富士通 PRIMEHPC FX10 チューニング連載講座

５．MPI 性能チューニング

片桐　孝洋

東京大学情報基盤センター　准教授

１．はじめに

本稿では、FX10 スーパーコンピュータシステム（以降、FX10 と記載）で特徴的な MPI チューニング技法について解説します。紙面の都合から網羅的なチューニング手法の説明は割愛します。チューニング技法について網羅的に知りたい場合は、富士通社が東大ユーザ向けに公開しているチューニングマニュアル[1]がありますので、そちらをご覧ください。

本稿では FX10 で特徴的な、（１）MPI 実行コマンド、（２）実行時の環境変数、を中心に紹介します。また、一般的な話題として、（３）非同期通信を用いた高速化、について説明します。

２．FX10 で特徴的な MPI 実行のさせ方

２．１　概要

富士通社が FX10 で提供する MPI は、MPI-2.1 仕様、および MPI-1.3 仕様をサポートしています。また、スレッドレベルとして、MPI_THREAD_SERIALIZED に対応しています。これは、プロセスがマルチスレッド化されるかもしれませんが、同時に 1 スレッドしか MPI の呼び出しができない挙動です。また、 富士通社が独自に MPI 拡張を行っているものがいくつかあります。それらは、ランク問い合わせインターフェースと拡張 RDMA インターフェースです[1]。

著者の経験では、富士通社は伝統的に MPI の内部機能を多くユーザに公開しています。FX10 でも、性能に影響を及ぼすシステムパラメタ（実行時引数や環境変数など）が指定できます。ここでは[1]から選抜した、代表的な事例をいくつか紹介します。

２．２　実行時オプション

■pjsub コマンドに指定する実行時オプション

MPI ジョブ投入に pjsub コマンドに指定できる実行時オプションを紹介します。
以下のオプションを付けることで、ジョブ全体のノード数を指定するとともに、1 次元、2 次元、３次元形状のノードを確保できます。

- -L node={ **X** | **XxY** | **XxYxZ** }
  - **X** ： X ノードを確保します。
  - **XxY** ： X×Y の 2 次元形状でノードを確保します。
  - **XxYxZ** ： X×Y×Z の 3 次元形状でノードを確保します。

例）3x4x5 の 3 次元形状で 60 ノード確保して MPI ジョブを実行する。foo.bash はジョブスクリプトで、MPI の実行方法が記載されている。

```
$ pjsub  -L node=3x4x5  foo.bash
```

上記の実行時オプションは、ジョブスクリプト中の**#PJM** 指示子の後に記載することもできます。

MPI_Bcast、MPI_Reduce、MPI_Allgather、MPI_Alltoall は、1 次元形状ではなく、3 次元形状でノードを確保することで、メッセージ長が大きい場合（約 100KB 以上のデータ量）で、大幅な性能向上（約 2 倍～約 4 倍のバンド幅の向上）が達成できる例があります[1]。

■mpiexec もしくは mpirun コマンドに指定する実行時オプション

mpiexec のオプションのうち、よく使うものを紹介します。

MPI プログラムをデバックする際、標準出力や標準エラー出力を、プロセスごとに出したい状況があります。通常はファイル数が増えてしますので、デフォルト実行では 1 つのファイルにまとめられて出力されます。

- {-oferr-proc|--oferr-proc} OFERR_PROC_FILE
  - 標準エラー出力を、プロセスごとに OFERR_PROC_FILE に出力します。
- {-ofout-proc|--ofout-proc} OFOUT_PROC_FILE
  - 標準出力を、プロセスごとに OFOUT_PROC_FILE に出力します。

例）標準エラー出力、および、標準出力をプロセスごとに出力させる。

mpirun -oferr-proc err -ofout-proc out 〈実行ファイル〉

以上の例では、標準エラー出力が err.〈ランク番号〉、標準出力が out.〈ランク番号〉というファイル名で出力されます。

■MPI の実行形態に関する実行時オプション

起動時に MPI プロセス全体が 1 つの集合として扱う実行形態(SPMD)が、起動時に MPI プロセスの集合を複数に分けた上で実行する形態(MPMD)かを指定できます。通常は、SPMD で利用することが多いと思います。たとえば連成計算など、複数の MPI プログラムがデータを引き渡すことで協調して実行する場合には、MPMD の形態で利用します。

- SPMD での実行方法
  - mpiexec [global options] [local options] 実行ファイル
- MPMD での実行方法
  - mpiexec [global options] ¥
    -n N1 [local options] 実行ファイル 1 : ¥
    -n N2 [local options] 実行ファイル 2 : …

ここで、global options とは、MPI 全体に影響を与える実行時オプションです。たとえば、先ほどの標準エラー出力をプロセスごとに出力するオプションの -oferr-proc です。一方、local options は、MPI 実行ごとに有効なオプションです。local options には、次節で説明する MCA パラメタの指定オプションがあります。

２．３　性能に影響を及ぼす実行時オプション

ここでは、MPI の実行性能に影響を及ぼす MCA パラメタについて説明します。

■MCA パラメタの設定方法

MCA パラメタは以下のように指定します。このオプションは、local options になります。

- {-mca|--mca} MCA_PARAM_NAME MCA_PARAM_VALUE
  - MCA パラメタ MCA_PARAM_NAME に、値 MCA_PARAM_VALUE を設定します。

■1 対 1 通信のための MCA パラメタ

表 1　1 対 1 通信のための MCA パラメタ（引用先：[1]）

MCA パラメタ	説明
btl_tofu_eager_limit	Eager 通信方式と Rendezvous 通信方式の切り替えしきい値を変更します。
common_tofu_fastmode_threshold	省メモリ型通信モードから高速型通信モードに切り替わるときの通信回数のしきい値を指定します。デフォルトは 16 です。
common_tofu_large_recv_buf_size	高速型通信モードで利用する Large 受信バッファのサイズを変更します。1024 以上を指定します。デフォルトは 1MiB です。
common_tofu_max_fastmode_procs	高速型通信モードのプロセス数の上限を指定します。デフォルトは 1024 です。
common_tofu_medium_recv_buf_size	Medium 受信バッファのサイズを変更します。256 以上を指定します。デフォルトは 2KiB です。
common_tofu_memory_limit	MPI が使用するメモリ使用量を指定します。単位は MiB です
common_tofu_memory_limit_peers	MPI が使用するメモリ使用量を計算するときに仮定する通信相手プロセス数を指定します。

表 1 から、通信量が多いプログラムにおいて、実行時に MPI のメモリ量が足りなくなる場合、common_tofu_memory_limit を指定することで、動作可能になる場合があります。

■集団通信のための MCA パラメタ

表 2　集団通信のための MCA パラメタ（引用先：[1]）

MCA パラメタ	説明
coll_base_reduce_commute_safe	リダクション演算で演算の順序保証を行います。指定すると、実行性能が悪くなります。たとえば、プロセス間のデータの加算時の順序変更による精度の変化が激しいとき以外は、指定しないほうがよいです。デフォルトは、演算の順序保証はしません（パラメタ値は 0）。
coll_tbi_use_on_bcast	MPI_Bcast 関数で Tofu 高機能バリア通信を使用しま

	す。デフォルトは、Tofu 高機能バリア通信を使用します。
coll_tuned_prealloc_size	集団通信関数で作業用バッファを使いまわすようにします。MPI_Allreduce と MPI_Reduce の一部のアルゴリズムに使用されます。デフォルトは 6MiB です。

表 2 の、coll_base_reduce_commute_safe は、例えば連立一次方程式の反復解法において、実行のたびごとに反復回数が異なることが問題になる場合に使えます。連立一次方程式の反復解法の中には、内積計算の値の精度が、全体の数値解の精度に影響を及ぼす場合があります。FX10 の MPI は、同じプロセス数であっても利用者が指定した次元・形状によっては結果が異なることがあるため、実行パターンによっては数値解に影響を与える場合があります。また、逐次計算の結果に対し、並列処理をすることで演算順序が変更されるので、演算順序の違いによる演算結果の違いは避けられません。その結果、収束判定に影響して、反復回数が異なる状況になります。実行ごとに反復回数が異なることに問題がある場合で、実行時間の劣化が問題にならない場合には、coll_base_reduce_commute_safe を設定した方が良いでしょう。

例）**mpiexec　--mca　coll_base_reduce_commute_safe　1　〈実行ファイル〉**

リダクション演算で演算の順序保証を行います。

■その他の MCA パラメタ

表 3　その他の MCA パラメタ（引用先：[1]）

MCA パラメタ	説明
mpi_check_buffer_write	ノンブロッキング送信関数のバッファ破壊を監視します。
mpi_deadlock_timeout	通信待ちの打ち切り時間を指定します。
mpi_print_stats	MPI 統計情報を出力します。

表 3 は、デバック中に MPI 通信で止まる状況（デッドロック時）のデバックで活用できるパラメタといえます。

例）**mpiexec　--mca　mpi_deadlock_timeout　3　〈実行ファイル〉**

受信開始時にタイマーを設定し、3 秒以上経過した場合に、プログラムを強制的に終了します。

例）**mpiexec　--mca mpi_check_buffer_write　1　〈実行ファイル〉**

ノンブロッキング送信中に送信バッファを書き換えようとした場合、異常終了させてプログラムを停止します。

２．４　Eager 通信方式と Rendezvous 通信方式

FX10 の MPI では、1 対 1 通信において、Eager(イーガー)通信方式と Rendezvous（ランデブー）通信方式の 2 種が実装されています。

Eager 通信方式は、1 回当たりのメッセージ量が小さい通信向きの方式です。送受信バッファを経由した通信です。通信バッファに空きがある限り、非同期に通信が行われます。ただし、送信バッファへのコピーと、受信バッファへのコピーが発生します。

一方、Rendezvous 通信方式は、1 回当たりのメッセージ量が大きい通信向きの方式です。送

受信場所を受信側に通知する制御通信が、内部で発生します。ここで、送受信配列の先頭アドレスが、4バイトの境界で、かつ、連続領域であるならば、RDMAを使った直接送信が行われますので、高性能になります。

Eager通信方式とRendezvous通信方式の切り替えは、メッセージサイズが13KiB付近に設定されています。通信ホップ数により異なる実装になっています[1]。切り替えしきい値は、MCAパラメタ btl_tofu_eager_limit で変更できます。ただし、デフォルト実行でも最適化されています。

メモリの余裕があり、非同期通信を促進したい場合は、btl_tofu_eager_limitを大きくすることで、Rendezvous通信方式の利用率を高めることができます。一方で、メモリの余裕が無く、受信バッファを削減したい場合には、btl_tofu_eager_limitを小さくすることで、MPIで用いるメモリ量を削減することができます。

２．５　ランク問い合わせインターフェースと富士通社拡張RDMAインターフェース

その他の機能として、MPIランクと、Tofuの物理ノード座標を変換するインターフェースが用意されています。また、Tofuの4つのネットワークインターフェースを使った通信、迂回経路を利用した通信など、通信ハードウェアの特性を活かした高速通信を行うRDMAインターフェースが用意されています。詳しくは、[1][2]を参照してください。

２．６　非同期通信および持続的な１対１通信による高速化

■非同期通信と持続的通信の概要

MPI の１対１通信の高速化手法として、非同期通信を用いて、通信と計算とをオーバーラッピングすることで、高速化する方法があります。

具体的には、MPI_SEND、MPI_RECVを用いた通信は、ブロッキング通信（同期）通信です。転送側と受信側が同期されて処理が進みます。一方、ノンブロッキング（非同期）通信を用いることで、プログラム上本当に同期が必要な場所まで、同期を後伸ばしにできます。必要な計算を先にさせることができます。この非同期通信をするための関数で代表的なものは、MPI_ISENDとMPI_IRECVです。同期が必要な箇所に使う関数は、MPI_WAITです。

以下に使用例を載せます。変数、myidは自分のランク番号、numprocsおよびMAX_RANK_SIZEは最大プロセス数、が入っていると仮定します。

■同期通信の例

```
integer istatus(MPI_STATUS_SIZE)
…
if (myid .eq. 0) then
  do i=1, numprocs-1
    call MPI_SEND(a, N, MPI_DOUBLE_PRECISION, i, 0, MPI_COMM_WORLD, ierr)
  enddo
else
   call MPI_RECV(a, N, MPI_DOUBLE_PRECISION, 0, 0, MPI_COMM_WORLD, istatus, ierr)
endif
/*  配列a(1:N)を用いた後続処理  */
```

ランク0のプロセスが、自分以外のプロセスにデータa(1:N)を送信

■非同期通信の例

```
integer istatus(MPI_STATUS_SIZE)
integer irequest(0:MAX_RANK_SIZE)
…
if (myid .eq. 0) then
  do i=1, numprocs-1
     call MPI_ISEND(a, N, MPI_DOUBLE_PRECISION,i,0,MPI_COMM_WORLD,
        irequest(i), ierr)
  enddo
    /* a(1:N)に依存しない後続処理 */
  do i=1, numprocs-1
     call MPI_WAIT(irequest(i), istatus, ierr)
  enddo
else
   call MPI_RECV(a, N, MPI_DOUBLE_PRECISION, 0, 0, MPI_COMM_WORLD, istatus, ierr)
endif
/*  配列 a(1:N)を用いた後続処理  */
```

ランク 0 のプロセスが、自分以外の
プロセスにデータ a(1:N)を非同期通信で送信

真の同期すべき場所

一方、上記の非同期通信の例では、MPI_ISEND の実装が、MPI_ISEND を呼ばれた時点で本当に通信を開始する実装になっていないと効果を奏しません。ところが、MPI の実装によっては、MPI_WAIT が呼ばれるまで、MPI_ISEND の通信を開始しない実装がされていることがあります。この場合には、非同期通信の効果が全くありません。ですから上記の MPI_ISEND の書き方は、利用している MPI の実装依存で、効果があるか決まる書き方になります。

持続的通信（もしくは、永続的通信）**(Persistent Communication)** を利用すると、MPI ライブラリの実装に依存して、非同期通信の効果が期待できる場合があります。

持続的通信の利用法は、通信を利用するループ等に入る前に 1 度、通信相手先を設定する初期化関数―この例の場合は MPI_SEND_INIT―を呼びます。その後、SEND をする箇所に MPI_START 関数を書きます。真の同期ポイントに使う関数(MPI_WAIT 等)は、ISEND と同じものが使えます。

MPI_SEND_INIT で通信情報を設定しておくと、MPI_START 時に通信情報の設定が行われません。したがって、同じ通信相手に何度でもデータを送る場合、通常の非同期通信に対し同等以上の性能が出ることが期待できます。具体例を以下に載せます。

■持続的通信を用いた非同期通信の例

```
integer istatus(MPI_STATUS_SIZE)
integer irequest(0:MAX_RANK_SIZE)
…
if (myid .eq. 0) then
  do i=1, numprocs-1
    call MPI_SEND_INIT(a,N,MPI_DOUBLE_PRECISION,i,0,MPI_COMM_WORLD,
         irequest(i), ierr)
  enddo
endif
…
if (myid .eq. 0) then
  do i=1, numprocs-1
    call MPI_START(irequest(i),ierr)
  enddo
 /* 以降は、ISEND の例と同じ */
```

メインループに入る前に、
送信データの相手先情報を初期化

ここでデータを送る（メインループ中）

テストプログラムと実験結果

以上の３つの通信方式を評価するため、以下のようなコードを作成しました。

■テストプログラムの概略

```
<1> do i_loop=1, 10     ←反復のメインループ
<2>   /* 前処理：a(1:N)の定義 */
<3>   先述のランク 0 が a(1:N)を転送し、それ以外のランクは受信する処理
<4>   /* a(1:N)を用いた後処理：a(1:N)は参照のみされる */
<5>   /* 真の同期点：非同期通信時の MPI_WAIT はここに入れる */
<6> enddo
```

ここで、<2>行の前処理では、基本処理単位を a とするとき、a *myid*N の負荷を与えることにします。また、<4>行の後処理は<2>行とは逆に、a*(numprocs-myid-1)*N の負荷を与えることにします。したがって上記の例では、プロセス 0 がデータ a(1:N)を先に転送するのですが、前処理時間が最も少ないので、その他のプロセスより速く転送箇所である<3>行に入ります。

一方、転送後の後処理については、プロセス 0 の計算量が最も多いです。ゆえに、<3>行の送信が同期的に行われると、プロセス 0 の通信待ち時間が理論上最も多くなります。また、通信後の後処理の実行時間も最も長いため、全体時間はプロセス 0 の実行時間に縛られます。

<3>行の転送を非同期にすると、プロセス 0 は、通信と<4>行の計算がオーバーラッピングできるはずですので、非同期通信の効果が出やすいといえます。

持続的通信を用いると、i_loop で 10 回ループが回りますので、MPI_SEND_INIT でメインループに入る前に一度通信情報を設定しておくと、その通信情報は 10 回再利用が出来ます。ですので、持続的通信も有利になる例といえます。

実験結果

以上のテストプログラムを用いて、FX10 の非同期通信の性能を評価しました。利用した FX10 のノード数は 60 ノード、1 ノードあたり 16MPI 実行(ピュア MPI 実行)です。総 MPI プロセス数は 960 プロセスです。実行時のノード確保の形状は指定していません。なお、実行時オプションとして、--mca common_tofu_max_fastmode_procs 0 を指定しています。実験結果を、図 1～図 2 に結果を載せます。

ここでは、前節で説明した、前処理と後処理で利用する基本処理単位 a に、O(N)の負荷を与えています。したがって、N の増加に伴い計算負荷の不均衡が O(N)で生じるため、N が増えるにつれ、同期通信である MPI_SEND は非同期通信 MPI_ISEND に対して理論的に不利になります。

考察

図 1 より、メッセージサイズ 8 バイト～3.2K バイトのメッセージサイズが小さい領域で、同期通信 send を使うより、非同期通信である isend および持続的通信である send_init を使うほうが高速となりました。この理由は、このプログラムでは非同期通信を利用することで、原理的に同期待ち時間を減らせることがあげられます。

一方、非同期通信と持続的通信との時間に差があります。N=201 のとき、それぞれ、0.166 秒と 0.150 秒となりました。したがって、持続的通信を使うと 10%ほど高速化されます。なお、持続的通信では、特定のメッセージサイズで実行時間が不安定（約 2 倍実行時間が増加する）になっています。この原因は解析中です。また、N=33 (264 バイト)で、実行時間が約 3 倍以上増加しています。したがって、このメッセージサイズ周辺で MPI ライブラリ内の処理が変わっていると思われます。

図 2 より、１通信当たりのメッセージサイズ 8K バイト～800K バイトと大きい領域でも、非同期通信の効果が観測できました。ただし、非同期通信 isend と持続的通信 send_init の実行時間の差は、図 1 の例ほど顕著でありません。

N=100,000 の時の実行時間は、同期通信 send の時で 1.86 秒、非同期通信 isend の時で 1.66 秒、および、持続的通信の時で 1.65 秒です。このことから、この例では、非同期通信を導入することで約 13%の速度向上になります。この速度向上の度合いは、通信時間と演算時間の比率で変わってきますので、あくまで、このプログラムを用いたときの速度向上率といえます。この結論は、オーバーラッピングさせる演算量と通信のパターン（どのようにデータを送るのか）で変わってきますので、注意してください。

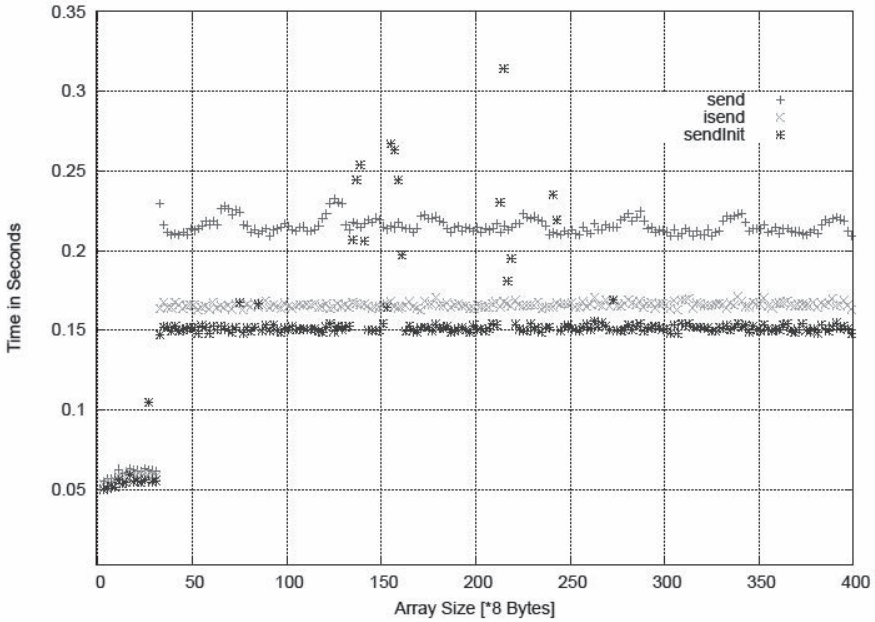

図１　非同期通信の効果

（N=1～400 で 2 間隔。１通信当たりのメッセージサイズ 8 バイト～3.2K バイト。）

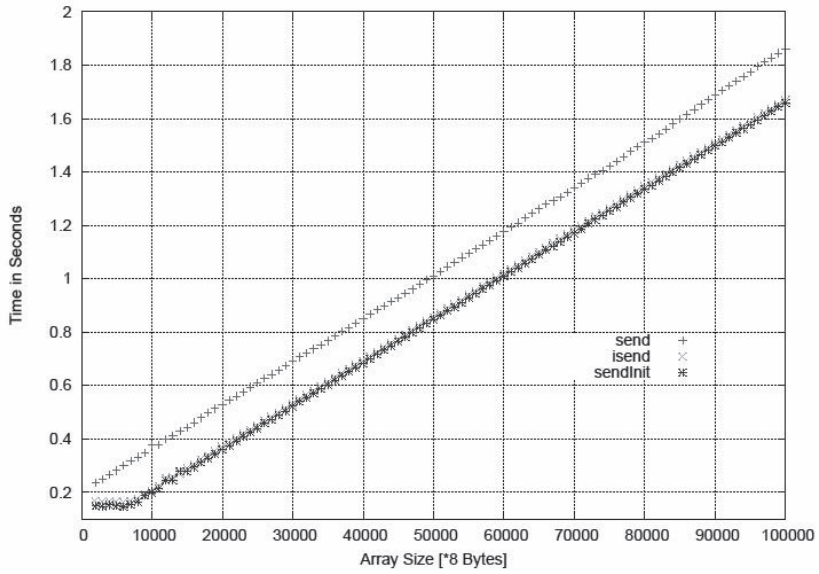

図 2 非同期通信の効果

（N=1,000～100,000 で 1000 間隔。1 通信当たりのメッセージサイズ 8K バイト～800K バイト。）

６．おわりに

本稿では、FX10 における、MPI による性能チューニング手法の一例について説明しました。

FX10 では、ノード確保の形状（1 次元形状、2 次元形状、3 次元形状）、MPI のシステムパラメタの変更、および、1 対 1 通信利用時の非同期通信の利用により高速化できる可能性があります。チューニングの際に参考いただければ幸甚です。

謝辞

本稿を執筆するに当たり、記載内容の校閲について、富士通株式会社の SE の皆様にご協力いただきました。ここに感謝の意を表します。

参　考　文　献

[1]富士通株式会社：FX10 利用者ガイド～MPI 編～（2012 年 12 月 28 日）

[2]富士通株式会社：Technical Computing Suite V1.0、MPI 使用手引書(PRIMEHPC FX10 用)、J2UL-1492-04Z0(01)（2013 年 6 月）

付録

２．６節で使われているプログラムで非同期通信の実装例を以下に載せます。

■同期通信のサンプルプログラム

```
      program main
      include 'mpif.h'
      integer NN
      parameter (NN = 1000)
      double precision a(NN), b(NN)
      double precision t1, t2, t0, t_w, d_c
      integer myid, numprocs, ierr
      integer i_loop
      integer istatus(MPI_STATUS_SIZE), irequest(0:960)
      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
       N=1
       do JJ=1, 200
         call MPI_BARRIER(MPI_COMM_WORLD, ierr)
         t1 = MPI_WTIME(ierr)
```

```
          do i_loop = 1, 10
c           --- Before Load
            do k=1, N
              a(k) = dble(myid)
            enddo
            do i=1, myid
              d_c = 1.0d0/dble(i)
              do k=1, N
                a(k) = a(k) + a(k)*d_c
              enddo
            enddo
c           --- Send data
            if (myid .eq. 0) then
              do i=1, N
                a(i) = 3.14159265d0
              enddo
              do i=1, numprocs-1
                call MPI_ISEND(a, N, MPI_DOUBLE_PRECISION,i,i_loop,
     &               MPI_COMM_WORLD, irequest(i), ierr)
              enddo
            else
              call MPI_RECV(a, N, MPI_DOUBLE_PRECISION, 0, i_loop,
     &              MPI_COMM_WORLD, istatus, ierr)
            endif
c           --- After Load
            do i=1,(numprocs-myid)
               d_c = 1.0d0/dble(i)
               do k=1, N
                  b(k) = a(k) + a(k)*d_c
               enddo
            enddo
c          --- check to finish sending
           if (myid .eq. 0) then
              do i=1, numprocs-1
                call MPI_WAIT(irequest(i), istatus, ierr)
              enddo
           endif
         enddo
```

```
      call MPI_BARRIER(MPI_COMM_WORLD, ierr)
      t2 = MPI_WTIME(ierr)
      t0 = t2 - t1
      call MPI_REDUCE(t0, t_w, 1, MPI_DOUBLE_PRECISION,
     &         MPI_MAX, 0, MPI_COMM_WORLD, ierr)
      if (myid .eq. 0) then
        print *," N = ", N
        print *,"  Execution time with MPI_Isend: ",t_w,"[sec.]"
      endif
      N=N+2
    enddo
    call MPI_FINALIZE(ierr)
    stop
    end
```

以上