, ホスト名 *Host* のポート番号 *Port* の TCP ポートに対して connect するよう指示する. 通信が成立したら, 送受信バッファのセットアップ, 相手を *Rank* として相手先テーブルへの登録などを行う.

- ox_reset_102(*Server*)

  Server に対し通信路リセット動作を指示する. この操作は, グループ内全ての server で行われなければならない.

5.2 server 間通信

- ox_send_102(*Rank*,*Data*)

  識別子 *Rank* の server に *Data* を OX データとして送信する. 識別子 *Rank* の server は対応する受信を開始しなければならない.

- ox_recv_102(*Rank*)

  識別子 *Rank* の server から OX データを受信する. 識別子 *Rank* の server は対応する送信を開始しなければならない.

- ox_bcast_102(*Root*[,*Data*])

  識別子 *Root* の server を root として, グループ内で broadcast する. *Data* が指定された場合, スタックにプッシュされる. を指定する必要がある. 識別子が *Root* に等しい server で, スタックからデータがポップされ, そのデータが, 各呼び出しの戻り値となる.

- `ox_reduce_102(`*Root*`,`*Operation*`[,`*Data*`])`

  グループ内の各 server のスタックからポップしたデータに対し *Operation* で指定される二項演算を行い, 結果を *Root* で指定される server での関数呼び出しの戻り値として返す. *Data* が指定された場合, スタックにプッシュしてから上記の操作を実行する. *Root* 以外の server での戻り値は 0 である.

参考文献

[1] Maekawa, M. et al, The Design and Implementation of OpenXM-RFC 100 and 101. Proceedings of ASCM2001, World Scientific, 102-111 (2001).

[2] Gropp, W., et al, Using MPI-2 Advanced Features of the Message-Passing Interface. The MIT Press (1999).