AICS公開ソフトウェア講習会 15回

- 表題　通信ライブラリとI/Oライブラリ
- 場所　AICS R104-2
- 時間　2016/03/23 (水) 13:30-17:00
  - 13:30 - 13:40 全体説明
  - 13:40 - 14:10 PRDMA
  - 14:10 - 14:40 MPICH
  - 14:40 - 15:10 PVAS
  - 15:10 - 15:30 休憩
  - 15:30 - 16:00 Carp
  - 16:00 - 16:30 MPI-IO
  - 16:30 - 17:00 Darshan

AICS公開ソフトウェア講習会
PRDMA

畑中正行

2016/03/23

PRDMA とは? (1)

- PRDMA; Persistent Remote Direct Memory Access
- PRDMA (libprdma) は、RDMA 転送が利用可能なインターコネクト上で、通信レイテンシや計算と通信のオーバーラップを改善するための MPI 永続通信 (MPI Persistent Communication) プリミティブの実装
- 現在、京コンピュータの Tofu インターコネクトをサポート

PRDMA とは? (2)

- やりたいこと
  - 広く使われる isend / irecv MPI 通信の通信レイテンシを改善したい

- 方法
  - インターコネクトの RDMA-Write/Read を使って、直接ユーザバッファに転送する
    - 生のハードウェアに近い性能を出せる可能性があります
    - そのために、既存の MPI_Isend / MPI_Irecv / MPI_Waitall のコードを、永続通信に書き換えてください
    - 意外と簡単だし、今後きっといいことがあるでしょう

PRDMA とは? (3)

- MPI Isend / Irecv における性能問題
  - 内部プロトコルのオーバヘッド
    - Rendezvous プロトコルにおけるハンドシェイク
    - Eager プロトコルにおけるデータ・コピーとフロー制御

  - 上記のオーバヘッドを削減するため:
    - isend/irecv 呼出しを直接下位の RDMA 操作に写像
    - しかし、isend/irecv セマンティックスが複雑すぎる
      - Tag Matching, Message Order Preservation, ...
        - » isend/irecv から離れれば、RMA や隣接集団 (MPI-3) 可
    - 実害のなさそうな範囲でセマンティックスを制限して、移植性を保持しつつ、永続通信の利点を活かし高速化を狙う

永続通信とは

- MPI 永続通信 (Persistent Communication)
  - ほとんどの MPI 実装で利用可能
    - MPI 1.1 仕様以来の MPI 標準
      - MPI-1.0 (1994), MPI-2.0 (1997), MPI-3.0 (2012)
      - 京の MPI 実装は MPI-2.2 版 (富士通MPI)
  - MPI 通信タイプからの分類
    - 基本 MPI 通信機能
      - 双方向(1対1)通信
      - 一方向(片側)通信
      - 集団通信
    - 双方向(1対1)通信
      - ブロッキング通信　　MPI_Send, MPI_Recv, ...
      - 非ブロッキング通信
        - » 非永続通信　　MPI_Isend, MPI_Irecv, ...
        - » 永続通信　　MPI_Send_init, MPI_Recv_init, ...

永続通信への書き換え

- MPI 1対1 (point-to-point) 通信

```
int MPI_Isend( const void *buf, int count, MPI_Datatype datatype, int dest, int tag,
               MPI_Comm comm, MPI_Request *request );
int MPI_Irecv( void *buf, int count, MPI_Datatype datatype, int source, int tag,
               MPI_Comm comm, MPI_Request *request );
int MPI_Waitall( int count, MPI_Request array_of_requests[], MPI_Status array_of_statuses[] );
```

※ MPI_Isend の変型に MPI_Ibsend, MPI_Issend, MPI_Irsend がある

- MPI 永続 (persistent) 通信

```
int MPI_Send_init( const void *buf, int count, MPI_Datatype datatype, int dest, int tag,
                   MPI_Comm comm, MPI_Request *request );
int MPI_Recv_init( void *buf, int count, MPI_Datatype datatype, int source, int tag,
                   MPI_Comm comm, MPI_Request *request );
int MPI_Startall( int count, MPI_Request array_of_requests[] );
```

※ MPI_Send_init の変型に MPI_Bsend_init, MPI_Ssend_init, MPI_Rsend_init がある

書き換え方 – 例

- 等価な書き換え例（1対1通信; NB → 永続通信; PC）

Non-Blocking send/recv (NB)

```
MPI_Request req[2];



do {
  MPI_Irecv(rb, cnt, dt,
    src, tag, comm, &req[0]);
  MPI_Isend(sb, cnt, dt,
    dst, tag, comm, &req[1]);
  /* computation */
  MPI_Waitall(2, req);
} while ( … );
```

Persistent Communication (PC)

```
MPI_Request req[2];

MPI_Recv_init(rb, cnt, dt,
  src, tag, comm, &req[0]);
MPI_Send_init(sb, cnt, dt,
  dst, tag, comm, &req[1]);

do {
  MPI_Startall(2, req);
  /* computation */
  MPI_Waitall(2, req);
} while ( … );
```

初期化時に作り置き

Startall に置換え

書き換え方 – 呼出し方法

- 書き換え方は簡単

```
int MPI_Send_init( const void *buf, int count, MPI_Datatype datatype, int dest, int tag,
                   MPI_Comm comm, MPI_Request *request );
int MPI_Recv_init( void *buf, int count, MPI_Datatype datatype, int source, int tag,
                   MPI_Comm comm, MPI_Request *request );
```

※ MPI_Send_init の変型に MPI_Bsend_init, MPI_Ssend_init, MPI_Rsend_init がある

– MPI_Isend と MPI_Send_init の呼出し形式は同じ
– MPI_Irecv と MPI_Recv_init の呼出し形式も同じ
– 違い
  • 初期化と通信開始が分離
  • MPI_Request のセットでの通信開始 (MPI_Startall)

書き換え方 – 呼出し方法(2)

- 書き換え方は簡単

```
int MPI_Startall( int count, MPI_Request array_of_requests[] );
int MPI_Waitall( int count, MPI_Request array_of_requests[], MPI_Status array_of_statuses[] );
```

※ MPI_Startall の変型に MPI_Start、MPI_Waitall の変型に MPI_Wait がある

– 元々あった MPI_Isend/Irecv を MPI_Startall で置換
– 元々あった MPI_Waitall はそのまま
  • isend/irecv と違って、MPI_Request は再利用可(永続)
  • MPI_Request を無効にするには、MPI_Request_free()

書き換え方 – まとめ

- isend/irecvと等価な永続通信コードへの置換え方を概説
  - プログラムの初期化時に MPI_Send_init/Recv_init
  - MPI_Startall で通信開始
  - MPI_Waitall 後も何度も MPI_Startall 可

- ここからは、PRDMAを使用する上での追加の制約を説明
  - 対にせよ
  - 組にせよ

書き換え方 – 注意 (1)

- 対にせよ (1)
  - 永続通信の通信相手は、永続通信に！
  - “×”は、MPI 仕様上許されているが、PRDMA では禁止

○

```
MPI_Request req[2];
MPI_Recv_init(rb, cnt, dt,
  src, tag, comm, &req[0]);
MPI_Send_init(sb, cnt, dt,
  dst, tag, comm, &req[1]);
do {
  MPI_Startall(2, req);
  /* computation */
  MPI_Waitall(2, req);
} while ( … );
```

×

```
MPI_Request req[2];
MPI_Send_init(sb, cnt, dt,
  dst, tag, comm, &req[1]);
do {
  MPI_Irecv(rb, cnt, dt,
    src, tag, comm, &req[0]);
  MPI_Startall(2, req);
  /* computation */
  MPI_Waitall(2, req);
} while ( … );
```

書き換え方 – 注意 (2)

- 対にせよ (2)
  - 通信相手の MPI_Send_init または MPI_Recv_init の引数 cnt と dt は一致すること
  - 不一致は、MPI 仕様上、許されているが、PRDMA では禁止

○□

```
MPI_Request req[2];
MPI_Recv_init(rb, cnt, dt,
  src, tag, comm, &req[0]);
MPI_Send_init(sb, cnt, dt,
  dst, tag, comm, &req[1]);
do {
  MPI_Startall(2, req);
  /* computation */
  MPI_Waitall(2, req);
} while ( … );
```

特殊な値の扱い

MPI_Recv_init
- 可能 cnt = 0
- 不可 dt = 不連続なデータ型
- 可能 src = MPI_PROC_NULL
- 不可 src = MPI_ANY_SOURCE
- 不可 tag = MPI_ANY_TAG

MPI_Send_init
- 可能 cnt = 0
- 不可 dt = 不連続なデータ型
- 可能 src = MPI_PROC_NULL

書き換え方 – 注意 (3)

- 対にせよ (3)
  - 動的に対 (MPI_Request) の一方を変更しない
    - req[x] は一意に通信相手のreq[y]を指す(≠<{src|dst},tag,comm>)
  - MPI 仕様上許されているが、PRDMA では禁止

×

```
MPI_Request req[2], alt[2];

MPI_Recv_init(rb, cnt, dt,
  src, tag, comm, &req[0]);
MPI_Send_init(sb, cnt, dt,
  dst, tag, comm, &req[1]);

MPI_Recv_init(rb2, cnt, dt,
  src, tag, comm, &alt[0]);
alt[1] = req[1];
```

```
do {
  if ((iteration == 10)
      (myrank == 0)) {
    MPI_Startall(2, req);
  } else {
    MPI_Startall(2, alt);
  }
  /* computation */
  MPI_Waitall(2, req);
} while ( … );
```

書き換え方 – 注意 (4)

- 組にせよ (1)
  - 永続通信に対する MPI_Startall と、対応する MPI_Waitall の引数 count と array_of_requests は一致すること
  - 主に、性能上の理由から、PRDMA では必須

○

```
MPI_Request req[2];
MPI_Recv_init(rb, cnt, dt,
  src, tag, comm, &req[0]);
MPI_Send_init(sb, cnt, dt,
  dst, tag, comm, &req[1]);
do {
  MPI_Startall(2, req);
  /* computation */
  MPI_Waitall(2, req);
} while ( … );
```

PRDMA 内部では、未知（最初）の count 及び array_of_requests の組に対し、通信パターンを分析し、可能なら最適化を行う。その組を管理し、以降の Startall/Waitall 呼出しで、多くのチェックを回避可能

×

```
MPI_Startall(2, req);
MPI_Waitall(1, &req[0]);
MPI_Waitall(1, &req[1]);
```

ばらさない

書き換え方 – 注意 (5)

- 組にせよ (2)
  - 永続通信に対する MPI_Startall の array_of_requests に、isend / irecv の request を混ぜ込まない
  - 主に、性能上の理由から、PRDMA では必須

×

```
MPI_Request req[2+2];
MPI_Recv_init(rb, cnt, dt,
  src, tag, comm, &req[0]);
MPI_Send_init(sb, cnt, dt,
  dst, tag, comm, &req[1]);
do {
  MPI_Irecv( ... , &req[2] );
  MPI_Isend( ... , &req[3] );
  MPI_Startall(2+2, req);
  /* computation */
  MPI_Waitall(2+2, req);
} while ( … );
```

PRDMA 内部では、未知（最初）の count 及び array_of_requests の組に対し、通信パターンを分析し、可能なら最適化を行う。最適化がうまく働かない可能性がある。

MPI_Startall を永続通信用とそうでないものの 2 つに分けることを推奨。

書き換え方 – 注意 (5)

- 組にせよ (3)
  - 永続通信に対する MPI_Startall 及び MPI_Waitall の array_of_requests の配列の要素の順序を入れ替えない
  - 主に、性能上の理由から、PRDMA では必須

×

```
MPI_Request req[2];
MPI_Recv_init(rb, cnt, dt,
  src, tag, comm, &req[0]);
MPI_Send_init(sb, cnt, dt,
  dst, tag, comm, &req[1]);
do {
  { MPI_Request tmp = req[0];
  req[0] = req[1]; req[1] = tmp; }
  MPI_Startall(2, req);
  /* computation */
  MPI_Waitall(2, req);
} while ( … );
```

PRDMA 内部では、MPI_Startallにおいて未知（最初）の count 及び array_of_requests の組を管理し、以降の Startall/Waitall 呼出しで、多くのチェックを回避可能

未知の組か、既知の組かを判断するために、一意の並びにするためreq[]の要素をソートした後で、全要素の比較が必要になる

書き換え方 – まとめ

- isend / irecv を RDMA 転送に直接マップする際の問題
  - MPI point-to-point（1対1通信）

```
int MPI_Isend( const void *buf, int count, MPI_Datatype datatype, int dest, int tag,
               MPI_Comm comm, MPI_Request *request );
int MPI_Irecv( void *buf, int count, MPI_Datatype datatype, int source, int tag,
               MPI_Comm comm, MPI_Request *request );
```

  - PRDMA で課した制約
    - RDMA 転送で通信相手の送信 or 受信バッファアドレスが不明
      - さらに、送信と受信とで count/datatype が異なる可能性
      - さらにさらに、buf/count/datatype が呼出し毎に変わる可能性
        - → MPI_Send_init / MPI_Recv_init で RDMA バッファアドレス交換
        - → その後は「制約」により、送受両側の MPI_Request の一意の関係を維持
    - isend/irecv 毎の同期の管理
      - ⇔ 理想：ユーザ定義の通信パターン全体で1回の同期（request）
        - → MPI_Startall / MPI_Waitall で全体同期
    - tag マッチング要
      - さらに、ワイルドカード MPI_ANY_SOURCE / MPI_ANY_TAG の存在（source / tag）
      - さらにさらに、isend の発行順序と到着順序との暗黙の順序制約あり
      - mpi_probe もあり
        - → MPI_Startall ＋「制約」により、Message Order Preservation を緩和
        - → MPI_ANY_SOURCE, MPI_ANY_TAG, mpi_probe は性能重視とは思えないので、PRDMA の対象外

PRDMA の使い方

- libprdma は、「京」で提供される Open MPI ベースの富士通製 MPI 実装と組合せで使用
  - オリジナル MPI 永続通信関連関数を置換え

- MPI アプリケーションの起動（mpiexec）時、LD_PRELOAD 環境変数を指定することによって、libprdma.so ライブラリを動的にリンク

```
PRDMA_PATH=/opt/aics/prdma/current/lib64/libprdma.so
mpiexec -x LD_PRELOAD=${PRDMA_PATH}/libprdma.so ./a.out ...
```

libprdma – 情報

- ジョブスクリプトサンプル
  - /opt/aics/prdma/templates/run-templ-01.sh

- 京でのインストール場所(計算ノード)
  - /opt/aics/prdma/current/lib64/libprdma.so
    - ログインノードにも、同パスにバイナリが配置されている

使用上の注意

- libprdma では、RDMA-Write/Read に、富士通の拡張 API である FJMPI_Rdma_* を使用
  - RDMA メモリ登録ができても、削除できないので、登録数が超過して失敗することがあります
  - 登録数を減らすため、Halo 通信などでは、袖ごとに MPI_Send_init/MPI_Recv_init する前に、行列全体を MPI_Send_init/MPI_Recv_init してください
    - MPI_Startall しなければ実害はありません

まとめ

- PRDMA
  - 永続通信化 + 実害の少ない制約 → 高速化が可能
  - コード改変は容易 + コード移植性も問題なし
    - PRDMA なしでも、改変後のコードはほとんどのMPI実装で、Isend/Irecv と遜色ない性能 (性能ポータビリティ)
  - バイナリはそのままで、LD_PRELOAD するだけで PRDMA 有効に
  - 改変後のコードは、MPI-3 隣接集団通信(Neighbor-hood Collective) や、標準化作業中の候補機能 MPI-4 永続集団通信 (Persistent Collective) と相性がよい

Backup Slides