MPE と MPI の応用

赤塚　浩太

1　概要

本資料では，MPI(Message Passing Interface) の拡張として開発された MPE について解説する．MPE は，並列プログラムのログファイル作成ライブラリであり，MPICH や LAM など MPI に対応している．具体的には，MPI の通信関数である MPI_Send や MPI_Recv，MPI_BCAST などがどのように実行されたかのタイムラインによるログや，ユーザーが指定した関数にどの程度時間がかかったかなどが把握できる (Fig. 1 に出力例を示す).

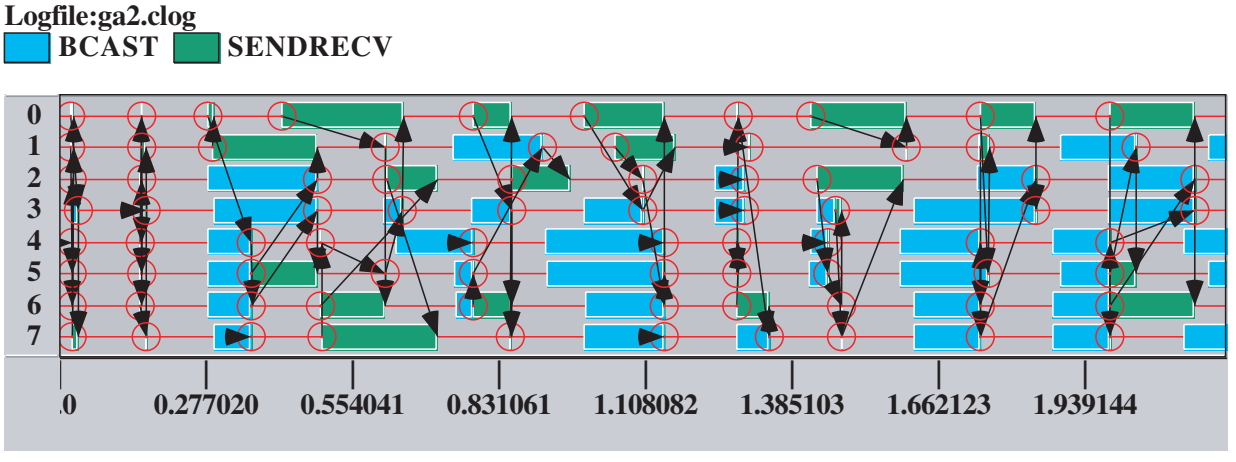

Fig. 1　MPE 実行結果例

この出力結果では横軸が時間，縦軸がプロセッサ番号となっている．薄い灰色が BCAST を行っている時間帯，濃い灰色が SENDRECV を行っている時間帯で，矢印が通信の方向である．

2　準備

MPE を使用するためには，MPE のライブラリが実行環境にインストールされている必要がある．詳しくは各システムの管理者に聞けばわかるが，通常通り mpich をインストールすると自動的に MPE もインストールされる．ただし，lam と共に用いたい場合には管理者の設定が必要である (詳細は [1] を参照).

また，mpe の出力結果を CUI で出力することも可能であるが非常に理解しづらいので，実質 GUI で表示する必要がある．従って，並列プログラムの実行環境 (クラスタ側) に x クライアントが，自分が使っているマシンに x サーバーが入っている必要がある．自分の使っているマシンに x サーバーが入っている場合，クラスタ側の環境変数「DISPLAY」を設定することで，mpe の出力結果を自分のマシンで見ることができる．具体的にはクラスタ側の.bash_profile に

```
export DISPLAY=192.168.6.*:0.0
```

を追加するか，ターミナルから同様のコマンドを打てばよい（192.168.6.*は自分のアドレス）.

3　MPI コマンドのログ

MPI の通信関数にどの程度時間がかかっているかや，実際にどのように通信が行われているかを把握には，コンパイル時に「-llmpe -lmpe -lpmpich」の 3 つのオプションを付け加えるだけでよい．たとえば，ga.cpp をコンパイルするのであれば以下のようになる.

```
$ mpiCC -llmpe -lmpe -lpmpich -o ga ga.cpp
```

これだけで，ga.cpp がどの関数を用いていつどのように通信しているかを知ることができる．続いて実行であるが，実行は通常通り mpirun などを用いて行えばよい.

```
$ mpirun -np 4 ga
```

実行後，自動的に実行ファイル名.clog(今の場合 ga.clog) というファイルが生成される．このファイルが MPE が出力するログファイルである．

これを閲覧するツールが「logviewer」で，コンソール上で

```
$ logviewer ga.clog
```

と打てばよい．

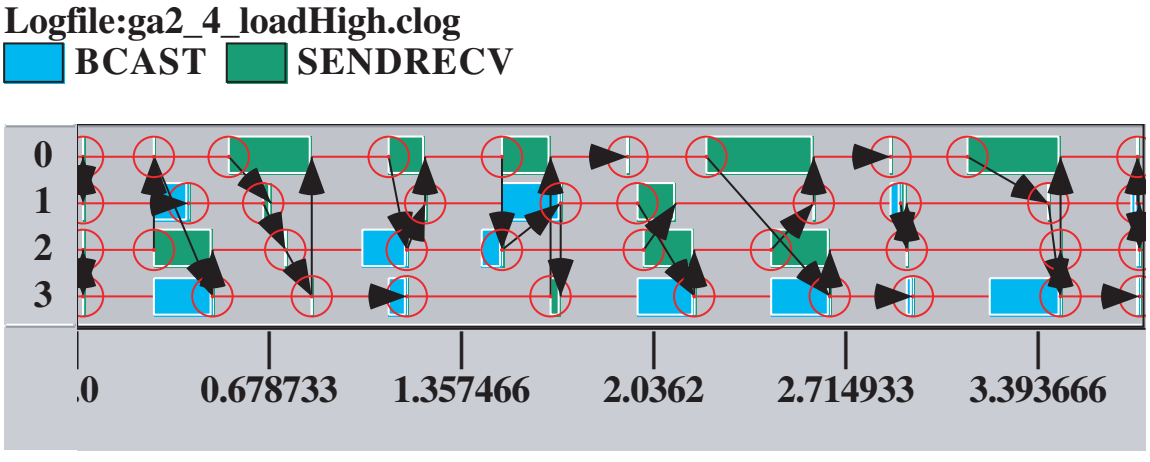

Fig. 2　ga の結果

Fig. 2 は，片浦君の協力（というか片浦君の GA）で得られた GA の通信記録である．このプログラムでは他プロセスとの通信を一定間隔おきにランダムなプロセスと行うので，各通信毎に通信相手が異なっている．これにより，通信が意図通り正しく行われていることを確認できる．また，このプログラムでは他プロセスからの受信時（MPI_SendRecv の recv 時）にデータが届くまでの待ち時間が発生しているのがよくわかる．

なお，この状態は実行環境の負荷が非常に高い状態で行ったため，通信による遅延が顕著であるが，負荷が低い場合には Fig. 3 のようにほとんど遅延が無いことがわかる．

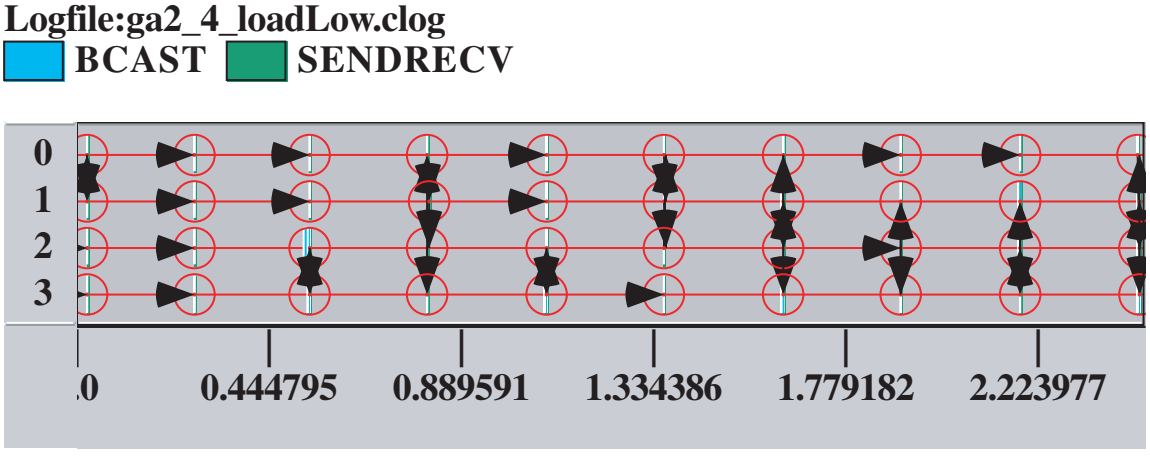

Fig. 3　負荷が少ない時の ga の結果

ここで，負荷が高い状態では各プロセスはデータ受信後すぐに処理を再開するため，一時的にではあるがプロセス毎に進捗状況が異なる (特にプロセス 0 は負荷が低いため他のプロセスより早い)．そこで (全く意味はないが) 各プロセスの同期をとる MPI_Barrier を通信後に行うように変更すると，Fig. 4 のように毎回同期を取るようになることがわかる．

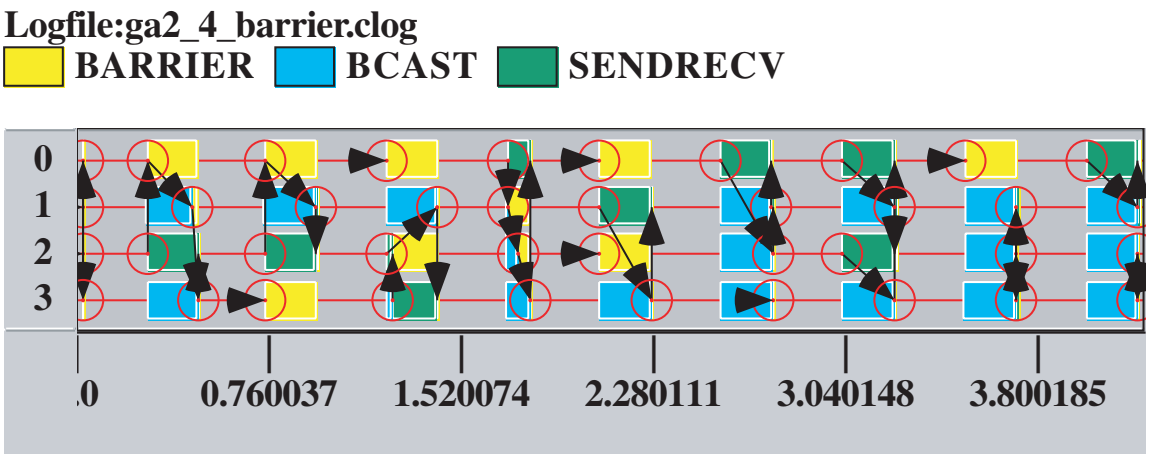

Fig. 4　通信時に完全に同期を取る場合の結果

4　自分のプログラムの各関数のログを取る

4.1　GA に適応した例

MPE では，MPI 関数のログを取るだけではなく，プログラムにいくつか MPE 関数の呼び出しを追加するだけで自分のプログラムのどの関数にどの程度時間がかかっているかを調べることができる．Fig. 5 は，片浦君のプログラムにおける各遺伝的操作の実行時間を出力に追加したものである．

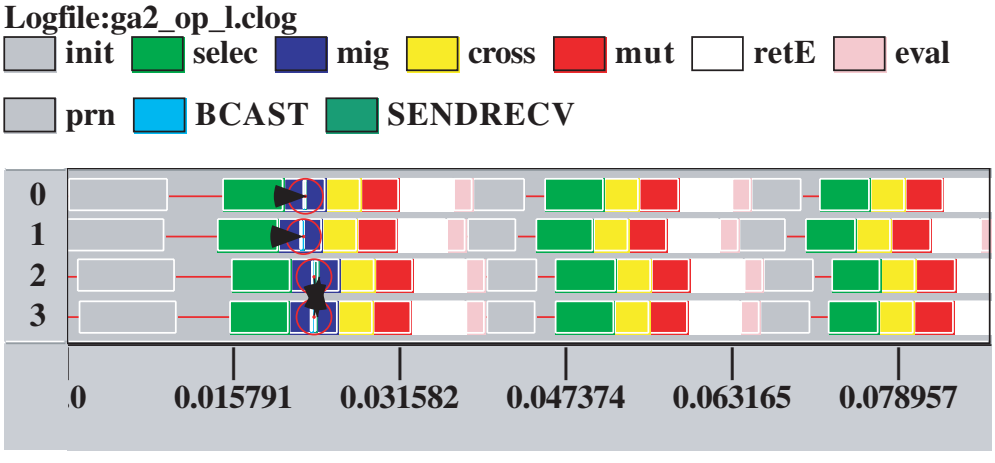

Fig. 5　GA 操作の時間も表示させた例

通信よりも選択やエリート保存，評価に時間がかかっていることがわかる．このように，自分のプログラムの実行時間を調べることで改善すべき箇所が容易にわかる．

4.2　プログラムの変更

実際にどのようにすれば Fig. 5 のような図ができるかを解説する．ここでは簡単のため，プログラム中の func(); の時間を計測したいとする．元のプログラムが

```
void main (int argc,char *argv[]){
 MPI_Init(&argc,&argv);
 func();
 MPI_Finalize();
}
```

だとすると，まず，プログラムのヘッダ部分で「mpe.h」をインクルードする必要がある．続いて，プログラム中でログに記録したい個所を，MPE_Log_event ではさむ．

```
#include "mpe.h"                          // <--------------------
void main (int argc,char *argv[]){
 MPI_Init(&argc,&argv);
  MPE_Log_event(ev1, 0,"func_start");    // <--------------------
 func();
  MPE_Log_event(ev2, 0,"func_end");      // <--------------------
 MPI_Finalize();
}
```

このとき，MPE_Log_event の引数の 2 番目には正数を，3 番目には適当な文字列を指定すればよい (いずれも logviewer で表示されるだけ)．また，1 番目の引数はイベントナンバーで，これはあらかじめ MPE_Log_get_event_number で取得しておく必要がある．

```
#include "mpe.h"
void main (int argc,char *argv[]){
 MPI_Init(&argc,&argv);
  int ev1,ev2;
  ev1=MPE_Log_get_event_number();  // <--------------------
  ev2=MPE_Log_get_event_number();  // <--------------------
  MPE_Log_event(ev1, 0,"func_start");
 func();
  MPE_Log_event(ev2, 0,"func_end");
 MPI_Finalize();
}
```

また，logviewer で表示する際の色や凡例を設定するには MPE_Describe_state を呼ぶ必要がある．そして，log の記録を開始するために MPE_Start_log を呼ぶ．

```
#include "mpe.h"
void main (int argc,char *argv[]){
 MPI_Init(&argc,&argv);
  int ev1,ev2;
  ev1=MPE_Log_get_event_number();
  ev2=MPE_Log_get_event_number();
  MPE_Describe_state(ev1,ev2,"func","red"); // <--------------------
  MPE_Start_Log();                          // <--------------------
  MPE_Log_event(ev1, 0,"func_start");
 func();
  MPE_Log_event(ev2, 0,"func_end");
 MPI_Finalize();
}
```

MPE_Describe_state の第一引数は定義するイベントの開始位置，第二引数が終了位置，第三引数が表示時の凡例，第四引数が表示時の色である．次に，片浦君の GA に MPE によるログイベントの指定を追加したプログラムの一部を掲載する (省略部分に MPE_Log_event 以外の MPE 関数呼び出しは無い)．このプログラムの実行結果が，Fig. 5 である．

```
(省略)
  for(i=0;i<16;i++)
       ev[i]=MPE_Log_get_event_number(); // イベントナンバーの取得

  MPE_Describe_state(ev[0],ev[1],"init","gray");    // 色と文字列の設定
  MPE_Describe_state(ev[2],ev[3],"selec","green");  // 色と文字列の設定
  MPE_Describe_state(ev[4],ev[5],"mig","blue");     // 色と文字列の設定
  MPE_Describe_state(ev[6],ev[7],"cross","yellow"); // 色と文字列の設定
  MPE_Describe_state(ev[8],ev[9],"mut","red");      // 色と文字列の設定
  MPE_Describe_state(ev[10],ev[11],"retE","white"); // 色と文字列の設定
  MPE_Describe_state(ev[12],ev[13],"eval","pink");  // 色と文字列の設定
  MPE_Describe_state(ev[14],ev[15],"prn","gray");   // 色と文字列の設定
  MPE_Start_log();                                  //ログ開始
  MPE_Log_event(ev[0],0,"st_init");                 //init 部ここから
  Individual *pop = NULL;
  initialize_pop(&pop, pop_size, glength);
  Individual *elite = NULL;
  initialize_pop(&elite, num_elite, glength);
  evaluate(pop, pop_size);
  MPE_Log_event(ev[1],0,"ed_init");                 //init 部ここまで
  for(g = 0; g < max_generation; g++){
    int i;
    input_elite(pop, pop_size, elite, num_elite);
    MPE_Log_event(ev[2],0,"st_sel");                //select 部ここから
    select(pop, pop_size);
    MPE_Log_event(ev[3],0,"ed_sel");                //select 部ここまで
    if(g % send_interbal == 0){
      MPE_Log_event(ev[4],0,"st_mig");              //Migrate 部ここから
      imigration(pop, pop_size, send_pop, glength, myid, island);
      MPE_Log_event(ev[5],0,"ed_mig");              //Migrate 部ここまで
    }
    MPE_Log_event(ev[6],0,"st_cro");                //Crossover 部ここから
    crossover(pop, pop_size, crossrate);
    MPE_Log_event(ev[7],0,"ed_cro");                //Crossover 部ここまで
    (省略)
  }
(省略)
```

Fig. 6 は GA の個体数を 5 倍にした場合の結果である．通信時間以外に，選択 (selection) 部分やエリート保存 (retE) 部分の割合が大きくなっていることがわかる．

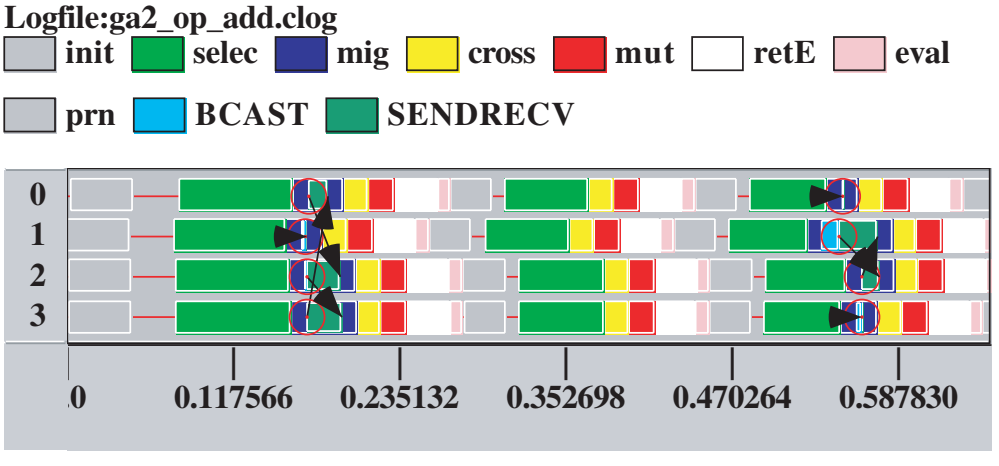

Fig. 6　GA の個体数を増加させた例

5　MPI について

5.1　MPI 各関数の速度

MPI の各関数の通信速度に関して簡単な実験を行った．実験は，long*1000 の配列をプロセス 0 から 0 以外のプロセスすべてに送り，続いて 0 以外のプロセスすべてからプロセス 0 に送り返す操作を 100 回繰り返すという簡単な方法で行った．その際に，MPI_Send と Recv の組み合わせ，MPI_Sendrecv，MPI_Isend と Irecv の組み合わせ，MPI_Gather と Scatter の組み合わせ，MPI_Bcast と Gather の組み合わせ，の計 5 種類の方法を用いた．

まず，この実験を 3 プロセスで行った場合の MPE による結果を Fig. 7 に示す．

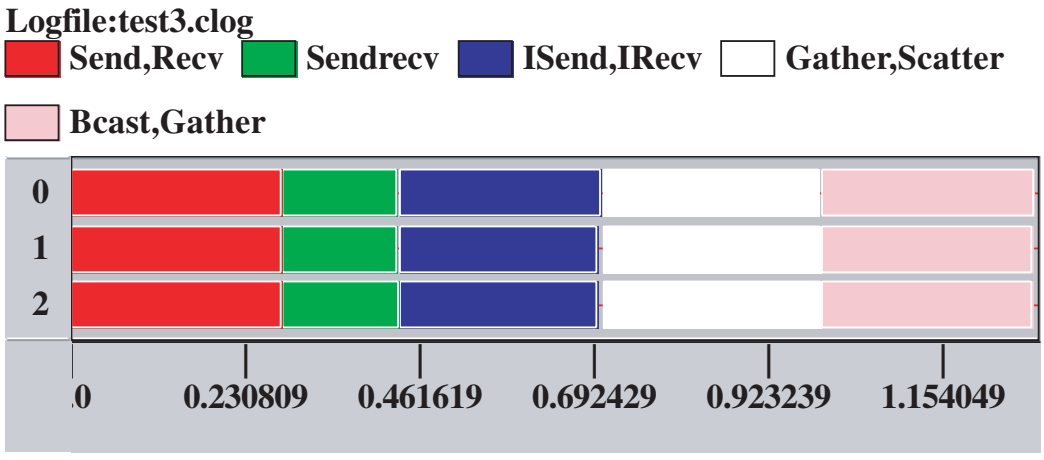

Fig. 7　3 プロセスの結果

結果は，Sendrecv で行う場合が最も速く 0.15sec で，その後 Isend と Irecv の 0.26sec が続く．後の 3 関数はほぼ同じ 0.28sec である．続いて，この実験を 9 プロセスで行った場合の MPE による結果を Fig. 8 に示す．

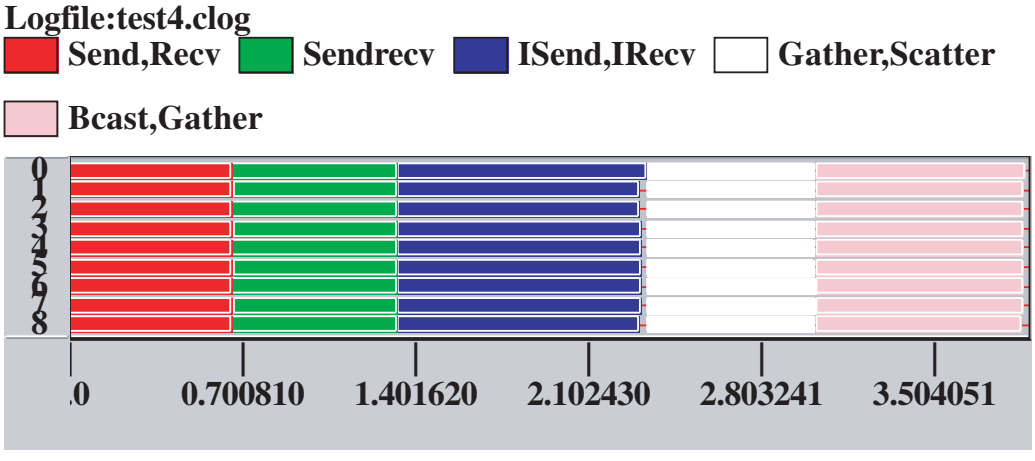

Fig. 8　9 プロセスの結果

この場合，最も速いのは Send,Recv と Sendrecv で 0.65sec，続いて Gather,Scatter が 0.68sec で，これら 3 つはほぼ同じといえる．遅いのは Bcast を用いる場合の 0.84sec と Isend を用いる場合の 1.00sec である．もちろん，今回は通信以外の処理を一切行っていないため，非同期の利点が生かされていない．

これらの結果から，単純なデータを繰り返し転送するのであれば Sendrecv がもっともよいと言うことになる．特に 9 プロセス程度の場合，Bcast ではなく Sendrecv でも十分な速度が得られることがわかる．ただし，参考文献 [2] によれば，Bcast はプロセス数が 32 程度になると高速になるようである．

5.2　並列モデル

逐次プログラムの並列化を考える場合，単に逐次プログラムの繰り返し部分を並列にすればよいというわけではない．たとえば，プログラムの先頭でファイルを読み込む場合，プロセス 0 がファイルを読み込み，それを他プロセスに配る方法では，通信負荷が高くなり実行速度が低下する．この場合，すべてのプロセスでファイルを読み込む方が効率的である．また，GA の並列化などで乱数を用いて通信の相手を決定する場合などで，プロセス 0 がすべてを決定し，その結果を配信する形をよく見かけるが，これも乱数の種をそろえるなどしてすべてのプロセスで通信相手を決定する方が効率的である．このように逐次プログラムでは無駄であった事が，並列プログラムでは効率的となる場合があるので注意が必要である．

5.3　Non-Blocking 通信

並列モデルについて，逐次モデルの単純な並列化では効率が悪い事があることを述べた．一方で，Non-Blocking 通信 (Isend, Irecv) は通信の完了を待たずに他の作業ができるため，効率よく使えば非常に有効である．もっとも簡単に Non-Blocking 通信の恩恵にあずかるには，逐次モデルを一部変更した近似モデルを用いて，非同期化すればよい．GA の並列化に際して GAPPA や PooledGA で用いられたのはこの方法である．しかし，DGA のアルゴリズムをそのままに (非同期にせず)，通信だけ Non-Blocking にすることでも若干の効率化を図ることが可能である．

たとえば，GA であれば移住個体を通信で送受信した後に個体をよい順に並べ替えることがある．この場合，通常は個体の受信を完了した後にソートするが，Non-Blocking 通信とマージソートを効果的に使えば，効率化が図れる．

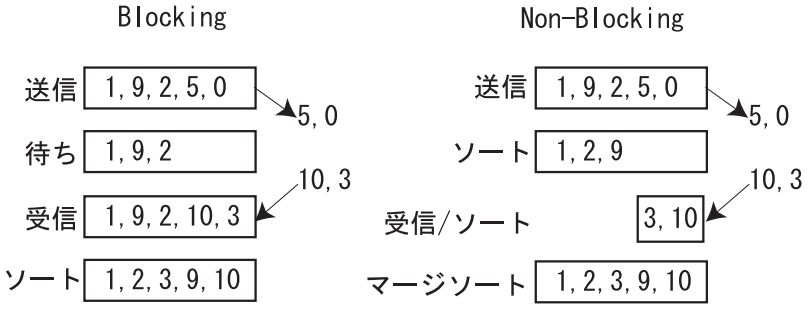

Fig. 9　Non-Blocking 通信とマージソート

Fig. 9 の左側のように，通常は通信時間は待ち時間となり，プロセスはアイドル状態となる．しかし，Non-Blocking 通信では引き続き計算ができるのでこの間に残りの個体のソートを行う．そして，受信した個体のソートを受信後に行った後，残りの個体とマージソートを行う．マージソートは通常のソートに比べ非常に高速である．ソートは要素数が多くなるほど非常に時間がかかるため，個体数の多い残り個体を通信時間を利用して先にソートしておくのは有効である．

また，GA において移住してきた個体が残り個体中に大きく影響するのは，次の世代の選択時である．そこで，移住を行う前に移住個体と交叉する相手個体を残しておき，移住後，残しておいた個体以外の交叉，突然変異，評価を先に行う．その後移住してきた個体と残しておいた個体で交叉を行い，突然変異，評価を行う (Fig. 10)．

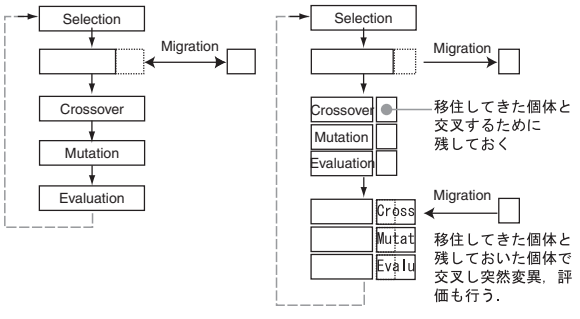

Fig. 10　Non-Blocking 通信向け GA

GA において多くの時間を費やすのは評価であるため，大半の個体の評価を通信中に行うため，このモデルでは通信時間をほとんど気にする必要が無くなる．

このように，Non-Blocking 通信を行い，アルゴリズムのブラッシュアップを行えば，通信の効率化をはかることができる．

謝辞

このゼミ資料を作成するに当たって，快くプログラムを提供してくださった片浦君，下坂君に感謝します．

参考文献

1) 佐野正樹: MPE の使い方，知的システムデザイン研究室 管理者委員会, (2001.1)

2) P. パチェコ: MPI 並列プログラミング，培風館, (2001)