HITACHI SR16000/M1 チューニング連載講座

４．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、HITACHI SR16000/M1（以降、SR16K 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、日立製作所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]等がありますので、そちらをご覧ください。

本稿では、SR16K の MPI チューニング技法うち、(１)プロセスバインド方法、(２)MPI_ALLTOALL の性能改善方法、（３）非同期通信を用いた高速化、について説明します。

２．SR16K のハードウェア特徴を考慮した MPI 最適化手法

２．１　ハードウェアのおさらい

前回の「HITACHI SR16000/M1 チューニング連載講座　２．単体ノード性能チューニング」記事で紹介したように、SR16K の最大の特徴は、１ノード上に物理的に 32 コア（物理コアとよぶ）からなる高スレッド並列の並列計算機であることです。SR16K のノードは、４つのチップ（もしくは、**ソケット**とよぶことがあります）で構成されています。

ノード内のメモリは物理的には離れていますが結線されており、共有メモリを構成します。ただし、チップに物理的に近いメモリと、物理的に遠いメモリがあるため、各チップは物理的に近いメモリへのアクセス（ローカルメモリ・アクセス）は高速ですが、物理的に遠いメモリへのアクセス（リモートメモリ・アクセス）は低速になります。このように、共有メモリ内のメモリアクセス速度が、ローカルメモリとリモートメモリで異なる構成を **NUMA(Non Uniform Memory Access)構成**と呼びます。

NUMA 構成のメモリをもつ計算機では、スレッド実行や MPI などでのプロセス実行をするとき、スレッドおよびプロセスのジョブを、どの物理コアに割り当てるか、および、割り当てたジョブが、どのメモリに配列確保するのかが、性能に影響を及ぼします（プロセス割り当てをしないと性能が変動し、チューニングが困難になります)。したがってユーザは、ジョブ割り当てやメモリ割り当ての方法を明示的に指定する必要があります。これが NUMA 構成の計算機における、一つのチューニング手法となります。

本節ではまず、MPI プロセスを固定する方法について紹介します。

２．２　bindprocessor コマンドによる MPI プロセスの割り当て

bindprocessor コマンド[2]は、IBM 社の開発した UNIX 系 OS の AIX(Advanced Interactive eXecutive）で提供されているコマンドです。bindprocessor コマンドにより、プロセスのカーネル・スレッドをプロセッサに割り当て（バインド)、または、取り外し（アンバインド）でき

ます[1]。以下に bindprocessor コマンドの構文を示します。

構文
bindprocessor Process [ProcessorNum] | ***-q*** | ***-u*** Process { ProcessID [ProcessorNum] | ***-u*** ProcessID | ***-s*** SmtSetID | ***-b*** bindID ProcessorNum | ***-q*** }

ここで、オプションの説明は以下です。

- ***-b*** ：アプリケーションの全スレッドを、同じ物理プロセッサのハードウェア・スレッドにバインドします。
- ***-q*** ：使用可能なプロセッサを表示します。
- ***-s*** ：各プロセッサの一覧を別個に作成し、アプリケーションの全スレッドを個別の物理プロセッサにバインドします。
- ***-u*** ：指定したプロセスのスレッドをアンバインドします。

使用例は以下になります。

- プロセス 19254 内のスレッドをプロセッサ 1 にバインドする。
  - ***bindprocessor*** 19254 1

具体的に SR16K/M1 において、1 ノード 64 スレッドの SMT 実行を、MPI で行う場合の MPI プロセスと、プロセッサに対する割り当ては、以下のようにします。

■ジョブスクリプト中の記述例

```
#!/bin/ksh93
#@$-q  parallel
#@$-N  8
#@$-J  T32
#@$-lM 170GB
#@$-lT 02:00:00
export MEMORY_AFFINITY=MCM
poe  ./exe.sh  ./a.out
```

■exe.sh の中身

```
#!/bin/ksh93
task="${MP_COMMON_TASKS%%:*}"
task=`expr $task + 1`
ndrank=`expr $MP_CHILD % $task`
ndrank=`expr $ndrank ¥* 2`
/usr/sbin/bindprocessor $$ $ndrank
$1 &
/usr/sbin/bindprocessor -u $$
wait
```

[1] bindprocessor は、ハイブリッド MPI 実行では使えません。また、より容易に MPI プロセスの割り当てを行うためには、mpibind コマンドを用いることを推奨いたします。mpibind コマンドの詳細は[5]をご覧ください。

exe.sh では、環境変数 MP_COMMON_TASK で MPI ランクを取得します。また、変数 ndrank で、ノード内で割り付ける 64 プロセス番号を同定します。

具体的には、以下のような命令が、全 MPI プロセス（この場合は、256 プロセス）で発行されることになります。ランク 255 のプロセスは以下のようになります。

```
MP_COMMON_TASKS=31:224:225:226:227:228:229:230:231:232:233:234:235:236:237:
   238:239:240:241:242:243:244:245:246:247:248:249:250:251:252:253:254
MP_CHILD=255
task=32
ndrank=62
/usr/sbin/bindprocessor  2622128  62      ←バインドの実行
/usr/sbin/bindprocessor  -u  2622128      ←アンバインドの実行
```

２．３　MPI_ALLTOALL 性能改善のための環境変数

MPI プログラム中に、MPI_ALLTOALL 関数が使われている場合、MPI プロセスのコアへの割り当てについて、通信トポロジーを考慮して変更すると高速化されることがあります。

ここで、ベンチマークとして、HPCC(HPC Challenge)[3]の１つである、mpifft を考えます。ここでは、HPCC 1.4.0 の mpifft を SR16K で実行させた結果を載せます。mpifft は、MPI を用いて並列に FFT を行うプログラムであり、MPI_ALLTOALL が使われています。

東京大学情報基盤センターの実行環境では、Parallel Environment Runtime Edition (MPI 並列実行環境、以降 PE と記載)の仕様変更により、PE の実行時に参照される環境変数 MP_S_IGNORE_COMMON_TASKS に YES を設定することで、性能改善することがあります。

具体的な例を、以下に載せます。

- HPCC 1.4.0 mpifft
  - N = 320000
  - P = 16, Q=16 (8 ノード、256MPI 実行)
- 環境変数適用前：116.94 GFLOPS
- **環境変数適用後：142.68 GFLOPS** (22%の速度向上)

以下に、MP_S_IGNORE_COMMON_TASKS 環境変数の設定例(ジョブスクリプト)を載せます。

■MP_S_IGNORE_COMMON_TASKS 環境変数の設定例

```
#!/bin/bash
#@$-q  parallel
#@$-N  8
#@$-J  T32
…(省略)
export MP_S_IGNORE_COMMON_TASKS=yes
poe ./exe.sh ../hpcc-1.4.0/hpcc
```

２．４　非同期通信および永続的な１対１通信による高速化

非同期通信と永続的通信の概要

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[2]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出来ます。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(0:N-1)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(0:N-1)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(0:N-1)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(0:N-1)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(0:N-1)を非同期通信で送信

真の同期すべき場所

[2] ただし、MPI のデフォルト通信モードでは、＜標準通信モード＞になっているため、システムバッファに載るサイズの通信メッセージを転送する場合には、MPI_SEND の挙動はシステムバッファにデータをコピーして、すぐにプログラムに戻る挙動になります。したがってこの状況では、ブロッキング（同期）はなされません。

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_ISND 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。

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。

永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。

■永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(0:N-1)の定義 */
<3>    先述のランク 0 が a(0:N-1)を転送し、それ以外のランクは受信する処理
<4>    /* a(0:N-1)を用いた後処理：a(0:N-1)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 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)*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(0:N-1)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早く、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。

永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、永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、SR16K の非同期通信の性能を評価しました。利用した SR16K のノードは 8 ノード、1 ノードあたり 64MPI プロセス実行(SMT 実行)です。総 MPI プロセス数は 512 プロセスです。

実験結果を、図 1～図 4 に結果を載せます。また、N=1～100 の実行においては、そのままだと実行時間がばらつきました。そこでこの実行結果のばらつきを防ぐため、2.1 節で説明した bindprocessor を用いて、MPI プロセス割り当てコアを固定し実行しました。

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、$O(N^2)$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 $O(N^2)$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。

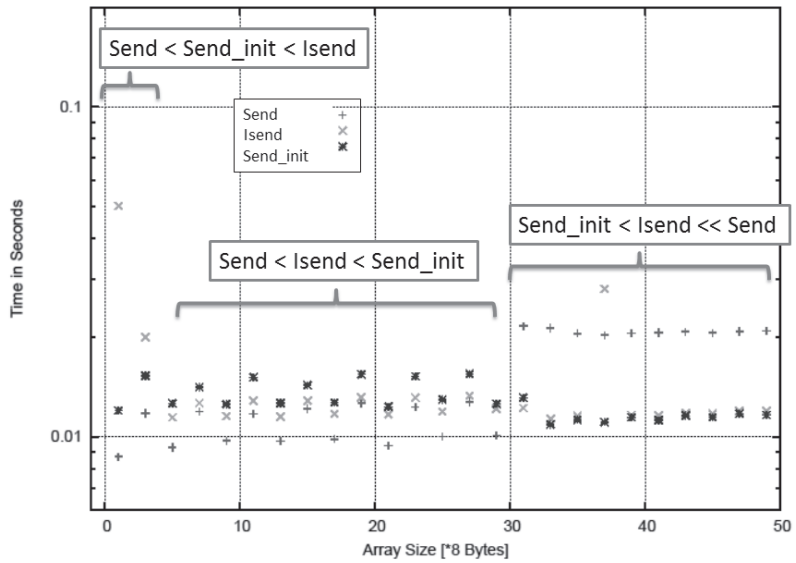

図 1　非同期通信の効果

(N=1～50 で 2 間隔。 1 通信当たりのメッセージサイズ 8 バイト～400 バイト。)

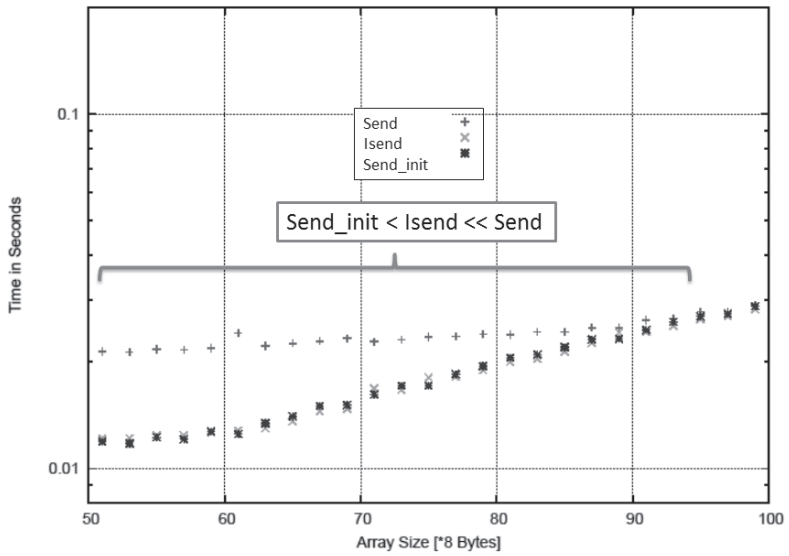

図 2 非同期通信の効果

(N=50～100 で 2 間隔。1 通信当たりのメッセージサイズ 400 バイト～800 バイト。)

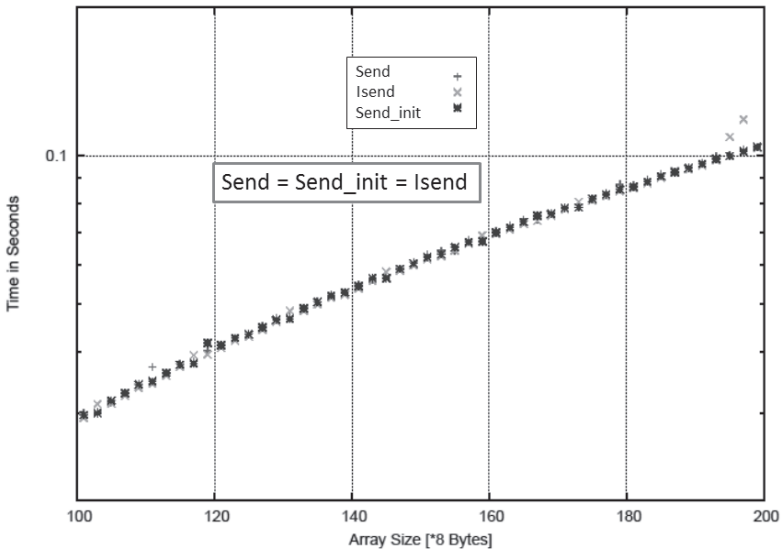

図 3 非同期通信の効果

(N=100～200 で 2 間隔。1 通信当たりのメッセージサイズ 800 バイト～1.6K バイト。)

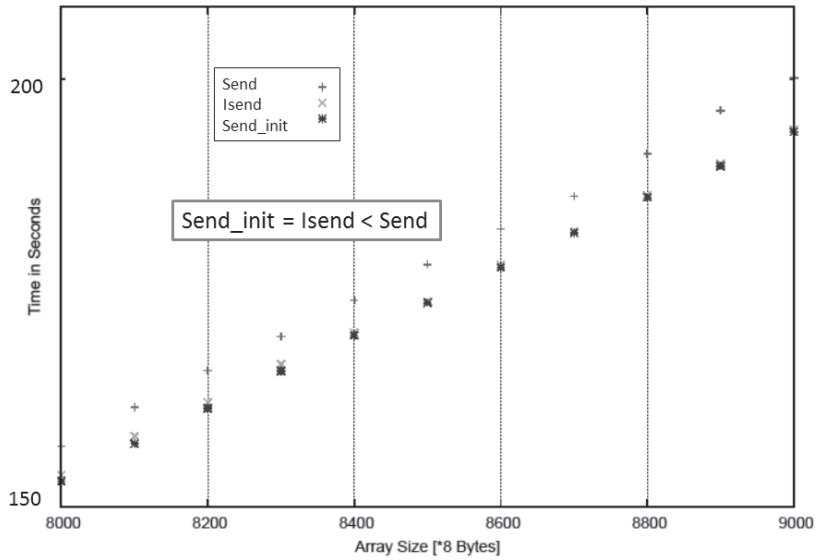

図 4 非同期通信の効果

(N=8000～9000 で 100 間隔。1 通信当たりのメッセージサイズ 64K バイト～72K バイト。)

考察

図 1 では、有効となる通信方式が N に応じて変わります。

N=1、3 では MPI_SEND が最高速です。具体的に N=1 の時、0.0087[秒](MPI_SEND)、0.0120[秒](MPI_SEND_INIT)、0.0500[秒](MPI_ISEND)です。

N=5～29 では、やはり MPI_SEND が最高速ですが、次に速いのが MPI_ISEND になります。N=29 では、0.0101[秒](MPI_SEND)、0.0121[秒](MPI_ISEND)、0.0125[秒](MPI_SEND_INIT)となります。このとき、1 通信当たりのメッセージサイズが 8 バイト～232 バイトですが、MPI_SEND が速い理由は、標準通信モードによりシステムバッファに全てメッセージがバッファされてプログラムに戻ることで、非同期通信より効率よい実装となったことが理由と考えられます。

N=31 以上 (248 バイト以上) になると、最高速なのは MPI_SEND_INIT になります。これは、期待する非同期通信による計算と通信のオーバーラップによる高速化と思われます。N=49 では、0.0116[秒] (MPI_SEND_INIT)、0.0119[秒](MPI_ISEND)、0.0209[秒] (MPI_SEND)、となります。またこの結果から、MPI_ISEND は内部で通信と計算とがオーバーラッピングする実装になっているものと推定されます。

図 2 では、最高速になるのは MPI_SEND_INIT で変わりませんが、MPI_SEND との速度差が N が増加するに従い無くなっていきます。N=99 (792 バイト) あたりで、3 者の性能が均衡します。また図 3 から、この性能差がない状態は、N=200(1.6K バイト)まで確認できます。この理由は不明ですが、一般に非同期通信のための余分なオーバーヘッド (システム内でのコピー作業) と、同期通信によるコピーなどによる余分なオーバーヘッドがないことによるコスト均衡から生じることになります。

図 4 では、再び非同期通信の効果が出てきます。図 4 では、N=8000 以上 (64K バイト以上) とメッセージサイズが大きく、前処理・後処理が計算量の観点での散らばりが大きいケースで

す。したがって、計算量的に負荷バランス劣化が生じるので、その差が出ているものと考えられます。具体的な実行時間は、N=9000 で、193.0[秒](MPI_SEND_INIT)、193.3[秒](MPI_ISEND)、200.1[秒](MPI_SEND)となります。

以上から、単純に非同期通信を導入すれば高速化されるわけではないことが言えます。また、同期通信が高速な場合もあります。したがって、メッセージサイズ、前処理・後処理の計算量、実行問題サイズ、を勘案して非同期通信を導入すべきといえます。

６．おわりに

本稿では、HITACHI SR16000/M1（以降、SR16K）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。

SR16K では、プロセスの固定、MPI_ALLTOALL のための環境変数、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。

参　考　文　献

[1]（株）日立製作所：スーパテクニカルサーバ HITACHI SR16000 モデル M1 チューニングマニュアル（2011）https://yayoi-man.cc.u-tokyo.ac.jp/manual-j/index.html

[2] Bindprocessor コマンド資料：
http://pic.dhe.ibm.com/infocenter/aix/v6r1/index.jsp?topic=%2Fcom.ibm.aix.cmds%2Fdoc%2Faixcmds1%2Fbindprocessor.htm

[3] HPCC ベンチマーク　http://icl.cs.utk.edu/hpcc/index.html

[4] MP_S_IGNORE_COMMON_TASKS 関連記事、
http://www-01.ibm.com/support/docview.wss?uid=isg1IV00043

[5]（株）日立製作所：大規模 SMP 並列 スーパーコンピューターシステム ジョブ実行方法（2011） https://yayoi-man.cc.u-tokyo.ac.jp/manual-j/index.html

付録

２．４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(0:512)
```

```
      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
             do j=1, N
               d_c = 1.0d0/(dble(i)+dble(j))
               do k=1, N
                 a(k) = a(k) + a(k)*d_c
               enddo
            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)
             do j=1, N
               d_c = 1.0d0/(dble(i) + dble(j))
               do k=1, N
                   b(k) = a(k) + a(k)*d_c
               enddo
             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