# 「情報システム学特別講義３」
# 非同期通信

東京大学情報基盤センター准教授　片桐孝洋

２０１４年７月１５日（火）１４：４０－１６：１０



iTC
東京大学情報基盤センター
Information Technology Center,The University of Tokyo

# 講義日程
## （情報システム学特別講義３）

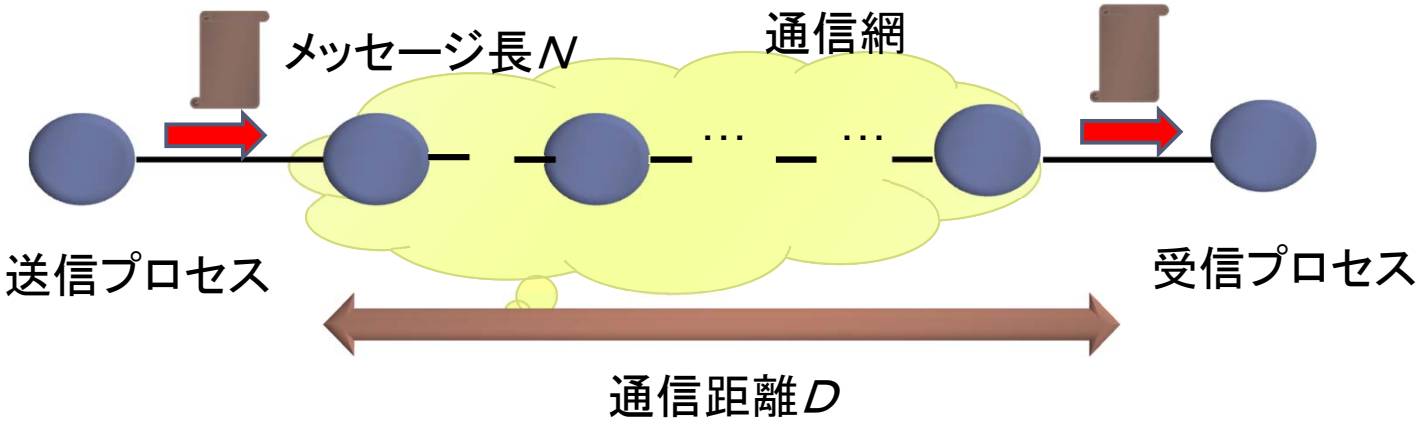
**レポートおよびコンテスト課題**
**（締切：**
**2014年8月11日（月）24時　厳守**

1. ~~4月8日：ガイダンス~~
2. ~~4月15日~~
   - ~~プログラム高速化の基礎（その1）~~
3. ~~4月22日~~
   - ~~プログラム高速化の基礎（その2）~~
4. ~~5月13日~~
   - ~~MPIの基礎~~
5. ~~5月20日~~
   - ~~OpenMPの基礎~~
6. ~~5月27日~~
   - ~~Hybrid並列化技法（MPIとOpenMPの応用編）~~
7. ~~6月3日~~
   - ~~プログラム高速化の応用~~
8. ~~6月10日~~
   - ~~行列ーベクトル積の並列化~~
9. ~~6月17日~~
   - ~~べき乗法の並列化~~
10. ~~6月24日~~
    - ~~行列ー行列積の並列化~~
11. ~~7月8日~~
    - ~~LU分解の並列化~~
12. 7月15日
    - 非同期通信
    - 疎行列反復解法の並列化
13. 7月22日
    - ソフトウェア自動チューニング
14. 8月5日（補講日）
    - エクサフロップスコンピューティングに向けて

# 通信最適化の方法

# 通信最適化の前に：通信時間のモデル化

- 通信時間を見積もるには？
- 通信時間　$t$ [秒]は、メッセージサイズ$N$ [byte]、プロセス数$p$、通信相手先との距離$D$、などなどの関数と考えられる
  - $t = f\ (N, p, D,$ などなど$)$
  - これでは、ちょっと難しすぎる



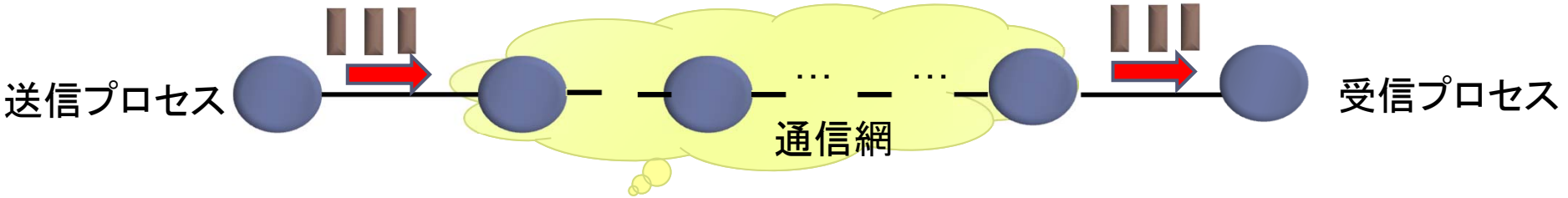
# 単純な通信モデルの採用：線形一次近似

- そこで、単純に、線形一次近似モデルで、通信時間の近似をすることが多い
  - $t = \alpha + \beta N$
  - $\alpha$ ： 通信立ち上がり時間（通信レイテンシ）[秒]
    - 0バイトのデータを送るときの時間
    - 通信オーバーヘッド
    - 小さいほど、性能が良い
  - $\beta$ ： [秒/バイト]
    - $1/\beta$: 通信バンド幅 [バイト/秒]
      - 1秒あたり送れるデータ長
      - 大きいほど、性能が良い
- 欠点
  - 通信距離$D$、メッセージ量の大きさの違い、などが表現されない

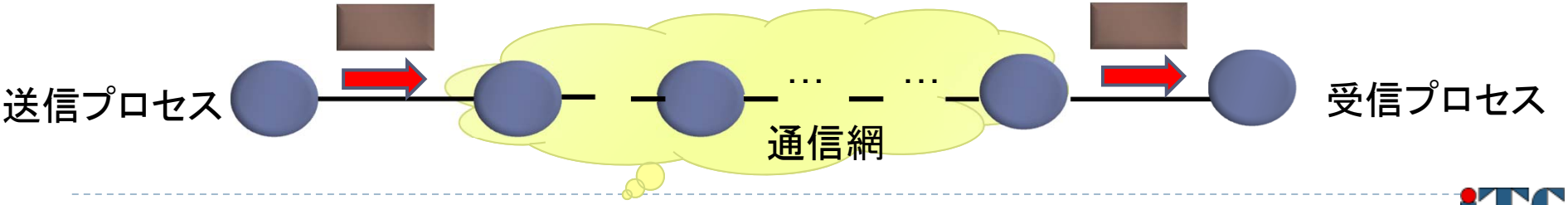
# 通信時間の予測における線形一次近似モデルの欠点

- 通信サイズNの大きさによる、α、βの値の違い
- 通信実装の方式（通信プロトコル）から、通常は、通信サイズの大きさを見て、通信のやり方を変える実装

● メッセージサイズNが小さい場合
  ● αを最小化するのが効果あるため、小さいパケットで送信



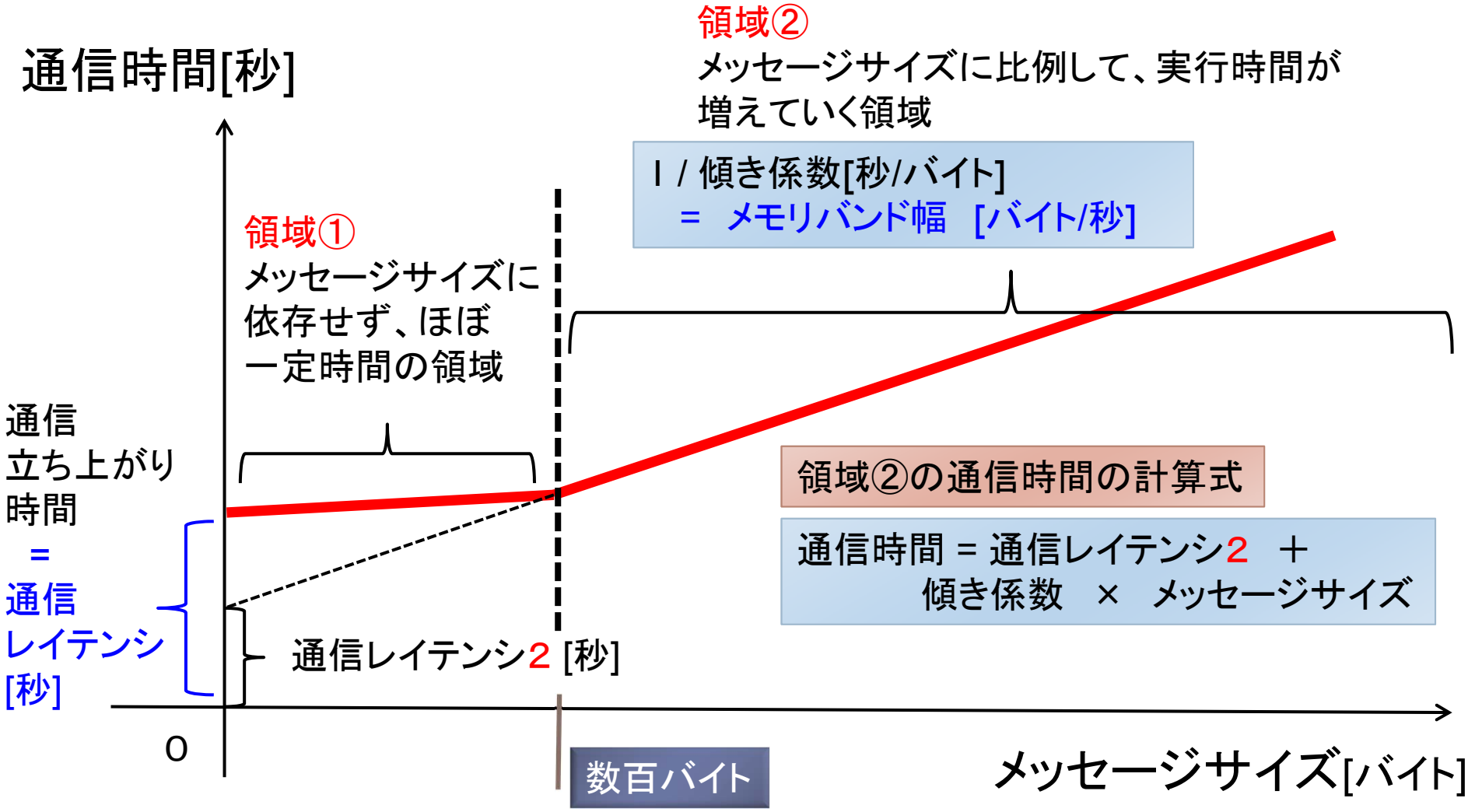

● メッセージサイズNが大きい場合
  ● βを最大化するのが効果あるため、大きいパケットで送信



# ここでのまとめ

- 線形一次近似モデルによる通信時間の予測では、少なくとも、メッセージサイズに応じて、パラメタであるαとβをきめる必要がある
  - ⇒メッセージサイズに応じた、２種以上のパラメタによる、線形一次近似によるモデル化が必要

# 想定する通信時間のモデル



# 通信最適化時に注意すること（その１）

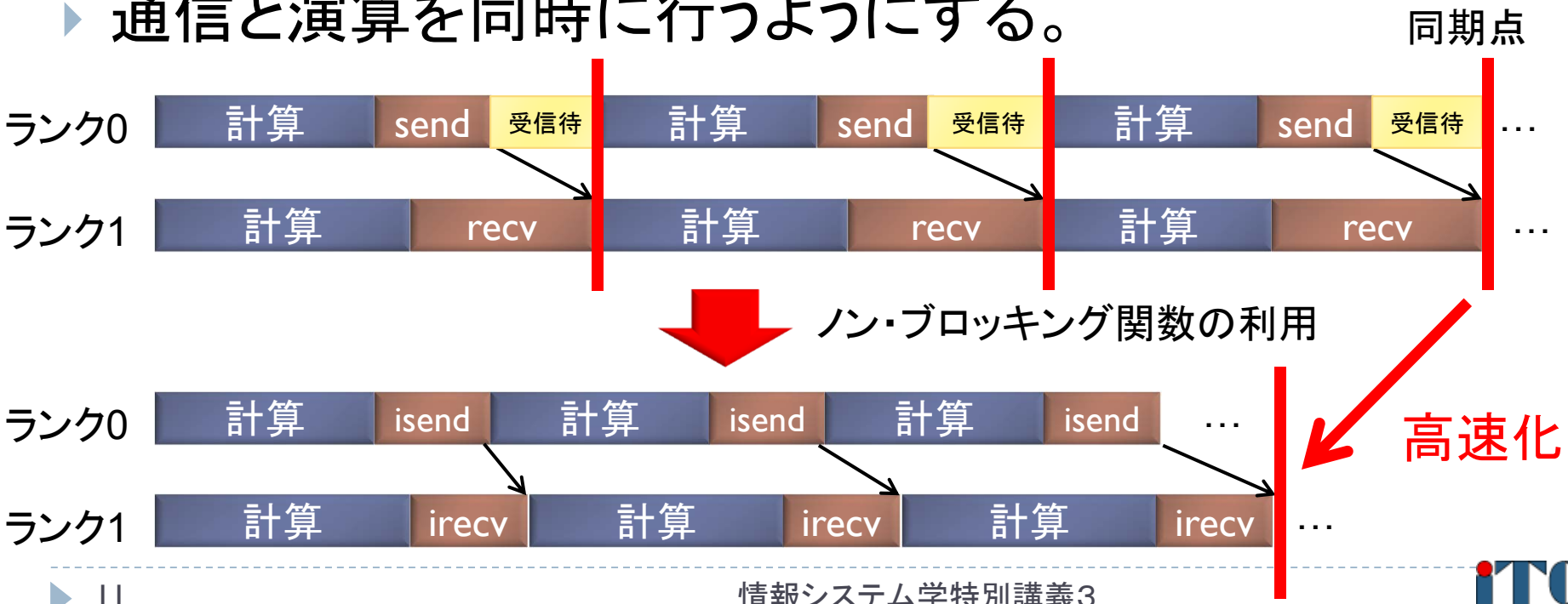
- 自分のアプリケーションの通信パターンについて、以下の観点を知らないと通信の最適化ができない
  - ＜領域①＞＜領域②＞のどちらになるのか
  - 通信の頻度（回数）はどれほどか
- 領域①の場合
  - 「通信レイテンシ」が実行時間のほとんど
  - 通信回数を削減する
    - 細切れに送っているデータをまとめて1回にする、など
- 領域②の場合
  - 「メッセージ転送時間」が実行時間のほとんど
  - メッセージサイズを削減する
  - 冗長計算をして計算量を増やしてでもメッセージサイズを削減する、など

# 領域①となる通信の例

- 内積演算のためのリダクション(MPI_Allreduce)などの送信データは倍精度1個分(8バイト)
- 8バイトの規模だと、数個分を同時にMPI_Allreduceする時間と、1個分をMPI_Allreduceをする時間は、ほぼ同じ時間となる
  - ⇒複数回分の内積演算を一度に行うと高速化される可能性あり
- 例)連立一次方程式の反復解法CG法中の内積演算
  - 通常の実装だと、1反復に3回の内積演算がある
  - このため、内積部分は通信レイテンシ律速となる
  - k反復を1度に行えば、内積に関する通信回数は1/k回に削減
    - ただし、単純な方法では、丸め誤差の影響で収束しない。
    - 通信回避CG法(Communication Avoiding CG, CACG)として現在活発に研究されている。

# 通信最適化時に注意すること（その２）

- 「同期点」を減らすことも高速化につながる
  - MPI関数の「ノン・ブロッキング関数」を使う
  - 例：　ブロッキング関数 MPI_SEND()
    → ノン・ブロッキング関数 MPI_ISEND()
  - 通信と演算を同時に行うようにする。



# 非同期通信：Isend、Irecv、永続的通信関数

# ブロッキング通信で効率の悪い例

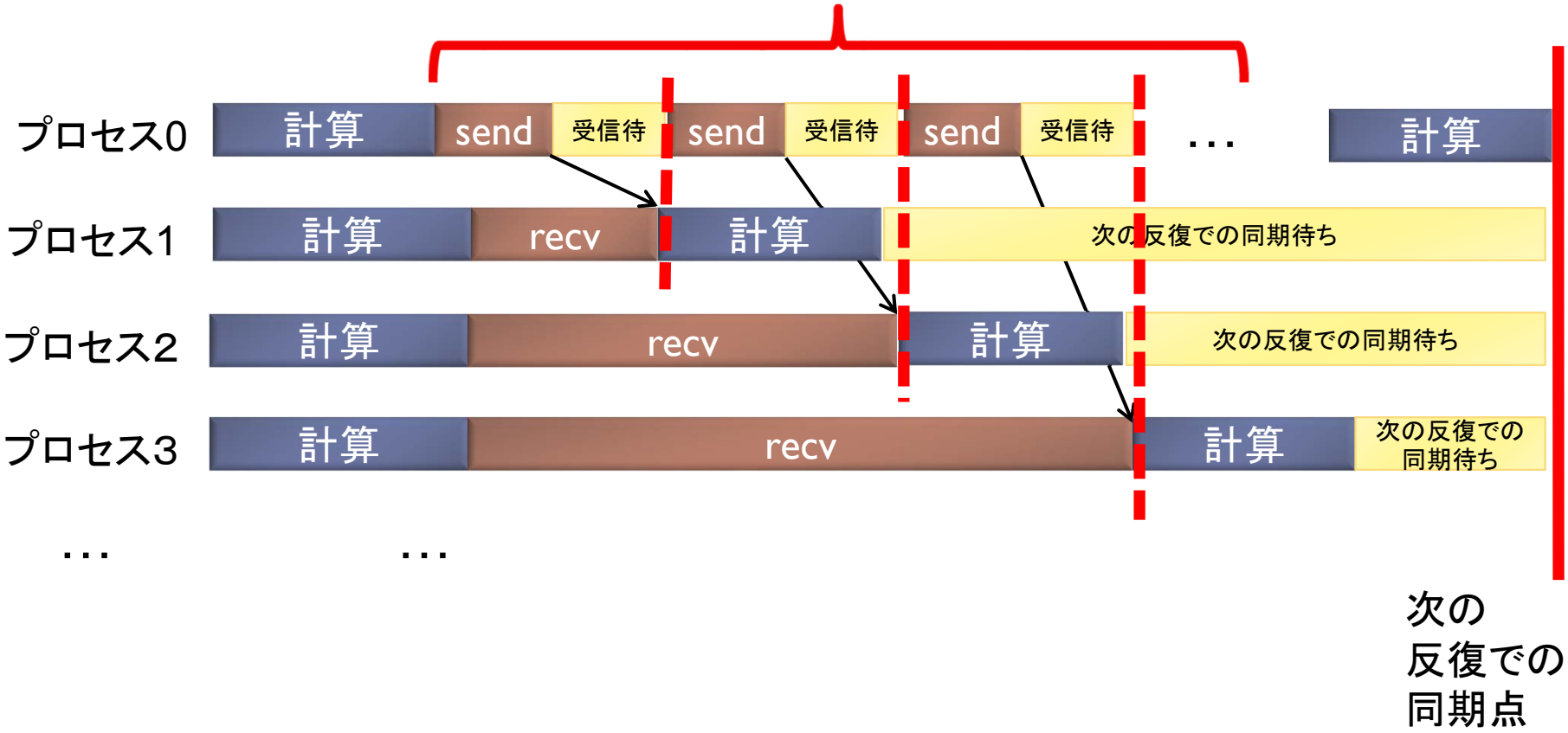
- プロセス0が必要なデータを持っている場合

```
double a[N];
MPI_Status istatus;
…
for (iter=0;  iter<MAX_ITER;  iter++) {  /* ←反復解法のメインループ  */
   …
   if (myid == 0) {
     for (i=1; i<numprocs; i++) {
        ierr = MPI_Send(a, N, MPI_DOUBLE, i, iter, MPI_COMM_WORLD);
     }
   } else {
        ierr = MPI_Recv(a, N, MPI_DOUBLE, 0, iter, MPI_COMM_WORLD, istatus);
   }
  …
   if (解が収束する)  break;
   MPI_Barrier(MPI_COMM_WORLD);
}
```

# ブロッキング通信で効率の悪い例

- プロセス0が必要なデータを持っている場合

連続するsendで、効率の悪い受信待ち時間が多発



# ここでの目的

- 非同期通信(ノン・ブロッキング通信)を用いて、通信待ち時間を減らす

  ⇒通信と演算の重複(オーバーラップ)をする

  ⇒見かけの通信時間を減らす

  (通信隠ぺい)

# １対１通信に対するMPI用語

ブロッキング？ノンブロッキング？

# ブロッキング、ノンブロッキング

1. ブロッキング
   - 送信／受信側のバッファ領域にメッセージが格納され、受信／送信側のバッファ領域が自由にアクセス・上書きできるまで、呼び出しが戻らない
   - バッファ領域上のデータの一貫性を保障
2. ノンブロッキング
   - 送信／受信側のバッファ領域のデータを保障せずすぐに呼び出しが戻る
   - バッファ領域上のデータの一貫性を保障せず
     - 一貫性の保証はユーザの責任

# ローカル、ノンローカル

- ローカル
  - 手続きの完了が、それを実行しているプロセスのみに依存する。
  - ほかのユーザプロセスとの通信を必要としない処理。
- ノンローカル
  - 操作を完了するために、別のプロセスでの何らかのMPI手続きの実行が必要かもしれない。
  - 別のユーザプロセスとの通信を必要とするかもしれない処理。

# 通信モード（送信発行時の場合）

1. 標準通信モード（ノンローカル）：デフォルト
   - 送出メッセージのバッファリングはMPIに任せる。
     - バッファリングされるとき：相手の受信起動前に送信を完了可能；
     - バッファリングされないとき：送信が完全終了するまで待機；
2. バッファ通信モード（ローカル）
   - 必ずバッファリングする。バッファ領域がないときはエラー。
3. 同期通信モード（ノンローカル）
   - バッファ領域が再利用でき、かつ、対応する受信／送信が開始されるまで待つ。
4. レディ通信モード（処理自体はローカル）
   - 対応する受信が既に発行されている場合のみ実行できる。それ以外はエラー。
     - ハンドシェーク処理を無くせるため、高い性能を発揮する。

# 実例－MPI_Send

- MPI_Send関数
  - ブロッキング
  - 標準通信モード（ノンローカル）
    - バッファ領域が安全な状態になるまで戻らない
    - バッファ領域がとれる場合：
      メッセージがバッファリングされる。対応する受信が起動する前に、送信を完了できる。
    - バッファ領域がとれない場合：
      対応する受信が発行されて、かつ、メッセージが受信側に完全にコピーされるまで、送信処理を完了できない。

# 非同期通信関数

```
ierr = MPI_Isend(sendbuf, icount, datatype,
    idest, itag, icomm, irequest);
```

- sendbuf：送信領域の先頭番地を指定する
- icount：整数型。送信領域のデータ要素数を指定する
- datatype：整数型。送信領域のデータの型を指定する
- idest：整数型。送信したいPEのicomm 内でのランクを指定する
- itag：整数型。受信したいメッセージに付けられたタグの値を指定する

# 非同期通信関数

- icomm： 整数型。PE集団を認識する番号であるコミュニケータを指定する。
  - 通常ではMPI_COMM_WORLD を指定すればよい。
- irequest： MPI_Request型（整数型の配列）。送信を要求したメッセージにつけられた識別子が戻る。
- ierr： 整数型。エラーコードが入る。

# 同期待ち関数

- ierr = MPI_Wait(irequest, istatus);

  - irequest： MPI_Request型（整数型配列）。
    送信を要求したメッセージにつけられた識別子。
  - istatus： MPI_Status型（整数型配列）。
    受信状況に関する情報が入る。
    - 要素数がMPI_STATUS_SIZEの整数配列を宣言して指定する。
    - 受信したメッセージの送信元のランクがistatus[MPI_SOURCE] 、タグがistatus[MPI_TAG] に代入される。

# 実例－MPI_Isend

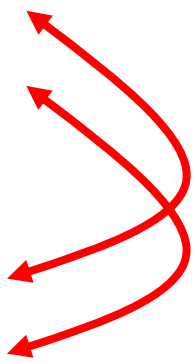
- MPI_Isend関数
  - ノンブロッキング
  - 標準通信モード（ノンローカル）
    - 通信バッファ領域の状態にかかわらず戻る
    - バッファ領域がとれる場合は、メッセージがバッファリングされ、対応する受信が起動する前に、送信処理が完了できる
    - バッファ領域がとれない場合は、対応する受信が発行され、メッセージが受信側に完全にコピーされるまで、送信処理が完了できない
    - MPI_Wait関数が呼ばれた場合の振舞いと理解すべき。

# 注意点

- 以下のように解釈してください：
  - MPI_Send関数
    - 関数中にMPI_Wait関数が入っている；
  - MPI_Isend関数
    - 関数中にMPI_Wait関数が入っていない；
    - かつ、すぐにユーザプログラム戻る；

# 並列化の注意（MPI_Send、MPI_Recv）

- 全員がMPI_Sendを先に発行すると、その場所で処理が止まる。(cf. 標準通信モードを考慮)
  （正確には、動いたり、動かなかったり、する）
  - MPI_Sendの処理中で、場合により、バッファ領域がなくなる。
  - バッファ領域が空くまで待つ（スピンウェイトする）。
  - しかし、送信側バッファ領域不足から、永遠に空かない。
- これを回避するためには、例えば以下の実装を行う。
  - ランク番号が2で割り切れるプロセス：
    - MPI_Send();
    - MPI_Recv();
  - それ以外：
    - MPI_Recv();
    - MPI_Send();



それぞれに対応

# 非同期通信 TIPS

- メッセージを完全に受け取ることなく、受信したメッセージの種類を確認したい
  - 送信メッセージの種類により、受信方式を変えたい場合
  - MPI_Probe 関数（ブロッキング）
  - MPI_Iprobe 関数（ノンブロッキング）
  - MPI_Cancel 関数（ノンブロッキング、ローカル）

# MPI_Probe 関数

- ierr = MPI_Probe(isource, itag, icomm, istatus) ;
  - isource: 整数型。送信元のランク。
    - MPI_ANY_SOURCE (整数型)も指定可能
  - itag: 整数型。タグ値。
    - MPI_ANY_TAG (整数型) も指定可能
  - icomm: 整数型。コミュニケータ。
  - istatus : ステータスオブジェクト。
  - isource, itagに指定されたものがある場合のみ戻る

# MPI_Iprobe関数

- ierr = MPI_Iprobe(isource, itag, icomm, iflag, istatus) ;
  - isource: 整数型。送信元のランク。
    - MPI_ANY_SOURCE (整数型) も指定可能。
  - itag: 整数型。タグ値。
    - MPI_ANY_TAG (整数型) も指定可能。
  - icomm: 整数型。コミュニケータ。
  - iflag: 論理型。isource, itagに指定されたものがあった場合はtrueを返す。
  - istatus : ステータスオブジェクト。

# MPI_Cancel 関数

- ierr = MPI_Cancel(irequest);
  - irequest: 整数型。通信要求(ハンドル)

- 目的とする通信が実際に取り消される以前に、可能な限りすばやく戻る。
- 取消しを選択するため、MPI_Request_free関数、MPI_Wait関数、又は MPI_Test関数（または任意の対応する操作）の呼出しを利用して完了されている必要がある。

# ノン・ブロッキング通信例（C言語）

```
if (myid == 0)  {
      …
    for (i=1;  i<numprocs;  i++)  {
        ierr = MPI_Isend( &a[0],  N,  MPI_DOUBLE,  i,
            i_loop,  MPI_COMM_WORLD,  &irequest[i] );
    }
 }  else  {
    ierr = MPI_Recv( &a[0],  N,  MPI_DOUBLE,  0,  i_loop,
                MPI_COMM_WORLD,  &istatus );
}
          a[ ]を使った計算処理;
if (myid == 0)   {
     for (i=1;  i<numprocs;  i++)  {
         ierr = MPI_Wait(&irequest[i],  &istatus);
     }
}
```

ランク0のプロセスは、
ランク1~numprocs-1までのプロセス
に対して、ノンブロッキング通信を
用いて、長さNのDouble型配列
データを送信

ランク1~numprocs-1までの
プロセスは、ランク0からの
受信待ち。

プロセス0は、recvを
待たず計算を開始

ランク0のPEは、
ランク1~numprocs-1までのプロセス
に対するそれぞれの送信に対し、
それぞれが受信完了するまで
ビジーウェイト（スピンウェイト）
する。

# ノン・ブロッキング通信の例（Fortran言語）

```
if (myid .eq. 0)  then
      …
    do i=1, numprocs - 1
       call MPI_ISEND( a,  N,  MPI_DOUBLE_PRECISION,
          i,  i_loop,  MPI_COMM_WORLD,  irequest, ierr )
    enddo
else
       call  MPI_RECV( a,  N,  MPI_DOUBLE_PRECISION ,
          0,  i_loop, MPI_COMM_WORLD,  istatus,  ierr )
endif
          a( )を使った計算処理
if (myid .eq.  0) then
     do  i=1,  numprocs - 1
        call  MPI_WAIT(irequest(i),  istatus, ierr )
     enddo
endif
```

ランク0のプロセスは、ランク1~numprocs-1までのプロセスに対して、ノンブロッキング通信を用いて、長さNのDOUBLE PRECISION型配列データを送信

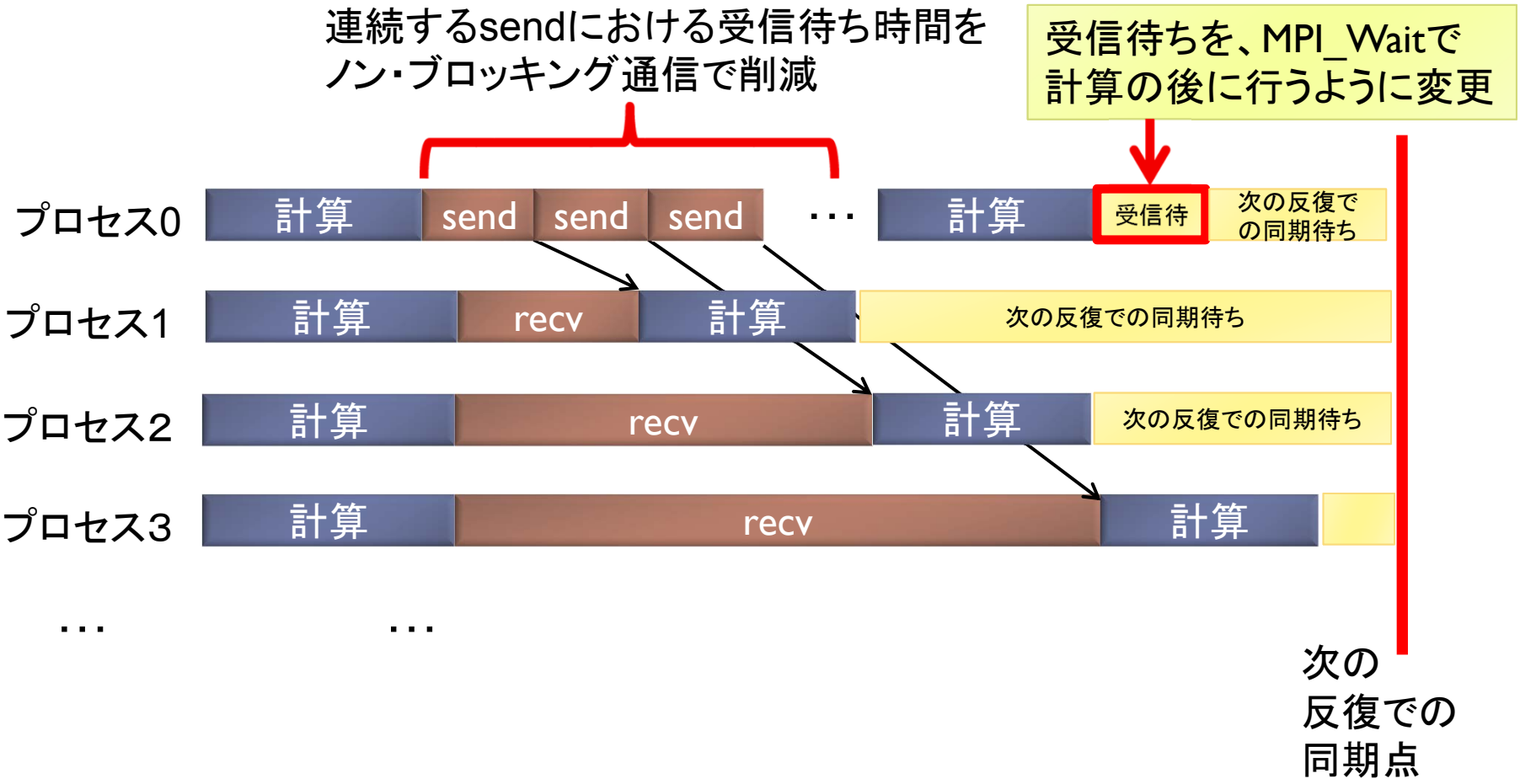
ランク1~numprocs-1までのプロセスは、ランク0からの受信待ち。

プロセス0は、recvを待たず計算を開始

ランク0のプロセスは、ランク1~numprocs-1までのプロセスに対するそれぞれの送信に対し、それぞれが受信完了するまでビジーウェイト（スピンウェイト）する。

# ノン・ブロッキング通信による改善

- プロセス0が必要なデータを持っている場合



# 永続的通信

# 永続的通信（その１）

- ノン・ブロッキング通信は、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意味がない。
- 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る。
  - この場合には、ノン・ブロッキング通信の効果が全くない。

# 永続的通信（その２）

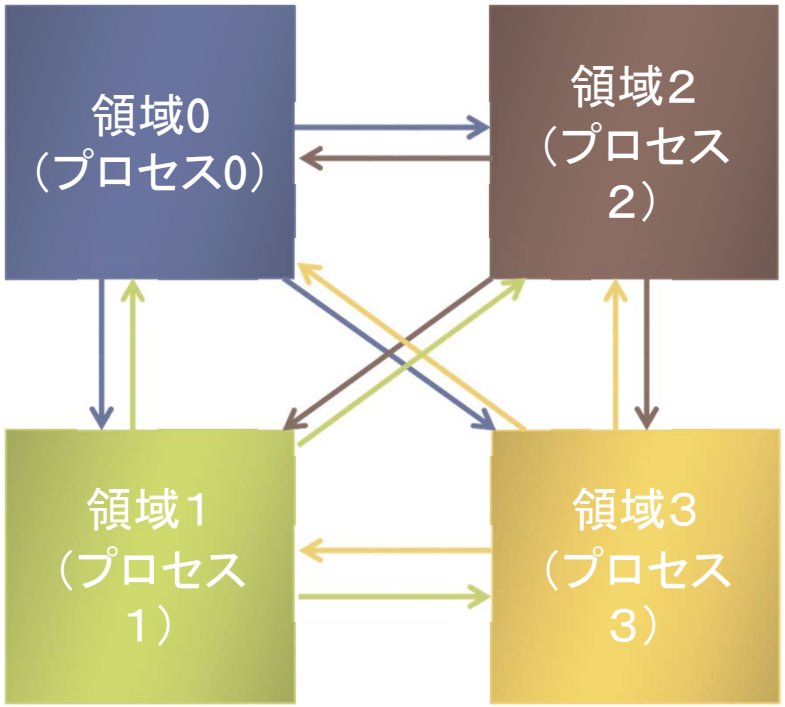
- 永続的通信（Persistent Communication）を利用すると、MPIライブラリの実装に依存し、ノン・ブロッキング通信の効果が期待できる場合がある。
  - 永続的通信は、MPI-1からの仕様（たいていのMPIで使える）
    - しかし、通信と演算がオーバラップできる実装になっているかは別問題

# 永続的通信（その３）

- 永続的通信の利用法
  1. 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を呼ぶ
  2. その後、SENDをする箇所にMPI_START関数を書く
  3. 真の同期ポイントに使う関数(MPI_WAIT等)は、ISENDと同じものを使う
- MPI_SEND_INIT関数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ない
  - 同じ通信相手に何度でもデータを送る場合、通常のノン・ブロッキング通信に対し、同等以上の性能が出ると期待
- 適用例
  - 領域分割に基づく陽解法
  - 陰解法のうち反復解法を使っている数値解法

# 永続的通信が使える例

- 領域分割法による計算
  - 原理的に、隣接領域をもつプロセスにしか、データをSend、Recvしない　←永続的通信が使える



# 永続的通信の実装例（C言語）

```
MPI_Status istatus;
MPI_Request  irequest;
…
if (myid == 0) {
  for (i=1; i<numprocs; i++) {
      ierr = MPI_Send_init (a,   N,   MPI_DOUBLE_PRECISION,   i,
                    0,   MPI_COMM_WORLD,   irequest );
  }
}
…
if (myid == 0) {
  for (i=1; i<numprocs; i++) {
     ierr = MPI_Start ( irequest );
  }
}

/* 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する

ここで、データを送る

# 永続的通信の実装例（Fortran言語）

```
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, ierr )
  enddo
endif

/* 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する

ここで、データを送る

# 永続的通信のまとめ

- 通信処理において、各プロセスの転送先のプロセスに変更がない場合は、永続的通信が使える
- 永続的通信を使うと、以下の点から、通常のノン・ブロッキング通信よりも高速になる可能性がある
  1. 内部的な通信処理の設定部分のオーバヘッドが少ない
  2. MPIの開発者が高速な転送方式（RDMA転送）などを使って高速化しやすい
- 永続的通信を使って、本当に高速化されるかどうかは、提供されているMPIの実装に依存する
  - 本当に高速化されるかは、ノン・ブロッキング通信の提供されているMPIの実装においても同じ

# MPI-3.0の動向

# MPI 3.0と集団通信に対する非同期通信実装

- MPI version 3.0 の仕様では、ノン・ブロッキングの集団通信APIが導入される
- 例) MPI_Allgather
- 従来（ブロッキングのみ）
  - int MPI_Allgather(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
- MPI 3.0
  - int MPI_Iallgather(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)

# 集団通信に対する非同期通信実装の効果

1. いままで、同期関数として書いていた、集団通信関数を非同期化しやすくなる
   - 従来
     - MPI_BCAST(…)
   - MPI 3.0
     - MPI_IBCAST(…) と MPI_Wait()で、非同期化しても通信データの一貫性を保証する書き方で、通信と計算のオーバラッピングを増進
2. いままで、自分で、MPI_Isend, MPI_Irecv, MPI_Wait関数で非同期化していたコードが簡素化、さらに、提供されるMPI関数の実装により高速化

- いずれにせよ、並列アルゴリズムとして非同期化されるプログラム（アルゴリズム）であること（プログラムできること）が必須

# サンプルプログラムの実行（非同期通信）

はじめてのMPI_Isend

# LU分解のサンプルプログラムの注意点

- C言語版／Fortran言語版のファイル名

  lsend-fx.tar

- ジョブスクリプトファイルisend.bash
  - lecture : 実習時間外のキュー

# MPI_Isendのサンプルプログラムの実行（C言語版/Fortran版共通）

- 以下のコマンドを実行する
  $ cp /home/z30082/ISend-fx.tar ./
  $ tar xvf ISend-fx.tar
  $ cd Isend
- 以下のどちらかを実行
  $ cd C ：C言語を使う人
  $ cd F ：Fortran言語を使う人
- 以下共通
  $ make
  $ pjsub isend.bash
- 実行が終了したら、以下を実行する
  $ cat isend.bash.oXXXXXX

# 出力結果

- 以下のような結果が出力される（C言語）

**Execution time using MPI_Isend : 30.3248 [sec.]**

# サンプルプログラムの説明（C言語版）

```
if (myid == 0)  {
      …
    for (i=1;  i<numprocs;  i++)  {
        ierr = MPI_Isend( &a[0],  N,  MPI_DOUBLE,  i,
            i_loop,  MPI_COMM_WORLD,  &irequest[i] );
    }
 }  else  {
    ierr = MPI_Recv( &a[0],  N,  MPI_DOUBLE,  0,  i_loop,
                MPI_COMM_WORLD,  &istatus );
}
…
if (myid == 0)   {
     for (i=1;  i<numprocs;  i++)  {
         ierr = MPI_Wait(&irequest[i],  &istatus);
     }
}
```

ランク0のPEは、
ランク1~191までのPEに対して、
ノンブロッキング通信を用いて、
長さNのDouble型配列データ
を送信

ランク1~191までのPEは、
ランク0からの受信待ち。

ランク0のPEは、
ランク1~191までのPEに対する
それぞれの送信に対し、
それぞれが受信完了するまで
ビジーウェイト（スピンウェイト）
する。

# サンプルプログラムの説明（Fortran言語版）

```
if (myid .eq. 0)  then
      …
    do i=1, numprocs - 1
       call MPI_ISEND( a,  N,  MPI_DOUBLE_PRECISION,
          i,  i_loop,  MPI_COMM_WORLD,  irequest, ierr )
    enddo
else
       call  MPI_RECV( a,  N,  MPI_DOUBLE_PRECISION ,
          0,  i_loop, MPI_COMM_WORLD,  istatus,  ierr )
endif
  …
if (myid .eq.  0) then
     do  i=1,  numprocs - 1
        call  MPI_WAIT(irequest(i),  istatus, ierr )
     enddo
endif
```

ランク0のPEは、
ランク1~191までのPEに対して、
ノンブロッキング通信を用いて、
長さNのDOUBLE PRECISION
型配列データを送信

ランク1~191までのPEは、
ランク0からの受信待ち。

ランク0のPEは、
ランク1~191までのPEに対する
それぞれの送信に対し、
それぞれが受信完了するまで
ビジーウェイト（スピンウェイト）
する。

# レポート課題（その１）

- 問題レベルを以下に設定

問題のレベルに関する記述:
- L00: きわめて簡単な問題。
- L10: ちょっと考えればわかる問題。
- L20: 標準的な問題。
- L30: 数時間程度必要とする問題。
- L40: 数週間程度必要とする問題。複雑な実装を必要とする。
- L50: 数か月程度必要とする問題。未解決問題を含む。

※L40以上は、論文を出版するに値する問題。

- 教科書のサンプルプログラムは以下が利用可能
  - 付属のサンプルプログラム全てが利用可能

# レポート課題（その２）

1. [L5] ブロッキングは同期でないことを説明せよ。
2. [L10] MPIにおけるブロッキング、ノンブロッキング、および通信モードによる分類に対応する関数を調べ、一覧表にまとめよ。
3. [L15] 利用できる並列計算機環境で、ノンブロッキング送信（MPI_Isend関数）がブロッキング送信（MPI_Send関数）に対して有効となるメッセージの範囲（N=0～適当な上限）について調べ、結果を考察せよ。
4. [L20] MPI_Allreduce関数 の＜限定機能＞版を、ブロッキング送信、およびノンブロッキング送信を用いて実装せよ。さらに、その性能を比べてみよ。なお、＜限定機能＞は独自に設定してよい。

# レポート課題（その３）

5. [L15] MPI_Reduce関数を実現するRecursive Halving アルゴリズムについて、その性能を調査せよ。この際、従来手法も調べて、その手法との比較も行うこと。
6. [L35] Recursive Halvingアルゴリズムを、ブロッキング送信／受信、および、ノンブロッキング送信／受信を用いて実装せよ。また、それらの性能を評価せよ。
7. [L15] 身近の並列計算機環境で、永続的通信関数の性能を調べよ。
8. [L10～] 自分が持っているプログラムに対し、ループ分割、ループ融合、その他のチューニングを試みよ。
9. [L10～] 自分が持っているMPIプログラムに対し、ノンブロッキング通信（MPI_Isend, MPI_Irecv）を実装し、性能を評価せよ。また永続的通信が使えるプログラムの場合は実装して評価せよ。

# レポート課題（その4）

10. [L10] MPI 3.0における拡張仕様（特に、ノン・ブロッキングの集団通信関数のAPI）を調べてまとめよ