並列プログラミング入門　MPI版
2012年6月1日版

理化学研究所　情報基盤センター
青山幸也

はじめに

昔は並列計算機と言えば、特別な人たちが特殊な用途で使用する珍しいマシンであるという印象がありました。ところがここ数年、並列計算機の導入が急激に増加してきています。並列計算機の主要な用途として、科学技術計算とデータベース検索がありますが、このうち科学技術計算ではプログラムを並列化する必要があります。並列コンピュータに関する数多くの本が出版されていますが、並列コンピュータを作成する人のための並列アーキテクチャーの本、並列コンパイラを作成する人のための本、情報系の大学院レベルを対象とした『理論』としての並列化の本などが多く、プログラムを並列化する方法について説明した本はあまり見当たりません。

並列計算機に関するこのような現状から、並列化を行う現場で役立つ分かり易いテキストを目標として本書を作成しました。しかし、並列プログラミングの技術はまだ発展途上であり、今後も多くの技法が出てくると思いますので、本書もそれに合わせて内容を加筆修正していきたいと思います。

なお、本書の内容に関する責任は負いかねますのでご了承願います。

1995 年　青山　幸也

本書は分散メモリー型並列計算機上でプログラムを並列化する方法についてのガイドで、初版を作成してから早いもので約 8 年経ちました。初版は筆者がコンピュータ会社に勤務していたときにユーザー教育用として作成しました。私事で恐縮ですが、2002 年秋に転職したのを機に、書名を改め、特定のマシンに固有の部分を削除し、その他の部分も加筆修正する事にしました。このような経緯のため、内容に一部整合性がとれていない部分があるかと思いますが、今後少しずつ改善していきたいと思います。

2003 年　青山　幸也

当資料は、日本アイ・ビー・エム株式会社の承諾を得て、「(虎の巻シリーズ 2) 並列プログラミング虎の巻 MPI 版」を利用し、理化学研究所の責任で作成したものです。

本書は不定期に加筆修正しており、最新版は `http://accc.riken.jp/HPC/training.html` にあります。

目 次

第 1 章　並列プログラミングとは **1**
1-1　分散メモリー型並列計算機とは . . . 2
1-2　並列プログラミング・モデル . . . 4

第 2 章　並列ジョブの実行方法 **9**
2-1　並列ジョブの実行方法 . . . 10

第 3 章　メッセージ交換サブルーチン **13**
3-1　メッセージ交換ライブラリー . . . 14
3-2　環境管理 . . . 16
3-3　集団通信 . . . 22
3-3-1　集団通信サブルーチンの一般的な規則 . . . 22
3-3-2　集団通信サブルーチンと同期 . . . 23
3-3-3　集団通信サブルーチンの種類 . . . 24
3-3-4　MPI_BCAST . . . 25
3-3-5　MPI_GATHER . . . 29
3-3-6　MPI_REDUCE . . . 31
3-3-6-1　MPI_MAXLOC/MPI_MINLOC . . . 34
3-3-6-2　ユーザーが定義する演算 . . . 35
3-4　1 対 1 通信 . . . 40
3-4-1　1 対 1 通信の全体の流れ . . . 40
3-4-2　ブロッキング通信と非ブロッキング通信 . . . 41
3-4-3　1 方向のみの通信 . . . 45
3-4-4　双方向通信 . . . 46
3-4-5　1 対 1 通信に関する補足 . . . 50
3-5　派生データ型 . . . 51
3-5-1　派生データ型とは . . . 51
3-5-2　派生データ型に関する主な MPI のサブルーチン . . . 53
3-5-3　部分配列を表す派生データ型の作成 . . . 56
3-5-3-1　MPI_TYPE_CREATE_SUBARRAY . . . 57
3-5-3-2　ユーティリティー・サブルーチン . . . 58
3-5-4　構造体を表す派生データ型の作成 . . . 64
3-6　新グループの作成 . . . 65
3-7　C 言語の MPI 関数 . . . 66
3-8　MPI-2 . . . 68
3-8-1　MPI_IN_PLACE . . . 68

第 4 章　プログラムの並列化の方法 **69**
4-1　パフォーマンスに関する考慮点 . . . 70
4-1-1　アムダールの法則 . . . 70
4-1-2　パフォーマンスを向上させるための考慮点 . . . 71
4-2　どんなところが並列化できるのか . . . 75
4-3　計算部分の並列化パターン . . . 78
4-4　並列化に伴う入出力部分の修正 . . . 84
4-4-1　並列化に伴う入力部分の修正 . . . 84
4-4-2　並列化に伴う出力部分の修正 . . . 88
4-4-3　入出力に関するその他の考慮点 . . . 91
4-5　ループの分割方法 . . . 92
4-5-1　ループ反復と配列 . . . 92

4-5-2 配列の分割方法 . . . 93
4-5-3 ブロック分割でロードバランスが不均等になる例 . . . 95
4-5-4 ブロック分割 . . . 98
4-5-5 サイクリック分割 . . . 101
4-5-6 ブロック・サイクリック分割 . . . 103
4-5-7 配列の縮小 . . . 105
4-5-7-1 配列の縮小方法 . . . 106
4-5-7-2 配列の縮小に伴う変更 (I/O 以外の部分) . . . 112
4-5-7-3 配列の縮小に伴う変更 (I/O 部分) . . . 114
4-5-8 多重ループの分割 . . . 117
4-6 並列化とメッセージ交換 . . . 121
4-6-1 メッセージ交換はどんなときに必要か . . . 121
4-6-2 1 次元差分法 . . . 122
4-6-3 集団通信サブルーチンの代替 . . . 127
4-6-3-1 MPI_GATHERV の代替 . . . 127
4-6-3-2 MPI_ALLGATHERV の代替 . . . 130
4-6-3-3 MPI_ALLTOALLV の代替 . . . 133
4-6-4 合計（内積）／最大（最小） . . . 136
4-6-5 特定の配列要素の参照 . . . 138
4-6-6 重ね合わせ . . . 139
4-6-6-1 データの収集 . . . 139
4-6-6-2 間接アドレスを使用した計算 . . . 141
4-6-7 パイプライン法 (ICCG 法の前進消去／後退代入などで使用) . . . 143
4-6-8 ツイスト分割法 (ADI 法などで使用) . . . 147
4-6-9 一次逐次演算（漸化式などで使用） . . . 152
4-6-10 受信するデータ量が不明な場合の通信 . . . 155
4-7 特殊な並列化方法 . . . 157
4-7-1 MPMD モデルでの並列化 . . . 157
4-7-1-1 実行方法 . . . 157
4-7-1-2 連成解析 . . . 158
4-7-1-3 マスター・スレーブ方式 . . . 159
4-7-2 フラット MPI とハイブリッド並列 . . . 162
4-8 並列化の手順と考慮点 . . . 164
4-8-1 並列化の手順 . . . 164
4-8-2 エラーかなとおもったら . . . 171
4-8-3 パフォーマンスの測定と評価 . . . 174

第 5 章 並列化の応用例 179
5-1 差分法徹底攻略 . . . 180
5-1-1 2 次元目でブロック分割した場合 . . . 181
5-1-2 1 次元目でブロック分割した場合 . . . 182
5-1-3 1，2 次元目でブロック分割した場合 . . . 183
5-1-4 1，2 次元目でブロック分割した場合（斜めの要素を参照） . . . 186
5-1-5 配列を縮小する場合 . . . 189
5-1-6 その他の考慮点 . . . 190
5-2 有限要素法 . . . 191
5-2-1 有限要素法の並列化 . . . 191
5-2-2 並列化の方法 1 . . . 193
5-2-3 並列化の方法 2 . . . 197
5-3 LU 分解 . . . 205
5-4 ICCG 法 . . . 211
5-4-1 パイプライン法による並列化 . . . 211
5-4-2 パラレル・ブロック・オーダリング法による並列化 . . . 224
5-5 マルチフロンタル法 . . . 236
5-6 SOR 法 . . . 237
5-6-1 SOR 法と並列性 . . . 237

5-6-2　プログラム例（ゼブラ法） . . . 239
5-6-3　その他の解法 . . . 242
5-6-4　各解法の具体的な並列化方法 . . . 245
5-7　モンテカルロ法 . . . 251
5-8　個別要素法／分子動力学法 . . . 253

第 6 章　並列版数値計算ライブラリー　257
6-1　並列版数値計算ライブラリー . . . 258

第 7 章　本当に並列化の必要があるか？　259
7-1　単体チューニング . . . 260
7-2　並列処理と分散処理 . . . 261
7-3　並列化とワークロード . . . 263

付 録　MPI サブルーチン一覧　265
環境管理サブルーチン　MPI_INIT . . . 266
環境管理サブルーチン　MPI_FINALIZE . . . 267
コミュニケータに関するサブルーチン　MPI_COMM_SIZE . . . 268
コミュニケータに関するサブルーチン　MPI_COMM_RANK . . . 269
環境管理サブルーチン　MPI_ABORT . . . 270
環境管理ファンクション　MPI_WTIME . . . 271
集団通信サブルーチン　MPI_BCAST . . . 272
集団通信サブルーチン　MPI_SCATTER . . . 274
集団通信サブルーチン　MPI_SCATTERV . . . 276
集団通信サブルーチン　MPI_GATHER . . . 278
集団通信サブルーチン　MPI_GATHERV . . . 280
集団通信サブルーチン　MPI_ALLGATHER . . . 282
集団通信サブルーチン　MPI_ALLGATHERV . . . 284
集団通信サブルーチン　MPI_ALLTOALL . . . 286
集団通信サブルーチン　MPI_ALLTOALLV . . . 288
【MPI-2】集団通信サブルーチン　MPI_ALLTOALLW . . . 290
集団通信サブルーチン　MPI_REDUCE . . . 292
集団通信サブルーチン　MPI_ALLREDUCE . . . 294
集団通信サブルーチン　MPI_SCAN . . . 296
【MPI-2】集団通信サブルーチン　MPI_EXSCAN . . . 298
集団通信サブルーチン　MPI_REDUCE_SCATTER . . . 300
集団通信サブルーチン　MPI_OP_CREATE . . . 302
集団通信サブルーチン　MPI_BARRIER . . . 304
【MPI-2】MPI_IN_PLACE の使い方　MPI_SCATTER . . . 306
【MPI-2】MPI_IN_PLACE の使い方　MPI_SCATTERV . . . 307
【MPI-2】MPI_IN_PLACE の使い方　MPI_GATHER . . . 308
【MPI-2】MPI_IN_PLACE の使い方　MPI_GATHERV . . . 309
【MPI-2】MPI_IN_PLACE の使い方　MPI_ALLGATHER . . . 310
【MPI-2】MPI_IN_PLACE の使い方　MPI_ALLGATHERV . . . 311
【MPI-2】MPI_IN_PLACE の使い方　MPI_REDUCE . . . 312
【MPI-2】MPI_IN_PLACE の使い方　MPI_ALLREDUCE . . . 313
【MPI-2】MPI_IN_PLACE の使い方　MPI_SCAN . . . 314
【MPI-2】MPI_IN_PLACE の使い方　MPI_REDUCE_SCATTER . . . 315
1 対 1 ブロッキング通信サブルーチン　MPI_SEND . . . 316
1 対 1 ブロッキング通信サブルーチン　MPI_RECV . . . 318
1 対 1 ブロッキング通信サブルーチン　MPI_SENDRECV . . . 320
1 対 1 非ブロッキング通信サブルーチン　MPI_ISEND . . . 322
1 対 1 非ブロッキング通信サブルーチン　MPI_IRECV . . . 324
1 対 1 非ブロッキング通信サブルーチン　MPI_WAIT . . . 326
1 対 1 通信サブルーチン　MPI_GET_COUNT . . . 328
派生データ型に関するサブルーチン　MPI_TYPE_SIZE . . . 330

派生データ型に関するサブルーチン　MPI_TYPE_CONTIGUOUS 331
派生データ型に関するサブルーチン　MPI_TYPE_VECTOR/HVECTOR 332
【MPI-2】派生データ型に関するサブルーチン　MPI_TYPE_CREATE_INDEXED_BLOCK 334
派生データ型に関するサブルーチン　MPI_TYPE_INDEXED/HINDEXED 336
派生データ型に関するサブルーチン　MPI_TYPE_STRUCT 338
【MPI-2】派生データ型に関するサブルーチン　MPI_TYPE_CREATE_SUBARRAY 340
派生データ型に関するサブルーチン　MPI_TYPE_COMMIT 342
【MPI-2】派生データ型に関するサブルーチン　MPI_TYPE_CREATE_RESIZED 343
コミュニケータに関するサブルーチン　MPI_COMM_SPLIT 344

付 録　参考文献 **347**

第1章　並列プログラミングとは

本章では、分散メモリー型並列コンピュータの使用形態と、並列プログラミングの基本的な動作について説明します。

注意

- 本書の内容はときどき追加、更新しております。最新版は下記からダウンロード可能です。
  http://accc.riken.jp/HPC/training/text.html
- 本書内の「〜はマシン環境に依存します」という記述は、その動作がマシンのハードウェア、OS、コンパイラなどの種類によって異なる可能性がある事を意味します。
- 本書のプログラム例は Fortran で書かれています。C 言語を使用していて Fortran をご存じない方は以下の点に注意して下さい。
  - 本書のプログラム例の中で特に宣言をしていない変数のデータ型は、先頭文字が I, J, K, L, M, N で始まっている変数は整数、それ以外の変数は実数です。
  - 実数が単精度 (4 バイト) か倍精度 (8 バイト) かは、プログラム内に「IMPLICIT REAL*8(A-H,O-Z)」が指定されているかどうかによって以下のように決まります。

IMPLICIT REAL*8(A-H,O-Z)	なし	あり
REAL A	単精度	単精度
REAL*8 B	倍精度	倍精度
DIMENSION C(10)	単精度	倍精度
何も宣言されていない変数 D	単精度	倍精度

  - 本書のプログラムは原則として大文字で書かれていますが、Fortran では大文字と小文字の区別がないので、どちらで書いても（あるいは混在しても）構いません。例えば変数 AA, Aa, aA, aa は同一です。
  - Fortran では配列は 1 から始まります。
    【Fortran】REAL A(2): A(1), A(2)
    【C 言語】float a[2]: a[0], a[1]
  - 多次元配列がメモリー上に配置される順番は、Fortran では左の添字が先に動く順番になります。
    【Fortran】REAL A(2,2): A(1,1), A(2,1), A(1,2), A(2,2)
    【C 言語】float a[2][2]: a[0][0], a[0][1], a[1][0], a[1][1]

1-1　分散メモリー型並列計算機とは

並列計算機の種類

(1)　単体計算機

単一 CPU の計算機（以後単体計算機）は、図 1-1-1（上段）に示すように、1 つの CPU とメモリーから構成され、UNIX や Linux などのオペレーティングシステム (以下 OS) が 1 つ存在してシステムを管理します。実行を開始した単体プログラムは単体プロセスと呼ばれ、メモリーにロードされます。図 1-1-1（下段）に示すように、単体プログラムは、並列化して実行することが不可能な部分①，② (例えば I/O) と、可能な部分1, 2 (例えば DO ループの各反復で計算が独立している場合の 1 反復目と 2 反復目) で構成されています。

(2)　分散メモリー型並列計算機

並列計算機には、分散メモリー型と共有メモリー型があります。分散メモリー型は、図 1-1-2（上段）に示すように、複数の単体計算機を、計算機間でデータの通信を行うためのネットワークで結合した計算機で、各 CPU にそれぞれメモリーと OS が存在します。各 CPU はお互いに完全に独立しています。

分散メモリー用に並列化されたプログラムを 2 CPU で実行した場合の動作を図 1-1-2（下段）に示します。

- ジョブが開始すると、並列化されていない①と②の部分が（基本的に）全 CPU で実行されます。
- 並列化されている1をプロセス 0 が、2をプロセス 1 が実行します。従ってこの部分の計算時間は理想的には半分になります。並列化された部分では、必要ならば、↔ に示すようにデータの通信を行います。
  このように、並列ジョブの開始から終了までの間、プロセスの数はずっと 2 つです。つまり、並列化されている部分もされていない部分も、2 つの CPU で並列に動作していることに注意して下さい。言いかえると、通信が可能な単体ジョブが、たまたま 2 つの CPU で同時に実行を開始したと考えることもできます。

(3)　共有メモリー型並列計算機

共有メモリー型では、図 1-1-3（上段）に示すように、複数の CPU が 1 つのメモリーを共有します。また OS は 1 つだけ存在し、システム全体を管理します。なお、共有メモリー型並列計算機のことを SMP (Symmetric Multi-Processing) と呼ぶこともあります。

共有メモリー用に並列化されたプログラムを 2 CPU で実行した場合の動作を図 1-1-3（下段）に示します。

- ジョブが開始すると、まず 1 つの並列プロセスが動作し、並列化されていない①を実行します。
- 並列化された部分に到達すると、並列プロセスはスレッドを 1 つ生成し、自分自身もスレッドとなります。スレッド 0 は1を、スレッド 1 は2を処理します。この部分の計算時間は理想的には半分になります。なお、スレッドというのは UNIX の用語ですが、本書のレベルではプロセスと同じような物だと考えて下さい。
- 1と2の処理が終了すると、スレッド 0 は並列化されていない②を処理し、スレッド 1 はスリープします。このように共有メモリー型マシンでは、1 つの並列ジョブの途中でスレッドの数が動的に変化します。

なお、分散メモリー型、共有メモリー型の並列計算機で、単体プログラムを実行する事も勿論可能です。

図 1-1-1　単体計算機　　図 1-1-2　分散メモリー型並列計算機　　図 1-1-3　共有メモリー型並列計算機

分散メモリー型並列計算機

以後、分散メモリー型並列計算機での並列化方法について説明します。分散メモリー型の並列計算機は、図 1-1-4 (1) に示すように、2 台の単体の計算機をプログラム間でデータのやりとりができるようにネットワークで結合したのが最初の形態です。その後並列計算機が普及するにつれ、図 1-1-4 (2) に示すように、各 CPU が並列計算機用の専用ラックに入ったタイプが出現しました。

一方図 1-1-4 (3) に示す共有メモリー型並列計算機では、通常は共有メモリー型の並列プログラムを実行しますが、マシン環境によっては（本書で説明する）分散メモリー型の並列プログラムを実行することもできます (4-7-2 節参照)。

最近は、図 1-1-4 (4) に示すように、複数の共有メモリー型並列計算機をネットワークで結合した分散メモリー型並列計算機が増えてきています。

並列計算機では、各 CPU をノード、あるいは PE (Processing Element) と呼ぶこともあります（本書では主に「ノード」を使用します)。並列プログラムでは、ノード間でデータを通信する速度が遅いと並列化の効果が出ないので、通常高速なネットワークを使用します。

図 1-1-4 (1)

図 1-1-4 (3)

図 1-1-4 (2)

図 1-1-4 (4)

並列計算機の使用形態

並列計算機には、以下のような使用形態があります。

(1) 並列ジョブを実行する。
(2) 単体ジョブしか実行しない。すなわち単体計算機の集合体として使用する。
(3) 両者を混合させて使用する。

並列ジョブを実行する場合、既に並列化されている市販のパッケージを使用する場合と、自分でプログラムを並列化する場合があります。以下に並列化されている市販パッケージの例を示しますが、使用できるかどうかはお使いのマシンに依存します。

- 構造解析：ABAQUS，LS-DYNA3D，PAM-CRASH，MARC，NASTRAN，RADIOSS
- 流体：STAR-CD，FLUENT，STREAM，SCRYU
- 計算化学：DISCOVER，HONDO，GAMESS，GAUSSIAN

一方自分でプログラムを並列化する場合、大きく次の 3 つの目的があります。

- 経過時間を短縮するため
- 計算に大容量のメモリーが必要なので
- 並列アルゴリズムを研究するため

1-2　並列プログラミング・モデル

SPMD モデルと MPMD モデル

図 1-2-1 (1) では、単体プログラムをコンパイル・リンクしてロードモジュール a.out を作成し、3 つのノードで同時に実行を開始しています。各単体プロセスの中身は同じですが、当然ながらお互いに独立に動作します。また各プロセスは同じ変数 A を使用していますが、異なるプロセスなので全く関係はありません。

次に並列プログラムの説明をします。並列プログラミング・モデルとして、SPMD (Single Program Multiple Data) モデルと MPMD (Multiple Program Multiple Data) モデルの 2 種類があります。

SPMD モデルでは、図 1-2-1 (2) に示すように、1 つの並列プログラムから作成された同じロードモジュール a.out が各ノードにロードされ、並列プロセスとして同時に実行を開始します。各プロセスにはランクという識別子 (ID) (3 プロセスの場合、0, 1, 2) が付きます。各プロセスの中身は同じですが、お互いに独立に動作します。また各プロセスは同じ変数 A を使用していますが、異なるプロセスなので全く関係はありません。つまり、図 1-2-1 (1) と同じ状態であると考えると分かりやすいと思います。図 1-2-1 (1) と唯一異なるのは、必要であれば各プロセス間で通信を行ってデータを交換することができるという点です。なお、SPMD モデルでは各プロセスの中身は同じですが、全く同じ動作をするわけではなく、一般に異なる動作をします。

一方 MPMD モデルでは、図 1-2-1 (3) に示すように、構造解析、流体解析という全く異なる複数の並列プログラム（ロジック、変数名も全く関係がない）から作成された異なるロードモジュール a.out と b.out が各ノードにロードされ、並列プロセスとして同時に実行を開始します。これは前述のように、構造解析、流体解析という単体プログラムを、本例では 2 本ずつ計 4 本同時に実行したのと同じ状態であると考えて下さい。唯一異なるのは、必要であれば各プロセス間で通信を行ってデータを交換することができるという点です。本例のような処理を連成解析と呼び、原子力の解析や、地球規模の解析 (大気 + 海洋) などで用いられます。

図 1-2-1 (4) はマスター・スレーブ（またはマスター・ワーカー）と呼ばれる方式で、1 つのノードで親プロセス a.out を、その他全てのノードで同一の子プロセス b.out を実行します。親プロセス（係長）は子プロセス（平社員）に仕事を与え、子プロセスはその仕事が終了したら親プロセスに報告し、親プロセスは次の仕事を子プロセスに与えます。本例はマスターとスレーブを別プログラムで作成しているので、MPMD モデルです。

ほとんどのプログラムは SPMD モデルで並列化するので、以後は SPMD モデルでの並列化方法を説明し、MPMD モデルは 4-7-1 節で説明します (マシン環境によっては、MPMD モデルを実行できない場合もあります)。

図 1-2-1 (1)　単体プログラム

図 1-2-1 (2)　並列プログラム (SPMD モデル)

図 1-2-1 (3)　並列プログラム (MPMD モデル)

図 1-2-1 (4)　並列プログラム (MPMD モデル)

SPMD モデルの動作例 (1)

SPMD モデルでは、1 つの並列プログラムからロードモジュールを作成し、全ノードで同じ並列プロセスを実行しますが、各プロセスは全く同じ動作をするわけではなく、一般に異なる動作をします。つまり、1 つの並列プログラム内に、全プロセスの動作が記述されています。これを簡単な例で説明します。

(1)　単体での動作

一般のプログラムはマクロに見ると、『データの入力』、『処理』、『結果の書き出し』の 3 つの部分に分かれています。これを簡単化したプログラムを単体で実行した例を図 1-2-2 (1) に示します。

- 1で入力データを読み込み、これを大きさ 6 の配列に入れます。
- 2で、配列内で処理するデータの最初と最後の位置を IS と IE に設定します。
- 3で IS から IE までの各データ (つまり 6 つのデータ全て) に対して処理を行います。処理前と処理後のデータを、○△□と●▲■で表します。3は DO 文 (Fortran の場合)、for 文 (C 言語の場合) に相当します。
- 最後に4で、処理したデータをファイルに書き出します。

(2)　SPMD モデルで並列化した場合の動作

図 1-2-2 (1) のプログラムを SPMD モデルで並列化し、3 つのプロセスをノード 2，4，6 の 3 つのノードで実行する例を図 1-2-2 (2) に示します。

- 並列化したプログラムのロードモジュールが a.out であるとします。並列ジョブを開始すると、a.out が各ノードにロードされ、並列プロセスとして動き始めます。例えば 3 つのプロセスを実行した場合、前述のように、各プロセスにはランクと呼ばれる識別子 (ID) (本例では 0，1，2) が付けられます。このランクの値が各プロセスで異なるため、SPMD モデルで、同じ中身のプロセスに異なる動作をさせることができます。
- 図中でプログラムが 3 つ示してありますが、SPMD モデルなので中身は同一であることに注意して下さい。
  ①で全プロセスがそれぞれ同じ入力データを大きさ 6 の配列に読み込みます。分散メモリー型マシンなので、データを読み込む配列は各ノードのローカルメモリー上にあり、お互いに全く関係がないのは言うまでもありません。なお、入力データを読み込む方法として、1 つのプロセスのみが代表して読み込み、それを他のプロセスに送信する方法もあります。
- ②で、自分のランクを取得します。例えば自分がランク 0 のプロセスであれば 0 が戻ります。なお、①と②の順序は逆でもかまいません。
- ③で、処理するデータの最初と最後の位置を IS と IE に設定します。ランクによって異なる部分（枠で囲んだ部分）が実行されるため、異なる IS と IE の値が設定されます。
- ④で処理を行いますが、③で自分が担当する IS と IE の値が設定されているので、各プロセスはそれぞれ自分の担当部分のみを処理します。例えばランク 0 のプロセスは●の部分のみを処理します。
- ④が終了した後、各プロセスは自分が処理した部分のデータを⑤で（例えば）ランク 0 のプロセスに送信します。
- ⑥で、ランク 0 のプロセスが代表して結果の書き出しを行います。

実際のプログラムを SPMD モデルで並列化した場合も、マクロに見るとこの例のようになります。ただし、実際のプログラムでは通常④の部分でデータの通信が発生します。

図 1-2-2 (1)　単体での動作

図 1-2-2 (2)　SPMD モデルで並列化した場合の動作

並列化とメッセージ交換

(1)　並列化

3 章以降で、ノード間でデータの通信 (図 1-2-2 (2) の⑤) を行うメッセージ交換サブルーチンの使用方法について説明していきます。メッセージ交換サブルーチンの使用方法を学ぶにつれ、あたかも『メッセージ交換 = 並列化』であると考えるようになりがちですが、これは誤りです。このことを説明する前に、まず本当の意味の並列化について定義します。

図 1-2-2 (2) から分かるように、各プロセスのプログラムは最初から最後まで並列に動作します。しかし④以外の部分は、単体で動作した場合と計算量が変わらない（または並列化に伴って新規に追加された）部分なので、本当の意味の並列化とは言えません。本当の意味の並列化と言えるのは、計算量が 1/3 に減少した④の部分のみです。

そこで本書では、本当の意味の並列化を次のように定義します。

本当の意味の並列化とは、計算量を $1/n$ に縮小すること (ノード数が n 台の場合)。

(2)　メッセージ交換

本当の意味の並列化とメッセージ交換 (図 1-2-2 (2) の⑤) はどのような関係になるのでしょうか。例えば図 1-2-2 (2) では、本当の意味の並列化を行った④が終了した直後、各プロセスは自分が処理した 1/3 のデータのみを持っており、他プロセスが処理した部分のデータは、（自分とは別のプロセスなので）直接参照／更新することはできません。

このことによって、本当の意味の並列化を行った部分以後の処理に矛盾（副作用）をきたさないのであれば、メッセージ交換を行う必要はありません。しかしこの例では、図 1-2-2 (1) と同じ結果を得るために、各プロセスが処理したデータを、⑤で（例えば）ランク 0 のプロセスに集める必要があり、このときメッセージ交換を行います

まとめると、本当の意味の並列化とメッセージ交換の関係は以下のようになります。

本当の意味の並列化に伴う矛盾（副作用）を解消するために、必要最小限、仕方なしに行うのがメッセージ交換。

SPMD モデルの動作例 (2)

SPMD モデルによる並列化のイメージをつかむために、動作例をもう一つ示します。まず単体での動作を図 1-2-3 (1) に示します。図中の大きさ 9 の配列に 1〜9 の値が入っており、それを総合計した結果をディスプレイに表示します。

これを SPMD モデルで並列化した例を図 1-2-3 (2) に示します。

- 図 1-2-2 (2) と同様に、自分のランクから、自分の担当するデータの最初と最後の位置を決め、これを IS、IE とします。
- 各プロセスは自分が担当する部分の合計（小計）を求めます。この部分は計算量が 1/3 に減少したので、『本当の意味の並列化』になります。
- この時点で、各プロセスが持っている値は自分が計算した部分の小計です。これが並列化に伴う矛盾（副作用）です。もちろんこのままでよいのであればそれで終わりですが、この例では小計の合計、すなわち総合計を求めたいので、メッセージ交換を行います。これによって、並列化に伴う矛盾（副作用）を解消することが出来ます。
- この場合、図 1-2-2 (2) と同様に、各プロセスが求めた小計 (6，15，24) を一度ランク 0 のプロセスに集め、その後ランク 0 のプロセスが代表して $(6 + 15 + 24 = 45)$ を計算するという方法もあります。しかしこれでは面倒なので図 1-2-3 (2) では、『通信しながら演算を行う』タイプのメッセージ交換サブルーチンを使用しました。これを使用すると、小計 (6，15，24) に対して通信しながら演算が行われ、結果が例えばランク 0 のプロセスに入ります。
- 最後に、ランク 0 のプロセスが代表して総合計をディスプレイに表示します。

図 1-2-3 (1)　単体での動作

図 1-2-3 (2)　SPMD モデルで並列化した場合の動作

第2章　並列ジョブの実行方法

本章では、簡単なプログラムを用いて、並列ジョブのコンパイル・リンク・実行方法について説明します。

2-1　並列ジョブの実行方法

コンパイルから実行までの流れ

簡単なプログラムを用いて、並列ジョブのコンパイルから実行までの一連の流れを見てみましょう。なお、具体的な操作方法はマシン環境によって異なりますので、お使いのマシンのユーザーズガイド等を参照して下さい。

- 図 2-1-1 の①に示す簡単なプログラム hello.f を並列に実行してみます。なお①のプログラムは（後述する）MPI のサブルーチン MPI_INIT や MPI_FINALIZE などが指定されていないため、マシン環境によっては正常に動作しなかったり異常終了することがあります。
- ②で hello.f をコンパイル・リンクすると、③でロードモジュール a.out が作成されます。ここでは、a.out は各ノードからアクセスできる共有ディスクに作成されたとします。なお、コンパイル・リンクの方法はマシン環境によって異なります。
- ④で、並列ジョブ a.out を 2 プロセスで実行します。すると⑤で a.out が各 CPU にロードされ、ランク 0、ランク 1 のプロセスとして実行を開始します。なお、並列ジョブの実行方法および、どのプロセスがどの CPU にロードされるかは、マシン環境によって異なります。
- ⑥と⑦の PRINT 文が並列に実行され、⑧と⑨が画面に表示されます。⑧と⑨の表示順序は早い者順になります。なお、この例のように複数のプロセスが同時に標準出力 (WRITE(6) または PRINT 文) に書き出しを行った場合、出力行数が多かったり各行の文字数が多いと、各行が混ざって表示が乱れる事がありますので、実際のプログラムでは標準出力への書き出しは（例えば）ランク 0 のプロセスのみが行うようにします (4-4-2 節参照)。

図 2-1-1

並列ジョブをサポートするソフトウェア

並列ジョブの開発や実行をサポートするために、以下のようなソフトウェアが提供されていることがありますが、マシン環境によって状況が異なるので、ここでは簡単に説明します。なお、私自身は (3)〜(5) のツールは使ったことがありません。

(1) 並列ジョブ管理
　並列ジョブを実行したとき、各プロセスをどのノードに割り当てるかなど、並列ジョブの実行に関する管理を行います。

(2) メッセージ交換ライブラリー
　並列プログラムが、ノード間でデータの通信を行うためにコールする、通信ルーチンのライブラリーです。MPI（後述）というライブラリーが一般的に使われます。

(3) 並列デバッガー
　並列プログラム用の `dbx` デバッガーです。なお、私は並列プログラムのデバッグは Fortran の `PRINT` 文のみで行っています。

(4) 並列プロファイラー
　プログラムに含まれる各サブルーチンの CPU 時間の割合を調べるための UNIX コマンド `prof`、`gprof` の並列版です。

(5) 可視化ツール
　並列ジョブの活動状況を可視化して表示します。

第3章　メッセージ交換サブルーチン

第１章ではメッセージ交換について、『本当の意味の並列化に伴う矛盾（副作用）を解消するために、必要最小限、仕方なしに行うのがメッセージ交換』などと、すげなく扱ってしまいました。メッセージ交換は英話で言えば単語に相当し、プログラムの並列化は英会話に相当します。従って、プログラムの並列化を行うためにはメッセージ交換サブルーチンの使い方をしっかり学ぶ必要があります。

3-1　メッセージ交換ライブラリー

（注）本書中で、「マシン環境によって動作が異なります。」等と記載した部分は、並列計算機のハードウェア、ソフトウェア、コンパイラ、MPI ライブラリー等によって動作の異なる（可能性のある）部分です。ただし、私は 2 種類の計算機しか使用した事がないので、上記の記載をした以外の部分でもマシン環境によって動作の異なることがあるかもしれません。

MPI 概要

分散メモリー型のマシンで並列計算を行う際、メッセージ交換（通信）が必要になります。MPI が作成される以前に、PVM (Parallel Virtual Machine) などいくつかのメッセージ交換ライブラリーが存在しました。

しかし並列プログラムの互換性の観点から、ライブラリーの標準化の機運が高まり、米国のコンピュータメーカーや大学の研究者が MPI フォーラムという組織を結成し、既存のライブラリーの長所を集めた MPI (Message-Passing Interface) と呼ばれる基本仕様を 1995 年に作成しました (参考文献 [15] [27] 参照)。この仕様を MPI-1 と呼んでいます。

その後、MPI-1 の拡張機能仕様である MPI-2 が作成されました。本書では MPI-1 の範囲で説明を行います。

MPI-1	MPI-2
基本仕様	拡張機能仕様

MPI は Fortran および C からコールすることができます (本書では全て Fortran で説明を行います)。

MPI-1 に含まれているサブルーチンは図 3-1-1 に示すように 6 つに分類されます。そしてサブルーチンの総数は何と 128 個（処理系によって多少異なります）にもなります。では、この 128 個のサブルーチンを全てマスターしないと並列プログラムは作成できないのでしょうか？

サブルーチン分類	数
1対1通信	39
集団通信	16
派生データ型	13
コミュニケータ	30
プロセス・トポロジー	16
環境管理	14
合　計	128

図 3-1-1

答えは『No!』です。実はこれらの 128 個の中には、何のために使用するのかよく分からないルーチンや、単に照会をするだけのサブルーチンも多く含まれており、コンピュータメーカーで MPI の処理系を開発する人や並列処理の研究者以外は、そのようなルーチンは知らなくてもよいと言っても過言ではありません。そして、よほど特殊な並列化を行わない限り、20 個程度を知っていれば十分です (その中でも特によく使用するのは 10 個程度です)。そこで本書では、重要と思われる約 20 個のみを取り上げ、それ以外のサブルーチンは全て割愛することにしました。

付録について

MPI で提供されている主なサブルーチンの使用方法を、巻末の付録にまとめました。また付録内の「機能」の「関連する節」の所に、各サブルーチンが出てくる本書内の節を示しました。

- 言うまでもありませんが、本書は正式マニュアルではありませんので、正式な使用方法は参考文献 [15] [27] を参照して下さい。
- 付録に掲載したのは Fortran 版の MPI ルーチンの使用方法です。例えば Fortran 版で引数が整数になっていても、C 言語版では別のデータ型で宣言しなければならない場合がありますので、C 言語のユーザーの方は、参考文献 [15] [27] 内の C 言語の説明箇所を参照して下さい。
- Fortran 版の MPI ルーチンで引数が整数の場合、下記の下線部を付けて宣言しなければならない場合があります (例えば付録の `MPI_TYPE_CREATE_RESIZED` 参照)。

```
INTEGER(KIND=MPI_ADDRESS_KIND) N
CALL MPI_xxx(〜,N,〜)
```

多次元配列のメモリー上での並び

多次元配列の一部のデータを通信するような場合、配列の要素がメモリー上でどのように並んでいるかを知っておく必要があります。

まず、本テキストでは、2 次元配列の図は、図 3-1-2 (1) に示すように、1 次元目を縦方向、2 次元目を横方向に描いています。この 2 次元配列の各要素が、メモリー上でどのように並んでいるかを説明します。

Fortran の場合、2 次元配列の各要素は、左側の添字が先に動く順番にメモリー上に並びます。例えば 2 次元配列 `A(3,4)` は、メモリー上では図 3-1-2 (2) に示す順に並びます。図 3-1-2 (1) では、■, ▲, ◆, ●の各要素は一見連続しているように見えますが、実際には図 3-1-2 (2) に示すように、メモリー上ではとびとびになっているため、それを意識して通信を行う必要があります。

3 次元以上の配列の場合も、最も左側の添字が一番先に動く順番にメモリー上に並びます

一方 C 言語の場合、Fortran と仕様が逆になっており、2 次元配列の各要素は、右側の添字が先に動く順番にメモリー上に並びます。例えば 2 次元配列 `a[3][4]` は、メモリー上では図 3-1-3 (2) に示す順に並びます。

■A(1,1)	▲A(1,2)	◆A(1,3)	●A(1,4)
A(2,1)	A(2,2)	A(2,3)	A(2,4)
A(3,1)	A(3,2)	A(3,3)	A(3,4)

図 3-1-2 (1)　(Fortran) 2 次元配列 `A(3,4)`

■a[0][0]	▲a[0][1]	◆a[0][2]	●a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

図 3-1-3 (1)　(C 言語) 2 次元配列 `a[3][4]`

:
■A(1,1)
A(2,1)
A(3,1)
▲A(1,2)
A(2,2)
A(3,2)
◆A(1,3)
A(2,3)
A(3,3)
●A(1,4)
A(2,4)
A(3,4)
:

図 3-1-2 (2)　(Fortran) `A(3,4)` のメモリー上の配置

:
■a[0][0]
▲a[0][1]
◆a[0][2]
●a[0][3]
a[1][0]
a[1][1]
a[1][2]
a[1][3]
a[2][0]
a[2][1]
a[2][2]
a[2][3]
:

図 3-1-3 (2)　(C 言語) `a[3][4]` のメモリー上の配置

3-2　環境管理

本節では環境管理に関するサブルーチンについて説明します。なお、本書の Fortran プログラム例は原則として大文字で書きますが、Fortran は大文字と小文字の区別がないので、小文字で書いても構いません。

まず図 3-2-1 (1) のプログラムを作成し、3 プロセスで実行した結果を図 3-2-1 (2) に示します。図中でプログラムは 3 つあるように見えますが、SPMD モデルのためプログラムは 1 つです。

図 3-2-1 (1) のプログラムは、MPI を使用した最も簡単なプログラムであると言えます。以後各ステートメントについて説明します。

図 3-2-1 (1)

```
[aoyama@node01]/u/aoyama: mpirun -np 3 a.out ↵
  NPROCS = 3 MYRANK = 0
  NPROCS = 3 MYRANK = 1
  NPROCS = 3 MYRANK = 2
```

図 3-2-1 (2)

インクルードファイル mpif.h

図 3-2-1 (2) ①のインクルードファイル mpif.h は「/usr」などのディレクトリーの下に置かれており、置かれる場所や中身は MPI の処理系によって異なります。ユーザーは mpif.h の中身について知る必要はありませんが、参考までに例を図 3-2-2 (1) に示します。図中の「MPI_xxx」という整数 (例えば下線の「MPI_COMM_WORLD」) は MPI が内部的に使用する定数で、各定数の具体的な値が PARAMETER 文で設定されています。ユーザーは「MPI_xxx」の各定数を、MPI のサブルーチンの引数で指定しますが、定数名だけを知っていれば良く、PARAMETER 文で設定されている具体的な値については関知する必要はありません。

なお、「MPI_」、「PMPI_」で始まる変数名、関数名、サブルーチン名をユーザープログラムで使用すると、MPI が使用する変数と偶然一致して誤動作する危険性がありますので、使用しないで下さい。

```
  :
integer    MPI_COMM_WORLD
parameter (MPI_COMM_WORLD=0)
integer    MPI_INTEGER   ,MPI_REAL4   ,MPI_REAL   ,MPI_REAL8
parameter (MPI_INTEGER=18,MPI_REAL4=19,MPI_REAL=19,MPI_REAL8=20)
  :
```

図 3-2-2 (1)　mpif.h の中身の例

次に mpif.h の指定方法について説明します。図 3-2-1 (1) で、「CALL MPI_〜」となっているサブルーチン (②，③，④，⑥) は MPI のサブルーチンです。プログラム内のメインルーチンとサブルーチンの内、MPI のサブルーチンを使用した全てのルーチンで、必ず①の「INCLUDE 'mpif.h'」を指定して下さい（必須)。なお、Fortran は大文字と小文字の区別がありませんが、「INCLUDE 'MPIF.H'」のように「mpif.h」の部分を大文字で書くとエラーになるマシン環境もありますので、小文字で書いた方がよいでしょう。

指定する場所は、図 3-2-2 (2) のⓐに示すように、宣言文の先頭 (ただし USE 文と IMPLICIT 文の後) に置くのがよいでしょう。理由ですが、例えばⓑの下線部の MPI_STATUS_SIZE (意味は 3-4 節で説明します) のように、MPI が使用する定数をプログラム内で使用することがあり、MPI_STATUS_SIZE の値は前述のようにⓐのファイル内の PARAMETER 文で定義されているため、ⓑはⓐの後に置く必要があるからです。

別の方法として、図 3-2-2 (3) のⓒのように、「INCLUDE 'mpif.h'」や並列化に関係するその他の変数を MODULE 文 (Fortran 90 の機能) の中に含める方法もあり、私はこの方法を用いています (4-8-1 節参照)。

```
PROGRAM MAIN(またはサブルーチン)
USE xxxx
IMPLICIT REAL*8(A-H,O-Z)
INCLUDE 'mpif.h'                ⓐ
通常の宣言文
INTEGER ISTATUS(MPI_STATUS_SIZE) ⓑ
  :
CALL MPI_xxx
  :
```

図 3-2-2 (2)

```
MODULE PARA
INCLUDE 'mpif.h'                ⓒ
  :
END
PROGRAM MAIN(またはサブルーチン)
USE PARA
IMPLICIT REAL*8(A-H,O-Z)
通常の宣言文
  :
```

図 3-2-2 (3)

MPI_INIT

まず付録の MPI_INIT の説明を一読して下さい…。

図 3-2-1 (1) の②は MPI 環境の初期化処理を行うサブルーチンで、全ての MPI ルーチンの一番最初に 1 回だけ、必ずコールする必要があります（必須)。MPI_INIT をコールする前にユーザープログラムが存在しても構いません。

MPI_INIT で指定する整数の引数 ierror (図 3-2-1 (1) では IERR) は、実は本サブルーチンに限らず MPI の全てのサブルーチンの最後の引数でも同様に指定します。引数 IERR は必ず指定する必要がありますが、使用することはほとんどありません。IERR には、コールした MPI のサブルーチンが正常に終了すれば「MPI_SUCCESS」(具体的な値はインクルードファイル mpif.h の中で与えられています) が、エラーが発生した場合はそれ以外の値が戻ってきます。しかし、並列ジョブを実行していて MPI のサブルーチンでエラーが発生した場合、自動的にジョブは異常終了し、エラーメッセージが標準エラー出力に表示されます。従って、通常、MPI のサブルーチンをコールした後にユーザープログラムで引数 IERR の値をいちいちチェックする必要はありません。しかし、必要がないからと言って IERR 自体を引数として指定しないと、「segmentation fault」等でエラーになりますので、IERR は「オマジナイ」として必ず指定するようにして下さい。

なお、コールしたサブルーチンでエラーが発生した場合、IERR に戻る「MPI_SUCCESS」以外の値と実際のエラーメッセージの対応は、「CALL MPI_ERROR_STRING」というサブルーチンをコールすることによって調べることができますが、本書では説明は省略します。

MPI_FINALIZE

まず付録の MPI_FINALIZE の説明を一読して下さい…。

図 3-2-1 (1) の⑥は MPI 環境の終了処理を行うサブルーチンで、全ての MPI ルーチンの一番最後に 1 回だけ、必ずコールする必要があります（必須)。MPI_FINALIZE の後にユーザープログラムが存在しても構いません。

前述の MPI_INIT は、コールし忘れると通常実行時にエラーで異常終了しますが、MPI_FINALIZE をコールせずに終了した場合、マシン環境によっては誤動作する（実行中の他のプロセスを強制的に終了させてしまう、あるいは実行するたびに計算結果が異なる再現性のないエラーとなる）ことがあります。そしてその原因が、よもや MPI_FINALIZE のコールし忘れだとは、通常気付きません。従って、プログラムが終了する全ての箇所 (STOP 文または END 文) で、必ず 1 度だけ MPI_FINALIZE をコールしてから終了するようにして下さい。

MPI_COMM_SIZE/MPI_COMM_RANK

MPI_COMM_SIZE と MPI_COMM_RANK は、図 3-1-1 の分類では「環境管理」でなく、「コミュニケータ」に属しますが、便宜的にここで説明します。

例えば「mpirun -np 3 a.out ⏎」で並列ジョブを実行すると、図 3-2-3 に示すように 3 つのプロセスが作成され、各プロセスにはランク 0〜2 のいずれかの値が付けられます。その際、この 3 つのプロセス（全プロセス）からなるグループが自動的に作成され、MPI_COMM_WORLD というグループ名（コミュニケータと呼ばれます）が付けられます。(正確には、MPI_COMM_WORLD は MPI が内部的に使用する定数（整数）で、前述のように mpif.h 内の PARAMETER 文で値が定義されていますが、この値をユーザーが関知する必要はありません。)

一方、ユーザーが何らかの理由で、図 3-2-3 のように例えば 2 つのプロセスからなる IWORLD2 というグループ名（コミュニケータ）のグループを作成する事もできます（作成方法は後述します)。グループを作成する際、どのプロセスをランク 0 にし、どのプロセスをランク 1 にするかをユーザーが指定します (ランク 0 と 1 は、ランク 0〜2 とは別です)。なお、このように新規にグループを作成する事はほとんどありません。

図 3-2-3

上記の例を元に、図 3-2-1 (1) の③について説明します。③はグループに関する情報を照会するサブルーチンです。まず付録の MPI_COMM_SIZE の説明を一読して下さい…。

上記の例では、コミュニケータが MPI_COMM_WORLD であるグループに所属するプロセスの数は 3 なので、図 3-2-1 (1) の③に示すように、変数 NPROCS に 3 が戻ります。なお本書では、全プロセス数を表す変数（整数）を NPROCS という名前に統一します。

上記のようにコミュニケータが IWORLD2 であるグループが作成されている場合、③で MPI_COMM_WORLD の代わりに IWORLD2 を指定すると、変数 NPROCS に 2 が戻ります。

なお、並列プログラムを 1 プロセスで実行すると、誤動作する可能性がありますので (4-8-2 節の「プログラムの並列化方法のミス (1)」参照)、並列プログラムの開始時に、全プロセス数 (NPROCS) が 1 以下の場合はエラーで終了させるようにして下さい (4-8-1 節の「(5) 並列に動作させる最低限の修正」参照)。

図 3-2-1 (1) の④もグループに関する情報を照会するサブルーチンです。付録の MPI_COMM_RANK の説明を一読してください…。

④に示すように、ランク 0〜2 のプロセスの変数 MYRANK にそれぞれ 0〜2 が戻ります。なお本書では、ランクを表す変数（整数）を MYRANK という名前に統一します。

上記のようにコミュニケータが IWORLD2 であるグループが作成されている場合、④で MPI_COMM_WORLD の代わりに IWORLD2 を指定すると、変数 MYRANK に 0 または 1 が戻ります。

③と④で得られた NPROCS と MYRANK の値を⑤で書き出したため、図 3-2-1 (2) の結果が得られました。

まとめ

以上をまとめます。まず、並列プログラムでは図 3-2-1 (1) の②と⑥は必須です。③と④は全プロセス数 (NPROCS) あるいは自分のランク (MYRANK) を取得したい場合に使用しますが、これらもほとんどの並列プログラムでは必須です。従って、②〜④はいわば三種の神器、あるいはオマジナイとして、プログラムの最初に 1 箇所にまとめて一度だけ実行するのがよいでしょう。そして MPI のサブルーチンをコールしたユーザールーチンでは、①のインクルード文を指定するのを忘れないようにして下さい。

本節では、環境管理に関するサブルーチンを 4 つ紹介しました。それ以外に「MPI_ABORT」というサブルーチンも知っておいた方が便利です。MPI_ABORT は 4-4-1 節と 4-8-1 節で説明します。

同じプログラムで各プロセスに異なる動作をさせるには

2章で説明した図2-1-1とその実行結果を以下に再掲します。このプログラムを並列に実行すると全プロセスが同じメッセージを書き出しました。SPMDモデルではプログラムは1つしかありませんが、図2-1-1では各プロセスの動作自体も全く同一であったため、全プロセスが同じ結果を出力したわけです。

なお、図2-1-1で説明したように、このプログラムはマシン環境によっては並列に動作しないことがあります。

```
PROGRAM MAIN
PRINT *,'Hello MPI'
END
```

図2-1-1と同じ

```
[aoyama@node01]/u/aoyama: mpirun -np 2 a.out ↵
  Hello MPI
  Hello MPI
```

それでは、1つのプログラムでありながら各プロセスで異なる動作をさせるにはどのようにしたらよいのでしょうか？実は、先ほど紹介した『CALL MPI_COMM_RANK』によって得られる自分のプロセスのランク(MYRANK)が、1つのプログラムでありながら各プロセスで異なる動作をさせるための全ての根源となります。それではMYRANKの使用例を見てみましょう。

【例1】MYRANKをIF文の条件式に使用

図3-2-4(1)では、MYRANKをIF文の条件式の中で使用しています。各プロセスでは、図中の下線を引いた部分のIF文のみが真となり、二重線のステートメントが実行されます。このようにすれば、同じプログラムでありながらプロセスによって異なるステートメントを実行させることができます。

ランク0

```
PROGRAM MAIN
INCLUDE 'mpif.h'
CALL MPI_INIT(IERR)
CALL MPI_COMM_SIZE
& (MPI_COMM_WORLD,NPROCS,IERR)
CALL MPI_COMM_RANK
& (MPI_COMM_WORLD,MYRANK,IERR)
↳0
IF (MYRANK == 0) THEN
  PRINT *,'I am Rank 0'
ELSEIF (MYRANK == 1) THEN
  PRINT *,'I am Rank 1'
ELSEIF (MYRANK == 2) THEN
  PRINT *,'I am Rank 2'
ENDIF
CALL MPI_FINALIZE(IERR)
END
```

ランク1

```
PROGRAM MAIN
INCLUDE 'mpif.h'
CALL MPI_INIT(IERR)
CALL MPI_COMM_SIZE
& (MPI_COMM_WORLD,NPROCS,IERR)
CALL MPI_COMM_RANK
& (MPI_COMM_WORLD,MYRANK,IERR)
↳1
IF (MYRANK == 0) THEN
  PRINT *,'I am Rank 0'
ELSEIF (MYRANK == 1) THEN
  PRINT *,'I am Rank 1'
ELSEIF (MYRANK == 2) THEN
  PRINT *,'I am Rank 2'
ENDIF
CALL MPI_FINALIZE(IERR)
END
```

ランク2

```
PROGRAM MAIN
INCLUDE 'mpif.h'
CALL MPI_INIT(IERR)
CALL MPI_COMM_SIZE
& (MPI_COMM_WORLD,NPROCS,IERR)
CALL MPI_COMM_RANK
& (MPI_COMM_WORLD,MYRANK,IERR)
↳2
IF (MYRANK == 0) THEN
  PRINT *,'I am Rank 0'
ELSEIF (MYRANK == 1) THEN
  PRINT *,'I am Rank 1'
ELSEIF (MYRANK == 2) THEN
  PRINT *,'I am Rank 2'
ENDIF
CALL MPI_FINALIZE(IERR)
END
```

図3-2-4(1)

```
[aoyama@node01]/u/aoyama: mpirun -np 3 a.out ↵
  I am Rank 0
  I am Rank 1
  I am Rank 2
```

図3-2-4(2)

【例 2】MYRANK を DO ループの分割に使用した例

次に、MYRANK を DO ループの分割に使用する例について説明します。図 3-2-5 (1) は、配列 N に入っている 9 個のデータの合計を変数 ISUM に求め、それを書き出すプログラムです (図 3-2-5 (2) 参照)。各 DO ループの反復は『DO I=1,9』なので、これを 3 つのランク 0, 1, 2 のプロセスにそれぞれ『DO I=1,3』、『DO I=4,6』、『DO I=7,9』のように割り振ることにします。

```
PROGRAM MAIN
INTEGER N(9)
DO I = 1, 9
  N(I) = I
ENDDO
ISUM = 0
DO I = 1, 9
  ISUM = ISUM + N(I)
ENDDO
PRINT *,'ISUM = ',ISUM
END
```

図 3-2-5 (1)

図 3-2-5 (2)

MYRANK を使用して DO ループを分割したプログラムを図 3-2-5 (3) に示します。

- まず①と②で、MYRANK の値を使用して、各プロセスが担当する ISTA と IEND の値を求めます。
- 次に各 DO ループの反復を『I=1,9』の代わりに『I=ISTA,IEND』とします。
- これを 3 プロセスで実行した場合、各プロセスは自分の担当するループ反復のみを計算し、変数 ISUM には各プロセスが求めた小計が入ります。実行結果は図 3-2-5 (4) のようになります。

この例は実際の並列プログラムにだいぶ近くなってきました。実際のプログラムでも多くの場合、この例のように DO ループを分割して並列化を行います。ただし、この例では DO ループの反復数 (9) とプロセス数 (3) があらかじめわかっており、しかもちょうど割りきれました。実際のプログラムでは、N1〜N2 までの反復を NPROCS 個のプロセスに割り振るというように、処理量とプロセス数が任意の値でも対処できるように分割を行います。この方法については 4 章で説明したいと思います。

ここで 1 章の説明を思い出して下さい。図 3-2-5 (3) で行ったのは、『本当の意味の並列化とは、計算量を $1/n$ に縮小すること (ノード数が n 台の場合)』の部分です。この修正の結果、『本当の意味の並列化に伴う矛盾（副作用）』が発生しました。この例では『変数 ISUM には合計でなく小計しか求まっていない』ことが矛盾（副作用）になります。そこで『矛盾（副作用）を解消するために、必要最小限、仕方なしにメッセージ交換』を行います。この例では、『小計 ISUM を全プロセスで合計するために通信』を行います。

図 3-2-5 (3) の説明はいったんここで中断し、続きはメッセージ交換の説明を終えた 3-3-6 節で説明します。

図 3-2-5 (3)

```
[aoyama@node01]/u/aoyama: mpirun -np 3 a.out ↵
  ISUM = 6
  ISUM = 15
  ISUM = 24
```

図 3-2-5 (4)

3-3　集団通信

本節では、MPI で提供されているサブルーチンの中で最もよく使用する、集団通信に関するサブルーチンの使用方法について説明します。

3-3-1　集団通信サブルーチンの一般的な規則

集団通信サブルーチンを使用すると、1 回のサブルーチンコールで複数プロセスがメッセージ交換（データのやり取り）を一斉に行うことができ、また使用方法も簡単なため大変便利です。各集団通信サブルーチンに共通する規則は次の通りです。

- その通信に参加する全プロセスが集団通信サブルーチンをコールします。
- 一斉に通信を行うプロセスの範囲をコミュニケータとしてサブルーチンコールの引数で指定します。

これを図 3-3-1 (1) (2) の例で示します。3 つのプロセスが並列に実行しているとします。また、集団通信サブルーチンの例として、あるプロセスからその他の全プロセスにデータを送信するサブルーチン『CALL MPI_BCAST』を取り上げます（詳細は後述します）。

図 3-3-1 (1) のように、引数にコミュニケータ MPI_COMM_WORLD を指定して『CALL MPI_BCAST』をコールすると、コミュニケータ MPI_COMM_WORLD のグループに含まれるプロセス、すなわち 3 つの全プロセスで一斉を通信を行います。これに対し、図 3-3-1 (2) のように、引数にコミュニケータ IWORLD2 を指定してコールすると、コミュニケータ IWORLD2 のグループに含まれる右の 2 つのプロセスのみで一斉に通信が行われます。

通常 SPMD モデルでプログラムを並列化した場合、図 3-3-1 (2) のように一部のプロセスだけで通信を行うケースはほとんどないので、以後の説明では、全プロセスで集団通信サブルーチンを行うことを想定します。

図 3-3-1 (1)

図 3-3-1 (2)

3-3-2　集団通信サブルーチンと同期

「並列プログラムは同期を取るのが難しい」という言葉を聞く事があります。結論を先に述べるとこれは間違いで、「並列プログラムでは、（通常）同期を取るということをいちいち気にする必要はない」が正解です。これについて、集団通信サブルーチンを例に以下で説明しますが、今の段階ではやや難しいので、内容は理解できなくても構いません。

図 3-3-2 (1) の並列プログラムは実際には 1 本のプログラムですが、ランク 0 とランク 1 のプロセスで一部処理が異なる部分があるため、別々に描かれています。ランク 0 はまず処理 1 を行い、処理 1 の最後に変数 A にデータをセットし、集団通信サブルーチンをコールします。この集団通信サブルーチンは内部的に、変数 A をランク 1 に送信する処理を行うとします。ランク 0 は最後に処理 2 を実行します。

一方ランク 1 は処理 1 が終わった後、集団通信サブルーチンをコールします。この集団通信サブルーチンは内部的に、ランク 0 から送られたデータを変数 B に受信する処理を行うとします。ランク 1 は最後に処理 2 を行い、受信した変数 B を参照して計算を行います。

ランク 0 とランク 1 のプロセスは独立に動くので、どちらのプロセスが先に処理 1 を終了して集団通信サブルーチンをコールするかは分かりません。まず図 3-3-2 (2) に示すように、ランク 0 が先に集団通信サブルーチンをコールした場合を説明します（図の下方向は時間軸）。ランク 0 が集団通信サブルーチンをコールした時点で、ランク 1 はまだ処理 1 を行っているので、コールされた集団通信サブルーチンは「待ち」になります。しかしランク 0 のユーザープログラムは、通信が行われるまで「待ち」になる必要はないので、処理 2 に進みます。そしてランク 1 が集団通信サブルーチンをコールした時点で通信が行われます。

逆に図 3-3-2 (3) のように、ランク 1 が先に集団通信サブルーチンをコールした場合、ランク 0 はまだ処理 1 を行っているので、コールされた集団通信サブルーチンは「待ち」になります。このときランク 1 のユーザープログラムは、処理 2 に進まずに自動的に「待ち」になります (もし「待ち」にならずに先に進むと、次の処理 2 で、まだ受信していない変数 B を参照してしまい、誤動作するので)。そしてランク 0 が集団通信サブルーチンをコールした時点で通信が行われます。受信したデータが変数 B に入ると、ランク 1 のユーザープログラムの「待ち」は解除され、処理 2 に進み、変数 B を参照して計算を行います。

上記のいずれの場合も、ランク 0、ランク 1 は共に正しく動作をしています。これは集団通信サブルーチンが、ユーザープログラムを先に進ませるか「待ち」にするかを適切に調整しているためです。従って集団通信サブルーチンをコールする場合、ユーザープログラム自身で、他のプロセスの処理がどこまで進んでいるのかをいちいち気にする必要はありません。言い換えると明示的に同期を取る必要はありません。

MPI では、同期を取るためだけのメッセージ交換サブルーチンとして、MPI_BARRIER（付録参照）が提供されています。初心者がプログラムを並列化し、なかなか答えが合わず、適当に MPI_BARRIER を挿入したらなぜか答えが合うようになった、という例がたまにあります。しかし上記で説明したように、同期を取らないと答えが合わないということは通常の並列プログラムでは発生しないので、MPI_BARRIER を挿入しないとなぜ答えが合わないのか、真の原因を把握するようにして下さい。

図 3-3-2 (1)

図 3-3-2 (2)

図 3-3-2 (3)

3-3-3　集団通信サブルーチンの種類

MPI で提供されている集団通信サブルーチン

MPI で提供されている集団通信サブルーチンを下記に示します (* は MPI-2 で提供されているサブルーチンです)。各サブルーチンは以下のように分類されます。次節以降では、(グループ 1)〜(グループ 3) のサブルーチンの代表例として、下線のサブルーチンを説明します。

なお、（その他）のサブルーチンはデータの通信自体は行ないません。

(グループ 1) ◎ MPI_BCAST

(グループ 2) ○ MPI_GATHER, MPI_SCATTER, ○ MPI_ALLGATHER, △ MPI_ALLTOALL

○ MPI_GATHERV, MPI_SCATTERV, ○ MPI_ALLGATHERV, △ MPI_ALLTOALLV, *MPI_ALLTOALLW

(グループ 3) ○ MPI_REDUCE, ○ MPI_ALLREDUCE, MPI_SCAN, *MPI_EXSCAN, MPI_REDUCE_SCATTER

（その他）△ MPI_BARRIER, △ MPI_OP_CREATE, MPI_OP_FREE

参考までに、上記のうち使用頻度の高いサブルーチンに以下の印を付けておきます。それ以外のサブルーチンは（少なくとも私は）ほとんど使用しません。

◎：よく使用するサブルーチンです。

○：よく使用するサブルーチンですが、MPI-1 では機能的に不便な点があります (3-3-5 節、4-6-3 節参照)。

△：たまに使用するサブルーチンです。

3-3-4　MPI_BCAST

まずMPI_BCASTについて説明します。このサブルーチンは、1つのプロセスから、コミュニケータで指定したグループ内のその他全てのプロセスにメッセージを一斉に送信します。例えば図3-3-3 (1) のように、大きさ4の整数配列IMSGがあり、ランク0のプロセスの配列IMSGには1〜4が、その他のプロセスのIMSGには0が入っているとします。MPI_BCASTを実行すると、ランク0のプロセスのIMSGの値がその他のプロセスのIMSGに一斉に送信されます。ここで、付録のMPI_BCASTの説明を一読して下さい…。

図3-3-3 (1) を行うプログラム例を図3-3-3 (2) に、実行結果を図3-3-3 (3) に示します。

図 3-3-3 (1)

```
PROGRAM MAIN
INCLUDE 'mpif.h'                          MPIが使用するインクルードファイルを指定します。
INTEGER IMSG(4)                           送受信に使用するバッファーを宣言します。
CALL MPI_INIT(IERR)                                ①MPIの環境を初期設定します。
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR) ②プロセス数NPROCSを取得します。
CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR) ③自分のランクMYRANKを取得します。
IF (MYRANK == 0) THEN
  DO I = 1, 4                             ④ランク0のプロセスだけがこの部分を実行し、配列IMSGに
    IMSG(I) = I                           データをセットします。
  ENDDO
ELSE
  DO I = 1, 4
    IMSG(I) = 0                           ⑤ランク0以外のプロセスがこの部分を実行し、配列IMSGを
  ENDDO                                   ゼロクリアします。
ENDIF
PRINT *,'MYRANK = ',MYRANK,'BEFORE = ',IMSG  ⑥通信前の配列IMSGの値を書き出します。
CALL MPI_BCAST(IMSG,4,MPI_INTEGER,0,MPI_COMM_WORLD,IERR)
                                          ⑦ランク0からその他全てのプロセスにIMSGの値を送信します。
PRINT *,'MYRANK = ',MYRANK,' AFTER = ',IMSG  ⑧通信後の配列IMSGの値を書き出します。
CALL MPI_FINALIZE(IERR)
END
```

図 3-3-3 (2)

```
[aoyama@node01]/u/aoyama: mpirun -np 3 a.out ↵
  MYRANK = 0 BEFORE = 1 2 3 4
  MYRANK = 0 AFTER  = 1 2 3 4
  MYRANK = 1 BEFORE = 0 0 0 0   表示される順序はタイミングによって異なります。
  MYRANK = 1 AFTER  = 1 2 3 4
  MYRANK = 2 BEFORE = 0 0 0 0
  MYRANK = 2 AFTER  = 1 2 3 4
```

図 3-3-3 (3)

以下に、図3-3-3 (2) の説明を行います。

- ①～③を行うと、前述のように全プロセス数 (NPROCS) と自分のランク (MYRANK) が得られます。ただし本プログラムではNPROCSは使用しません。次に④と⑤で、ランク0のプロセスでは配列IMSGに1～4を、その他のプロセスでは配列IMSGに0をセットします。なお、⑤では説明の都合上、配列IMSGをゼロクリアしていますが、実際にはMPI_BCASTによって送信されたデータによって上書きされるため、ゼロクリアする必要はありません。
- ⑦で集団通信サブルーチンMPI_BCASTをコールします。1つ目の引数 (IMSG) には、通信に使用するバッファーを指定します。バッファーとは、送信したいデータ、または受信したデータを入れる変数や配列（の先頭アドレス）のことです。なお、本書では、バッファーを送信（受信）バッファーと呼ぶ場合もあります。
  MPI_BCASTでは、1つ目の引数の意味が送信元のプロセスと宛先のプロセスで異なります。送信元のプロセス (本例ではランク0のプロセス) では、送信したいデータの入った送信バッファーを、宛先のプロセス (本例ではランク0以外のプロセス) では、受信したデータを入れる受信バッファーを指定します。⑦では全プロセスが同じ名前の配列IMSGを指定しています。
  配列を指定する場合、⑦のように配列名そのもの（つまり配列の先頭アドレス）を指定する方法以外に、図3-3-3 (4) のように配列の途中のアドレスを指定したり、図3-3-3 (5) のようにプロセスのランクによって配列の異なるアドレスを指定してもかまいません。
  ⑦の前後 (⑥と⑧) で全プロセスの配列IMSGの値を書き出しています。集団通信サブルーチンの使用法を習得する段階では、このように、通信前と通信後のバッファーの値をランクの値とともに書き出して、通信が行われている様子を確認するのがよいでしょう。

```
    :
  CALL MPI_BCAST(IMSG(2),2,MPI_INTEGER,0,MPI_COMM_WORLD,IERR)
    :
```

図3-3-3 (4)

```
    :
  CALL MPI_BCAST(IMSG(MYRANK+1),2,MPI_INTEGER,0,MPI_COMM_WORLD,IERR)
    :
```

図3-3-3 (5)

図3-3-3 (6)

図3-3-3 (7)

```
  N = 4
  CALL MPI_BCAST(IMSG,N,MPI_INTEGER,0,MPI_COMM_WORLD,IERR)
    :
```

図3-3-3 (8)

- MPI では、送受信するデータのことをメッセージと呼びます。MPI_BCAST の 2 つ目の引数には、送信するメッセージを構成するデータの個数を、3 つ目の引数にはメッセージのデータ型を指定します。
  データ型には、MPI があらかじめ提供している MPI の基本データ型、またはユーザーが自分で作成する派生データ型 (3-5 節参照) を指定します。
  MPI の基本データ型と Fortran のデータ型の対応を図 3-3-3 (9) に示します。マシン環境によっては、カッコをつけたデータ型やこれ以外のデータ型が提供されている場合もあります。なお、C 言語の基本データ型については 3-7 節を参照して下さい。

	Fortranのデータ型	MPIの基本データ型
整数型	INTEGER (INTEGER*1) (INTEGER*2) (INTEGER*4)	MPI_INTEGER (MPI_INTEGER1) (MPI_INTEGER2) (MPI_INTEGER4)
実数型	REAL (REAL*2) (REAL*4) (REAL*8) DOUBLE PRECISION	MPI_REAL (MPI_REAL2) (MPI_REAL4) (MPI_REAL8) MPI_DOUBLE_PRECISION
複素数型	COMPLEX (COMPLEX*16) (DOUBLE COMPLEX)	MPI_COMPLEX (MPI_COMPLEX16) (MPI_DOUBLE_COMPLEX)
その他	LOGICAL CHARACTER(1) 対応なし	MPI_LOGICAL MPI_CHARACTER MPI_BYTE, MPI_PACKED

図 3-3-3 (9)

（注）マシン環境によってはカッコをつけたデータ型などが提供されている場合もあります。
例えば図 3-3-3 (1) の例では整数型のデータを 4 つ送信するので、図 3-3-3 (2) の⑦に示すように 2 つ目の引数は『4』、3 つ目の引数は『MPI_INTEGER』になります。また、図 3-3-3 (4) (5) の例では整数型のデータを 2 つ送信するので、送信結果はそれぞれ図 3-3-3 (6) (7) のようになります。データの個数は、図 3-3-3 (8) の下線部のように変数で指定しても勿論かまいません。
いくつか補足します。

- 1 回の通信で送信できるバイト数はマシン環境によって異なり、無制限の場合と、制限のある場合があります。制限のある場合に、制限を越えたバイト数を送ってもエラーにならず、不完全なメッセージが送られてしまうマシン環境もありますので、お使いのマシンのマニュアルを確認して下さい。
- 集団通信ルーチンでは、送信バッファーで指定するバイト数 (データ個数 × データ型のバイト数) と、受信バッファーで指定するバイト数は一致していなければなりません。
- メッセージを、MPI のようにデータの個数とデータ型の 2 つで指定するのではなく、単にバイト数で指定した方が引数が 1 つなので簡単ですが、MPI では個数とデータ型を分離したことによって、ユーザーが自分独自のデータ型 (3-5 節の派生データ型) を作成することができます。

- MPI_BCAST の 4 つ目の引数には、送信元プロセスのランクを指定します。図 3-3-3 (1) ではランク 0 のプロセスが送信元なので『0』を指定します。この引数は、全プロセスが同じ値を指定する必要があります。SPMD モデルの場合、通常は図 3-3-3 (2) のように全プロセスが同一プログラムの同一の『CALL MPI_BCAST』を実行します。この場合、MPI_BCAST の 4 つ目の引数が⑦のように定数になっていれば、全プロセスが必然的に同じ値を指定することになるため問題はありません。しかし、図 3-3-3 (10) のような使い方は誤りです。この例では 4 つ目の引数に MYRANK を使用していますが、MYRANK の値はプロセスのランクによって異なっているため、どのプロセスが送信元なのかがわからなくなってしまいます。
  なお、全プロセスが同じ値を指定すべき引数は、付録の各サブルーチンの引数の説明の所に明記しました。

```
CALL MPI_BCAST(IMSG,4,MPI_INTEGER,MYRANK,MPI_COMM_WORLD,IERR)
```

図 3-3-3 (10)　×　送信元プロセスのランクの誤った使用例

- MPI_BCAST の 5 つ目の引数には、この通信に参加する全プロセスから構成されるグループのコミュニケータを指定します。図 3-3-3 (2) では前述の MPI_COMM_WORLD を指定しています。前にも述べましたが、5 つ目の引数に指定したコミュニケータのグループに含まれる全プロセスが『CALL MPI_BCAST』を実行する必要があります。SPMD モデルの場合、図 3-3-3 (2) のように、全プロセスが同一プログラムの同一の『CALL MPI_BCAST』を実行するので通常問題はありません。
- 今までは全て、各プロセスが全く同一の『CALL MPI_BCAST』を実行する例を示しました。SPMD モデルではこのようにプログラミングするのが一般的ですが、図 3-3-3 (11) のように、プロセスによって異なるステートメントの『CALL MPI_BCAST』を実行してもかまいません（ただしこのような使用法はまれですが）。
  なお、図 3-3-3 (11) では、下線部に示すように、プロセスによって異なる配列 (ランク 0 のプロセスでは IMSG、それ以外のプロセスでは KMSG) を送受信バッファーとして使用していますが、このような使い方も問題ありません。図 3-3-3 (11) の実行結果を図 3-3-3 (12) に示します。

```
   :
IF (MYRANK == 0) THEN
  CALL MPI_BCAST(IMSG,4,MPI_INTEGER,0,MPI_COMM_WORLD,IERR)
ELSE
  CALL MPI_BCAST(KMSG,4,MPI_INTEGER,0,MPI_COMM_WORLD,IERR)
ENDIF
   :
```

図 3-3-3 (11)

図 3-3-3 (12)

- MPI のサブルーチンの引数が少なかったり指定がおかしくても、一般にコンパイル・リンクではエラーになりません。この場合、実行時に Segmentation fault などのエラーで異常終了しますが、エラーメッセージを見ても原因が分からない場合がありますので、異常が起きた場合には MPI のサブルーチンの引数の数や属性に誤りがないかどうかをチェックして下さい。
- MPI の仕様書 (参考文献 [15]) では、ユーザーが MPI の使い方を間違えた場合、実行時に MPI が行うエラー処理について一般に規定していません。このため、例えば MPI_BCAST などの集団通信ルーチンで、送信バッファーの大きさより受信バッファーの大きさを誤って小さくしてしまったような場合でも、受信時にエラーにならず、エラーのリターンコード (ierror) にも正常終了の「MPI_SUCCESS」が返るマシン環境もあります。

3-3-5　MPI_GATHER

次に、MPI_GATHER について説明します。MPI_GATHER は図 3-3-4 (1) のように、各プロセスの送信バッファーのデータを、例えばランク 0 のプロセスの受信バッファーに集めます。実際のプログラムでは、並列化によって各プロセスが計算した部分的な計算結果を最後にランク 0 のプロセスに集め、ランク 0 のプロセスが代表して書き出しを行うような場合に使用します。

ここで、付録の MPI_GATHER の説明を一読して下さい…。

以下に MPI_GATHER に関する注意点を述べます。

- 前述の MPI_BCAST で説明した項目はほとんど MPI_GATHER でも適用されますが、1 つだけ異なる点があります。MPI_BCAST では引数で指定する送受信バッファーは 1 つだけで、送信元のプロセスでは送信バッファーとして、宛先のプロセスでは受信バッファーとして使用しました。実は、引数でバッファーを 1 つだけ指定するのは MPI_BCAST のみであり、その他の全ての集団通信サブルーチン (MPI_BARRIER を除く) では、送信バッファーと受信バッファーの 2 つを指定します。
- 以下に示すように、(グループ 2) のサブルーチンは、末尾に『V』が付いていないルーチンとついているルーチンがあります。以下で両者の相違を説明します。なお、この相違は MPI_SCATTER(V), MPI_ALLGATHER(V), MPI_ALLTOALL(V)(W) でも同様です。

  (グループ 2) MPI_GATHER, MPI_SCATTER, MPI_ALLGATHER, MPI_ALLTOALL

  MPI_GATHERV, MPI_SCATTERV, MPI_ALLGATHERV, MPI_ALLTOALLV, MPI_ALLTOALLW

【末尾に V のついていないルーチン (MPI_GATHER など)】

(1) 図 3-3-4 (1) の↕に示すように、送信するメッセージの長さは全プロセスで同一でなければなりません。

(2) 受信したメッセージは、送信元のランクの小さい順に受信バッファーに入ります。

【末尾に V のついているルーチン (MPI_GATHERV など)】

(1) 図 3-3-4 (2) に示すように、各プロセスで異なる長さの送信メッセージを送る事ができます。

(2) 受信したメッセージは、受信バッファーの任意の位置に置くことができます。

図 3-3-4 (1)　MPI_GATHER

図 3-3-4 (2)　MPI_GATHERV

- 以下に示す、送信バッファーと受信バッファーの両方を指定する集団通信サブルーチンでは、以下の制限があります。この制限に違反すると、タイミングによって実行結果が変わったりする可能性があるので十分注意して下さい。
  (グループ 2) MPI_GATHER, MPI_SCATTER, MPI_ALLGATHER, MPI_ALLTOALL
  MPI_GATHERV, MPI_SCATTERV, MPI_ALLGATHERV, MPI_ALLTOALLV(W)
  (グループ 3) MPI_REDUCE, MPI_ALLREDUCE, MPI_SCAN, MPI_REDUCE_SCATTER

送信バッファーと受信バッファー（の実際に使用する部分）はメモリー上で重なってはならない。

実際にプログラムを並列化する場合、手抜きで、配列を縮小 (4-5-7 節参照) せずに並列化することがよくありますが、このとき上記の制限が問題になります (1-2 節の図 1-2-2 (2) がその例です)。

例えば図 3-3-5 (1) のように、元の単体プログラムで使用していた配列 A をそのまま並列プログラムで全プロセスが使用し、各プロセスは配列 A のうち自分が計算した部分のみに値を入れたとします (図 3-3-5 (1) ではランク 0, 1, 2 はそれぞれ A(1), A(2), A(3))。そして、これらの計算結果を最後にランク 0 のプロセスに集めるとします。このとき、ランク 0 のプロセスの配列 A は、自分が計算した A(1) 以外の部分が空いているため、ランク 1 からの結果を A(2) に、ランク 2 からの結果を A(3) に入れるのが自然です。

ところが MPI_GATHER を使用した場合、図 3-3-5 (1) に示すようにランク 0 のプロセスの送信バッファーと受信バッファーが重複しており、上記の制限に引っ掛かってしまいます。従って、MPI_GATHER を使用する場合には、図 3-3-5 (2) に示すように新たに受信バッファー用の配列 B を宣言しなければなりません。

このような場合、後述する MPI_SEND や MPI_RECV などの 1 対 1 通信サブルーチンを使用すると、図 3-3-5 (1) のようにデータを収集することができますが、これについては 4-6-3 節で説明します。

なお、MPI-2 で追加された MPI_IN_PLACE という変数を使用すると、この制限を回避できる場合もあります。詳細は 3-8-1 節を参照して下さい。

図 3-3-5 (1)

図 3-3-5 (2)

3-3-6　MPI_REDUCE

MPI_REDUCE は通信しながら演算を行うタイプのサブルーチンで、同じグループのサブルーチンとしては他に MPI_ALL REDUCE, MPI_SCAN, MPI_REDUCE_SCATTER があります。まず、例によって付録の MPI_REDUCE の説明を一読して下さい…。

さて、MPI_REDUCE では通信しながら演算を行うわけですが、この『演算』というのは、積極的に演算しようということではなく、『通信しながら演算もいっしょに行った方が若干便利だから』と言う程度の意味合いです。とは言ってもわかりにくいと思いますので、これを図 3-3-6 (1) と図 3-3-6 (2) で説明します。なお、この例は 3-2 節の図 3-2-5 (3) の続きです。

図 3-3-6 (1)

```
  PROGRAM MAIN
  INCLUDE 'mpif.h'                          MPIが使用するインクルードファイルを指定します。
  INTEGER N(9)                              計算に使用する配列を宣言します。
  CALL MPI_INIT(IERR)                                MPIの環境を初期設定します。
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)  プロセス数NPROCSを取得します。
  CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)  自分のランクMYRANKを取得します。
  ISTA = MYRANK*3 + 1                       自分が担当する下限(ISTA)を設定します。
  IEND = ISTA + 2                           自分が担当する上限(IEND)を設定します。
  DO I = ISTA, IEND
    N(I) = I                                計算に使用する値を設定します。
  ENDDO
  ISUM = 0
  DO I = ISTA, IEND
    ISUM = ISUM + N(I)                      ①自分が担当する部分の小計を求め、結果をISUMに入れます。
  ENDDO
  CALL MPI_REDUCE(ISUM,ITMP,1,MPI_INTEGER,        ②ISUMの値を通信しながら合計し、総合計を
&                 MPI_SUM,0,MPI_COMM_WORLD,IERR)  ランク0のプロセスのITMPに入れます。
  ISUM = ITMP                               ③ITMPの値をISUMに戻します。
  IF (MYRANK==0) PRINT *,'ISUM = ',ISUM           ④ISUMの値を書き出します。
  CALL MPI_FINALIZE(IERR)
  END
```

図 3-3-6 (2)

```
[aoyama@node01]/u/aoyama: mpirun -np 3 a.out ↵
  ISUM = 45
```

図 3-3-6 (3)

このプログラムでは、①が終了した直後には、図 3-3-6 (1) に示すように配列 N のうち各プロセスが担当した部分の小計が ISUM に求まっています。最終的には小計の総合計をランク 0 のプロセスが持てばよいわけですから、例えばランク 1 と 2 のプロセスの ISUM (15 と 24) を前述の MPI_GATHER などでいったんランク 0 のプロセスに集め、その後でランク 0 のプロセスが『6+15+24』を計算するという方法でもよいわけです。しかしそれが若干面倒なので、通信しながら合計も同時に求めてしまおうということで、MPI_REDUCE が提供されています。

以下、図 3-3-6 (2) について補足します。

- MPI_REDUCE では引数に通信しながら行う演算を指定します。MPI の定義済み演算 (MPI があらかじめ提供している演算) を図 3-3-7 (1) (2) に示します (C 言語の場合は 3-7 節を参照して下さい)。図 3-3-6 (2) の②では、変数 ISUM に対して加算の演算を行うため、演算には『MPI_SUM』を指定しました。また図 3-3-7 (1) (2) からわかるように、各演算には、その演算で使用することのできる MPI のデータ型が決められています。なお、図 3-3-7 (1) (2) はマシン環境によって若干異なることがあります。
  MPI の定義済み演算の中でよく使用するのは MPI_SUM、MPI_MAX、MPI_MIN です。MPI_PROD は例えば階乗の計算を並列化する場合などに使用します。MPI_MAXLOC、MPI_MINLOC については、3-3-6-1 節で説明します。図 3-3-7 以外のデータ型で演算を行いたい場合（例えば倍精度複素数の加算)、3-3-6-2 節で説明するように、ユーザーが自分で演算を定義することができます。

MPIの基本データ型	MPIの定義済み演算
MPI_INTEGER (注1)	MPI_SUM (合計)
MPI_REAL (注2)	MPI_PROD(積)
MPI_DOUBLE_PRECISION (注3)	
MPI_COMPLEX (注4)	
MPI_INTEGER (注1)	MPI_MAX (最大)
MPI_REAL (注2)	MPI_MIN (最小)
MPI_DOUBLE_PRECISION (注3)	
MPI_2INTEGER	MPI_MAXLOC(最大と位置)
MPI_2REAL	MPI_MINLOC(最小と位置)
MPI_2DOUBLE_PRECISION(注5)	

図 3-3-7 (1)

下記が使用できるマシン環境もあります。
(注 1) MPI_INTEGER4
(注 2) MPI_REAL4
(注 3) MPI_REAL8
(注 4) MPI_COMPLEX16, MPI_DOUBLE_COMPLEX
(注 5) MPI_2REAL8
ただし、

$$\mathrm{MPI_2INTEGER} = \{\mathrm{MPI_INTEGER}, \mathrm{MPI_INTEGER}\}$$
$$\mathrm{MPI_2REAL} = \{\mathrm{MPI_REAL}, \mathrm{MPI_REAL}\}$$
$$\mathrm{MPI_2DOUBLE_PRECISION} = \{\mathrm{MPI_DOUBLE_PRECISION}, \mathrm{MPI_DOUBLE_PRECISION}\}$$

MPIの基本データ型	MPIの定義済み演算	ランク0		ランク1		ランク2		結　果
MPI_LOGICAL	MPI_LAND(論理AND)	.TRUE.	.AND.	.TRUE.	.AND.	.FALSE.	→	.FALSE.
	MPI_LOR (論理OR)	.TRUE.	.OR .	.TRUE.	.OR .	.FALSE.	→	.TRUE.
	MPI_LXOR(論理XOR)	.TRUE.	.XOR.	.TRUE.	.XOR.	.FALSE.	→	.FALSE.
MPI_INTEGER(注1)	MPI_BAND(ビットAND)	00001111	.AND.	00110011	.AND.	01010101	→	00000001
MPI_BYTE	MPI_BOR (ビットOR)	00001111	.OR .	00110011	.OR .	01010101	→	01111111
	MPI_BXOR(ビットXOR)	00001111	.XOR.	00110011	.XOR.	01010101	→	01101001

図 3-3-7 (2)　(注 1) 図 3-3-7 (1) の (注 1) と同じ。

- DO ループの並列化と、MPI_REDUCE (または MPI_ALLREDUCE)、MPI の定義済み演算の関係を整理します。
  図 3-3-6 (2) の 2 つ目のループのように、並列化したループの中に合計や内積を求める計算が含まれている場合、ループを終了した直後に、「各プロセスが求めた合計の総合計」を求めるために、MPI_SUM を使用して、MPI_REDUCE または MPI_ALLREDUCE で通信を行う必要があります。
  また、並列化したループの中に、最大（最小）値を求める計算が含まれている場合、ループを終了した直後に、「各プロセスが求めた最大（最小）値の全体の最大（最小）値」を求めるために、MPI_MAX (MPI_MIN) を使用して、MPI_REDUCE または MPI_ALLREDUCE で通信を行う必要があります。

- MPI_GATHER でも説明しましたが、送信バッファーと受信バッファー（の実際に使用する部分）はメモリー上で重なってはいけませんので注意して下さい。なお、MPI-2 で追加された MPI_IN_PLACE という変数を使用すると、この制限を回避できる場合もあります。詳細は 3-8-1 節を参照して下さい。
  図 3-3-6 (2) では、通信の最終結果は変数 ISUM ではなく受信バッファー ITMP に入ります。これにともない④の ISUM も ITMP に修正する必要があります。しかし、元のプログラムの④以降で変数 ISUM を何度も使用していて ISUM を ITMP に直すのが面倒な場合、あるいは元のプログラムをあまり修正したくない場合には、③のように ITMP を元の ISUM に代入し直し、以後は ISUM をそのまま使用するという方法もあります。
- 実数の合計や内積を求める DO ループを並列化すると、単体版と並列版で結果が若干異なったり、あるいは同じ並列版でもプロセス数によって結果が異なる場合があります。その理由を以下に示します。
  a，b，c が実数のとき、数学の世界では下記の式が成立します。

$$(a+b)+c=a+(b+c)$$

  一方計算機の世界では、取り扱う桁数が有限なので、実数の加算の計算順序を変えると、丸め誤差や桁落ちの影響で計算結果が若干変わることがあり、厳密には下記のようになります。

$$(a+b)+c \neq a+(b+c)$$

例えば配列 A(1)〜A(8) の合計を求めるプログラムを単体版、並列版 (2 プロセス)、並列版 (4 プロセス) で実行した場合、以下の (A)，(B)，(C) のように加算が行われます。[ ] を先に計算するため、それぞれ計算順序は異なっており、その結果、合計が若干変わることがあります。

(A) 単体版：A(1) + A(2) + A(3) + A(4) + A(5) + A(6) + A(7) + A(8)

(B) 並列版 (2 プロセス): $[\underbrace{\texttt{A(1) + A(2) + A(3) + A(4)}}_{\text{ランク 0}}] + [\underbrace{\texttt{A(5) + A(6) + A(7) + A(8)}}_{\text{ランク 1}}]$

(C) 並列版 (4 プロセス): $[\underbrace{\texttt{A(1) + A(2)}}_{\text{ランク 0}}] \underline{+} [\underbrace{\texttt{A(3) + A(4)}}_{\text{ランク 1}}] \underline{+} [\underbrace{\texttt{A(5) + A(6)}}_{\text{ランク 2}}] \underline{+} [\underbrace{\texttt{A(7) + A(8)}}_{\text{ランク 3}}]$

また (C) の波線部に示す各プロセス間の加算を集団通信ルーチン MPI_REDUCE (MPI_SUM を指定) を用いて行なった場合、各プロセスの加算順序はマシン環境によって異なり、以下のいずれかになります。

[1] 各プロセスの加算順序は常に同じ
　この場合、同じプロセス数であれば合計は必ず同じになります。

[2] 各プロセスの加算順序は早いもの勝ち
　この場合、同じプロセス数でもタイミングによって合計が若干変わることがあります。

例えば連立一次方程式の反復解法 (CG 法など) では、答えが収束するまで（計算結果がある条件を満足するまで）計算を何度も繰り返します。この解法で現れる合計や内積の DO ループを並列化すると、上記の理由で、単体版、並列版 (2 プロセス)、並列版 (4 プロセス) で収束回数が異なる現象が発生することがあります。

また上記 [2] のマシン環境の場合は、例えば同じ並列版 (4 プロセス) で実行しても、収束回数が異なる現象が発生することがあります。例えば MPI_ALLREDUCE の場合であれば、各プロセスの合計を MPI_ALLGATHER などで各プロセスに集めた後、各プロセスが例えばランクの小さい方から順番に加算するようにすれば、回避することができます。ただし恐らく MPI_ALLREDUCE より速度は遅くなると思われます。

3-3-6-1　MPI_MAXLOC/MPI_MINLOC

前述の図 3-3-7 (1) の演算 MPI_MAXLOC と MPI_MINLOC の使用方法を説明します。図 3-3-8 (2) の DO ループでは図 3-3-8 (1) の配列 N の最大値 (IMAX = 90) と配列内の位置 (LOC = 4) を求めています。このループを並列化した図 3-3-8 (3) では、まず各プロセスは②で、配列 N のうち自分の担当範囲 (ISTA〜IEND) の最大値と位置を求めます。これを④の MPI_REDUCE で通信しながら演算し、全プロセスでの最大値とその位置を求めたいのですが、前述の演算 MPI_MAX では最大値しか求めることができません。このような場合、①で大きさ 2 の送受信バッファー ISEND を宣言し、③で ISEND の 1 つ目と 2 つ目の要素に最大値 (IMAX) と位置 (ILOC) を代入します。

④の MPI_MAXLOC は、最大値とそれに付随する位置を同時に求める演算で、MPI_2INTEGER は連続した 2 個の整数を意味するデータ型です。2 個の整数が 1 つの単位となるので、④の 3 つ目の引数は「1」となります。

④を実行すると、図 3-3-8 (1) に示すように、ランク 0 のプロセスの受信バッファー IRECV の 1 つ目の要素には全プロセスの最大値が、2 つ目の要素にはその最大値の位置が入ります。以下に補足を述べます。

- 異なるプロセスの最大（小）値が同じで位置の値が異なる場合、位置の値は小さい方が取られます。
- MPI_MAXLOC で指定する「位置」の中身は、「位置」の値でなくても（任意の付加情報でも）構いません。
- MPI_MAXLOC では位置の情報は 1 つしか指定できません。2 つ以上指定したい場合は 3-3-6-2 節の【例 2】を参照して下さい。
- 図 3-3-8 (1) では最大値と位置がともに整数だったので、送受信バッファーは大きさ 2 の整数とし、MPI で提供する基本データ型の MPI_2INTEGER を使用しました。最大値が実数で位置が整数の場合、C 言語の MPI では、MPI の基本データ型として構造型 (MPI_FLOAT_INT など) が用意されています (3-7 節参照)。
  一方 Fortran の MPI では構造型が用意されていないので、図 3-3-8 (4) のように、送受信バッファーは実数とし、位置の整数値を実数に変換して通信し、後で再び整数に戻すのが 1 つの方法です。この場合、MPI の基本データ型は、図 3-3-7 (1) に示すように、送受信バッファーが単精度であれば MPI_2REAL、倍精度であれば MPI_2DOUBLE_PRECISION となります。
  ただし単精度実数の場合、整数の方が単精度実数よりも有効桁数が多いため、位置の値が大きい場合は (例えば 66666666)、右の例に示すように、整数値を単精度実数に代入した時に、元の値と違う値になってしまうので注意して下さい。
  別の方法として、構造型の送受信バッファー（最大値が実数、位置が整数）を使用する演算をユーザーが自分で定義する方法を、3-3-6-2 節の【例 1】に示します。

```
INTEGER I,J(整数)
REAL A(単精度実数)
I = 66666666
A = I
J = A
PRINT *,I,A,J
```

（結果）66666666 66666664.0 66666664

図 3-3-8 (1)

図 3-3-8 (4)

```
INTEGER N(9)
  :
IMAX = -999
DO I = 1, 9
  IF (N(I) > IMAX) THEN
    IMAX = N(I)
    ILOC = I
  ENDIF
ENDDO
PRINT *,'MAX=',IMAX,'LOCATION=',ILOC
```

図 3-3-8 (2)

```
INTEGER N(9),ISEND(2),IRECV(2)                ①
  :
IMAX = -999
DO I = ISTA, IEND
  IF (N(I) > IMAX) THEN
    IMAX = N(I)                               ②
    ILOC = I
  ENDIF
ENDDO
ISEND(1) = IMAX                               ③
ISEND(2) = ILOC                               ③
CALL MPI_REDUCE(ISEND,IRECV,1,MPI_2INTEGER,
&     MPI_MAXLOC,0,MPI_COMM_WORLD,IERR)       ④
PRINT *,'MAX=',IRECV(1),'LOCATION=',IRECV(2)
```

図 3-3-8 (3)

3-3-6-2　ユーザーが定義する演算

前述の図 3-3-7 (1) (2) と同様の「通信しながら行う演算」をユーザーが自分で作成することができます。どういうときに作成が必要になるかについては本節の後半で説明します。

以下で「通信しながら行う演算」の作成方法を説明します。以下の説明で、ユーザーが作成する通信しながら行う演算を $\oplus$（例えば加算）で表します。また 3 プロセスで実行する場合を想定し、ランク 0，1，2 の各プロセスが提供する送信データを 0，1，2 で表すことにします。

結合則と交換則

【結合則】

演算 $\oplus$ は結合則を満足する必要があります。すなわち $(0 \oplus 1) \oplus 2 = 0 \oplus (1 \oplus 2)$ である必要があります。ここで $(0 \oplus 1) \oplus 2$ は、まずランク 0 とランク 1 の演算が行われ、次にその中間結果とランク 2 の演算が行われることを意味します。

【交換則】

[演算 $\oplus$ が交換則を満足する場合]

演算 $\oplus$ が、結合則の他に交換則を満足する場合 ($0 \oplus 1 = 1 \oplus 0$ など)、後述する図 3-3-9 の②の 2 つ目の引数に「`.TRUE.`」を指定します。これを指定すると、通信しながら行う演算の順序は以下のいずれかになります。言いかえると、以下の中に正しくない演算順序がある場合、「`.TRUE.`」を指定することはできません。

実行中に、以下のどの順序で演算が行われるかはマシン環境によって異なります。一般にそのマシン環境で高速になるような順序で行われます。

$$0 \oplus (1 \oplus 2) \quad (1 \oplus 2) \oplus 0 \quad 1 \oplus (0 \oplus 2) \quad (0 \oplus 2) \oplus 1 \quad 2 \oplus (0 \oplus 1) \quad (0 \oplus 1) \oplus 2$$
$$0 \oplus (2 \oplus 1) \quad (2 \oplus 1) \oplus 0 \quad 1 \oplus (2 \oplus 0) \quad (2 \oplus 0) \oplus 1 \quad 2 \oplus (1 \oplus 0) \quad (1 \oplus 0) \oplus 2$$

[演算 $\oplus$ が交換則を満足しない場合]

演算 $\oplus$ が、結合則のみ満足し交換則を満足しない場合 ($0 \oplus 1 \neq 1 \oplus 0$ など)、後述する図 3-3-9 の②の 2 つ目の引数に「`.FALSE.`」を指定します。これを指定すると、演算はランクの小さい方から昇順に、以下のいずれかの順序で行われます。例えば 3 つの行列 A，B，C の乗算の場合、$(A \cdot B) \cdot C = A \cdot (B \cdot C)$ で結合則は満足しますが、$A \cdot B \neq B \cdot A$ で交換則は満足しないので、こちらに該当します。

$$0 \oplus (1 \oplus 2) \quad (0 \oplus 1) \oplus 2$$

例（通信しながら整数の加算を行う演算）

例として、「通信しながら整数の加算を行う演算」(図 3-3-7 (1) の `MPI_SUM` に相当) をユーザーが作成する方法を、図 3-3-9 のプログラムで説明します。

- ②で、ユーザーが作成する演算を MPI に登録するサブルーチン『`MPI_OP_CREATE`』（詳細は付録参照）を一度だけコールします。
  - 1 つ目の引数では、通信しながら行う演算のロジックを記述する④のサブルーチン名を指定します。また①で④のサブルーチン名を `EXTERNAL` 文で外部関数として宣言します。
  - 2 つ目の引数では、前述の [演算 $\oplus$ が交換則を満足する場合] は「`.TRUE.`」を [演算 $\oplus$ が交換則を満足しない場合] は「`.FALSE.`」を指定します。
  - 3 つ目の引数では、③の通信で使用する演算の名前 (図 3-3-7 (1) の `MPI_SUM` に相当する任意の名前の整数) を指定します。
- ②を実行すると、演算が MPI に登録され、その ID（識別子）が変数 `ISUM` に戻ります。この変数を③の 5 つ目の引数で指定します。なお、②を含むサブルーチン以外のサブルーチンで③の通信を行う場合は、変数 `ISUM` をそのサブルーチンに引数、`COMMON` 文、`MODULE` 文などで渡す必要があります。
- ③の通信を実行すると、内部的に④がコールされ、引数 `IN`，`INOUT`，`LEN`，`ITYPE`（名前は任意）に値が渡されます（引数の詳細は後述します）。なお C 言語の場合、④の部分の ANSI-C プロトタイプは以下のようになります (詳細は参考文献 [15] [27] の 4.9.4 節参照)。

```
typedef void 関数名(void *invec, void *inoutvec, int *len, MPI_Datatype *datatype);
```

- 以下でサブルーチン SAMPLE の動作を説明します。

【引数が「.TRUE.」の場合】［演算が交換則を満足する場合］

②の 2 つ目の引数が「.TRUE.」の場合、動作は以下のようになります。

– ⑤で引数 IN と INOUT を、③の 4 つ目の引数と同じ型の変数または配列で宣言します。IN と INOUT には、自分のプロセスの送信バッファー (③の ISEND) で供給した値と、他のプロセスから送られた計算の中間結果が渡されますが、どちらがどちらに渡されるか (図 3-3-10 (1) と (2) のどちらになるか) はマシン環境によって異なります。
– 引数 LEN（整数）には、③の 3 つ目の引数で指定した「要素数」(本例では 1) が渡されます。
– 引数 ITYPE（整数）には、③の 4 つ目の引数で指定した「データ型」(例えば MPI_INTEGER) が渡ります。ただし Fortran の場合、この値を使用することはほとんどないと思われます。
– ⑥で IN と INOUT を使用して、このプロセスでの計算を要素数分だけ行い、それを図 3-3-10 (1) (2) に示すように INOUT に保管します。INOUT はこのプロセスまでの中間結果として、次のプロセスに送られます。
– ④のサブルーチン内では MPI の通信ルーチンを使用することはできません。ただしエラーが発生した時に MPI_ABORT を使用することはできます。

【引数が「.FALSE.」の場合】［演算が交換則を満足しない場合］

②の 2 つ目の引数が「.FALSE.」の場合、以下の点が「.TRUE.」の場合と異なります。自プロセスと、中間結果を送ってきた他プロセスのうち、ランクの小さい方のプロセスのデータ (ランクの小さいプロセスが他プロセスならそのプロセスから送られた中間データ、自プロセスなら送信バッファー ISEND のデータ) が IN に入り、ランクの大きいプロセスのデータが INOUT に入ります。例えば 0〜4 の 5 プロセスで実行している場合、ランク 2 のプロセスでは図 3-3-10 (3) (4) のようになります。

演算が交換則を満足していないので、IN と INOUT にどちらのプロセスのデータが入っているかを考慮して⑥の演算を行う必要があります。引数を「.FALSE.」に設定した例を、後述する【例 2】で紹介します。

```
    :
  EXTERNAL SAMPLE                                  ①
  CALL MPI_OP_CREATE(SAMPLE,.TRUE.,ISUM,IERR)      ②
  送信バッファーISENDに送信データをセットする。
  CALL MPI_REDUCE(ISEND,IRECV,1,MPI_INTEGER,       ③
&                 ISUM,0,MPI_COMM_WORLD,IERR)
    :
```

```
  SUBROUTINE SAMPLE(IN,INOUT,LEN,ITYPE)  ④
  INTEGER IN(*),INOUT(*)                 ⑤
  DO I = 1, LEN
    INOUT(I) = IN(I) + INOUT(I)          ⑥
  ENDDO
  END
```

図 3-3-9

【引数が「.TRUE.」の場合】（以下のいずれか）　【引数が「.FALSE.」の場合】

図 3-3-10 (1)　図 3-3-10 (3)

図 3-3-10 (2)　図 3-3-10 (4)

多くの場合、MPI の定義済み演算だけで事が足りますが、ユーザーが「通信しながら行う演算」を作成しなければならない例をいくつか紹介します。

【例 1】最大（小）値が実数、位置が整数

3-3-6-1 節で説明したように、Fortran の場合、最大（小）値が実数で位置が整数という送受信データを、MPI_REDUCE などで MPI_MAXLOC などを使用して通信することができません。このデータが通信できるように、MPI_MAXLOC などに相当する演算をユーザーが作成する方法を図 3-3-11 (2) のプログラムで説明します。

- 図 3-3-11 (1) に示すような、単精度実数 A と整数 I から構成される構造体 KOUZOUTAI を図 3-3-11 (2) の①で定義し、この構造体型の送信バッファー X と受信バッファー XX を④で宣言します。
- ⑨のサブルーチン PARA_MAXLOC で、通信しながら行う MPI_MAXLOC に相当する演算を定義します。⑩で最大値が大きい方のプロセスの最大値と位置の値を、⑪で XINOUT に入れて次のプロセスに渡します。
- 図 3-3-11 (1) のように異なるプロセスの最大値 (60.) が同じで位置の値 (9と6) が異なる場合、MPI_MAXLOC と同様に、位置の値は小さい方 (6) を取るようにします。このため、⑫で異なるプロセスの最大値が同じ場合、⑬で位置の値は小さい方を取るようにします。
- ⑤でサブルーチン PARA_MAXLOC を、通信しながら行う演算 IMAXLOC として MPI に登録します。
- ③と⑥で、この構造体を表す派生データ型 (3-5-4 節参照) IKOUZOU を作成します。IKOUZOU は⑧の通信で使用します。
- ⑦で、構造体の送信バッファー X に、通信したい最大値（単精度実数）と位置（整数）を代入します。
- ⑧で、新たに作成した演算 IMAXLOC を使用して MPI_REDUCE の通信を行います。すると図 3-3-11 (1) に示すように、全プロセスの実数の最大値 (本例では 60.0) とそれに付随する整数の位置 (本例では9と6のうち小さい方の6) がランク 0 の受信バッファー XX に入ります。

図 3-3-11 (1)

```
  MODULE PARA
  TYPE::KOUZOUTAI
    REAL    A
    INTEGER I                                  ①
  END TYPE KOUZOUTAI
  END
  PROGRAM MAIN
  USE PARA
  INCLUDE 'mpif.h'
  EXTERNAL PARA_MAXLOC                         ②
  INTEGER IBLOCK(2),IDISP(2),ITYPE(2)          ③
  TYPE(KOUZOUTAI) X,XX                         ④
  CALL MPI_INIT(IERR)
  CALL MPI_COMM_SIZE
&    (MPI_COMM_WORLD,NPROCS,IERR)
  CALL MPI_COMM_RANK
&    (MPI_COMM_WORLD,MYRANK,IERR)
  CALL MPI_OP_CREATE
&    (PARA_MAXLOC,.TRUE.,IMAXLOC,IERR)           ⑤
```

```
  IBLOCK(1) = 1
  IBLOCK(2) = 1
  IDISP(1) = 0
  IDISP(2) = 4
  ITYPE(1) = MPI_REAL                          ⑥
  ITYPE(2) = MPI_INTEGER
  CALL MPI_TYPE_STRUCT
&    (2,IBLOCK,IDISP,ITYPE,IKOUZOU,IERR)
  CALL MPI_TYPE_COMMIT(IKOUZOU,IERR)
    :
  X%A = ～  最大値(単精度実数)                   ⑦
  X%I = ～  位置(整数)                          ⑦
  CALL MPI_REDUCE(X,XX,1,IKOUZOU,IMAXLOC,
&                 0,MPI_COMM_WORLD,IERR)         ⑧
    :
  SUBROUTINE PARA_MAXLOC(XIN,XINOUT,LEN,ITYPE)
  USE PARA                                     ↳⑨
  TYPE(KOUZOUTAI) XIN(*),XINOUT(*)
  DO I=1,LEN
    IF (XINOUT(I)%A < XIN(I)%A) THEN            ⑩
      XINOUT(I)%A = XIN(I)%A                    ⑪
      XINOUT(I)%I = XIN(I)%I                    ⑪
    ELSEIF (XINOUT(I)%A == XIN(I)%A) THEN       ⑫
      XINOUT(I)%I = MIN(XINOUT(I)%I,XIN(I)%I)
    ENDIF                                      ↳⑬
  ENDDO
  END
```

図 3-3-11 (2)

【例 2】最大値（最小値）と 2 つの位置

図 3-3-12 (1) に示す 2 次元配列 Z の最大値 (本例では 60.0) と、その 1 次元目と 2 次元目の位置を求めるループを並列に実行した場合、ループ終了後に、各プロセスが求めた最大値の最大値とその 2 つの位置を求めるため、MPI_REDUCE などの通信を行います。ところが 3-3-6-1 節で説明したように、MPI_MAXLOC では、最大値に付随する位置は 1 つしか指定できません。2 つ以上指定したい場合は、MPI_MAXLOC に相当する演算をユーザーが定義する必要があります。プログラム例を図 3-3-12 (3) に示しますが、図 3-3-11 (2) とほとんど同じなので、異なる部分のみ説明します。

- ①の構造体に J を追加し、構造体の派生データ型を作成する③の部分を 2 とします (3-5-4 節参照)。
- ④で J 方向の位置を設定します。
- 本例のように、異なるプロセスの最大値 (60.) が同じで位置の値が異なる場合、図 3-3-11 (2) では MPI_MAXLOC と同様に位置の値は小さい方を取りましたが、本例では位置が 2 つあってこの方法は適用できないため、ランクの小さいプロセスの位置の値を取ることにします。まず②の 2 つ目の引数を「.FALSE.」とします。これによって、本節の前半で説明したように、⑤でランクの小さい方の位置が配列 XIN に、ランクの大きい方の位置が配列 XINOUT に入ります。⑥で 2 つのプロセスの最大値が等しいときは、⑦でランクの小さいプロセスの位置を採用するので、上記下線部が実現されます。

J→	ランク0			ランク1			ランク2		
I ↓ Z	1	2	3	4	5	6	7	8	9
1	10.	20.	10.	30.	20.	40.	10.	20.	60.
2	10.	10.	20.	10.	20.	10.	40.	20.	30.
3	20.	30.	10.	20.	10.	30.	30.	10.	20.
4	20.	20.	20.	30.	60.	40.	20.	40.	10.
5	10.	10.	10.	40.	30.	10.	30.	20.	10.

図 3-3-12 (1)

図 3-3-12 (2)

```
MODULE PARA
TYPE::KOUZOUTAI
  REAL    A   最大値(単精度実数)
  INTEGER I   I方向の位置(整数)
  INTEGER J   J方向の位置(整数)                ①
END TYPE KOUZOUTAI
END
```

```
PROGRAM MAIN
USE PARA
INCLUDE 'mpif.h'
EXTERNAL PARA_MAXLOC2
INTEGER IBLOCK(2),IDISP(2),ITYPE(2)
TYPE(KOUZOUTAI) X,XX
REAL Z(5,9)
CALL MPI_INIT(IERR)
CALL MPI_COMM_SIZE
&    (MPI_COMM_WORLD,NPROCS,IERR)
CALL MPI_COMM_RANK
&    (MPI_COMM_WORLD,MYRANK,IERR)
CALL MPI_OP_CREATE
&    (PARA_MAXLOC2,.FALSE.,IMAXLOC2,IERR)    ②
IBLOCK(1) = 1
IBLOCK(2) = 2                                ③
IDISP(1) = 0
IDISP(2) = 4
ITYPE(1) = MPI_REAL
ITYPE(2) = MPI_INTEGER
CALL MPI_TYPE_STRUCT
&    (2,IBLOCK,IDISP,ITYPE,IKOUZOU,IERR)
CALL MPI_TYPE_COMMIT(IKOUZOU,IERR)
  :
X%A = ～  最大値(単精度実数)
X%I = ～  I方向の位置(整数)
X%J = ～  J方向の位置(整数)                  ④
CALL MPI_REDUCE(X,XX,1,IKOUZOU,IMAXLOC2,
&                0,MPI_COMM_WORLD,IERR)
  :
```

```
SUBROUTINE PARA_MAXLOC2(XIN,XINOUT,LEN,ITYPE)
USE PARA                                     ↳⑤
TYPE(KOUZOUTAI) XIN(*),XINOUT(*)
DO I=1,LEN
  IF (XINOUT(I)%A <= XIN(I)%A) THEN          ⑥
    XINOUT(I)%A = XIN(I)%A
    XINOUT(I)%I = XIN(I)%I                   ⑦
    XINOUT(I)%J = XIN(I)%J
  ENDIF
ENDDO
END
```

図 3-3-12 (3)

【例 3】倍精度複素数の加算

前述の図 3-3-7 (1) に示すように、MPI の仕様書 (参考文献 [15], [27]) には、MPI_SUM、MPI_PROD の演算に倍精度複素数 (MPI_COMPLEX16 または MPI_DOUBLE_COMPLEX) が含まれていません。実際には多くのマシン環境では使うことができますが、もし使うことができない場合、ユーザーが「通信しながら倍精度複素数の加算を行う演算」を定義する必要があります。データの動きとプログラム例を図 3-3-13 (1) (2) に示します。

図 3-3-13 (1)

```
  PROGRAM MAIN
  INCLUDE 'mpif.h'
  EXTERNAL PARA_SUMC16
  COMPLEX*16 C,CC
  CALL MPI_INIT(IERR)
  CALL MPI_COMM_SIZE
&    (MPI_COMM_WORLD,NPROCS,IERR)
  CALL MPI_COMM_RANK
&    (MPI_COMM_WORLD,MYRANK,IERR)
```

```
  CALL MPI_OP_CREATE
&    (PARA_SUMC16,.TRUE.,ISUMC16,IERR)
  C = 複素数の値を設定
  CALL MPI_REDUCE(C,CC,1,MPI_DOUBLE_COMPLEX,
&          ISUMC16,0,MPI_COMM_WORLD,IERR)
    :
  SUBROUTINE PARA_SUMC16(CIN,CINOUT,LEN,ITYPE)
  COMPLEX*16 CIN(*),CINOUT(*)
  DO I = 1, LEN
    CINOUT(I) = CIN(I) + CINOUT(I)
  ENDDO
  END
```

図 3-3-13 (2)

【例 4】合計と最大値を 1 回の通信を行う

合計と最大値の両方を求めるループを並列化した場合、ループ終了後に、各プロセスが求めた合計の合計と最大値の最大値を求めるため、MPI_REDUCE を 2 回コールする必要があります。ところが通信を 1 回行うたびに立ち上がり時間がかかります (4-1-2 節参照)。ユーザーが「合計の合計と最大値の最大値を同時に求める演算」を定義すれば、通信回数は 1 回となり、速度が速くなる可能性があります（ただし効果があるかどうかはマシン環境に依存します)。データの動きとプログラム例を図 3-3-14 (1) (2) に示します。

図 3-3-14 (1)

```
  PROGRAM MAIN
  INCLUDE 'mpif.h'
  EXTERNAL PARA_SUMMAX
  REAL X(2),XX(2)
  CALL MPI_INIT(IERR)
  CALL MPI_COMM_SIZE
&    (MPI_COMM_WORLD,NPROCS,IERR)
  CALL MPI_COMM_RANK
&    (MPI_COMM_WORLD,MYRANK,IERR)
```

```
  CALL MPI_OP_CREATE
&    (PARA_SUMMAX,.TRUE.,ISUMMAX,IERR)
  X(1) = 合計を設定
  X(2) = 最大値を設定
  CALL MPI_REDUCE(X,XX,1,MPI_2REAL,ISUMMAX,
&                0,MPI_COMM_WORLD,IERR)
    :
  SUBROUTINE PARA_SUMMAX(XIN,XINOUT,LEN,ITYPE)
  REAL XIN(2,*),XINOUT(2,*)
  DO I = 1, LEN
    XINOUT(1,I) =     XIN(1,I) + XINOUT(1,I)
    XINOUT(2,I) = MAX(XIN(2,I) , XINOUT(2,I))
  ENDDO
  END
```

図 3-3-14 (2)

3-4　1 対 1 通信

1 対 1 通信サブルーチンは、2 つのプロセス間での通信を行います。1 対 1 通信サブルーチンは、本当に 2 つのプロセス間のみで通信を行なうために使用する場合の他に、集団通信サブルーチンの機能を補う目的で、集団通信サブルーチンの代用として使用する場合があります (このときは全プロセスが 1 対 1 通信を行います)。

なお、マシン環境によっては本節の説明と動作が異なる場合もあります。

3-4-1　1 対 1 通信の全体の流れ

まず 1 対 1 通信の動作の概要を図 3-4-1 に示します。

この例ではランク 0 のプロセスからランク 1 のプロセスにメッセージを送信しています。送信側アプリケーションは MPI_SEND などの送信サブルーチンをコールし、受信側アプリケーションは MPI_RECV などの受信サブルーチンをコールします。ただし SPMD モデルのプログラムでは、プログラム自体は 1 つなので、1 つのプログラム内に送信と受信の両方の動作を記述します。

プログラムで使用する変数または配列である送信バッファー、受信バッファーとは別に、MPI の通信サブルーチンは内部的にシステムバッファーを使用します。システムバッファーもメモリー上に作成されますが、アプリケーションプログラムから見ることはできません。

送信側アプリケーションプログラムは、まず①で送信バッファーにデータ（メッセージ）をセットし、②で MPI の送信サブルーチン (例えば MPI_SEND) をコールします。MPI_SEND は③でメッセージをいったんシステムバッファーにコピーし、④で宛先プロセスに送信します。

受信側アプリケーションプログラムは、まず1で MPI の受信サブルーチン (例えば MPI_RECV) をコールします。MPI_RECV は2で送信元プロセスからのメッセージをいったんシステムバッファーに受信し、3で受信バッファーにコピーします。最後に4で、受信側アプリケーションプログラムは受信バッファーを参照します。

図 3-4-1

3-4-2　ブロッキング通信と非ブロッキング通信

1 対 1 通信サブルーチンの分類

図 3-4-1 で説明したように、1 対 1 通信サブルーチンには送信サブルーチンと受信サブルーチンがあります。これとは別に、1 対 1 通信サブルーチンは、ブロッキング通信と非ブロッキング通信の 2 種類に分類することができます。ここでいうブロックとは、塊という意味ではなく、野球でキャッチャーが三塁からの走者をブロックする、すなわち阻止するという意味です。

以上を合わせると、MPI で提供されている主要な 1 対 1 通信サブルーチンは右のように分類されます。

実は、MPI ではこれ以外にも多くの 1 対 1 通信サブルーチンが提供されています。しかし、よほど特殊な用途以外では、右のサブルーチンだけで十分だと思われますので、本書では説明を省略します。

	送信	受信
ブロッキング通信	MPI_SEND	MPI_RECV
	MPI_SENDRECV	
非ブロッキング通信	MPI_ISEND ＋MPI_WAIT	MPI_IRECV ＋MPI_WAIT

ブロッキング送信 (MPI_SEND)

まず、ブロッキング送信サブルーチン MPI_SEND について説明します。ここで、付録の MPI_SEND の説明を一読して下さい…。

付録と同じ図を図 3-4-2 に示します。④でアプリケーションプログラムは『待ち』の状態に入りますが、もし『待ち』に入らないとどうなるか考えてみましょう。

通常、③の MPI_SEND が一度だけコールされることはあまりなく、以下のように MPI_SEND が何度もコールされます。もし④で『待ち』の状態に入らないとすると、アプリケーションプログラムはどんどん進み、ループが一周して再び②を実行します。ところがこの時点で、MPI_SEND が前回の1を完了している保証はありません。もし完了していないうちに 2 回目の②を実行すると、送信バッファー内にある前回の（仕掛かり中の）データを壊してしまい、誤ったメッセージが送信されてしまいます。

ブロッキング送信サブルーチン MPI_SEND では、このような問題が発生しないように、③を実行した後、④でアプリケーションプログラムを『待ち』の状態にし、先に進むのをブロック（阻止）します。そして、1で全てのメッセージを送信バッファー buf からシステムバッファーにコピーしたら、アプリケーションプログラムが送信バッファー buf を再利用することが可能になるので、2でアプリケーションプログラムの『待ち』を解除します。

なお、付録にも書きましたが、『待ち』が解除されたということは、送信したメッセージが宛先プロセスに到着したことを意味するわけではありません。宛先プロセスがまだ MPI_RECV (または MPI_IRECV) をコールしていない場合すらあります。次のページで説明する MPI_ISEND の場合も同様です。

図 3-4-2

非ブロッキング送信 (MPI_ISEND)

次に、非ブロッキング送信サブルーチン MPI_ISEND と MPI_WAIT を説明します。付録の MPI_ISEND と MPI_WAIT の説明を一読して下さい…。

付録と同じ図を図 3-4-3 に示します。アプリケーションプログラムは③でサブルーチン MPI_ISEND に対してメッセージを『送れ！』と指示した後、④に示すようにどんどん先に進みます。しかし、これではループが一周して再び②を実行したときに、MPI_SEND の説明で述べた『送信バッファー内にある前回の（仕掛かり中の）データを壊してしまう』という問題が発生してしまいます。

これを防ぐため、2 回目の②を実行するよりも前のどこかで、⑤に示すようにサブルーチン MPI_WAIT をコールします。するとアプリケーションプログラムは⑥で『待ち』に入り、MPI_ISEND が2を実行するまで『待ち』の状態が続きます。つまりサブルーチン MPI_WAIT はストッパーの役割を果たします。

もう少し正確に述べると、③をコールすると、『送れ！』と指示したメッセージに対して識別子 (ID) が MPI_ISEND によって付けられ、その値が③の引数 request に戻ります。この値を⑤の最初の引数で指定すると、そのメッセージに対して2が実行されるまで⑥で『待ち』の状態になります。つまり引数 request は③と⑤を対応付ける役割を持ちます。

付録にも書きましたが、MPI_ISEND を指定して MPI_WAIT を指定し忘れた場合、タイミングによる（再現性のない）エラーが発生します。また MPI_ISEND と MPI_WAIT の引数 request がスペルミスで異なっている場合も、当然ながら MPI_WAIT は働かず、タイミングによる（再現性のない）エラーが発生しますので、スペルミスには十分注意してください（一方のスペルをカット・アンド・ペーストで他方にコピーするのが安全です）。

MPI_ISEND をコールした直後に MPI_WAIT をコールした場合、MPI_SEND をコールしたのと機能的には同じことになります。それでは一体何のために MPI_ISEND があるのでしょうか？　以下の 2 つの理由がありますが、本書では (1) の理由で使用します。

(1) MPI_SEND を使用した場合、書き方が悪いと、デッドロックという現象が発生してプログラムが止まってしまう可能性があります。これに対し『MPI_ISEND＋MPI_WAIT』という 2 つの機能に分かれていると、デッドロックを起こさないプログラムを書くことができます (3-4-4 節参照)。

(2) 計算を行う CPU と独立に動作する通信専用の CPU が付いているマシン環境の場合、図 3-4-3 の⑤を④のずっと先 (ただし 2 回目の②よりは前) に置くと、③の通信と④の計算がオーバーラップして実行し、通信時間が隠蔽されるため、通信のオーバーヘッドを減らす事ができる場合があります。

図 3-4-3

ブロッキング受信 (MPI_RECV)

次に、ブロッキング受信サブルーチン MPI_RECV を説明します。付録の MPI_RECV の説明を一読して下さい…。

付録と同じ図を図 3-4-4 に示します。③でアプリケーションプログラムは『待ち』の状態に入りますが、もし『待ち』に入らないとどうなるか考えてみましょう。

メッセージを受信するということは、受信した後にメッセージを参照するという目的があるはずです。以下の図では⑥がそれに該当します。もし③で『待ち』の状態に入らないとすると、アプリケーションプログラムはどんどん進み、⑥で受信バッファー buf を参照します。ところがこの時点で、MPI_RECV が2を完了している保証はありません。もし完了していないうちに⑥を実行すると、誤ったメッセージを参照してしまいます。

ブロッキング受信サブルーチン MPI_RECV では、このような問題が発生しないように、②を実行した後、③でアプリケーションプログラムを『待ち』の状態にし、先に進むのをブロック（阻止）します。そして、2で全てのメッセージをシステムバッファーから受信バッファー buf にコピーしたら、アプリケーションプログラムが受信バッファー buf を参照することが可能になるので、3でアプリケーションプログラムの『待ち』を解除します。

図 3-4-4

非ブロッキング受信 (MPI_IRECV)

最後に、非ブロッキング受信サブルーチン MPI_IRECV と MPI_WAIT を説明します。付録の MPI_IRECV と MPI_WAIT の説明を一読して下さい…。

付録と同じ図を図 3-4-5 に示します。アプリケーションプログラムは②でサブルーチン MPI_IRECV に対してメッセージを『受け取れ！』と指示した後、③に示すようにどんどん先に進みます。しかし、これでは⑧を実行したときに、MPI_RECV の説明で述べた『受信バッファー buf を参照したときに正しいメッセージが入っていない』という問題が発生してしまいます。

これを防ぐため、⑧を実行するよりも前のどこかで、④に示すようにサブルーチン MPI_WAIT をコールします。するとアプリケーションプログラムは⑤で『待ち』に入り、MPI_IRECV が3を実行するまで『待ち』の状態が続きます。つまりサブルーチン MPI_WAIT はストッパーの役割を果たします。

もう少し正確に述べると、②をコールすると、『受け取れ！』と指示したメッセージに対して識別子 (ID) が MPI_IRECV によって付けられ、その値が②の引数 request に戻ります。この値を④の最初の引数で指定すると、そのメッセージに対して3が実行されるまで⑤で『待ち』の状態になります。つまり引数 request は②と④を対応付ける役割を持ちます。

付録にも書きましたが、MPI_IRECV を指定して MPI_WAIT を指定し忘れた場合、タイミングによる（再現性のない）エラーが発生します。また MPI_IRECV と MPI_WAIT の引数 request がスペルミスで異なっている場合も、当然ながら MPI_WAIT は働かず、タイミングによる（再現性のない）エラーが発生しますので、スペルミスには十分注意して下さい（一方のスペルをカット・アンド・ペーストで他方にコピーするのが安全です）。

MPI_IRECV をコールした直後に MPI_WAIT をコールした場合、MPI_RECV をコールしたのと機能的には同じことになります。MPI_RECV でなく『MPI_IRECV + MPI_WAIT』を使用する理由は、MPI_ISEND で説明したのと同様に、2 つの機能に分かれていることによってデッドロックを起こさないプログラムを書くことができるため (3-4-4 節参照) と、計算と通信をオーバーラップさせて通信時間を隠蔽し、通信のオーバーヘッドを減少させるため (ただし計算を行う CPU と独立に動作する通信専用の CPU が付いているマシン環境の場合) です。

図 3-4-5

3-4-3　1 方向のみの通信

本節からは、1 対 1 通信サブルーチンを使用して実際に通信を行う際の考慮点について説明します。まず図 3-4-6 (1) のように、ランク 0 のプロセスの送信バッファー ISEND からランク 1 のプロセスの受信バッファー IRECV に（一方向に）送信する場合について説明します。

図 3-4-6 (1)

SPMD モデルではプログラムは 1 つなので、1 つのプログラム内に、送信と受信の両方の動作を記述します。

MPI_SEND で送信しても MPI_ISEND で送信しても、同じメッセージが宛先プロセスに届きます。同じメッセージなので、MPI_RECV でも MPI_IRECV でも受信することができます。

従って、MPI_SEND と MPI_ISEND のどちらで送信した場合でも、MPI_RECV と MPI_IRECV のどちらで受信しても構いません。すなわち送受信には、図 3-4-6 (2) に示すように全部で 4 通りの組み合わせがあります。

	受信がMPI_RECV	受信がMPI_IRECV

送信がMPI_SEND	:
	IF (MYRANK == 0) THEN
	CALL MPI_SEND(ISEND,～)
	ELSEIF (MYRANK == 1) THEN
	CALL MPI_RECV(IRECV,～)
	ENDIF
	:
		:
		IF (MYRANK == 0) THEN
		CALL MPI_SEND(ISEND,～)
		ELSEIF (MYRANK == 1) THEN
		CALL MPI_IRECV(IRECV,～,IREQ,～)
		CALL MPI_WAIT(IREQ,～)
		ENDIF
		:

送信がMPI_ISEND	:
	IF (MYRANK == 0) THEN
	CALL MPI_ISEND(ISEND,～,IREQ,～)
	CALL MPI_WAIT(IREQ,～)
	ELSEIF (MYRANK == 1) THEN
	CALL MPI_RECV(IRECV,～)
	ENDIF
	:
		:
		IF (MYRANK == 0) THEN
		CALL MPI_ISEND(ISEND,～,IREQ,～)
		ELSEIF (MYRANK == 1) THEN
		CALL MPI_IRECV(IRECV,～,IREQ,～)
		ENDIF
		CALL MPI_WAIT(IREQ,～)
		:

図 3-4-6 (2)

3-4-4　双方向通信

双方向通信とデッドロック

次に、図 3-4-7 に示すように、ランク 0 のプロセスからランク 1 のプロセスに送信し、ランク 1 からランク 0 にも送信するという、双方向の通信について説明します。ここでは双方向とは、たまたまプログラム内のほぼ同じ場所で、ランク 0 からランク 1 へのメッセージの送信と、ランク 1 からランク 0 へのメッセージの送信が必要になったという意味であると考えて下さい。

図 3-4-8 に示すように、2 人が相手の家に引っ越すとします。その時 2 人とも、相手の家が空いたら荷物を運ぼうと考えたとすると、2 人とも動けなくなってしまいます。このように、お互いに相手の動作待ちになり、こう着状態に陥ることをデッドロックと呼びます。

図 3-4-7

図 3-4-8

双方向に通信を行う場合、通信順序がまずいとデッドロックが発生し、上記の例と同様に、各プロセスは相手プロセスの動作待ちになり、両方のプロセスが動けなくなってしまいます。デッドロックが発生すると、プログラムはその箇所で永久に止まったままになるため、並列ジョブをキャンセルするしかありません。これは、前述の再現性のないエラーと並んで、並列ジョブに特有の現象です。

図 3-4-7 のように双方向に通信を行う場合、各プロセスは相手のプロセスに対して送信と受信の両方を行います。各プロセスの送信と受信の実行順序によって、図 3-4-9 に示す 3 つのパターンが考えられます。

ブロッキングと非ブロッキングのどちらで通信するかまで考慮すると、さらにバリエーションが増えます。なお、図中の「MPI_(I)SEND」は、MPI_SEND と MPI_ISEND のいずれかであることを示します。また、この図では MPI_WAIT は省略しました。

このように、双方向通信には多くのバリエーションが考えられます。それでは、図 3-4-9 のそれぞれのパターンについてデッドロックの可能性を検討してみましょう。

パターン1

```
    :
IF (MYRANK == 0) THEN
  CALL MPI_(I)SEND(～)
  CALL MPI_(I)RECV(～)
ELSEIF (MYRANK == 1) THEN
  CALL MPI_(I)SEND(～)
  CALL MPI_(I)RECV(～)
ENDIF
    :
```

ランク0、1のプロセスがともに送信、受信の順にコール。

パターン2

```
    :
IF (MYRANK == 0) THEN
  CALL MPI_(I)RECV(～)
  CALL MPI_(I)SEND(～)
ELSEIF (MYRANK == 1) THEN
  CALL MPI_(I)RECV(～)
  CALL MPI_(I)SEND(～)
ENDIF
    :
```

ランク0、1のプロセスがともに受信、送信の順にコール。

パターン3

```
    :
IF (MYRANK == 0) THEN
  CALL MPI_(I)SEND(～)
  CALL MPI_(I)RECV(～)
ELSEIF (MYRANK == 1) THEN
  CALL MPI_(I)RECV(～)
  CALL MPI_(I)SEND(～)
ENDIF
    :
```

ランク0のプロセスは送信、受信、ランク1のプロセスは受信、送信の順にコール
(ランク0と1は逆でも可)。

図 3-4-9

パターン1

最初に結論を述べます。図3-4-10 (1) のように、両プロセスがともにブロッキングの送信MPI_SEND、受信の順に通信を行った場合、デッドロックが発生する可能性があります。図3-4-10 (2) の②のように非ブロッキング送信MPI_ISENDの直後にMPI_WAITをコールした場合も、①と機能的に同じなのでデッドロックが発生します。図3-4-10 (3) のように、③に対するMPI_WAITを④の後ろの⑤に置いた場合は、デッドロックにはなりません（理由は後述します）。

```
    :
IF (MYRANK == 0) THEN
  CALL MPI_SEND(～)                ①
  CALL MPI_(I)RECV(～)
ELSEIF (MYRANK == 1) THEN
  CALL MPI_SEND(～)
  CALL MPI_(I)RECV(～)
ENDIF
    :
```

図3-4-10 (1)　×

```
    :
IF (MYRANK == 0) THEN
  CALL MPI_ISEND(～,IREQ)          ②
  CALL MPI_WAIT(IREQ,～)
  CALL MPI_(I)RECV(～)
ELSEIF (MYRANK == 1) THEN
  CALL MPI_ISEND(～,IREQ)
  CALL MPI_WAIT(IREQ,～)
  CALL MPI_(I)RECV(～)
ENDIF
    :
```

図3-4-10 (2)　×

```
    :
IF (MYRANK == 0) THEN
  CALL MPI_ISEND(～,IREQ)          ③
  CALL MPI_(I)RECV(～)             ④
  CALL MPI_WAIT(IREQ,～)           ⑤
ELSEIF (MYRANK == 1) THEN
  CALL MPI_ISEND(～,IREQ)
  CALL MPI_(I)RECV(～)
  CALL MPI_WAIT(IREQ,～)
ENDIF
    :
```

図3-4-10 (3)　○

デッドロックが発生する理由を説明します（マシン環境によってはメカニズムが異なるため、発生しない場合もあります）。図3-4-11で、送信メッセージの長さ「4」が、システムバッファーの大きさ「2」より長いとします。この場合、ランク0のMPI_SENDは、まず■をランク1に送信し、次に残りの□を送信します。

ランク0のプログラムは①をコールすると、②で待ちに入ります。コールされたMPI_SENDは、(1) でまず■をシステムバッファーにコピーします。(2) の時点でランク1のプログラムが④のMPI_(I)RECVをすでにコールしていれば、(3) で■を送信できますが、まだコールしていない場合はコールするまで (2) で待ちます。

一方ランク1のプログラムとMPI_SENDもランク0と同じ状態になっていて、それぞれ②と [2] で止まっています。このため④のMPI_(I)RECVがコールされず、前述のランク0のMPI_SENDは (2) で止まったまま先に進めません。すなわちランク0, 1のプログラムとMPI_SENDは、全員下線部分で止まり、先に進めなくなります。

このデッドロックは、④と④のMPI_(I)RECVを動かすことができれば回避することができます。MPI_SENDはMPI_ISENDとMPI_WAITという2つのルーチンに分けることができます。そこで図3-4-10 (3) のようにすれば、MPI_ISENDの他にMPI_(I)RECVも動くので、デッドロックを回避することができます。

なおこのデッドロックは、送信メッセージがシステムバッファーより短い場合は発生しません。しかしそのようなプログラムは「誤り」ですから作成しないようにして下さい。例えば、並列化したプログラムがテスト段階では小規模なデータを使用したため、たまたまデッドロックにならず、本番で大規模なデータを使用したらデッドロックになったり、あるいはマシン環境を変えたらデッドロックになる恐れがあります。

図3-4-11

パターン 2

最初に結論を述べます。図 3-4-12 (1) のように、両プロセスがともにブロッキングの受信 MPI_RECV、送信の順に通信を行った場合、必ずデッドロックが発生します。図 3-4-12 (2) の②のように非ブロッキング受信 MPI_IRECV の直後に MPI_WAIT をコールした場合も、①と機能的に同じなのでデッドロックが発生します。図 3-4-12 (3) のように、③に対する MPI_WAIT を④の後ろの⑤に置いた場合は、デッドロックにはなりません（理由は後述します）。

```
  :
IF (MYRANK == 0) THEN
  CALL MPI_RECV(～)                ①
  CALL MPI_(I)SEND(～)
ELSEIF (MYRANK == 1) THEN
  CALL MPI_RECV(～)
  CALL MPI_(I)SEND(～)
ENDIF
  :
```

図 3-4-12 (1)　×

```
  :
IF (MYRANK == 0) THEN
  CALL MPI_IRECV(～,IREQ)          ②
  CALL MPI_WAIT(IREQ,～)
  CALL MPI_(I)SEND(～)
ELSEIF (MYRANK == 1) THEN
  CALL MPI_IRECV(～,IREQ)
  CALL MPI_WAIT(IREQ,～)
  CALL MPI_(I)SEND(～)
ENDIF
  :
```

図 3-4-12 (2)　×

```
  :
IF (MYRANK == 0) THEN
  CALL MPI_IRECV(～,IREQ)          ③
  CALL MPI_(I)SEND(～)             ④
  CALL MPI_WAIT(IREQ,～)           ⑤
ELSEIF (MYRANK == 1) THEN
  CALL MPI_IRECV(～,IREQ)
  CALL MPI_(I)SEND(～)
  CALL MPI_WAIT(IREQ,～)
ENDIF
  :
```

図 3-4-12 (3)　○

このデッドロックは、前述のパターン 1 のデッドロックに比べると、発生のメカニズムは簡単です。図 3-4-13 に示すように、ランク 0 のプログラムは①をコールすると、②で待ちに入ります。コールされた MPI_RECV は、(1) の時点でランク 1 のプログラムが④の MPI_(I)SEND をすでにコールしていれば、(2) で●を受信できますが、まだコールしていない場合はコールするまで (1) で待ちます。

一方ランク 1 のプログラムと MPI_RECV もランク 0 と同じ状態になっていて、それぞれ②と [1] で止まっています。このため④の MPI_(I)SEND がコールされず、前述のランク 0 の MPI_RECV は (1) で止まったまま先に進めません。すなわちランク 0, 1 のプログラムと MPI_RECV は、全員下線部分で止まり、先に進めなくなります。

このデッドロックは、④と④の MPI_(I)SEND を動かすことができれば回避することができます。MPI_RECV は MPI_IRECV と MPI_WAIT という 2 つのルーチンに分けることができます。そこで図 3-4-12 (3) のようにすれば、MPI_IRECV の他に MPI_(I)SEND も動くので、デッドロックを回避することができます。

パターン 1 と違い、このデッドロックは送信メッセージが短くても必ず発生します。実際にこのデッドロックが発生するプログラム例を、4-6-2 節の「境界条件の処理」に示します。

図 3-4-13

パターン 3

前述の図 3-4-7 の双方向通信は、図 3-4-6 (1) の一方向通信を、向きを変えて 2 回行ったとみなすことができ、プログラムで書くと図 3-4-14 (1) のようになります (図で MPI_WAIT は省略しています)。まず①でランク 0 のプロセスからランク 1 のプロセスに一方向通信を行い、次に②でランク 1 のプロセスからランク 0 のプロセスに一方向通信を行ないます。

①と②の IF 文を整理すると図 3-4-14 (2) になります。これをよく見ると、図 3-4-9 のパターン 3 と同じです。そして図 3-4-6 (2) で述べたように、一方向通信はブロッキングと非ブロッキングの送受信のいずれの組み合わせでも可能なので、図 3-4-14 (1) (2) もいずれの組み合わせでも可能です。

まとめると、パターン 3 は、ブロッキングと非ブロッキングのいずれの通信ルーチンの組み合わせでもデッドロックにはなりません。

```
   : ランク0 ➡ ランク1 の一方向通信
IF (MYRANK == 0) THEN
  CALL MPI_(I)SEND(～)
ELSEIF (MYRANK == 1) THEN          ①
  CALL MPI_(I)RECV(～)
ENDIF
   ランク0 ⬅ ランク1 の一方向通信
IF (MYRANK == 1) THEN
  CALL MPI_(I)SEND(～)
ELSEIF (MYRANK == 0) THEN          ②
  CALL MPI_(I)RECV(～)
ENDIF
   :
```

図 3-4-14 (1)　1 方向通信を 2 回行う

→

```
   :
IF (MYRANK == 0) THEN
  CALL MPI_(I)SEND(～)
  CALL MPI_(I)RECV(～)
ELSEIF (MYRANK == 1) THEN
  CALL MPI_(I)RECV(～)
  CALL MPI_(I)SEND(～)
ENDIF
   :
```

図 3-4-14 (2)　パターン 3

全ての通信を非ブロッキング通信にした場合

双方向通信の場合、パターン 1 とパターン 2 で説明したように、送信ルーチンや受信ルーチンが動けなくなるためにデッドロックが発生することが分かりました。従って、図 3-4-15 のように送信と受信ルーチンを全て非ブロッキング通信にし、全ての送受信ルーチンをいっせいに動かすようにすれば、デッドロックになることはありません。

```
   :
IF (MYRANK == 0) THEN
  CALL MPI_ISEND(～,IREQ1,～)
  CALL MPI_IRECV(～,IREQ2,～)
ELSEIF (MYRANK == 1) THEN
  CALL MPI_ISEND(～,IREQ1,～)
  CALL MPI_IRECV(～,IREQ2,～)
ENDIF
CALL MPI_WAIT(IREQ1,～)
CALL MPI_WAIT(IREQ2,～)
   :
```

図 3-4-15　○

1 対 1 ブロッキング通信ルーチン MPI_SENDRECV

1 対 1 通信ルーチン MPI_SENDRECV は送信と受信が合体したルーチンで、図 3-4-16 (1) (2) のような通信を行うことができます (詳細は 4-6-2 節と付録参照)。このルーチンはブロッキング通信ルーチンですが、デッドロックにはなりません。4-6-2 節、5-1 節で説明する、シフトと呼ばれる通信パターンに特に有効です。分類上は 1 対 1 通信ルーチンですが、集団通信ルーチンの 1 つと考えることもできます。

MPI_SEND, MPI_RECV, MPI_ISEND, MPI_IRECV, MPI_SENDRECV のどれが最も速いかはマシン環境に依存します。

図 3-4-16 (1)　　図 3-4-16 (2)

3-4-5　1 対 1 通信に関する補足

いくつか補足します。

- 1 対 1 通信ルーチンでは、受信バッファーで指定するバイト数 (データ個数 × データ型のバイト数) は、送信バッファーで指定するバイト数と等しいか、または大きくても構いません。
- 3-3-4 節でも書きましたが、1 対 1 通信および集団通信ルーチンで一度に送信できる最大のバイト数はマシン環境によって異なり、無制限の場合と、制限のある場合があります。制限のある場合に、制限を越えたバイト数を送ってもエラーにならず、不完全なメッセージが送られてしまうマシン環境もありますので、お使いのマシンのマニュアルを確認して下さい。
- ランク 0 のプロセスからランク 1 のプロセスに 1 対 1 送信 (MPI_SEND など) を 2 回連続して行った場合、メッセージはコールした順番にランク 1 のプロセスに到着します。言いかえると、あるメッセージが途中で前のメッセージを追い越すことはないので、プログラムではメッセージの順序を気にする必要はありません。
- MPI で並列化したプログラムを 1 プロセスで実行した場合、通常は正常に動作します。しかし例えば図 3-4-17 のように、1 対 1 通信ルーチンを含む並列プログラムを 1 プロセスで実行した場合、そのプロセスのランクは 0 なので①が真となり②が実行されます。しかし③は偽となり④が実行されないので、以後の計算がおかしくなってしまいます。このような問題が発生する可能性があるので、1 プロセスで実行するときは、並列版ではなく単体版のプログラムを使用することを強くお勧めします (4-6-2 節の「境界条件の処理」参照)。

```
  :
IF     (MYRANK==0) THEN            ①
  CALL MPI_SEND(A,1,MPI_REAL,1,～) ②
ELSEIF (MYRANK==1) THEN            ③
  CALL MPI_RECV(B,1,MPI_REAL,0,～) ④
ENDIF
  :
```

図 3-4-17

- 以下のような場合、仕掛かり中のメッセージがどんどん累積していきますが、その後の動作はマシン環境によって異なりますので（例えば受信バッファーがパンクして異常終了するなど)、お使いのマシン環境のマニュアルで確認して下さい。
  - 図 3-4-18 (1) のように、ランク 1 からランク 0 に一度に複数のメッセージを送信したものの、ランク 0 は別の処理中のため、まだ受信できない場合。
  - 図 3-4-18 (2) のように、複数のプロセス (本例では 1～3) からランク 0 に一度にメッセージを送信したものの、ランク 0 は別の処理中のため、まだ受信できない場合。

図 3-4-18 (1)

図 3-4-18 (2)

3-5　派生データ型

3-5-1　派生データ型とは

図 3-5-1 (1) で、ランク 0 が持つ配列 A のうち●の部分のみを、ランク 1 が持つ配列 B の●の部分に送信する場合を考えます。なお、図 3-5-1 (1) (2) では、前節で説明したシステムバッファーは省略していますので注意して下さい。

今まで説明した方法では、MPI の通信ルーチンで指定する送受信バッファーは、連続したデータしか扱うことができず、図 3-5-1 (1) のようにとびとびのデータのみを通信することはできませんでした。このためランク 0 のプロセスは、配列 A のうち●部分のみを一時配列 TEMP にコピーしてから送信し、ランク 1 のプロセスは、受信したメッセージを一時配列 TEMP に受信した後、配列 B にコピーしなければなりませんでした。

この方法は、一時配列 TEMP を新規に確保し、コピーする必要があるため面倒です。本節で紹介する派生データ型という機能を使用すると、図 3-5-1 (2) のように、メモリー上でとびとびのデータを、一時配列 TEMP を介さずに直接通信することができます。言い換えると、とびとびのデータ部分のみを送受信バッファーとして指定することができます。

なお、図 3-5-1 (1) と図 3-5-1 (2) のどちらのパフォーマンスが良いかは、マシン環境に依存します。

図 3-5-1 (1)

図 3-5-1 (2)

図 3-5-2 (1) に示すように、配列 IBUF の中の●のデータのみを通信する場合、まず後述する MPI のルーチンを使用して、INEWTYPE1 というパターンのデータ型を作成します。MPI_INTEGER のように、MPI があらかじめ提供しているデータ型を MPI の基本データ型と呼びます (3-3-4 節参照)。これに対し、INEWTYPE1 のようにユーザーが作成するデータ型を派生データ型と呼びます。

派生データ型 INEWTYPE1 を使用して、例えば集団通信ルーチン MPI_BCAST で通信する場合、図 3-5-2 (1) の下線部のように指定します。これは送受信バッファーが、「IBUF(4) から始まる 1 個の INEWTYPE1 型の部分」であることを意味します。この通信を行うと、送信元のランク 0 のプロセスでは、配列 IBUF 内にとびとびに入っている 6 個の●のみが抽出されて送られます。宛先のランク 1 のプロセスでは、受け取った 6 個の●が配列 IBUF 内のとびとびの部分に直接入ります。

図 3-5-2 (1)

同様に、図 3-5-2 (2) の INEWTYPE2 というパターンの派生データ型を作成した場合、送受信バッファーが、「IBUF(4) から始まる 3 個の INEWTYPE2 型の部分」であることを意味します。

図 3-5-2 (2)

あまり一般的な例ではありませんが、図 3-5-2 (3) のように、通信ルーチンの引数で指定するデータ型がプロセスによって異なっていてもかまいません。ただし、送受信するバイト数の合計は一致している必要があります。

図 3-5-2 (3)

3-5-2　派生データ型に関する主なMPIのサブルーチン

MPIが提供する派生データ型の主なサブルーチンを紹介します。使用方法の詳細は付録を参照して下さい。なお、付録に掲載したサブルーチンのうち、MPI_TYPE_SIZEの説明は、以下では省略します。

以下の説明中で、古いデータ型とは、MPIが提供している基本データ型(MPI_INTEGERなど)、または、その時点までにユーザーが作成した派生データ型を意味します。新しいデータ型とは、古いデータ型に基づいて、以下のサブルーチンを使って新たに作成された派生データ型を意味します。

(1)　MPI_TYPE_CONTIGUOUS

古いデータ型の要素を、指定した個数(以下では3個)だけ連結して新しいデータ型を作成します。

(2)　MPI_TYPE_VECTOR/MPI_TYPE_HVECTOR

以下の(2)〜(5)のサブルーチンは、複数(1つでも可)のブロックから構成される新しいデータ型を作成します。

本サブルーチンでは、各ブロックの先頭間の間隔は一定、各ブロックに含まれる古いデータ型の要素の数は同一、各ブロックに含まれる要素の古いデータ型の種類も同一です。MPI_TYPE_VECTORとMPI_TYPE_HVECTORは、ブロックの先頭間の間隔を指定する方法が異なる（要素数／バイト）以外は同じです。

(3)　MPI_TYPE_CREATE_INDEXED_BLOCK

本サブルーチンはMPI-2で提供されています。ブロックごとに、新しいデータ型の先頭から各ブロックの先頭までの変位を指定します。各ブロックに含まれる古いデータ型の要素の数は同一、各ブロックに含まれる要素の古いデータ型の種類も同一です。最初のブロックの前にブランクがあっても構いません(後述する(7)参照)。間接アドレスを使ったリストベクトルの派生データ型を作成するときに便利です(4-6-6-1節参照)。

(4)　MPI_TYPE_INDEXED/MPI_TYPE_HINDEXED

ブロックごとに、新しいデータ型の先頭から各ブロックの先頭までの変位、各ブロックに含まれる古いデータ型の要素の数を指定します。各ブロックに含まれる要素の古いデータ型の種類は同一です。最初のブロックの前にブランクがあっても構いません(後述する(7)参照)。MPI_TYPE_INDEXEDとMPI_TYPE_HINDEXEDは、先頭からの変位を指定する方法が異なる（要素数／バイト）以外は同じです。

(5) MPI_TYPE_STRUCT

ブロックごとに、新しいデータ型の先頭から各ブロックの先頭までの変位、各ブロックに含まれる古いデータ型の要素の数、各ブロックに含まれる要素の古いデータ型を指定します。最初のブロックの前にブランクがあっても構いません (後述する (7) 参照)。本サブルーチンを使用して、構造体の派生データ型を作成することができます (3-5-4 節参照)。

(6) MPI_TYPE_COMMIT

作成した派生データ型をさらに組み合わせて派生データ型を作成することもできます。以下の例では、まず前述の (2) の派生データ型を作成し、それを (1) で説明した MPI_TYPE_CONTIGUOUS で 2 つ連結して、最終的な派生データ型を作成しています。

最終的な派生データ型が完成した後、それをサブルーチン MPI_TYPE_COMMIT で MPI に登録します。これが完了すると、その派生データ型を、集団通信ルーチンや 1 対 1 通信ルーチンのデータ型の引数に指定することができます。なお、作成途中の一時的な派生データ型は、MPI_TYPE_COMMIT で登録する必要はありません。

(7)　派生データ型の前と後のブランクについて

MPI が提供する基本データ型 (MPI_INTEGER など) や、前述の (1) (2) で作成した派生データ型の場合、以下の上図に示すように、そのままでは前後にブランクは入りません。以下の下図のように、前や後にブランク（着色部分）を入れたい場合、(8) か (9) の方法で行います。

一方、前述の (3) (4) (5) で作成した派生データ型の場合、新しいデータ型の先頭から最初のブロックの先頭までの変位を 0 より大きな値にすると、図 3-5-3 (1)（左図）のように最初のブロックの前にブランクを入れることができます。ただし、この方法では派生データ型の最後のブロックの後にブランクを入れることはできません。

また、この派生データ型を使用してデータを 2 個以上通信する場合や、この派生データ型を MPI_TYPE_CONTIGUOUS で 2 個以上連続させた場合、図 3-5-3 (1)（右図）のように、2 個目以降の派生データ型では前のブランクが取れてしまいます。図 3-5-3 (2)（右図）のようにブランクを残したい場合、(8) か (9) の方法で行います。

図 3-5-3 (1)

図 3-5-3 (2)

(8)　派生データ型の前後にブランクを入れる方法 (1) (MPI_TYPE_STRUCT を使用)

まず、(5) で説明した MPI_TYPE_STRUCT のみを用いて、派生データ型の前後にブランクを入れる方法を説明します。派生データ型の最初のブロックの前にブランクを入れたい場合、図 3-5-3 (3)（上図）に示すように、ブランクの前に要素数 1、大きさ 0、データ型が「MPI_LB」(LB は Lower Bound の略) のブロックを定義します。同様に派生データ型の最後のブロックの後にブランクを入れたい場合、ブランクの後に要素数 1、大きさ 0、データ型が「MPI_UB」(UB は Upper Bound の略) のブロックを定義します。

この派生データ型を使用してデータを 2 個以上通信する場合や、この派生データ型を MPI_TYPE_CONTIGUOUS で 2 個以上連続させた場合、図 3-5-3 (3)（下図）に示すように、2 個目以降の派生データ型では前のブランクが残ります。

図 3-5-3 (3)

次に、MPI が提供する基本データ型 (MPI_INTEGER など) や、前述の (1)〜(4) で作成した派生データ型の前後にブランクを入れる方法を、(1) の派生データの例で説明します。まず図 3-5-3 (4)（上図）に示すように、派生データ型（以後「途中のデータ型」）を作成します。次に図 3-5-3 (4)（下図）に示すように、MPI_TYPE_STRUCT を使用し、上記で説明した「MPI_LB」と「MPI_UB」をブランクの前後に指定して「最終のデータ型」を作成します。このとき、「途中のデータ型」の要素の数は「3」ではなく「1」となります。なお、この方法で作成した派生データ型も、図 3-5-3 (3) と同様に、2 個目以降の派生データ型では前のブランクが残ります。

図 3-5-3 (4)

(9)　派生データ型の前後にブランクを入れる方法 (2) (MPI_TYPE_CREATE_RESIZED を使用)

MPI-2 で提供された MPI_TYPE_CREATE_RESIZED（付録参照）を使って、派生データ型の前後にブランクを入れる方法を説明します。本サブルーチンは、MPI が提供する基本データ型 (MPI_INTEGER など) や、前述の (1)〜(5) で作成した派生データ型に対して「範囲」を指定します。元の大きさより大きな範囲を指定すると、図 3-5-3 (5) に示すように、派生データ型の後にブランクを入れることができます。

この方法で派生データ型の前と後の両方にブランクを入れたい場合、次のようにします。まず (7) で説明した方法で、(3) (4) (5) を使って図 3-5-3 (6)（上図）に示すように前のブランクを入れます。この派生データ型に対して、図 3-5-3 (6)（下図）に示すように、MPI_TYPE_CREATE_RESIZED で元の大きさと同じ大きさの範囲を指定すると、前と後の両方にブランクが入ります。2 個目以降の派生データ型では前のブランクが残ります。

図 3-5-3 (5)

図 3-5-3 (6)

3-5-3　部分配列を表す派生データ型の作成

本節と次節では、並列化したときに派生データ型が必要になる例を紹介します。図 3-5-4 (1) で、ランク 0 が着色した部分を担当しているとします。Fortran では、2 次元配列はメモリー上で 1 次元方向に連続 (C 言語では逆) になっています (3-1 節参照)。この例では計算領域を 2 次元方向に分割しており、着色した部分はメモリー上で連続なので直接送信バッファーとして送信することができます。ただし、図 3-5-4 (2) のように周囲にノリシロがある場合、着色した部分は不連続になるので、着色した部分のみを送信バッファーにすることはできません。なお本節では送信バッファーで説明しますが、受信バッファーの場合も全く同じです。

一方計算領域を 1 次元方向に分割した図 3-5-4 (3) では、着色した部分がメモリー上で不連続なので、着色部分のデータを一度連続な送信バッファーに詰め込んでから送信する必要があります。ただし、図 3-5-4 (4) のように計算領域が縮小 (4-5-7 節参照) されている場合は連続なので、直接送信することができます。

図 3-5-4 (5) (6) のように境界部分の 1 行 (または 1 列) のみを通信する場合も、図 3-5-4 (6) では着色した部分がメモリー上で不連続なので、着色部分のデータを一度連続な送信バッファーに詰め込んでから送信する必要があります。この場合は図 3-5-4 (7) に示すように、計算領域が縮小されていても不連続になります。

図 3-5-4 (1)　連続

図 3-5-4 (2)　不連続

図 3-5-4 (3)　不連続

図 3-5-4 (4)　連続

図 3-5-5 (1)

図 3-5-4 (5)　連続

図 3-5-4 (6)　不連続

図 3-5-4 (7)　不連続

図 3-5-5 (2)

図 3-5-4 (2) (3) (6) (7) の着色した部分のようにメモリー上で不連続な部分を送信する場合、送信バッファーを新たに確保し、送信バッファーに詰め込むロジックをプログラムに追加しなければなりません。また（通常は）受信側でも同様の処理をしなければならず、修正が若干面倒になります。このような場合、派生データ型を使用すると、メモリー上で不連続な部分を直接送受信することができます。

図 3-5-4 (2) (3) (6) (7) の形状を一般化すると、計算領域が 2 次元であれば図 3-5-5 (1)、3 次元であれば図 3-5-5 (2) のようになります。本書では図 3-5-5 (1) (2) の外枠の配列を全体配列、着色した部分を部分配列と呼びます。部分配列の派生データ型を、前節で紹介した MPI の派生データを作成するサブルーチンで 1 から作成してもよいのですが、その手間を減らすため、MPI-2 ではこのパターンの派生データ型を簡単に作成するサブルーチン `MPI_TYPE_CREATE_SUBARRAY` が用意されており、使い方を次節で説明します。

一方 MPI-2 が使用できないマシン環境の場合は、`MPI_TYPE_CREATE_SUBARRAY` の代わりに、3-5-3-2 節で紹介する（筆者が作成した）ユーティリティー・サブルーチンを使用すれば、同様の派生データ型を作成することができます (Fortran のみ)。ただし、`MPI_TYPE_CREATE_SUBARRAY` は全体配列が何次元でも使用可能なのに対し、ユーティリティー・サブルーチンは 2，3 次元のみにしか使用できません。

なお、本書の 4、5 章に、このパターンの派生データ型を使用するプログラム例を数ヶ所で掲載していますが、本書のプログラムは基本的に MPI-1 の範囲で作成したため、ユーティリティー・サブルーチンを使っています。しかし、MPI-2 が使用できる環境の場合は、MPI-2 で提供されている `MPI_TYPE_CREATE_SUBARRAY` を使用するようにして下さい。

3-5-3-1　MPI_TYPE_CREATE_SUBARRAY

MPI-2 で提供されている MPI_TYPE_CREATE_SUBARRAY を使用して、図 3-5-6 (1) に示す全体配列 N(2:5,-1:4) 内にある、●に示す部分配列の派生データ型を作成する方法を図 3-5-6 (2) で説明します。なお、MPI_TYPE_CREATE_SUBARRAY の各引数の詳細は付録を参照して下さい。また、MPI-2 が使用できないマシン環境の場合は、次節の方法で作成して下さい。

- 図 3-5-6 (2) の①で、MPI_TYPE_CREATE_SUBARRAY の引数で使用する各配列（整数で名前は任意）を宣言します。図 3-5-6 (1) の全体配列が 2 次元なので、配列の大きさは 2 となります。
- ②で、全体配列の 1 次元目と 2 次元目の要素数を配列 ISIZE に設定します。
- ③で、部分配列の 1 次元目と 2 次元目の要素数を配列 ISUBSIZE に設定します。
- ④で、全体配列の各次元の先頭を [0] としたときの、部分配列の先頭の相対座標 (図 3-5-6 (1) の○) を配列 ISTART に指定します。
- ⑤の引数に、全体配列の次元数 2、上記で作成した各配列などを指定します。なお C 言語の場合、二重線の部分は「MPI_ORDER_C」となります。⑤を実行すると、全体配列 N の先頭（★）を起点とした部分配列を表す派生データ型 INEWTYPE が作成されます。
- ⑥で INEWTYPE を MPI に登録します。これで INEWTYPE を通信ルーチンの引数で使用することができます。
- ⑦の通信で INEWTYPE を使用します。★の部分を示す配列 N (または N(2,-1)) を送受信バッファーの先頭アドレスに指定します。⑦の最初の 3 つの引数は、送受信バッファーが、「配列 N の先頭から始まる、1 個の INEWTYPE 型の部分」であることを意味します。

図 3-5-6 (1)

```
   :
 INCLUDE 'mpif.h'
 INTEGER N(2:5,-1:4)
 INTEGER ISIZE(2),ISUBSIZE(2),ISTART(2)                    ①
 CALL MPI_INIT(IERR)
 CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)
 CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)
 ISIZE(1) = 4
 ISIZE(2) = 6                                              ②
 ISUBSIZE(1) = 2
 ISUBSIZE(2) = 3                                           ③
 ISTART(1) = 1
 ISTART(2) = 2                                             ④
 CALL MPI_TYPE_CREATE_SUBARRAY(2,ISIZE,ISUBSIZE,ISTART,
&     MPI_ORDER_FORTRAN,MPI_INTEGER,INEWTYPE,IERR)         ⑤
 CALL MPI_TYPE_COMMIT(INEWTYPE,IERR)                       ⑥
   :
 CALL MPI_BCAST(N,1,INEWTYPE,0,MPI_COMM_WORLD,IERR)        ⑦
   :
```

図 3-5-6 (2)

3-5-3-2　ユーティリティー・サブルーチン

前節で説明した MPI_TYPE_CREATE_SUBARRAY は MPI-2 で提供されているサブルーチンなので、MPI-2 が使えないマシン環境では使うことができません。この場合、MPI-1 の派生データ型サブルーチンを組み合わせて（筆者が）作成したユーティリティー・サブルーチン（詳細は次ページ以降参照）を使用すれば、部分配列の派生データ型を作成することができます (Fortran のみ)。ただし全体配列が 2，3 次元の場合にのみ使用可能です。

2 次元用のサブルーチン PARA_TYPE_BLOCK2A と PARA_TYPE_BLOCK2 の使用例を以下に示します。

ユーティリティー・サブルーチン PARA_TYPE_BLOCK2A

図 3-5-7 (1) に示す 2 次元全体配列 N 内にある●の部分配列を示す派生データ INEWTYPE を、ユーティリティー・サブルーチン PARA_TYPE_BLOCK2A（引数の詳細は次ページ以降参照）を使用して作成するプログラム例を図 3-5-7 (2) に示します。①の各引数で図 3-5-7 (1) の○の部分を指定します。

PARA_TYPE_BLOCK2A では、図 3-5-7 (1) に示すように、作成した派生データ型 INEWTYPE の起点は部分配列の先頭★なので (前節の MPI_TYPE_CREATE_SUBARRAY と違う)、派生データ INEWTYPE を使って通信を行うときは、②の二重線に示すように、★の部分を示す N(3,1) を送受信バッファーの先頭アドレスに指定します。②の最初の 3 つの引数は、送受信バッファーが、「配列 N(3,1) から始まる、1 個の INEWTYPE 型の部分」であることを意味します。

図 3-5-7 (1)

```
  :
INTEGER N(2:5,-1:4)
  :
CALL PARA_TYPE_BLOCK2A(2,5,2,3,MPI_INTEGER,INEWTYPE)     ①
CALL MPI_BCAST(N(3,1),1,INEWTYPE,0,MPI_COMM_WORLD,IERR)  ②
  :
```

図 3-5-7 (2)

ユーティリティー・サブルーチン PARA_TYPE_BLOCK2

一方ユーティリティー・サブルーチン MPI_TYPE_BLOCK2 では、図 3-5-8 (2) の③の各引数で図 3-5-8 (1) の○の部分を指定します。図 3-5-8 (1) に示すように、作成した派生データ型 INEWTYPE の起点は全体配列 N の先頭★なので (前節の MPI_TYPE_CREATE_SUBARRAY と同じ)、派生データ INEWTYPE を使って通信を行うときは、④の二重線に示すように、★の部分を示す配列 N (または N(2,-1)) を送受信バッファーの先頭アドレスに指定します。②の最初の 3 つの引数は、送受信バッファーが、「配列 N の先頭から始まる、1 個の INEWTYPE 型の部分」であることを意味します。

この派生データは全体配列の先頭から部分配列の先頭までの変位の情報も含むため、作成するときは①より引数が多くなりますが、通信時の送受信バッファーの指定は②より簡単になります。

図 3-5-8 (1)

```
  :
INTEGER N(2:5,-1:4)
  :                                                  ③
CALL PARA_TYPE_BLOCK2(2,5,-1,3,4,1,3,MPI_INTEGER,INEWTYPE)
CALL MPI_BCAST(N,1,INEWTYPE,0,MPI_COMM_WORLD,IERR)     ④
  :
```

図 3-5-8 (2)

上記の各サブルーチンの 3 次元版が PARA_TYPE_BLOCK3A と PARA_TYPE_BLOCK3 です。各ユーティリティー・サブルーチンの引数の詳細と実際のプログラムは次ページ以降を参照して下さい。

（注）図 3-5-9 (2)〜(4) の各プログラム内のサブルーチン MPI_TYPE_SIZE は、本書の 2007 年 1 月 1 日より前の版では MPI_TYPE_EXTENT になっていましたが、MPI_TYPE_SIZE の方が適当なので修正しました (MPI_TYPE_EXTENT でも正しい結果が得られます)。

サブルーチン PARA_TYPE_BLOCK2A

要素のデータ型が ioldtype の 2 次元全体配列 (imin:imax, 任意) 中にある、大きさが ilen × jlen の 2 次元部分配列を表す派生データ型を inewtype に戻します。この派生データ型の起点は★なので、MPI で通信を行う際、★の部分を送受信バッファーの先頭アドレスに指定して下さい。

CALL PARA_TYPE_BLOCK2A(imin, imax, ilen, jlen, ioldtype, inewtype)

- imin: 整数。2 次元全体配列の 1 次元目の下限値を指定します。
- imax: 整数。2 次元全体配列の 1 次元目の上限値を指定します。
- ilen: 整数。2 次元部分配列の 1 次元目の大きさを指定します。
- jlen: 整数。2 次元部分配列の 2 次元目の大きさを指定します。
- ioldtype: 整数。古いデータ型を指定します。例えば整数配列ならば MPI_INTEGER を指定して下さい。
- inewtype: 整数。作成された新しい派生データ型が戻ります。

```
  SUBROUTINE PARA_TYPE_BLOCK2A
&          (IMIN,IMAX,ILEN,JLEN,IOLDTYPE,INEWTYPE)
  INCLUDE 'mpif.h'
  CALL MPI_TYPE_VECTOR(JLEN,ILEN,IMAX-IMIN+1,IOLDTYPE,INEWTYPE,IERR)
  CALL MPI_TYPE_COMMIT(INEWTYPE,IERR)
  END
```

図 3-5-9 (1)

サブルーチン PARA_TYPE_BLOCK2

（注）図3-5-9 (2) 内のMPI_TYPE_SIZEは、本書の2007年1月1日より前の版ではMPI_TYPE_EXTENTになっていましたが、MPI_TYPE_SIZEの方が適当なので修正しました(MPI_TYPE_EXTENTでも正しい結果が得られます)。

要素のデータ型がioldtypeの2次元全体配列（imin:imax, jmin:任意）中にある、位置と大きさが（ista:iend, jsta:jend）の部分配列を表す派生データ型をinewtypeに戻します。この派生データ型の起点は★なので、MPIで通信を行う際、★の部分を送受信バッファーの先頭アドレスに指定して下さい。

CALL PARA_TYPE_BLOCK2(imin, imax, jmin, ista, iend, jsta, jend, ioldtype, inewtype)

- imin: 整数。2次元全体配列の1次元目の下限値を指定します。
- imax: 整数。2次元全体配列の1次元目の上限値を指定します。
- jmin: 整数。2次元全体配列の2次元目の下限値を指定します。
- ista: 整数。2次元部分配列の1次元目の下限値を指定します。
- iend: 整数。2次元部分配列の1次元目の上限値を指定します。
- jsta: 整数。2次元部分配列の2次元目の下限値を指定します。
- jend: 整数。2次元部分配列の2次元目の上限値を指定します。
- ioldtype: 整数。古いデータ型を指定します。例えば整数配列ならばMPI_INTEGERを指定して下さい。
- inewtype: 整数。作成された新しい派生データ型が戻ります。

```
 SUBROUTINE PARA_TYPE_BLOCK2
&          (IMIN,IMAX,JMIN,ISTA,IEND,JSTA,JEND,IOLDTYPE,INEWTYPE)
 INCLUDE 'mpif.h'
 INTEGER IBLOCK(2),IDISP(2),ITYPE(2)
 CALL MPI_TYPE_SIZE(IOLDTYPE,ISIZE,IERR)
 ILEN = IEND - ISTA + 1
 JLEN = JEND - JSTA + 1
 CALL MPI_TYPE_VECTOR(JLEN,ILEN,IMAX-IMIN+1,IOLDTYPE,ITEMP,IERR)
 IBLOCK(1) = 1
 IBLOCK(2) = 1
 IDISP(1) = 0
 IDISP(2) = ((IMAX-IMIN+1)*(JSTA-JMIN)+(ISTA-IMIN))*ISIZE
 ITYPE(1) = MPI_LB
 ITYPE(2) = ITEMP
 CALL MPI_TYPE_STRUCT(2,IBLOCK,IDISP,ITYPE,INEWTYPE,IERR)
 CALL MPI_TYPE_COMMIT(INEWTYPE,IERR)
 END
```

図3-5-9 (2)

サブルーチン PARA_TYPE_BLOCK3A

（注）図 3-5-9 (3) 内の MPI_TYPE_SIZE は、本書の 2007 年 1 月 1 日より前の版では MPI_TYPE_EXTENT になっていましたが、MPI_TYPE_SIZE の方が適当なので修正しました (MPI_TYPE_EXTENT でも正しい結果が得られます)。

要素のデータ型が ioldtype の 3 次元全体配列 (imin:imax, jmin:jmax, 任意) 中にある、大きさが ilen × jlen × klen の 3 次元部分配列を表す派生データ型を inewtype に戻します。この派生データ型の起点は★なので、MPI で通信を行う際、★の部分を送受信バッファーの先頭アドレスに指定して下さい。

CALL PARA_TYPE_BLOCK3A(imin, imax, jmin, jmax, ilen, jlen, klen, ioldtype, inewtype)

- imin: 整数。3 次元全体配列の 1 次元目の下限値を指定します。
- imax: 整数。3 次元全体配列の 1 次元目の上限値を指定します。
- jmin: 整数。3 次元全体配列の 2 次元目の下限値を指定します。
- jmax: 整数。3 次元全体配列の 2 次元目の上限値を指定します。
- ilen: 整数。3 次元部分配列の 1 次元目の大きさを指定します。
- jlen: 整数。3 次元部分配列の 2 次元目の大きさを指定します。
- klen: 整数。3 次元部分配列の 3 次元目の大きさを指定します。
- ioldtype: 整数。古いデータ型を指定します。例えば整数配列ならば MPI_INTEGER を指定して下さい。
- inewtype: 整数。作成された新しい派生データ型が戻ります。

```
  SUBROUTINE PARA_TYPE_BLOCK3A
&          (IMIN,IMAX,JMIN,JMAX,ILEN,JLEN,KLEN,IOLDTYPE,INEWTYPE)
  INCLUDE 'mpif.h'
  CALL MPI_TYPE_SIZE(IOLDTYPE,ISIZE,IERR)
  CALL MPI_TYPE_VECTOR(JLEN,ILEN,IMAX-IMIN+1,IOLDTYPE,ITEMP,IERR)
  IDIST = (IMAX-IMIN+1)*(JMAX-JMIN+1)*ISIZE
  CALL MPI_TYPE_HVECTOR(KLEN,1,IDIST,ITEMP,INEWTYPE,IERR)
  CALL MPI_TYPE_COMMIT(INEWTYPE,IERR)
  END
```

図 3-5-9 (3)

サブルーチン PARA_TYPE_BLOCK3

（注）図 3-5-9 (4) 内の MPI_TYPE_SIZE は、本書の 2007 年 1 月 1 日より前の版では MPI_TYPE_EXTENT になっていましたが、MPI_TYPE_SIZE の方が適当なので修正しました (MPI_TYPE_EXTENT でも正しい結果が得られます)。

要素のデータ型が ioldtype の 3 次元全体配列 (imin:imax, jmin:jmax, kmin:任意) 中にある、位置と大きさが (ista:iend, jsta:jend, ksta:kend) の 3 次元部分配列を表す派生データ型を inewtype に戻します。この派生データ型の起点は★なので、MPI で通信を行う際、★の部分を送受信バッファーの先頭アドレスに指定して下さい。

CALL PARA_TYPE_BLOCK3(imin, imax, jmin, jmax, kmin, ista, iend, jsta, jend, ksta, kend, ioldtype, inewtype)

- imin: 整数。3 次元全体配列の 1 次元目の下限値を指定します。
- imax: 整数。3 次元全体配列の 1 次元目の上限値を指定します。
- jmin: 整数。3 次元全体配列の 2 次元目の下限値を指定します。
- jmax: 整数。3 次元全体配列の 2 次元目の上限値を指定します。
- kmin: 整数。3 次元全体配列の 3 次元目の下限値を指定します。
- ista: 整数。3 次元部分配列の 1 次元目の下限値を指定します。
- iend: 整数。3 次元部分配列の 1 次元目の上限値を指定します。
- jsta: 整数。3 次元部分配列の 2 次元目の下限値を指定します。
- jend: 整数。3 次元部分配列の 2 次元目の上限値を指定します。
- ksta: 整数。3 次元部分配列の 3 次元目の下限値を指定します。
- kend: 整数。3 次元部分配列の 3 次元目の上限値を指定します。
- ioldtype: 整数。古いデータ型を指定します。例えば整数配列ならば MPI_INTEGER を指定して下さい。
- inewtype: 整数。作成された新しい派生データ型が戻ります。

```
 SUBROUTINE PARA_TYPE_BLOCK3(IMIN,IMAX,JMIN,JMAX,KMIN,
&           ISTA,IEND,JSTA,JEND,KSTA,KEND,IOLDTYPE,INEWTYPE)
 INCLUDE 'mpif.h'
 INTEGER IBLOCK(2),IDISP(2),ITYPE(2)
 CALL MPI_TYPE_SIZE(IOLDTYPE,ISIZE,IERR)
 ILEN = IEND - ISTA + 1
 JLEN = JEND - JSTA + 1
 KLEN = KEND - KSTA + 1
 CALL MPI_TYPE_VECTOR(JLEN,ILEN,IMAX-IMIN+1,IOLDTYPE,ITEMP,IERR)
 IDIST = (IMAX-IMIN+1)*(JMAX-JMIN+1)*ISIZE
 CALL MPI_TYPE_HVECTOR(KLEN,1,IDIST,ITEMP,ITEMP2,IERR)
 IBLOCK(1) = 1
 IBLOCK(2) = 1
 IDISP(1) = 0
 IDISP(2) = (  (IMAX-IMIN+1)*(JMAX-JMIN+1)*(KSTA-KMIN)
&            + (IMAX-IMIN+1)*(JSTA-JMIN) + (ISTA-IMIN)  )*ISIZE
 ITYPE(1) = MPI_LB
 ITYPE(2) = ITEMP2
 CALL MPI_TYPE_STRUCT(2,IBLOCK,IDISP,ITYPE,INEWTYPE,IERR)
 CALL MPI_TYPE_COMMIT(INEWTYPE,IERR)
 END
```

図 3-5-9 (4)

3-5-4　構造体を表す派生データ型の作成

構造体で定義した変数や配列は、MPI が提供するデータ型 (MPI_INTEGER など) では通信できないので、構造体を表す派生データ型を作成して通信を行います。図 3-5-10 (1) の構造体の派生データ型を作成するプログラム例を図 3-5-10 (2) に示します。

- ①で図 3-5-10 (1) に示す KOUZOUTAI という名の構造体を定義し、②で変数 WORK をこの構造体で宣言します。
- ③～⑧で、この変数 WORK を通信するために使用する派生データ型 IKOUZOU を、MPI のサブルーチン MPI_TYPE_STRUCT を使用して作成します。MPI_TYPE_STRUCT の引数の詳細は付録を参照して下さい。
  図 3-5-10 (1) の構造体 KOUZOUTAI は、以下の 3 つの部分（以後ブロックと呼びます）から構成されています。まず③で、MPI_TYPE_STRUCT の各引数に使用する、大きさ 3 (= ブロック数) の配列（整数で名前は任意）を宣言します。
  [ブロック 1] 4 バイトの整数が 2 個
  [ブロック 2] 8 バイトの倍精度実数が 1 個
  [ブロック 3] 4 バイトの単精度実数が 2 個
- ④で配列 IBLOCK に、上記の各ブロックに含まれる変数の数（上記下線部の数字）を代入します。
- ⑤で配列 IDISP に、構造体の先頭から各ブロックの先頭までの変位（バイト）を代入します (図 3-5-10 (1) の○参照)。
- ⑥で配列 ITYPE に、上記の各ブロックに含まれる変数のデータ型を指定します。
- ⑦で MPI のサブルーチン MPI_TYPE_STRUCT をコールすると、下線部の IKOUZOU（任意の整数）に、構造体 KOUZOUTAI を表す派生データ型が戻ります。
- ⑧で派生データ型 IKOUZOU を MPI に登録します。これで IKOUZOU を MPI の通信のデータ型として使うことができます。
- ⑨でランク 0 は、構造体で宣言した変数 WORK に値を代入し、⑩で派生データ型 IKOUZOU を使用して他のプロセスに通信します。言うまでもありませんが、⑩で例えば配列 WORK(2) の 2 つのデータを送信する場合は「CALL MPI_BCAST (WORK,2,IKOUZOU,～)」のようになります。

図 3-5-10 (1)

```
   :
 TYPE::KOUZOUTAI
   INTEGER::I
   INTEGER::J
   REAL*8 ::X                                  ①
   REAL   ::A
   REAL   ::B
 END TYPE KOUZOUTAI
 TYPE(KOUZOUTAI) WORK                          ②
 INTEGER IBLOCK(3),IDISP(3),ITYPE(3)           ③
 CALL MPI_INIT(IERR)
 CALL MPI_COMM_SIZE
&    (MPI_COMM_WORLD,NPROCS,IERR)
 CALL MPI_COMM_RANK
&    (MPI_COMM_WORLD,MYRANK,IERR)
 IBLOCK(1) = 2
 IBLOCK(2) = 1                                 ④
 IBLOCK(3) = 2
 IDISP (1) = 0
 IDISP (2) = 8                                 ⑤
 IDISP (3) = 16
 ITYPE (1) = MPI_INTEGER
 ITYPE (2) = MPI_REAL8                         ⑥
 ITYPE (3) = MPI_REAL
 CALL MPI_TYPE_STRUCT(3,IBLOCK,IDISP,ITYPE,
&                    IKOUZOU,IERR)            ⑦
 CALL MPI_TYPE_COMMIT(IKOUZOU,IERR)            ⑧
 IF (MYRANK==0) THEN
   WORK%I = 1
   WORK%J = 2
   WORK%X = 3.0D0                              ⑨
   WORK%A = 4.0
   WORK%B = 5.0
 ENDIF
 CALL MPI_BCAST(WORK,1,IKOUZOU,0,
&               MPI_COMM_WORLD,IERR)           ⑩
   :
```

図 3-5-10 (2)

3-6　新グループの作成

3-2 節で説明したように、プロセスの集合からなるグループのグループ名をコミュニケータと呼びます。このうち、全プロセスから構成されるグループは MPI があらかじめ作成しており、そのコミュニケータを MPI_COMM_WORLD と呼びます。これとは別に、任意のプロセスから構成される新グループをユーザーが作成することができます。

3-1 節で紹介した MPI サブルーチンの分類のうち、コミュニケータに所属するサブルーチンで新グループを作成しますが、このうち MPI_COMM_SPLIT のみでも新グループを作成できるので、本書では MPI_COMM_SPLIT のみを紹介します。

付録の MPI_COMM_SPLIT の説明を一読して下さい…。

某国の政界では、新グループや新党、新新党が頻繁に発生と消滅を繰り返していますが (本書の初版を作成した 1995 年頃の話です)、実は MPI の世界では、新グループを作成する必要がある場合はほとんどありません。しいて挙げれば、全く別の複数のプログラム（例えば構造解析と流体解析）をそれぞれ並列化して実行する、連成解析のような場合に必要となります (4-7-1-2 節参照)。

上記で紹介した MPI_COMM_SPLIT を使用して、例えば図 3-6-1 に示すように、構造解析を担当するプロセスのグループ (コミュニケータ IKOUZOU) と、流体解析を担当するプロセスのグループ (コミュニケータ IRYUTAI) を作成します。このときグループ内の各プロセスには、ランクの値を付けます (下記の例ではランク 0，1，2)。

例えば構造解析のプロセス間で通信を行う場合、通信ルーチンの引数にはコミュニケータ IKOUZOU を指定します。一方、構造解析のプロセスと流体解析のプロセス間で通信を行う場合、通信ルーチンの引数には MPI_COMM_WORLD を指定します。また通信ルーチンの引数にランクの値を指定する場合、コミュニケータで指定したグループ内でのランクの値を指定します。

図 3-6-1

3-7　C 言語の MPI 関数

C 言語でも MPI の関数を使用することができます。以下に、本書の付録に掲載した MPI_BCAST のサンプルプログラムを C 言語で書いたプログラムを示します。

```
#include "mpi.h"
int main(argc, argv)
int argc;
char **argv;
{
     int nprocs, myrank, ibuf;
     MPI_Init(&argc, &argv);
     MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
     if (myrank == 0) {
       ibuf = 1;
     } else {
       ibuf = 0;
     }
     MPI_Bcast(&ibuf, 1, MPI_INT, 0, MPI_COMM_WORLD);
     printf("MYRANK = %d IBUF = %d\n",myrank,ibuf);
     MPI_Finalize();
}
```

図 3-7-1

Fortran と C 言語の MPI の主な相違点を以下に示します。

なお、本書の付録に掲載したのは Fortran 版の MPI ルーチンの使用方法です。例えば Fortran 版で引数が整数になっていても、C 言語版では別のデータ型で宣言しなければならない場合がありますので、C 言語のユーザーの方は、参考文献 [15] [27] の C 言語の説明箇所を参照して下さい。また C 言語用の MPI のテキストが参考文献 [37] [38] にあります。

- C 言語の MPI 関数は大文字と小文字の区別があり、関数名の一部と、MPI が使用する定数 (MPI_COMM_WORLD など) は大文字になります。
- 引数は Fortran と C 言語でほぼ同じですが、Fortran の最後の引数 ierror（戻り値が返る）が C 言語にはなく、下記に示すように MPI の関数自体が戻り値となります。

  ```
  int ierr;
  ierr = MPI_Finalize();
  ```

- 引数の一部がポインターになります。
- Fortran のインクルードファイル「mpif.h」は、C 言語では「mpi.h」となります。
- 例えば Fortran の「INTEGER ISTATUS(MPI_STATUS_SIZE)」の宣言は、C 言語では「MPI_Status istatus;」となります。
- MPI の通信ルーチンの送受信バッファーに多次元配列（のポインター）を指定する場合、配列内の各要素がメモリー上で連続している多次元配列のみ、指定することができます。

C 言語の場合の MPI の基本データ型、MPI の定義済み演算を以下に示します。MPI_FLOAT_INT など (Fortran の MPI_2REAL などに相当) は、異なる 2 つのデータ型のペアを（構造体を使用して）扱うことができます (3-3-6 節参照)。

	C言語のデータ型	MPIの基本データ型
整数型	signed short int	MPI_SHORT
	signed int	MPI_INT
	signed long int	MPI_LONG
	unsigned short int	MPI_UNSIGNED_SHORT
	unsigned int	MPI_UNSIGNED
	unsigned long int	MPI_UNSIGNED_LONG
	(longlong int)	(MPI_LONG_LONG_INT)
実数型	float	MPI_FLOAT
	double	MPI_DOUBLE
	long double	MPI_LONG_DOUBLE
その他	signed char	MPI_CHAR
	unsigned char	MPI_UNSIGNED_CHAR
	対応なし	MPI_BYTE, MPI_PACKED

図 3-7-2 (1)

（注）マシン環境によってはカッコをつけたデータ型などが提供されている場合もあります。

MPIの基本データ型	MPIの定義済み演算
上記の整数型と実数型	MPI_SUM（合計） MPI_PROD(積)
上記の整数型と実数型	MPI_MAX（最大） MPI_MIN（最小）
MPI_FLOAT_INT MPI_DOUBLE_INT MPI_LONG_INT MPI_2INT MPI_SHORT_INT MPI_LONG_DOUBLE_INT	MPI_MAXLOC(最大と位置) MPI_MINLOC(最小と位置)
上記の整数型	MPI_LAND(論理AND) MPI_LOR（論理OR) MPI_LXOR(論理XOR)
上記の整数型とMPI_BYTE	MPI_BAND(ビットAND) MPI_BOR（ビットOR) MPI_BXOR(ビットXOR)

図 3-7-2 (2)

ただし、

$$\mathtt{MPI_FLOAT_INT} = \{\mathtt{MPI_FLOAT}, \mathtt{MPI_INT}\}$$

$$\mathtt{MPI_DOUBLE_INT} = \{\mathtt{MPI_DOUBLE}, \mathtt{MPI_INT}\}$$

$$\mathtt{MPI_LONG_INT} = \{\mathtt{MPI_LONG}, \mathtt{MPI_INT}\}$$

$$\mathtt{MPI_2INT} = \{\mathtt{MPI_INT}, \mathtt{MPI_INT}\}$$

$$\mathtt{MPI_SHORT_INT} = \{\mathtt{MPI_SHORT}, \mathtt{MPI_INT}\}$$

$$\mathtt{MPI_LONG_DOUBLE_INT} = \{\mathtt{MPI_LONG_DOUBLE}, \mathtt{MPI_INT}\}$$

3-8　MPI-2

3-1 節で紹介したように、MPI の新機能が MPI-2 で追加になりました。本節ではそのうちの一部を紹介します。これらの機能は、使用するマシン環境に MPI-2 が導入されている場合のみ、使用することができます。

3-8-1　MPI_IN_PLACE

3-3-5 節で説明したように、下記の集団通信ルーチンには、「送信バッファーと受信バッファー（の実際に使用する部分）は　メモリー上で重なってはならない」という制限があります。このため通信の際、送信バッファーとは別に、送信バッファーと同じ大きさの受信バッファーを新たに用意しなければならず、バッファーの大きさが大きいときはメモリーが無駄になりました。

(グループ 2) ○ MPI_GATHER, ○ MPI_SCATTER, ○ MPI_ALLGATHER, × MPI_ALLTOALL
○ MPI_GATHERV, ○ MPI_SCATTERV, ○ MPI_ALLGATHERV, × MPI_ALLTOALLV, × MPI_ALLTOALLW
(グループ 3) ○ MPI_REDUCE, ○ MPI_ALLREDUCE, ○ MPI_SCAN, × MPI_EXSCAN, ○ MPI_REDUCE_SCATTER

MPI-2 では、上記のうち○をつけたルーチンについては、以下に示す方法を使用すれば「送信バッファーと受信バッファーをメモリー上で同一にする」ことができます。

MPI_ALLREDUCE の例で説明します。まず図 3-8-1 (1) に今までの方法を示します。送信バッファー S にセットされた送信データが、MPI_ALLREDUCE によって図 3-8-1 (2) に示すように通信しながら合計され、結果が全プロセスの受信バッファー SS に入ります。上記の制限があるため、送信バッファー S とは別に、受信バッファー SS を用意しなければなりません。

一方 MPI-2 の新機能を使用した図 3-8-2 (1) では、（本来送信バッファーを指定する）1 つ目の引数に、MPI-2 で新たに提供された変数 MPI_IN_PLACE を指定します。そして（本来受信バッファーを指定する）2 つ目の引数に指定した変数は、送信バッファーと受信バッファーを兼用します。まず図 3-8-2 (2) に示すように変数 S に送信データをセットし、MPI_ALLREDUCE をコールします。すると通信しながら合計された結果が変数 S にセットされます（元の値が更新されます）。このため受信バッファー SS を新たに用意する必要はありません。なお、MPI_IN_PLACE の使い方は各集団通信サブルーチンで若干異なりますので、付録の「MPI_IN_PLACE の使い方」を参照して下さい。

この機能は、使用する MPI が MPI-2 の MPI_IN_PLACE オプションをサポートしている場合にのみ使用することができます。MPI-2 をサポートしている MPI でも、MPI-2 の全ての機能はサポートしておらず、MPI_IN_PLACE を使用することができない場合もありますので、MPI_IN_PLACE オプションが使用できるかどうかを、マニュアルなどで確認してからお使い下さい。

```
   :
 CALL MPI_ALLREDUCE
&    (S,SS,1,MPI_INTEGER,
&     MPI_SUM,MPI_COMM_WORLD,IERR)
 S = SS
 PRINT *,S
   :
```

図 3-8-1 (1)

図 3-8-1 (2)

```
   :
 CALL MPI_ALLREDUCE
&    (MPI_IN_PLACE,S,1,MPI_INTEGER,
&     MPI_SUM,MPI_COMM_WORLD,IERR)
 PRINT *,S
   :
```

図 3-8-2 (1)

図 3-8-2 (2)

第4章　プログラムの並列化の方法

第3章で、主なメッセージ交換サブルーチンの使用方法について説明しました。英語で言えばこれで主な英単語を覚えたことになります。ここからは、覚えた単語を使って会話を行うフェーズになります。本章では、実際のプログラムを並列化する方法について説明します。

4-1　パフォーマンスに関する考慮点

例えばノード数 8 台で並列プログラムを実行した場合、1 台の場合の 8 倍とは言わないまでも、7 倍程度の速度向上が得られるのではないかと考えがちです。勿論 8 倍近く出るプログラムもありますが、台数を増やせば増やすほど却って遅くなってしまうプログラムもあります。この差は一体どこから来るのでしょうか。

4-1-1　アムダールの法則

図 4-1-1 で、単体プログラムを実行した場合の経過時間（の割合）を 100 (%) とします。このプログラムの P (%) (本例では 75%) の部分が並列化可能であるとします (これを並列化率が P (%) であると言います。N 台 (本例では 3 台) で並列に実行すると、並列化されていない $100 - P$ (%) の部分はそのまま残り、並列化された P (%) の部分は N 台で実行して P/N (%) となるので、合計 $(100 - P) + P/N$ (%) となります。1 台の経過時間と比べて N 台で何倍速くなったかを表す比率を速度向上率と呼び、本例では 3 台で並列に実行して速度向上率は 2 倍となります。

これを一般化すると以下のようになり、アムダールの法則と呼びます。ただし、この式および以下の説明では、並列化に伴う通信などのオーバーヘッドは考慮に入っていません。

```
並列化率P(%)のプログラムをノード数N台で実行したとき、
  速度向上率(倍) ＝ 100          ･･･ 1台での経過時間(%)
                 (100-P)+P/N     ･･･ N台での経過時間(%)
```

図 4-1-1

P:並列化率(%)	50	60	70	80	90	95	100
ノード数 2台	1.33	1.43	1.54	1.67	1.82	1.90	2.0
ノード数 8台	1.78	2.1	2.58	3.33	4.7	5.93	8.0
ノード数∞台	2.0	2.5	3.33	5.0	10.0	20.0	∞

図 4-1-2 (1)

図 4-1-2 (2)

ノード数が 2 台、8 台、∞ 台の場合の速度向上率を図 4-1-2 (1) に、グラフを図 4-1-2 (2) に示します。

- ノード数が 8 台の（多い）場合と 2 台の（少ない）場合を比較してみます。8 台の場合、図 4-1-2 (1) から分かるように、並列化率が 70% 程度だと速度向上率は 2.58 倍にしかならず、95% でようやく 5.93 倍になります。つまりノード数が多い場合、並列化率がかなり高くないと並列化の効果は得られません。
  一方ノード数が 2 台の場合は、並列化率が 70% 程度でも 1.54 倍となります。つまりノード数が少ない場合、並列化率が多少低くても、ある程度並列化の効果が得られます。
  別の見方として、並列化率が 70% の場合、ノードを 1 台から 1 台増やしただけで 1.54 倍になりますが、そこからさらに 6 台増やしても 2.58 倍にしかならないと言うこともできます。
- アムダールの法則でノード数 N を ∞ 台にすると、図 4-1-2 (3) に示すように並列化可能な部分はほぼゼロになりますが（ただし実際は通信などのオーバーヘッドが加わります）、並列化不可能な部分が残ってしまいます。例えば並列化率が 80% の場合、台数をいくら増やしても速度向上率は 5 倍以上にはなりません。すなわち図 4-1-2 (1) (2) で ∞ 台の場合の速度向上率は、各並列化率での速度向上率の上限を意味します。

図 4-1-2 (3)

4-1-2　パフォーマンスを向上させるための考慮点

速度向上率についてもう少し細かく検討してみます。例えば図 4-1-3 (1) のようにノード数が 4 台の場合を考えます。並列化可能な 80% の部分が並列化されますが、この例では 80 ÷ 4 の 20% にならず、1 台目のノードは 15%、4 台目のノードは 25% などと、計算時間がバラついてしまいました。並列ジョブでは、このようにバラつきがある場合、一番遅いプロセスに足をひっぱられてしまいます。言い換えると、処理を行わずに遊んでいるプロセスがあるので、計算時間が遅くなります。

これに加え、並列化にともなう通信時間が加わります。その結果、図 4-1-3 の例では、4 台で 2 倍の速度向上率しか得ることができませんでした。

図 4-1-3 (1)

以上のことから、速度向上率を上げるためには、以下の点に考慮する必要があることが分かります。

(1) (ノード数が多い場合) 並列化率をなるべく高くする。
(2) プロセス間のロードバランスをなるべく均等にする。
(3) 通信時間をなるべく少なくする。

- (1) はプログラムのアルゴリズムの問題であり、本質的に並列性のないプログラムを並列化することは不可能ですが、アルゴリズムによっては並列性を持つアルゴリズムに変更できる場合もあります。
  例えば差分法で使用される SOR 法は、そのままでは計算に依存関係があるため並列化できませんが、ゼブラなどの方法に変更すると、依存関係がなくなり並列化できるようになります (5-6 節参照)。また、一次逐次演算 (4-6-9 節) も同様です。
- (2) に関しては、ブロック分割 (DO ループの反復を例えばプロセス数が 4 ならば 1/4 ずつ分割) で分割すれば、多くの場合ロードバランスは均等になります。ブロック分割でロードバランスが均等にならない（ロードインバランスと言います）例と対処方法については 4-5-3 節で説明します。
- (3) がもっとも注意すべき項目です。分散メモリー型並列計算機での並列化はまさに通信時間との戦いです。並列化を行うときは通信時間をいかにして短縮するかを常に念頭に置く必要があります。以下、通信時間を短縮するための考慮点について述べます。
- 余談ですが、プログラムによっては並列化すると CPU 台数以上の速度向上率 (例えば 3 台で 3 倍以上) が出ることがあり、これをスーパーリニアと呼ぶことがあります。この理由ですが、単体プログラムで図 4-1-3 (2) の配列内の要素を数字の順に処理するとします。このようにメモリー内の要素を飛び飛びに処理すると、スカラー計算機（ワークステーションやパソコン）の場合はキャッシュミスが発生して速度が低下します (参考文献 [6] の 4 章参照)。これを並列化した図 4-1-3 (3) でもメモリー内の要素を飛び飛びに処理しますが、1 プロセス当たりの担当する範囲が狭くなり、前に処理した要素の近隣の要素がキャッシュに残っている可能性が高くなるため、キャッシュミスによる速度低下の影響が少なくなります。このような現象は、計算領域が非構造格子から構成されていて間接アドレスを使ったプログラムなどで現れます。

図 4-1-3 (2)

図 4-1-3 (3)

通信の特性

通信の特性は一般に次のようになっています（ただしマシン環境によって異なります）。

- 図 4-1-4 に示すように、通信時間は通信バイト数に比例します。すなわち、通信バイト数が 2 倍になれば通信時間も 2 倍になります。
- 図 4-1-4 に示すように、一回通信を行うごとに一定の立上り時間 (latency) がかかります。
- 並列計算機内の各ノード間の通信時間はほぼ同じであるマシンが一般的ですが、これはマシン環境に依存します。

図 4-1-4

通信に関する考慮点 (1)

以上の通信特性から、ユーザーは次の 2 点に考慮する必要があります。

(1) 同じ通信量なら通信回数を減らす。

例えば要素数が 100 個のデータを 1 度に送信すれば、立上り時間は 1 回で済みますが、これを 1 要素ずつ 100 回送信すると、立上り時間も 100 回分かかってしまいますので、なるべく通信回数を少なくして下さい。

では、このような状態はどのような場合に発生するのでしょうか？　例えば図 4-1-5 のように、配列 A を 1 次元目 (I 方向) で分割しており、ランク 0 のプロセスは斜線の 1 行を他のプロセスに送信する必要があるとします。この場合、3-1 節「付録について」で説明したように、この配列はメモリー上で図の縦方向に連続しています (図の ↓ 参照)。従って、斜線の部分は図 4-1-5 では連続して見えますが、実際には各要素はメモリー上でとびとびになっています。

この場合、(a) のように 1 度に 1 要素ずつ 100 回送信すると、100 回分の立ち上がり時間がかかってしまいます。これに対して、(b) のように送信したい要素をいったん連続した送信バッファー（配列）に入れ、その後で一気に送信すれば、立ち上がり時間は 1 回で済むわけです。この場合、もちろん要素を送信バッファーに詰め込むオーバーヘッドはかかりますが、通信というのは I/O のようなものなので、それに比べると CPU が行う詰め込むための時間は（一般に）無視することができます。

図 4-1-5

なお、MPI で提供されている派生データ型 (3-5 節参照) を使用すれば、図 4-1-5 のようにメモリー上で規則的に離れて配置されているデータは直接送信できますので、通信回数は 1 度で、さらに送信バッファーに詰め込む作業も不要になります（ただしパフォーマンスが良いかどうかはマシン環境に依存します)。

初心者が並列化したプログラムを見ると、図 4-1-6 (1) のように、DO ループ内でメッセージ交換を行っていることがあります。この例では 10000 回「CALL MPI_BCAST」を実行するため、10000 回分の立ち上がり時間がかかってしまいます。従って、メッセージ交換は DO ループ内では行わず、図 4-1-6 (2) のように DO ループの外で一度だけ行うのが普通です。

```
  :
DO I=1,10000
  計算
  CALL MPI_BCAST(A(I),～)
ENDDO
  :
```

図 4-1-6 (1)　×

```
  :
DO I=1,10000
  計算
ENDDO
CALL MPI_BCAST(A,～)
  :
```

図 4-1-6 (2)　○

通信に関する考慮点 (2)

次に 2 つ目の考慮点を以下に示します。

(2) 通信量自体が少なくなるようにする（配列の分割方法に注意)。

通信を行う必要が生じた場合でも、並列化の方法によっては通信量を少なくできる場合があります。差分法では、あるメッシュの値を、前後左右のメッシュの値から求める演算が発生します。図 4-1-7 (1) では長方形の計算領域を 1 次元目で分割しています。ランク 0 のプロセスが1の計算をする場合、前後左右の●はすべて自分が持っているので問題ありません。ところが2の計算をする場合、■は持っていますが、□はランク 1 のプロセスが担当するので持っていません。従って、計算を行う前に、斜線の 1 行のデータをランク 1 のプロセスから送ってもらう必要があります。

この場合、図 4-1-7 (2) のように計算領域を分割することもできます。両者を比較すると、図 4-1-7 (2) の方が境界の長さが短いため、通信量も少なくなります。このように、計算領域が長方形 (2 次元) や直方体 (3 次元) の場合、どの次元で分割すれば通信量が少なくなるかを考慮に入れる必要があります。これについては 4-5-8 節で説明します。

図 4-1-7 (1)

図 4-1-7 (2)

計算量と通信量の関係

並列化の効果を見積もる一つの目安を紹介します。図 4-1-8 (1) で各プロセスは一辺が n のオーダーの領域を計算し、その後自分が計算した部分を他のプロセスに通信したとします。この場合、大ざっぱに言うと計算量は αn^2、通信量は βn^2 とどちらも n^2 のオーダーになります。計算に比べると、通信というのは I/O のようなものですから、係数は $\alpha \ll \beta$ になります。従って、(通信量)/(計算量) $= \beta/\alpha \gg 1$ となりますので、図 4-1-8 (1) のように計算量と通信量が同じ次数の場合、よほど計算部分が多くないと通信量の割合が多くなって並列化の効果が出ません。

一方、図 4-1-8 (2) のように、境界の 1 列のみを通信する場合、通信量は βn とオーダーが n になるので、(通信量)/(計算量) $= \beta/\alpha n$ となり、特に n が大きくなるほど通信量の割合は減少します。

以上のことから、一般に、計算量より通信量が少なくとも次数が 1 つ以上少なくなる部分を並列化すれば、特に問題のサイズが大きいほど通信量の割合が減少し、並列化の効果が出ると言うことができます。

図 4-1-8 (1)　　図 4-1-8 (2)

ノード数を増やすとなぜサチュレートするのか

大部分の並列プログラムでは、ノード数を増やしていくとどこかでサチュレートし、それ以上はノードを増やすとかえって遅くなってしまいます。なぜこのようなことが起こるのでしょうか？　これを以下のグラフで示します。なお、この例は図 4-1-2 と同じく並列化可能部分が 80% の例です。

まず並列化できない部分に関しては、ノード数を何台にしようとも、20% のまま残ります。次に並列化可能部分ですが、これはプロセス間のロードバランスが均一であれば、2 台でほぼ 1/2、4 台なら 1/4、…のように減少していきます。

問題なのは通信時間で、多くの並列プログラムでは、図 4-1-3 のようにノード数が増えるにつれて通信時間が増加します(そうでないプログラムもあります)。このため、ノード数を増やしていくと、並列化可能部分の CPU 時間の減少よりも通信時間の増加の方がまさってしまいあるノード数で経過時間が最短となり、それ以上ノード数を増やすと逆に遅くなってしまいます。以下の例では、8 台 → 16 台では経過時間はわずかに減少していますが、16 台で最短になり、それ以上はかえって遅くなっています。

なお、ノード数を増やして通信時間が増加する割合は、何をどのように通信するかによって異なります。プログラムによってはほとんど増加しなかったり、逆にかえって減少する場合もあります。

4-2　どんなところが並列化できるのか

並列性とは何か

プログラムの並列化を行う前に、そもそもMPIを使用した分散メモリー型並列計算機（以後並列計算機）ではどのようなところが並列化できるのでしょうか？　これをまず知っておく必要があります。

並列計算機では、並列性さえあればどんなところでも並列化することができます。

それでは並列性とは一体何なのでしょうか？　私は厳密な定義は知りませんが、DOループの並列性について説明してみます。

DOループの各反復間に依存性がなく独立に実行できる場合、並列性があると言います。依存関係がない場合は、ループのある反復で更新された変数（左辺の変数）が、それ以後の反復の右辺で参照されません。例えば、図4-2-1 (1) のループを展開すると図4-2-1 (2) のようになりますが、更新された値A(1) は以後の右辺で参照されません。従って、各ステートメントは完全に独立しており、どのような順序で実行してもかまわないわけです。このような場合は、例えば4-8 (2) のように、3つのプロセスが2ステートメントずつ計算して並列化を行うことができます。

これに対し、図4-2-2 (1) を展開した図4-2-2 (2) では、更新されたA(1) は次の反復の右辺で参照されているため、必ず図4-2-2 (2) の順序でステートメントを実行する必要があります。これを、このループには回帰参照がある、あるいは依存関係があると呼ぶこともあります。この場合、図4-2-2 (2) のように並列化すると、例えばランク1のプロセスはランク0のプロセスが計算するA(2) の値が確定する前にA(2) の値を参照してしまい、結果がおかしくなってしまいます。従って、DOループ内に回帰参照がある場合には、原則として並列化することはできません (ただし図4-2-2 (1) のループは、特別な方法で並列化することができます (4-6-9節参照))。

なお、合計を求める『S = S + A(I)』や最大値を求める『X = MAX(X,A(I))』のような縮約演算は、回帰参照がありますが並列化することができます (4-6-4節参照)。

```
   :
DO I = 1, 6
  A(I) = B(I) + C(I)
ENDDO
   :
```

図4-2-1 (1)

```
   :
DO I = 1, 6
  A(I) = A(I-1) + B(I)
ENDDO
   :
```

図4-2-2 (1)

図4-2-1 (2)

図4-2-2 (2)

並列計算機では、各ループに並列性があるかどうかをチェックするツールなどは提供されていませんので、ユーザーが次のような観点で自分で判断する必要があります。なお、実際のプログラムでは、並列性があるかどうかがわかりにくい DO ループはめったにありません。

- 図 4-2-2 (1) のように、ループ内のステートメントの左辺に A(I) がある場合、右辺に A(I-1) がないかチェックして下さい。もしある場合には回帰参照があるため、通常は並列化できません。
- ループに回帰参照があるかあるかどうか分からない場合、図 4-2-1 (2) のようにループ内のステートメントを展開して書き下したり、配列間のデータの動きを図示して調べるのも一つの方法です。
- 極めて邪道な方法ですが、単体版でループの反復順序を逆にして (例：DO I=10,1,-1)、結果が合うかどうかを調べるという方法もあります。もしループの各反復間に依存関係があれば、ループの反復順序を逆にした場合は通常結果がおかしくなります。ただしこの調査方法は完全ではありませんので注意して下さい。
- DO ループの一部に回帰参照のあるステートメントがある場合、そのループ全体が並列化できなくなるのを避けるため、以下のように回帰参照のある部分だけ別のループにして、元のループを並列化するという方法もあります。

図 4-2-3 (1)

→

図 4-2-3 (2)

どんなところが並列化できるのか

本節の冒頭で、並列計算機では、並列性さえあればどんなところでも並列化することができると述べました。その例を以下に示し、ベクトル計算機（スーパーコンピュータ）の場合と比較してみます。なお、ベクトル計算機では、コンパイラが DO ループを自動的にベクトル化します。また、ベクトル化とは並列化の一種であると考えて下さい。

(1)　多重ループ

ベクトル計算機では、原則として多重ループの内側のループしかベクトル化できません（ただしコンパイラがループを入れ替えることがあります）。並列計算機では、並列化とは単に計算量を $1/n$ に減少させることなので、図 4-2-4 (2) (3) に示すように、内側のループでも外側のループでも並列性さえあれば並列化が可能です。

図 4-2-4 (1)

→

図 4-2-4 (2)

図 4-2-4 (3)

(2)　サブルーチンコールを含む DO ループ

ベクトルコンパイラは、各ループがベクトル化できるかどうかを解析しますが、DO ループにサブルーチンコールを含んでいる場合、解析が複雑となるためベクトル化を行いません。一方並列計算機の場合、図 4-2-5 (2) のように、並列性さえあれば並列化が可能です。

```
   :
DO I = 1, 100
  CALL SUB
ENDDO
   :
```

図 4-2-5 (1)

→

```
   :
ISTA =
IEND =
DO I = ISTA, IEND
  CALL SUB
ENDDO
   :
```

図 4-2-5 (2)

(3)　I/O を含む DO ループ

ベクトル計算機の場合、I/O 命令を含む DO ループはベクトル化できません (ただし、I/O 命令を DO ループの外に出して、その他の演算をベクトル化することがあります)。一方並列計算機の場合、図 4-2-6 (2) に示すように、並列性さえあれば、各プロセスが自分のローカルディスクに対して I/O を行うことによって、I/O そのものを並列化することも可能です。

```
   :
DO I = 1, 100
  WRITE (10,*)
ENDDO
   :
```

図 4-2-6 (1)

→

```
   :
ISTA =
IEND =
DO I = ISTA, IEND
  WRITE (10+MYRANK,*)
ENDDO
   :
```

図 4-2-6 (2)

(4)　ブロック

図 4-2-7 (1) で処理 1〜処理 3 が独立に実行できるとします。ベクトル計算機では、DO ループの形態になっていないとベクトル化できないので、図 4-2-7 (1) はベクトル化できません。一方並列計算機の場合、図 4-2-7 (2) のようにすれば並列化は可能になります。この例として、3 本の偏微分方程式（実際には連立一次方程式）を解くプログラムがあり、それぞれ独立に解くことができるのですが、連立一次方程式の解法自体には並列性がないとします。この場合、図 4-2-7 (2) のようにすれば一応並列化することができます (ただし 3 ノードですが)。

なお、処理 1〜処理 3 が独立かどうかがわからない場合は、処理の順序を例えば処理 3 → 処理 2 → 処理 1 のように変えてみて、結果が合うかどうかを調べるのも（邪道ですが）1 つの方法です。

```
   :
処理1
処理2
処理3
   :
```

図 4-2-7 (1)

→

```
   :
IF (MYRANK == 0) THEN
  処理1
ELSEIF (MYRANK == 1) THEN
  処理2
ELSEIF (MYRANK == 2) THEN
  処理3
ENDIF
   :
```

図 4-2-7 (2)

4-3　計算部分の並列化パターン

前節に説明したように、並列計算機の場合は並列性さえあればどんなところでも並列化は可能であり、ベクトル計算機と比較して一見いいことずくめのように見えます。しかし、どこでも並列化できるということは、裏返すと、どこを並列化するのかをユーザーが決定しなければならないということになります。

逆にベクトル計算機の場合は、基本的に多重ループの内側の DO ループしかベクトル化できないかわりに、ほぼ自動的にコンパイラがベクトル化を行ってくれますので、ユーザーがいちいちベクトル化に関する方針を決定する必要はありません。ただし、ホットスポット（計算の集中している部分）がベクトル化されていなかったり、ベクトル化されていても効果が出ていない場合は、ベクトルチューニングを行う必要があります。

並列計算機に話を戻すと、実は、プログラムのどこを並列化するかという選択が、並列化が成功するかどうかの最大のポイントになります。この選択を誤ると、いくら頑張っても並列化による効果は得られません。そして、これは各プログラムに固有の話になるため、ユーザーが自分で考えなくてはなりません。

とはいえ、巷のプログラムも並列化という観点で言えばそうそう無限のバリエーションがあるわけではなく、マクロに見ると多くのプログラムは、これから述べるいくつかのパターンかその組み合わせで並列化できることが分かっています。

計算パターン 1

プログラム全体のうち、極めて限られた部分がホットスポット（計算の集中している部分）になっている場合があります。例えば図 4-3-1 (1) のようにタイムステップのループの中に①、②、③があり、このうち②の部分が計算時間のほとんどを占め、①と③の部分はステップ数は多いのですが計算時間は短いとします。また、②では配列 A の値を求めることが最終目的であるとします。言いかえると、②で更新された配列のうち配列 A は①や③で参照または更新が行われますが、配列 W は配列 A の計算を行うための作業配列で、②のみで使用され、①や③では参照や更新が行われないとします。

この場合、①と③の部分を並列化しても、ステップ数が多いため修正の手間はかかりますが、計算時間がかかっていないためパフォーマンスの向上は期待できません。このような場合、②の部分だけを並列化し、①と③の部分はブラックボックスとして扱います。この場合の動作を以下で説明します。

- ②の DO ループの反復を図 4-3-1 (2) の⑤のように修正します。これによって⑤の部分が並列化、すなわち計算量が $1/n$ に縮小されました。
- ⑤で最終的に求めたいのは配列 A なので、⑤が終了した直後の⑥で、各プロセスは、配列 A のうち自分が計算した部分のデータを、他の全プロセスに送信します (図 4-3-1 (4) 参照)。つまりお互いがお互いにデータをやり取りします。この通信はメッセージ交換ルーチンの MPI_ALLGATHER(V) に相当します (4-6-3-2 節参照)。
- ⑥の通信が終了した時点で、各プロセスは単体で実行したのと (すなわち②の終了時と) 全く同じ状態になります。正確にいうと配列 W の中身は異なっていますが、前述のように配列 W は②のみでしか使用されないので問題ありません。
- 次に⑦、④で全プロセスが単体での③、①と全く同じ動作をします。
- 再び⑤に来ますが、⑦と④で全プロセスが単体と同じ動作をしたため、この時点で全プロセスは計算に必要な全データを持っています。従って、⑤に入る前にデータの送信を行う必要はありません。

このパターンは、並列化のための修正作業が少なくて済むという長所があります。一方、4-1 節の最後で説明しましたが、例えば図 4-3-1 (4) で計算量と通信量のオーダーはどちらも n で次数が同じなので、通常は並列化すると却って遅くなってしまいます。しかし筆者の経験では、②で計算時間のほとんどを占めるようなプログラムの場合、②の中にもう 1 重ループが存在して計算量のオーダーが n^2 になっていたり、②で行う計算が非常に複雑だったりして、⑥のように通信時間のかかる派手な通信をしても並列化の効果が出る場合が多いようです。ただしノード数が増えると、次第に通信の割合が増えてサチュレートしてしまいます。

```
   :
DO タイムステップ
  ブラックボックス                ①
  DO I = 1, 6
     :
    W(I) = ～                     ②
    A(I) = W(I)+ ～
  ENDDO
  ブラックボックス                ③
  ～ = A(I) + ～
ENDDO
   :
```

図 4-3-1 (1)

```
   :
DO タイムステップ
  ブラックボックス                ④
  DO I = ISTA, IEND
     :
    W(I) = ～                     ⑤
    A(I) = W(I)+ ～
  ENDDO
  CALL MPI_ALLGATHER(A)           ⑥
  ブラックボックス                ⑦
  ～ = A(I) + ～
ENDDO
   :
```

図 4-3-1 (2)

図 4-3-1 (3)　　図 4-3-1 (4)

計算パターン 2

次のパターンは、同じような多くの DO ループがあり、ホットスポットがそれらの各ループに分散している、言いかえると顕著なホットスポットがない場合です。このようなパターンは差分法のプログラムでよく現れます。

この例を図 4-3-2 (1) に示します。タイムステップのループの中に①〜③のループがありますが、同じようなパターンで計算量にそれほど差がありません。従って、並列化の効果を得るためには一部のループだけでなく、タイムステップ内のほとんど全てのループを並列化する必要があります。

また、通常このパターンでは、並列化したループが終了した直後に、「計算パターン 1」のような派手な通信を行うと、通信量の方がまさってしまい、並列化の効果が出ません。

以上のことから、このパターンの場合、次の方針で並列化を行います。

(1) タイムステップのループの中にあるループは基本的に全て並列化します。

(2) タイムステップのループが反復する間、各プロセスは自分の担当するデータのみを保持し (図 4-3-2 (4))、通信が必要な場合も「計算パターン 1」のような通信ではなく、必要最小限の通信のみを行ないます。

図 4-3-2 を並列化した場合の動作を説明します。なお、この例は 1 次元の差分法（モドキ）であり、図中の『境』は固定境界、すなわちこの部分の値は変わらないことを意味します。

- ①のループ反復を④のように修正します。これによって④が並列化、すなわち計算量が $1/n$ に縮小されました。
- ②のループの反復を⑥のように修正します。⑥では配列 A を求めるのに、配列 B の自分の前後の値を使用しています。従って⑥に入る前に、自分の担当部分の境界を 1 つ越えたデータを隣のプロセスから送信してもらう必要があります。この通信を⑤に示す CALL SHIFT (4-6-2 節と 5-1 節参照) で行います。
  なお、この例では、タイムステップ・ループを一周する間に配列 B の値が (④で) 更新されているので⑤の通信が必要になりますが、配列 B が不変の値（例えば座標値など）で、タイムステップを一周しても値が更新されないのであれば、⑤の通信を行う必要はありません。
- ③のループの反復を⑦のように修正します。

このパターンは、タイムステップ・ループ内のほとんど全てのループを並列化することと、並列化に伴う通信を必要最小限しか行わないことから、通常は並列化の効果が出ます。

ところで 1-1 節で簡単に触れましたが、共有メモリー型並列計算機の場合、共有メモリー型並列計算機用に並列化したプログラム (コンパイラによる自動並列化、または OpenMP の指示行による並列化：4-7-2 節参照) の他に、マシン環境によっては MPI で並列化したプログラムも実行できる場合があります。差分法のようなパターン 2 のプログラムの場合、どちらで並列化するのがよいか検討してみます。

パターン 2 のプログラムは通常各 DO ループの構造が簡単なので、コンパイラによる自動並列化、または OpenMP の指示行で、比較的簡単に並列化することができます。一方上記で説明した MPI による並列化の場合、特にプログラムの規模が大きいと、並列化に伴う修正箇所が多くなるため、ワークロードがかかってしまいます。

従って、並列化するワークロードの観点では、共有メモリー型並列計算機用の並列化に軍配が上がります。ただしどちらがパフォーマンスが良いかは計算機の特性に依存します。

```
    :
DO タイムステップ
  DO I = 1, 6
    B(I) = B(I) + A(I)        ①
  ENDDO
  DO I = 1, 6
    A(I) = B(I-1)+B(I+1)      ②
  ENDDO
  DO I = 1, 6
    A(I) = A(I) + 1.0         ③
  ENDDO
ENDDO
    :
```

図 4-3-2 (1)

```
    :
DO タイムステップ
  DO I = ISTA, IEND
    B(I) = B(I) + A(I)        ④
  ENDDO
  CALL SHIFT(B)               ⑤
  DO I = ISTA, IEND
    A(I) = B(I-1)+B(I+1)      ⑥
  ENDDO
  DO I = ISTA, IEND
    A(I) = A(I) + 1.0         ⑦
  ENDDO
ENDDO
    :
```

図 4-3-2 (2)

図 4-3-2 (3)

図 4-3-2 (4)

計算パターン 3

プログラムは一般に、図 4-3-3 (1) のようにメインルーチンから末端のサブルーチンまで連綿と続く階層構造になっています。図 4-3-3 (1) の場合、■のついている末端の DO ループを図 4-3-3 (2) のように並列化することができます（「並」は並列化されたループ）。前述の計算パターン 1 や計算パターン 2、あるいはベクトル計算機でのベクトル化がこれに相当します。ところで図 4-3-3 (1) では、上位の●の部分にも並列性があり、図 4-3-3 (3) のように並列化することができます。本書では、図 4-3-3 (2) のような並列化を末端の部分での並列化、図 4-3-3 (3) のような並列化を上位の部分での並列化と呼ぶことにします。なお、前者を粒度が小さい部分での並列化、後者を粒度が大きい部分での並列化と言う場合もあります。

図 4-3-3 (1)

末端の部分のみで並列化できるプログラムもあれば、末端でも上位でも並列化できたり、あるいは末端では並列化できずに上位だけで並列化できるプログラムもあります。それでは末端でも上位でも並列化できる場合、一般にどちらで並列化するのがよいのでしょうか？　これはもちろんプログラム自体に依存しますが、図 4-3-3 (2) と (3) を比較すると次のことが分かります。

- 実線と二重線に示すように、上位の部分で並列化した場合は並列化のための修正量は少ないですが、末端の部分で並列化した場合、各ループを並列化しなければならないため、修正量が多くなります。
- 二重線に示すように、上位の部分で並列化した場合は計算の最後に 1 回のみ通信を行いますが、末端の部分で並列化した場合は何度も通信する必要があり、通信の立上り時間がかかります。

以上のことから、どこを並列化するかを検討する際、末端の部分で並列化できる場合でも、上位の部分で並列化できないかどうか必ず検討して下さい。上位のループに並列性があるかどうかわからない場合、4-2 節で述べた邪道な方法として、ループの反復順序を逆にして調べるという手もあります。なお、タイムステップや収束反復のループには並列性はありません。

『上位の部分と末端の部分の両方で並列化することはできないのか？』という質問が出ることがありますが、上位の部分で並列化すれば末端では並列化する必要はなく、その逆も同様です。

上位の部分で並列化できるプログラムとして、次のような例があります。

- 連立一次方程式を例えばガウスの消去法で解いているプログラムで、解くべき連立一次方程式自体が 100 本あり、各連立一次方程式が独立に解けるとします。この場合、1 回のガウスの消去法自体を並列化するのが「末端の部分の並列化」にあたります。一方例えば 4 プロセスの場合、各プロセスに 25 本ずつ連立一次方程式を割り当てるのが「上位の部分の並列化」になります。

● 例えば流体の計算を行うプログラムで、境界の流速を (時速 1 m, 2 m, ...) と変えて計算し、それぞれの条件で得られた物理量を最後にまとめるようなプログラムがあるとします。通常は前述の『計算パターン 2』で末端の部分の並列化を行いますが、各流速での計算を独立に行うことができるのであれば、上位の部分で並列化を行うことができます。もっともこの例などは、例えば単体ジョブを 4 本同時に実行し、各ジョブは 100 個の境界の流速のデータのうち 25 個のデータのみを処理するのとほとんど同じであり、並列ジョブとは言えないかもしれませんが…。

図 4-3-3 (2)

図 4-3-3 (3)

4-4　並列化に伴う入出力部分の修正

計算部分の並列化に伴い、入出力部分の修正が必要になります。本節では修正方法について説明します。なお、ファイルは順次編成ファイルを想定します。

4-4-1　並列化に伴う入力部分の修正

まず入力部分 (READ 文) の修正方法について説明します。

標準入力

MPI で並列化したプログラムの場合、標準入力 (装置番号 5 番) から読み込みを行うと、マシン環境によってはエラーになる場合があるのでお勧めできません。標準入力以外の装置番号に変更し、以下に示す他の入力パターンで読み込みを行うようにして下さい。なおマシン環境によっては、標準入力から読み込むための環境変数などが提供されている場合もありますが、並列化したプログラムを他のマシン環境へ移植する可能性がある場合は、やはり標準入力以外の装置番号に変更した方が良いでしょう。

入力パターン 1

図 4-4-1 (1) に示すように、「0 1 2」の入った入力ファイルが共有ディスクに置かれていて、各プロセスが「0 1 2」の全てのデータを使用したい場合（★の付いたデータのみを使用したい場合も同様)、図 4-4-1 (3) の READ 文（標準入力以外）を何も修正せずに実行すると、各プロセスが「0 1 2」を読み込みます。図中の10は装置番号を示します。

図 4-4-1 (1) (3) の方法はプログラムを修正する必要がないので簡単ですが、以下の欠点があります。

- 入力ファイルが共有ディスクにあるので、読み込む際にプロセス間で競合が発生する。
- 読み込んだデータを共有ディスクのファイルサーバーから各ノードに通信するための通信時間がかかる。

一方図 4-4-1 (2) では、あらかじめ、入力ファイルを rcp コマンドなどで各ノードのローカルディスクにコピーし、各プロセスは自分のローカルディスクからデータを読み込むので、読み込み時間が短縮されます。

大量の入力データを読み込まなければならなかったり、プロセス数が非常に多くて読み込みの競合が異常に発生して入力データを読み込む時間が無視できないほど長い場合は、図 4-4-1 (2) の方がよいかもしれません。ただしファイルをあらかじめローカルディスクにコピーする手間がかかります。

図 4-4-1 (1)

図 4-4-1 (2)

```
REAL X(0:2)
  :
READ(10) (X(I),I=0,2)
  :
```

図 4-4-1 (3)

入力パターン 2

図 4-4-2 (1) (2) に示すように、例えばランク 0 のプロセスが代表して入力データを読み込み、その全てを MPI_BCAST で他のプロセスに送信する（細い実線部分）という方法もあります。この方法は通信のオーバーヘッドが発生し、またプログラムによっては、たくさんの配列を送る必要があるため修正が面倒になるという欠点があります。なお、市販の並列プログラムでは、この方式になっているプログラムが多いようです。

各プロセスが全入力データの一部しか使用しないのであれば、図 4-4-2 (3) (4) のように、例えばランク 0 のプロセスが、読み込んだデータのうち各プロセスが必要とする部分のみを、MPI_SCATTER などを使用して送信する（細い実線部分）という方法もあります。なお、MPI_SCATTER を使用した場合、配列 X とは別に配列 Y が必要となりますが、配列 Y を使用しない方法もあります (3-3-5 節、3-8-1 節、4-6-3 節参照)。

図 4-4-2 (1)

```
 REAL X(0:2)
   :
 IF (MYRANK==0) READ(10) (X(I),I=0,2)
 CALL MPI_BCAST(X,3,MPI_REAL,
&               0,MPI_COMM_WORLD,IERR)
```

図 4-4-2 (2)

図 4-4-2 (3)

```
 REAL X(0:2),Y(0:2)
   :
 IF (MYRANK==0) READ(10) (X(I),I=0,2)
 CALL MPI_SCATTER(X        ,1,MPI_REAL,
&                 Y(MYRANK),1,MPI_REAL,
&                 0,MPI_COMM_WORLD,IERR)
```

図 4-4-2 (4)

図 4-4-2 (5) に示すように、入力データを読み込んで、エラーを発見したらエラーメッセージを表示して処理を終了する単体プログラムを並列化し、図 4-4-2 (6) のようにランク 0 が代表して入力データを読み込むようにした場合、ランク 0 はエラーを発見し、STOP 文で終了しますが、他のプロセスは①でランク 0 から送られるデータの受信待ちになり、終了しません。全プロセスを終了させたい場合は、図 4-4-2 (7) に示すように MPI_ABORT（付録参照）を使用します。「入力パターン 1」の場合は、図 4-4-2 (8) に示すように全プロセスが入力データを読み込んでエラーを発見し、STOP 文で終了するので、MPI_ABORT を実行する必要はありません。

```
 READ(10,*) A
 IF (ｴﾗｰ) THEN
   PRINT *,'==ERROR=='
   STOP
 ENDIF
   :
```

図 4-4-2 (5)　単体プログラム

```
   :
 IF (MYRANK==0) THEN
   READ(10,*) A
   IF (ｴﾗｰ) THEN
     PRINT *,'==ERROR=='
     CALL MPI_FINALIZE(～)
     STOP
   ENDIF
 ENDIF
 CALL MPI_BCAST(A,～)    ①
   :
```

図 4-4-2 (6)　×

```
   :
 IF (MYRANK==0) THEN
   READ(10,*) A
   IF (ｴﾗｰ) THEN
     PRINT *,'==ERROR=='
     CALL MPI_ABORT
&        (MPI_COMM_WORLD,9,IERR)
     CALL MPI_FINALIZE(～)
     STOP
   ENDIF
 ENDIF
 CALL MPI_BCAST(A,～)
   :
```

図 4-4-2 (7)　○

```
 READ(10,*) A
 IF (ｴﾗｰ) THEN
   IF (MYRANK==0)
&    PRINT *,'==ERROR=='
   CALL MPI_FINALIZE(～)
   STOP
 ENDIF
   :
```

図 4-4-2 (8)　○

入力パターン 3

並列ジョブを実行する前に、図 4-4-3 (1) の [1] に示すように、単体ジョブを実行し、入力ファイルを並列ジョブで各プロセスが担当するデータごとに別ファイルに分けます。分割したファイルを共有ディスクに作成する場合は、下線に示すようにファイルの装置番号をランクの値で区別します。各プロセスのローカルディスクに作成する場合は同一の装置番号で構いません。

次に [2] と図 4-4-3 (2) に示すように、並列ジョブを実行し、各プロセスは自分の担当するデータの入ったファイルから読み込みを行います。なお、ファイルを NFS マウントされた共有ディスクに作成する場合、上記の単体ジョブと並列ジョブを連続に実行すると、[1] で書き出したデータが、[2] を開始した時点ではキャッシュに入っていてファイルに入っておらず、[2] の読み込みで誤動作する可能性がありますので、ファイルにデータが入ったのを確認してから並列ジョブを実行するようにして下さい。

図 4-4-3 (1)

```
REAL X(0:2)
  :
READ(10+MYRANK) X(MYRANK)
  :
```

図 4-4-3 (2)

入力パターン 4

図 4-4-4 (1) で、まずランク 0 が①で自分の担当範囲をファイルから読み込みます。次にランク 0 は②でランク 1 の担当範囲を読み込んでランク 1 に送信し、ランク 1 は③で受信します。最後にランク 0 は④でランク 2 の担当範囲を読み込んでランク 2 に送信し、ランク 2 は⑤で受信します。

プログラム例を図 4-4-4 (2) に示します。

図 4-4-4 (1)

```
REAL X(0:2),WORK
  :
IF (MYRANK==0) THEN
  READ(10) X(0)
  DO IRANK=1,2
    READ(10) WORK
    WORKをランクIRANKに送信する。
  ENDDO
ELSE
  ランク0から受信し、X(MYRANK)に入れる。
ENDIF
  :
```

図 4-4-4 (2)

入力パターン 5

図 4-4-5 (1) で、各プロセスは共有ディスクに作成された入力データのうち、自分が担当する部分のみを読み込み、それ以外の部分は空読みします。バイナリーファイルの場合、以下の (2) と (3) の方法で作成されたファイルは、それぞれ (4) と (5) の方法で読み込みます（空読みする部分を下線で示します）。

図 4-4-5 (1)

```
REAL X(0:2)
  :
WRITE(10) (X(I),I=0,2)
 または
WRITE(10) X
  :
```

図 4-4-5 (2)

```
REAL X(0:2)
  :
DO I=0,2
  WRITE(10) X(I)
ENDDO
  :
```

図 4-4-5 (3)

=

```
REAL X(0:2)
  :                    ↓空読み
READ(10) (DUMMY,I=0,MYRANK-1),
&        (X(I) ,I=MYRANK,MYRANK),
&        (DUMMY,I=MYRANK+1,2)
  :                    ↑空読み
```

図 4-4-5 (4)

=

```
REAL X(0:2)
  :
DO I=0,MYRANK-1
  READ(10) ← 空読み
ENDDO
DO I=MYRANK,MYRANK
  READ(10) X(I)
ENDDO
DO I=MYRANK+1,2
  READ(10) ← 空読み
ENDDO
  :
```

図 4-4-5 (5)

入力パターン 6

3-1 節で紹介した MPI の拡張機能である MPI-2 では、MPI-IO（アイ・オー）という I/O ルーチン群が提供されています。この機能が使用できるかどうかはマシン環境に依存します。本書は MPI-1 の範囲で説明しているので、MPI-IO の説明は省略します。

4-4-2　並列化に伴う出力部分の修正

次に出力部分 (WRITE 文，PRINT 文) の修正方法について説明します。

標準出力

図 4-4-6 (1) のような標準出力 (WRITE(6) または PRINT 文) は、並列に実行すると全プロセスが同じ内容を早い者順に出力します。そこで例えばランク 0 のみが出力するように、図 4-4-6 (2) の下線部を追加します。

並列化のデバッグの際、図 4-4-6 (3) のようにわざと全プロセスに標準出力を書かせることがあります。この場合、出力行数が多かったり各行の文字数が多いと、各行が混ざって表示が乱れることがあります。

プログラムによっては、多くの箇所で標準出力を行っていて、修正が非常に面倒になる場合があります。マシン環境によっては、図 4-4-6 (1) のまま、以下のように書き出すことのできる環境変数などが提供されている場合があります。これは MPI 自体の機能ではなく、マシン環境（並列処理環境）が提供する機能です。

- 標準出力を、ランクごとに別のファイルに書き出す。
- 指定したランクのプロセスのみが標準出力を書き出す。

図 4-4-6 (1)

```
  :
IF (MYRANK==0) WRITE(6,*) '--START--'
  :
```

図 4-4-6 (2)

```
  :
WRITE(6,*) 'MYRANK=',MYRANK,'X=',X
  :
```

図 4-4-6 (3)

出力パターン 1

標準出力以外の出力ファイル（順次編成ファイルを想定）に関しては、図 4-4-7 (1) (2) のように複数のプロセスが共有ディスク上の同じファイルに単に書き出しを行うと、中身がグチャグチャになってしまいます。

この場合、図 4-4-8 (1) (2) のように、各プロセスは自分が計算した部分のデータを、MPI_GATHER(V) を使用して例えばランク 0 のプロセスに送り（細い実線部分)、ランク 0 のプロセスが代表して出力ファイルに書き出します。なお、MPI_GATHER を使用した場合は新たに配列 Y が必要となりますが、配列 Y を使わずランク 0 の配列 X に集める方法もあります (4-6-3-1 節参照)。

また、配列 X を縮小 (4-5-7 節) している場合の出力方法については 4-5-7-3 節で説明します。

図 4-4-7 (1)

```
REAL X(0:2)
  :
WRITE(10) X(MYRANK)
  :
```

図 4-4-7 (2)　×このままでは不可

図 4-4-8 (1)

```
 REAL X(0:2),Y(0:2)
   :
 CALL MPI_GATHER(X(MYRANK),1,MPI_REAL,
&                Y        ,1,MPI_REAL,
&                0,MPI_COMM_WORLD,IERR)
 IF (MYRANK==0) WRITE(10) (Y(I),I=0,2)
   :
```

図 4-4-8 (2)

ランク 0 が代表して出力する方法で誤動作する例

出力パターン 1 のようにランク 0 のプロセスが代表して出力を行う場合、誤動作する例を紹介します。

まず並列化する前の単体プログラムを図 4-4-9 (1) に示します。実行前に 10 番ファイルが存在していない場合、①が失敗し、err=3 に従って③に飛び、④〜⑥を実行して 10 番ファイルを作成します。

このプログラムを並列化し、ランク 0 が代表して出力するように7と8を追加したのが図 4-4-9 (2) です。実行前に 10 番ファイルが存在しない場合、「全プロセスが①を失敗し、err=3 に従って③に飛び、ランク 0 だけが④〜⑥を実行して 10 番ファイルを作成する。」のが正常な動作です。ところが、たまたまランク 0 が他のプロセス (例えばランク 1) より先行していて④〜⑥で 10 番ファイルを作成し、その後でランク 1 が①を実行したとすると、ランク 1 は①が成功して②を実行し、上記の正常な動作と異なってしまいます。

この場合、図 4-4-9 (3) の9に示すように、同期をとるだけの集団通信ルーチン MPI_BARRIER（付録参照）を挿入します。すると、実行前に 10 番ファイルが存在していない場合、ランク 0 がランク 1 より先行しても、ランク 0 は9で (ランク 1 を含む) 全プロセスが9に到着するまで待つので、後から①に到達したランク 1 は①を失敗して②を実行せずに③に飛び、正常に動作します。

```
    :
  OPEN(10,STATUS='OLD',err=3)①
  計算                        ②
3 CONTINUE                    ③
  OPEN(10,STATUS='NEW')       ④
  WRITE(10,*) Y               ⑤
  CLOSE(10)                   ⑥
    :
```

図 4-4-9 (1)

```
    :
  OPEN(10,STATUS='OLD',err=3)①
  計算                        ②
3 CONTINUE                    ③
  IF (MYRANK==0) THEN         7
    OPEN(10,STATUS='NEW')     ④
    WRITE(10,*) Y             ⑤
    CLOSE(10)                 ⑥
  ENDIF                       8
    :
```

図 4-4-9 (2)

```
    :
  OPEN(10,STATUS='OLD',err=3)①
  計算                        ②
3 CONTINUE                    ③
  CALL MPI_BARRIER(〜)        9
  IF (MYRANK==0) THEN         7
    OPEN(10,STATUS='NEW')     ④
    WRITE(10,*) Y             ⑤
    CLOSE(10)                 ⑥
  ENDIF                       8
    :
```

図 4-4-9 (3)

出力パターン 2

この方法は入力パターン 3 の反対です。図 4-4-10 (1) の [1] と図 4-4-10 (2) に示すように、並列ジョブを実行し、各プロセスは自分が計算した部分のデータをファイルに書き出します。ファイルを共有ディスクに作成する場合は、下線に示すようにファイルの装置番号をランクの値で区別します。各プロセスのローカルディスクに作成する場合は同一の装置番号で構いません。

次に [2] に示すように、各プロセスが作成したファイルを、単体ジョブや cat コマンドで 1 つのファイルに合体します。なお、ファイルを NFS マウントされた共有ディスクに作成する場合、上記の並列ジョブと単体ジョブを連続に実行すると、[1] で書き出したデータが、[2] を開始した時点ではキャッシュに入っていてファイルに入っておらず、[2] の読み込みで誤動作する可能性がありますので、ファイルにデータが入ったのを確認してから単体ジョブを実行するようにして下さい。

図 4-4-10 (1)

```
REAL X(0:2)
  :
WRITE(10+MYRANK) X(MYRANK)
  :
```

図 4-4-10 (2)

出力パターン 3

この方法は入力パターン 4 の反対です。図 4-4-11 (1) で、まずランク 0 が①で自分の担当範囲をファイルに書き出します。次にランク 1 が②で自分の担当範囲をランク 0 に送信し、ランク 0 は③でファイルに書き出します。最後にランク 2 が④で自分の担当範囲をランク 0 に送信し、ランク 0 は⑤でファイルに書き出します。プログラム例を図 4-4-11 (2) に示します。詳細は 4-5-7-3 節を参照して下さい。

図 4-4-11 (1)

```
REAL X(0:2),WORK
  :
IF (MYRANK==0) THEN
  WRITE(10) X(0)
  DO IRANK=1,2
    ランクIRANKから受信し、WORKに入れる。
    WRITE(10) WORK
  ENDDO
ELSE
  X(MYRANK)をランク0に送信する。
ENDIF
  :
```

図 4-4-11 (2)

出力パターン 4

入力パターン 6 で説明したように、MPI-2 では、MPI-IO（アイ・オー）という I/O ルーチン群が提供されています。この機能が使用できるかどうかはマシン環境に依存します。本書は MPI-1 の範囲で説明しているので、MPI-IO の説明は省略します。

4-4-3　入出力に関するその他の考慮点

- I/O が非常に多いプログラムの場合、図 4-4-12 に示すように、CPU 時間をいくら並列化して速くしてもジョブ全体としてはパフォーマンスは向上しません。このような場合、もし I/O 自体に並列性があるのであれば、4-2 節で述べたように、各プロセスが自分のローカルディスクに I/O を行うことによって、I/O 自体を並列化することができます。

図 4-4-12

- 単一のマシンではメモリーが足りないため、本来ならメモリー上に展開できるデータを作業ファイルを使用して計算しているために I/O が多くなっている場合があります。この場合、4-5-7 節で説明するように、並列化の際に配列を縮小すれば、1 つの並列ジョブが全ノードのメモリーを使用できるようになります。それによってメモリーが足りるようになれば、作業ファイルを使用しないようにプログラムを修正して I/O を減らすことができます。
- 一般にプログラムは図 4-4-13 のように 3 つの部分に分けられます。このうち入出力部分は経過時間がほとんどかからないので並列化せず、計算部分のみを並列化するのが一般的です。

4-4-2 節の「出力パターン 1」で、WRITE 文をランク 0 のプロセスだけが書き出すように IF 文を付加する方法を説明しました。図 4-4-13 のように WRITE 文が多くていちいち IF 文を付加するのが面倒な場合は、①で必要なデータをランク 0 のプロセスに送信した後、③でランク 0 以外のプロセスが終了するようにすれば、追加する IF 文は③の 1 行で済みます。なお、③を実行するよりも前に、②の MPI_FINALIZE を必ず 1 度実行しなければなりません。②を実行せずに③を実行した場合、マシン環境によっては誤動作しますので注意して下さい (「付録」の MPI_FINALIZE 参照)。

```
  :
READ(10,*) A
PAI = 3.14                          入力部分
  :
  :
DO I = ISTA, IEND
  X(I) = ～
  Y(I) = ～                         計算部分
  Z(I) = ～
ENDDO
  :
X,Y,Zをランク0に通信する。①
CALL MPI_FINALIZE(IERR)      ②
IF (MYRANK /= 0) STOP        ③
WRITE(10,*) X                       出力部分
WRITE(11,*) Y
WRITE(12,*) Z
  :
```

図 4-4-13

4-5　ループの分割方法

4-5-1　ループ反復と配列

4-2 節で、『並列計算機では、並列性さえあればどんなところでも並列化できる』と説明しました。とはいえ一般的にはほとんどの場合、DO ループを並列化します。DO ループの各反復で計算量が同じであるとすれば、各プロセスの仕事量を均等にするためには、（全反復回数）÷（プロセス数）（回）の反復を各プロセスに割り当てればよいことになります。例えば図 4-5-1 (1) (3) では、ループの全反復回数が 9 回でプロセス数が 3 つなので、各プロセスには 9 ÷ 3 = 3（回）の反復を割り当てています。このうち図 4-5-1 (1) は、DO ループの反復とループ内で使用している配列 A，B，C が 1 対 1 に対応しています (図 4-5-1 (2) 参照)。これに対し図 4-5-1 (3) では、指標ベクトル N を使用しているため、DO ループの反復とループ内で使用している配列 A は 1 対 1 に対応していません。このような場合、計算を各プロセスに割り当てる方法として、図 4-5-1 (4) の右図のように DO ループの反復を各プロセスに分ける方法のほかに、図 4-5-1 (5) のように配列 A で分割する方法もあります。

本節では、DO ループの反復を各プロセスに分ける方法を説明します。図 4-5-1 (1) のように、ループの反復と配列が 1 対 1 に対応している方が、分割方法を図示できて説明がしやすいのでこちらを想定します。従って『DO ループの反復を分ける』のと『配列を分割する』のは同じ意味で使用します。なお、図 4-5-1 (3) (4) を並列化する方法については 4-6-6 節で説明します。

図 4-5-1 (1)

図 4-5-1 (2)

図 4-5-1 (3)

図 4-5-1 (4)

図 4-5-1 (5)

4-5-2　配列の分割方法

2 次元配列を想定し、配列の分割方法について説明します。

ブロック分割

例えばプロセス数が 4 の場合、各プロセスに 1/4 ずつの連続した領域を割り当てる方法をブロック分割と呼びます。大半の並列化はブロック分割で行います。なお、2 次元配列の場合、図 4-5-2 (1) のように 2 次元目（列方向）を分割する方法と、1 次元目（行方向）を分割する方法があり、並列化によって発生する通信量を考慮して分割の方向を決定します (4-5-8 節参照)。

図 4-5-2 (1)

2 次元配列の場合、1 方向（行方向または列方向）のみでブロック分割するほかに、図 4-5-2 (2) に示すように両方向でブロック分割する方法もあります。この方法は、差分法などで、1 方向のみのブロック分割よりも通信量を減らしたい場合などに使用します (4-5-8 節参照)。

図 4-5-2 (2)

サイクリック分割

図 4-5-2 (3) のように、各列（または行）を、ランク 0，1，2，3，0，1，2，3，. . . のように各プロセスに割り当てる方法をサイクリック分割と呼びます。ブロック分割では何らかの理由でプロセス間のロードバランスが均等にならない場合に使用します (4-5-3 節参照)。

図 4-5-2 (3)

ブロック・サイクリック分割

図 4-5-2 (4) に示すように、何列（または行）かをまとめて 1 つのブロックとし、そのブロックを単位としてサイクリック分割を行う方法を、ブロック・サイクリック分割と呼びます。全ての列をブロックとした場合はブロック分割に、1 列のみをブロックとした場合はサイクリック分割になりますので、ブロック分割とサイクリック分割はブロック・サイクリック分割の両極端であると考えることもできます。

図 4-5-2 (4)

ブロック・サイクリック分割は、「本当はサイクリック分割にしたいが、サイクリック分割にすると別の問題が発生するため、多少サイクリック性を犠牲にする代わりにその問題を改善する」場合に使用します。

以下に例を示します。

(1)　キャッシュミスの減少

図 4-5-2 (5) に示すように、配列を縮小 (4-5-7 節参照) せずにサイクリック分割を行い、要素 A の次に要素 B を処理するとします。この場合、A と B のメモリー内の距離が離れているため、キャッシュ・ミス (参考文献 [6] 参照) が発生してパフォーマンスが低下する可能性が高くなります。キャッシュミスは図中の ↗ の部分で発生します。

ブロック・サイクリック分割にした図 4-5-2 (6) では、↗ の数が減るため、キャッシュミスの頻度が減ります。

図 4-5-2 (5)

図 4-5-2 (6)

(2)　通信量の減少

図 4-2-5 (5) (6) で、隣のプロセスとの境界（太い縦線部分）で横方向の通信が発生するとします。ブロック・サイクリック分割の方が境界の数が少ないので、通信量と通信回数が共に減少します。

4-5-3　ブロック分割でロードバランスが不均等になる例

本節では、ブロック分割では各プロセスのロードバランスが不均等になる例を紹介します。

例 1

大気の汚染を解析するプログラムがあり、汚染の激しい所ほど計算量が多いとします。このプログラムで東京湾の周辺を計算したところ、例えば図 4-5-3 (2) のようになっていたとします。黒い部分は汚染が激しい（つまり計算量が多い）ことを意味します。図 4-5-3 (2) の計算量を列ごとに合計すると、図 4-5-3 (1) のようになります。

この場合、図 4-5-3 (2) の上段のように、列方向にブロック分割すると、図 4-5-3 (3) のようにプロセス間でのロードバランスが著しく不均等になります。ところがサイクリック分割にすると、図 4-5-3 (3) のようにロードバランスはプロセス間でほぼ均等になります。

通常、自然現象はなめらかに変化するので、個々の列では計算量がバラついていても、サイクリック分割にして合計すれば、（よほど運が悪くない限り）プロセス間でのバラつきは相殺されます。

ただし、図 4-5-3 (2) で横方向の通信が必要なプログラムの場合は、サイクリック分割にすると他のプロセスとの境界が増えるので、通信量が多くなってしまいます（前節で説明したブロック・サイクリック分割にすれば、境界の数が減るので通信量をある程度減らすことができます）。

図 4-5-3 (1)

図 4-5-3 (2)

	ランク0	ランク1	ランク2
ブロック分割	1	40	0
サイクリック分割	14	15	12

図 4-5-3 (3)　計算量の合計

例 2

図 4-5-4 (1) の 2 次元領域のうち、実際に計算するのは着色した格子部分のみだとします（例えば湖）。この場合、例えば図 4-5-4 (1) のように列方向にブロック分割すると、ロードバランスが不均等になります。また解法が差分法の場合、一般に横方向の通信が必要になるので、例 1 のようにサイクリック分割にすると、他のプロセスとの境界が増えて通信量が多くなってしまいます。

この場合、図 4-5-4 (2) のように、実際に計算する格子の数が各プロセスでほぼ均等になるようにブロック分割するという方法が考えられます。

図 4-5-4 (1)　　図 4-5-4 (2)

例 3

図 4-5-4 (3) に示すように、●の要素だけを計算する三角行列の場合、列方向にブロック分割するとロードバランスは不均等になります。

この場合、図 4-5-4 (4) のように、各プロセスが担当する要素数がほぼ均等になるようにブロック分割するか、または図 4-5-4 (5) のようにサイクリック分割（またはブロック・サイクリック分割）にします。

図 4-5-4 (3)　　図 4-5-4 (4)　　図 4-5-4 (5)

例 4

図 4-5-5 (1) は例えば 1000 × 1000 の行列を表します。実際に計算する部分（太線内）が、最初は 1000 × 1000（正方行列全体）、次が 999 × 999、次が 998 × 998、…のように、計算が進むにつれて変化するとします (例えば 5-3 節の密行列の連立一次方程式の LU 分解)。

これを列方向にブロック分割すると、最初は各プロセスのロードバランスは均等ですが、計算が進むにつれてランク 0 の計算部分が終了し、さらにランク 1 の計算部分も終了し、不均等になってしまいます。

これを図 4-5-5 (2) のようにサイクリック分割（またはブロック・サイクリック分割）にすれば、計算が進んでもロードバランスは常にほぼ一定になります。

図 4-5-5 (1)　ブロック分割

図 4-5-5 (2)　サイクリック分割

例 5

箱の中に入った粒子の動きを調べるプログラムを考えます。各粒子は、自分を中心とした円内（点線）の粒子のみと力が働き、力、速度、位置の計算を行ないます。例えば図 4-5-4 (1) の粒子 (12) は、円内に他の粒子が 4 個あるため多くの計算を行います。一方粒子①は、円内に他の粒子がないため計算をほとんど行いません。

また、粒子の存在する位置が箱の中央に近いほど、円内にたくさん粒子があるとします。

この例では、箱の左端の粒子から順に粒子の番号が付いています。粒子番号でブロック分割した場合、図 4-5-6 (1) に示すように、箱の端付近の粒子を担当するランク 0 と 2 のプロセスは計算量が少なく、中央付近の粒子を担当するランク 1 のプロセスは計算量が多くなり、ロードバランスが不均等になってしまいます。

この場合、図 4-5-6 (2) に示すように、粒子番号でサイクリック分割（またはブロック・サイクリック分割）にすれば、各プロセスが担当する粒子の（箱内の）位置が偏らなくなるため、ロードバランスはほぼ均等になります。

図 4-5-6 (1)

図 4-5-6 (2)

4-5-4　ブロック分割

DO ループを各プロセスに分割する場合、いままでは説明を簡単にするため、1〜9 までの 9 個の要素を 3 つのプロセスに分割するというように、具体的な値がわかっていて、しかもちょうど割りきれる例で説明しました (図 4-5-7 (1))。ところが実際のプログラムでは、図 4-5-7 (2) のように、DO ループの反復が未知数になっていたり、プロセス数を 3 個でも 4 個でも分割できるようにしたいという場合もあります。本節では、処理する要素数、プロセス数がともに任意の値で、要素数がプロセス数で割りきれない可能性がある場合のブロック分割方法について説明します。

```
    :
DO I = 1, 9
  A(I) = B(I) + C(I)
ENDDO
    :
```

図 4-5-7 (1)

```
    :
DO I = N1, N2
  A(I) = B(I) + C(I)
ENDDO
    :
```

図 4-5-7 (2)

分割方法の種類

ブロック分割には 2 つの方法があります。これを要素が①〜⑥までの 6 個、プロセス数が 4 個 (ランク 0〜3) の場合で説明します。この例では要素数がプロセス数で割りきれないことに注意して下さい。

(1)　分割方法 1

最初の分割方法では、要素を各プロセスに図 4-5-7 (3) のように分配します。この意味ですが、各プロセスが図 4-5-7 (4) のようにそれぞれ箱をもっているとします。そして要素を 1 つずつ矢印の順序で箱に入れていき、要素を全部箱に入れ終わった状態が図 4-5-7 (3) ということになります。

ランク	0	1	2	3
要素	① ②	③ ④	⑤	⑥

図 4-5-7 (3)

図 4-5-7 (4)

(2)　分割方法 2

2 つ目の分割方法では、図 4-5-7 (5) に示すように、要素数が全プロセス数で割りきれない場合、最後以外のプロセスは同じ要素数を担当し、最後のプロセスだけがそれより少ない要素を担当します。

なお、図 4-5-7 (6) のように要素数がプロセス数より少しだけ多い場合、最後以外のプロセス (ランク 2) も、少ない要素数になることがあります。

ランク	0	1	2	3
要素	① ②	③ ④	⑤ ⑥	

図 4-5-7 (5)　6 個の要素を 4 プロセスで分割

ランク	0	1	2	3
要素	① ②	③ ④	⑤	

図 4-5-7 (6)　5 個の要素を 4 プロセスで分割

分割方法 1 と 2 の比較

分割方法 1 と 2 のどちらを選択するかは好みによります。並列ジョブでは最も処理時間の長いプロセスに足をひっぱられますが、図 4-5-7 (3) (5) から分かるように、どちらの分割方法でも各プロセスが担当する最多の要素数は同じなので、パフォーマンスはほとんど同じです。

実際の分割方法

実際のプログラムでは、分割を行うためのサブルーチンを使用するのが便利です。筆者が作成したサブルーチン例を以下に示します。例えば、要素番号の最小値 n1 (= 1) から最大値 n2 (= 20) までを、プロセス数 nprocs (= 4) で分割したとき、ランクの値が irank (= 0) のプロセスの担当範囲の下限 ista (= 1) と上限 iend (= 5) が戻ります（割りきれない場合も考慮されます)。

分割方法 1 と分割方法 2 に対応するサブルーチンをそれぞれ図 4-5-8 (1) と図 4-5-8 (2) に示します。

```
CALL PARA_RANGE(     n1     , n2 , nprocs, irank, ista,    iend   )
                （値を指定する）1 =  20 =     4 =     0 =     = 1    = 5（値が戻る）
```

- n1: 整数。分割したい要素番号の最小値を指定します。
- n2: 整数。分割したい要素番号の最大値を指定します。
- nprocs: 整数。分割に使用する全プロセス数を指定します。
- irank: 整数。担当範囲を調べたいプロセスのランクを指定します。0 ≦ irank ≦ (nprocs-1) です。
- ista: 整数。ランク irank のプロセスが担当する要素の下限が戻ります。
- iend: 整数。ランク irank のプロセスが担当する要素の上限が戻ります。

```
SUBROUTINE PARA_RANGE(N1,N2,NPROCS,IRANK,ISTA,IEND)
IWORK1 = (N2-N1+1)/NPROCS
IWORK2 = MOD(N2-N1+1,NPROCS)
ISTA = IRANK*IWORK1 + N1 + MIN(IRANK,IWORK2)
IEND = ISTA + IWORK1 - 1
IF (IWORK2 > IRANK) IEND = IEND + 1
END
```

図 4-5-8 (1)　分割方法 1 に対応するサブルーチン

```
SUBROUTINE PARA_RANGE(N1,N2,NPROCS,IRANK,ISTA,IEND)
IWORK = (N2-N1)/NPROCS + 1
ISTA = MIN(IRANK*IWORK + N1, N2+1)
IEND = MIN(ISTA+IWORK-1, N2)
END
```

図 4-5-8 (2)　分割方法 2 に対応するサブルーチン

なお、1 つのプロセスが担当する要素数 (IEND-ISTA+1) が少なすぎると、並列プログラムによっては誤動作しますので (4-8-2 節の「プログラムの並列化方法のミス (2)」参照)、各プロセスが担当する要素数が決定した時点で、正常に動作する最低限の要素数以上かどうかをチェックするようにして下さい (4-8-1 節の「(5) 並列に動作させる最低限の修正」参照)。

実際の適用例

図 4-5-9 (1) の A(1)〜A(N) (N = 1000) までの合計を求めるプログラムをサブルーチン PARA_RANGE (分割方法 1) を使用して並列化した例を図 4-5-9 (2) に、実行結果を図 4-5-9 (3) に示します。

実行結果から分かるように、1000 個の要素を 3 プロセスでブロック分割した場合、ピッタリは割りきれませんがほぼ均等に分割が行われ、またプロセス数がいくつでも同じ計算結果となっています。

```
      PROGRAM MAIN
      PARAMETER (N=1000)
      DIMENSION A(N)
      DO I = 1, N
        A(I) = I
      ENDDO
      SUM = 0.0
      DO I = 1, N
        SUM = SUM + A(I)
      ENDDO
      PRINT *,'SUM =',SUM
      END
```

図 4-5-9 (1)

```
      PROGRAM MAIN
      INCLUDE 'mpif.h'
      PARAMETER (N=1000)
      DIMENSION A(N)
      CALL MPI_INIT(IERR)
      CALL MPI_COMM_SIZE
     &    (MPI_COMM_WORLD,NPROCS,IERR)
      CALL MPI_COMM_RANK
     &    (MPI_COMM_WORLD,MYRANK,IERR)
      CALL PARA_RANGE
     &    (1,N,NPROCS,MYRANK,ISTA,IEND)
      DO I = ISTA, IEND
        A(I) = I
      ENDDO
      SUM = 0.0
      DO I = ISTA, IEND
        SUM = SUM + A(I)
      ENDDO
      CALL MPI_REDUCE(SUM,SSUM,1,MPI_REAL,
     &     MPI_SUM,0,MPI_COMM_WORLD,IERR)
      SUM = SSUM
      IF (MYRANK == 0) PRINT *,'SUM = ',SUM
      PRINT *,'MYRANK = ',MYRANK,
     &        'ISTA = ',ISTA,'IEND = ',IEND
      CALL MPI_FINALIZE(IERR)
      END
```

図 4-5-9 (2)

```
[aoyama@node01]/u/aoyama: mpirun -np 2 a.out
  SUM = 500500.000
  MYRANK = 0 ISTA = 1 IEND = 500
  MYRANK = 1 ISTA = 501 IEND = 1000
[aoyama@node01]/u/aoyama: mpirun -np 3 a.out
  SUM = 500500.000
  MYRANK = 0 ISTA = 1 IEND = 334
  MYRANK = 1 ISTA = 335 IEND = 667
  MYRANK = 2 ISTA = 668 IEND = 1000
```

図 4-5-9 (3)

補足ですが、例えば図 4-5-9 (4) の①のループを分割して ISTA と IEND を求めた場合、②のループに対しては図 4-5-9 (5) の下線部のように分割します (図 4-5-9 (6) 参照)。

```
      DO I=1,100 ①
        A(I) = ～
      ENDDO
      DO I=30,90 ②
        A(I) = ～
      ENDDO
```

図 4-5-9 (4)

→

```
      DO I=ISTA,IEND
        A(I) = ～
      ENDDO
      DO I=MAX(ISTA,30),MIN(IEND,90)
        A(I) = ～
      ENDDO
```

図 4-5-9 (5)

図 4-5-9 (6)

4-5-5　サイクリック分割

4-5-3 節で説明したように、ブロック分割ではプロセス間のロードバランスが不均等になり、サイクリック分割では均等になる場合があります。本節では、サイクリック分割の具体的な方法を説明します。

分割方法 1

図 4-5-10 (1) の DO ループを、プロセス数 NPROCS でサイクリック分割する方法を図 4-5-10 (2) に、3 プロセスでの分割の様子を図 4-5-10 (3) に示します。ループ反復の下限が「MYRANK+N1」なので、各プロセスは下限の 1 つずつずれた位置から間隔 NPROCS で反復します。この方法は最も簡単なので、特に問題がなければこの方法を用います。

```
    :
DO I = N1, N2
    計 算 部 分
ENDDO
    :
```

図 4-5-10 (1)　元のプログラム

```
    :
DO I = N1+MYRANK, N2, NPROCS
    計 算 部 分
ENDDO
    :
```

図 4-5-10 (2)

図 4-5-10 (3)

分割方法 2

プログラムの分かり易さなどの理由で、『DO I=N1,N2』をそのまま残しておきたい場合などに、図 4-5-10 (4) の方法を用います (5-3 節参照)。まず①で、ループ反復 I とそれを担当するランク値との対応表 (MAP) を 1 回だけ作成しておきます。②では DO ループを元のプログラムと同じように反復させます。③の IF 文で、①で作成した対応表を用いて現在の反復を担当するランクを調べ、それが自分のランク (MYRANK) と一致しているときのみ④の計算を行います。

なお、④の「計算部分」内にさらに DO ループがある場合、通常③の IF 文のオーバーヘッドは無視できますが、④内に DO ループがなく、いきなりステートメントがある場合は、③の IF 文によってパフォーマンスが低下してしまうので、この方法は用いない方が良いでしょう。

図 4-5-10 (4) で配列 MAP を確保するのが面倒な場合、図 4-5-10 (5) のようにすることもできます。

```
    :
INTEGER MAP(N2)
    :
DO I = N1, N2
  MAP(I) = MOD(I-N1,NPROCS)    ··· ①
ENDDO
DO I = N1, N2                  ··· ②
  IF (MYRANK == MAP(I)) THEN   ··· ③
    計 算 部 分                ··· ④
  ENDIF
ENDDO
    :
```

図 4-5-10 (4)

```
    :
DO I = N1, N2
  IF (MYRANK == MOD(I-N1,NPROCS)) THEN
    計 算 部 分
  ENDIF
ENDDO
    :
```

図 4-5-10 (5)

分割方法 3

多重ループの内側でサイクリック分割したい場合、以下のように例えば 0 → 1 → 2 → 0 → 1 → 2･･･ と反復するカウンター (IRANK) を使用します。

```
    :
  IRANK = -1
  DO J = J1, J2
    DO I = I1, I2
      IRANK = IRANK + 1
      IF (IRANK == NPROCS) IRANK = 0
      IF (MYRANK == IRANK) THEN
          計 算 部 分
      ENDIF
    ENDDO
  ENDDO
    :
```

図 4-5-10 (6)

実際の適用例

前節の図 4-5-9 (1) と同じ例を「分割方法 1」でサイクリック分割した例です。

```
  PROGRAM MAIN
  PARAMETER (N=1000)
  DIMENSION A(N)
  DO I = 1, N
    A(I) = I
  ENDDO
  SUM = 0.0
  DO I = 1, N
    SUM = SUM + A(I)
  ENDDO
  PRINT *,'SUM =',SUM
  END
```

図 4-5-11 (1)

```
  PROGRAM MAIN
  INCLUDE 'mpif.h'
  PARAMETER (N=1000)
  DIMENSION A(N)
  CALL MPI_INIT(IERR)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)
  DO I = 1+MYRANK, N, NPROCS ← サイクリック分割
    A(I) = I
  ENDDO
  SUM = 0.0
  DO I = 1+MYRANK, N, NPROCS ← サイクリック分割
    SUM = SUM + A(I)
  ENDDO
  CALL MPI_REDUCE(SUM,SSUM,1,MPI_REAL,
&                 MPI_SUM,0,MPI_COMM_WORLD,IERR)
  SUM = SSUM
  IF (MYRANK == 0) PRINT *,'SUM = ',SUM
  CALL MPI_FINALIZE(IERR)
  END
```

図 4-5-11 (2)

4-5-6　ブロック・サイクリック分割

4-5-2 節で説明したように、「本当はサイクリック分割にしたいが、サイクリック分割にすると別の問題が発生するため、多少サイクリック性を犠牲にする代わりにその問題を改善する」場合に使用します。本節では、ブロック・サイクリック分割の具体的な方法を説明します。

分割方法 1

図 4-5-12 (1) の DO ループを、プロセス数 NPROCS でブロック・サイクリック分割する方法を図 4-5-12 (2) に、3 プロセスでの分割の様子を図 4-5-12 (3) に示します。この方法は前節の「分割方法 1」に対応します。

長さ IBLOCK (本例では 2) のブロックを 1 つの単位としてサイクリック分割を行います。図 4-5-12 (2) の外側の DO II ループは図 4-5-12 (3) の → に対応し、内側の DO I ループは〓に対応します。

```
    :
  DO I = N1, N2
      計 算 部 分
  ENDDO
    :
```

図 4-5-12 (1)　元のプログラム

```
    :
  IBLOCK = ～ ← ブロックの大きさを指定します。
  DO II = N1+MYRANK*IBLOCK, N2, NPROCS*IBLOCK
    DO I = II, MIN(II+IBLOCK-1,N2)
        計 算 部 分
    ENDDO
  ENDDO
    :
```

図 4-5-12 (2)

図 4-5-12 (3)

分割方法 2

この方法は前節の「分割方法 2」に対応し、あらかじめ右図のような対応表 (MAP) を作成します。図 4-5-12 (4) で配列 MAP を確保したくない場合、図 4-5-12 (5) のようにすることもできます。なお、前述のように、IF 文を挿入することによってパフォーマンスが低下する可能性がありますので注意して下さい。

```
    :
  INTEGER MAP(N1:N2)
    :
  IBLOCK = ～ ← ブロックの大きさを指定します。
  DO I = N1, N2
    MAP(I) = MOD(I-N1,NPROCS*IBLOCK)/IBLOCK
  ENDDO
  DO I = N1, N2
    IF (MYRANK == MAP(I)) THEN
        計 算 部 分
    ENDIF
  ENDDO
    :
```

図 4-5-12 (4)

```
    :
  DO I = N1, N2
    IF (MYRANK ==
&       MOD(I-N1,NPROCS*IBLOCK)/IBLOCK) THEN
        計 算 部 分
    ENDIF
  ENDDO
    :
```

図 4-5-12 (5)

分割方法 3

図 4-5-12 (6) の最初のループで、図 4-5-12 (7) に示すように、配列 MAP にブロックサイクリック分割で自プロセスが担当するループ反復の番号を入れます。最初のループが終了すると、自プロセスが担当する全ループ反復数が ICOUNT に入ります。

図 4-5-12 (6) の 2 つ目のループで、MAP と ICOUNT を使用して、ブロックサイクリック分割で計算を行います。

```
      :
  INTEGER MAP(*)
      :
  IBLOCK = ～ ← ブロックの大きさを指定します。
  ICOUNT = 0
  DO I = N1, N2
    IF (MYRANK ==
&       MOD(I-N1,NPROCS*IBLOCK)/IBLOCK) THEN
      ICOUNT = ICOUNT + 1
      MAP(ICOUNT) = I
    ENDIF
  ENDDO
  DO II = 1, ICOUNT
    I = MAP(II)
    [      計 算 部 分      ]
  ENDDO
      :
```

図 4-5-12 (6)

図 4-5-12 (7)

4-5-7　配列の縮小

図 4-5-13 (1) に示す格子（地球）の大気の流れを計算するプログラムを単体ジョブとして実行するとします。ノード（〓）あたり 256 MB のメモリーが搭載されており、図 4-5-13 (1) は 256 MB の配列 A を表すとします。このプログラムを今まで説明した方法で並列化し、4 ノードで実行した場合、図 4-5-13 (2) に示すように、各プロセスが使用するのは配列 A の 1/4 の部分（図中の着色した部分）のみで、それ以外の部分は使用されません。この並列プログラムを、図 4-5-13 (3) のように、各プロセスが配列 A 全体を利用するように改造すれば、全格子数は図 4-5-13 (1) の 4 倍となり、よりきめの細かい解析ができるようになります。このように、配列を無理なく使用することを本書では配列の縮小と呼びます。本節では配列を縮小する方法について説明します。

図 4-5-13 (1)

図 4-5-13 (2)

図 4-5-13 (3)

4-5-7-1　配列の縮小方法

分散メモリー型並列計算機とメモリー

分散メモリー型並列計算機では、ノード数を増やせばいくらでもメモリーの合計を増やすことができます。従って、配列を縮小すれば、1 つの並列ジョブで非常に大きなメモリーを使用することができます。

ところで、配列の縮小は自動的に行われるわけではなく、ユーザーがそのようにプログラムを作成すれば、配列の縮小を行うことができるわけです。プログラムによっては配列の縮小は非常に面倒だったり、ほとんど不可能な場合もあります。

一般に、配列を縮小することを意識して最初から並列プログラムを作成する場合はそれほど問題ありませんが、既存の単体プログラムを並列化して配列を縮小する場合は、まず配列を縮小せずに並列化し、答えが合うようになってから配列の縮小を行うのがよいでしょう。

なお、配列の縮小とは関係ありませんが、プログラム内で使用する大きな配列のうち、メモリー上に常に存在する必要のない配列（作業配列）については、Fortran 90 の動的割り当て (参考文献 [6] の 2-6 節参照) の機能を使用し、その配列が必要になった時点でメモリー上に割り当て (ALLOCATE 文)、不要になったらメモリー上から削除する (DEALLOCATE 文) ようにすれば、メモリーを節約することができます。

ブロック分割による配列の縮小方法 1

それでは配列の縮小の方法について説明します。例によって図 4-5-15 (1) の配列の合計を求めるプログラムで説明します。一般化のため、DO ループの範囲が N1 から N2 までになっていることに注意して下さい。

これを、まず 4-5-4 節のブロック分割で並列化したのが図 4-5-15 (2) です。この時点ではまだ配列は縮小されておらず、例えばプロセス数が 3 つの場合、各プロセスは配列 A のうち図 4-5-14 の『配列縮小前』の部分を担当しています。

図 4-5-14

```
PROGRAM MAIN
PARAMETER(N1=1,N2=10)
DIMENSION A(N2)
DO I = N1, N2
  A(I) = I
ENDDO
SUM = 0.0
DO I = N1, N2
  SUM = SUM + A(I)
ENDDO
PRINT *,'SUM =',SUM
END
```

図 4-5-15 (1)

図 4-5-15 (2) に対して配列の縮小を行うと図 4-5-15 (3) になります。図の『←』は図 4-5-15 (2) と異なる部分を示します。図 4-5-15 (3) について以下に説明します。

- 例えば大きさ 100 の配列を縮小した場合、ノード数が 2 ならば 100/2 = 50、ノード数が 4 ならば 100/4 = 25 のように、ノード数によって縮小する配列の大きさを変える必要があります。従って、縮小する配列 A を宣言する際にノード数がわかっている必要があるため、⑤でノード数 (NCPU) を指定します。⑥と⑦で配列 A を縮小して宣言しますが、このとき NCPU の値を使用します。なお、⑥では、縮小する前の配列の大きさがノード数で割りきれない場合を考慮しています。
- ⑧では念のために、⑤で指定したノード数 (NCPU) と『mpirun -np 3 a.out ☰』で指定したプロセス数 (NPROCS) が一致しているかどうかのチェックを行います。
- ⑨と⑪では、図 4-5-15 (2) の①、③と同じ反復 (ISTA と IEND) をそのまま使用することにします。
- ⑩の左辺と⑫の右辺の配列 A の添字では、ループ反復 I が図 4-5-14 の『配列縮小後』になるようにアドレス変換を行っています。例えば⑨と⑪の I が ISTA の場合、⑩の左辺と⑫の右辺の配列 A の添字は『ISTA-ISTA+1』、つまり『1』となります。
- 図 4-5-15 (2) の②の右辺では、ループ反復①の I という変数を直接使用しています。一方⑨で①と同じ反復の範囲を使用しているため、⑩の右辺でも I をそのまま使用します。

```
  PROGRAM MAIN
  INCLUDE 'mpif.h'
  PARAMETER(N1=1,N2=10)
  DIMENSION A(N2)
  CALL MPI_INIT(IERR)
  CALL MPI_COMM_SIZE(～_WORLD,NPROCS,IERR)
  CALL MPI_COMM_RANK(～_WORLD,MYRANK,IERR)
  CALL PARA_RANGE
&    (N1,N2,NPROCS,MYRANK,ISTA,IEND)
  DO I = ISTA, IEND                          ①
    A(I) = I                                 ②
  ENDDO
  SUM = 0.0
  DO I = ISTA, IEND                          ③
    SUM = SUM + A(I)                         ④
  ENDDO
  CALL MPI_REDUCE(SUM,SSUM,1,MPI_REAL,
&          MPI_SUM,0,MPI_COMM_WORLD,IERR)
  SUM = SSUM
  IF (MYRANK==0) PRINT *,'SUM = ',SUM
  CALL MPI_FINALIZE(IERR)
  END
```

図 4-5-15 (2)

```
  PROGRAM MAIN
  INCLUDE 'mpif.h'
  PARAMETER(N1=1,N2=10)
  PARAMETER(NCPU=3)                          ⬅ ⑤
  PARAMETER(NN=(N2-N1)/NCPU+1)               ⬅ ⑥
  DIMENSION A(NN)                            ⬅ ⑦
  CALL MPI_INIT(IERR)
  CALL MPI_COMM_SIZE(～_WORLD,NPROCS,IERR)
  CALL MPI_COMM_RANK(～_WORLD,MYRANK,IERR)
  IF (NPROCS /= NCPU) THEN
    IF (MYRANK==0) PRINT *,'===ERROR==='
    CALL MPI_FINALIZE(IERR)                  ⬅ ⑧
    STOP
  ENDIF
  CALL PARA_RANGE
&    (N1,N2,NPROCS,MYRANK,ISTA,IEND)
  DO I = ISTA, IEND                          ⑨
    A(I-ISTA+1) = I                          ⬅ ⑩
  ENDDO
  SUM = 0.0
  DO I = ISTA, IEND                          ⑪
    SUM = SUM + A(I-ISTA+1)                  ⬅ ⑫
  ENDDO
  CALL MPI_REDUCE(SUM,SSUM,1,MPI_REAL,
&          MPI_SUM,0,MPI_COMM_WORLD,IERR)
  SUM = SSUM
  IF (MYRANK==0) PRINT *,'SUM = ',SUM
  CALL MPI_FINALIZE(IERR)
  END
```

図 4-5-15 (3)

ブロック分割による配列の縮小方法 2

図 4-5-15 (3) では、⑨と⑪のループ反復は図 4-5-15 (2) の①、③をそのまま使用しました。ところで、配列 A を縮小した場合、図 4-5-14 の『配列縮小後』に示すように、各プロセスは配列 A を『1』から使用します。これにともない、⑨と⑪のループ反復を『1』から開始させるという方法もあります。これを図 4-5-15 (4) に示します。図 4-5-15 (4) の『←』は図 4-5-15 (3) と異なる部分を示します。

- ⑬と⑮に示すように、ループ反復の下限を『1』から開始しています。ループ反復の上限は、⑨と⑪からわかるように、『IEND-ISTA+1』となります。ループ反復の下限と上限を変更したため、⑭の左辺と⑯の右辺の配列 A の添字は I となります。これは元の②、④と同じです。
- ⑩では②の右辺の I をそのまま使用しましたが、今度は⑬の反復を『1』から開始するようにしたため、I を⑭の右辺のように変更する必要があります。⑬で I = 1 のとき⑭の右辺は『I+ISTA-1』、つまり ISTA となります。⑨で I = ISTA のとき⑩の右辺も ISTA となるので、両者は同じであることがわかります。

図 4-5-15 (3) と (4) のどちらの形式を使用するかは好みによります。プログラムをまず縮小せずに並列化し、その後で縮小する場合は、図 4-5-15 (3) の方が DO ループの反復を修正する必要がないため簡単でしょう。逆に、配列を縮小した並列化を最初から意識してプログラムを作成する場合は、図 4-5-15 (4) の方が自然かもしれません。なお、⑩と⑫の『-STA+1』、⑬と⑮の『IEND-STA+1』などは定数にするともう少しすっきりします。

```
      PROGRAM MAIN
      INCLUDE 'mpif.h'
      PARAMETER(N1=1,N2=10)
      PARAMETER(NCPU=3)
      PARAMETER(NN=(N2-N1)/NCPU+1)
      DIMENSION A(NN)
      CALL MPI_INIT(IERR)
      CALL MPI_COMM_SIZE(～_WORLD,NPROCS,IERR)
      CALL MPI_COMM_RANK(～_WORLD,MYRANK,IERR)
      IF (NPROCS /= NCPU) THEN
        IF (MYRANK==0) PRINT *,'===ERROR==='
        CALL MPI_FINALIZE(IERR)
        STOP
      ENDIF
      CALL PARA_RANGE
     &    (N1,N2,NPROCS,MYRANK,ISTA,IEND)
      DO I = 1, IEND-ISTA+1                  ← ⑬
        A(I) = I+ISTA-1                      ← ⑭
      ENDDO
      SUM = 0.0
      DO I = 1, IEND-ISTA+1                  ← ⑮
       SUM = SUM + A(I)                      ← ⑯
      ENDDO
       CALL MPI_REDUCE(SUM,SSUM,1,MPI_REAL,
     &          MPI_SUM,0,MPI_COMM_WORLD,IERR)
      SUM = SSUM
      IF (MYRANK==0) PRINT *,'SUM = ',SUM
      CALL MPI_FINALIZE(IERR)
      END
```

図 4-5-15 (4)

ブロック分割による配列の縮小方法 3

Fortran 90 の新機能であるメモリーの動的割振り機能 (C 言語の関数 malloc に相当) を用いると、配列の縮小を比較的簡単に行うことができます。まず図 4-5-16 (1) の例で、メモリーの動的割振りの方法について簡単に説明します。なお、メモリーの動的割振りの詳細は、参考文献 [6] の 2-6 節を参照して下さい。

- ①の下線部で、REAL*8 の配列 A と B がプログラムの実行時に動的に割振られることを宣言します。配列が 1 次元の場合は A(:)、2 次元の場合は B(:,:) のように指定します。下線部の途中のブランクはなくてもかまいません。
- この例では、②で標準入力から配列 A と B の大きさ IMAX, JMAX を読み込みます。読み込んだ IMAX, JMAX を使用して③を実行すると、配列 A と B が実際にメモリー上に割振られます。以後は配列 A と B を通常の配列と同様に使用することができます。なお③の下線部（任意）を指定すると、整変数 IERR（名前は任意）に、ALLOCATE 文が成功したときはゼロが、（メモリー不足などで）失敗したときはゼロ以外の値が戻ります。これを3でチェックし、エラーの場合は終了します。
- 配列 A と B を引数でサブルーチンに渡す場合は④のようにします。
- 使用が終了して不要になった配列を解放したい場合は⑤を実行します。もちろん解放せずにプログラムを終了してもかまいません。⑤の下線部は任意です。
- 配列 A と B を図 4-5-16 (2) の⑥のようにしてサブルーチンに渡すこともできます。

```
PROGRAM MAIN
REAL*8, ALLOCATABLE :: A(:),B(:,:)                ①
READ(5,*) IMAX,JMAX                               ②
ALLOCATE(A(IMAX),B(IMAX,JMAX),STAT=IERR)          ③
IERRが0以外だったらエラーで終了する。             3
CALL SUB(A,B,IMAX,JMAX)                           ④
DEALLOCATE(A,B,STAT=IERR)                         ⑤
IERRが0以外だったらエラーで終了する。
END
SUBROUTINE SUB(A,B,IMAX,JMAX)                     ④
REAL*8 A(IMAX),B(IMAX,JMAX)                       ④
  :
END
```

図 4-5-16 (1)

```
PROGRAM MAIN
COMMON /COM/ IMAX, JMAX                           ⑥
REAL*8, ALLOCATABLE :: A(:),B(:,:)
READ(5,*) IMAX,JMAX
ALLOCATE(A(IMAX),B(IMAX,JMAX),STAT=IERR)
IERRが0以外だったらエラーで終了する。
CALL SUB(A,B)                                     ⑥
DEALLOCATE(A,B,STAT=IERR)
IERRが0以外だったらエラーで終了する。
END
SUBROUTINE SUB(A,B)                               ⑥
COMMON /COM/ IMAX, JMAX                           ⑥
REAL*8 A(IMAX),B(IMAX,JMAX)
  :
END
```

図 4-5-16 (2)

次に、動的に割振られた配列を引数でなく COMMON 文でサブルーチンに渡す方法を説明します。実は動的に割振られた配列を COMMON 文の中に指定することはできません。その代わりに、Fortran 90 の新機能である MODULE 文と USE 文を使用します。MODULE 文は COMMON 文を含んだ INCLUDE ファイルに相当し、USE 文は INCLUDE 文に相当します。図 4-5-16 (3) の⑦に示すように MODULE 文に適当な名前 (本例では COM) をつけます。MODULE〜END で囲まれた⑧の部分には、グローバルとして扱いたい変数や配列を指定します。この部分に、動的割振りする変数や配列を指定することができます。⑧で指定した配列や変数は、⑨の USE 文を指定したメイン（サブ）ルーチンのみから参照／更新することができます。

MODULE 文と USE 文に関する注意点を以下に述べますが、使用法に少しクセがあるので注意して下さい (詳細は参考文献 [6] の 2-6 節参照)。

- USE 文はメイン（サブ）ルーチンの先頭で指定する必要があります。
- MODULE〜END は INCLUDE ファイルと異なりプログラムと同じファイルに入れることができますが、ファイル内の最初の USE 文よりも前に置かれなければなりません。従ってファイル内の先頭に置くのがよいでしょう。
- MODULE〜END の中に IMPLICIT 文を指定することもできますが、これは MODULE〜END 内のみで有効となり、USE 文を指定したメイン（サブ）ルーチン側には適用されないので、必要ならばメイン（サブ）ルーチンで再度 IMPLICIT 文を宣言して下さい。

なお、この例では⑧で IMAX, JMAX を指定していますが、これは動的割振りとは関係なく、サブルーチン SUB で IMAX と JMAX を使用しなければこの指定は不要です。

MODULE 文は COMMON 文を含んだ INCLUDE ファイルに相当しますが、INCLUDE ファイルと異なりプログラムの入ったファイルと別ファイルにする必要はありません。

```
MODULE COM                                  ⑦
REAL*8, ALLOCATABLE :: A(:),B(:,:)          ⑧
INTEGER IMAX,JMAX                           ⑧
END
PROGRAM MAIN
USE COM                                     ⑨
  :
READ(5,*) IMAX,JMAX
ALLOCATE(A(IMAX),B(IMAX,JMAX),STAT=IERR)
IERRが0以外だったらエラーで終了する。
  :
CALL SUB
  :
DEALLOCATE(A,B,STAT=IERR)
END
SUBROUTINE SUB
USE COM                                     ⑨
  :
END
```

図 4-5-16 (3)

```
PROGRAM MAIN
INCLUDE 'mpif.h'
PARAMETER(N1=1,N2=10)
REAL, ALLOCATABLE :: A(:)                   ①
CALL MPI_INIT(IERR)
CALL MPI_COMM_SIZE(～_WORLD,NPROCS,IERR)
CALL MPI_COMM_RANK(～_WORLD,MYRANK,IERR)
CALL PARA_RANGE
&    (N1,N2,NPROCS,MYRANK,ISTA,IEND)        ②
ALLOCATE(A(ISTA:IEND),STAT=IERR)            ③
IERRが0以外だったらエラーで終了する。
DO I = ISTA, IEND
  A(I) = I
ENDDO
SUM = 0.0
DO I = ISTA, IEND
  SUM = SUM + A(I)
ENDDO
 CALL MPI_REDUCE(SUM,SSUM,1,MPI_REAL,
&          MPI_SUM,0,MPI_COMM_WORLD,IERR)
SUM = SSUM
IF (MYRANK==0) PRINT *,'SUM = ',SUM
CALL MPI_FINALIZE(IERR)
END
```

図 4-5-17 (1)

メモリーの動的割振り機能を用いて図 4-5-15 (1) の配列を縮小した例を上記の図 4-5-17 (1) に示します。まず①で配列 A が動的に割振られることを宣言します。②で自分の担当範囲の下限 (ISTA) と上限 (IEND) を確定した後、③で自分の担当部分の 1 次元配列 A(ISTA:IEND) をメモリー上に割振ります。これで全て完了です。この方法では図 4-5-14 は図 4-5-17 (2) のようになります。プロセスによって配列 A の下限と上限、大きさが異なっていることに注意して下さい。

この方法では、図 4-5-15 (3) の⑤ ⑥ ⑧は全て不要になり、また⑩や⑫のように配列の添字を修正したり、図 4-5-15 (4) の⑬や⑮のように DO ループの反復を修正する必要もなくなります。

図 4-5-17 (2)

配列を縮小するメリット

通常、配列を縮小するのは、1 台のノードが持っているメモリーだけではメモリーが足りない場合です。逆に言うと、1 台のノードが持っているメモリーで足りるのならば、手間をかけて配列を縮小する必要はないわけです。ところがこのような場合でも、計算と通信のパフォーマンスの点から、やはり配列を縮小した方がよい場合があります。

図 4-5-18 (1) で、縦方向が配列の 1 次元目、すなわちメモリーが連続する方向であるとします。図 4-5-18 (1) 上段のように、配列を 1 次元目でブロック分割し縮小をしない場合、ランク 0 のプロセスが処理する斜線の部分のうち、1と2の要素は↓に示すようにメモリー上で離れています。このため、ランク 0 のプロセスが斜線の部分の要素を順に計算する場合、キャッシュ・ミスが発生してパフォーマンスが若干低下します。また、斜線の部分を通信する際にもこの部分で同様にキャッシュ・ミスが発生します。

これを図 4-5-18 (1) 下段のように配列を縮小すれば、メモリー上で1と2が連続するため、これらのキャッシュ・ミスがなくなります。なお、配列の 2 次元目でブロック分割した場合は、縮小しなくても全ての要素がメモリー上で連続しているため、この問題はありません。

図 4-5-18 (2) 上段のように、配列をサイクリック分割し縮小をしない場合も、メモリー上で3と4が離れているため、図 4-5-18 (1) と同様の問題が発生します。

図 4-5-18 (1)

図 4-5-18 (2)

4-5-7-2　配列の縮小に伴う変更 (I/O 以外の部分)

単体プログラムをまず配列を縮小せずに並列化し、その後で配列を縮小した場合、以下の箇所を修正する必要があります。この他に次節で説明する I/O 部分の修正が必要になります（プログラムによってはそれ以外にも修正の必要な箇所があるかも知れません）。

なお、以下の例では 4-5-7-1 節の「ブロック分割による配列の縮小方法 3」で縮小しています。

サブルーチン化した通信ルーチンの配列宣言の変更

配列を縮小せずに並列化したプログラムを図 4-5-20 (1) に示します。このプログラムでは異なる配列 A と B に対して①で同じパターンの通信を行うため、②のように通信部分をサブルーチン化しています。このプログラムの配列 A と B を図 4-5-20 (2) の③のように縮小した場合、④の部分も修正する必要があります。

```
  :
REAL A(N),B(N)
ISTA = ～
IEND = ～
CALL SHIFT(A,N,ISTA,IEND)          ①
CALL SHIFT(B,N,ISTA,IEND)          ①
  :
SUBROUTINE SHIFT(X,N,ISTA,IEND)    ②
REAL X(N)
CALL MPI_xxxx(X(ISTA),～)
  :
```

図 4-5-20 (1)　並列化（縮小前）

```
  :
REAL,ALLOCATABLE::A(:),B(:)                ③
ISTA = ～
IEND = ～
ALLOCATE(A(ISTA:IEND),B(ISTA:IEND))        ③
CALL SHIFT(A,N,ISTA,IEND)
CALL SHIFT(B,N,ISTA,IEND)
  :
SUBROUTINE SHIFT(X,N,ISTA,IEND)
REAL X(ISTA:IEND)                          ④
CALL MPI_xxxx(X(ISTA),～)
  :
```

図 4-5-20 (2)　並列化（縮小後）

それまで並列化していなかった DO ループの並列化

図 4-5-21 (1) の単体プログラムでは、②のタイムステップ内に含まれる③の部分で計算時間がかかっており、①の部分の計算時間はわずかであるとします。配列を縮小しないで並列化する場合は、図 4-5-21 (2) のように、計算時間のかかる⑤の部分のみを並列化することもできます。

一方配列 A を縮小した図 4-5-21 (3) では、計算時間のかからない⑥の部分も並列化する必要があります。

```
  :
DO I=1,N                ①
  A(I) = 1.0
ENDDO
DO ITIME=1,1000         ②
  DO I=1,N              ③
    A(I) = A(I) + 1.0
  ENDDO
    :
ENDDO
  :
```

図 4-5-21 (1)　並列化前

→

```
  :
DO I=1,N                ④
  A(I) = 1.0
ENDDO
DO ITIME=1,1000
  DO I=ISTA,IEND        ⑤
    A(I) = A(I) + 1.0
  ENDDO
    :
ENDDO
  :
```

図 4-5-21 (2)　並列化（縮小前）

→

```
  :
ALLOCATE(A(ISTA:IEND))
DO I=ISTA,IEND          ⑥
  A(I) = 1.0
ENDDO
DO ITIME=1,1000
  DO I=ISTA,IEND
    A(I) = A(I) + 1.0
  ENDDO
    :
ENDDO
  :
```

図 4-5-21 (3)　並列化（縮小後）

派生データ型を使用した通信の修正

メモリー上で飛び飛びになっているデータを送（受）信する場合、派生データ型 (3-5 節参照) を使用することができます。そして飛び飛びのデータを含む配列を縮小した場合、データは（一般的に）連続するので、派生データ型を使用した通信部分を修正する必要があります。

図 4-5-22 (1) で、各プロセスは 2 次元配列 A(6,2) のうち着色した部分のデータを送信するとします。着色した部分に含まれる要素はメモリー上で飛び飛びなので、派生データ型を使用します。図 4-5-23 (1) の①で、3-5-3-2 節で説明した自作のサブルーチン PARA_TYPE_BLOCK2A をコールし、図 4-5-22 (1) の○の要素から始まって着色した部分を表す派生データ型 ITYPE を作成し、これを使用して②で通信を行います。

図 4-5-23 (2) では、③で自作のサブルーチン PARA_TYPE_BLOCK2 コールし (MPI-2 が使用できるマシン環境の場合は、MPI-2 で提供されている MPI_TYPE_CREATE_SUBARRAY (3-5-3-1 節参照) を使用して下さい)、図 4-5-22 (2) の○から始まって着色した部分を表す派生データ型 ITYPE を作成し、これを使用して④で通信を行います。なお④の送信データの先頭アドレスは②と異なります。

この配列 A を図 4-5-22 (3) のように縮小した場合、送信データはメモリー上で連続するので、通信で派生データ型を使用する必要はありません。図 4-5-23 (3) の⑤と⑥で縮小した配列 A をアロケートし、⑦の下線部で送信するデータの個数を指定し、（派生データ型を使用せずに）通常の方法でデータを送信します。

図 4-5-22 (1)

```
   :
 REAL A(6,2)
 ISTAとIENDを決定する。
 CALL PARA_TYPE_BLOCK2A                    ①
&    (1,6,IEND-ISTA+1,2,MPI_REAL,ITYPE)    ①
 CALL MPI_SEND(A(ISTA,1),1,ITYPE,～)       ②
   :
```

図 4-5-23 (1)

図 4-5-22 (2)

```
   :
 REAL A(6,2)
 ISTAとIENDを決定する。
 CALL PARA_TYPE_BLOCK2                     ③
&    (1,6,1,ISTA,IEND,1,2,MPI_REAL,ITYPE)  ③
 CALL MPI_SEND(A,1,ITYPE,～)               ④
   :
```

図 4-5-23 (2)

図 4-5-22 (3)

```
   :
 REAL,ALLOCATABLE::A(:,:)                  ⑤
 ISTAとIENDを決定する。
 ALLOCATE(A(ISTA:IEND,2))                  ⑥
 CALL MPI_SEND(A(ISTA,1),(IEND-ISTA+1)*2,  ⑦
&     MPI_REAL,～)                         ⑦
   :
```

図 4-5-23 (3)

4-5-7-3　配列の縮小に伴う変更 (I/O 部分)

本節では、並列プログラム内で I/O が行われる配列を縮小した場合の修正方法について説明します。図 4-5-24 (1) の単体プログラムでは、①で配列 X(2,6) を計算し、②で 1 回の WRITE 命令で 10 番ファイルに書き出します。また図 4-5-24 (2) の単体プログラムでは、③で 10 番ファイルを 1 回の READ 命令で配列 Y(2,6) に読み込んだ後、④で計算を行います。この 2 つのプログラムを並列化し、配列 X と Y を縮小するとします。

```
    :
  REAL X(2,6)
  DO J=1,6
    DO I=1,2
      X(I,J) = ～                          ①
    ENDDO
  ENDDO
  WRITE(10) ((X(I,J),I=1,2),J=1,6)         ②
    :
```

図 4-5-24 (1)　出力プログラム

```
    :
  REAL Y(2,6)
  READ(10) ((Y(I,J),I=1,2),J=1,6)          ③
  DO J=1,6
    DO I=1,2
      ～ = Y(I,J)                          ④
    ENDDO
  ENDDO
    :
```

図 4-5-24 (2)　入力プログラム

まず図 4-5-24 (1) の並列化ですが、4-4-2 節の出力パターンのうち、「出力パターン 2」は単一ファイルにならず、計算が終了した後で合体しなければならないので面倒です。そこで「出力パターン 1」を採用します。なお、「出力パターン 3」の MPI-IO の機能を使用することも可能ですが、説明は省略します。

配列を縮小する場合は（一般に）メモリーに余裕がないので、各プロセスから送信されたデータ全体を同時にランク 0 の受信用配列に集めることはできません。そこでランク 0 のプロセスは、まずランク 1 のプロセスから送信されたデータを受信するために必要な最小限の大きさの一時配列 temp を確保してから受信を行い、それを出力ファイルに書き出し、不要になった配列 temp を解放します。そして次にランク 2 に対して同じ処理を行います (プロセス数が 3 の場合)。

図 4-5-24 (1) を並列化し、配列 X を縮小したプログラムとその動作を図 4-5-25 (1) (2) に示します。なお、図 4-5-25 (1) の出力ファイルは図 4-5-24 (2) のプログラムで読むことはできません（レコード形式が異なるので)。読む場合は図 4-5-25 (3)（単体プログラム）または図 4-5-28 (1)（並列プログラム）を使用して下さい。

- 図 4-5-25 (1) の [1] で、配列 X(2,6) の 2 次元目を、4-5-4 節で説明したサブルーチン PARA_RANGE を使用してブロック分割し、各プロセスが担当する 2 次元目の下限と上限を配列 JJSTA, JJEND に設定します。
- [2] で各プロセスは、自分が担当する 2 次元目の下限と上限を JSTA, JEND に設定します。
- [3] で各プロセスは、自分が担当する大きさの配列 X を動的割振りで縮小して確保し、[4] で計算を行います。
- [5] でランク 0 のプロセスは、まず自分が担当した部分をファイルに書き出します (図 4-5-25 (2) の [5] 参照)。図 4-5-24 (1) では配列 X 全体を 1 回の WRITE 文で書き出しましたが、[5] では 1 列ずつ書き出します。この理由は（注）を参照して下さい。
- [6] の DO ループでまず IRANK = 1 のとき、ランク 0 のプロセスはランク 1 のプロセスから送信されたデータを以下のように書き出します。
  - ランク 1 (正確には 0 以外) のプロセスは、[11] で自分の担当部分をランク 0 のプロセスに送信します。
  - ランク 0 のプロセスは、[7] で、ランク 1 のプロセスから送信されたデータを受信するために必要な最小限の大きさの一時配列 temp を動的割振りで確保します。次に [8] でデータを受信し、[9] で 10 番ファイルに列ごとに書き出します。最後に [10] で不要となった一時配列 temp を解放します。
- [6] の DO ループが IRANK = 2 となり、ランク 2 から送信されたデータを上記と同様に受信して書き出します。以後同様に処理を行います。

（注）[5] と [9] で、配列全体を 1 回の WRITE 文で書き出した場合は図 4-5-26 (1) の下線のようになり、図 4-5-26 (2) のファイル内の実線で囲んだ部分が 1 回の WRITE で 1 レコードとして書き出されます (例えば 3 プロセスで並列化した場合、3 レコードが書き出されます)。すると、後でこのファイルを単体ジョブ、またはプロセス数が 3 以外の並列ジョブで読み込む場合、読み込みが面倒になります、このため図 4-5-25 (1) では図 4-5-25 (2) のように 1 列ずつバラバラに書き出しました。ただし図 4-5-26 (1) のように配列全体を書き出すよりも、パフォーマンスは多少低下するかも知れません。

```
      :
      INCLUDE 'mpif.h'
      REAL,ALLOCATABLE::TEMP(:,:)
      REAL,ALLOCATABLE::X(:,:)
      INTEGER ISTATUS(MPI_STATUS_SIZE)
      PARAMETER(NCPU=3)
      INTEGER JJSTA(0:NCPU-1),JJEND(0:NCPU-1)
        :
      DO IRANK=0,NPROCS-1
        CALL PARA_RANGE(1,6,NPROCS,IRANK,                [1]
     &    JJSTA(IRANK),JJEND(IRANK))
      ENDDO                                              A
      JSTA = JJSTA(MYRANK)                               [2]
      JEND = JJEND(MYRANK)                               [2]
      ALLOCATE(X(2,JSTA:JEND))                           [3]
      DO J=JSTA,JEND
        DO I=1,2
          X(I,J) = ～                                    [4]
        ENDDO
      ENDDO
      IF (MYRANK==0) THEN
        DO J=JSTA,JEND
          WRITE(10)(X(I,J),I=1,2)                        [5]
        ENDDO
        DO IRANK=1,NPROCS-1                              [6]
          ALLOCATE                                       [7]
     &     (TEMP(2,JJSTA(IRANK):JJEND(IRANK)))           [7]
          ISIZE=2*(JJEND(IRANK)-JJSTA(IRANK)+1)
          CALL MPI_IRECV(TEMP,ISIZE,MPI_REAL,            [8]
     &      IRANK,1,MPI_COMM_WORLD,IREQ,IERR)            [8]
          CALL MPI_WAIT(IREQ,ISTATUS,IERR)               [8]
          DO J=JJSTA(IRANK),JJEND(IRANK)
            WRITE(10)(TEMP(I,J),I=1,2)                   [9]
          ENDDO
          DEALLOCATE(TEMP)                               [10]
        ENDDO
      ELSE
        CALL MPI_ISEND(X,2*(JEND-JSTA+1),MPI_REAL,
     &    0,1,MPI_COMM_WORLD,IREQ,IERR)                  B [11]
        CALL MPI_WAIT(IREQ,ISTATUS,IERR)
      ENDIF
      :
```

図 4-5-25 (1)　図 4-5-25 (2) のプログラム例

図 4-5-25 (2)

```
      :
      INCLUDE 'mpif.h'
      REAL,ALLOCATABLE::TEMP(:,:)
      図4-5-25(1)のAと同じ
      IF (MYRANK==0) THEN
        WRITE(10)((X(I,J),I=1,2),J=JSTA,JEND)
        DO IRANK=1,NPROCS-1
          ALLOCATE
     &     (TEMP(2,JJSTA(IRANK):JJEND(IRANK)))
          ISIZE=2*(JJEND(IRANK)-JJSTA(IRANK)+1)
          CALL MPI_IRECV(TEMP,ISIZE,MPI_REAL,
     &      IRANK,1,MPI_COMM_WORLD,IREQ,IERR)
          CALL MPI_WAIT(IREQ,ISTATUS,IERR)
          WRITE(10)((TEMP(I,J),I=1,2),
     &      J=JJSTA(IRANK),JJEND(IRANK))
          DEALLOCATE(TEMP)
        ENDDO
      図4-5-25(1)のBと同じ
      :
```

図 4-5-26 (1)　図 4-5-26 (2) のプログラム例

図 4-5-26 (2)

```
      :
      REAL Y(2,6)
      DO J=1,6
        READ(10) (Y(I,J),I=1,2)   ③
      ENDDO
      :
```

図 4-5-25 (3)

図 4-5-25 (1) では、1 つのプロセスから送信されたデータが全て入る大きさの一時配列 temp で受信しましたが、ファイルへの書き出しは 1 列ずつ行いました。もし一時配列 temp の大きさも節約したいのであれば、図 4-5-27 (1) (2) に示すように、配列 temp を 1 列分の大きさとし、各プロセスからのデータを 1 列ずつ受信してはファイルに書き出すこともできます。ただし通信回数が増えるので、パフォーマンスは若干低下します。

最後に図 4-5-24 (2) の入力プログラムを 4-4-1 節の「入力パターン 3」で並列化する方法を図 4-5-28 (1) (2) に示します。入力ファイルは図 4-5-25 (1) または図 4-5-27 (1) で作成したファイルであるとします。

- 図 4-5-28 (1) の [1] で 2 次元目をブロック分割し、[2] で自分が担当する 2 次元目の下限と上限を JSTA, JEND に設定します。[3] で配列 Y を縮小して動的割振りで確保します。
- [4] で入力ファイルのうち自分の担当範囲より前の部分を 1 列ずつ空読みします。
- [5] で入力ファイルのうち自分の担当範囲の部分を配列 Y に 1 列ずつ読み込みます。
- [6] で自分の担当範囲の部分を計算します。

```
    :
  INCLUDE 'mpif.h'
  REAL TEMP(2)
  図4-5-25(1)のAと同じ
  IF (MYRANK==0) THEN
    DO J=JSTA,JEND
      WRITE(10)(X(I,J),I=1,2)
    ENDDO
    DO IRANK=1,NPROCS-1
      DO J=JJSTA(IRANK),JJEND(IRANK)
        CALL MPI_IRECV(TEMP,2,MPI_REAL,
&         IRANK,1,MPI_COMM_WORLD,IREQ,IERR)
        CALL MPI_WAIT(IREQ,ISTATUS,IERR)
        WRITE(10)(TEMP(I),I=1,2)
      ENDDO
    ENDDO
  ELSE
    DO J=JSTA,JEND
      CALL MPI_ISEND(X(1,J),2,MPI_REAL,
&       0,1,MPI_COMM_WORLD,IREQ,IERR)
      CALL MPI_WAIT(IREQ,ISTATUS,IERR)
    ENDDO
  ENDIF
    :
```

図 4-5-27 (1)

図 4-5-27 (2) のプログラム例

図 4-5-27 (2)

```
    :
  INCLUDE 'mpif.h'
  REAL,ALLOCATABLE::TEMP(:,:)
  REAL,ALLOCATABLE::Y(:,:)
  PARAMETER(NCPU=3)
  INTEGER JJSTA(0:NCPU-1),JJEND(0:NCPU-1 )
    :
  DO IRANK=0,NPROCS-1                          [1]
    CALL PARA_RANGE(1,6,NPROCS,IRANK,
&         JJSTA(IRANK),JJEND(IRANK))
  ENDDO
  JSTA = JJSTA(MYRANK)                         [2]
  JEND = JJEND(MYRANK)
  ALLOCATE(Y(2,JSTA:JEND))                     [3]
  DO J=1,JSTA-1
    READ(10) ← 空読み                          [4]
  ENDDO
  DO J=JSTA,JEND
    READ(10) (Y(I,J),I=1,2)                    [5]
  ENDDO
  DO J=JSTA,JEND
    DO I=1,2
      ～ = Y(I,J)                              [6]
    ENDDO
  ENDDO
    :
```

図 4-5-28 (1)

図 4-5-28 (2) のプログラム例

図 4-5-28 (2)

4-5-8　多重ループの分割

多重ループを並列化する場合、内側のループで並列化するか、外側のループで並列化するか、あるいは両方で並列化するかを決定する必要があります。本節ではこれを決定する際の考慮点について説明します。

多重ループで内側と外側のどちらを並列化するか

まず以下の 2 つの DO ループを単体で実行したときの速度を比較します。図 4-5-29 (1) では内側のループが配列の左側の添字で反復しており、図 4-5-29 (2) ではその逆になっています。Fortran では、2 次元配列 A はメモリー上で A(1,1), A(2,1), ... の順に配置されますが、スカラー計算機では、図 4-5-29 (1) のようにメモリー上に配置された順番に要素を処理した方が、キャッシュ・ミスが発生しにくいため速くなります。一方 C 言語では、メモリー上での要素の配置順序が逆になるので、図 4-5-29 (2) の方が速くなります (キャッシュ・ミスの詳細は参考文献 [6] の 4 章を参照)。

```
DIMENSION A(100,100)
DO J = 1, 100
  DO I = 1, 100
    A(I,J) = ～
  ENDDO
ENDDO
  :
```

図 4-5-29 (1)

```
DIMENSION A(100,100)
DO I = 1, 100
  DO J = 1, 100
    A(I,J) = ～
  ENDDO
ENDDO
  :
```

図 4-5-29 (2)

図 4-5-29 (1) をブロック分割で並列化するとして、内側と外側のどちらのループを並列化 (計算量を $1/n$ に縮小) するのがよいでしょうか？　外側の J で並列化した例を図 4-5-30 (1) に、内側の I で並列化した例を図 4-5-30 (2) に示します。

配列を縮小していない場合を想定すると、前節の説明と重複しますが、図 4-5-30 (2) では各プロセスの2と3、4と5などがメモリー上で離れているため、計算および通信を行う際にキャッシュ・ミスが発生する可能性が高くなります。しかし図 4-5-30 (1) では1～12までの各要素がメモリー上で連続しているため、キャッシュ・ミスは発生しません。

一方、配列を縮小している場合はこのような問題はありませんが、いずれにしても、他の条件が全く同じであれば、図 4-5-30 (1) のように外側のループを並列化するのがよいでしょう。

図 4-5-30 (1)

図 4-5-30 (2)

一方向にだけ通信が必要な場合

図 4-5-31 (1) では、図 4-5-31 (2) に示すように、配列 A の左と右の要素の値から配列 B の要素を計算します。このため、図 4-5-31 (3) のように J で分割した場合、計算の前に、境界を越えた 1 列（黒の部分）を隣のプロセスから送信してもらう必要があります。これに対し、図 4-5-31 (4) のように I で分割した場合、通信は必要ありません。このように、一方向にだけ通信が必要な場合には、通信が不要な方向に分割するようにします。

このような例は差分法のプログラムで現れます。実際のプログラムでは、ある方向に通信が全く必要ないような場合はまずありませんが、例えば計算領域が 3 次元で Z 方向が重力方向の場合、プログラム内で並列化する全 DO ループを検討してみると、X または Y 方向の通信に比べて Z 方向の通信が少ない（あるいはその逆の）場合があります。

```
    :
  DIMENSION A(100,100), B(100,100)
  DO J = 1, 100
    DO I = 1, 100
      B(I,J) = A(I,J-1) + A(I,J+1)
    ENDDO
  ENDDO
    :
```

図 4-5-31 (1)

図 4-5-31 (2)

図 4-5-31 (3)

図 4-5-31 (4)

別の例として、例えば差分法の境界条件処理の部分が図 4-5-31 (5) のようになっていて、図 4-5-31 (6) のように境界の 1 列を反対側にコピーし、境界の行についてはコピーしないとします。この場合、図 4-5-31 (7) のように分割すると通信が発生しますが、図 4-5-31 (8) の分割では通信は発生しません。

```
    :
  DIMENSION A(100,100)
  DO I = 1, 100
    A(I,1) = A(I,100)
  ENDDO
    :
```

図 4-5-31 (5)

図 4-5-31 (6)

図 4-5-31 (7)

図 4-5-31 (8)

計算領域が長方形（直方体）の場合

図 4-5-32 (1) では、図 4-5-32 (2) に示すように、配列 A の上下左右の値から配列 B を計算します。このため計算領域を I で分割しても J で分割しても通信が発生します。

ところで今までの例では、計算領域が正方形 (3 次元では立方体) の場合を扱ってきましたが、この例では計算領域が長方形 (3 次元では直方体) になっています（例えば新幹線のような細長い物体を解析する場合)。

I 方向の方が J 方向よりも長いので、図 4-5-32 (3) のように J で分割するよりも図 4-5-32 (4) のように I で分割した方が、分割した境界の長さが短くなるため通信量も少なくなります。

このように、どちらの方向にも通信が必要な多重ループを分割する場合には、通信量が少なくなるような方向に分割するようにして下さい。

```
    :
  DIMENSION A(200,100), B(200,100)
  DO J = 1, 100
    DO I = 1, 200
      B(I,J) = A(I-1,J) + A(I,J-1) + A(I,J+1) + A(I+1,J)
    ENDDO
  ENDDO
    :
```

図 4-5-32 (1)

図 4-5-32 (2)

図 4-5-32 (3)

図 4-5-32 (4)

2 方向のブロック分割

今までは全て、I または J のどちらか 1 方向でブロック分割を行いましたが、4-5-2 節で紹介したように I と J の 2 方向でブロック分割を行うこともできます。プログラム例を図 4-5-33 (1) に示します。

1 方向だけで分割した図 4-5-33 (2) では、プロセス数を増やしていっても境界の長さは変わらないので、次第に通信の割合が増えてサチュレートします。一方 2 方向でブロック分割した図 4-5-33 (3) では、プロセス数が増えるにつれて境界の長さが短くなり通信量が減るため、サチュレートしにくくなります (ただし通信回数は 2 倍になって 1 方向だけの分割より時間がかかり、また修正も面倒になります)。従って、差分法でノード数が多い場合に良好なパフォーマンスを得るためには、2 方向の分割が必須です。なお、この分割はノード数が整数 × 整数で表せる場合 (例えば $4 \times 2 = 8$) にのみ可能です。適用例については 5-1-3 節を参照して下さい。

```
    :
DO J = JSTA, JEND
  DO I = ISTA, IEND
    B(I,J) = A(I-1,J) + A(I,J-1) + A(I,J+1) + A(I+1,J)
  ENDDO
ENDDO
    :
```

図 4-5-33 (1)

図 4-5-33 (2)

図 4-5-33 (3)

4-6　並列化とメッセージ交換

前節では、本当の意味の並列化の方法、すなわち計算量を $1/n$ に減少させる方法について説明し、その具体的な方法としてブロック分割、サイクリック分割などの方法を紹介しました。一方、1 章で説明しましたが、『本当の意味の並列化に伴う矛盾（副作用）を解消するために、必要最小限、仕方なしに行う』のがメッセージ交換であり、メッセージ交換サブルーチンの使い方自体は 3 章で説明しました。本節では、メッセージ交換が実際のプログラムのどのような局面で必要になり、具体的にどのように行うのかを説明します。

4-6-1　メッセージ交換はどんなときに必要か

メッセージ交換がどんなときに必要となるかについて、原点に戻って考えてみましょう。図 4-6-1 (1) の単体プログラムでは、DO ループで配列 A の値を更新し、DO ループから抜けた後の①で A(6) の値を参照します。これを図 4-6-1 (4) のように表現します。

このプログラムを並列化すると図 4-6-1 (2) のようになります。①のステートメントは全プロセスが実行しますが、図 4-6-1 (5) に示すように、A(6) に正しい値が入っているのはランク 2 のプロセスのみなので、このままでは (ランク 0 とランク 1 のプロセスでは) 結果がおかしくなってしまいます。これが 1 章で説明した「本当の意味の並列化に伴う矛盾（副作用）」です。

そこで図 4-6-1 (3) に示すように、並列化した DO ループの終了後の②で、ランク 2 のプロセスが A(6) をランク 0 と 1 のプロセスに送信し、その後で各プロセスが①を実行すれば正しく動作します。これが「本当の意味の並列化に伴う矛盾（副作用）を解消するために、必要最小限、仕方なしに行うメッセージ交換」です。

以上をまとめます。DO ループに対して本当の意味の並列化すると、各プロセスでは計算を $1/n$ (ノード数が n 台の場合) しか行わないので、その DO ループより後の部分で矛盾（副作用）が発生する可能性があり、それを解消するためにメッセージ交換を行います。そして、プログラムのどの部分で、どの変数を、どのプロセス間で通信すれば、本当の意味の並列化に伴う矛盾（副作用）を解消できて、しかも通信量を最小にできるか、がポイントであり、これを決定するのはユーザーということになります。

このように一般的に説明すると、メッセージ交換にはいろいろなバリエーションがあって難しく感じられるかも知れませんが、普通のプログラムではそれほど多くのパターンがあるわけではありません。典型的なメッセージ交換のパターンについて次節以降で説明します。

```
DIMENSION A(6)
  :
DO I=1,6
  A(I) = I
ENDDO
  :
B = A(6)+1.0   ①
  :
```

図 4-6-1 (1)

```
DIMENSION A(6)
  :
DO I=ISTA,IEND
  A(I) = I
ENDDO
  :
B = A(6)+1.0   ①
  :
```

図 4-6-1 (2)

```
DIMENSION A(6)
  :
DO I=ISTA,IEND
  A(I) = I
ENDDO
A(6)を全プロセスに送信 ②
  :
B = A(6)+1.0          ①
  :
```

図 4-6-1 (3)

図 4-6-1 (4)

図 4-6-1 (5)

図 4-6-1 (6)

4-6-2　1 次元差分法

キングオブ科学技術計算の称号を差分法に与えても異議をとなえる人はいないでしょう。実際、過去に私が並列化したプログラムのうち、約半数が差分法のプログラムでした。

4-3 節の「計算パターン 2」で簡単に説明したように、差分法を並列化する場合、境界のデータを隣接するプロセス間でお互いに交換するパターン（以後シフト）が現れます。本節では最も簡単な 1 次元の差分法でのシフトについて説明し、一般的な 2 次元のシフトについては 5-1 節で説明します。

シフト (1 次元の場合)

差分法モドキの簡単なプログラムとして図 4-6-2 (1) を例に取り上げます。まず①で配列 A に適当な値を設定し、次に②で、配列 A の前後の値から配列 B の値を求めます。データの動きを図示すると図 4-6-2 (2) のようになります。

これを例えば 3 つのプロセスで並列に計算する場合、配列 A と B を例えば図 4-6-2 (3) のようにブロック分割し、各プロセスは自分の担当する要素のみを計算します。例えばランク 1 は、①では A(5)〜A(8) を計算し、②では B(5)〜B(8) を計算します。②の計算で A(4)〜A(9) の値を参照しますが、A(4) は①でランク 0 が計算し、A(9) は①でランク 2 が計算しているので、ランク 1 はこれらの値を持っていません。そこで②の計算に入る前に、A(4) をランク 0 から、A(9) をランク 2 からそれぞれ送ってもらう必要があります。逆に、ランク 1 は A(5) をランク 0 に、A(8) をランク 2 に送信してあげる必要があります。このように、②の計算に入る前に境界のデータを隣接するプロセス間でお互いにシフトする必要があります。

なお、この例では、例えばランク 1 のプロセスは A(4) と A(9) の値を①で計算しておけばシフトの必要はありませんが、実際のプログラムではほとんどの場合シフトが必要になると考えて下さい。

```
PROGRAM MAIN
IMPLICIT REAL*8(A-H,O-Z)
PARAMETER (N=11)
DIMENSION A(N),B(N)
DO I = 1, N
  A(I) = I                                  ①
ENDDO
DO I = 2, N-1
  B(I) = A(I-1) + A(I+1)                    ②
ENDDO
END
```

図 4-6-2 (1)

図 4-6-2 (2)

図 4-6-2 (3)

図 4-6-2 (1) を並列化したプログラムを図 4-6-3 (1) に、データの動きを図 4-6-3 (4) に示します (図 4-6-3 (1) と図 4-6-3 (4) の⑥〜⑨は対応しています)。

- まず各プロセスが担当する配列 A と B の範囲を決定します。このとき図 4-6-2 (3) に示したように、配列 A と B では分割した境界の位置（図中の縦線部分）が一致していなければなりません。しかし配列 A (①のループ) の範囲は 1〜N, 配列 B (②のループ) の範囲は 2〜N-1 と異なっているので、配列 A を基準にして分割し、それに配列 B の分割の境界を合わせるか、その逆にするか、いずれかを選択します（どちらでも構いません）。ここでは前者の方法を選択することにします。

図 4-6-3 (1) の③で、まずサブルーチン PARA_RANGE (4-5-4 節参照) を使用して配列 A の範囲 (1〜N) をブロック分割し、各プロセスが担当する配列 A (①のループの反復) の下限 (ISTA) と上限 (IEND) を取得します。③の残りの部分では各プロセスが担当する配列 B (②のループの反復) の下限 (ISTA2) と上限 (IEND1) を決定します。図 4-6-2 (3) の縦線を一致させるため、端以外のプロセスでは ISTA2 を ISTA と、IEND2 を IEND と等しくし、端を担当するランク 0 のプロセスの ISTA2 と ISTA、およびランク 2 のプロセスの IEND1 と IEND だけが異なるようにします。例えばランク 0 のプロセスの担当範囲 ISTA, IEND, ISTA2, IEND1 は図 4-6-2 (3) のようになります。なお、この例では配列 A を基準に分割し配列 B の分割をそれに合わせたため、配列 B の分割のロードバランスが各プロセス間で不均等になっていますが、要素数 (N) が多くなればロードバランスはほぼ均等になり、問題はありません。

- ③で得た ISTA, IEND, ISTA2, IEND1 を使用して、図 4-6-2 (1) の①と②のループ反復を、図 4-6-3 (1) の⑤と⑪のように修正します。
- 前述のように、⑪の計算に入る前に、境界のデータを隣接するプロセス間でシフトする必要があります。シフトで通信されるデータの動きを図 4-6-3 (4) の（実線の）『〓』と『〓』に示します。
  以後『〓』のシフトについて説明します。まず図 4-6-3 (1) の④で、『〓』方向の宛先プロセスのランク値を変数 IUP に、送信元プロセスのランク値を変数 IDOWN に設定します。ところでこの例では『〓』の点線部分のシフトは行なっていません。これを非循環シフトと呼び、点線部分のシフトを行う場合を循環シフトと呼びます。多くのプログラムでは非循環シフトになっているようです。非循環シフトでは、一番右端 (ランク NPROCS-1) のプロセスは『〓』の送信をしないので、IUP には MPI_PROC_NULL を指定します (付録の MPI_ISEND, MPI_IRECV の引数参照)。同様に一番左端 (ランク 0) のプロセスは『〓』の受信をしないので、IDOWN には MPI_PROC_NULL を指定します。
- 図 4-6-3 (1) の⑥〜⑩の二重線で囲んだ部分でシフトを行います。なお、シフトには以下で説明する通信方法以外に、(注) で説明する 2 つの方法があり、どの方法が最も速いかはマシン環境によって異なります。
  ⑥では『〓』の送信を行います。宛先プロセスのランクは IUP、送信するデータは図 4-6-3 (4) に示すように A(IEND) となります。この送信に対応する『〓』の受信は⑧で、送信元プロセスのランクは IDOWN、受信したデータを入れる場所は図 4-6-3 (4) に示すように A(ISTA-1) となります。
  同様に⑦と⑨で『〓』の送信と受信を行います。宛先プロセスと送信元プロセスが『〓』の場合と逆になるので注意して下さい。
  ⑥〜⑨は非ブロッキング通信ルーチンなので一気に実行（が開始）されます。そしてそれぞれに対し、ストッパーとして⑩で MPI_WAIT を実行します。MPI_WAIT の引数で配列 ISTATUS を使用するので、ISTATUS を配列宣言するのを忘れないで下さい。
- ⑥〜⑩で⑪の計算に必要なデータがそろったので⑪を計算します。

以上で図 4-6-3 (1) の説明を終了します。このように、差分法で並列化する DO ループ内のステートメントの右辺に A(I-1) や A(I+1) のような項がある場合、そのループに入る前に境界のデータをシフトするのが基本的なパターンとなります。言うまでもありませんが、例えば A(I-1) のみが現れている場合は図 4-6-3 (4) の『〓』のシフト、すなわち⑥、⑧と MPI_WAIT のみを行います。

なお、本例は 1 次元なのでプロセス間の境界の 1 要素（点）をシフトしましたが、2 次元の場合は『線』を、3 次元の場合は『面』をシフトすることになります。

別の通信方法（重要）

図 4-6-3 (1) の二重線で囲んだ部分の通信（シフト）を行う別の方法を紹介します。まず図 4-6-3 (1) の⑥〜⑩の順序を変更した図 4-6-3 (2) で、同じ通信を行うことができます。

また MPI ではシフト専用の 1 対 1 通信サブルーチン MPI_SENDRECV（付録参照）が提供されており、このルーチンを使用した場合は図 4-6-3 (3) のようになります。例えば図 4-6-3 (3) の1は図 4-6-3 (2) の1と同じです。このルーチンはブロッキング通信ルーチンですが、デッドロックにはなりません。

以上 3 つの通信方法のうち、どの方法がもっとも速いかは、マシン環境によって異なります。なお、本書の以後のプログラム例では、シフトの通信は便宜的に図 4-6-3 (1) で行いました。

```
  PROGRAM MAIN
  IMPLICIT REAL*8(A-H,O-Z)
  INCLUDE 'mpif.h'
  PARAMETER (N=11)
  DIMENSION A(N),B(N)
  INTEGER ISTATUS(MPI_STATUS_SIZE)
  CALL MPI_INIT(IERR)
  CALL MPI_COMM_SIZE(～,NPROCS,IERR)
  CALL MPI_COMM_RANK(～,MYRANK,IERR)
  CALL PARA_RANGE(1,N,NPROCS,MYRANK,ISTA,IEND)
  ISTA2 = ISTA
  IEND1 = IEND                                  ③
  IF (MYRANK == 0        ) ISTA2 = 2
  IF (MYRANK == NPROCS-1) IEND1 = N-1
  IUP   = MYRANK + 1
  IDOWN = MYRANK - 1                            ④
  IF (MYRANK == NPROCS-1) IUP  =MPI_PROC_NULL
  IF (MYRANK == 0)        IDOWN=MPI_PROC_NULL
  DO I = ISTA, IEND
    A(I) = I                                    ⑤
  ENDDO
  CALL MPI_ISEND(A(IEND  ),1,MPI_REAL8,         ⑥
&      IUP  ,1,MPI_COMM_WORLD,ISEND1,IERR)      ⑥
  CALL MPI_ISEND(A(ISTA  ),1,MPI_REAL8,         ⑦
&      IDOWN,1,MPI_COMM_WORLD,ISEND2,IERR)      ⑦
  CALL MPI_IRECV(A(ISTA-1),1,MPI_REAL8,         ⑧
&      IDOWN,1,MPI_COMM_WORLD,IRECV1,IERR)      ⑧
  CALL MPI_IRECV(A(IEND+1),1,MPI_REAL8,         ⑨
&      IUP  ,1,MPI_COMM_WORLD,IRECV2,IERR)      ⑨
  CALL MPI_WAIT(ISEND1,ISTATUS,IERR)
  CALL MPI_WAIT(ISEND2,ISTATUS,IERR)            ⑩
  CALL MPI_WAIT(IRECV1,ISTATUS,IERR)
  CALL MPI_WAIT(IRECV2,ISTATUS,IERR)
  DO I = ISTA2, IEND1
    B(I) = A(I-1) + A(I+1)                      ⑪
  ENDDO
  CALL MPI_FINALIZE(IERR)
  END
```

図 4-6-3 (1)　通信方法 1

```
    :
  CALL MPI_ISEND(A(IEND  ),1,MPI_REAL8,         ⑥
&      IUP  ,1,MPI_COMM_WORLD,ISEND,IERR)       ⑥
  CALL MPI_IRECV(A(ISTA-1),1,MPI_REAL8,         ⑧
&      IDOWN,1,MPI_COMM_WORLD,IRECV,IERR)   [1] ⑧
  CALL MPI_WAIT(ISEND,ISTATUS,IERR)             ⑩
  CALL MPI_WAIT(IRECV,ISTATUS,IERR)             ⑩
  CALL MPI_ISEND(A(ISTA  ),1,MPI_REAL8,         ⑦
&      IDOWN,1,MPI_COMM_WORLD,ISEND,IERR)       ⑦
  CALL MPI_IRECV(A(IEND+1),1,MPI_REAL8,         ⑨
&      IUP  ,1,MPI_COMM_WORLD,IRECV,IERR)   [2] ⑨
  CALL MPI_WAIT(ISEND,ISTATUS,IERR)             ⑩
  CALL MPI_WAIT(IRECV,ISTATUS,IERR)             ⑩
    :
```

図 4-6-3 (2)　通信方法 2

```
    :
  CALL MPI_SENDRECV
&        (A(IEND  ),1,MPI_REAL8,IUP  ,1,         ⑥
&         A(ISTA-1),1,MPI_REAL8,IDOWN,1,     [1] ⑧
&         MPI_COMM_WORLD,ISTATUS,IERR)
  CALL MPI_SENDRECV
&        (A(ISTA  ),1,MPI_REAL8,IDOWN,1,         ⑦
&         A(IEND+1),1,MPI_REAL8,IUP  ,1,     [2] ⑨
&         MPI_COMM_WORLD,ISTATUS,IERR)
    :
```

図 4-6-3 (3)　通信方法 3

図 4-6-3 (4)

境界条件の処理

差分法を並列化した場合、境界条件の処理を行う部分で以下のような修正が必要になることがあります。

まず図 4-6-4 (1) の①では、境界の要素 A(1) だけが特別な処理をしています。並列化した図 4-6-4 (2) で A(1) を担当しているのはランク 0 のプロセスなので、②の IF 文を追加して、ランク 0 のプロセスのみが①の処理を行うようにします。

```
   :
 DO I = 1, N
    A(I) = B(I) + C(I)
 ENDDO
 A(1) = A(1) + 1.0      ①
   :
```

図 4-6-4 (1)

```
   :
 DO I = ISTA, IEND
    A(I) = B(I) + C(I)
 ENDDO
 IF (MYRANK==0) A(1) = A(1) + 1.0  ②
   :
```

図 4-6-4 (2)

次に、境界条件が周期的境界になっている場合の例を示します。図 4-6-5 (1) と図 4-6-6 (1) の①では、各境界のデータを反対側の境界にコピーしています。これを並列化した場合、図 4-6-5 (2) に示すように、A(0), A(1) を担当するランク 0 のプロセスと、A(N), A(N+1) を担当するランク 2 のプロセスの間で、〓と〓の通信が必要となります。この通信を行うプログラムを図 4-6-6 (2) に示します。なお、このプログラムを 2 プロセス以上で実行した場合は問題ありませんが、1 プロセスで実行すると誤動作します。参考までにその理由と対処方法を以下に示します。いずれにしても、並列化したプログラムを 1 プロセスで実行すると、他の箇所でも同様の問題が発生する可能性があるのでお勧めできません (3-4-5 節の「1 対 1 通信を 1 プロセスで実行した場合」参照)。

図 4-6-6 (2) を 1 プロセスで実行すると、④が ElSEIF になっていてランク 0 は⑤と⑥の通信を実行しないので、②から③への通信のみがランク 0 内で行われてしまい、結果は図 4-6-5 (3) のようになります。

そこで、ELSEIF を分離して別の IF 文にすると図 4-6-6 (3) のようになります。ここで本来は ⑦ → ⑪ と ⑩ → ⑧ の通信が行われるはずですが、⑦, ⑧, ⑩, ⑪の下線に示すタグの値が同一 (「1」) のため、⑦ → ⑧ と ⑩ → ⑪ の通信が行われてしまい、結果は図 4-6-5 (4) のようになってしまいます。

そこで、図 4-6-6 (4) の⑫, ⑬, ⑮, ⑯の下線に示すように、送信と受信のペアごとにタグの値を区別してみます。プログラムを実行すると、⑬の受信命令が実行された後、その受信が実際に行われるまで⑭で WAIT します。この WAIT は⑮の送信が行われれば解除されますが、⑭で WAIT しているため⑮が実行出来ず、デッドロックになってしまいます (これは 3-4-4 節の「パターン 2」に相当します)。

そこで、図 4-6-6 (5) のように、送信命令と受信命令を全て実行してから WAIT するようにすれば、デッドロックを回避でき、1 プロセスでも正常に動作します。

境界を担当する 2 プロセス間のみで通信が必要となる並列プログラムを、1 プロセスで実行する可能性がある場合、図 4-6-6 (5) のようにプログラムが複雑になってしまいます。このような場合は、図 4-6-6 (6) のように、プロセス数が 2 プロセス以上の場合と 1 プロセスの場合で処理を分けた方が良いかも知れません。

図 4-6-5 (1)

図 4-6-5 (3)

図 4-6-5 (4)

図 4-6-5 (2)

```
   :
 PARAMETER(N=6)
 REAL A(0:N+1)
 DO I=1,N
   A(I) = ～
 ENDDO
 A(0  ) = A(N)                                  ①
 A(N+1) = A(1)                                  ①
   :
```

図 4-6-6 (1)

```
   :
 IF (MYRANK==0) THEN
  CALL MPI_ISEND(A(1),1,MPI_REAL,NPROCS-1,1,
&      MPI_COMM_WORLD,IREQ1,IERR)              ②
  CALL MPI_IRECV(A(0),1,MPI_REAL,NPROCS-1,1,
&      MPI_COMM_WORLD,IREQ2,IERR)              ③
   CALL MPI_WAIT(IREQ1,ISTATUS,IERR)
   CALL MPI_WAIT(IREQ2,ISTATUS,IERR)
 ELSEIF(MYRANK==NPROCS-1) THEN                  ④
   CALL MPI_ISEND(A(N  ),1,MPI_REAL,0,1,
&       MPI_COMM_WORLD,IREQ1,IERR)              ⑤
   CALL MPI_IRECV(A(N+1),1,MPI_REAL,0,1,
&       MPI_COMM_WORLD,IREQ2,IERR)              ⑥
   CALL MPI_WAIT(IREQ1,ISTATUS,IERR)
   CALL MPI_WAIT(IREQ2,ISTATUS,IERR)
 ENDIF
   :
```

図 4-6-6 (2)

```
   :
 IF (MYRANK==0) THEN
  CALL MPI_ISEND(A(1),1,MPI_REAL,NPROCS-1,1,
&      MPI_COMM_WORLD,IREQ1,IERR)              ⑦
  CALL MPI_IRECV(A(0),1,MPI_REAL,NPROCS-1,1,
&      MPI_COMM_WORLD,IREQ2,IERR)              ⑧
   CALL MPI_WAIT(IREQ1,ISTATUS,IERR)
   CALL MPI_WAIT(IREQ2,ISTATUS,IERR)
 ENDIF
 IF(MYRANK==NPROCS-1) THEN                      ⑨
   CALL MPI_ISEND(A(N  ),1,MPI_REAL,0,1,
&       MPI_COMM_WORLD,IREQ1,IERR)              ⑩
   CALL MPI_IRECV(A(N+1),1,MPI_REAL,0,1,
&       MPI_COMM_WORLD,IREQ2,IERR)              ⑪
   CALL MPI_WAIT(IREQ1,ISTATUS,IERR)
   CALL MPI_WAIT(IREQ2,ISTATUS,IERR)
 ENDIF
   :
```

図 4-6-6 (3)

```
   :
 IF (MYRANK==0) THEN
  CALL MPI_ISEND(A(1),1,MPI_REAL,NPROCS-1,1,
&      MPI_COMM_WORLD,IREQ1,IERR)              ⑫
  CALL MPI_IRECV(A(0),1,MPI_REAL,NPROCS-1,2,
&      MPI_COMM_WORLD,IREQ2,IERR)              ⑬
   CALL MPI_WAIT(IREQ1,ISTATUS,IERR)
   CALL MPI_WAIT(IREQ2,ISTATUS,IERR)            ⑭
 ENDIF
 IF(MYRANK==NPROCS-1) THEN
   CALL MPI_ISEND(A(N  ),1,MPI_REAL,0,2,
&       MPI_COMM_WORLD,IREQ1,IERR)              ⑮
   CALL MPI_IRECV(A(N+1),1,MPI_REAL,0,1,
&       MPI_COMM_WORLD,IREQ2,IERR)              ⑯
   CALL MPI_WAIT(IREQ1,ISTATUS,IERR)
   CALL MPI_WAIT(IREQ2,ISTATUS,IERR)
 ENDIF
   :
```

図 4-6-6 (4)

```
   :
 IF (MYRANK==0) THEN
  CALL MPI_ISEND(A(1),1,MPI_REAL,NPROCS-1,1,
&      MPI_COMM_WORLD,IREQ1,IERR)
  CALL MPI_IRECV(A(0),1,MPI_REAL,NPROCS-1,2,
&      MPI_COMM_WORLD,IREQ2,IERR)
 ENDIF
 IF(MYRANK==NPROCS-1) THEN
   CALL MPI_ISEND(A(N  ),1,MPI_REAL,0,2,
&       MPI_COMM_WORLD,IREQ3,IERR)
   CALL MPI_IRECV(A(N+1),1,MPI_REAL,0,1,
&       MPI_COMM_WORLD,IREQ4,IERR)
 ENDIF
 IF (MYRANK==0) THEN
   CALL MPI_WAIT(IREQ1,ISTATUS,IERR)
   CALL MPI_WAIT(IREQ2,ISTATUS,IERR)
 ENDIF
 IF(MYRANK==NPROCS-1) THEN
   CALL MPI_WAIT(IREQ3,ISTATUS,IERR)
   CALL MPI_WAIT(IREQ4,ISTATUS,IERR)
 ENDIF
   :
```

図 4-6-6 (5)

```
   :
 IF (NPROCS==1) THEN
   A(0  ) = A(N)
   A(N+1) = A(1)
 ELSE
   図4-6-6(2)と同じ
 ENDIF
   :
```

図 4-6-6 (6)

4-6-3　集団通信サブルーチンの代替

4-6-3-1　MPI_GATHERV の代替

各プロセスが計算した結果を、例えばランク 0 に収集する通信パターンが現れます (4-4-2 節参照)。

(ケース 1) MPI-2 の MPI_IN_PLACE が使用でき、データが連続な場合

配列を縮小していない場合、図 4-6-7 (1) のように、ランク 0 の配列 A の空いている部分に収集できればよいのですが、集団通信サブルーチンには『送信バッファーと受信バッファーがメモリー上で重なってはいけない』(3-3-5 節参照) という制限があるため、図 4-6-7 (2) のように、新たに受信バッファー用の配列 B を設ける必要があります。しかし、MPI-2 の MPI_IN_PLACE (3-8-1 節参照) という変数を使用すると、この制限を回避することができます (MPI-2 が使用できない環境では使用できません)。

図 4-6-7 (3) に示す M×N の 2 次元配列 A を 2 次元方向に分割する場合を想定します。一般にプロセスによってデータ量が異なるため、MPI_GATHER ではなく MPI_GATHERV を使用して、各プロセスのデータをランク 0 に収集します。図 4-6-7 (4) の [1] で以下の 2 つの配列を作成し、ランク 0 は [2] で、その他のプロセスは [3] で、MPI_GATHERV (付録の「MPI_IN_PLACE の使い方」参照) をコールします。[2] では送信バッファーの代わりに MPI_IN_PLACE を指定します。[3] では受信バッファーは不要なので、[3] の下線部は実際には無視されます。

- IRCNT(i): ランク i のプロセスが担当する要素数を設定します。
- IDISP(i): ランク i のプロセスが担当する最初の要素の、配列の先頭からの変位 (⓪, ⑧, (14)) を設定します。

図 4-6-7 (1)

図 4-6-7 (2)

図 4-6-7 (3)

```
PROGRAM MAIN
INCLUDE 'mpif.h'
PARAMETER(M=～,N=～,NCPU=3)
REAL A(M,N)
INTEGER IRCNT(0:NCPU-1),IDISP(0:NCPU-1)  [1]
CALL MPI_INIT(IERR)
CALL MPI_COMM_SIZE(～,NPROCS,IERR)
CALL MPI_COMM_RANK(～,MYRANK,IERR)
DO IRANK=0,NPROCS-1
  CALL PARA_RANGE
&      (1,N,NPROCS,IRANK,JSTA,JEND)
  IRCNT(IRANK) = M*(JEND-JSTA+1)          [1]
  IDISP(IRANK) = M*(JSTA-1)
ENDDO
CALL PARA_RANGE
&    (1,N,NPROCS,MYRANK,JSTA,JEND)
  :
IF (MYRANK==0) THEN
  CALL MPI_GATHERV
&      (MPI_IN_PLACE,IDUMMY      ,IDUMMY,  [2]
&       A            ,IRCNT,IDISP,MPI_REAL,
&       0,MPI_COMM_WORLD,IERR)
ELSE
  CALL MPI_GATHERV
&      (A(1,JSTA),IRCNT(MYRANK),MPI_REAL,
&       IDUMMY    ,IDUMMY,IDUMMY,IDUMMY   , [3]
&       0,MPI_COMM_WORLD,IERR)
ENDIF
  :
```

図 4-6-7 (4)

(ケース 2) MPI-2 の MPI_IN_PLACE が使用できず、データが連続な場合

MPI-2 の MPI_IN_PLACE の機能が使用できないマシン環境で、前述のケース 1 のように、配列 B を新たに設けずに配列 A の空いている部分に直接収集したい場合、MPI_GATHERV と同等の通信を、1 対 1 通信サブルーチンを使用して行います。

まず図 4-6-8 (2) の [1] で以下の配列を作成します。

- JJSTA(i): ランク i のプロセスが担当する部分の、2 次元方向の最初の要素番号を設定します。
- JJLEN(i): ランク i のプロセスが担当する要素数を設定します。

[2] と [3] が MPI_GATHERV の代替で通信を行う部分です。図 4-6-8 (1) からわかるように、ランク 0 以外のプロセスは、[3] で自分が担当したデータをランク 0 のプロセスに送信し、ランク 0 のプロセスは、[2] で自分以外のプロセスから送られたデータを配列 A のしかるべき部分に受信します。[2] の下線部に示すように、DO ループが「0」からではなく「1」から開始することに注意して下さい。

図 4-6-8 (1)

```
      PROGRAM MAIN
      INCLUDE 'mpif.h'
      PARAMETER(M=～,N=～,NCPU=3)
      REAL A(M,N)
      INTEGER ISTATUS(MPI_STATUS_SIZE)
      INTEGER JJSTA(0:NCPU-1),JJLEN(0:NCPU-1)  [1]
      CALL MPI_INIT(IERR)
      CALL MPI_COMM_SIZE(～,NPROCS,IERR)
      CALL MPI_COMM_RANK(～,MYRANK,IERR)
      DO IRANK=0,NPROCS-1
        CALL PARA_RANGE
     &      (1,N,NPROCS,IRANK,JSTA,JEND)
        JJSTA(IRANK) = JSTA
        JJLEN(IRANK) = M*(JEND-JSTA+1)          [1]
      ENDDO
      CALL PARA_RANGE
     &    (1,N,NPROCS,MYRANK,JSTA,JEND)
        :
      IF (MYRANK==0) THEN
        DO IRANK=1,NPROCS-1
          CALL MPI_IRECV(A(1,JJSTA(IRANK)),
     &      JJLEN(IRANK),MPI_REAL,IRANK,1,        [2]
     &      MPI_COMM_WORLD,IREQ,IERR)
          CALL MPI_WAIT(IREQ,ISTATUS,IERR)
        ENDDO
      ELSE
        CALL MPI_ISEND(A(1,JSTA),
     &       JJLEN(MYRANK),MPI_REAL,0,1,          [3]
     &       MPI_COMM_WORLD,IREQ,IERR)
        CALL MPI_WAIT(IREQ,ISTATUS,IERR)
      ENDIF
        :
```

図 4-6-8 (2)

(ケース 3) データが不連続で派生データ型を使う場合

ケース 1 とケース 2 では、各プロセスの送信データがメモリー上で連続していました。図 4-6-9 (1) では配列 A を 1 次元方向に分割しているので、各プロセスの送信データはメモリー上でとびとびになっています (Fortran の場合)。とびとびのデータを、いったん連続した一時配列に入れてから送信（受信はその逆）する方法もありますが、それが面倒な場合、3-5 節で説明した派生データ型を使用します。

図 4-6-9 (3) の [1] で、MPI_TYPE_CREATE_SUBARRAY (3-5-3-1 節参照) (MPI-2 が使用できる場合のみ) を使用して、ランク i のプロセスの担当部分の派生データ型を配列 ITYPE(i) に作成します (図 4-6-9 (2) 参照)。

ケース 1 で使用した MPI_GATHERV では、受信バッファーのデータ型を 1 種類しか指定できない（つまりプロセスごとに配列で別々に指定できない）ため、ケース 2 と同じ方法で通信を行います。

[2] の MPI_IRECV は、『ランク IRANK のプロセスから送られたデータを、配列 A（の先頭）から始まる 1 個の ITYPE(IRANK) 型の受信バッファーに受信する』という意味になり、[3] の MPI_ISEND は、『配列 A（の先頭）から始まる 1 個の ITYPE(MYRANK) 型のデータを、ランク 0 のプロセスに送信する』という意味になります。

図 4-6-9 (1)

図 4-6-9 (2)

```
PROGRAM MAIN
INCLUDE 'mpif.h'
PARAMETER(M=～,N=～,NCPU=3)
REAL A(M,N)
INTEGER ISTATUS(MPI_STATUS_SIZE)
INTEGER ISIZE(2),ISSIZE(2),ISTART(2)        [1]
INTEGER ITYPE(0:NCPU-1)                     [1]
CALL MPI_INIT(IERR)
CALL MPI_COMM_SIZE(～,NPROCS,IERR)
CALL MPI_COMM_RANK(～,MYRANK,IERR)
ISIZE(1) = M
ISIZE(2) = N
ISSIZE(2) = N
ISTART(2) = 0
DO IRANK=0,NPROCS-1                         [1]
  CALL PARA_RANGE
&      (1,M,NPROCS,IRANK,ISTA,IEND)
  ISSIZE(1) = IEND-ISTA+1
  ISTART(1) = ISTA-1
  CALL MPI_TYPE_CREATE_SUBARRAY(2,ISIZE,
&       ISSIZE,ISTART,MPI_ORDER_FORTRAN,
&       MPI_REAL,ITYPE(IRANK),IERR)
  CALL MPI_TYPE_COMMIT(ITYPE(IRANK),IERR)[1]
ENDDO
CALL PARA_RANGE
&    (1,M,NPROCS,MYRANK,ISTA,IEND)
   :
IF (MYRANK==0) THEN
  DO IRANK=1,NPROCS-1
    CALL MPI_IRECV(A,1,ITYPE(IRANK),IRANK,
&       1,MPI_COMM_WORLD,IREQ,IERR)         [2]
    CALL MPI_WAIT(IREQ,ISTATUS,IERR)
  ENDDO
ELSE
  CALL MPI_ISEND(A,1,ITYPE(MYRANK),0,
&           1,MPI_COMM_WORLD,IREQ,IERR)     [3]
  CALL MPI_WAIT(IREQ,ISTATUS,IERR)
ENDIF
   :
```

図 4-6-9 (3)

4-6-3-2　MPI_ALLGATHERV の代替

4-3 節の『計算パターン 1」で並列化した場合、各プロセスは、自分が計算した部分のデータを、他の全プロセスに送信するという通信パターンが現れます。またこのパターンは、並列化の途中でデバッグを行う場合にも使用します (4-8-1 節参照)。通信の方法は前節とほぼ同様です。

(ケース 4) MPI-2 の MPI_IN_PLACE が使用でき、データが連続な場合

配列を縮小していない場合、図 4-6-10 (1) のように、各プロセスの配列 A の空いている部分に送信できればよいのですが、集団通信サブルーチンには『送信バッファーと受信バッファーがメモリー上で重なってはいけない』(3-3-5 節参照) という制限があるため、図 4-6-10 (2) のように、新たに受信バッファー用の配列 B を設ける必要があります。しかし前述のケース 1 と同様に、MPI-2 の MPI_IN_PLACE (3-8-1 節参照) という変数を使用すると、この制限を回避することができます (MPI-2 が使用できない環境では使用できません)。

図 4-6-10 (3) に示す M × N の 2 次元配列 A を 2 次元方向に分割する場合を想定します。一般にプロセスによってデータ量が異なるため、MPI_ALLGATHER ではなく MPI_ALLGATHERV を使用して、各プロセスのデータを他の全プロセスに送信します。図 4-6-10 (4) の [1] で以下の 2 つの配列を作成し、[2] で MPI_ALLGATHERV (付録の「MPI_IN_PLACE の使い方」参照) をコールします。[2] では送信バッファーの代わりに MPI_IN_PLACE を指定します。

- IRCNT(i): ランク i のプロセスが担当する要素数を設定します。
- IDISP(i): ランク i のプロセスが担当する最初の要素の、配列の先頭からの変位 (⓪, ⑧, (14)) を設定します。

図 4-6-10 (1)

図 4-6-10 (2)

図 4-6-10 (3)

```
PROGRAM MAIN
INCLUDE 'mpif.h'
PARAMETER(M=～,N=～,NCPU=3)
REAL A(M,N)
INTEGER IRCNT(0:NCPU-1),IDISP(0:NCPU-1)  [1]
CALL MPI_INIT(IERR)
CALL MPI_COMM_SIZE(～,NPROCS,IERR)
CALL MPI_COMM_RANK(～,MYRANK,IERR)
DO IRANK=0,NPROCS-1
  CALL PARA_RANGE
&      (1,N,NPROCS,IRANK,JSTA,JEND)
  IRCNT(IRANK) = M*(JEND-JSTA+1)          [1]
  IDISP(IRANK) = M*(JSTA-1)
ENDDO
CALL PARA_RANGE
&    (1,N,NPROCS,MYRANK,JSTA,JEND)
  :
CALL MPI_ALLGATHERV
&      (MPI_IN_PLACE,IDUMMY      ,IDUMMY,  [2]
&       A            ,IRCNT,IDISP,MPI_REAL,
&       MPI_COMM_WORLD,IERR)
  :
```

図 4-6-10 (4)

(ケース 5) MPI-2 の MPI_IN_PLACE が使用できず、データが連続な場合

MPI-2 の MPI_IN_PLACE の機能が使用できないマシン環境で、前述のケース 4 のように、配列 B を新たに設けずに配列 A の空いている部分に直接受信したい場合、MPI_ALLGATHERV と同等の通信を、集団通信サブルーチン MPI_BCAST を使用して行います。

まず図 4-6-11 (2) の [1] で以下の配列を作成します。

- JJSTA(i): ランク i のプロセスが担当する部分の、2 次元方向の最初の要素番号を設定します。
- JJLEN(i): ランク i のプロセスが担当する要素数を設定します。

[2] が MPI_ALLGATHERV の代替で通信を行う部分です。図 4-6-11 (1) から分かるように、各プロセスは自分が担当するデータを他の全プロセスに送信すればよいので、MPI_BCAST を使用して通信を行います。[2] の下線部に示すように、送信元のプロセスが 0, 1, 2 と順に変化します。

図 4-6-11 (1)

```
  PROGRAM MAIN
  INCLUDE 'mpif.h'
  PARAMETER(M=～,N=～,NCPU=3)
  REAL A(M,N)
  INTEGER ISTATUS(MPI_STATUS_SIZE)
  INTEGER JJSTA(0:NCPU-1),JJLEN(0:NCPU-1)  [1]
  CALL MPI_INIT(IERR)
  CALL MPI_COMM_SIZE(～,NPROCS,IERR)
  CALL MPI_COMM_RANK(～,MYRANK,IERR)
  DO IRANK=0,NPROCS-1
    CALL PARA_RANGE
&       (1,N,NPROCS,IRANK,JSTA,JEND)
    JJSTA(IRANK) = JSTA
    JJLEN(IRANK) = M*(JEND-JSTA+1)          [1]
  ENDDO
  CALL PARA_RANGE
&     (1,N,NPROCS,MYRANK,JSTA,JEND)
    :
  DO IRANK=0,NPROCS-1
    CALL MPI_BCAST(A(1,JJSTA(IRANK)),
&            JJLEN(IRANK),MPI_REAL,IRANK,   [2]
&            MPI_COMM_WORLD,IERR)
  ENDDO
    :
```

図 4-6-11 (2)

(ケース 6) データが不連続で派生データ型を使う場合

ケース 4 とケース 5 では、各プロセスの送信データがメモリー上で連続していました。図 4-6-12 (1) では配列 A を 1 次元方向に分割しているので、各プロセスの送信データはメモリー上でとびとびになっています (Fortran の場合)。とびとびのデータを、いったん連続した一時配列に入れてから送信（受信はその逆）する方法もありますが、それが面倒な場合、前述のケース 3 と同様に、3-5 節で説明した派生データ型を使用します。

図 4-6-12 (3) の [1] で、MPI_TYPE_CREATE_SUBARRAY (3-5-3-1 節参照) (MPI-2 が使用できる場合のみ) を使用して、ランク i のプロセスの担当部分の派生データ型を配列 ITYPE(i) に作成します (図 4-6-12 (2) 参照)。

ケース 4 で使用した MPI_ALLGATHERV では、受信バッファーのデータ型を 1 種類しか指定できない（つまりプロセスごとに配列で別々に指定できない）ため、ケース 5 と同じ方法で通信を行います。

[2] の MPI_BCAST は、送信元のプロセス (ランク IRANK) では、『配列 A（の先頭）から始まる 1 個の ITYPE(IRANK) 型のデータを、他の全てのプロセスに送信する』という意味になり、その他のプロセスでは、『ランク IRANK のプロセスから送られたデータを、配列 A（の先頭）から始まる 1 個の ITYPE(IRANK) 型の受信バッファーに受信する』という意味になります。

図 4-6-12 (1)

図 4-6-12 (2)

```
      PROGRAM MAIN
      INCLUDE 'mpif.h'
      PARAMETER(M=～,N=～,NCPU=3)
      REAL A(M,N)
      INTEGER ISTATUS(MPI_STATUS_SIZE)
      INTEGER ISIZE(2),ISSIZE(2),ISTART(2)        [1]
      INTEGER ITYPE(0:NCPU-1)                     [1]
      CALL MPI_INIT(IERR)
      CALL MPI_COMM_SIZE(～,NPROCS,IERR)
      CALL MPI_COMM_RANK(～,MYRANK,IERR)
      ISIZE(1) = M
      ISIZE(2) = N
      ISSIZE(2) = N                               [1]
      ISTART(2) = 0
```

```
 DO IRANK=0,NPROCS-1
    CALL PARA_RANGE
&       (1,M,NPROCS,IRANK,ISTA,IEND)
    ISSIZE(1) = IEND-ISTA+1
    ISTART(1) = ISTA-1
    CALL MPI_TYPE_CREATE_SUBARRAY(2,ISIZE,
&        ISSIZE,ISTART,MPI_ORDER_FORTRAN,       [1]
&        MPI_REAL,ITYPE(IRANK),IERR)
    CALL MPI_TYPE_COMMIT(ITYPE(IRANK),IERR)
 ENDDO
 CALL PARA_RANGE
&     (1,M,NPROCS,MYRANK,ISTA,IEND)
    :
 DO IRANK=0,NPROCS-1
    CALL MPI_BCAST(A,1,ITYPE(IRANK),IRANK,
&        MPI_COMM_WORLD,IERR)                   [2]
 ENDDO
    :
```

図 4-6-12 (3)

4-6-3-3　MPI_ALLTOALLV の代替

本節で説明する通信パターンは、多次元 FFT、ADI 法などを並列化する場合に現れます。なお、前者は並列版の数値計算ライブラリー (6 章参照) で多次元 FFT のサブルーチンが提供されていることが多く、後者は本節の方法よりもパイプライン法 (4-6-7 節) やツイスト分割法 (4-6-8 節) を使用した方が通信量が少なくなります。

例えば並列プログラムの前半では図 4-6-13 (1) のように配列 A を 2 次元目でブロック分割して処理し、プログラムの後半では図 4-6-13 (2) のように配列 B (または A) を 1 次元目でブロック分割して処理する場合、後半の処理を行う前に、配列 A から B (または A) へ矢印に示す通信が必要になります。

図 4-6-13 (1)

図 4-6-13 (2)

図 4-6-14 (1)

図 4-6-14 (2)

通信方法 1

図 4-6-13 の通信パターンは、MPI の集団通信サブルーチン MPI_ALLTOALLV に相当します。図 4-6-13 では配列を縮小 (4-5-7 節参照) していないので、例えば〓で囲んだ②の 6 つの要素はメモリー上で不連続になっています。不連続なデータをいったん送（受）信バッファーに圧縮するのは面倒なので、ここでは図 4-6-14 (1) のような派生データ型 (3-5 節参照) を作成して通信で使用することにします。

図 4-6-14 (1) で、ISTYPE(i) はランク i のプロセスに送信するデータが入っている部分の派生データ型、IRTYPE(i) はランク i のプロセスから受信したデータが入る部分の派生データ型を示します。図から分かるように、相手プロセスごとに別々の派生データ型を用意する必要があります。ところがサブルーチン MPI_ALLTOALLV のデータ型を指定する引数 (sendtype と recvtype) には 1 種類しか指定することができません。このような場合、MPI-2 で追加された集団通信サブルーチン MPI_ALLTOALLW（付録参照）を使用します (MPI-2 を使用できないマシン環境では使用する事はできません)。

なお図 4-6-13 では、図 4-6-14 (2) のように配列 A から配列 A に通信せずに、配列 B に通信しています。この理由ですが、集団通信サブルーチンには「送信バッファーと受信バッファーは重なってはならない」という制限があります (3-3-5 節参照)。配列 A から配列 A に通信した場合、図 4-6-13 で例えばランク 0 の①の部分が、送信バッファーと受信バッファーの両方に属するため、この制限に引っ掛かります。このため、配列 A とは別の配列 B を受信バッファーとして使用します (MPI_ALLTOALLW では、MPI-2 で追加された MPI_IN_PLACE (3-8 節参照) も使用できません)。プログラム例を図 4-6-15 (1) に示します。

- 図 4-6-15 (1) の [1] で、図 4-6-14 (1) に示す派生データ型を、3-5-3-2 節で説明した自作のサブルーチン PARA_TYPE_BLOCK2 を使用して作成し (MPI-2 が使用できるマシン環境の場合は、MPI-2 で提供されている MPI_TYPE_CREATE_SUBARRAY (3-5-3-1 節参照) を使用して下さい)、配列 ISTYPE と IRTYPE に入れます。
- [2] で、MPI_ALLTOALLW の引数（詳細は付録参照）で指定する以下の配列を設定します。
  - 配列 ISCNT(i) にはランク i に送信するデータ数を設定し、配列 IRCNT(i) にはランク i から受信するデータ数を設定します。本例では派生データ型を使用するので、どちらも「1（個）」を設定します。
  - 配列 ISDISP(i) に、送信バッファー A の先頭アドレスから、ランク i に送信するデータが入っている場所までの変位（単位はバイト）を設定します。同様に IRDISP(i) に、受信バッファー B の先頭アドレスから、ランク i から受信したデータを入れる場所までの変位（単位はバイト）を設定します。派生データ型 ISTYPE, IRTYPE の起点は配列 A, B の先頭アドレスなので (3-5-3 節参照)、先頭アドレスを示す「0」を設定します。
- [3] を実行すると図 4-6-13 の矢印の通信が行われます。

通信方法 2

図 4-6-13 では受信バッファーとして配列 B を使用しましたが、図 4-6-14 (2) のように配列 B を使用せずに配列 A から配列 A に直接通信する方法を、図 4-6-15 (2) のプログラムで説明します。

- [5] で図 4-6-15 (1) と同様に図 4-6-14 (1) に示す派生データ型 ISTYPE, IRTYPE を作成します。
- [6] のループが反復し、[7] で自分以外の全プロセスに対してデータを送信し、[8] で自分以外の全プロセスからデータを受信します。このとき IREQ1, IREQ2 は、[4] に示すように各プロセスごとの配列となります。
- [9] で、[7] と [8] の通信に対する MPI_WAIT を実行します。これで図 4-6-14 (2) の矢印に示す通信が行われます。

通信方法 3，4

上記の「通信方法 2」では自分以外の全プロセスに対して一気に送受信を行いましたが、この方法では 1 つずつ順に通信を行うので、通信する相手のプロセス数が多い時は「通信方法 2」よりも無難だと思われます。

- 図 4-6-15 (3) の [10] で I = 1 のとき、図 4-6-14 (2) の太い矢印に示すように、各プロセスは自分よりランクが 1 つ大きいプロセスに対して送信し (ランク 2 はランク 0 に送信)、ランクが 1 つ小さいプロセスから受信します。
- 次に [10] で I = 2 となり、図 4-6-14 (2) の点線の矢印に示すように、ランクが 2 つ大きいプロセスに対して送信し、ランクが 2 つ小さいプロセスから受信します。
- このように [10] の I を 1 つずつ増やしながら、上記の送受信を自分以外の全ての相手と行います。
- 図 4-6-15 (3) と同様の通信を MPI_SENDRECV (4-6-2 節，付録参照) を用いて行ったのが図 4-6-15 (4) です。

```
      :
      INCLUDE 'mpif.h'
      PARAMETER(M=5,N=7,NCPU=3)
      REAL A(M,N),B(M,N)
      INTEGER ISCNT (0:NCPU-1),IRCNT (0:NCPU-1)
      INTEGER ISDISP(0:NCPU-1),IRDISP(0:NCPU-1)
①     INTEGER ISTYPE(0:NCPU-1),IRTYPE(0:NCPU-1)
      CALL MPI_INIT(IERR)
      CALL MPI_COMM_SIZE
     &    (MPI_COMM_WORLD,NPROCS,IERR)
      CALL MPI_COMM_RANK
     &    (MPI_COMM_WORLD,MYRANK,IERR)
      CALL PARA_RANGE
     &    (1,M,NPROCS,MYRANK,ISTA,IEND)
      CALL PARA_RANGE
     &    (1,N,NPROCS,MYRANK,JSTA,JEND)
      DO IRANK=0,NPROCS-1
        CALL PARA_RANGE(1,M,NPROCS,IRANK,
                        IISTA,IIEND)
        CALL PARA_TYPE_BLOCK2(1,M,1,IISTA,
     &        IIEND,JSTA,JEND,MPI_REAL,ITYPE)
        ISTYPE(IRANK) = ITYPE
        CALL PARA_RANGE(1,N,NPROCS,IRANK,          [1]
     &                  JJSTA,JJEND)
        CALL PARA_TYPE_BLOCK2(1,M,1,ISTA,IEND,
     &        JJSTA,JJEND,MPI_REAL,ITYPE)
        IRTYPE(IRANK) = ITYPE
      ENDDO
      DO IRANK=0,NPROCS-1
        ISCNT (IRANK)  = 1
        IRCNT (IRANK)  = 1                         [2]
        ISDISP(IRANK) = 0
        IRDISP(IRANK) = 0
      ENDDO
      :
      CALL MPI_ALLTOALLW(A,ISCNT,ISDISP,ISTYPE,
     &                   B,IRCNT,IRDISP,IRTYPE,[3]
     &                   MPI_COMM_WORLD,IERR)
      :
```

図 4-6-15 (1)　通信方法 1

```
      :
      INCLUDE 'mpif.h'
      PARAMETER(M=5,N=7,NCPU=3)
      REAL A(M,N)
      IREQ1(0:NCPU-1),IREQ2(0:NCPU-1)              [4]
      INTEGER ISTATUS(MPI_STATUS_SIZE)
      左図の①と同じ                                [5]
      :
      DO IRANK=0,NPROCS-1                          [6]
        IF (IRANK/=MYRANK) THEN
          CALL MPI_ISEND(A,1,ISTYPE(IRANK),
     &       IRANK,1,MPI_COMM_WORLD,               [7]
     &       IREQ1(IRANK),IERR)
          CALL MPI_IRECV(A,1,IRTYPE(IRANK),
     &       IRANK,1,MPI_COMM_WORLD,               [8]
     &       IREQ2(IRANK),IERR)
        ENDIF
      ENDDO
      DO IRANK=0,NPROCS-1
        IF (IRANK/=MYRANK) THEN
          CALL MPI_WAIT
     &       (IREQ1(IRANK),ISTATUS,IERR)           [9]
          CALL MPI_WAIT
     &       (IREQ2(IRANK),ISTATUS,IERR)
        ENDIF
      ENDDO
      :
```

図 4-6-15 (2)　通信方法 2

```
      :
      INCLUDE 'mpif.h'
      PARAMETER(M=5,N=7,NCPU=3)
      REAL A(M,N)
      INTEGER ISTATUS(MPI_STATUS_SIZE)
      左図の①と同じ
      :
      DO I=1,NPROCS-1                              [10]
        IUP   = MOD(MYRANK+I        ,NPROCS)
        IDOWN = MOD(MYRANK-I+NPROCS,NPROCS)
        CALL MPI_ISEND(A,1,ISTYPE(IUP),
     &      IUP   ,1,MPI_COMM_WORLD,IREQ1,IERR)
        CALL MPI_IRECV(A,1,IRTYPE(IDOWN),
     &      IDOWN,1,MPI_COMM_WORLD,IREQ2,IERR)
        CALL MPI_WAIT(IREQ1,ISTATUS,IERR)
        CALL MPI_WAIT(IREQ2,ISTATUS,IERR)
      ENDDO
      :
```

図 4-6-15 (3)　通信方法 3

```
      :
      DO I=1,NPROCS-1
        IUP   = MOD(MYRANK        +I,NPROCS)
        IDOWN = MOD(MYRANK+NPROCS-I,NPROCS)
        CALL MPI_SENDRECV
     &      (A,1,ISTYPE(IUP)  ,IUP  ,1,
     &       A,1,IRTYPE(IDOWN),IDOWN,1,
     &       MPI_COMM_WORLD,ISTATUS,IERR)
      ENDDO
      :
```

図 4-6-15 (4)　通信方法 4

4-6-4　合計（内積）／最大（最小）

図 4-6-16 (1) を並列化したプログラムを図 4-6-16 (2) に示します。

図 4-6-16 (2) の最初のループでは、②で合計を、③で内積を求めています。このように、並列化したループの中に合計や内積を求める計算が含まれている場合、ループを終了した直後に、「各プロセスが求めた合計の総合計」を求めるために、MPI_SUM を使用して、MPI_REDUCE または MPI_ALLREDUCE で通信を行う必要があります。②と③は、⑤で全プロセスが SUM1 と SUM2 を参照するので、MPI_ALLREDUCE で通信を行い、結果を全プロセスに送信します。また SUM1 と SUM2 を別々に MPI_ALLREDUCE で通信すると、4-1 節で述べたように通信の立上り時間によるオーバーヘッドが 2 回かかってしまうので、⑦の作業配列 WORKS と WORKR を使用し、⑨では 1 回で通信します。

図 4-6-16 (2) の④で最大値を求めています。このように、並列化したループの中に、最大（最小）値を求める計算が含まれている場合、ループを終了した直後に、「各プロセスが求めた最大（最小）値の全体の最大（最小）値」を求めるために、MPI_MAX (MPI_MIN) を使用して、MPI_REDUCE または MPI_ALLREDUCE で通信を行う必要があります。④は⑥で（例えば）ランク 0 のプロセスだけが結果を得ればよいので、MPI_ALLREDUCE ではなく、⑩のように MPI_REDUCE で通信します。

4 章では合計を求めるプログラムを並列化の例題として取り上げました。しかし実際のプログラムでは、合計、内積、最大値（最小値を）を求めるだけのループが独立しているというのはまれで、①のように派手な計算の片隅（通常はループの最後）に、②，③，④が『さりげなく』現れているのが一般的です。しかしループを⑧のように並列化したとき、そのループの中に②，③，④が含まれているのを見落とすと、結果がおかしくなりますので注意して下さい。

```
    :
  SUM1 = 0.0
  SUM2 = 0.0
  AMAX = 0.0
  DO I = 1, N
    A(I) = A(I) + B(I)
    C(I) = C(I) + D(I)
    E(I) = E(I) + F(I)                ①
    G(I) = G(I) + H(I)
    X(I) = X(I) + Y(I)
    SUM1 = SUM1 + A(I)                ②
    SUM2 = SUM2 + P(I)*Q(I)           ③
    IF (A(I) > AMAX) AMAX = A(I)      ④
  ENDDO
  DO I = 1, N
    G(I) = G(I)*SUM1 + SUM2           ⑤
  ENDDO
  PRINT *,AMAX                        ⑥
    :
```

図 4-6-16 (1)

→

```
    :
  INCLUDE 'mpif.h'
  DIMENSION WORKS(2),WORKR(2)                    ⑦
  SUM1 = 0.0
  SUM2 = 0.0
  AMAX = 0.0
  DO I = ISTA, IEND                              ⑧
    A(I) = A(I) + B(I)
    C(I) = C(I) + D(I)
    E(I) = E(I) + F(I)                           ①
    G(I) = G(I) + H(I)
    X(I) = X(I) + Y(I)
    SUM1 = SUM1 + A(I)                           ②
    SUM2 = SUM2 + P(I)*Q(I)                      ③
    IF (A(I) > AMAX) AMAX = A(I)                 ④
  ENDDO
  WORKS(1) = SUM1
  WORKS(2) = SUM2
  CALL MPI_ALLREDUCE(WORKS,WORKR,2,MPI_REAL,     ⑨
&                MPI_SUM,MPI_COMM_WORLD,IERR)
  SUM1 = WORKR(1)
  SUM2 = WORKR(2)
  CALL MPI_REDUCE(AMAX,AAMAX,1,MPI_REAL,
&            MPI_MAX,0,MPI_COMM_WORLD,IERR)      ⑩
  AMAX = AAMAX
  DO I = ISTA, IEND
    G(I) = G(I)*SUM1 + SUM2                      ⑤
  ENDDO
  IF (MYRANK==0) PRINT *,AMAX                    ⑥
    :
```

図 4-6-16 (2)

一方、並列化するループの外にタイムステップのループがある場合、合計（または内積）を求める演算には図 4-6-16 (3) のような 3 つのパターンがあります。

- ①ではタイムステップごとに内側のループの合計を求め、結果を書き出しています。
- ②ではタイムステップ全体での合計を求め、結果を最後に 1 回だけ書き出しています。
- ③ではタイムステップごとに、タイムステップの最初からその時点までの累計を求め、結果を書き出しています。

内側のループを並列化した場合、それぞれのパターンに応じて図 4-6-16 (4) のように修正します。

- ①では内側のループが終了するたびに MPI_REDUCE を行います。
- ②ではタイムステップループが終了した後に 1 回だけ MPI_REDUCE を行います。
- ③では少し注意が必要です。まず内側のループに入る前に SUM3 に入っているその時点までの累計値を一時変数 TEMP に保管します。そして SUM3 をゼロクリアしてから内側のループを実行します。ループ終了後に MPI_REDUCE を行い、その結果が入る SSUM3 と、TEMP に保管した値を合計して SUM3 に戻します。

```
  :
SUM2 = 0.0                ②
SUM3 = 0.0                ③
DO タイムステップ・ループ
  :
  SUM1 = 0.0              ①
  DO I = 1, N
    SUM1 = SUM1 + A(I) ①
    SUM2 = SUM2 + B(I) ②
    SUM3 = SUM3 + C(I) ③
  ENDDO
  PRINT *,SUM1,SUM3       ①③
  :
ENDDO
PRINT *,SUM2              ②
  :
```

図 4-6-16 (3)

→

```
  :
INCLUDE 'mpif.h'
  :
SUM2 = 0.0                ②
SUM3 = 0.0                ③
DO タイムステップ・ループ
  :
  SUM1 = 0.0              ①
  TEMP = SUM3             ③
  SUM3 = 0.0              ③
  DO I = ISTA, IEND
    SUM1 = SUM1 + A(I) ①
    SUM2 = SUM2 + B(I) ②
    SUM3 = SUM3 + C(I) ③
  ENDDO
  CALL MPI_REDUCE(SUM1,SSUM1,～) ①
  SUM1 = SSUM1                   ①
  CALL MPI_REDUCE(SUM3,SSUM3,～) ③
  SUM3 = TEMP + SSUM3            ③
  IF (MYRANK==0) PRINT *,SUM1,SUM3 ①③
  :
ENDDO
CALL MPI_REDUCE(SUM2,SSUM2,～)   ②
SUM2 = SSUM2                     ②
IF (MYRANK==0) PRINT *,SUM2      ②
  :
```

図 4-6-16 (4)

4-6-5　特定の配列要素の参照

本例は、4-6-1 節の「メッセージ交換はどんなときに必要か」の例と同じです。実際のプログラムではこのような例はめったに現れませんが、もし現れた場合には以下のような対応をとる必要がありますので注意して下さい。

図 4-6-17 (1) の下線部に、配列の添字にループ反復『I』以外の値 (この例では『1』) が現れています。まず配列を縮小せずに並列化した図 4-6-17 (2) では、①のループを並列化したため、ループを抜けた直後にはランク 0 のプロセスのみが A(1) の値を持っています。③で全プロセスが A(1) の値を参照しますが、ランク 1 と 2 のプロセスは A(1) の値を持っていないため、③を行う前に②でランク 0 からその他のプロセスに A(1) の値を送信します (図 4-6-17 (4) 参照)。

一方、配列を縮小して並列化した図 4-6-17 (3) では、ランク 1 と 2 のプロセスは A(1) に自分の担当する要素の値を持っています。このため、②のようにランク 0 からその他のプロセスに A(1) の値を送信すると、図 4-6-17 (5) に示すように、ランク 1 と 2 のプロセスでは自分の担当する A(1) の値が壊されてしまいます（図中の×）。これを防ぐため、ランク 0 のプロセスは A(1) の値を④で一時変数 TEMP に入れ、それを⑤でその他のプロセスに送信し、⑥では TEMP の値を参照するようにします。

```
   :
 DIMENSION A(9),B(9)
   :
 DO I = 1, 9
   A(I) = ～
 ENDDO
 DO I = 1, 9
   B(I) = B(I)*A(1)
 ENDDO
   :
```

図 4-6-17 (1)

```
   :
 DIMENSION A(9),B(9)
   :
 DO I = ISTA, IEND                ①
   A(I) = ～
 ENDDO
 CALL MPI_BCAST(A(1),1,MPI_REAL,
&  0,MPI_COMM_WORLD,IERR) ･･･ ②
 DO I = ISTA, IEND
   B(I) = B(I)*A(1)                ③
 ENDDO
   :
```

図 4-6-17 (2)

```
   :
 DIMENSION A(3),B(3)
   :
 DO I = 1, IEND-ISTA+1
   A(I) = ～
 ENDDO
 IF (MYRANK == 0) TEMP = A(1) ④
 CALL MPI_BCAST(TEMP,1,MPI_REAL,
&  0,MPI_COMM_WORLD,IERR)       ⑤
 DO I = 1, IEND-ISTA+1
   B(I) = B(I)*TEMP              ⑥
 ENDDO
   :
```

図 4-6-17 (3)

図 4-6-17 (4)

図 4-6-17 (5)

4-6-6　重ね合わせ

通常、MPI_(ALL)REDUCE などの通信しながら演算を行うサブルーチンは、並列化したループ内に合計、内積、最大（小）値などを求める計算が含まれている場合、ループ終了後に「合計の合計」や「最大の最大」を求めるために使用しますが(4-6-4 節参照)、そのほかにも便利な使用方法があります。

4-6-6-1　データの収集

図 4-6-18 (1) の DO ループを並列化する際、何らかの理由で、1〜6 の各ループ反復を図 4-6-18 (2) の配列 MAP 内のランクが担当するとします。また MAP 内のランク値は不規則であるとします（計算領域が非構造格子で構成されているプログラムなどでこのパターンが現れます)。

図 4-6-18 (3) に MAP を使って並列化したプログラムを示します。DO ループを 3 プロセスで並列に実行し、DO ループを抜けた直後、配列 X の中身は図 4-6-18 (4) の上部のようになります。

次に、各プロセスが計算した配列 X の要素をランク 0 に収集します。「正しい」（通信量が最も少ない）収集方法を図 4-6-18 (4) の下部に示します。まず各プロセスは自分が担当した部分のデータを圧縮し（一般にメモリー上でとびとびになっているため)、それを MPI_GAHTERV などでランク 0 に収集し、ランク 0 が配列 X の元の位置に復元します。しかしこのようにデータを圧縮、復元するのは若干面倒です。

このような場合、以下で説明する重ね合わせを用いると、圧縮、復元をせずに簡単にデータの収集を行うことができます。ただし、後述するように通信時間がかかります。

なお、3-5-2 節で紹介した MPI_TYPE_CREATE_INDEXED_BLOCK を使って、とびとびのデータの派生データ型を作成すると、圧縮、復元は不要になり、通信時間も短くて済みますが、以下では説明は省略します。

```
REAL X(6)
  :
DO I=1,6
  X(I) = I
ENDDO
PRINT *,X
  :
```

図 4-6-18 (1)

図 4-6-18 (2)

```
INTEGER MAP(6)
  :
DO I=1,6
  IF (MYRANK==MAP(I)) X(I) = I
ENDDO
配列Xをランク0に収集
IF (MYRANK==0) PRINT *,X
```

図 4-6-18 (3)

図 4-6-18 (4)

重ね合せを使用したプログラムを図 4-6-18 (5) に、データの動きを図 4-6-18 (7) に示します。各プロセスは、図 4-6-18 (5) の①で配列 X 全体をゼロクリア（これが前提になります）してから、②で自分の担当部分を計算します。計算終了後、③で配列 X 全体を MPI_REDUCE で通信すると、配列 X の各要素が要素ごとに加算されてランク 0 の配列 XX に入り、図 4-6-18 (4) と同じ結果を簡単に得ることができます。

なお 3-3-5 節で説明したように、MPI_REDUCE では、送信バッファー X と受信バッファー XX は別の配列にしなければならないので注意して下さい (ただし、MPI-2 で追加された MPI_IN_PLACE という変数を使用すると、この制限を回避することができます。詳細は 3-8-1 節を参照して下さい。)。

重ね合わせでは、値の入っていない部分のゼロクリアが前提になります。図 4-6-18 (5) では計算前に①で配列 X 全体をゼロクリアしましたが、図 4-6-18 (6) の④に示すように、通信前に配列 X の自分の担当部分以外の要素のみをゼロクリアする方法もあります。

```
  :
DO I=1,6
  X(I) = 0.0                                  ①
ENDDO
DO I=1,6
  IF (MYRANK==MAP(I)) X(I) = I                ②
ENDDO
CALL MPI_REDUCE(X,XX,6,MPI_REAL,MPI_SUM,0,
&                MPI_COMM_WORLD,IERR)          ③
IF (MYRANK==0) PRINT *,XX
  :
```

図 4-6-18 (5)

```
  :
DO I=1,6
  IF (MYRANK==MAP(I)) X(I) = I
ENDDO
DO I=1,6
  IF (MYRANK/=MAP(I)) X(I) = 0.0              ④
ENDDO
CALL MPI_REDUCE(X,XX,6,MPI_REAL,MPI_SUM,0,
&                MPI_COMM_WORLD,IERR)
IF (MYRANK==0) PRINT *,XX
  :
```

図 4-6-18 (6)

図 4-6-18 (7)

図 4-6-19 (1)～(3) のように、ブロック分割したループに対して重ね合わせを行った場合、ゼロクリアは⑤（計算前）、または⑥（通信前）のように行います。

重ね合せは図 4-6-18 (4) の「正しい」収集方法よりも簡単なので、例えば並列プログラムのテスト段階で、とりあえず各プロセスが計算した結果を簡単に収集して単体プログラムの結果と比較したい場合などに用いることができます。

ただし、余分なデータ（ゼロ）を通信するため、図 4-6-18 (4) の「正しい」通信方法に比べて通信時間がかかります。本番プログラムの通信時間がクリティカルな部分では用いないで下さい。

（注）上図の ISTA,IEND は
ランク 1 用

図 4-6-19 (1)

```
  :
DO I=1,6
  X(I) = 0.0                                  ⑤
ENDDO
DO I=ISTA,IEND
  X(I) = I
ENDDO
CALL MPI_REDUCE(～)
  :
```

図 4-6-19 (2)

```
  :
DO I=ISTA,IEND
  X(I) = I
ENDDO
DO I=1,ISTA-1
  X(I) = 0.0
ENDDO                                         ⑥
DO I=IEND+1,6
  X(I) = 0.0
ENDDO
CALL MPI_REDUCE(～)
  :
```

図 4-6-19 (3)

4-6-6-2　間接アドレスを使用した計算

重ね合せの別の例を図 4-6-20 (1) (2) で説明します。図 4-6-20 (1) の配列 A と X に値が入っており、配列 A の値を、配列 X (のインデックス配列 IND で示す場所) に加算します。

このように間接アドレスを使用する計算は、有限要素法（陽解法）で要素の値から節点の値を求める計算（またはその逆）などで現れます。

配列 A と X を図 4-6-20 (3) に示すようにブロック分割して並列化する場合、配列 A を配列 X に加える矢印がプロセスの境界を越えているので、変数 A の境界を越えたデータを他のプロセスに通信する必要があります。「正しい」（通信量が最も少ない）通信方法を図 4-6-20 (5) に示します。まず各プロセスは配列 A のうち他のプロセスに渡すデータをプロセス別に圧縮し（一般にメモリー上でとびとびになっているため）、それを MPI_ALLTOALLV などで他のプロセスに送信し、逆に他の各プロセスから送信されたデータを受信して自分の配列 A の元の位置に復元し、最後に配列 A を配列 X に加えます。しかしこのようにデータを圧縮、復元するのは若干面倒です。このような場合、重ね合わせを用いると簡単にデータの通信を行うことができます。

図 4-6-20 (1)

```
REAL A(9),X(6)
INTEGER IND(9)
  :
DO I=1,9
  X(IND(I)) = X(IND(I)) + A(I)
ENDDO
PRINT *,X
  :
```

図 4-6-20 (2)

図 4-6-20 (3)

```
INTEGER IRCNT(0:2)
  :
DO J=1,JSTA-1
  X(J) = 0.0
ENDDO
DO J=JEND+1,9                          ①
  X(J) = 0.0
ENDDO
DO I=ISTA,IEND
  X(IND(I)) = X(IND(I)) + A(I)         ②
ENDDO
DO IRANK=0,2
  IRCNT(IRANK) = 2                     ③
ENDDO
CALL MPI_REDUCE_SCATTER
&    (X,XX,IRCNT,MPI_REAL,MPI_SUM,     ④
&     MPI_COMM_WORLD,IERR)
PRINT *,MYRANK,XX
  :
```

図 4-6-20 (4)

重ね合せを使用したプログラムを図 4-6-20 (4) に、データの動きを図 4-6-20 (6) に示します。各プロセスは図 4-6-20 (4) の①で、配列 X のうち自分の担当以外の部分をゼロクリア（これが前提になります）し、②で配列 A の自分の担当部分を配列 X に加えます。

計算終了後、④で MPI_REDUCE_SCATTER (MPI_REDUCE を行った後、MPI_SCATTER を行う集団通信ルーチン：付録参照) を使用して通信を行ないます。まず MPI_REDUCE で配列 X の全要素が要素ごとに加算され、その結果図 4-6-20 (6) の点線部分のようになります。次に点線部分が MPI_SCATTER で各プロセスに 2 要素ずつ (③で指定) 分配されて各プロセスの配列 XX に入り、図 4-6-20 (5) と同じ結果を簡単に得ることができます。

なお、3-3-5 節で説明したように、MPI_REDUCE_SCATTER では、送信バッファー X と受信バッファー XX は別の配列にしなければならないので注意して下さい (ただし、MPI-2 で追加された MPI_IN_PLACE という変数を使用すると、この制限を回避することができます。詳細は 3-8-1 節を参照して下さい。)。

前述のように、重ね合せは「正しい」通信と比較して通信時間がかかるので、通信時間がクリティカルな部分では使用しないで下さい。

図 4-6-20 (5)

図 4-6-20 (6)

4-6-7　パイプライン法 (ICCG 法の前進消去／後退代入などで使用)

パイプライン法とは

本節で説明する方法は、実際のプログラムではそれほど現れることはないので、お急ぎの方は飛ばしてもかまいません。

図 4-6-22 (1) (2) では、計算領域の各要素に対し、上の要素と左の要素から自分を計算するという依存関係があり、簡単には並列化できません（なお、『境』は境界上のデータを表します）。例えば図 4-6-22 (2) を 2 次元方向にブロック分割したとします。このときランク 0 と 1 が行う■と●の計算がすべて終了した後でないと、ランク 2 は★を計算することができません。また★の計算には●が必要なので、ランク 2 のプロセスは★の計算の前にランク 1 のプロセスから●を送信してもらう必要があります。

このパターンはパイプライン法という方法で（何とか）並列化することができます。このパターンは実際には対称帯行列の反復解法である ICCG 法の前進消去、後退代入の計算などで発生します (5-4-1 節参照)。

一方、図 4-6-23 (1) の①のような『普通の』ループが続いていて、これを図 4-6-23 (2) のように 2 次元方向に分割して並列処理していたところへ、突然②のように横方向に依存関係があるループが現れたような場合にも、②のループに対してパイプライン法を適用することができます。

```
PROGRAM MAIN
PARAMETER(MX=7,MY=6)
DIMENSION X(0:MX,0:MY)
  :
DO J=1,MY
  DO I=1,MX
    X(I,J) = X(I,J) + X(I-1,J) + X(I,J-1)
  ENDDO
ENDDO
  :
```

図 4-6-22 (1)

図 4-6-22 (2)

```
PROGRAM MAIN
PARAMETER(MX=6,MY=6)
DIMENSION X(MX,0:MY)
  :
DO J=1,MY
  DO I=1,MX
    X(I,J) = ～                    ①
  ENDDO
ENDDO
  :
DO J=1,MY
  DO I=1,MX
    X(I,J) = X(I,J) + X(I,J-1)     ②
  ENDDO
ENDDO
  :
```

図 4-6-23 (1)

図 4-6-23 (2)

パイプライン法のプログラム例

図 4-6-22 (1) をパイプライン法で並列化した場合、配列 X を 1 次元方向と 2 次元方向のどちらでブロック分割することもできますが、ここでは図 4-6-24 に示すように 2 次元方向にブロック分割したとします。それとは別に 1 次元方向を適当な大きさ (IBLOCK) に分割します。分割された領域 (例えば図 4-6-24 の着色部分) をブロックと呼び、これが一度に処理を行う単位となります。

3 つのプロセスで並列に実行した場合の動作概要を説明します。図 4-6-24 の () 内の数字の順に斜線が移動していき、斜線上の各ブロックが並列に処理されるというイメージになります。つまり各プロセスは隣のプロセスと 1 つだけずれたブロックを並列に処理します。

- まず (1) に示すように、ランク 0 のプロセスのみが動作し、1のブロックを計算します。
- 次に (2) に示すように、ランク 0, 1 の各プロセスがそれぞれ3のブロックを計算します。
- 次に (3) に示すように、ランク 0, 1, 2 の各プロセスがそれぞれ5のブロックを計算します。この時点で初めて全プロセスが稼働します。以下同様に処理が行なわれます。

この処理に伴い境界の要素の通信が必要になります。例えば図 4-6-24 でランク 1 のプロセスは、着色したブロックを計算するためにランク 0 のプロセスが所有する〓の要素が必要になります。従ってランク 1 のプロセスは、ランク 0 のプロセスが1のブロックを計算した後、〓の要素を送ってもらい、その後で3の計算を行う必要があります。

図 4-6-22 (1) を並列化したプログラムを図 4-6-25 に、プロセスごとのデータの動きを図 4-6-26 に示します。

- 図 4-6-25 の [1] で 2 次元方向にブロック分割し、各プロセスの担当範囲 JSTA, JEND を決定します。
- [3] でブロックの 1 次元方向の大きさを IBLOCK に設定します。後述するように IBLOCK の大きさはパフォーマンスに影響し、最適な大きさは試行錯誤で決定します。
- [4] のループ指標 II はこれから処理するブロックの 1 次元目の先頭アドレスを表し、図 4-6-26 の → に示すように順に移動します。II が 1 つ決まると、そのブロックの 1 次元方向の大きさ (図 4-6-26 の ↕) を [5] で IBLKLEN に設定します。IBLKLEN は基本的には IBLOCK と同じ大きさですが、図 4-6-26 のように MX が IBLOCK で割りきれない場合、最後の半端な部分では IBLOCK より小さくなるのでそれを調整します。
- 次に実際の計算部分について説明します。図 4-6-26 でランク 1 が行う着色したブロックの処理に着目すると、1 つのブロックの処理は以下の 3 つの部分から構成されていることがわかります。この処理を図 4-6-25 の [6]〜[8] で行います。
  (1) 左のプロセス (ランク 0) から送信された要素を、ブロックの左の境界を越えた 1 列 (〓で囲んだ②) に受信します (図 4-6-25 の [6])。
  (2) ブロック内の3を計算します (図 4-6-25 の [7])。
  (3) ブロックの右の境界の 1 列 (〓で囲んだ3) を、右のプロセス (ランク 2) に送信します (図 4-6-25 の [8])。

このように両端以外のプロセスでは、1 つのブロックを受信、計算、送信の順に処理しますが、両端のプロセスでは送信もしくは受信のどちらか一方を行います (ランク 0 のプロセスでは送信のみ、ランク 2 のプロセスでは受信のみ)。これを図 4-6-25 の [2] で調整します。

図 4-6-24

```
PROGRAM MAIN
INCLUDE 'mpif.h'
PARAMETER(MX=7,MY=6)
DIMENSION X(0:MX,0:MY)
INTEGER ISTATUS(MPI_STATUS_SIZE)
CALL MPI_INIT(IERR)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)
CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)
CALL PARA_RANGE(1,MY,NPROCS,MYRANK,JSTA,JEND)      [1]
IUP = MYRANK+1
IF (IUP==NPROCS) IUP=MPI_PROC_NULL
IDOWN = MYRANK-1                                   [2]
IF (IDOWN==-1) IDOWN=MPI_PROC_NULL
IBLOCK=2                                           [3]
  :
DO II=1,MX,IBLOCK                                  [4]
  IBLKLEN = MIN(IBLOCK,MX-II+1)                    [5]
  CALL MPI_IRECV(X(II,JSTA-1),IBLKLEN,MPI_REAL,
&          IDOWN,1,MPI_COMM_WORLD,IREQR,IERR)      [6]
  CALL MPI_WAIT(IREQR,ISTATUS,IERR)
  DO J=JSTA,JEND
    DO I=II,II+IBLKLEN-1
      X(I,J) = X(I,J) + X(I-1,J) + X(I,J-1)        [7]
    ENDDO
  ENDDO
  CALL MPI_ISEND(X(II,JEND),IBLKLEN,MPI_REAL,
&          IUP,1,MPI_COMM_WORLD,IREQS,IERR)        [8]
  CALL MPI_WAIT(IREQS,ISTATUS,IERR)
ENDDO
CALL MPI_FINALIZE(IERR)
  :
```

図 4-6-25

図 4-6-26

パフォーマンス上の考慮点

パイプライン法のパフォーマンス上の考慮点について説明します。

- パイプライン法では 1 次元方向と 2 次元方向のどちらでブロック分割することもできますが、境界のデータの通信が必要となるので、計算領域が長方形の場合には以下のように通信量が少なくなる（境界が短くなる）方向 (図 4-6-27 (2), 図 4-6-27 (4)) で分割を行って下さい。

図 4-6-27 (1)　×

図 4-6-27 (2)　○

図 4-6-27 (3)　×

図 4-6-27 (4)　○

- ブロックの大きさ (IBLOCK) を変えると、ある値でパフォーマンスがもっともよくなり、それより大きくしても小さくしてもパフォーマンスは低下します。この理由を以下に説明します。
  図 4-6-28 (1) (2) (3) ではブロックの数がそれぞれ 16，8，4 個になっており、この 3 ケースをまず並列度について比較すると、全プロセスが稼働するのは着色した部分になるので、図 4-6-28 (1) は並列度が高くなりますが図 4-6-28 (3) では低くなります。
  次に通信について比較します。各プロセスが行う通信量は境界の長さになるので図 4-6-28 (1) (2) (3) は同じです。次に通信回数ですが、1 つのブロックを処理するごとに通信が必要になるので、ブロック数の多い図 4-6-28 (1) では通信回数が多くなり、図 4-6-28 (3) では少なくなります。4-1 節で説明したように、通信を 1 回行うごとに立上り時間がかかるので、通信回数の点ではブロック数の少ない図 4-6-28 (3) が最もよくなります。
  以上のように、ブロック数は多すぎても少なすぎてもパフォーマンスが低下しますが、最適な IBLOCK の大きさは計算量とも関係するので試行錯誤で決定して下さい。

図 4-6-28 (1)

図 4-6-28 (2)

図 4-6-28 (3)

4-6-8　ツイスト分割法 (ADI 法などで使用)

本節で説明する方法は、実際のプログラムではそれほど現れることはないので、お急ぎの方は飛ばしてもかまいません。

本節では、参考文献 [31] に紹介されているツイスト分割法について説明します。ツイスト分割法は、差分法などで使用される ADI 法を並列化する場合などに適用することができます。

図 4-6-30 の①〜④の DO ループでは、それぞれ図 4-6-31 (1)〜(4) の計算を行います（図中の「境」は境界の要素を示します)。計算に使用する各要素には、矢印に示す依存関係があります。このように 1 次元方向 (↓ ↑) と 2 次元方向 (→ ←) の両方向に依存関係のあるパターンを並列化する場合、例えば図 4-6-31 の点線に示すように、配列 X を 2 次元方向にブロック分割して①，②の DO ループを普通に並列化し、③，④の DO ループは 4-6-7 節で説明したパイプライン法で並列化するという方法もありますが、パイプライン法では並列度や通信回数などの問題があります。

このパターンはツイスト分割法で並列化することもできます。ツイスト分割法では、配列 X の 1 次元目と 2 次元目をともに「プロセス数」でブロック分割します。例えば図 4-6-32 (1) に示す 10 × 10 の要素を 5 プロセスでツイスト分割すると、図 4-6-32 (2) のようになります。図中の着色部分を本書ではブロックと呼びます。図から分かるように、ツイスト分割ではどの行、どの列のブロックも 0〜4 の全プロセスが担当しており、これによって、どの方向の計算も各プロセスのロードバランスが均等になります。

```
PROGRAM MAIN
PARAMETER(IMAX=6,JMAX=6)
REAL X(0:IMAX+1,0:JMAX+1)
  :
DO J=1,JMAX      ① (↓)
  DO I=1,IMAX
    X(I,J) = X(I,J)+X(I-1,J)
  ENDDO
ENDDO
DO J=1,JMAX      ② (↑)
  DO I=IMAX,1,-1
    X(I,J) = X(I,J)+X(I+1,J)
  ENDDO
ENDDO
DO J=1,JMAX      ③ (→)
  DO I=1,IMAX
    X(I,J) = X(I,J-1)+X(I,J)
  ENDDO
ENDDO
DO J=JMAX,1,-1 ④ (←)
  DO I=1,IMAX
    X(I,J) = X(I,J+1)+X(I,J)
  ENDDO
ENDDO
  :
```

図 4-6-30

図 4-6-31 (1)　①のループ

図 4-6-31 (3)　③のループ

図 4-6-31 (2)　②のループ

図 4-6-31 (4)　④のループ

図 4-6-32 (1)

図 4-6-32 (2)

図 4-6-33 (1) の計算領域を 3 プロセスでツイスト分割して並列化したプログラムを図 4-6-34 に、動作を図 4-6-35 (1)〜(4) に示します。図 4-6-33 (1) の〇で囲んだ部分は各ブロックの座標を示します。

- 図 4-6-34 の (1) で、自分のランク値より 1 つ上と下のランク値を IUP と IDOWN にセットします。この値は 0 → 1 → 2 → 0 のように循環します。
- (2) で、図 4-6-33 (1) (2) に示すように、II 座標が 0，1，2 のときに自分のプロセスが担当するブロックの JJ 座標を配列 JJJ にセットします。同様に、JJ 座標が 0，1，2 のときに自分のプロセスが担当するブロックの II 座標を配列 III にセットします。配列 III と JJJ の値は各プロセスで異なることに注意して下さい。
- (3) で、図 4-6-33 (1) に示すように、各ブロックの I 方向と J 方向をブロック分割し、各ブロックの最初と最後の位置を配列 ISTA、IEND，JSTA，JEND にセットします。ブロック分割には 4-5-4 節で紹介したサブルーチン PARA_RANGE を使用しました。
- (4) と (5) で、データを通信するときに使う派生データ型を、3-5-3-2 節で説明した自作のサブルーチン PARA_TYPE_BLOCK2 を使用して、ブロックごとに作成します (MPI-2 が使用できるマシン環境の場合は、MPI-2 で提供されている MPI_TYPE_CREATE_SUBARRAY (3-5-3-1 節参照) を使用して下さい)。なお、派生データ型は全て配列 X の先頭を起点とします。

  例えばブロック (1,1) の場合を図 4-6-33 (1) の〓と〓に示します。(4) で、各ブロックの I 方向の最初の 1 行、最後の 1 行、およびそれぞれの境界の外の 1 行の派生データ型を作成し、配列 ITSTA，ITEND，ITSTAM，ITENDP に設定します。同様に (5) で、J 方向の最初の 1 列、最後の 1 列、およびそれぞれの境界の外の 1 列の派生データ型を作成し、JTSTA，JTEND，JTSTAM，JTENDP に設定します。

  なお、例えば図 4-6-33 (1) の ITSTAM(1,1) と、1 つ上のブロックの ITEND(0,1) は実際には同一ですが、配列 X を縮小 (4-5-7 節参照) する場合は配列名が別になっていた方が便利なので（詳細は省略します)、あえて別の配列にしました。
- (6)，(12)，(13)，(14) の DO ループで、図 4-6-35 (1)〜(4) の計算を並列に行います。どの計算も同様なので、ここでは (6) の DO ループについて図 4-6-35 (1) で説明します。
- (6) の DO ループで、ブロックの I 方向の座標 II が 0，1，2 と変化すと、図 4-6-35 (1) の一番上、上から 2 番目、3 番目のブロックのうち、自分が担当するブロックが順に処理されます。まず II = 0 となり、各プロセスは (7) で、II = 0 のときに自分が担当するブロックの J 方向の座標を JJ にセットします。例えばランク 0 の場合は JJ = 0 となり、図 4-6-35 (1) の着色したブロックが選択されます。
- (8) で、各プロセスは自分が担当するブロック内の各要素を計算します。
- 計算終了後、(10) で、計算したブロックの一番下の 1 行 (例えばランク 0 の場合は図 4-6-35 (1) の A) を、次に担当するプロセス (本例ではランク 2) に送信します。この送信には、各ブロックの一番下の 1 行を表す派生データ型 ITEND を使用します。
- 逆に他のプロセス (本例ではランク 1) から送られたデータを (11) で受信します (例えばランク 0 の場合は図 4-6-35 (1) の B)。この受信には、各ブロックの一番上の境界の外の 1 行を表す派生データ型 ITSTAM を使用します。
- (6) のループが II = 2, 3 と進み、上から 2 つ目、3 つ目のブロックを同様に計算します。なお、3 つ目のブロックの終了後には (10) と (11) の通信は行わないので、(9) の IF 文が入っています。

図 4-6-31 のパターンの場合、パイプライン法 (4-6-7 節) とツイスト分割法は下記のように長所・短所があり、どちらがよいかは一概には言えません。

- パイプライン法はロードバランス（並列度）の点で問題がありましたが、ツイスト分割法では 1、2 次元目ともに常に同じ数のプロセスが同じ量の計算を行うので、各プロセスのロードバランスは均等になります。
- パイプライン法では（並列度を高めるために）ブロック数を多くすると通信回数が多くなってしまいましたが、ツイスト分割法ではブロック数（通信回数）は 1、2 次元目ともにプロセス数と同じで、（プロセス数が少ない場合は）それほど多くはありません。
- 右図で例えば 3 プロセスの場合のランク 1 の通信量（太線の長さ）を比較すると、パイプライン法では $1 + 1 = 2$ に対し、ツイスト分割法では $(1/3) \times 4 \times 3 = 4$ となり、ツイスト分割法の方が通信量は 2 倍多くなります。
- 1 つの配列をツイスト分割すると、その配列を使用する全てのループで使われる全ての配列を、動作の複雑なツイスト分割で並列化しなければならず、パイプライン法に比べて修正が面倒になります。

図 4-6-33 (1)

図 4-6-33 (2)

```
      PROGRAM MAIN
      INCLUDE 'mpif.h'
      PARAMETER(IMAX=53,JMAX=47,NCPU=10)
      INTEGER ISTATUS(MPI_STATUS_SIZE)
      REAL X(0:IMAX+1,0:JMAX+1)
      INTEGER III(0:NCPU-1),JJJ(0:NCPU-1)
      INTEGER ISTA(0:NCPU-1),IEND(0:NCPU-1)
      INTEGER JSTA(0:NCPU-1),JEND(0:NCPU-1)
      INTEGER ITSTA (0:NCPU-1,0:NCPU-1)
     &       ,ITEND (0:NCPU-1,0:NCPU-1)
      INTEGER JTSTA (0:NCPU-1,0:NCPU-1)
     &       ,JTEND (0:NCPU-1,0:NCPU-1)
      INTEGER ITSTAM(0:NCPU-1,0:NCPU-1)
     &       ,ITENDP(0:NCPU-1,0:NCPU-1)
      INTEGER JTSTAM(0:NCPU-1,0:NCPU-1)
     &       ,JTENDP(0:NCPU-1,0:NCPU-1)
      REAL XX(0:IMAX+1,0:JMAX+1)
      CALL MPI_INIT(IERR)
      CALL MPI_COMM_SIZE
     &    (MPI_COMM_WORLD,NPROCS,IERR)
      CALL MPI_COMM_RANK
     &    (MPI_COMM_WORLD,MYRANK,IERR)
      IUP   = MOD(MYRANK+1        ,NPROCS)        (1)
      IDOWN = MOD(MYRANK-1+NPROCS,NPROCS)         (1)
      DO II=0,NPROCS-1
        JJJ(II) = MOD(II+MYRANK        ,NPROCS)
      ENDDO                                       (2)
      DO JJ=0,NPROCS-1
        III(JJ) = MOD(JJ-MYRANK+NPROCS,NPROCS)
      ENDDO
```

```
      DO II=0,NPROCS-1
        CALL PARA_RANGE
     &      (1,IMAX,NPROCS,II,ISTA(II),IEND(II))
        CALL PARA_RANGE                           (3)
     &      (1,JMAX,NPROCS,II,JSTA(II),JEND(II))
      ENDDO
      DO JJ=0,NPROCS-1
        JS = JSTA(JJ)
        JE = JEND(JJ)
        DO II=0,NPROCS-1
          IS = ISTA(II)
          IE = IEND(II)
          CALL PARA_TYPE_BLOCK2
     &          (0,IMAX+1,0,IS,IS,JS,JE,
     &           MPI_REAL,ITSTA(II,JJ))
          CALL PARA_TYPE_BLOCK2
     &          (0,IMAX+1,0,IE,IE,JS,JE,
     &           MPI_REAL,ITEND(II,JJ))          (4)
          CALL PARA_TYPE_BLOCK2
     &          (0,IMAX+1,0,IS-1,IS-1,JS,JE,
     &           MPI_REAL,ITSTAM(II,JJ))
          CALL PARA_TYPE_BLOCK2
     &          (0,IMAX+1,0,IE+1,IE+1,JS,JE,
     &           MPI_REAL,ITENDP(II,JJ))
```

（次ページへ）

```
      CALL PARA_TYPE_BLOCK2
&             (0,IMAX+1,0,IS,IE,JS,JS,
&              MPI_REAL,JTSTA(II,JJ))
      CALL PARA_TYPE_BLOCK2
&             (0,IMAX+1,0,IS,IE,JE,JE,
&              MPI_REAL,JTEND(II,JJ))          (5)
      CALL PARA_TYPE_BLOCK2
&             (0,IMAX+1,0,IS,IE,JS-1,JS-1,
&              MPI_REAL,JTSTAM(II,JJ))
      CALL PARA_TYPE_BLOCK2
&             (0,IMAX+1,0,IS,IE,JE+1,JE+1,
&              MPI_REAL,JTENDP(II,JJ))
    ENDDO
  ENDDO
    :
  DO II=0,NPROCS-1                            (↓) (6)
    JJ = JJJ(II)                                  (7)
    DO J=JSTA(JJ),JEND(JJ)
      DO I=ISTA(II),IEND(II)
        X(I,J) = X(I,J) + X(I-1,J)                (8)
      ENDDO
    ENDDO
    IF (II/=NPROCS-1) THEN                        (9)
      CALL MPI_ISEND
&         (X,1,ITEND(II,JJ)          ,IDOWN,1,
&          MPI_COMM_WORLD,IREQS,IERR)             (10)
      CALL MPI_IRECV
&         (X,1,ITSTAM(II+1,JJJ(II+1)),IUP  ,1,
&          MPI_COMM_WORLD,IREQR,IERR)             (11)
      CALL MPI_WAIT(IREQS,ISTATUS,IERR)
      CALL MPI_WAIT(IREQR,ISTATUS,IERR)
    ENDIF
  ENDDO
  DO II=NPROCS-1,0,-1                         (↑) (12)
    JJ = JJJ(II)
    DO J=JSTA(JJ),JEND(JJ)
      DO I=IEND(II),ISTA(II),-1
        X(I,J) = X(I,J) + X(I+1,J)
      ENDDO
    ENDDO
    IF (II/=0) THEN
      CALL MPI_ISEND
&         (X,1,ITSTA(II,JJ)          ,IUP  ,1,
&          MPI_COMM_WORLD,IREQS,IERR)
      CALL MPI_IRECV
&         (X,1,ITENDP(II-1,JJJ(II-1)),IDOWN,1,
&          MPI_COMM_WORLD,IREQR,IERR)
      CALL MPI_WAIT(IREQS,ISTATUS,IERR)
      CALL MPI_WAIT(IREQR,ISTATUS,IERR)
    ENDIF
  ENDDO
```

```
  DO JJ=0,NPROCS-1                            (→) (13)
    II = III(JJ)
    DO J=JSTA(JJ),JEND(JJ)
      DO I=ISTA(II),IEND(II)
        X(I,J) = X(I,J-1) + X(I,J)
      ENDDO
    ENDDO
    IF (JJ/=NPROCS-1) THEN
      CALL MPI_ISEND
&         (X,1,JTEND(II,JJ)          ,IUP  ,1,
&          MPI_COMM_WORLD,IREQS,IERR)
      CALL MPI_IRECV
&         (X,1,JTSTAM(III(JJ+1),JJ+1),IDOWN,1,
&          MPI_COMM_WORLD,IREQR,IERR)
      CALL MPI_WAIT(IREQS,ISTATUS,IERR)
      CALL MPI_WAIT(IREQR,ISTATUS,IERR)
    ENDIF
  ENDDO
  DO JJ=NPROCS-1,0,-1                         (←) (14)
    II = III(JJ)
    DO J=JEND(JJ),JSTA(JJ),-1
      DO I=ISTA(II),IEND(II)
        X(I,J) = X(I,J+1) + X(I,J)
      ENDDO
    ENDDO
    IF (JJ/=0) THEN
      CALL MPI_ISEND
&         (X,1,JTSTA(II,JJ)          ,IDOWN,1,
&          MPI_COMM_WORLD,IREQS,IERR)
      CALL MPI_IRECV
&         (X,1,JTENDP(III(JJ-1),JJ-1),IUP  ,1,
&          MPI_COMM_WORLD,IREQR,IERR)
      CALL MPI_WAIT(IREQS,ISTATUS,IERR)
      CALL MPI_WAIT(IREQR,ISTATUS,IERR)
    ENDIF
  ENDDO
    :
```

図 4-6-34

図 4-6-35 (1)　(6) のループ

図 4-6-35 (2)　(12) のループ

図 4-6-35 (3)　(13) のループ

図 4-6-35 (4)　(14) のループ

4-6-9　一次逐次演算（漸化式などで使用）

本節で説明する方法は、実際のプログラムではそれほど現れることはないので、お急ぎの方は飛ばしてもかまいません。

図 4-6-36 (1) では、ループの左辺で定義された X(I) が、次のループ反復の右辺で X(I-1) として参照されています。このように、あるループ反復で定義された値がそれ以後のループ反復で参照されることを、「ループの各反復に回帰参照（または依存関係）がある」と言い、通常は並列化できません (4-2 節参照)。

図 4-6-36 (1) のように「X(I) = X(I-1) + ･･･」の形をした、漸化式のような演算を一次逐次演算と呼びます (参考文献 [18] 参照)。図 4-6-36 (1) には回帰参照が存在しますが、本節に示すように並列化が可能です。図 4-6-36 (2) (3) も並列化が可能ですが、下記にポイントのみを簡単に記述します (詳細は参考文献 [4] 参照)。

```
PROGRAM MAIN
PARAMETER(N=12)
REAL X(0:N),Y(N)
  :
DO I=1,N
  X(I) = X(I-1) + Y(I)
ENDDO
  :
```

図 4-6-36 (1)

X(I) = A(I)*X(I-1) + Y(I)

$$\begin{cases} x_1 = a_1 x_0 + y_1 \\ 1 = 1 \end{cases} \rightarrow \begin{pmatrix} x_1 \\ 1 \end{pmatrix} = \begin{pmatrix} a_1 & y_1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ 1 \end{pmatrix}$$

$$\begin{cases} x_2 = a_2 x_1 + y_2 \\ 1 = 1 \end{cases} \rightarrow \begin{pmatrix} x_2 \\ 1 \end{pmatrix} = \begin{pmatrix} a_2 & y_2 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ 1 \end{pmatrix}$$

$$= \begin{pmatrix} a_2 & y_2 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a_1 & y_1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ 1 \end{pmatrix}$$

図 4-6-36 (2)

X(I) = A(I)*X(I-1) + B(I)*X(I-2)

$$\begin{cases} x_2 = a_2 x_1 + b_2 x_0 \\ x_1 = x_1 \end{cases} \rightarrow \begin{pmatrix} x_2 \\ x_1 \end{pmatrix} = \begin{pmatrix} a_2 & b_2 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_0 \end{pmatrix}$$

$$\begin{cases} x_3 = a_3 x_2 + b_3 x_1 \\ x_2 = x_2 \end{cases} \rightarrow \begin{pmatrix} x_3 \\ x_2 \end{pmatrix} = \begin{pmatrix} a_3 & b_3 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} x_2 \\ x_1 \end{pmatrix}$$

$$= \begin{pmatrix} a_3 & b_3 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} a_2 & b_2 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_0 \end{pmatrix}$$

図 4-6-36 (3)

まず、図 4-6-36 (1) のループを実行する直前の配列の状態を図 4-6-37 (1) に、ループを実行した直後の状態を図 4-6-37 (2) に示します。図 4-6-37 (1) の配列 X(0) と配列 Y(1)～Y(12) には適当な値が入っているとします。図 4-6-37 (1) (2) から分かるように、X(0) から X(1) を求め、X(1) から X(2) を求めるという因果関係が連綿と続くため、通常の方法では並列化できないのがお分かりになると思います。

図 4-6-37 (1)

→

図 4-6-37 (2)

並列化の方法 (以下のアルゴリズムは参考文献 [4] [18] を参考にしました。)

まず並列化したときの動作の概要を説明します。計算する配列が X(1)〜X(12)、プロセス数が 4 とすると、配列 X は図 4-6-38 (1) のようにブロック分割されます。計算は大きく 2 つのステップに分かれ、ステップ 1 で図 4-6-38 (1) の★（各プロセスが担当する最初の要素）を求め、ステップ 2 で★から残りの↓を順に求めます。★を求める計算と↓を求める計算は同じなので、各プロセスは同じ計算を 2 回行うことになります。

図 4-6-36 (1) を並列化したプログラムを図 4-6-38 (2) に、データの動きを図 4-6-38 (3) に示します。

- 図 4-6-38 (2) (3) の①で N (= 12) 個の要素をブロック分割し、各プロセスが担当する配列 X の下限 ISTA と上限 IEND を、4-5-4 節で紹介したサブルーチン PARA_RANGE を使用して求めます。
- ②で変数 SUM をゼロクリアします。ただしランク 0 のプロセスだけは X(0) を代入します。
- ③で、配列 Y のうち自分の担当範囲の値を変数 SUM に合計します。
- ④で、MPI-2 の集団通信サブルーチン MPI_EXSCAN（付録参照）を使用して変数 SUM の縮約演算を行い、結果を変数 SSUM に入れます。MPI_EXSCAN を実行すると、図 4-6-38 (3) の〓で囲んだ部分の通信が行われ、自分よりランクが小さい全てのプロセスの SUM の合計が自分の SSUM に入ります。ただしランク 4 の SUM は使用されず、ランク 0 の SSUM に値は入りません。
  なお、MPI-1 の集団通信サブルーチン MPI_SCAN を使用しても、ロジックは若干変わりますが、同様の並列化を行うことは可能です。
- ⑤で、SSUM に受信データが入っていないランク 0 のプロセスのみ、X(0) を SSUM に代入します。
- ⑥で、ランク 0〜3 の各プロセスが担当する最初の要素 X(1), X(4), X(7), X(10) を求めます (これは図 4-6-38 (1) の★に相当します)。
- X(1), X(4), X(7), X(10) の値を元に、⑦で残りの要素の計算を行います (図 4-6-38 (1) の↓に相当します)。

並列化した場合の計算量

この方法では同じ計算（加算）を 2 度行うため、無駄に感じられるかもしれません。しかし、例えば要素数が m とすると、単体で実行した場合の加算回数は m 回で、並列化した場合、各プロセスの加算回数は $(2 \times m$ 回$)/($プロセス数$)$ となるため（ただし通信時間は除く）、プロセス数が多いときは一応並列化の効果が出ます。

図 4-6-38 (1)

```
PROGRAM MAIN
INCLUDE 'mpif.h'
PARAMETER(N=12)
REAL X(0:N),Y(N),SUM,SSUM
CALL MPI_INIT(IERR)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)
CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)
CALL PARA_RANGE(1,N,NPROCS,MYRANK,ISTA,IEND) ①
   :
SUM = 0.0                                    ②
IF (MYRANK==0) SUM = X(0)                    ②
DO I=ISTA,IEND
  SUM = SUM + Y(I)                           ③
ENDDO
CALL MPI_EXSCAN(SUM,SSUM,1,MPI_REAL,         ④
&         MPI_SUM,MPI_COMM_WORLD,IERR)
IF (MYRANK==0) SSUM = X(0)                   ⑤
X(ISTA) = Y(ISTA) + SSUM                     ⑥
DO I=ISTA+1,IEND
  X(I) = X(I-1) + Y(I)                       ⑦
ENDDO
   :
```

図 4-6-38 (2)

【ランク0】
X(0)
99
+ ②
⑤ SSUM
99
① ISTA
Y(1)
1
+ ③
⑥ +
99+1
X(1)
Y(2)
2
+ ③
⑦ +
99+1+2
X(2)
① IEND
Y(3)
3
+ ③
⑦ +
99+1+2+3
X(3)
SUM
99+1+2+3
④
【ランク1】
SSUM
99+1+2+3
① ISTA
Y(4)
4
+ ③
⑥ +
99+1+2+3+4
X(4)
Y(5)
5
+ ③
⑦ +
99+1+2+3+4+5
X(5)
① IEND
Y(6)
6
+ ③
⑦ +
99+1+2+3+4+5+6
X(6)
SUM
4+5+6
【ランク2】
SSUM
99+1+2+3+4+5+6
① ISTA
Y(7)
7
+ ③
⑥ +
99+1+2+3+4+5+6+7
X(7)
Y(8)
8
+ ③
⑦ +
99+1+2+3+4+5+6+7+8
X(8)
① IEND
Y(9)
9
+ ③
⑦ +
99+1+2+3+4+5+6+7+8+9
X(9)
SUM
7+8+9
【ランク3】
SSUM
99+1+2+3+4+5+6+7+8+9
① ISTA
Y(10)
10
+ ③
⑥ +
99+1+2+3+4+5+6+7+8+9+10
X(10)
Y(11)
11
+ ③
⑦ +
99+1+2+3+4+5+6+7+8+9+10+11
X(11)
① IEND
Y(12)
12
+ ③
⑦ +
99+1+2+3+4+5+6+7+8+9+10+11+12
X(12)
SUM
10+11+12
(未使用)

図 4-6-38 (3)

4-6-10　受信するデータ量が不明な場合の通信

MPI_ALLTOALLV の場合

図 4-6-39 (1) では、ランク 0〜2 の各プロセスが、お互いに他のプロセスに個数の異なるデータを送信します。このとき受信側のプロセスは受信する（相手から送られる）個数を知らないとします。

この通信は集団通信ルーチン MPI_ALLTOALLV で行なうことができます。集団通信ルーチンでは、3-3-4 節に示したように、送信バッファーで指定するバイト数 (個数 × データ型のバイト数) と受信バッファーで指定するバイト数は一致している必要があるため、受信する個数を知る必要があります。本例のように受信する個数を知らない場合の対処方法を図 4-6-40 のプログラムで説明します。

- 図 4-6-40 の [3] で、図 4-6-39 (2) の上段に示すように、ランク i のプロセスに送るデータの個数を配列 ISCNT(i) に入れます。何も送らないプロセスには 0 を入れる必要があるため、[3] の前に [2] でゼロクリアします。
- [4] で、図 4-6-39 (3) の上段に示すように、ランク i のプロセスに送るデータを送信バッファー ISBUF(*,i) に入れます。
- [5] で MPI_ALLTOALL の通信を行うと、図 4-6-39 (2) の下段に示すように、ランク i のプロセスから送られるデータの個数が受信バッファー IRCNT(i) に入ります。
- [6] で、図 4-6-39 (3) の上段に示す ISBUF の先頭から、ランク i のプロセス宛てのデータが入る最初の位置までの変位を ISDISP(i) に要素数で指定します。本例では、[1] に示すように配列 ISBUF の 1 次元目の大きさが 10 なので、[6] の下線部で 10 を指定しました。受信バッファー IRBUF についても同様に [6] で IRDISP に指定します。
- [7] で MPI_ALLTOALLV を使用して通信を行います。このとき、[5] で得られた IRCNT を使用して受信するデータの個数を指定します。その結果、図 4-6-39 (3) に示す通信が行われます。なお、点線の部分（自分宛ての通信）は、ISCNT と IRCNT 内のデータ数がゼロなので通信は行なわれません。

図 4-6-39 (1)

図 4-6-39 (2)

図 4-6-39 (3)

```
   :
 PARAMETER(NCPU=3)
 INTEGER ISCNT (0:NCPU-1),IRCNT (0:NCPU-1)
 INTEGER ISDISP(0:NCPU-1),IRDISP(0:NCPU-1)
 REAL SBUF(10,0:NCPU-1),RBUF(10,0:NCPU-1)          [1]
   :
 DO IRANK=0,NPROCS-1
   ISCNT(IRANK) = 0                                [2]
 ENDDO
 ISCNTに送信するデータ数を入れる。                  [3]
 SBUFに送信するデータを入れる。                     [4]
 CALL MPI_ALLTOALL(ISCNT,1,MPI_INTEGER,
&                  IRCNT,1,MPI_INTEGER,            [5]
&                  MPI_COMM_WORLD,IERR)
 DO IRANK=0,NPROCS-1
   ISDISP(IRANK) = IRANK*10                        [6]
   IRDISP(IRANK) = IRANK*10
 ENDDO
 CALL MPI_ALLTOALLV(SBUF,ISCNT,ISDISP,MPI_REAL,
&                   RBUF,IRCNT,IRDISP,MPI_REAL,    [7]
&                   MPI_COMM_WORLD,IERR)
   :
```

図 4-6-40

1 対 1 通信の場合

1 対 1 通信ルーチンでは、受信バッファーで指定するバイト数 (データ個数 × データ型のバイト数) は、送信バッファーで指定するバイト数と等しいか、または大きくても構いません (3-4-5 節)。従って、例えばランク 1 がランク 0 からデータを受信する場合、受信するデータ個数より受信バッファーの方が大きいのが確実であれば、ランク 0 から送られるデータの個数が分からなくても、1 対 1 通信ルーチンを使用して受信することができます。受信後に、受信したデータの個数を知りたい場合、MPI_GET_COUNT を使用します。

MPI_GET_COUNT の使い方を図 4-6-41 のプログラムで説明します。

- 図 4-6-40 に示すように、ランク 0 の大きさ 5 の配列 SBUF のうち最初の 3 要素が送信バッファー、ランク 1 の大きさ 10 の配列 RBUF が受信バッファーだとします。
- 図 4-6-41 の [2] で、ランク 0 は 3 個のデータをランク 1 に送信します。
- [3] でランク 1 は、ランク 0 からのデータを受信します。このとき受信するデータの個数が分からないので、受信バッファー RBUF の大きさである 10 を指定します。通信の結果、図 4-6-40 に示すように、受信バッファー RBUF の最初の 3 要素に受信データが入ります。
- ランク 1 は受信後に、[4] でサブルーチン MPI_GET_COUNT を実行します。MPI_GET_COUNT の引数には [1] と [3] の二重線に示す配列 ISTATUS を指定します（引数の詳細は付録参照)。実行の結果、引数 ICOUNT に、[3] で受信したデータの個数 (本例では 3) が戻り、受信した個数を知ることができます。

図 4-6-40

```
   :
  include 'mpif.h'
  INTEGER ISTATUS(MPI_STATUS_SIZE)                    [1]
  REAL SBUF(5)
  REAL RBUF(10)
   :
  IF (MYRANK==0) THEN
    CALL MPI_SEND(SBUF,3,MPI_REAL,1,1,                [2]
&                 MPI_COMM_WORLD,IERR)
  ELSEIF (MYRANK==1) THEN
    CALL MPI_RECV(RBUF,10,MPI_REAL,0,1,               [3]
&                 MPI_COMM_WORLD,ISTATUS,IERR)
    CALL MPI_GET_COUNT(ISTATUS,MPI_REAL,ICOUNT,IERR) [4]
  ENDIF
   :
```

図 4-6-41

4-7　特殊な並列化方法

4-7-1　MPMD モデルでの並列化

本節では、複数の異なるプログラムを 1 つの並列ジョブとして実行する MPMD (Multiple Program Multiple Data) モデルについて説明します (1-2 節参照)。

4-7-1-1　実行方法

まず MPMD モデルのプログラムを実行する方法について説明します。

- 【方法 1】マシン環境によっては、各プロセスごとに異なるロードモジュール名を指定できる機能が提供されている場合があります。これは MPI の機能ではなく、そのマシンの並列処理環境が提供する機能で、提供されているかどうかはマシン環境に依存します。また具体的な指定方法もマシン環境によって異なりますので、マニュアルなどで確認して下さい。
  一例を図 4-7-2 (1) に示します。構造解析と流体解析のロードモジュールを別の名前 (`a.out` と `b.out`) で作成し、対応表（ファイル）に「ランク 0 のプロセスは `a.out`、ランク 1 のプロセスは `b.out` を実行する。」と設定します。そして環境変数で対応表のファイル名を指定し、並列ジョブを実行します。
- 【方法 2】1 つのロードモジュール名しか指定できないマシン環境では、通常は MPMD モデルのプログラムを実行することができません。しかし、指定した 1 つのロードモジュールを各ノードの（異なる）ローカルディスクからロードできるマシン環境であれば、MPMD モデルのプログラムを実行することができます。
  図 4-7-2 (2) に示すように、構造解析と流体解析プログラムのロードモジュール名を例えば `a.out` のように同一名とし（ただし中身は異なります）、各ノードのローカルディスクに置いて並列に実行します。すると各ノードで異なる中身の `a.out` がロードされるので、MPMD モデルのプログラムを実行することができます。
- 【方法 3】図 4-7-2 (3) に示すように、構造解析と流体解析のプログラムのメインルーチンをサブルーチン化し、その上に形式的なメインルーチンを置きます（最近流行？の××ホールディングス（持株会社）のようになります）。そしてメインルーチンの最初にランクの値を調べ、いずれかのサブルーチンをコールします。
  この方法は、2 つのプログラムで同じ名前の `COMMON` を使っていると、うまくいきません。
- 【方法 4】MPI-1 の拡張機能である MPI-2 (3-1 節参照) には、「プロセスの生成と制御」という機能があり、並列ジョブの実行中に別プロセスを作成することができますが、本書では説明を省略します。

図 4-7-2 (1)

図 4-7-2 (2)

図 4-7-2 (3)

4-7-1-2　連成解析

連成解析のプログラム例を示します。図 4-7-3 (1) は構造解析、図 4-7-3 (2) は流体解析のプログラムで、これらを並列に実行します。両プログラムは似ているように見えますが、①に示すように使用する配列は異なっており、全く異なるプログラムだと考えて下さい。

両プログラムとも②に示す同じタイムステップ（変数名はわざと変えています）で反復し、毎日の反復でまず③の計算を行います。

計算が終了すると、④〜⑦でお互いに計算結果（圧力や温度などの物理量）を交換します。このとき、例えばランク 0 は④で配列 A のデータを送信し、ランク 1 は⑤でデータを (配列 A ではなく) 配列 AA に受信しますが、これは全く問題ありません。④の MPI_SEND は単に『配列 A のデータをランク 1 に送れ』と指示するだけで、ランク 1 が自分と同じプログラムなのか、あるいは受信する配列名は何なのかは全く関知しません。同様に⑤の MPI_RECV は単に『ランク 0 から送信されたデータを受信して配列 AA に入れよ』と指示するだけで、ランク 0 が自分と同じプログラムなのか、あるいは送信元の配列名は何なのかは全く関知しません。

⑧でランク 0 は、ランク 1 から送られた配列 X の値を使用して配列 A の値を補正します。ランク 1 も同様です。

補足ですが、④〜⑦で、ランク 0 とランク 1 では MPI_SEND と MPI_RECV の順序が逆になっています。これは 3-4-4 節で説明したデッドロックの発生を抑止するためです。もちろん MPI_ISEND, MPI_IRECV, MPI_WAIT を使用しても構いません。

ランク0

```
      PROGRAM KOUZOU
      PARAMETER (N=1000)
      REAL A(N),X(N)                                  ①
      DO ITIME = 1, 100                               ②
        DO I=1,N
          A(I) = ～            構造解析の計算          ③
        ENDDO
        CALL MPI_SEND(A,N,MPI_REAL,1,1,               ④
     &        MPI_COMM_WORLD,IERR)
        CALL MPI_RECV(X,1,MPI_REAL,1,1,               ⑥
     &        MPI_COMM_WORLD,ISTATUS,IERR)
        DO I=1,N
          A(I) = A(I)+X(I)     流体解析から             ⑧
        ENDDO    送られたデータを使って補正
      ENDDO
        :
```

図 4-7-3 (1)　構造解析

ランク1

```
      PROGRAM RYUTAI
      PARAMETER (NN=1000)
      REAL XX(NN),AA(NN)                              ①
      DO ISTEP = 1, 100                               ②
        DO I=1,NN
          XX(I) = ～           流体解析の計算          ③
        ENDDO
        CALL MPI_RECV(AA,NN,MPI_REAL,0,1,             ⑤
     &        MPI_COMM_WORLD,ISTATUS,IERR)
        CALL MPI_SEND(XX,NN,MPI_REAL,0,1              ⑦
     &        MPI_COMM_WORLD,IERR)
        DO I=1,NN
          XX(I) = XX(I)*AA(I) 構造解析から             ⑧
        ENDDO    送られたデータを使って補正
      ENDDO
        :
```

図 4-7-3 (2)　流体解析

- 上記の例では各プログラム内は並列化されていませんが、並列化することも可能です。図 4-7-4 では、構造解析のプログラムをランク 0〜2 で、流体解析のプログラムをランク 3〜5 で並列に実行しています。この場合、3-6 節で説明したように、構造解析と流体解析でそれぞれ新グループを作成し、新グループ内のプロセス間の通信は新グループのコミュニケータ (以下の例では IKOUZOU と IRYUTAI) を指定して行い、構造プロセスと流体プロセス間の通信は MPI_COMM_WORLD を指定して行います。

図 4-7-4

4-7-1-3　マスター・スレーブ方式

マスター・スレーブ方式とは

本節では、1-2 節で紹介したマスター・スレーブ（またはマスター・ワーカー）方式について説明します。マスター・スレーブ方式では、1 つの親プロセス（係長）が、その他全ての子プロセス（係員）に仕事を与え、子プロセス（係員）は仕事が終了したら親プロセス（係長）に報告し、親プロセス（係長）はその子プロセス（係員）に次の仕事を渡す、という動作をします。

マスター・スレーブ方式で並列化するのはどのような場合でしょうか？　例えば反復回数が 9 回の DO ループがあり、ループの各反復での仕事量が図 4-7-6 (1) のように著しくバラついているとします。これを、マシン性能（処理能力）が同一である 3 つのノードで並列に実行する場合、図 4-7-6 (2) のようにブロック分割しても、図 4-7-6 (3) のようにサイクリック分割しても、一般にロードバランスは不均等になります。

これをマスター・スレーブ方式で並列化した場合の動作を図 4-7-6 (4) に示します。まずランク 0（親プロセス）はランク 1〜3（子プロセス）に 1〜3 反復目の仕事を与え、各子プロセスは与えられた反復での仕事を行います。最も仕事量の少ないランク 3 が最初に終了し、ランク 0 に仕事の終了を報告します。ランク 0 はその時点での次の仕事である 4 反復目の仕事をランク 3 に与えます。以下同様に計算が行われます。

このようにマスター・スレーブ方式では、子プロセスが仕事を終了するたびに親プロセスはその時点での次の仕事を与え、各子プロセスは絶え間なく仕事をするので、ロードバランスは（通常）ほぼ均等になります。すなわちマスター・スレーブ方式は、ループの各反復での仕事量が著しくバラついている場合に有効となります。

図 4-7-6 (1)

図 4-7-6 (2)

図 4-7-6 (3)

図 4-7-6 (4)

上記は、各仕事量が著しくバラついていて各マシン性能（処理能力）は同一の場合でしたが、逆に、ループの各反復での仕事量は一定で、マシン性能（処理能力）がバラついている場合も有効となります。例えば性能の異なるノードから構成される並列計算機で並列ジョブを実行する場合や、性能の同じノードであっても、一部のノードで他のプログラムが動いていて、ノードによって処理能力が異なっている場合などが考えられます。

図 4-7-7 (1) に示すように、反復回数が 9 回の DO ループがあり、各反復での仕事量は同じであるとします。またランク 1 を実行するノードはマシン性能が高く、ランク 2 のノードは普通、ランク 3 のノードは低いとします。これをマスター・スレーブ方式で並列化した場合の動作を図 4-7-7 (2) に示します。

図 4-7-7 (1)

図 4-7-7 (2)

マスター・スレーブ方式に関していくつか補足します。

- 前述の例では、通信は親プロセスと子プロセス間のみで、子プロセス間の通信はありませんでしたが、子プロセス間で通信が必要なプログラムの場合、その時点で各子プロセスがどのループ反復を実行しているかはタイミングによって変わるので、マスター・スレーブ方式で並列化するのは難しくなります。
- 前述の例では、親プロセスは仕事の割り振りのみをしているので、通常、子プロセスと比べて仕事量は少なくなります。その場合、どれか 1 つのノードに親プロセスと 1 つの子プロセスの両方を実行させるという方法もあります。

マスター・スレーブ方式のプログラム例

図 4-7-8 の単体プログラムを例に、マスター・スレーブ方式で並列化する方法を説明します。①のループでは 1 回の反復で 1 つの仕事を行います。

各反復の②で [0〜1 の一様乱数] を 1 つ発生させ、それを 10 倍して整数 NSLEEP に代入します。つまり NSLEEP には 0〜9 のいずれかの整数が入ります。次に③で NSLEEP 秒間スリープします。従ってループの各反復での経過時間は、0〜9 秒のいずれかにバラつくことになります。

なお、[0〜1 の一様乱数] を生成する組込関数、およびスリープさせる組み込み関数はコンパイラによって異なるので、ここでは具体的に書きませんでした。

```
PROGRAM MAIN
DO I=1,10                          ①
  NSLEEP = [0〜1の一様乱数]*10.0 ②
  NSLEEP秒間スリープする。         ③
  PRINT *,'SLEEP TIME = ',NSLEEP
ENDDO
END
```

図 4-7-8

図 4-7-8 をマスター・スレーブ方式で並列化したプログラムのマスター部分を図 4-7-9 (1) に、スレーブ部分を図 4-7-9 (2) に示します。なお、本例ではマスター部分とスレーブ部分を同一プログラムとしましたが、別プログラムにする場合は (4-7-1-1 節参照)、図 4-7-9 (1) (2) の②，⑭，㉑を除去し、①を⑬にコピーし、⑫に END 文を追加して下さい。

(1)　マスタープログラムの動作

- まず図 4-7-9 (1) の④で、任意の子プロセスから仕事の要求が来るのを待ちます。送信元のプロセスが「任意」なので、下線に示すようにワイルドカード MPI_ANY_SOURCE を指定します（注）。④は単に子プロセスからの仕事の要求を受けるだけで、受信するデータそのものには意味がないので、受信バッファーには適当な変数 (IDUMMY) を指定します。
- ④が完了すると、配列 ISTATUS には送信元のプロセスのランクがセットされるので、⑤でランクの値を取り出して変数 IDEST にセットします（注）。
- ⑥で、乱数を発生させて子プロセスをスリープさせる秒数を変数 NSLEEP にセットし、⑦でランクが IDEST (つまり④の受信の送信元) のプロセスに対して NSLEEP の値を送信します。
- ③の DO ループを 10 回反復させた後、⑧の DO ループで全子プロセスを終了させます。まず⑨で任意の子プロセスから仕事の要求が来たら、仕事の終了を知らせるために、そのプロセスに対して⑩で NSLEEP に『-999』をセットし、⑪で送信します。⑧の DO ループは子プロセスの個数分 (= 全プロセス数 − 1) だけ反復します。

（注）「付録」のサブルーチン MPI_RECV，MPI_IRECV，MPI_WAIT の引数を参照して下さい。

(2)　スレーブプログラムの動作

- 各子プロセスは、図 4-7-9 (2) の⑮で無限ループします (この DO 文は Fortran 90 でサポートされている機能で、無限ループを意味します)。まず⑯で親プロセスに仕事の要求をします。これは④または⑨に対応する送信になります。
- ⑰で親プロセスからの応答を受信します。⑦から来た応答の場合はスリープする秒数が NSLEEP にセットされ、⑪から来た応答の場合はジョブの終了を示す『-999』が NSLEEP にセットされます。
- ⑱で NSLEEP の値を調べ、『-999』の場合は⑲で終了します。『-999』以外の場合は⑳で NSLEEP 秒間スリープします。

4 プロセス (ランク 0 が親プロセス、ランク 1〜3 が子プロセス) で実行した場合の結果を右に示します。

```
[aoyama@node01]/u/aoyama: mpirun -np 4 a.out ↵
 MYRANK = 3 SLEEP TIME = 3
 MYRANK = 3 SLEEP TIME = 1
 MYRANK = 1 SLEEP TIME = 5
 MYRANK = 2 SLEEP TIME = 5
 MYRANK = 2 SLEEP TIME = 5
 MYRANK = 1 SLEEP TIME = 6
 MYRANK = 3 SLEEP TIME = 7
 MYRANK = 3 SLEEP TIME = 1
 MYRANK = 2 SLEEP TIME = 4
 MYRANK = 1 SLEEP TIME = 8
```

```
      PROGRAM MAIN
      INCLUDE 'mpif.h'
      INTEGER ISTATUS(MPI_STATUS_SIZE)
      CALL MPI_INIT(IERR)
      CALL MPI_COMM_SIZE                                  ①
     &    (MPI_COMM_WORLD,NPROCS,IERR)
      CALL MPI_COMM_RANK
     &    (MPI_COMM_WORLD,MYRANK,IERR)
      IF(MYRANK==0) THEN マスタープロセスの処理②
        DO I=1,10          本処理                         ③
          CALL MPI_RECV(IDUMMY,1,MPI_INTEGER,             ④
     &          MPI_ANY_SOURCE,1,MPI_COMM_WORLD,
     &          ISTATUS,IERR)
          IDEST = ISTATUS(MPI_SOURCE)                     ⑤
          NSLEEP = [0〜1の一様乱数]*10.0                  ⑥
          CALL MPI_SEND(NSLEEP,1,MPI_INTEGER,             ⑦
     &          IDEST,1,MPI_COMM_WORLD,IERR)
        ENDDO
        DO I=1,NPROCS-1   終了処理                        ⑧
          CALL MPI_RECV(IDUMMY,1,MPI_INTEGER,             ⑨
     &          MPI_ANY_SOURCE,1,MPI_COMM_WORLD,
     &          ISTATUS,IERR)
          IDEST = ISTATUS(MPI_SOURCE)
          NSLEEP = -999                                   ⑩
          CALL MPI_SEND(NSLEEP,1,MPI_INTEGER,             ⑪
     &          IDEST,1,MPI_COMM_WORLD,IERR)
        ENDDO
        CALL MPI_FINALIZE(IERR)
                                                          ⑫
```

図 4-7-9 (1)

```
                                                          ⑬
      ELSE                スレーブプロセスの処理 ⑭
        DO                無限ループ             ⑮
          CALL MPI_SEND(IDUMMY,1,MPI_INTEGER,0,1,
     &          MPI_COMM_WORLD,IERR)                        ⑯
          CALL MPI_RECV(NSLEEP,1,MPI_INTEGER,0,1,
     &          MPI_COMM_WORLD,ISTATUS,IERR)                ⑰
          IF(NSLEEP==-999) THEN                           ⑱
            CALL MPI_FINALIZE(IERR)
            STOP                                          ⑲
          ENDIF
          NSLEEP秒間スリープする。                        ⑳
          PRINT *,'MYRANK = ',MYRANK,
     &          'SLEEP TIME = ',NSLEEP
        ENDDO
      ENDIF                                               ㉑
      END
```

図 4-7-9 (2)

4-7-2　フラット MPI とハイブリッド並列

本節では、図 4-7-10 (1) に示すように各ノードに複数 CPU が接続されていて、ノード間は分散メモリー型、ノード内は共有メモリー型 (以後 SMP) となっている並列計算機での並列化について説明します。なお、具体的なコンパイル・実行方法はマシン環境に依存しますので、マニュアルなどを参照して下さい。

2 CPU を使用する場合

元のプログラムで、図 4-7-10 (1) の DO ループは 1〜100 まで反復するとします。これを並列化し、4 CPU のうち 2 CPU を利用して実行する 3 つの方法を比較します。どの方法が最もパフォーマンスがよいかは、プログラムの特性に依存し、ケースバイケースです。

- 図 4-7-10 (1) では、MPI で並列化したプログラムを、ノード間にまたがる 2 CPU を使用して並列に実行しています。〓の数字は、各プロセスが担当する DO ループの下限と上限を示します。↔ に示すように、ノード間の通信のオーバーヘッドがかかります。
- 図 4-7-10 (2) では、SMP で並列化したプログラムを、ノード内の 2 CPU を使用して並列に実行しています。SMP の並列化では基本的に DO ループが並列化の対象となり、①に示すように指示行を指定して強制的に並列化するか、またはコンパイラが自動的に並列化します。並列化された DO ループは、スレッド（プロセスのようなもの）に分かれて実行が行なわれますが、本書ではこれ以上の説明は省略します。SMP では並列化された DO ループの前後 (DO ループに入る時と終了して出るとき) でオーバーヘッドが発生します。また同一ノード内の複数 CPU を使用するため、資源の競合 (メモリーや CPU など) によるオーバーヘッドが発生します。
- 図 4-7-10 (3) では、MPI で並列化したプログラムを、同一ノード内の 2 CPU を使用して並列に実行しています。同一ノード内の CPU 間で MPI の通信を行う場合、マシン環境によっては、実際に通信せずにメモリーコピーで代替し、通信時間を短縮できる場合があります。一方、同一ノード内の 2 CPU を使用するため、図 4-7-10 (2) と同様に資源の競合 (メモリーや CPU など) によるオーバーヘッドが発生します。

図 4-7-10 (1)

図 4-7-10 (2)

図 4-7-10 (3)

4 CPU を使用する場合

図 4-7-10 (1) の 4 CPU 全てを利用して並列化する 2 つの方法を比較します。どちらの方法がパフォーマンスがよいかはプログラムの特性に依存し、ケースバイケースです。

- 図 4-7-10 (4) では、MPI で並列化したプログラムを 4 プロセスで実行しています。このように MPI のみで並列化する方法をフラット MPI と呼びます。ノード間の通信によるオーバーヘッドと、ノード内の資源の競合 (メモリーや CPU など) によるオーバーヘッドが発生します。
- 図 4-7-10 (5) では、MPI で並列化したプログラムを 2 プロセスで実行し、さらに DO ループを SMP で並列化しています。このように MPI と SMP の両方で並列化する方法をハイブリッド並列と呼びます。ノード間の通信によるオーバーヘッド、図 4-7-10 (2) と同様にスレッド起動によるオーバーヘッドおよびノード内の資源の競合 (メモリーや CPU など) によるオーバーヘッドが発生します。
- 配列 A, B を縮小 (4-5-7 節参照) しなかった場合に、並列ジョブ全体で使用できる配列の大きさを比較してみます。配列 A のうちランク 0〜3 のプロセスおよびスレッド 0, 1 が担当する部分を、図 4-7-10 (4) (5) の下の着色した部分に示します。図 4-7-10 (4) では配列 A のうち 25% しか使用していませんが、図 4-7-10 (5) では 50% を使用しています。以上から次の事が分かります。MPI で並列化した図 4-7-10 (4) のプログラムに対し、大量のメモリーを使用するため配列の縮小を試み、何らかの理由（ロジックの修正が面倒など）で縮小できなかった場合、図 4-7-10 (5) のように SMP でさらに並列化すれば、図 4-7-10 (4) よりもメモリーを多く使用することができます。

図 4-7-10 (4)

図 4-7-10 (5)

4-8　並列化の手順と考慮点

本章ではプログラムを実際に並列化するための方法について述べてきました。本節ではそのまとめとして、並列化を行う手順およびパフォーマンスの評価方法について説明します。

4-8-1　並列化の手順

プログラムの並列化には、単体で稼働しているプログラムを並列化する場合と、最初から並列化を想定してプログラムを作成する場合があります。ここでは前者を想定し、単体プログラムを並列化する一般的な手順について説明します。

(1)　ホットスポットの特定

まず、単体でプログラムを実行した場合のホットスポット (CPU 時間を多く費やしているサブルーチンまたはループ) を、`prof` コマンドなど（マシン環境に依存します）を使用して調べます。

(2)　単体チューニング

ホットスポットの部分について単体チューニング (参考文献 [6] 参照) を行います。単体チューニングを行なって効果が出た場合には、CPU 時間の分布が変わるので、チューニングが終了したら再度 `prof` コマンドなどでサブルーチンごとの CPU 時間の割合を調べます。

(3)　並列化の全体方針の決定

`prof` コマンドなどの報告書をもとに、4-3 節のどのパターンで並列化するかを決定します。なお、並列化を行う前に、以下の方法で並列化の効果を見積り、並列化作業自体を行うべきかどうかを検討することができます。

- 並列化する部分の CPU 時間の割合がわかれば、並列化した場合の効果の上限を 4-1 節に述べたアムダールの法則を使って知ることができます。例えば並列化する部分が全体の 8 割しか占めていなければ、理想的な並列計算機を ∞ 台使用しても速度向上率は 5 倍以上にはなりません。
- 例えば並列化したい部分の CPU 時間が 400 秒かかり、並列化によって発生する通信量が大体 n バイトであることがわかっているとします。簡単なプログラムで n バイトの通信時間を測定し、例えば 4 ノードで 10 秒かかったとすると、CPU 時間は 4 ノードでほぼ 1/4 の 100 秒になりますので、$400 \div (100 + 10) = 3.64$ 倍の速度向上率になると見積もることができます。

(4)　並列化の詳細方針の決定

以下の点について並列化の詳細方針を決定します。

- 多重ループの場合、どのループを分割するか。言いかえると、多次元配列をどの次元で分割するか。また分割方法をブロック分割、サイクリック分割、ブロック・サイクリック分割のどれにするか。
- 配列の縮小を行うかどうか。
- どの変数（配列）をどこでどのプロセスに通信すれば、並列ジョブとして結果が正しくなり、しかも通信量が最小になるか。
- 並列版の数値計算ライブラリーで提供されているサブルーチンを使用するかどうか。

(5)　並列に動作させる最低限の修正

詳細方針が決定したら実際の並列化に入ります。まず、並列に動作させるための最低限の修正を行います。並列化する前のプログラムを図 4-8-1 (1) に示します。サブルーチン SUB の中に並列化したいループ (⑦) があり、その反復回数 N が①で与えられているとします。

並列に動作させるための最低限の修正を加えたプログラムを図 4-8-1 (2) に示します。

- 図 4-8-1 (2) のプログラムの開始直後、⑩で「三種の神器」を実行して全プロセス数 NPROCS、自分のランク値 MYRANK を取得します。
  これらの変数は、並列化する各サブルーチン (本例では SUB のみ) で使用しますが、引数で渡すとプログラムがゴチャゴチャするので、ここでは Fortran 90 の MODULE 文を使って渡します。⑧で PARA という名の MODULE 文を作成し、その中で NPROCS と MYRANK を宣言します。さらに INCLUDE 'mpif.h' や、1 対 1 受信サブルーチンで使用する配列 ISTATUS なども、並列化する各サブルーチンで毎回宣言するのは面倒なので、MODULE 文の中で宣言します。
  そして、MODULE 文内の変数を参照／更新したいサブルーチンは、MAIN 文または SUBROUTINE 文の直後（宣言文より前）に、⑨と⑰に示すように USE 文を指定するだけで、参照／更新が可能となります。なお、MODULE 文は使い方に少しクセがあるので、4-5-7-1 節「ブロック分割による配列の縮小方法 3」の注意を厳守して下さい。
  並列プログラムを 1 プロセスで実行すると、誤動作する可能性がありますので (4-8-2 節の「プログラムの並列化方法のミス (1)」参照)、⑪でチェックを行い、1 以下ならばエラーで終了します。
- プログラムが終了する前に MPI_FINALIZE を一度だけ実行します。本例では③と⑥でプログラムが終了するので、⑯と⑲を追加します。3-2 節で説明したように、MPI_FINALIZE を指定せずに終了すると、マシン環境によっては誤動作することがありますので、プログラムが終了する箇所が複数ある場合、全ての箇所で必ず 1 度 MPI_FINALIZE を実行してから終了するようにして下さい。
- 次に出力部分を修正します。②と⑤の出力部分 (OPEN 文や CLOSE 文がある場合はそれも同様に) は、ランク 0 だけが出力するように、⑮と⑱の IF 文を追加します (入出力部分の修正方法の詳細は 4-4 節を参照)。
  ④はプログラム内でエラーをチェックしている IF 文です。最終的に並列化したプログラムで、このエラーを発見するのが全プロセスの場合は⑱の修正になります。一方、エラーを発見するのが特定のプロセスのみの場合、⑱だと、エラーを発見したプロセスのみが STOP 文で終了し、他のプロセスは IF 文が偽になるので終了しません。特定のプロセスがエラーを発見したとき、全てのプロセスを終了させたい場合は、図 4-8-1 (3) の㉒に示すように、CALL MPI_ABORT（付録参照）を実行して下さい。また特定のプロセスがランク 0 以外の場合、⑱の IF 文がついていると PRINT 文が実行されないので、㉑に示すように IF 文をはずします。また、どのプロセスがこのエラーを発見したのかを知りたい場合は、㉑の下線部を追加します。

これでとりあえず、プログラムが並列に実行できるようになりました。ただしこの時点では本当の意味の並列化をした部分はなく、プログラムを並列に実行すると、全プロセスが単体ジョブと全く同じ動作をし、ランク 0 のプロセスのみが出力を行います。

元の単体プログラムと上記の並列プログラムで、標準出力とその他の出力ファイルの内容が一致しているのを確認した後、本当の意味の並列化を行います。以下で概要を説明し、次の項で並列化を効率的に行うための方法を紹介します。

- ⑫ (4-5-4 節参照) で、⑦のループ反復のうち自プロセスが担当する反復の下限 (ISTA) と上限 (IEND) を求め、それを⑧に追加し、⑳のようにループ反復を修正します。
  なお、1 つのプロセスが担当する要素数 (本例では IEND-ISTA+1) が少なすぎると、並列プログラムによっては誤動作しますので (4-8-2 節の「プログラムの並列化方法のミス (2)」参照)、⑬で正常に動作する最低限の要素数 (例えば 2) 以上かどうかをチェックし、それより少なければエラーで終了します（しきい値はプログラムのロジックに依存しますので、ロジックに応じて適切な値を設定して下さい)。
  各プロセスが計算した結果を最後にランク 0 に集めるため、⑭の通信を行います。

```
PROGRAM MAIN
  :
N = 1000                          ①
CALL SUB(N)
WRITE(10) X                       ②
STOP                              ③
END
SUBROUTINE SUB(N)
  :
IF (エラーを発見したら)            ④
  PRINT *,'===ERROR==='           ⑤
  STOP                            ⑥
ENDIF
DO I=1,N                          ⑦
  X(I) = A(I) + B(I)
ENDDO
  :
```

図 4-8-1 (1)

→

```
MODULE PARA
INCLUDE 'mpif.h'
INTEGER ISTATUS(MPI_STATUS_SIZE)          ⑧
INTEGER NPROCS,MYRANK,ISTA,IEND
END
PROGRAM MAIN
USE PARA                                  ⑨
  :
N = 1000
CALL MPI_INIT(IERR)
CALL MPI_COMM_SIZE
&    (MPI_COMM_WORLD,NPROCS,IERR)         ⑩
CALL MPI_COMM_RANK
&    (MPI_COMM_WORLD,MYRANK,IERR)
IF (NPROCS<=1) THEN
  IF (MYRANK==0)
&      PRINT *,'===PARALLEL ERROR1==='    ⑪
  CALL MPI_FINALIZE(IERR)
  STOP
ENDIF
CALL PARA_RANGE                           ⑫
&    (1,N,NPROCS,MYRANK,ISTA,IEND)
IF (IEND-ISTA+1<2) THEN
  PRINT *,'===PARALLEL ERROR2===',
&          'MYRANK=',MYRANK
  CALL MPI_ABORT                          ⑬
&      (MPI_COMM_WORLD,9,IERR)
  CALL MPI_FINALIZE(IERR)
  STOP
ENDIF
CALL SUB(N)
各プロセスが計算した配列Xの結果を          ⑭
ランク0に集める通信を行う。
IF (MYRANK==0) WRITE(10) X                ⑮
CALL MPI_FINALIZE(IERR)                   ⑯
STOP
END
SUBROUTINE SUB(N)
USE PARA                                  ⑰
  :
IF (エラーを発見したら)
  IF (MYRANK==0) PRINT *,'===ERROR==='    ⑱
  CALL MPI_FINALIZE(IERR)                 ⑲
  STOP
ENDIF
DO I=ISTA,IEND                            ⑳
  X(I) = B(I) + C(I)
ENDDO
  :
```

図 4-8-1 (2)　全プロセスがエラーを発見する場合

```
  :
IF (エラーを発見したら)
  PRINT *,'===ERROR===','MYRANK=',MYRANK  ㉑
  CALL MPI_ABORT(MPI_COMM_WORLD,9,IERR)   ㉒
  CALL MPI_FINALIZE(IERR)
  STOP
ENDIF
  :
```

図 4-8-1 (3)　特定のプロセスだけがエラーを発見する場合

(6)　並列化

並列に動作させるための最低限の修正を行ったプログラムに対し、前述のように本当の意味の並列化を行います。並列化というのは通常のプログラム作成と感覚が違っており、慣れないうちは答えが合う状態に持っていくのがなかなか大変です。以下で、効率的に並列化するための方法をいくつか紹介します。

【プロトタイプの作成】

大規模なプログラムや、並列化によって複雑な動作をするプログラムの場合、いきなり並列化するのではなく、実際のプログラムのうち並列化に関係のある本質的な特徴のみを抽出した100行以下の簡単な単体プログラムを作成し、それを並列化して、単体版と答えが同じになることを確認します。この過程で、本物のプログラムを並列化するために必要なノウハウ（例えば並列化に伴いプログラムのどこでどのような通信を行えば良いか、そのためにはどのようなテーブル類が必要か、など）を蓄積することができます。それに基づいて実際のプログラムを並列化すると、効率的に並列化することができます。

例えば図4-8-2 (1) が10000行のプログラムで、そのうち計算時間のかかっている部分が①の2重ループだとします。これに対し、以下の点を考慮して簡単化したプログラム（単体版）を図4-8-2 (2) に示します。

- ①のループは1回計算するだけでテストできますので、タイムステップ・ループは削除しました。
- ①のループ以外の計算部分を削除しました。
- ①のループの2次元配列Pを1次元にし、サイズも小さくしました。
- ①のループで並列化に関係のある本質的な特徴は、P(I-1,J), P(I,J), P(I+1,J) を使用してP(I,J) を計算していることなので、③ではその部分のみを抽出し、関係のない配列や変数 (A, B など) は全て除去しました。
- 本物のデータを使用するのではなく、②に示すように人工的で簡単なデータを設定しました。これによってデバッグがしやすくなります。

```
    :
 入力データの読み込みと初期設定
 DO ITIME=1,ITMAX (タイムステップループ)
    :
   EPS = 0.0
   DO J=2,999        ①
     DO I=2,999      ①
       S = A(I,J)*(P(I+1,J  )+P(I  ,J+1))
&          +B(I,J)*(P(I+1,J+1)-P(I+1,J-1)
&                  -P(I-1,J+1)+P(I-1,J-1))
&          +C(I,J)*(P(I-1,J  )+P(I,J-1  ))
       S = (S*D(I,J)-P(I,J))*E(I,J)
       EPS = EPS + S*S
       P(I,J) = P(I,J) + OMEGA*S
     ENDDO
   ENDDO
    :
 ENDDO
 WRITE(10) P
 END
```

図4-8-2 (1)　10000行のプログラム

→

```
    :
 DO I=1,10
   P(I) = I                    ②
 ENDDO
 DO I=2,9
   P(I) = P(I-1)+P(I)+P(I+1)  ③
 ENDDO
 DO I=1,10
   PRINT *,I,P(I)
 ENDDO
 END
```

図4-8-2 (2)　100行以下のプログラム

【データの動きが分かる図の作成】

1 次元 SOR 法 (5-6 節参照) の例で説明します。

図 4-8-3 (1) を実行したときの配列 P の状態の推移を図 4-8-3 (2) に示します。○が計算前、□が計算後の要素、「境」は境界条件、↘と↙は要素の依存関係を表します。II = 0 のとき②, ④, ...を計算し、II = 1 のとき③, ⑤, ...を計算します。

これを並列化した図 4-8-3 (3) (4) では、まず↘の通信を行ってから②, ④, ...を計算し、次に↙の通信を行ってから③, ⑤, ...を計算します。

この例のように、並列化したときのデータの動きが複雑な場合、図 4-8-3 (2) (4) のように、データの動きが分かる図 (要素数は必要最小限とし、プロセス数は必要最小限の 3) を作成すると、並列化するときの参考になります。

```
DIMENSION P(13)
  :
DO II=0,1
  DO I=2+II,12,2
    P(I)=P(I-1)+P(I+1)
  ENDDO
ENDDO
  :
```

図 4-8-3 (1)

図 4-8-3 (2)

```
DIMENSION P(13)
  :
DO II=0,1
  II=0のとき↘の通信
  II=1のとき↙の通信
  DO I=ISTA+II,IEND,2
    P(I)=P(I-1)+P(I+1)
  ENDDO
ENDDO
  :
```

図 4-8-3 (3)

図 4-8-3 (4)

【結果を確認しながら少しずつ並列化】

並列化に伴う修正箇所が多い場合 (例えば差分法で現れる 4-3 節の「計算パターン 2」)、並列化に伴う修正を全て行なってからデバッグするという方法では、バグがプログラムのあちこちに混入してしまい、バグを探すのが『太平洋に落ちた針を探す』ように非常に難しくなります。

このような場合、少し並列化しては結果が正しいことを確認し、『だましだまし』少しずつ並列化の範囲を広げていくと、効率よく機械的に並列化することができます。この方法を図 4-8-4 (1) の例で説明します。

1 図 4-8-4 (1) のタイムステップ・ループ内に 3 つの（実際にはもっと多くの）DO ループがあり、これを全て並列化するとします。まず図 4-8-4 (2) のように 1 つ目のループを（本当の意味の）並列化します (ループ反復を ISTA, IEND に修正)。それに伴って図 4-8-5 (1) のようなシフト通信が必要であれば、それを行うサブルーチン SHIFT (4-6-2 節参照) をコールします。なお、図中で（本当の意味の）並列化した部分を『並』で表します。

1 つ目のループを並列化したため、1 つ目のループの終了後、各プロセスは、このループ内で値が設定された配列 A のうち、自分が計算した部分のデータしか持っていません。通常、配列 A は 1 つ目のループ以後の処理で参照／更新が行われるので、ここまでの修正を行っただけでは、並列に実行しても単体版と同じ計算結果にはなりません (4-3 節の「計算パターン 1」と同様)。

従って、この段階で正しい計算結果を得るためには、1 つ目のループを終了した直後に、図 4-8-5 (2) に示すように、各プロセスは、配列 A のうち自分が計算した部分のデータを、他の全プロセスに送信する必要があります。そこで、これを行うサブルーチン DEBUG (4-6-3-2 節参照) をコールします。これで各プロセスの配列 A には全てのデータが入り、単体プログラムと同じ状態になるため、並列に実行すると（バグがなければ）正しい計算結果が得られます。もし結果がおかしい場合は、並列化した 1 つ目のループ内か、サブルーチン SHIFT、DEBUG のいずれかにバグがあることになります。このようにバグの存在する範囲が狭い範囲に限定されるので、バグを探すのが容易になります。なお、サブルーチン DEBUG は通信量が多いので、この時点でパフォーマンスは（一般に）却って低下します。

2 次に図 4-8-4 (3) に示すように 2 つ目のループを並列化し、ループ内で値が設定された配列 W と B に対し、デバッグ用のサブルーチン DEBUG コールします。このとき、図 4-8-4 (2) で挿入した CALL DEBUG(A) を『並』部分の先頭に移動します。このように、CALL DEBUG は常に『並』部分の先頭に置くようにします。そして並列に実行し、結果が正しくなるまでデバッグします。

3 図 4-8-4 (4) に示すように 3 つ目のループを同様に並列化し、ループ内で値が設定された配列 X に対し、デバッグ用のサブルーチン DEBUG をコールします。

配列 W は 2 つ目のループで値が設定され、3 つ目のループで参照され、それ以外の部分では使われていないとします。つまり配列 W は、2 つ目と 3 つ目のループ間でデータを受け渡す作業配列としてのみ使用されているとします。この場合、もはや CALL DEBUG(W) は不要なので、『並』部分の先頭に移動せずに削除します。CALL DEBUG(A) と CALL DEBUG(B) を『並』部分の先頭に移動して並列に実行し、結果が正しくなるまでデバッグします。

4 このように、結果が正しいことを確認しながら少しずつ並列化の範囲を拡大していくと、ついには図 4-8-4 (4) のようにタイムステップ内が全て『並』となり、しかも正しい結果が維持されています。ただしこの時点では、CALL DEBUG の影響でパフォーマンスは（一般に）かなり低下しています。

5 プログラムの最後で、配列 X の結果を 10 番ファイルに書き出しています。そこで例えば図 4-8-4 (5) に示すように、図 4-8-5 (3) の通信を行うサブルーチン GATHER (4-6-3-1 節を参照) をコールし、ランク 0 が代表してファイルに書き出すようにし (他の出力方法については 4-4-2 節参照)、結果が正しくなるまでデバッグします。

最後にデバッグ用の CALL DEBUG を全て取り除くと、一気にパフォーマンスは向上し、結果も正しいはず…なのですが、もしここで結果がおかしくなった場合は、CALL DEBUG を（全てではなく）一つずつ取り除いていき、どれを取り除くと結果がおかしくなるのかを調べて下さい。

```
    :
DO タイムステップ
  DO I=1,N
      :
    A(I) = ～
  ENDDO
  DO I=1,N
      :
    W(I) = ～
    B(I) = ～
  ENDDO
  DO I=1,N
      :
    ～ = W(I)
    X(I) = ～
  ENDDO
ENDDO
WRITE(10) X
    :
```

図 4-8-4 (1)

```
    :
DO タイムステップ
並 CALL SHIFT(A)
並 DO I=ISTA,IEND
並     :
並   A(I) = ～
並 ENDDO
  CALL DEBUG(A)
  DO I=1,N
      :
    W(I) = ～
    B(I) = ～
  ENDDO
  DO I=1,N
      :
    ～ = W(I)
    X(I) = ～
  ENDDO
ENDDO
WRITE(10) X
    :
```

図 4-8-4 (2)

```
    :
DO タイムステップ
並 CALL SHIFT(A)
並 DO I=ISTA,IEND
並     :
並   A(I) = ～
並 ENDDO
並 CALL SHIFT(B)
並 DO I=ISTA,IEND
並     :
並   W(I) = ～
並   B(I) = ～
並 ENDDO
  CALL DEBUG(A)
  CALL DEBUG(W)
  CALL DEBUG(B)
  DO I=1,N
      :
    ～ = W(I)
    X(I) = ～
  ENDDO
ENDDO
WRITE(10) X
    :
```

図 4-8-4 (3)

```
    :
DO タイムステップ
並 CALL SHIFT(A)
並 DO I=ISTA,IEND
並     :
並   A(I) = ～
並 ENDDO
並 CALL SHIFT(B)
並 DO I=ISTA,IEND
並     :
並   W(I) = ～
並   B(I) = ～
並 ENDDO
並 CALL SHIFT(X)
並 DO I=ISTA,IEND
並     :
並   ～ = W(I)
並   X(I) = ～
並 ENDDO
  CALL DEBUG(A)
  CALL DEBUG(B)
  CALL DEBUG(X)
ENDDO
WRITE(10) X
    :
```

図 4-8-4 (4)

```
    :
DO タイムステップ
並 CALL SHIFT(A)
並 DO I=ISTA,IEND
並     :
並   A(I) = ～
並 ENDDO
並 CALL SHIFT(B)
並 DO I=ISTA,IEND
並     :
並   W(I) = ～
並   B(I) = ～
並 ENDDO
並 CALL SHIFT(X)
並 DO I=ISTA,IEND
並     :
並   ～ = W(I)
並   X(I) = ～
並 ENDDO
  CALL DEBUG(A)
  CALL DEBUG(B)
  CALL DEBUG(X)
ENDDO
CALL GATHER(X)
IF(MYRANK==0)
  WRITE(10) X
ENDIF
    :
```

図 4-8-4 (5)

図 4-8-5 (1)　サブルーチン SHIFT

図 4-8-5 (2)　サブルーチン DEBUG

図 4-8-5 (3)　サブルーチン GATHER

4-8-2　エラーかなとおもったら

プログラムを並列化した場合の典型的なエラーとその対処法について説明します（本文ですでに説明済みの項目もありますが、念のため再掲します）。なお、下記の内容はマシン環境によっても異なります。

システムのトラブル

並列プログラムは正しいはずなのに実行すると挙動がおかしい場合、システムのトラブルの可能性がありますので、3-2 節の図 3-2-1 (1) のような簡単なプログラムを実行し、システムが正常に動作しているかどうかをまず確認して下さい。

実行方法のミス

各ノードのローカルディスクに置かれたロードモジュール a.out を使用して並列ジョブを実行する環境では、並列ジョブの実行前にあらかじめ a.out を各ノードのローカルディスクにコピーします。例えば並列プログラムの誤りを修正し、再コンパイル・リンクして実行したところ、結果が修正前と同じでおかしいなと思い、原因を調べたところ a.out をコピーし忘れていたという状況が発生しますので、コピーのし忘れには注意して下さい。

結果の合法的な相違（誤差など）

- 並列化に伴う修正によって、その付近のロジックに対するコンパイラの最適化の度合が変わり、計算結果が変わることがあります。この場合、試しにコンパイルオプションを指定せずに単体版プログラムと並列版プログラムを実行し、もし結果が同一になれば、（コンパイラのバグでない限り）最適化の度合が変わったことが原因であることになります。
- 合計や内積を求める DO ループを並列化した場合、単体で実行したのと計算順序が変わるので、誤差によって合計の結果も若干変わります (3-3-6 節参照)。敏感なプログラムの場合、これがその後の計算結果に影響することがあります。これは合法的な誤差ですが、プログラムが単精度の場合には倍精度にすれば回避できる場合もあります。
- 乱数を使用したプログラムを並列化した場合は、発生した乱数系列が単体版のプログラムと（通常）変わるので、結果も変わります。

MPI 自体のバグ

MPI のルーチン自体にバグがある可能性が全くないとは言えません（例えば集団通信ルーチンの内部でデッドロックしてしまうなど）。MPI のルーチンがおかしいと思われる場合、もし他の MPI を使用するマシン環境があればそこで実行し、同様の現象が起こらなければ、最初のマシン環境での MPI のバグだと思われます。

その場合、電算部門やメーカーの担当者に問い合わせますが、その際プログラムや入出力データを、なるべく以下のように簡単化してから問い合わせた方がよいでしょう。その目的は、担当者がバグの解析をしやすく回答が速く得られることと、簡単化する過程で、例えばプログラムのこの部分を削除したらエラーが起きなくなったなど、エラーの手がかりが見つかることがあるためです。

- プログラムを、問題が発生する最低限の行数 (出来れば数 10 行) に縮小する。
- 入出力ファイルの数やデータ量を最小限（出来ればゼロ）にする。
- 問題が発生するまでの実行時間を最短（出来れば数秒）にする。

元の単体プログラムに含まれている潜在的なバグ

並列化する前の単体プログラムに含まれている潜在的なバグ（例えば配列の範囲を越えて代入を行ったため記憶域を壊していたなど）が、並列化したことによって計算結果の相違として表面に現れる場合があります。その場合、単体版または並列版のプログラムにデバッグ用のコンパイル・リンクオプション（例えばモジュール間の整合性をチェックするオプション）を付けて実行すると、その箇所を発見できる場合があります。あるいは上の項目で説明した、プログラムを簡単化する過程でその箇所を発見できる場合もあります。

マシン環境による MPI の制限

- (4-4-1 節) 標準入力 (装置番号 5 番) から読み込みを行うと、マシン環境によってはエラーになる場合があるのでお勧めできません。
- (3-3-4 節、3-4-5 節) 1 対 1 通信および集団通信ルーチンで一度に送信できる最大のバイト数はマシン環境によって異なり、無制限の場合と、制限のある場合があります。制限のある場合に、制限を越えたバイト数を送ってもエラーにならず、不完全なメッセージが送られてしまうマシン環境もあります。
- (3-4-5 節) 1 対 1 通信ルーチンで仕掛かり中のメッセージがどんどん累積していった場合、マシン環境によっては例えば受信バッファーがパンクして異常終了するなどの現象が発生します。

MPI の使い方のミス

- MPI の使い方自体に不明な点がある場合は、参考文献 [15] に記載したアドレスにメール（ただし英語）すると、回答が得られる場合があります。
- (3-2 節) 「MPI_」、「PMPI_」で始まる変数名、関数名、サブルーチン名をユーザープログラムで使用すると、MPI が使用する変数と偶然一致して誤動作する危険性がありますので、使用しないで下さい。
- (3-2 節) MPI_FINALIZE をコールせずに終了した場合、マシン環境によっては誤動作する（実行中の他のプロセスを強制的に終了させてしまう、あるいは実行するたびに計算結果が異なる再現性のないエラーとなる）ことがあります。プログラムが終了する全ての箇所 (STOP 文または END 文) で、必ず 1 度だけ MPI_FINALIZE をコールしてから終了するようにして下さい。
- (3-3-4 節) MPI のサブルーチンの引数が少なかったり指定がおかしくても、一般にコンパイル・リンクではエラーになりません。この場合、実行時に Segmentation fault などのエラーで異常終了しますが、エラーメッセージを見ても原因が分からない場合がありますので、異常が起きた場合には MPI のサブルーチンの引数の数や属性に誤りがないかどうかをチェックして下さい。特に多いのは以下のエラーです。
  - CALL MPI_xxx(〜,IERR) の整数 IERR を指定し忘れた場合。
  - MPI_RECV, MPI_WAIT で使用する引数 status を、INTEGER ISTATUS(MPI_STATUS_SIZE) で宣言し忘れた場合。
  - INCLUDE 'mpif.h' を指定し忘れた場合。
- (3-3-4 節) MPI の仕様書 (参考文献 [15] では、ユーザーが MPI の使い方を間違えた場合、実行時に MPI が行うエラー処理について一般に規定していません。このため、例えば MPI_BCAST などの集団通信ルーチンで、送信バッファーの大きさより受信バッファーの大きさを誤って小さくしてしまったような場合でも、受信時にエラーにならず、エラーのリターンコード (ierror) にも正常終了の「MPI_SUCCESS」が返るマシン環境もあります。
- (3-4-2 節) 非ブロッキング通信の MPI_ISEND または MPI_IRECV を指定して MPI_WAIT を指定し忘れた場合、タイミングによる（再現性のない）エラーが発生します。また MPI_ISEND または MPI_IRECV と MPI_WAIT の引数 request がスペルミスで異なっている場合も、当然ながら MPI_WAIT は働かず、タイミングによる（再現性のない）エラーが発生しますので、スペルミスには十分注意して下さい（一方のスペルをカット・アンド・ペーストで他方にコピーするのが安全です）。
- (3-4-4 節) 双方向通信を行う際、1 対 1 通信ルーチンを実行する順序がおかしいと、デッドロックが発生してジョブが止まってしまいます。デッドロックになっているかどうかは、通信の前後に PRINT 文を入れてランク番号を書き出し、各プロセスがその通信を終了しているかどうかで調べることができます。
- (3-4-5 節、3-3-4 節) 1 対 1 通信ルーチンでは、受信バッファーで指定するバイト数 (データ個数 × データ型のバイト数) は、送信バッファーで指定するバイト数と等しいか、または大きくても構いません。一方集団通信ルーチンでは、送受信バッファーで指定するバイト数は一致していなければなりません。

プログラムの並列化方法のミス

(1) 並列プログラムを 1 プロセスで実行すると、誤動作する可能性がありますので (3-4-5 節の「1 対 1 通信を 1 プロセスで実行した場合」、4-6-2 節の「境界条件の処理」参照)、1 プロセスで実行するときは、並列版ではなく単体版のプログラムを使用することを強くお勧めします。並列プログラムの開始時に、全プロセス数が 1 以下の場合はエラーで終了させるようにして下さい (4-8-1 節の「(5) 並列に動作させる最低限の修正」参照)。

(2) 図 4-8-3 (2) のプログラムでは、図 4-8-3 (1) に示すように、配列 A の 3 つの要素から配列 B の要素を計算します。これを並列化した図 4-8-3 (3) では、各プロセスはブロック分割 (4-5-4 節参照) で決定した ISTA から IEND までの要素を担当します。これを 2 プロセスで並列に実行した場合、図 4-8-3 (4) に示すように、ランク 0 の右端の 2 要素をランク 1 のプロセスに通信 (↘ の部分) してから計算に入ります。

3 プロセスで並列に実行した場合、図 4-8-3 (5) に示すように、ランク 0 とランク 1 のプロセスの右端の 2 要素を右隣のプロセスに通信します。ところがランク 1 の×（値が未定義）の要素がランク 2 に送られ、ランク 2 は×の要素を使用して計算を行うため、誤った計算結果になってしまいます。

これはランク 1 のプロセスが担当する要素数が 1 つしかないのが原因です。つまりこの例では、各プロセスが担当する要素数が最低 2 以上でないと、正常に動作しません。

このように、1 つのプロセスが担当する要素数が少なすぎると、並列プログラムによっては誤動作します。その並列プログラムで、1 つのプロセスが担当する要素数が最低いくつあれば正常に動作するかを把握し、プログラムの実行が開始して各プロセスが担当する要素数が決定した時点で、正常に動作する最低限の要素数以上かどうかをチェックするようにして下さい (4-8-1 節の「(5) 並列に動作させる最低限の修正」参照)。

図 4-8-3 (1)

```
   :
DO I=1,4
  B(I)=A(I-2)+A(I-1)+A(I)
ENDDO
   :
```
図 4-8-3 (2)

```
   :
 配列Aの右端の2要素を通信
DO I=ISTA,IEND
  B(I)=A(I-2)+A(I-1)+A(I)
ENDDO
   :
```
図 4-8-3 (3)

図 4-8-3 (4)

図 4-8-3 (5)

(3) 4-5-4 節で説明したブロック分割がうまくいっていないため、各プロセスが担当するループ反復の下限 (ISTA) と上限 (IEND) の値がおかしい場合があります。全プロセスの ISTA と IEND の値を PRINT 文などで書き出し、確認して下さい。

(4) 並列性のないループ（回帰参照のあるループ）をうっかり並列化した場合、当然ながら結果がおかしくなります。単体版プログラムでそのループを逆に反復させてみて、元の計算結果と変わるようであれば、一般にそのループに並列性はありません（ただし例外もあります）。

4-8-3　パフォーマンスの測定と評価

ここまでで一応並列化が完了しました。実は、並列化には大きく以下の 2 つの関門があり、まだ (1) の関門を突破したに過ぎないのです。

(1) 並列化して結果が合うこと
(2) 良好なパフォーマンスが得られること

本節では、(2) に関係のある、パフォーマンスの測定および評価方法について説明します。

並列ジョブの評価方法

単体ジョブの場合、主に CPU 時間で測定を行いますが、並列ジョブの場合は以下のように通信時間と通信待ち時間が加わるため、経過時間で測定を行います。

図 4-8-5

並列ジョブの効果を評価する方法として、以下に示す速度向上率がよく使用されます。例えば 4 台で実行した場合、完全に並列化できた場合には速度向上率が 4 倍となりますが、通常は通信などのオーバーヘッドで効率が低下し、4 倍以下になります。

$$\text{速度向上率} = (1\text{ 台での経過時間})/(n\text{ 台での経過時間})$$

速度向上率に関する注意点を以下に示します。

- 分子の (1 台での経過時間) は、並列版のプログラムをプロセス数 1 として実行した場合の経過時間ではなく、単体のプログラムで測定した経過時間にして下さい。また経過時間は、他のジョブが流れていない占有環境で測定して下さい。
- ロードモジュール `a.out` や入力データをローカルディスクに置いた場合と共有ディスクに置いた場合で経過時間が異なります。特にノード数が多い場合、共有ディスクを使用すると多くのノードが同じファイルをアクセスして競合するため、最初のノードにロードモジュールがロードされた時刻と最後のノードにロードされた時刻が大きく離れてしまう場合があります。
- 一般的なプログラムでは処理は以下のような流れになります。
  (1) 入力データの読み込みや初期設定
  (2) 計算部分
  (3) 計算結果の出力
  通常は (2) の部分が経過時間のほとんどを占め、(1) と (3) の割合がわずかです。ところが、測定時間を短縮するために通常よりも短いジョブで測定した場合、(1) や (3) の相対的な割合が高くなってしまいます。さらに、(1) や (3) は通常並列化しないのでますます割合が高くなってしまい、本来ならば計算時間のほとんどを占めるはずの (2) の部分が正当に評価できなくなってしまいます。このような場合、(2) の部分だけをプログラム内でタイマーを使用して測定した方がよいでしょう。タイマーを使用した測定方法については後述します。

- 速度向上率はあくまで相対的な尺度であり、絶対的な評価基準はあくまで経過時間そのものであることに注意して下さい。

例えば図 4-8-6 (1) に示すように、ある単体プログラムの CPU 時間が 400 分かかっており、計算部分全体をロードバランスが均等になるように並列化できたとします。これを 4 ノードで実行したところ、CPU 時間は 100 分（着色部分）になります。また並列化に伴い、通信時間が 20 分（白い部分）かかったとします。この場合、速度向上率は 3.3 倍と、まずまずの値となります。

このプログラムを単体チューニングし、図 4-8-6 (2) に示すように、単体プログラムの CPU 時間が 1/4 の 100 分になったとします。これを並列に実行すると、通信時間は単体チューニング前と変わらないので、経過時間は 45 分になり、速度向上率は 2.2 倍にしかなりません。

このように、単体チューニングをすると、経過時間自体は短くなりますが、速度向上率は低下します。

一方、ある並列プログラムをある並列計算機で実行したところ、図 4-8-6 (2) のように速度向上率が 2.2 倍だったとします。同じ並列プログラムを、通信性能が同じで CPU 性能が 4 倍遅い並列計算機で実行すると、図 4-8-6 (1) のように速度向上率は 3.3 倍になります。

このように、同じ並列プログラムの場合、通信性能が同じであれば、CPU 性能の低い並列計算機の方が、速度向上率は高くなります。また同じ理屈で、同じ並列プログラムの場合、CPU 性能が同じであれば、通信性能の高い並列計算機の方が、速度向上率は高くなります。

まとめると、同じ並列プログラムでも、並列計算機の CPU 性能と通信性能の比率によって、速度向上率は変動します。

図 4-8-6 (1)　単体チューニング前／CPU 性能の低い並列計算機

図 4-8-6 (2)　単体チューニング後／CPU 性能の高い並列計算機

経過時間の測定方法

図 4-8-7 の①のように time コマンド (または timex コマンド) を指定してジョブを実行すると、②のような結果が表示されます（表示内容はシェルによって異なります）。real はジョブ全体の経過時間（秒）を示します。user は、単体ジョブの場合は CPU 時間ですが、並列ジョブの場合、マシン環境によって表示が異なり、図のようにほとんどゼロの値が表示される場合と、全プロセスの CPU 時間の合計が表示される場合がありますので、マニュアルなどで確認して下さい。

```
[aoyama@node01]/u/aoyama: time mpirun -np 3 a.out ↵  ··· ①

real    0m16.84s
user    0m0.04s     ②
sys     0m0.09s
```

図 4-8-7

プログラム内の特定の部分の経過時間の測定

並列化したプログラム内の特定の部分の経過時間を測定したい場合、MPI のファンクション MPI_WTIME() を使用します（「付録」参照）。図 4-8-8 (1) に示すように、「測定対象部分」の前後で②と③を実行すると、変数 ELP1 と ELP2（変数名は任意）に（内部的な）時刻が戻ります。④で両者の差をとれば、測定部分の経過時間（単位は秒）が求まります。なお、MPI_WTIME() は倍精度なので、①の指定を必ず行って下さい。図 4-8-8 (1) を 3 プロセスで実行した場合のタイムチャート例を図 4-8-8 (2) に示します。

ところで、並列プログラムのロードモジュール a.out が共有ディスクにある場合、a.out のロードが完了してプログラムが開始する時刻がプロセスによって若干異なります。また各プロセスが同一ファイルから入力データを読み込む場合も、読み込みを完了する時刻がプロセスによって異なります。このように、プログラムが開始した直後は一般に各プロセスの進み具合が若干異なっており、その影響で、図 4-8-8 (2) の★に示すように、各プロセスが「測定対象部分」に到達する時刻が若干異なることがあります。

この影響をなくすためには、図 4-8-8 (3) の⑤に示すように集団通信サブルーチン MPI_BARRIER（「付録」参照）を使用します。すると⑤の時点で同期が取られるため、タイムチャートは図 4-8-8 (4) のようになり、「測定対象部分」の開始時点でのバラつきはなくなります。

例えば、DO ループをブロック分割で並列化し、各プロセスが担当している計算量のロードバランスが均等になっているかどうかを調べたい場合にこの方法を用います。

一方、図 4-8-8 (5) の⑥，⑦に示すように、「測定対象部分」の前後に MPI_BARRIER を使用すると、タイムチャートは図 4-8-8 (6) のようになり、「測定対象部分」の最長の経過時間が測定されます。

例えば、プログラムの主要計算部分（プログラムの最初と最後に 1 度だけ行う入出力や初期設定を除いた部分）のみの経過時間を調べたい場合にこの方法を用います。

```
  REAL*8 ELP1,ELP2                          ①
  ELP1 = MPI_WTIME()                        ②
  ////////////測定対象部分////////////
  ELP2 = MPI_WTIME()                        ③
  PRINT *,'MYRANK = ',MYRANK,
&         'ELAPSE = ',ELP2-ELP1             ④
```

図 4-8-8 (1)　×

図 4-8-8 (2)

```
  REAL*8 ELP1,ELP2
  CALL MPI_BARRIER(MPI_COMM_WORLD,IERR)     ⑤
  ELP1 = MPI_WTIME()
  ////////////測定対象部分////////////
  ELP2 = MPI_WTIME()
  PRINT *,'MYRANK = ',MYRANK,
&         'ELAPSE = ',ELP2-ELP1
```

図 4-8-8 (3)

図 4-8-8 (4)

```
  REAL*8 ELP1,ELP2
  CALL MPI_BARRIER(MPI_COMM_WORLD,IERR)     ⑥
  ELP1 = MPI_WTIME()
  ////////////測定対象部分////////////
  CALL MPI_BARRIER(MPI_COMM_WORLD,IERR)     ⑦
  ELP2 = MPI_WTIME()
  IF (MYRANK==0)
&   PRINT *,'ELAPSE = ',ELP2-ELP1
```

図 4-8-8 (5)

図 4-8-8 (6)

並列化したプログラムのパフォーマンスが悪い場合の調査方法

並列化したプログラムのパフォーマンスが悪い場合、タイマールーチンを使用して原因を調べます。例えば図 4-8-9 のプログラムで、①は（本当の意味の）並列化をしていない部分、②は（本当の意味の）並列化をした部分、③は並列化に伴って必要となった通信部分だとします。1, 2に示すように、各部分の前後に前述のタイマールーチン MPI_WTIME() を挿入して経過時間を測定し、それを3で EELP(1)～EELP(3) に累積します (①, ②, ③が複数箇所ある場合は、それぞれの箇所で測定を行います)。

①と③では、一番長くかかったプロセスの経過時間を測定したいので、4, 5に示すように、1 回目と 2 回目の MPI_WTIME() の前に MPI_BARRIER を挿入して同期を取ります。一方②では、各プロセスのロードバランスのばらつきを調べたいので、6に示すように、1 回目の MPI_WTIME() の前にだけ MPI_BARRRIER を挿入します。

そして他のジョブが流れていない占有使用の状態で、例えば 1, 2, 4 プロセスで測定を行います。なお、1 プロセスでの測定は、並列版ではなく単体版のプログラムを使用します (ただし単体版では MPI_WTIME() が使用できないので、コンパイラが提供している経過時間測定ルーチンを使用します)。

並列ジョブのパフォーマンスが悪い場合、4-1-2 節で説明したように、以下の 3 つの原因が考えられますので、測定結果からどのケースに該当するのかを調べ、その問題を（可能であれば）取り除きます。

- 図 4-8-10 (1) では、並列化率が低過ぎるため、並列化しても速くなりません。
- 図 4-8-10 (2) では、各プロセスのロードバランスが不均等なため、並列化しても速くなりません。
- 図 4-8-10 (3) では、通信に時間がかかっているため、並列化しても速くなりません。

```
   :
 REAL*8 ELP1,ELP2,EELP(3)
   :
 DO I=1,3
   EELP(I) = 0.0
 ENDDO
 DO ITIME=1,ITMAX  タイムステップ・ループ
   CALL MPI_BARRIER(MPI_COMM_WORLD,IERR) 4
   ELP1 = MPI_WTIME()                    1
   DO I=1,N
     :                  ①未並列化部分
   ENDDO
   CALL MPI_BARRIER(MPI_COMM_WORLD,IERR) 5
   ELP2 = MPI_WTIME()                    2
   EELP(1) = EELP(1)+ELP2-ELP1           3
   CALL MPI_BARRIER(MPI_COMM_WORLD,IERR) 6
   ELP1 = MPI_WTIME()                    1
   DO I=ISTA,IEND
     :                  ②並列化部分
   ENDDO
   ELP2 = MPI_WTIME()                    2
   EELP(2) = EELP(2)+ELP2-ELP1           3
   CALL MPI_BARRIER(MPI_COMM_WORLD,IERR) 4
   ELP1 = MPI_WTIME()                    1
   CALL MPI_BCAST(～) ③通信部分
   CALL MPI_BARRIER(MPI_COMM_WORLD,IERR) 5
   ELP2 = MPI_WTIME()                    2
   EELP(3) = EELP(3)+ELP2-ELP1           3
 ENDDO
 PRINT *,'MYRANK=',MYRANK,EELP
   :
```

図 4-8-9

図 4-8-10 (1)　並列化率が低いケース

図 4-8-10 (2)　ロードバランスが不均等なケース

図 4-8-10 (3)　通信が多いケース

第5章　並列化の応用例

第4章までで、プログラムを並列化するための技法についての説明は終わりました。本章では、データの動きが複雑なプログラムの並列化例、ややマニアックな並列化例を紹介します。

5-1　差分法徹底攻略

キング・オブ科学技術計算の称号を差分法に与えても異議をとなえる人はいないでしょう。実際、過去に筆者が並列化したプログラムのうち、約半数が差分法のプログラムでした。差分法で行うシフトについては、4-6-2 節で 1 次元差分法モドキを例に基本的な考え方を説明しました。本節では 2 次元差分法でシフトを行う際に必要となる各種のテクニックを徹底攻略します。

なお、シフトの通信方法には、4-6-2 節の図 4-6-3 (1)〜(3) に示す「通信方法 1」〜「通信方法 3」の 3 種類があり、どの方法が最も速いかはマシン環境によって異なります。本節では便宜的に「通信方法 1」を使用します。

また、3 次元差分法は 2 次元差分法での方法を単に拡張するだけなので説明は省略します。

4-6-2 節の 1 次元の差分法モドキのプログラム (図 4-6-2 (1)) を 2 次元に拡張したのが図 5-1-1 (1) です。まず①で配列 A に適当な値を設定し (図 5-1-1 (2) の○と●)、②で配列 A の上下左右の値（●）から配列 B の値（■）を求めます。

```
      PROGRAM MAIN
      IMPLICIT REAL*8(A-H,O-Z)
      PARAMETER (M=4,N=9)
      DIMENSION A(M,N),B(M,N)
      DO J = 1, N
        DO I = 1, M
          A(I,J) = I + 10.0*J                    ①
        ENDDO
      ENDDO
      DO J = 2, N-1
        DO I = 2, M-1
          B(I,J) = A(I-1,J) + A(I,J-1)           ②
     &           + A(I,J+1) + A(I+1,J)
        ENDDO
      ENDDO
      END
```

図 5-1-1 (1)

図 5-1-1 (2)

2 次元の差分法プログラムを並列化する場合、配列 A, B の分割には以下の 3 つの方法があり、通信量などの点からどれを選ぶかを決めます (4-5-8 節参照)。以下の各節では、それぞれの分割の具体的な方法について説明します。

2次元目でブロック分割
➡ 5 − 1 − 1 節で説明

図 5-1-2 (1)

1次元目でブロック分割
➡ 5 − 1 − 2 節で説明

図 5-1-2 (2)

1,2次元目でブロック分割
➡ 5 − 1 − 3 節で説明

図 5-1-2 (3)

5-1-1　2次元目でブロック分割した場合

本節では、配列A，Bを図5-1-2 (1)のように2次元目でブロック分割する場合のシフト方法を説明します。並列化したプログラムを図5-1-3 (1)に、データの動きを図5-1-3 (2)に示します(配列AとBのうち、各プロセスが実際に使用する部分を太線で示します)。図4-6-3 (1) (2)と図5-1-3 (1) (2)内の番号は対応しています。

図4-6-3 (1)ではシフトするデータが1つの要素でしたが、図5-1-3 (1)では1列全体をシフトするので、⑥〜⑨で送受信するデータの要素数は『M』になります。それ以外は図4-6-3 (1)と同じなので、これ以上の説明は省略します。

```
      PROGRAM MAIN
      IMPLICIT REAL*8(A-H,O-Z)
      INCLUDE 'mpif.h'
      PARAMETER (M=4,N=9)
      DIMENSION A(M,N),B(M,N)
      INTEGER ISTATUS(MPI_STATUS_SIZE)
      CALL MPI_INIT(IERR)
      CALL MPI_COMM_SIZE
     &    (MPI_COMM_WORLD,NPROCS,IERR)
      CALL MPI_COMM_RANK
     &    (MPI_COMM_WORLD,MYRANK,IERR)
      CALL PARA_RANGE
     &    (1,N,NPROCS,MYRANK,JSTA,JEND)
      JSTA2 = JSTA                                  ③
      JEND1 = JEND
      IF (MYRANK == 0        ) JSTA2 = 2
      IF (MYRANK == NPROCS-1) JEND1 = N-1
      IUP   = MYRANK + 1
      IDOWN = MYRANK - 1                            ④
      IF (MYRANK==NPROCS-1) IUP  =MPI_PROC_NULL
      IF (MYRANK==0)        IDOWN=MPI_PROC_NULL
      DO J = JSTA, JEND
        DO I = 1, M
          A(I,J) = I + 10.0*J                       ⑤
        ENDDO
      ENDDO
      CALL MPI_ISEND(A(1,JEND)  ,M,
     &       MPI_REAL8,IUP  ,1,                     ⑥
     &       MPI_COMM_WORLD,ISEND1,IERR)
      CALL MPI_ISEND(A(1,JSTA)  ,M,
     &       MPI_REAL8,IDOWN,1,                     ⑦
     &       MPI_COMM_WORLD,ISEND2,IERR)
      CALL MPI_IRECV(A(1,JSTA-1),M,
     &       MPI_REAL8,IDOWN,1,                     ⑧
     &       MPI_COMM_WORLD,IRECV1,IERR)
      CALL MPI_IRECV(A(1,JEND+1),M,
     &       MPI_REAL8,IUP  ,1,                     ⑨
     &       MPI_COMM_WORLD,IRECV2,IERR)
      CALL MPI_WAIT(ISEND1,ISTATUS,IERR)
      CALL MPI_WAIT(ISEND2,ISTATUS,IERR)            ⑩
      CALL MPI_WAIT(IRECV1,ISTATUS,IERR)
      CALL MPI_WAIT(IRECV2,ISTATUS,IERR)
      DO J = JSTA2, JEND1
        DO I = 2, M-1
          B(I,J) = A(I-1,J) + A(I,J-1)              ⑪
     &           + A(I,J+1) + A(I+1,J)
        ENDDO
      ENDDO
      CALL MPI_FINALIZE(IERR)
      END
```

図5-1-3 (1)

図5-1-3 (2)

5-1-2　1 次元目でブロック分割した場合

配列 A，B を図 5-1-2 (2) のように 1 次元目でブロック分割する場合のシフト方法を説明します (紙面の関係で配列 A，B の大きさは前節と異なり 9 × 4 とします)。並列化したプログラムを図 5-1-4 (1) に、データの動きを図 5-1-4 (2) に示します。図 5-1-4 (1) (2) 内の番号はそれぞれ対応しています。

配列 A，B を 1 次元目でブロック分割した場合、通信するデータはプロセス間の境界の 1 行ですが、1 行内の各要素はメモリー上でとびとびになっているので、簡単には通信できません。そこで派生データ型を使用して通信を行います。まず⑫で、配列 A（と同じ大きさの配列）の任意の行の先頭から始まる長さ N の 1 行を表す派生データ型 ITYPE を、3-5-3-2 節で説明した自作のサブルーチン PARA_TYPE_BLOCK2A を使用して作成します (MPI-2 が使用できるマシン環境の場合は、MPI-2 で提供されている MPI_TYPE_CREATE_SUBARRAY (3-5-3-1 節参照) を使用できますが、派生データ型の起点が異なるので注意して下さい)。そして⑥〜⑨の下線部で、派生データ型 ITYPE を指定して実際の送受信を行います。派生データ型を使用する以外の部分は前節の例と同じなので、これ以上の説明は省略します。

```
      PROGRAM MAIN
      IMPLICIT REAL*8(A-H,O-Z)
      INCLUDE 'mpif.h'
      PARAMETER (M=9,N=4)
      DIMENSION A(M,N),B(M,N)
      INTEGER ISTATUS(MPI_STATUS_SIZE)
      CALL MPI_INIT(IERR)
      CALL MPI_COMM_SIZE
     &    (MPI_COMM_WORLD,NPROCS,IERR)
      CALL MPI_COMM_RANK
     &    (MPI_COMM_WORLD,MYRANK,IERR)
      CALL PARA_RANGE
     &    (1,M,NPROCS,MYRANK,ISTA,IEND)
      ISTA2 = ISTA
      IEND1 = IEND                                  ③
      IF (MYRANK == 0        ) ISTA2 = 2
      IF (MYRANK == NPROCS-1) IEND1 = M-1
      IUP   = MYRANK + 1
      IDOWN = MYRANK - 1
      IF (MYRANK==NPROCS-1) IUP  =MPI_PROC_NULL④
      IF (MYRANK==0)        IDOWN=MPI_PROC_NULL
      CALL PARA_TYPE_BLOCK2A                        ⑫
     &          (1,M,1,N,MPI_REAL8,ITYPE)           ⑫
      DO J = 1, N
        DO I = ISTA, IEND
          A(I,J) = I + 10.0*J                       ⑤
        ENDDO
      ENDDO
      CALL MPI_ISEND(A(IEND,1),1,ITYPE,             ⑥
     &     IUP  ,1,MPI_COMM_WORLD,ISEND1,IERR)      ⑥
      CALL MPI_ISEND(A(ISTA,1),1,ITYPE,             ⑦
     &     IDOWN,1,MPI_COMM_WORLD,ISEND2,IERR)      ⑦
      CALL MPI_IRECV(A(ISTA-1,1),1,ITYPE,           ⑧
     &     IDOWN,1,MPI_COMM_WORLD,IRECV1,IERR)      ⑧
      CALL MPI_IRECV(A(IEND+1,1),1,ITYPE,           ⑨
     &     IUP  ,1,MPI_COMM_WORLD,IRECV2,IERR)      ⑨
      CALL MPI_WAIT(ISEND1,ISTATUS,IERR)
      CALL MPI_WAIT(ISEND2,ISTATUS,IERR)
      CALL MPI_WAIT(IRECV1,ISTATUS,IERR)            ⑩
      CALL MPI_WAIT(IRECV2,ISTATUS,IERR)
      DO J = 2, N-1
        DO I = ISTA2, IEND1
          B(I,J) = A(I-1,J) + A(I,J-1)
     &           + A(I,J+1) + A(I+1,J)              ⑪
        ENDDO
      ENDDO
      CALL MPI_FINALIZE(IERR)
      END
```

図 5-1-4 (1)

図 5-1-4 (2)

5-1-3　1，2次元目でブロック分割した場合

配列 A，B を図 5-1-2 (3) のように1次元目と2次元目の両方でブロック分割する場合 (4-5-8 節参照) のシフト方法を説明します (紙面の関係で配列 A，B の大きさは 12 × 9 とします)。

まず配列 A のデータの動きを図 5-1-5 (5) で概観します。例えばランク 4 のプロセスは、中央の図に示すように、上下左右の隣接する各プロセスとの間でシフトを行います。このうち左右のシフトは配列を 2 次元目でブロック分割した 5-1-1 節のシフトと同じ動作に、上下のシフトは配列を 1 次元目でブロック分割した 5-1-2 節のシフトと同じ動作になります。

並列化したプログラムを図 5-1-5 (4) に示し、以下で説明します。図 5-1-5 (4) (5) 内の番号はそれぞれ対応しています。なお、図 5-1-5 (4) で⑥と[6]のように異なる記号の同じ数字がある場合、⑥が左右方向のシフト、[6]が上下方向のシフトに関連する部分を示します。

- 図 5-1-5 (4) の⑬で、配列 A，B を分割したときの 1 次元方向と 2 次元方向のプロセス数を IPROCS，JPROCS に設定します。この例では図 5-1-5 (5) に示すように IPROCS，JPROCS ともに 3 で、全部で 9 プロセスを使用します。⑭で IPROCS × JPROCS が NPROCS と等しいかどうか念のためチェックします。
- ⑮で図 5-1-5 (1) に示す配列 ITABLE（以下プロセス格子テーブル）を作成します。これは各プロセスが自分の上下左右の隣のプロセスのランク値を知るために使用します。本例は非循環シフトなので、プロセスの存在しない×の部分には MPI_PROC_NULL (4-6-2 節参照) を入れます。また自プロセスのプロセス格子テーブル内の 1 次元目と 2 次元目の座標を MYRANKI と MYRANKJ に設定します (例えば自分がランク 1 の場合、→ と ↓ に示すように MYRANKI = 0，MYRANKJ = 1 となります)。なお、以下ではテーブル内のプロセス値の順番を横方向に 0，1，2，... としましたが、縦方向に 0，1，2，... としてもかまいません。

図 5-1-5 (1)

- ③と[3]で、自プロセスの配列 A，B の 1 次元目と 2 次元目の担当範囲を求めます。前節までの例では 1 つの次元のみで分割していたので、サブルーチン PARA_RANGE の引数を図 5-1-5 (2) の「今までの分割」のように指定しましたが、今回は 2 つの次元でそれぞれ分割するため、図 5-1-5 (2) の「1 次元目の分割」と「2 次元目の分割」のように指定します。次に④と[4]で、図 5-1-5 (1) のプロセス格子テーブルを使用して、自プロセスの上下左右のプロセスのランク値を取得します。例えばランク 4 のプロセスの場合、図 5-1-5 (3) のようになります。
⑯で、各プロセスが通信する境界の 1 列の要素数 ILEN を求めます (この値は⑥〜⑨で使用します)。また⑫で、各プロセスが通信する境界の 1 行を表す派生データ型 ITYPE を作成します (詳細は 5-1-2 節参照)。

	PARA_RANGEで指定する引数		PARA_RANGEで戻る引数	
	プロセス数	自分のランク	担当部分下限	担当部分上限
今までの分割	NPROCS	MYRANK	ISTA(JSTA)	IEND(JEND)
1次元目の分割	IPROCS	MYRANKI	ISTA	IEND
2次元目の分割	JPROCS	MYRANKJ	JSTA	JEND

図 5-1-5 (2)

図 5-1-5 (3)

- 図 5-1-1 (1) ①のループを図 5-1-5 (4) ⑤で I と J の両方向で並列化します。次に配列 A の境界のデータのシフトを行います。図 5-1-5 (5) の左右方向のシフトは⑥〜⑩で行い、上下方向のシフトは[6]〜[10]で行います。このとき図 5-1-5 (3) の各変数を送信元または宛先プロセスのランク値として使用します。シフト終了後、図 5-1-1 (1) ②のループを、図 5-1-5 (4) ⑪で I と J の両方向で並列化します。

```
      PROGRAM MAIN
      IMPLICIT REAL*8(A-H,O-Z)
      INCLUDE 'mpif.h'
      PARAMETER (M=12,N=9)
      DIMENSION A(M,N),B(M,N)
      INTEGER ISTATUS(MPI_STATUS_SIZE)
      PARAMETER(IPROCS=3,JPROCS=3)                    ⑬
      INTEGER ITABLE(-1:IPROCS,-1:JPROCS)
      CALL MPI_INIT(IERR)
      CALL MPI_COMM_SIZE
     &    (MPI_COMM_WORLD,NPROCS,IERR)
      CALL MPI_COMM_RANK
     &    (MPI_COMM_WORLD,MYRANK,IERR)
      IF (NPROCS /= IPROCS*JPROCS) THEN
        IF (MYRANK==0) PRINT *,'===ERROR==='
        CALL MPI_FINALIZE(IERR)                       ⑭
        STOP
      ENDIF
      DO J = -1, JPROCS
        DO I = -1, IPROCS
          ITABLE(I,J) = MPI_PROC_NULL
        ENDDO
      ENDDO
      IRANK = 0
      DO I = 0, IPROCS-1
        DO J = 0, JPROCS-1
          ITABLE(I,J) = IRANK                         ⑮
          IF (MYRANK == IRANK) THEN
            MYRANKI = I
            MYRANKJ = J
          ENDIF
          IRANK = IRANK + 1
        ENDDO
      ENDDO
      CALL PARA_RANGE
     &    (1,N,JPROCS,MYRANKJ,JSTA,JEND)
      JSTA2 = JSTA
      JEND1 = JEND                                    ③
      IF (MYRANKJ == 0        ) JSTA2 = 2
      IF (MYRANKJ == JPROCS-1) JEND1 = N-1
      CALL PARA_RANGE
     &    (1,M,IPROCS,MYRANKI,ISTA,IEND)
      ISTA2 = ISTA
      IEND1 = IEND                                    [3]
      IF (MYRANKI == 0        ) ISTA2 = 2
      IF (MYRANKI == IPROCS-1) IEND1 = M-1
      JUP   = ITABLE(MYRANKI  ,MYRANKJ+1)             ④
      JDOWN = ITABLE(MYRANKI  ,MYRANKJ-1)             ④
      IUP   = ITABLE(MYRANKI+1,MYRANKJ  )             [4]
      IDOWN = ITABLE(MYRANKI-1,MYRANKJ  )             [4]
      ILEN = IEND-ISTA+1                              ⑯
      CALL PARA_TYPE_BLOCK2A                          ⑫
     &   (1,M,1,JEND-JSTA+1,MPI_REAL8,ITYPE)          ⑫
      DO J = JSTA, JEND
        DO I = ISTA, IEND
          A(I,J) = I + 10.0*J                         ⑤
        ENDDO
      ENDDO
      CALL MPI_ISEND(A(ISTA,JEND),ILEN,
     &       MPI_REAL8,JUP  ,1,                       ⑥
     &       MPI_COMM_WORLD,ISEND1,IERR)
      CALL MPI_ISEND(A(ISTA,JSTA),ILEN,
     &       MPI_REAL8,JDOWN,1,                       ⑦
     &       MPI_COMM_WORLD,ISEND2,IERR)
      CALL MPI_ISEND(A(IEND,JSTA),1,
     &       ITYPE,IUP  ,1,                           [6]
     &       MPI_COMM_WORLD,JSEND1,IERR)
      CALL MPI_ISEND(A(ISTA,JSTA),1,
     &       ITYPE,IDOWN,1,                           [7]
     &       MPI_COMM_WORLD,JSEND2,IERR)
      CALL MPI_IRECV(A(ISTA,JSTA-1),ILEN,
     &       MPI_REAL8,JDOWN,1,                       ⑧
     &       MPI_COMM_WORLD,IRECV1,IERR)
      CALL MPI_IRECV(A(ISTA,JEND+1),ILEN,
     &       MPI_REAL8,JUP  ,1,                       ⑨
     &       MPI_COMM_WORLD,IRECV2,IERR)
      CALL MPI_IRECV(A(ISTA-1,JSTA),1,
     &       ITYPE,IDOWN,1,                           [8]
     &       MPI_COMM_WORLD,JRECV1,IERR)
      CALL MPI_IRECV(A(IEND+1,JSTA),1,
     &       ITYPE,IUP  ,1,                           [9]
     &       MPI_COMM_WORLD,JRECV2,IERR)
      CALL MPI_WAIT(ISEND1,ISTATUS,IERR)
      CALL MPI_WAIT(ISEND2,ISTATUS,IERR)
      CALL MPI_WAIT(JSEND1,ISTATUS,IERR)
      CALL MPI_WAIT(JSEND2,ISTATUS,IERR)              ⑩
      CALL MPI_WAIT(IRECV1,ISTATUS,IERR)              [10]
      CALL MPI_WAIT(IRECV2,ISTATUS,IERR)
      CALL MPI_WAIT(JRECV1,ISTATUS,IERR)
      CALL MPI_WAIT(JRECV2,ISTATUS,IERR)
      DO J = JSTA2, JEND1
        DO I = ISTA2, IEND1
          B(I,J) = A(I-1,J) + A(I,J-1)
     &           + A(I,J+1) + A(I+1,J)                ⑪
        ENDDO
      ENDDO
      CALL MPI_FINALIZE(IERR)
      END
```

図 5-1-5 (4)

図 5-1-5 (5)

5-1-4　1，2 次元目でブロック分割した場合（斜めの要素を参照）

差分法のプログラムによっては、図 5-1-6 (1) の下線部が追加されていることがあります。この場合、まず①で配列 A に適当な値を設定し (図 5-1-6 (2) の○，●，○)、②で配列 A の上下左右（●）と斜め隣の値（○）から配列 B の値（■）を求めます (紙面の関係で、図 5-1-6 (2) の配列の大きさは 4 × 9 になっています)。

```
      PROGRAM MAIN
      IMPLICIT REAL*8(A-H,O-Z)
      PARAMETER (M=12,N=9)
      DIMENSION A(M,N),B(M,N)
      DO J = 1, N
        DO I = 1, M
          A(I,J) = I + 10.0*J                    ①
        ENDDO
      ENDDO
      DO J = 2, N-1
        DO I = 2, M-1
          B(I,J) = A(I-1,J  ) + A(I  ,J-1)
     &           + A(I  ,J+1) + A(I+1,J  )
     &           + A(I-1,J-1) + A(I+1,J-1)          ②
     &           + A(I-1,J+1) + A(I+1,J+1)
        ENDDO
      ENDDO
      END
```

図 5-1-6 (1)

図 5-1-6 (2)

このプログラムを 1 次元目または 2 次元目のどちらか一方でブロック分割して並列化する場合には、5-1-1 節と 5-1-2 節の方法をそのまま使用することができます。しかし 1 次元目と 2 次元目の両方でブロック分割する場合、5-1-3 節の方法をそのまま使用することはできません。

その理由を説明します。並列化した場合の配列 A のデータの動きは図 5-1-6 (4) のようになりますが、中央の図に示すように、例えばランク 4 のプロセスは図 5-1-6 (1) ②の下線部の計算を行うため、1，3，5，7 の各要素の他に、四隅の 0，2，6，8 の要素（斜め隣の各プロセスが持っている）が必要になります。ところが 5-1-3 節の方法では、図 5-1-5 (5) に示すようにこれらの要素を通信していないので、この方法をそのまま使用することはできません。

四隅の要素は斜め隣のプロセスから直接送信することもできますが、ここでは通信回数を減らすため直接通信しない方法を説明します。

まずデータの動きを図 5-1-6 (4) で概観します。最初に⑥～⑨で境界の 1 列を横方向に通信します。これを矢印（実線）で示します。この通信は図 5-1-5 (5) の通信と同じです。

横方向の通信が全て完了した後、6～9で境界の 1 行を縦方向に通信します。これを矢印（点線）で示します。このとき図 5-1-5 (5) の場合よりも左右に 1 要素分余分に通信します。例えばランク 1 のプロセスは『01112』をランク 4 のプロセスに送信します（★の部分の通信)。

これによって、四隅の要素を含む全ての要素の通信が完了します。例えばランク 4 のプロセスの四隅には、0，2，6，8 の要素が入っていますが、そのうち左上の要素 0 は、ランク 0 から直接ランク 4 に送られたのではなく、ランク 0 からランク 1 を経由してランク 4 に送られたことに注意して下さい。このような動作を行うためには、先に横方向の通信を行って、それが完了した後で縦方向の通信を (左右に 1 要素分余分に) 行う必要があります。

次にプログラム例を図 5-1-6 (3) に示します。図 5-1-6 (3) (4) 内の番号はそれぞれ対応しています。まず⑫で、縦方向に通信する 1 行 (左右に余分な要素を 1 要素分持つ) の下限と上限を JJSTA，JJEND とし (図 5-1-6 (4) の↓参照)、この 1 行の派生データ型 ITYPE を、3-5-3-2 節で説明した自作のサブルーチン PARA_TYPE_BLOCK2A を使用して作成します (MPI-2 が使用できるマシン環境の場合は、MPI-2 で提供されている MPI_TYPE_CREATE_SUBARRAY (3-5-3-1 節参照) を使用できますが、派生データ型の起点が異なるので注意して下さい)。そして図 5-1-6 (1) ⑤の計算が終了した後、まず⑥～⑩で境界の 1 列を横方向に通信し、それが完了した後、6～10で境界の 1 行を派生データ型 ITYPE を使用して縦方向に通信します。

```
      PROGRAM MAIN
      IMPLICIT REAL*8(A-H,O-Z)
      INCLUDE 'mpif.h'
      PARAMETER (M=12,N=9)
      DIMENSION A(M,N),B(M,N)
      INTEGER ISTATUS(MPI_STATUS_SIZE)
      PARAMETER(IPROCS=3,JPROCS=3)                    ⑬
      INTEGER ITABLE(-1:IPROCS,-1:JPROCS)
      CALL MPI_INIT(IERR)
      CALL MPI_COMM_SIZE
     &    (MPI_COMM_WORLD,NPROCS,IERR)
      CALL MPI_COMM_RANK
     &    (MPI_COMM_WORLD,MYRANK,IERR)
      IF (NPROCS /= IPROCS*JPROCS) THEN
        IF (MYRANK==0) PRINT *,'===ERROR==='
        CALL MPI_FINALIZE(IERR)                       ⑭
        STOP
      ENDIF
      DO J = -1, JPROCS
        DO I = -1, IPROCS
          ITABLE(I,J) = MPI_PROC_NULL
        ENDDO
      ENDDO
      IRANK = 0
      DO I = 0, IPROCS-1
        DO J = 0, JPROCS-1
          ITABLE(I,J) = IRANK                         ⑮
          IF (MYRANK == IRANK) THEN
            MYRANKI = I
            MYRANKJ = J
          ENDIF
          IRANK = IRANK + 1
        ENDDO
      ENDDO
      CALL PARA_RANGE
     &    (1,N,JPROCS,MYRANKJ,JSTA,JEND)
      JSTA2 = JSTA
      JEND1 = JEND                                    ③
      IF (MYRANKJ == 0        ) JSTA2 = 2
      IF (MYRANKJ == JPROCS-1) JEND1 = N-1
      CALL PARA_RANGE
     &    (1,M,IPROCS,MYRANKI,ISTA,IEND)
      ISTA2 = ISTA
      IEND1 = IEND                                    3
      IF (MYRANKI == 0        ) ISTA2 = 2
      IF (MYRANKI == IPROCS-1) IEND1 = M-1
      JUP   = ITABLE(MYRANKI  ,MYRANKJ+1)             ④
      JDOWN = ITABLE(MYRANKI  ,MYRANKJ-1)             ④
      IUP   = ITABLE(MYRANKI+1,MYRANKJ  )             4
      IDOWN = ITABLE(MYRANKI-1,MYRANKJ  )             4
      ILEN = IEND-ISTA+1                              ⑯
      JJSTA = MAX(1,JSTA-1)
      JJEND = MIN(N,JEND+1)
      CALL PARA_TYPE_BLOCK2A                          ⑫
     &   (1,M,1,JJEND-JJSTA+1,MPI_REAL8,ITYPE)
      DO J = JSTA, JEND
        DO I = ISTA, IEND
          A(I,J) = I + 10.0*J                         ⑤
        ENDDO
      ENDDO
      CALL MPI_ISEND(A(ISTA,JEND),ILEN,
     &       MPI_REAL8,JUP  ,1,                       ⑥
     &       MPI_COMM_WORLD,ISEND1,IERR)
      CALL MPI_ISEND(A(ISTA,JSTA),ILEN,
     &       MPI_REAL8,JDOWN,1,                       ⑦
     &       MPI_COMM_WORLD,ISEND2,IERR)
      CALL MPI_IRECV(A(ISTA,JSTA-1),ILEN,
     &       MPI_REAL8,JDOWN,1,                       ⑧
     &       MPI_COMM_WORLD,IRECV1,IERR)
      CALL MPI_IRECV(A(ISTA,JEND+1),ILEN,
     &       MPI_REAL8,JUP  ,1,                       ⑨
     &       MPI_COMM_WORLD,IRECV2,IERR)
      CALL MPI_WAIT(ISEND1,ISTATUS,IERR)
      CALL MPI_WAIT(ISEND2,ISTATUS,IERR)
      CALL MPI_WAIT(IRECV1,ISTATUS,IERR)              ⑩
      CALL MPI_WAIT(IRECV2,ISTATUS,IERR)
      CALL MPI_ISEND(A(IEND,JJSTA),1,
     &       ITYPE,IUP  ,1,                           6
     &       MPI_COMM_WORLD,JSEND1,IERR)
      CALL MPI_ISEND(A(ISTA,JJSTA),1,
     &       ITYPE,IDOWN,1,                           7
     &       MPI_COMM_WORLD,JSEND2,IERR)
      CALL MPI_IRECV(A(ISTA-1,JJSTA),1,
     &       ITYPE,IDOWN,1,                           8
     &       MPI_COMM_WORLD,JRECV1,IERR)
      CALL MPI_IRECV(A(IEND+1,JJSTA),1,
     &       ITYPE,IUP  ,1,                           9
     &       MPI_COMM_WORLD,JRECV2,IERR)
      CALL MPI_WAIT(JSEND1,ISTATUS,IERR)
      CALL MPI_WAIT(JSEND2,ISTATUS,IERR)
      CALL MPI_WAIT(JRECV1,ISTATUS,IERR)              10
      CALL MPI_WAIT(JRECV2,ISTATUS,IERR)
      DO J = JSTA2, JEND1
        DO I = ISTA2, IEND1
          B(I,J) = A(I-1,J  ) + A(I  ,J-1)
     &           + A(I  ,J+1) + A(I+1,J  )
     &           + A(I-1,J-1) + A(I+1,J-1)              ⑪
     &           + A(I-1,J+1) + A(I+1,J+1)
        ENDDO
      ENDDO
      CALL MPI_FINALIZE(IERR)
      END
```

図 5-1-6 (3)

図 5-1-6 (4)

5-1-5　配列を縮小する場合

本節では、4-5-7 節で説明した動的割振りを用いて配列を縮小する方法について説明します。前節までのいずれの分割でも縮小方法は同じなので、配列を 2 次元方向で分割した図 5-1-3 (1) (2) を例とし、縮小後のプログラムを図 5-1-7 (1) に、縮小した配列 A, B を図 5-1-7 (2) の実線部に示します。

図 5-1-7 (1) では、まず⑰で縮小する配列 A, B の動的割振りを宣言します。そして③で 2 次元方向の自分の担当範囲を求めた後、⑱で配列 A を確保します。図 5-1-7 (2) の上段のランク 1 に示すように、JSTA の前と JEND の後にシフト用の 1 列が必要になるので、J 方向の大きさは (JSTA-1)〜(JEND+1) となります。ただし、この例では端のプロセスであるランク 0 では JSTA-1 が、ランク 2 では JEND+1 が不要なので、組込関数 MIN, MAX を使用して調整します。

次に⑲で配列 B を確保します。⑪の DO ループの反復から分かるように、1 次元方向の大きさは 2〜M-1、2 次元方向の大きさは JSTA2〜JEND1 となります。

なお、派生データ型 (3-5 節参照) を使用して通信した配列を縮小する場合、4-5-7-2 節の「派生データ型を使用した通信の修正」で説明した修正が必要になります。

```
      :
      INCLUDE 'mpif.h'
      PARAMETER (M=4,N=9)
      REAL*8,ALLOCATABLE::A(:,:),B(:,:)          ⑰
      CALL PARA_RANGE
     &    (1,N,NPROCS,MYRANK,JSTA,JEND)
      JSTA2 = JSTA
      JEND1 = JEND                               ③
      IF (MYRANK == 0        ) JSTA2 = 2
      IF (MYRANK == NPROCS-1) JEND1 = N-1
      ALLOCATE(A(M,MAX(JSTA-1,1):MIN(JEND+1,N)))⑱
      ALLOCATE(B(2:M-1,JSTA2:JEND1))             ⑲
      :
```

```
DO J = JSTA, JEND
  DO I = 1, M
    A(I,J) = I + 10.0*J                          ⑤
  ENDDO
ENDDO
  :
DO J = JSTA2, JEND1
  DO I = 2, M-1
    B(I,J) = A(I-1,J) + A(I,J-1)
&          + A(I,J+1) + A(I+1,J)                 ⑪
  ENDDO
ENDDO
CALL MPI_FINALIZE(IERR)
  :
```

図 5-1-7 (1)

図 5-1-7 (2)

5-1-6　その他の考慮点

例えば図 5-1-8 (1) の①，②の DO ループを J で並列化する場合、配列 B，D，F に対してシフトを行なう必要があります。差分法のプログラムを並列化する場合、このように通常多くの配列に対し、多くの箇所でシフトを行う必要がありますが、通信が必要になるたびに通信ルーチンを毎回記述するとプログラムがゴチャゴチャになります。このような場合、図 5-1-8 (2) のような方法で並列化するとプログラムがすっきりします。

- 並列化に関係のある汎用的な変数は全て③の MODULE 文内に記述します (詳細は 4-8-1 節を参照して下さい)。MODULE 文は少し使い方にクセがあるので、4-5-7 節を参照して下さい。
- 実際にシフトを行う部分を⑥のようにサブルーチン化します。
- ①と②の DO ループの直前の④と⑤で、シフトを行うサブルーチン SHIFT をコールします。

```
      PROGRAM MAIN
      PARAMETER(M=4,N=9)
      REAL*8 A(M,N),B(M,N),C(M,N),D(M,N)
     &       E(M,N),F(M,N)
        :
      DO J = 2, N-1
        DO I = 2, M-1
          A(I,J) = B(I-1,J) + B(I,J-1)
     &           + B(I,J+1) + B(I+1,J)
          C(I,J) = D(I-1,J) + D(I,J-1)                ①
     &           + D(I,J+1) + D(I+1,J)
        ENDDO
      ENDDO
        :
      DO J = 2, N-1
        DO I = 2, M-1
          E(I,J) = F(I-1,J) + F(I,J-1)
     &           + F(I,J+1) + F(I+1,J)                ②
        ENDDO
      ENDDO
        :
```

図 5-1-8 (1)

```
      MODULE PARA
      INCLUDE 'mpif.h'
      INTEGER NPROCS,MYRANK,
     &        JSTA,JEND,JSTA2,JEND1,IUP,IDOWN         ③
      INTEGER ISTATUS(MPI_STATUS_SIZE)
      END
      PROGRAM MAIN
      USE PARA
      PARAMETER(M=4,N=9)
      REAL*8 A(M,N),B(M,N),C(M,N),D(M,N)
     &       E(M,N),F(M,N)
        :
      CALL SHIFT(B,M,N)                                ④
      CALL SHIFT(D,M,N)                                ④
      DO J = JSTA2, JEND1
        DO I = 2, M-1
          A(I,J) = B(I-1,J) + B(I,J-1)
     &           + B(I,J+1) + B(I+1,J)
          C(I,J) = D(I-1,J) + D(I,J-1)                ①
     &           + D(I,J+1) + D(I+1,J)
        ENDDO
      ENDDO
        :
      CALL SHIFT(F,M,N)                                ⑤
      DO J = JSTA2, JEND1
        DO I = 2, M-1
          E(I,J) = F(I-1,J) + F(I,J-1)
     &           + F(I,J+1) + F(I+1,J)                ②
        ENDDO
      ENDDO
        :
      SUBROUTINE SHIFT(X,M,N)
      USE PARA
      REAL*8 X(M,N)
      CALL MPI_ISEND(X(1,JEND)  ,M,
     &       MPI_REAL8,IUP  ,1,
     &       MPI_COMM_WORLD,ISEND1,IERR)
      CALL MPI_ISEND(X(1,JSTA)  ,M,
     &       MPI_REAL8,IDOWN,1,
     &       MPI_COMM_WORLD,ISEND2,IERR)
      CALL MPI_IRECV(X(1,JSTA-1),M,
     &       MPI_REAL8,IDOWN,1,                        ⑥
     &       MPI_COMM_WORLD,IRECV1,IERR)
      CALL MPI_IRECV(X(1,JEND+1),M,
     &       MPI_REAL8,IUP  ,1,
     &       MPI_COMM_WORLD,IRECV2,IERR)
      CALL MPI_WAIT(ISEND1,ISTATUS,IERR)
      CALL MPI_WAIT(ISEND2,ISTATUS,IERR)
      CALL MPI_WAIT(IRECV1,ISTATUS,IERR)
      CALL MPI_WAIT(IRECV2,ISTATUS,IERR)
      END
```

図 5-1-8 (2)

5-2　有限要素法

5-2-1　有限要素法の並列化

有限要素法は、要素の形状が不規則なため通信がしにくく、差分法と比べて一般に並列化が難しくなります。まず単体版の有限要素法のプログラムをそのまま並列化する場合を想定します。

陰解法のプログラムの場合、不規則対称疎行列の連立一次方程式を解く部分がホットスポットになります。これを直接法で解くための、数値計算サブルーチンの並列版 (第 6 章参照) が提供されているかどうかは未調査です。

一方陽解法のプログラムの場合、本節に述べるような方法で（一応）並列化することができます。

なお、市販の有限要素法プログラムの並列版では、領域分割法という方法で並列化するのが主流になっているようです。領域分割法では、例えば 3 ノードで計算する場合、計算領域を 3 つの完全に独立した領域として解き、領域の継ぎ目の部分を調整するという方法をとります。従って単体版のプログラムをそのまま並列化できるわけではなく、計算のアルゴリズム自体を並列用に変更する必要があります。領域分割法については参考文献 [16], [17] などを参照して下さい。

オリジナル版プログラム

以下では有限要素法の陽解法モドキのプログラムである図 5-2-2 で説明します。配列 Y は要素、配列 S は節点の配列です。計算領域は図 5-2-1 (1) のようになっており、□が要素番号、○が節点番号を示します。なお、本例では説明を簡単にするために四角形要素を使用し、計算領域の形状も単純ですが、三角形要素や領域の形状が複雑な場合でも同じ方法が適用できます。

以下に図 5-2-2 のプログラムの動作を示します。

- [1] で配列 Y, S に初期値を設定します。
- [2] では図 5-2-1 (1) の矢印に示すように、要素の値を使用してその要素の四隅の節点の値を更新します。このとき、要素番号と周囲の 4 節点の節点番号の対応表である配列 INDEX (図 5-2-1 (5)) を使用します。
- [3] では図 5-2-1 (2) の ↓ に示すように、節点の値を更新します。
- [4] では図 5-2-1 (3) の矢印に示すように、要素の値を、その要素の四隅の節点の値を使用して更新します。
- [5] では図 5-2-1 (4) の ↓ に示すように、要素の値を更新します。

一般に有限要素法（陽解法）のプログラムでは、このように 3 つのタイプのループ (要素のループ、節点のループ、要素 ↔ 節点のループ) が現れるのが特徴です。

図 5-2-1 (1)

図 5-2-1 (2)

図 5-2-1 (3)

図 5-2-1 (4)

INDEX

1	2	3	4	5	6	7	8	9	10	11	12
①	⑧	②	⑨	③	⑩	④	⑪	⑤	⑫	⑥	⑬
②	⑨	③	⑩	④	⑪	⑤	⑫	⑥	⑬	⑦	⑭
⑨	⑯	⑩	⑰	⑪	⑱	⑫	⑲	⑬	⑳	⑭	㉑
⑧	⑮	⑨	⑯	⑩	⑰	⑪	⑱	⑫	⑲	⑬	⑳

図 5-2-1 (5)

```
    :
  PARAMETER(IYMAX=12,ISMAX=21)
  REAL*8 Y(IYMAX),S(ISMAX)
  INTEGER INDEX(4,IYMAX)
    :
  DO IY=1,IYMAX
    Y(IY) = IY*10.0
  ENDDO
  DO IS=1,ISMAX                                    [1]
    S(IS) = IS*100.0
  ENDDO
  DO ITIME=1,10
    DO IY=1,IYMAX     要素から4節点を更新
      DO J=1,4
        S(INDEX(J,IY))=S(INDEX(J,IY))+Y(IY)        [2]
      ENDDO
    ENDDO
    DO IS=1,ISMAX     節点を更新
      S(IS) = S(IS)*0.25                           [3]
    ENDDO
    DO IY=1,IYMAX     4節点から要素を更新
      DO J=1,4
        Y(IY) = Y(IY) + S(INDEX(J,IY))             [4]
      ENDDO
    ENDDO
    DO IY=1,IYMAX     要素を更新
      Y(IY) = Y(IY)*0.25                           [5]
    ENDDO
  ENDDO
  PRINT *,'結果',S,Y
    :
```

図 5-2-2

5-2-2　並列化の方法 1

まず、4-6-6 節で説明した『重ね合せ』を用いて簡単に（手抜きで）並列化する方法について説明します。並列化したプログラムを図 5-2-5 に示します。

- [1] で、要素数 IYMAX をブロック分割し、各プロセスの担当する要素の範囲 IYSTA，IYEND を求めます (図 5-2-3 (1) 参照)。同様に節点数 ISMAX をブロック分割し、各プロセスの担当する節点の範囲 ISSTA，ISEND を求めます (図 5-2-3 (2) 参照)。要素の分割と節点の分割には何の関係もないことに注意して下さい。

図 5-2-3 (1)　要素の分割

図 5-2-3 (2)　節点の分割

- [2] で配列 Y，S をゼロクリアしておきます（この目的については後述します）。次に [3] を並列化します。
- [5] (図 5-2-2 の [2] の部分) を並列化します。配列 S には現在までの計算結果の累計が入っています。ここでは配列 S を直接更新するのではなく、一次配列 SS に現タイムステップでの増分を求め、それを配列 S に加算するという方法をとります。

  まず [4] で一次配列 SS を初期化し、次に [5] でこのタイムステップでの増分を求めます。[5] が終了すると図 5-2-4 (1) のようになり、プロセス間の境界にある節点には、完全な増分値ではなく各プロセスが計算した部分和が求まります。例えば③の場合、ランク 0 のプロセスには部分和〓が、ランク 1 のプロセスには部分和〓が求まります。〓と〓の値は合計して完全な増分値③にしてから配列 S に加算する必要があります。しかしどの節点に部分和が入っているのか（つまりどの節点がプロセス間の境界にあるのか）を調べるのが面倒なので、ここでは手抜きの方法として、[6] で配列 SS のすべての要素を重ね合せて結果を全プロセスの一時配列 SSS に求め (図 5-2-4 (2))、それを [7] で元の配列 S に加算するようにしました。なお [4] で配列 SS をゼロクリアしたのは、[6] で配列 SS について重ね合せを行うためです。
- [8] (図 5-2-2 の [3] の部分) を並列化します。この結果、図 5-2-4 (3) で ↓ の付いた節点が更新されます。
- [10] (図 5-2-2 の [4] の部分) を並列化します。図 5-2-4 (5) に示すように、例えばランク 0 のプロセスは①～③，⑧～⑩，⑮～⑰を使用して計算しますが、[8] が終了した直後には図 5-2-4 (3) の ↓ に示すように①～⑦しか持っていません。そこでまず [9] で配列 S の値を重ね合せて結果を全プロセスの一時配列 SS に入れ (図 5-2-4 (4))、その後に [10] で SS を参照するようにします (図 5-2-4 (5))。なお [2] で配列 S をゼロクリアしたのは、[9] で配列 S について重ね合せを行うためです。
- [11] (図 5-2-2 の [5] の部分) を並列化します。この結果、図 5-2-4 (6) の ↓ の付いた要素が更新されます。
- タイムステップループが終了した後、[12] で、各プロセスが計算した配列 Y と S の値を重ね合せによってランク 0 のプロセスに集めます。配列 Y，S について重ね合せを行うため、[2] で配列 Y，S をゼロクリアしてあります。

この方法は、次に述べる必要最小限のデータだけを通信する方法と比べて修正が簡単ですが、重ね合せによって余分な通信を行なうため通信時間が増大します。ただし計算部分が多いプログラムの場合はこの方法でも十分『ペイ』できます。また、最終的には次に述べる方法を使用する場合でも、とりあえず並列化がうまくいくかを確認したい場合に上記の方法を使用するという手もあります。

図 5-2-4 (1)

図 5-2-4 (2)

図 5-2-4 (3)

図 5-2-4 (4)

図 5-2-4 (5)

図 5-2-4 (6)

```
   :
 INCLUDE 'mpif.h'
 PARAMETER(IYMAX=12,ISMAX=21)
 REAL*8 Y(IYMAX),S(ISMAX)
 REAL*8 YY(IYMAX),SS(ISMAX),SSS(ISMAX)
 INTEGER INDEX(4,IYMAX)
   :
 CALL MPI_INIT(IERR)
 CALL MPI_COMM_SIZE(～,NPROCS,IERR)
 CALL MPI_COMM_RANK(～,MYRANK,IERR)
 CALL PARA_RANGE
&      (1,IYMAX,NPROCS,MYRANK,IYSTA,IYEND)
 CALL PARA_RANGE                                  [1]
&      (1,ISMAX,NPROCS,MYRANK,ISSTA,ISEND)
 DO IY=1,IYMAX
   Y(IY) = 0.0
 ENDDO
 DO IS=1,ISMAX                                    [2]
   S(IS) = 0.0
 ENDDO
 DO IY=IYSTA,IYEND
   Y(IY) = IY*10.00
 ENDDO
 DO IS=ISSTA,ISEND                                [3]
   S(IS) = IS*100.0
 ENDDO
 DO ITIME=1,10
   DO IS=1,ISMAX
     SS(IS) = 0.0                                 [4]
   ENDDO
```

ⓐへ

ⓐ

```
   DO IY=IYSTA,IYEND   要素から4節点を更新
     DO J=1,4
     SS(INDEX(J,IY))=SS(INDEX(J,IY))+Y(IY)[5]
     ENDDO
   ENDDO
   CALL MPI_ALLREDUCE(SS,SSS,ISMAX,MPI_REAL8,
&          MPI_SUM,MPI_COMM_WORLD,IERR)  [6]
   DO IS=ISSTA,ISEND
     S(IS) = S(IS) + SSS(IS)                      [7]
   ENDDO
   DO IS=ISSTA,ISEND   節点を更新
     S(IS) = S(IS)*0.25                           [8]
   ENDDO
   CALL MPI_ALLREDUCE(S,SS,ISMAX,MPI_REAL8,
&          MPI_SUM,MPI_COMM_WORLD,IERR)  [9]
   DO IY=IYSTA,IYEND   4節点から要素を更新
     DO J=1,4
       Y(IY) = Y(IY) + SS(INDEX(J,IY))            [10]
     ENDDO
   ENDDO
   DO IY=IYSTA,IYEND   要素を更新
     Y(IY) = Y(IY)*0.25                           [11]
   ENDDO
 ENDDO
 CALL MPI_REDUCE(S,SS,ISMAX,MPI_REAL8,
&          MPI_SUM,0,MPI_COMM_WORLD,IERR)
 CALL MPI_REDUCE(Y,YY,IYMAX,MPI_REAL8,            [12]
&          MPI_SUM,0,MPI_COMM_WORLD,IERR)
 IF (MYRANK==0) PRINT *,'結果',S,Y
 CALL MPI_FINALIZE(IERR)
   :
```

図 5-2-5

余談ですが、上記の方法の通信部分には改良の余地があります。まず [6] で各プロセスに全ての節点の結果を渡しましたが、MPI_REDUCE_SCATTER を使用して、各プロセスごとに必要な節点の結果のみを送信することもできます (4-6-6 節参照)。このとき図 5-2-4 (2) は以下のようになります。

この方法を行うには図 5-2-5 を以下のⓐ，ⓑ，ⓒのように修正します。

また、[9] では『重ね合せ』で全ての節点の値を通信しましたが、図 5-2-4 (3) が終了した後で、各プロセスは、自分が計算した節点部分 (↓ の部分) のみを他の全プロセスに送信するようにすれば通信量が減ります。この方法を行うには図 5-2-5 を以下のⓐ，ⓑ，ⓓのように修正します。

```
      :
  PARAMETER(NCPU=3)
  INTEGER ICNT(0:NCPU-1),IDISP(0:NCPU-1)
      :
  DO IRANK = 0, NPROCS-1
    CALL PARA_RANGE
&         (1,ISMAX,NPROCS,IRANK,ISSTA,ISEND)
    ICNT(IRANK)  = ISEND-ISSTA+1
    IDISP(IRANK) = ISSTA-1
  ENDDO
      :
  CALL MPI_REDUCE_SCATTER(SS,SSS(ISSTA),
         ICNT,MPI_REAL8,MPI_SUM,
&        MPI_COMM_WORLD,IERR)
      :
  CALL MPI_ALLGATHERV(S(ISSTA),
&       ISEND-ISSTA+1,MPI_REAL8,SS,ICNT,
&       IDISP,MPI_REAL8,MPI_COMM_WORLD,IERR)
      :
```

➡ ⓐ：宣言文に追加する。

➡ ⓑ：[1]の前に入れる。

➡ ⓒ：[6]を置き換える。

➡ ⓓ：[9]を置き換える。

5-2-3　並列化の方法 2

前節で説明した方法は修正は比較的簡単ですが、全てのデータを「重ね合せ」で通信するため通信量が多くなります。本節では必要最小限のデータのみを通信して並列化する方法について説明します。

まず計算領域の分割方法について説明します。要素は図 5-2-6 (1) に示すように要素数でブロック分割します。次に節点ですが、前節の方法のように節点数でブロック分割するのではなく、各プロセスは自分が担当する要素の四隅の節点を担当します。ただし複数のプロセスの境界にある節点 (図 5-2-6 (1) の〓で囲んだ部分) は、ランクの最も小さいプロセスが担当することにします。例えば、③，⑩，⑰は (ランク 1 でなく) ランク 0 が、⑤，⑫，⑲は (ランク 2 でなく) ランク 1 が担当します。まとめると、各プロセスが担当する節点番号は図 5-2-6 (2) のようになります。並列化したプログラムを図 5-2-10 に示し、以下で説明します。

テーブル類の作成

まず、後の並列計算で使用するテーブル類 (図 5-2-6 (4)〜図 5-2-6 (9)) を作成します。

- [1] でテーブル類を初期化し、[2] で図 5-2-6 (4) に示すテーブル IDISP に値を設定します。IDISP は [14]，[18]，[23] の通信の引数を指定します (付録の MPI_ALLTOALLV の引数 sdispls と rdispls、MPI_GATHERV の引数 displs 参照)。
- [3] で要素数 IYMAX をブロック分割し (4-5-4 節参照)、各プロセスが担当する要素の開始番号を IIYSTA に、担当する要素数を IYCNT にセットします (図 5-2-6 (5) 参照)。
- [4] で作業配列 ITEMP に値を設定します (図 5-2-6 (6) 参照)。図 5-2-6 (1) で、例えばランク 0 が担当する要素は$\boxed{1}$〜$\boxed{4}$ですが、これらの要素の四隅にある節点番号を図 5-2-6 (3) の配列 INDEX を使用して調べると、①〜③，⑧〜⑩，⑮〜⑰であることが分かります。そこで配列 ITEMP(〜,0) のこれらの節点番号のところに「1」をセットします (ITEMP のその他の部分はゼロになっている必要があるため、[1] でゼロクリアしています)。
- [5] で改めて、自分が担当する要素の開始番号を IYSTA に、終了番号を IYEND に設定します (IYSTA は [3] で求めた IIYSTA (MYRANK) と同じ値です)。
- [6] でテーブル IBUF1，ICNT1，IBUF2，ICNT2 を作成します。例えば図 5-2-6 (1) で節点③はランク 0 と 1 の境界にあります。この場合、前述のようにランク 0 と 1 のうちランクの小さいランク 0 のプロセスが節点③を担当します。これを [6] では次のように処理します。図 5-2-6 (6) の節点③（〓で囲んだ部分）に複数の「1」が付いていますが、最も左 (本例ではランク 0) の「1」のみを残し、それ以外 (本例ではランク 1) の「1」を「0」に変更します (1 → 0 の部分)。「1」を「0」に変更した場合は、ランク 0 のプロセスでは IBUF2(〜,1) に③を入れて ICNT2(1) に 1 を加え (図 5-2-6 (9) の○で囲んだ部分)、ランク 1 のプロセスでは IBUF1(〜,0) に③を入れて ICNT1(0) に 1 を加えます (図 5-2-6 (8) の○で囲んだ部分)。
  図 5-2-6 (8) (9) から分かるように、作成された IBUF1，ICNT1，IBUF2，ICNT2 の値は、各プロセスで異なることに注意して下さい（それ以外のテーブル類の値は全プロセスで同一です）。
- [7] では先程作成した ITEMP を使用して、各プロセスが担当する節点番号を ISNUM に、節点の個数を ISCNT にセットします (図 5-2-6 (7) 参照)。以上で並列計算で必要となるテーブル類の作成は完了しました。

並列計算部分

- [8] と [9] で図 5-2-2 の [1] を並列化します。このうち [9] は節点番号で並列化するので、先程作成したテーブル ISNUM と ISCNT (図 5-2-6 (7)) を使用します。
- [10] からタイムステップのループが開始します。以後の説明では図 5-2-7 (1)〜(4) も参照して下さい (図中で [11] などの数字は図 5-2-10 内の数字に対応します)。
  図 5-2-7 (1) で、プロセス間の境界にある節点のうち、そのプロセスが担当する節点を〓で、そのプロセスが担当しない節点を〓で囲みます。まず [11] で、各プロセスは〓内の節点の配列 S の値をゼロクリアします (〓以外の節点については配列 S に現在の値がそのまま入っています)。

- 次に [12] で図 5-2-2 の [2] を並列化します (図 5-2-7 (1) の要素の四隅の矢印参照)。例えば境界上の節点③に着目すると、ランク 0 では配列 S の現在の値に要素 Y の値が加算され（これを〓で示します)、ランク 1 では [11] で〓をゼロクリアしたので配列 S に要素 Y の値がそのまま入ります（これを〓で示します)。つまりランク 0 の〓にランク 1 の〓を加算すると正しい③になります (これを以下の [13]〜[15] で行います)。
- 正しい境界の値を求めるため、[13]〜[15] で〓を通信して〓に加算します (図 5-2-7 (1) と (2) の間の ↙ 参照)。この部分を図 5-2-8 の上半分で説明します。図中で [13] などの数字は図 5-2-10 内の数字に対応します。また例えば③は節点番号（整数）を示し、(3) は節点③の配列 S 内の物理量を示します。なお、配列 S は 1 次元ですが、紙面の都合で 2 次元のように表します。
  まず [13] で、配列 S のうち送信する節点（〓内の節点）の物理量を、送信用配列 BUF1 の宛先プロセスのランクの所に入れます。次に [14] で集団通信ルーチン MPI_ALLTOALLV（付録参照）を用いて通信を行います。なお、配列 ICNT1, ICNT2 には各プロセスに送（受）信する個数がセットされていますが、そのプロセスに対して送（受）信を行わない場合はゼロになっている必要があるため、[1] で配列 ICNT1, ICNT2 をゼロクリアしています。
  最後に [15] で、配列 BUF2 に受信した境界の節点の物理量を、配列 S の該当箇所（〓内の節点）に加算します。以上で図 5-2-7 (2) の〓内の節点は正しい物理量になりました。
- [16] で図 5-2-2 の [3] を並列化します (図 5-2-7 (2) の ↓ 参照)。[16] が終了した時点で、図 5-2-7 (2) の〓の部分には節点の物理量の正しい値が入っていますが、〓の部分には正しい値が入っていません。そこで [17]〜[19] で〓を通信して〓に代入します (図 5-2-7 (2) と (3) の間の ↘ 参照)。この部分の通信方法は [13]〜[15] と同様なので ([15] は加算で、[19] は代入であるところが異なります)、説明は省略します (図 5-2-8 の下半分参照)。
- [19] が終了した時点で、図 5-2-7 (3) の〓と〓の両方に節点の物理量の正しい値が入りました。続いて [20] で図 5-2-2 の [4] を並列化し (図 5-2-7 (3) の要素の四隅の矢印参照)、[21] で図 5-2-2 の [5] を並列化します (図 5-2-7 (4) の ↓ 参照)。

計算結果の収集

タイムステップのループが終了し、各プロセスが計算した結果をランク 0 のプロセスに収集します。

- [22] で図 5-2-9 (1) に示すように、ランク 0 以外のプロセスは配列 S（節点の物理量）のうち自分が担当した部分を 1 次元配列 BUF にセットします。続いて [23] で、集団通信ルーチン MPI_GATHERV を用いてランク 0 の配列 BUF1 に収集します (付録と、4-6-3-1 節の「ケース 1」参照)。次に [24] で、ランク 0 のプロセスは配列 BUF1 に受信した値を配列 S の該当箇所に代入します。
- [25] では、図 5-2-9 (2) に示すように、配列 Y（要素の物理量）のうち各プロセスが担当した部分をランク 0 の配列 Y に収集します (図で例えば1は要素番号（整数）を示し、[1] は要素1の配列 Y 内の物理量を示します)。この通信を集団通信ルーチン MPI_GATHERV で行うと、図 5-2-9 (2) のランク 0 の配列 Y の空いている部分に集めることができないため (3-3-5 節で説明した、送信バッファーと受信バッファーがメモリー上で重なってはいけないという制限により)、MPI_ISEND, MPI_IRECV を使用して通信を行いました (4-6-3-1 節の「ケース 2」参照)。

以上でプログラムの説明を終了します。この方法は修正が面倒ですが、必要最小限のデータのみを通信するため、前節の方法と比較して（一般に）通信量は減少します。なお以下の点に考慮する必要があります。

- 要素番号のつけ方によっては、プロセス間の境界上の節点の数が多くなり、通信量が多くなってしまう可能性があります。
- 各プロセスが担当する節点数がプロセス間で異なり、ロードバランスが不均等になる可能性があります。
- 図 5-2-10 では、説明を簡単にするため、配列 ISNUM, IBUF1, IBUF2, BUF1, BUF2 の大きさが、必要な最小限の大きさよりかなり大きくなっており、使用できるメモリーが少ない場合は改良する必要があります。

図 5-2-6 (1)　要素(□)の分割

図 5-2-6 (2)　節点(○)の分割

各要素の四隅の節点番号

INDEX

1	2	3	4	5	6	7	8	9	10	11	12
①	⑧	②	⑨	③	⑩	④	⑪	⑤	⑫	⑥	⑬
②	⑨	③	⑩	④	⑪	⑤	⑫	⑥	⑬	⑦	⑭
⑨	⑯	⑩	⑰	⑪	⑱	⑫	⑲	⑬	⑳	⑭	㉑
⑧	⑮	⑨	⑯	⑩	⑰	⑪	⑱	⑫	⑲	⑬	⑳

図 5-2-6 (3)

BUF1とBUF2内で、各プロセスの送受信データが入る最初の位置(先頭からの変位)をセットします。

	0	1	2
IDISP	0	21	42

BUF1とBUF2	0	1	2
1	0	21	42
2	1	22	43
3	2	23	44
:	:	:	:
ISMAX=21	20	41	62

節点の物理量を入れる送受信バッファー

図 5-2-6 (4)

各プロセスが担当する最初の要素番号

	0	1	2
IIYSTA	1	5	9
IYCNT	4	4	4

各プロセスが担当する要素数

図 5-2-6 (5)

各プロセスに所属する節点番号に「1」を付けます。その他の部分は「0」です。

ITEMP	0	1	2
①	1		
②	1		
③	1	1→0	
④		1	
⑤		1	1→0
⑥			1
⑦			1
⑧	1		
⑨	1		
⑩	1	1→0	
⑪		1	
⑫		1	1→0
⑬			1
⑭			1
⑮	1		
⑯	1		
⑰	1	1→0	
⑱		1	
⑲		1	1→0
⑳			1
ISMAX=㉑			1

→

各プロセスが担当する節点番号

ISNUM	0	1	2
1	①	④	⑥
2	②	⑤	⑦
3	③	⑪	⑬
4	⑧	⑫	⑭
5	⑨	⑱	⑳
6	⑩	⑲	㉑
7	⑮		
8	⑯		
9	⑰		
:			
ISCNT	9	6	6

各プロセスが担当する節点数

図 5-2-6 (7)

図 5-2-6 (6)

他のプロセスとの境界上の節点のうち、自分が担当しない節点の番号が、その節点を担当するプロセスのランクの所に入ります。

【ランク0】

IBUF1	0	1	2
1			
2			
3			
:			
ICNT1	0	0	0

【ランク1】

IBUF1	0	1	2
1	③		
2	⑩		
3	⑰		
:			
ICNT1	3	0	0

【ランク2】

IBUF1	0	1	2
1		⑤	
2		⑫	
3		⑲	
:			
ICNT1	0	3	0

各プロセスのIBUF1内の節点数

図 5-2-6 (8)

他のプロセスとの境界上の節点のうち、自分が担当する節点の番号が、その節点を担当しないプロセスのランクの所に入ります。

【ランク0】

IBUF2	0	1	2
1		③	
2		⑩	
3		⑰	
:			
ICNT2	0	3	0

【ランク1】

IBUF2	0	1	2
1			⑤
2			⑫
3			⑲
:			
ICNT2	0	0	3

【ランク2】

IBUF2	0	1	2
1			
2			
3			
:			
ICNT2	0	0	0

各プロセスのIBUF2内の節点数

図 5-2-6 (9)

（注）以下の図で
－ ○：節点の物理量を入れる配列 S
－ □：要素の物理量を入れる配列 Y
を示します。

図 5-2-7 (1)

↓↙ [13] [14] [15] 通信して加算 ↓↙ [13] [14] [15] 通信して加算 ↓

図 5-2-7 (2)

↓ [17] [18] [19] 通信して代入 ↘ ↓ [17] [18] [19] 通信して代入 ↘ ↓

図 5-2-7 (3)

図 5-2-7 (4)

図 5-2-8　境界の節点（の物理量）の通信

図 5-2-9 (1)　配列 S の収集

図 5-2-9 (2)　配列 Y の収集

```
    :
 INCLUDE 'mpif.h'
 PARAMETER(IYMAX=12,ISMAX=21)
 REAL*8 Y(IYMAX),S(ISMAX)
 INTEGER INDEX(4,IYMAX)
 PARAMETER(NCPU=3)
 INTEGER NPROCS,MYRANK
 INTEGER ISTATUS(MPI_STATUS_SIZE)
 INTEGER ITEMP(ISMAX,0:NCPU-1)
 INTEGER IDISP(0:NCPU-1),ISCNT(0:NCPU-1),
&        ISNUM(ISMAX,0:NCPU-1)
 INTEGER IIYSTA(0:NCPU-1),IYCNT(0:NCPU-1)
 INTEGER ICNT1(0:NCPU-1),
&        ICNT2(0:NCPU-1)
 INTEGER IBUF1(ISMAX,0:NCPU-1),
&        IBUF2(ISMAX,0:NCPU-1)
 REAL*8  BUF(ISMAX)
 REAL*8  BUF1(ISMAX,0:NCPU-1),
&        BUF2(ISMAX,0:NCPU-1)
    :
 CALL MPI_INIT(IERR)
 CALL MPI_COMM_SIZE
&    (MPI_COMM_WORLD,NPROCS,IERR)
 CALL MPI_COMM_RANK
&    (MPI_COMM_WORLD,MYRANK,IERR)
 DO IRANK=0,NPROCS-1        ↓テーブル類の作成
   DO I=1,ISMAX
     ITEMP(I,IRANK) = 0
   ENDDO
   ICNT1(IRANK) = 0                          [1]
   ICNT2(IRANK) = 0
   ISCNT(IRANK) = 0
 ENDDO
 DO IRANK=0,NPROCS-1
   IDISP(IRANK) = IRANK*ISMAX                [2]
 ENDDO
 DO IRANK=0,NPROCS-1
   CALL PARA_RANGE
&     (1,IYMAX,NPROCS,IRANK,IYSTA,IYEND)
   IIYSTA(IRANK) = IYSTA                     [3]
   IYCNT(IRANK) = IYEND-IYSTA+1
   DO IY=IYSTA,IYEND
     DO J=1,4
       ITEMP(INDEX(J,IY),IRANK) = 1          [4]
     ENDDO
   ENDDO
 ENDDO
 CALL PARA_RANGE                             [5]
&   (1,IYMAX,NPROCS,MYRANK,IYSTA,IYEND)     [5]
```

Ⓐ

```
 DO I=1,ISMAX
   IFLG = 0
   DO IRANK=0,NPROCS-1
     IF (ITEMP(I,IRANK)==1) THEN
       IF (IFLG == 0) THEN
         IFLG = 1
         IIRANK = IRANK
       ELSE
         ITEMP(I,IRANK) = 0
         IF (IRANK == MYRANK) THEN
           ICNT1(IIRANK)=ICNT1(IIRANK)+1     [6]
           IBUF1(ICNT1(IIRANK),IIRANK)=I
         ELSEIF (IIRANK == MYRANK) THEN
           ICNT2(IRANK)=ICNT2(IRANK)+1
           IBUF2(ICNT2(IRANK),IRANK)=I
         ENDIF
       ENDIF
     ENDIF
   ENDDO
 ENDDO
 DO IRANK=0,NPROCS-1
   DO I=1,ISMAX
     IF (ITEMP(I,IRANK)==1) THEN
       ISCNT(IRANK) = ISCNT(IRANK) + 1
       ISNUM(ISCNT(IRANK),IRANK) = I           [7]
     ENDIF
   ENDDO
 ENDDO
 DO IY=IYSTA,IYEND          ↓並列計算部分
   Y(IY) = IY*10.0                           [8]
 ENDDO
 DO I=1,ISCNT(MYRANK)
   IS = ISNUM(I,MYRANK)
   S(IS) = IS*100.0                          [9]
 ENDDO
 DO ITIME=1,10                               [10]
   DO IRANK=0,NPROCS-1
     DO I=1,ICNT1(IRANK)
       S(IBUF1(I,IRANK)) = 0.0               [11]
     ENDDO
   ENDDO
   DO IY=IYSTA,IYEND
     DO J=1,4
       S(INDEX(J,IY))=S(INDEX(J,IY))+Y(IY)   [12]
     ENDDO
   ENDDO
```

Ⓑ

```
                                                  Ⓑ
  DO IRANK=0,NPROCS-1
    DO I=1,ICNT1(IRANK)
      BUF1(I,IRANK)=S(IBUF1(I,IRANK))                [13]
    ENDDO
  ENDDO
  CALL MPI_ALLTOALLV
&      (BUF1,ICNT1,IDISP,MPI_REAL8,
&       BUF2,ICNT2,IDISP,MPI_REAL8,                   [14]
&       MPI_COMM_WORLD,IERR)
  DO IRANK=0,NPROCS-1
    DO I=1,ICNT2(IRANK)
      S(IBUF2(I,IRANK)) =
&     S(IBUF2(I,IRANK)) + BUF2(I,IRANK)              [15]
    ENDDO
  ENDDO
  DO I=1,ISCNT(MYRANK)
    IS = ISNUM(I,MYRANK)
    S(IS) = S(IS)*0.25                               [16]
  ENDDO
  DO IRANK=0,NPROCS-1
    DO I=1,ICNT2(IRANK)
      BUF2(I,IRANK)=S(IBUF2(I,IRANK))                [17]
    ENDDO
  ENDDO
  CALL MPI_ALLTOALLV
&      (BUF2,ICNT2,IDISP,MPI_REAL8,
&       BUF1,ICNT1,IDISP,MPI_REAL8,                   [18]
&       MPI_COMM_WORLD,IERR)
  DO IRANK=0,NPROCS-1
    DO I=1,ICNT1(IRANK)
      S(IBUF1(I,IRANK))=BUF1(I,IRANK)                [19]
    ENDDO
  ENDDO
  DO IY=IYSTA,IYEND
    DO J=1,4
      Y(IY) = Y(IY) + S(INDEX(J,IY))                 [20]
    ENDDO
  ENDDO
  DO IY=IYSTA,IYEND
    Y(IY) = Y(IY)*0.25                               [21]
  ENDDO
ENDDO
                                                  Ⓒ
```

```
                                                  Ⓒ
IF (MYRANK/=0) THEN          ⬇計算結果の収集
  DO I=1,ISCNT(MYRANK)
    BUF(I) = S(ISNUM(I,MYRANK))                      [22]
  ENDDO
ENDIF
CALL MPI_GATHERV
&    (BUF ,ISCNT(MYRANK),MPI_REAL8,
&     BUF1,ISCNT,IDISP  ,MPI_REAL8,                  [23]
&     0,MPI_COMM_WORLD,IERR)
IF (MYRANK==0) THEN
  DO IRANK=1,NPROCS-1
    DO I=1,ISCNT(IRANK)
      S(ISNUM(I,IRANK)) = BUF1(I,IRANK)              [24]
    ENDDO
  ENDDO
ENDIF
IF (MYRANK==0) THEN
  DO IRANK=1,NPROCS-1
    CALL MPI_IRECV(Y(IIYSTA(IRANK)),
&       IYCNT(IRANK),MPI_REAL8,IRANK,1,
&       MPI_COMM_WORLD,IREQ,IERR)
    CALL MPI_WAIT(IREQ,ISTATUS,IERR)
  ENDDO                                              [25]
ELSE
  CALL MPI_ISEND(Y(IYSTA),IYEND-IYSTA+1,
&  MPI_REAL8,0,1,MPI_COMM_WORLD,IREQ,IERR)
  CALL MPI_WAIT(IREQ,ISTATUS,IERR)
ENDIF
IF (MYRANK==0) PRINT *,'RESULT = ',S,Y
CALL MPI_FINALIZE(IERR)
  :
```

図 5-2-10

5-3　LU 分解

密行列連立一次方程式の直接解法である LU 分解の並列化について説明します。なお、密行列の LU 分解のルーチンは並列化が容易なため、6 章で紹介する並列版の数値計算ライブラリーに通常含まれており、ライブラリールーチンの方が Fortran で作成した並列版よりも一般に高速です。

分割方法

ここでは簡単のため 6 × 6 の行列で説明します。LU 分解では、図 5-3-1 (1) の左から右に計算ステップが進んでいきます。各ステップでは、図の着色した部分を計算します。このように LU 分解では、ステップが進むにつれて、計算する部分が 1 行 1 列ずつ少なくなるという特徴があります。

図 5-3-1 (1)

並列化した場合の分割方法について検討します。ここではプロセス数が 3 個であるとします。図 5-3-1 (2) のようにブロック分割で分割した場合、最初のうちはプロセス間のロードバランスは均等ですが、ステップが進むにつれ、ランク 0 のプロセスの計算部分は終了し、さらに進むとランク 1 のプロセスの計算部分も終了し、次第にロードバランスが不均等になってしまいます。

一方、サイクリック分割を行った図 5-3-1 (3) では、どのステップでも、プロセス間のロードバランスは、ほぼ一定になっています。なお、この例では行列の大きさが 6 × 6 なのでロードバランスがやや不均等ですが、実際には行列は 1000 × 1000 などの大きさなので、ほぼ一定であると言えます。このように、計算が進むにつれて計算部分の大きさが変化する場合、サイクリック分割が有効になります。

なお、並列版の数値計算ライブラリーに含まれる LU 分解では、高速化のために行と列の両方向をブロック・サイクリック分割したり、配列を縮小している場合がありますが、以下のプログラムでは説明を分かりやすくするため、列方向のみをサイクリック分割し、配列の縮小はしません。

図 5-3-1 (2)　ブロック分割

図 5-3-1 (3)　サイクリック分割

プログラム例

連立一次方程式 $Ax = b$ を LU 分解で解く単体版のプログラムを図 5-3-2 に示します。なお、説明を分かりやすくするため、ループ・アンローリング、ブロック化などのチューニング、およびピボッティングは行っていません。

LU 分解では、以下に示すように、まず連立一次方程式 $Ax = b$ の行列 A を、LU という 2 つの三角行列に分解します。次に行列 LU とベクトル b を使って求解（前進消去と後退代入）を行い、作業ベクトル y を経て答えのベクトル x が求まります。

$$A = LU \quad \text{【分解】}$$
$$y = L^{-1}b \quad \text{【求解（前進消去）】}$$
$$x = U^{-1}y \quad \text{【求解（後退代入）】}$$

上記の行列 A と LU が図 5-3-2 の配列 A に対応し、ベクトル b, y, x が配列 B に対応します。つまり最初設定されていた行列 A の値は壊されて LU (行列の▬部分が L、▬部分が U) になります。ベクトル b の値も壊されて作業ベクトル y になり、最後に答えのベクトル x になります。

求解部分の解法には、図 5-3-2 に示す内積型と、図 5-3-3 に示す三項演算型があります。内積型はメモリー上で飛び飛びの要素を参照するため、スカラー計算機ではキャッシュミスが発生して速度が低下しますが (参考文献 [6] の 4 章参照)、三項演算型より並列化がしやすいので、ここでは内積型を使用します。分解より求解の方が計算時間ははるかに短いので、分解、求解をともに一度だけ行う場合は内積型で問題ないですが、分解が 1 回のみで求解だけを何度も行う場合は、内積型は速度が遅いので使用しないで下さい。

```
PROGRAM MAIN
PARAMETER(N=6)
DIMENSION A(N,N),B(N)
  :
DO K=1,N        【分解】                          ①
  DO I=K+1,N
    A(I,K) = A(I,K)/A(K,K)                        ②
  ENDDO
  DO J=K+1,N
    DO I=K+1,N
      A(I,J) = A(I,J) - A(I,K)*A(K,J)             ③
    ENDDO
  ENDDO
ENDDO
DO I=1,N        【求解(前進消去)】                ④
  SUM = 0.0
  DO J=1,I-1
    SUM = SUM + A(I,J)*B(J)                       ⑤
  ENDDO
  B(I) = B(I) - SUM                               ⑥
ENDDO
DO I=N,1,-1     【求解(後退代入)】                ⑦
  SUM = 0.0
  DO J=I+1,N
    SUM = SUM + A(I,J)*B(J)                       ⑧
  ENDDO
  B(I) = (B(I)-SUM)/A(I,I)                        ⑨
ENDDO
  :
```

図 5-3-2　求解が内積型（単体版）

```
  :
DO J=1,N        【求解(前進消去)】
  DO I=J+1,N
    B(I) = B(I) - A(I,J)*B(J)
  ENDDO
ENDDO
DO J=N,1,-1     【求解(後退代入)】
  B(J) = B(J)/A(J,J)
  DO I=1,J-1
    B(I) = B(I) - A(I,J)*B(J)
  ENDDO
ENDDO
  :
```

図 5-3-3　求解が三項演算型（単体版）

並列化したプログラムを以下に示し、次のページ以降で説明します。

```
  PROGRAM MAIN
  INCLUDE 'mpif.h'
  PARAMETER(N=6)
  DIMENSION A(N,N),B(N)
  INTEGER MAP(N)
  CALL MPI_INIT(IERR)
  CALL MPI_COMM_SIZE
&    (MPI_COMM_WORLD,NPROCS,IERR)
  CALL MPI_COMM_RANK
&    (MPI_COMM_WORLD,MYRANK,IERR)
  DO I=1,N
    MAP(I) = MOD(I-1,NPROCS)                    1
  ENDDO
    :
  DO K=1,N        【分解】                      2
    IF (MAP(K)==MYRANK) THEN                    3
      DO I=K+1,N
        A(I,K) = A(I,K)/A(K,K)                  4
      ENDDO
    ENDIF
    CALL MPI_BCAST(A(K+1,K),N-K,MPI_REAL,       5
&            MAP(K),MPI_COMM_WORLD,IERR)
    DO J=K+1,N
      IF (MAP(J) == MYRANK) THEN                6
        DO I=K+1,N
          A(I,J) = A(I,J) - A(I,K)*A(K,J)       7
        ENDDO
      ENDIF
    ENDDO
  ENDDO
```

```
  DO I=1,N        【求解(前進消去)】            8
    SUM = 0.0
    DO J=1,I-1
      IF (MAP(J)==MYRANK)                       9
&         SUM = SUM + A(I,J)*B(J)
    ENDDO
    CALL MPI_REDUCE(SUM,SSUM,1,MPI_REAL,        10
&        MPI_SUM,MAP(I),MPI_COMM_WORLD,IERR)
    IF (MAP(I)==MYRANK) B(I) = B(I) - SSUM      11
  ENDDO
  DO I=N,1,-1     【求解(後退代入)】            12
    SUM = 0.0
    DO J=I+1,N
      IF (MAP(J)==MYRANK)                       13
&         SUM = SUM + A(I,J)*B(J)
    ENDDO
    CALL MPI_REDUCE(SUM,SSUM,1,MPI_REAL,        14
&        MPI_SUM,MAP(I),MPI_COMM_WORLD,IERR)
    IF (MAP(I)==MYRANK)
&       B(I) = (B(I)-SSUM)/A(I,I)               15
  ENDDO
  DO I=1,N
    IF (MAP(I)/=MYRANK) B(I) = 0.0              16
  ENDDO
  CALL MPI_ALLREDUCE(MPI_IN_PLACE,B,N,          17
&     MPI_REAL,MPI_SUM,MPI_COMM_WORLD,IERR)
    :
```

図 5-3-4　求解が内積型（並列版）

分解

【単体版の動作】

- 図 5-3-5 (1) に、6×6 の行列 A を行列 LU に変換する過程の一部を示します。
- 図 5-3-2 の①のループで K = 1 のとき、②で、図 5-3-5 (1)（上段）の各★に対し、「★ = ★ / ●」を行います。
- 次に③で、図 5-3-5 (1)（上段）の各■に対し、「■ = ■ - ★ * ▲」を行います。
- ①のループで K = 2 となり、図 5-3-5 (1)（下段）の太線内に対し、上記と同様の処理を行います。
- 以後①のループで K = 3, 4, 5 と同様の処理を行います。前述のように、計算する部分は次第に少なくなります。

【並列版の動作】

- 前述のように、列方向をサイクリック分割して並列化を行います。
- 図 5-3-4 の1で、行列 A の各列番号と、その列を担当するプロセスのランク値の対応を表す、MAP という対応表 (図 5-3-5 (2)（上段)) を作成します (4-5-5 節の分割方法 2 参照)。なお、MAP を使用すると後述するように IF 文が必要となり、計算部分の速度が低下します。MAP を使用しない方法 (4-5-5 節の分割方法 1) もありますが、ここでは説明を分かりやすくするため使用しました。
- 2のループで K = 1 のとき、4で、図 5-3-6 (2)（上段）の 1 列目を担当するランク 0 のみが、各★に対し、「★ = ★ / ●」を行います。3の IF 文があるため、ランク 0 のみがこの処理を行ないます。
- 5で、ランク 0 からその他のプロセスに、計算した★のデータを送信します。このとき送信元のプロセスは、1 列目を担当する MAP(1) (ランク 0) のプロセスとなります。
- 7で、各プロセスは自分が担当する列の各■に対し、「■ = ■ - ★ * ▲」を行います (ランク 0 以外のプロセスでは「■ = ■ - ☆ * ▲」)。6の IF 文があるため、各プロセスは自分が担当する列のみを処理します。
- 2のループで K = 2 となり、図 5-3-5 (2)（下段）の太線内に対し、上記と同様の処理を行います。5の通信で、送信元のプロセスは、2 列目を担当する MAP(2) (ランク 1) のプロセスとなります。
- 以後2のループで K = 3, 4, 5 と同様の処理を行います。5の通信で、送信元のプロセスはランク 2, 0, 1 と変化します。

図 5-3-5 (1)　単体版の動作

図 5-3-5 (2)　並列版の動作

求解（前進消去）

【単体版の動作】

- 図 5-3-6 (1) に、ベクトル b を作業ベクトル y に変換する過程の一部を示します。
- 説明の都合上、図 5-3-2 の④のループで、I = 1, 2, 3, 4 の順に以下と同様の処理がすでに行われ、B(1)〜B(4) は計算が終了しているとします。
- ④のループで I = 5 のとき、⑤で、図 5-3-6 (1)（上段）の〓と〓の内積を計算し、結果が変数 SUM に入ります。
- ⑥で★に対し、「★ = ★ - SUM」を行います。これで B(5) の計算が終了しました。
- ④のループで I = 6 となり、図 5-3-6 (1)（下段）に示すように上記と同様の処理をし、B(6) を計算します。

【並列版の動作】

- LU 分解の結果、図 5-3-6 (2)（上段）に示すように、配列 A のうち、対応表 MAP に従って各プロセスが担当した列のみに正しい値（□と■）が入っています。配列 B の各要素も、対応表 MAP に従って上から順にランク 0, 1, 2, 0, 1, 2 のプロセスが担当します。
- 単体版の説明と同様に、図 5-3-4 の8のループで I = 5 の場合から説明します。9で、各プロセスは図 5-3-6 (2)（上段）の〓と〓のうち、自分が担当する要素の内積を計算し、結果が変数 SUM に入ります。9の IF 文があるため、各プロセスは自分が担当する要素のみを計算します。前述のように、IF 文を使用すると速度が低下しますが、ここでは説明を分かりやすくするため使用しました。
- 各プロセスが求めた部分的な内積 SUM の値を、10で通信しながら合計し、結果が★を担当するランク 1 のプロセスの変数 SSUM に送られます。
- 11で、ランク 1 のプロセスは★に対し、「★ = ★ - SSUM」を行います。これで B(5) の計算が終了しました。
- 8のループで I = 6 となり、図 5-3-6 (2)（下段）に示すように上記と同様の処理をし、B(6) を計算します。10の通信で、宛先のプロセスは★を担当するランク 2 となります。

（以上略）

図 5-3-6 (1)　単体版の動作

図 5-3-6 (2)　並列版の動作

求解（後退代入）

【単体版の動作】

- 図 5-3-7 (1) に、作業ベクトル y を答えのベクトル x に変換する過程の一部を示します。
- 説明の都合上、図 5-3-2 の⑦のループで、I = 6,5,4,3 の順に以下と同様の処理がすでに行われ、B(6)〜B(3) は計算が終了しているとします。
- ⑦のループで I = 2 のとき、⑧で、図 5-3-7 (1)（上段）の〓と〓の内積を計算し、結果が変数 SUM に入ります。
- ⑨で★に対し、「★ = (★ -SUM)/ ●」を行います。これで B(2) の計算が終了しました。
- ⑦のループで I = 1 となり、図 5-3-7 (1)（下段）に示すように上記と同様の処理をし、B(1) を計算します。

【並列版の動作】

- 単体版の説明と同様に、図 5-3-4 の12のループで I = 2 の場合から説明します。13で、各プロセスは図 5-3-7 (2)（上段）の〓と〓のうち、自分が担当する要素の内積を計算し、結果が変数 SUM に入ります。13の IF 文があるため、各プロセスは自分が担当する要素のみを計算します。前述のように、IF 文を使用すると速度が低下しますが、ここでは説明を分かりやすくするため使用しました。
- 各プロセスが求めた部分的な内積 SUM の値を、14で通信しながら合計し、結果が★を担当するランク 1 のプロセスの変数 SSUM に送られます。
- 15で、ランク 1 のプロセスは★に対し、「★ = (★ -SSUM)/ ●」を行います。これで B(2) の計算が終了しました。
- 12のループで I = 1 となり、図 5-3-7 (2)（下段）に示すように上記と同様の処理をし、B(1) を計算します。14の通信で、宛先のプロセスは★を担当するランク 0 となります。
- 後退代入が終了し、最後に各プロセスが担当した配列 B の要素を 1 箇所に集めます。16で、各プロセスは配列 B のうち自分が担当しない要素に 0 を入れます。そして17で、図 5-3-7 (2)（下段）に示すように、配列 B の各要素を通信しながら合計し (4-6-6 節参照)、結果が全プロセスの配列 B に送られます。これが答えのベクトル x となります。

図 5-3-7 (1)　単体版の動作

図 5-3-7 (2)　並列版の動作

5-4　ICCG 法

本節では、差分法などで現れる対称帯行列の反復解法である ICCG 法を並列化する方法として、パイプライン法とパラレル・ブロック・オーダリング法を紹介します。これらの方法は非対称帯行列の反復解法 (ILUBCG 法、ILUCGS 法、ILU Bi-CGSTAB 法など) にも適用することができます。

なお、第 6 章で紹介する並列版の数値計算ライブラリーに、反復法のルーチンが含まれているものがあり (例えば PCP)、実用上はそちらを使用した方がよいと思われます。

本書では紹介しませんが、ICCG 法の前処理部分を簡単にした SCG 法 (参考文献 [49]) は、前処理が簡単なので収束性にやや難がありますが、並列計算に向いています。さらに、ベクトル化効率を高めるための TF 法 (参考文献 [3]) は、並列化も可能ではないかと思われます（未調査)。

5-4-1　パイプライン法による並列化

ICCG 法のアルゴリズム

まず ICCG 法をパイプライン法 (4-6-7 節参照) で並列化する方法を説明します。単体版の ICCG 法のアルゴリズムを図 5-4-1 (1) に、プログラムを図 5-4-1 (2) に示します。これらは参考文献 [22] [23] を参照しました。なお、参考文献で説明されている MICCG 法 (ICCG 法の加速版) のロジックは、説明を簡単にするために省略していますので、図 5-4-1 (2) のプログラムはあくまでサンプルと考えて下さい。

図 5-4-1 (1) で行列 A、ベクトル x, b は連立一次方程式 $Ax = b$ を表し、ベクトル p, q, r は作業用ベクトル、α, β, c_1, c_2, c_3 はスカラー変数です。実線、二重線、波線の計算はサブルーチンを使用します。

① x_0, p_0, q, r_0を初期化します。
② $A \fallingdotseq LDL^T$　　(サブルーチンDECOMPを使用)
③ $q = Ax_0$　　(サブルーチンAXSUBを使用)
④ $r_0 = b - q$
⑤ $p_0 = (LDL^T)^{-1} r_0$　　(サブルーチンLDLTを使用)
⑥ $c_1 = (r_0, p_0)$

DO $k = 0, 1, \cdots$　　(収束反復のループ)
　⑦ $q = Ap_k$　　(サブルーチンAXSUBを使用)
　⑧ $c_2 = (p_k, q)$
　⑨ $\alpha_k = c_1 / c_2$
　⑩ $x_{k+1} = x_k + \alpha_k p_k$
　⑪ $r_{k+1} = r_k - \alpha_k q$
　⑫ $\| x_{k+1} - x_k \| / \| x_k \| < \varepsilon$ のとき終了します。
　⑬ $q = (LDL^T)^{-1} r_{k+1}$　　(サブルーチンLDLTを使用)
　⑭ $c_3 = (r_{k+1}, q)$
　⑮ $\beta_k = c_3 / c_1$
　⑯ $c_1 = c_3$
　⑰ $p_{k+1} = q + \beta_k p_k$
ENDDO

図 5-4-1 (1)

単体版のプログラムの説明

図 5-4-1 (2) のプログラムを説明します。なお、図 5-4-1 (1) 内の番号と図 5-4-1 (2) 内の番号は対応しています。図 5-4-1 (2) の [1] で連立一次方程式 $Ax = b$ の A と b に値を設定し（設定方法については後述します）、[2] で計算条件を設定し、[3] で ICCG 法の計算を実際に行うサブルーチン ICCG をコールします。

サブルーチン ICCG の各引数の意味を以下に示します（配列の構造は後述します）。

- A, X, B: 連立一次方程式 $Ax = B$ に相当します。A と B を設定して本サブルーチンをコールすると、解が X に戻ります。
- D: 図 5-4-1 (1) で $A \quad LDL^{\mathrm{T}}$ と分解したときの対角要素 D に相当する配列です。
- MX, MY: 図 5-4-2 (4) に示すように、行列 A のバンド幅が MX、行列 A の行数が MX × MY となります。
- P, Q, R: 図 5-4-1 (1) の作業用ベクトル p, q, r に相当する配列です。
- EPS: 収束判定値を設定します。サブルーチンが終了すると、収束したときの相対誤差 $\|x_{k+1} - x_k\|/\|x_k\|$ が戻ります。
- ITR: 打ち切りまでの最大繰り返し回数を設定します。サブルーチンが終了すると、収束したときの繰り返し回数が戻ります。
- IER: サブルーチンが終了すると、エラーコードが戻ります。IER = 0 ならば正常終了、IER = 1 ならば収束しなかったことを意味します。

```
      PROGRAM MAIN
      IMPLICIT REAL*8(A-H,O-Z)
      PARAMETER (MX=6,MY=6)
      DIMENSION A(0:MX+1,0:MY+1,3),B(MX,MY)
      DIMENSION X(0:MX+1,0:MY+1)
      DIMENSION D(0:MX+1,0:MY+1),P(0:MX+1,0:MY+1)
      DIMENSION Q(0:MX+1,0:MY+1),R(0:MX+1,0:MY+1)
        :
      配列AとBに値を設定します。                    [1]
      EPS=1.0D-7                                    [2]
      ITR=1000                                      [2]
      CALL ICCG(A,MX,MY,B,EPS,ITR,X,D,P,Q,R,IER)[3]
        :
      END
```

Ⓐ

Ⓐ

```
      SUBROUTINE ICCG(A,MX,MY,B,EPS,ITR,
     &                X,D,P,Q,R,IER)
      IMPLICIT REAL*8(A-H,O-Z)
      DIMENSION A(0:MX+1,0:MY+1,3),B(MX,MY)
      DIMENSION X(0:MX+1,0:MY+1)
      DIMENSION D(0:MX+1,0:MY+1),P(0:MX+1,0:MY+1)
      DIMENSION Q(0:MX+1,0:MY+1),R(0:MX+1,0:MY+1)
      DO J = 0, MY+1
        DO I = 0, MX+1
          D(I,J) = 0.0D0
          X(I,J) = 0.0D0
          P(I,J) = 0.0D0                            ①
          Q(I,J) = 0.0D0
          R(I,J) = 0.0D0
        ENDDO
      ENDDO
      CALL DECOMP(A,D,MX,MY)                        ②
      CALL AXSUB(A,X,Q,MX,MY)                       ③
      DO J = 1, MY
        DO I = 1, MX
          R(I,J) = B(I,J) - Q(I,J)                  ④
        ENDDO
      ENDDO
      CALL LDLT(A,D,R,P,MX,MY)                      ⑤
      C1 = 0.0D0
      DO J = 1, MY
        DO I = 1, MX
          C1 = C1 + R(I,J)*P(I,J)                   ⑥
        ENDDO
      ENDDO
```

Ⓑ

Ⓑ

```
DO K = 1, ITR      収束反復ループ
  CALL AXSUB(A,P,Q,MX,MY)                    ⑦
  C2 = 0.0D0
  DO J = 1, MY
    DO I = 1, MX
      C2 = C2 + P(I,J)*Q(I,J)                ⑧
    ENDDO
  ENDDO
  ALPHA = C1 / C2                            ⑨
  X1 = 0.0D0
  X2 = 0.0D0
  DO J = 1, MY
    DO I = 1, MX
      Y = X(I,J)
      X(I,J) = X(I,J) + ALPHA*P(I,J)         ⑩
      R(I,J) = R(I,J) - ALPHA*Q(I,J)         ⑪
      X1 = X1 + Y*Y
      X2 = X2 + (X(I,J)-Y)**2
    ENDDO
  ENDDO
  IF (X1 /= 0.0) THEN
    RES = DSQRT(X2/X1)
    IF (RES <= EPS) THEN
      ITR = K
      EPS = RES                              ⑫
      IER = 0
      RETURN
    ENDIF
  ENDIF
  CALL LDLT(A,D,R,Q,MX,MY)                   ⑬
  C3 = 0.0D0
  DO J = 1, MY
    DO I = 1, MX
      C3 = C3 + R(I,J)*Q(I,J)                ⑭
    ENDDO
  ENDDO
  BETA = C3/C1                               ⑮
  C1 = C3                                    ⑯
  DO J = 1, MY
    DO I = 1, MX
      P(I,J) = Q(I,J) + BETA*P(I,J)          ⑰
    ENDDO
  ENDDO
ENDDO
IER = 1
EPS = RES
END
```

Ⓒ

```
SUBROUTINE DECOMP(A,D,MX,MY)
IMPLICIT REAL*8(A-H,O-Z)
DIMENSION A(0:MX+1,0:MY+1,3)
DIMENSION D(0:MX+1,0:MY+1)
DO J = 1, MY
  DO I = 1, MX
    D(I,J) = 1.0D0/( A(I,J,3)
&                 -D(I,J-1)*A(I,J,1)**2      3
&                 -D(I-1,J)*A(I,J,2)**2 )
  ENDDO
ENDDO
END
```

```
SUBROUTINE AXSUB(A,X,Y,MX,MY)
IMPLICIT REAL*8(A-H,O-Z)
DIMENSION A(0:MX+1,0:MY+1,3)
DIMENSION X(0:MX+1,0:MY+1),Y(0:MX+1,0:MY+1)
DO J = 1, MY
  DO I = 1, MX
    Y(I,J) = A(I,J,1)*X(I,J-1)
&          + A(I,J,2)*X(I-1,J)
&          + A(I,J,3)*X(I,J)                5
&          + A(I+1,J,2)*X(I+1,J)
&          + A(I,J+1,1)*X(I,J+1)
  ENDDO
ENDDO
END
```

```
SUBROUTINE LDLT(A,D,X,Y,MX,MY)
IMPLICIT REAL*8(A-H,O-Z)
DIMENSION A(0:MX+1,0:MY+1,3)
DIMENSION D(0:MX+1,0:MY+1)
DIMENSION X(0:MX+1,0:MY+1),Y(0:MX+1,0:MY+1)
DO J = 1, MY
  DO I = 1, MX
    Y(I,J) = D(I,J)*( X(I,J)
&                   -A(I,J,1)*Y(I,J-1)       10
&                   -A(I,J,2)*Y(I-1,J) )
  ENDDO
ENDDO
DO J = MY, 1, -1
  DO I = MX, 1, -1
    Y(I,J) = Y(I,J)-D(I,J)*(
&                    A(I+1,J,2)*Y(I+1,J)     11
&                   +A(I,J+1,1)*Y(I,J+1) )
  ENDDO
ENDDO
END
```

Ⓒ

図 5-4-1 (2)

行列 A として、本節では 2 次元差分法で現れる 5 重対角対称帯行列を想定します。例えば図 5-4-2 (1) のような計算領域の場合、行列 A は図 5-4-2 (4) のように（対称なので）3 本の帯 a_i，b_i，c_i から構成されます (紛らわしいですが、a_i，b_i，c_i は図 5-4-1 (1) の b や c とは異なることに注意して下さい)。

通常 ICCG 法のプログラムでは a_i，b_i，c_i や図 5-4-1 (1) の b，x，p，q，r を 1 次元ベクトルとして保持しますが、並列化する場合は、2 次元配列 (つまり図 5-4-2 (1) の計算領域と同じ形状) で扱った方が処理が簡単なので、以後これらのベクトルを 2 次元配列で表現することにします。このとき図 5-4-2 (2) のように要素の添字は 1 次元ベクトルのままで a_1，a_2，...と表すこともできますが、2 次元配列として a_{11}，a_{12}，...のように表した方が自然なので、図 5-4-2 (3) のように表すことにします。

プログラム内で使用している実際の配列名と構造を図 5-4-3 に示します。各配列 (B を除く) は計算領域よりも周囲が 1 要素分大きくなっていることに注意して下さい（理由は各サブルーチンでの計算を簡単にするためです)。図 5-4-1 (2) の [1] では、配列 A と B に値を設定しますが、図 5-4-3 の配列 A 内で『0』となっている部分には（配列の周囲の部分も含め）必ずゼロを入れて下さい。なお、図 5-4-1 (1) のその他のベクトル D，p，q，r は図 5-4-3 の配列 X と同じ構造をしています。

	1	2	3	MY
1	(1)	(5)	(9)	(13)
2	(2)	(6)	(10)	(14)
3	(3)	(7)	(11)	(15)
MX	(4)	(8)	(12)	(16)

図 5-4-2 (1)

	1	2	3	MY
1	a_1	a_5	a_9	a_{13}
2	a_2	a_6	a_{10}	a_{14}
3	a_3	a_7	a_{11}	a_{15}
MX	a_4	a_8	a_{12}	a_{16}

図 5-4-2 (2)

	1	2	3	MY
1	a_{11}	a_{12}	a_{13}	a_{14}
2	a_{21}	a_{22}	a_{23}	a_{24}
3	a_{31}	a_{32}	a_{33}	a_{34}
MX	a_{41}	a_{42}	a_{43}	a_{44}

図 5-4-2 (3)

MX×MY

A	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	x		b
↑ (1)	a_1	b_2			c_5												x_1		B_1
MX (2)	b_2	a_2	b_3			c_6											x_2		B_2
(3)		b_3	a_3	b_4			c_7										x_3		B_3
↓ (4)			b_4	a_4	0			c_8									x_4		B_4
(5)	c_5			0	a_5	b_6			c_9								x_5		B_5
(6)		c_6			b_6	a_6	b_7			c_{10}							x_6		B_6
(7)			c_7			b_7	a_7	b_8			c_{11}						x_7	=	B_7
(8)				c_8			b_8	a_8	0			c_{12}					x_8		B_8
(9)					c_9			0	a_9	b_{10}			c_{13}				x_9		B_9
(10)						c_{10}			b_{10}	a_{10}	b_{11}			c_{14}			x_{10}		B_{10}
(11)							c_{11}			b_{11}	a_{11}	b_{12}			c_{15}		x_{11}		B_{11}
(12)								c_{12}			b_{12}	a_{12}	0			c_{16}	x_{12}		B_{12}
(13)									c_{13}			0	a_{13}	b_{14}			x_{13}		B_{13}
(14)										c_{14}			b_{14}	a_{14}	b_{15}		x_{14}		B_{14}
(15)											c_{15}			b_{15}	a_{15}	b_{16}	x_{15}		B_{15}
MX×MY → (16)												c_{16}			b_{16}	a_{16}	x_{16}		B_{16}

図 5-4-2 (4)

A(I,J,1) (=c_i)

	0	1	2	3	MY	MY+1
0	0	0	0	0	0	0
1	0	0	c_{12}	c_{13}	c_{14}	0
2	0	0	c_{22}	c_{23}	c_{24}	0
3	0	0	c_{32}	c_{33}	c_{34}	0
MX	0	0	c_{42}	c_{43}	c_{44}	0
MX+1	0	0	0	0	0	0

A(I,J,2) (=b_i)

	0	1	2	3	MY	MY+1
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	b_{21}	b_{22}	b_{23}	b_{24}	0
3	0	b_{31}	b_{32}	b_{33}	b_{34}	0
MX	0	b_{41}	b_{42}	b_{43}	b_{44}	0
MX+1	0	0	0	0	0	0

A(I,J,3) (=a_i)

	0	1	2	3	MY	MY+1
0	0	0	0	0	0	0
1	0	a_{11}	a_{12}	a_{13}	a_{14}	0
2	0	a_{21}	a_{22}	a_{23}	a_{24}	0
3	0	a_{31}	a_{32}	a_{33}	a_{34}	0
MX	0	a_{41}	a_{42}	a_{43}	a_{44}	0
MX+1	0	0	0	0	0	0

X(I,J) (=x,D,p,q,r)

	0	1	2	3	MY	MY+1
0	0	0	0	0	0	0
1	0	x_{11}	x_{12}	x_{13}	x_{14}	0
2	0	x_{21}	x_{22}	x_{23}	x_{24}	0
3	0	x_{31}	x_{32}	x_{33}	x_{34}	0
MX	0	x_{41}	x_{42}	x_{43}	x_{44}	0
MX+1	0	0	0	0	0	0

B(I,J) (=ベクトルb)

	1	2	3	MY
1	B_{11}	B_{12}	B_{13}	B_{14}
2	B_{21}	B_{22}	B_{23}	B_{24}
3	B_{31}	B_{32}	B_{33}	B_{34}
MX	B_{41}	B_{42}	B_{43}	B_{44}

図 5-4-3

配列の分割方法

以後、サブルーチン ICCG 内の各部の並列化について説明します。並列化後のプログラムを図 5-4-10 に示します (図 5-4-1 (1) 内の番号と図 5-4-10 内の番号は対応しています)。以後、計算領域を 6×6 とし、図 5-4-3 の各配列を図 5-4-4 のように表現します。

まず配列の分割方法ですが、パイプライン法では 1 次元目または 2 次元目のどちらか一方でブロック分割します。本節では説明を簡単にするため配列の縮小は行っていません。このため通信の容易さから 2 次元目でブロック分割することにします。各プロセスの担当部分を図 5-4-4 に示します (図中の JSTA, JEND はランク 1 のプロセスの範囲を示します)。この分割を図 5-4-10 の [5] で行います。

実際のプログラムでは、配列の大きさが長方形の場合、境界の長さが短くなる（通信量が少なくなる）次元で分割するのがよいでしょう (4-5-8 節参照)。

図 5-4-4

ベクトル同志の演算と内積計算の並列化

図 5-4-1 (1) のうち、①，④，⑥，⑧，⑩，⑪，⑭，⑰はベクトル同志の演算や内積計算なので、容易に並列化することができます。このうち①以外の部分を並列化しました (図 5-4-10 の該当番号参照)。それに伴い⑥，⑧，⑩，⑪，⑭の内積（や合計の）計算の後、求めた部分和を [6]〜[9] で通信しながら合計し、結果を全プロセスに渡します。なお、3-3-6 節で説明したように、内積や合計の計算部分を並列化すると計算結果が若干変わります。

計算が収束したら最後に、各プロセスは、自分が求めた解ベクトル x の値を [10] で他の全プロセスに送信します。あらかじめ [4] で、各プロセスの担当部分の派生データ型を、3-5-3-2 節で説明した自作のサブルーチン PARA_TYPE_BLOCK2 で作成して配列 ITYPE に代入し (MPI-2 が使用できるマシン環境の場合は、MPI-2 で提供されている MPI_TYPE_CREATE_SUBARRAY (3-5-3-1 節参照) を使用して下さい)、4-6-3-2 節の方法で全プロセスに通信します。なお、[3] 以降の処理で解ベクトル x の値を例えばランク 0 のプロセスしか使用しないのであれば、4-6-3-1 節の方法でランク 0 に収集します。

サブルーチン DECOMP

本サブルーチンは図 5-4-1 (1) の②でコールされ、ICCG 法の収束性を高めるための前処理として図 5-4-5 (1) のように $A \quad LDL^{\mathrm{T}}$ の分解を行います。これを不完全 $\mathrm{LDL}^{\mathrm{T}}$ 分解と呼びます。分解の程度によって ICCG(1, 1)、ICCG(1, 2) などの方法があり (参考文献 [22] 参照)、ICCG 法の収束性に影響しますが、ここでは最も簡単な ICCG(1, 1) で分解します。分解された LDL^{T} を 4 本の帯 A_i, B_i, C_i, D_i で表すことにします。このとき $D_i = 1/A_i$ となるようにします。

図 5-4-5 (1)

ICCG(1, 1) の場合、行列 A の i 行目の 5 つの各要素と、$L \cdot D \cdot L^{\mathrm{T}}$ を実際に掛けた行列の同じ位置の要素のみが等しくなるようにします。ちなみにこれ以外の要素については両辺で異なっていてもよいので、この分解方法を不完全な $\mathrm{LDL}^{\mathrm{T}}$ 分解と呼びます。その結果以下の1の関係式が導かれます。この式から分かるように、分解後の B_i, C_i は分解前の b_i, c_i と一致するので実際には求める必要はありません。また1の 3 本目の式の A_i を左辺にし、$D_i = 1/A_i$, $B_i = b_i$, $C_i = c_i$ を使用して整理すると2式となり、これを前述のように 2 次元配列で表すと3式となります。本サブルーチンでは結局3式から対角要素 $D_{i,j}$ だけを求めます。

3式で配列 D に着目すると図 5-4-5 (2) のようになり、配列 D には「左と上の要素から自分を求める」という依存関係があるため完全な並列性はありませんが、このパターンは 4-6-7 節で説明したパイプライン法で並列化することができます。ただし図 5-4-1 (1) の②から分かるように、本サブルーチンは収束反復のループに入る前に 1 回しかコールされず通常ホットスポットにはならないので、今回は並列化を行わず、図 5-4-10 の3に示すように全プロセスが配列 D の全ての要素を求めるようにしました (もし並列化する場合は後述するサブルーチン LDLT と同じ処理になります)。なお、図 5-4-1 (2), 図 5-4-10 の□の番号と本文内の□の番号は対応しています。

余談ですが、図 5-4-5 (2) のうち着色した部分は上または左の要素からの依存関係がないので、本来は3式を場合分けする必要があります。ところが場合分けすると、サブルーチン LDLT をパイプライン法で並列化する際に処理が面倒になるので、ここでは図 5-4-5 (3) のように周囲の 1 要素分にゼロを入れ、全要素に対して3式をそのまま適用するようにしました。

1
$$\begin{cases} c_i = C_i \\ b_i = B_i \\ a_i = D_{i-1}B_i^2 + D_{i-m}C_i^2 + A_i \\ b_{i+1} = B_{i+1} \\ c_{i+m} = C_{i+m} \end{cases}$$

```
DO i = 1, n
  Di = 1/(ai - Di-m ci² - Di-1 bi²) ··· 2
ENDDO

DO j = 1, MY
  DO i = 1, MX
    Di,j = 1/(ai,j - Di,j-1 ci,j² - Di-1,j bi,j²) ··· 3
  ENDDO
ENDDO
```

図 5-4-5 (2)

図 5-4-5 (3)

サブルーチン AXSUB

本サブルーチンは図 5-4-1 (1) の③，⑦でコールされ、行列ベクトル積 $y = Ax$ を計算します。ここで x は解ベクトルではなく、x と y は任意のベクトルを表します。図 5-4-6 (1) の i 行目の関係式から4式が得られ、これを 2 次元配列で表すと5式となります。

図 5-4-6 (1)

```
DO i = 1, n
  y_i = c_i x_{i-m} + b_i x_{i-1} + a_i x_i + b_{i+1} x_{i+1} + c_{i+m} x_{i+m}  ··· 4
ENDDO
```

```
DO j = 1, MY
  DO i = 1, MX
    y_{i,j} = c_{i,j} x_{i,j-1} + b_{i,j} x_{i-1,j} + a_{i,j} x_{i,j} + b_{i+1,j} x_{i+1,j} + c_{i,j+1} x_{i,j+1}  ··· 5
  ENDDO
ENDDO
```

5式で配列 x と y に着目すると図 5-4-6 (2) のようになり、配列 y のある要素 Y は配列 x の同じ位置と上下左右の要素 X を使用して計算します。配列 y の要素間には依存関係がない（つまりどの順に計算してもよい）ので5式のループは並列化することができます。ただし、他の部分の並列化によって各プロセスは配列 x のうち自分の担当部分しか持っていないため、5式の計算に入る前に図 5-4-6 (3) に示す非循環シフトで境界の 1 列を隣のプロセスとの間で交換する必要があります (4-6-2 節参照)。これを図 5-4-10 の [11] で行います。なお、5式の a，b，c の各配列は全プロセスが全ての値を持っているので、計算前に通信を行う必要はありません。

図 5-4-6 (2)

図 5-4-6 (3)

サブルーチン LDLT

本サブルーチンは図 5-4-1 (1) の⑤，⑬でコールされ、$y = (LDL^{\mathrm{T}})^{-1}x$ の計算を行ないます。ここで x は解ベクトルではなく、x と y は任意のベクトルを表します。この演算を以下のように変形して作業ベクトル w を導入すると、前進消去と後退代入の 2 つのステップに分解されます。

$$y = (LDL^{\mathrm{T}})^{-1}x = (DL^{\mathrm{T}})^{-1}\underbrace{L^{-1}x}_{=w}$$

【前進消去】$w = L^{-1}x$ から w を求める。→ 実際には $x = Lw$ から w を求めます。
【後退代入】$y = (DL^{\mathrm{T}})^{-1}w$ から y を求める。→ 実際には $w = DL^{\mathrm{T}}y$ から y を求めます。
まず前進消去のステップでは、図 5-4-7 (1) に示すように i 行目の関係式は

$$x_i = C_i w_{i-m} + B_i w_{i-1} + A_i w_i$$

となります。w_i を左辺にし、$C_i = c_i$，$B_i = b_i$，$A_i = 1/D_i$ を使用して整理すると6式が得られます。
一方後退代入のステップでは、図 5-4-7 (2) に示すように i 行目の関係式は

$$w_i = A_i D_i y_i + B_{i+1} D_i y_{i+1} + C_{i+m} D_i y_{i+m}$$

となります。y_i を左辺にし、$C_i = c_i$，$B_i = b_i$，$A_i = 1/D_i$ を使用して整理すると7式が得られます。
実際のプログラムでは、6，7式の作業ベクトル w_i を、最終結果を入れるベクトル y_i で代用し、8，9式で計算します。これを 2 次元配列で表すと10，11式のようになります。

図 5-4-7 (1)

図 5-4-7 (2)

```
DO i = 1, n
  wi = Di･(xi - ciwi-m - biwi-1)        ･･･ 6
ENDDO
DO i = n, 1, -1
  yi = wi - Di･(bi+1yi+1 + ci+myi+m)   ･･･ 7
ENDDO
```

```
DO i = 1, n
  yi = Di･(xi - ciyi-m - biyi-1)        ･･･ 8
ENDDO
DO i = n, 1, -1
  yi = yi - Di･(bi+1yi+1 + ci+myi+m)   ･･･ 9
ENDDO
```

```
DO j = 1, MY
  DO i = 1, MX
    yi,j = Di,j･(xi,j - ci,jyi,j-1 - bi,jyi-1,j)        ･･･ 10
  ENDDO
ENDDO
DO j = MY, 1, -1
  DO i = MX, 1, -1
    yi,j = yi,j - Di,j･(bi+1,jyi+1,j + ci,j+1yi,j+1)   ･･･ 11
  ENDDO
ENDDO
```

図 5-4-8 (1)　　図 5-4-8 (2)

前進消去を行う10式で配列 y に着目すると図 5-4-8 (1) のようになり、配列 y には「左と上の要素から自分を求める」という依存関係があるため完全な並列性はありませんが、このパターンは 4-6-7 節で説明したパイプライン法で並列化することができます。

まず配列 y を 1 次元方向に大きさ IBLOCK (図 5-4-8 (3) の ↕) のブロックに分割します。IBLOCK の大きさは図 5-4-10 の [12] で指定します（最適な大きさは試行錯誤で調整して下さい)。各プロセスの計算順序は図 5-4-8 (3) のようになり、□内の数字の順に処理を行い、○内の数字の順に通信を行います。これを図 5-4-10 の10と [14], [15] で行います (詳細は 4-6-7 節を参照して下さい)。[13] の IBLKLEN は処理するブロックの 1 次元方向の大きさで基本的には IBLOCK ですが、MX が IBLOCK で割りきれない場合、最後の半端な部分では IBLOCK より小さくなるのでそれを調整します。なお、10式の b, c, D の各配列は全プロセスが全ての値を持っており、配列 X は各プロセスが自分の担当部分の値を持っているので計算の前に通信する必要はありません。

一方後退代入を行う11式で配列 y に着目すると図 5-4-8 (2) のようになり、配列 y に図 5-4-8 (1) と逆の依存関係があるので、図 5-4-8 (4) のような順序で処理を行います。これを図 5-4-10 の11と [17], [18] で行います。[16] の IBLKLEN は前述の [13] と同じで、計算領域の 1 次元方向が『1』から開始している場合はこのようになりますが、例えば『2』から開始している場合は IBLKLEN = MIN(IBLOCK,II-1) となるので注意して下さい。

図 5-4-8 (3)

図 5-4-8 (4)

```
MODULE PARA
INCLUDE 'mpif.h'
PARAMETER(NCPU=3)
INTEGER NPROCS,MYRANK,JSTA,JEND
INTEGER ISTATUS(MPI_STATUS_SIZE)
INTEGER IUP,IDOWN,ITYPE(0:NCPU-1)
END
```

```
PROGRAM MAIN
USE PARA
IMPLICIT REAL*8(A-H,O-Z)
PARAMETER (MX=6,MY=6)
DIMENSION A(0:MX+1,0:MY+1,3),B(MX,MY)
DIMENSION X(0:MX+1,0:MY+1)
DIMENSION D(0:MX+1,0:MY+1),P(0:MX+1,0:MY+1)
DIMENSION Q(0:MX+1,0:MY+1),R(0:MX+1,0:MY+1)
CALL MPI_INIT(IERR)
CALL MPI_COMM_SIZE(～,NPROCS,IERR)
CALL MPI_COMM_RANK(～,MYRANK,IERR)
DO IRANK=0,NPROCS-1
  CALL PARA_RANGE(1,MY,NPROCS,IRANK,
&                 JSTA,JEND)
  CALL PARA_TYPE_BLOCK2(0,MX+1,0,0,MX+1,  [4]
&       JSTA,JEND,MPI_REAL8,ITYPE(IRANK))
ENDDO
CALL PARA_RANGE(1,MY,NPROCS,MYRANK,        [5]
&                 JSTA,JEND)               [5]
```

Ⓓ

```
IUP   = MYRANK+1
IDOWN = MYRANK-1
IF (IUP==NPROCS) IUP = MPI_PROC_NULL
IF (IDOWN==-1) IDOWN = MPI_PROC_NULL
  :
配列AとBに値を設定します。                  [1]
EPS=1.0D-7                                  [2]
ITR=1000                                    [2]
CALL ICCG(A,MX,MY,B,EPS,ITR,X,D,P,Q,R,IER)[3]
  :
CALL MPI_FINALIZE(IERR)
END
```

```
SUBROUTINE ICCG(A,MX,MY,B,EPS,ITR,
&               X,D,P,Q,R,IER)
USE PARA
IMPLICIT REAL*8(A-H,O-Z)
DIMENSION A(0:MX+1,0:MY+1,3),B(MX,MY)
DIMENSION X(0:MX+1,0:MY+1)
DIMENSION D(0:MX+1,0:MY+1),P(0:MX+1,0:MY+1)
DIMENSION Q(0:MX+1,0:MY+1),R(0:MX+1,0:MY+1)
DIMENSION WORKS(2),WORKR(2)
DO J = 0, MY+1
  DO I = 0, MX+1
    D(I,J) = 0.0D0
    X(I,J) = 0.0D0
    P(I,J) = 0.0D0                          ①
    Q(I,J) = 0.0D0
    R(I,J) = 0.0D0
  ENDDO
ENDDO
```

Ⓔ

```
                                                          Ⓔ
CALL DECOMP(A,D,MX,MY)                                    ②
CALL AXSUB(A,X,Q,MX,MY)                                   ③
DO J = JSTA,JEND
  DO I = 1, MX
    R(I,J) = B(I,J) - Q(I,J)                              ④
  ENDDO
ENDDO
CALL LDLT(A,D,R,P,MX,MY)                                  ⑤
CC1 = 0.0D0
DO J = JSTA,JEND
  DO I = 1, MX
    CC1 = CC1 + R(I,J)*P(I,J)                             ⑥
  ENDDO
ENDDO
CALL MPI_ALLREDUCE(CC1,C1,1,MPI_REAL8,                    [6]
&          MPI_SUM,MPI_COMM_WORLD,IERR)                   [6]
DO K = 1, ITR     収束反復ループ
  CALL AXSUB(A,P,Q,MX,MY)                                 ⑦
  CC2 = 0.0D0
  DO J = JSTA,JEND
    DO I = 1, MX
      CC2 = CC2 + P(I,J)*Q(I,J)                           ⑧
    ENDDO
  ENDDO
  CALL MPI_ALLREDUCE(CC2,C2,1,MPI_REAL8,                  [7]
&            MPI_SUM,MPI_COMM_WORLD,IERR)                 [7]
  ALPHA = C1 / C2                                         ⑨
  X1 = 0.0D0
  X2 = 0.0D0
  DO J = JSTA,JEND
    DO I = 1, MX
      Y = X(I,J)
      X(I,J) = X(I,J) + ALPHA*P(I,J)                      ⑩
      R(I,J) = R(I,J) - ALPHA*Q(I,J)                      ⑪
      X1 = X1 + Y*Y
      X2 = X2 + (X(I,J)-Y)**2
    ENDDO
  ENDDO
                                                          Ⓕ
```

```
                                                          Ⓕ
  WORKS(1) = X1
  WORKS(2) = X2
  CALL MPI_ALLREDUCE(WORKS,WORKR,2,                       [8]
&    MPI_REAL8,MPI_SUM,MPI_COMM_WORLD,IERR)
  X1 = WORKR(1)
  X2 = WORKR(2)
  IF (X1 /= 0.0) THEN
    RES = DSQRT(X2/X1)
    IF (RES <= EPS) THEN
      ITR = K
      IER = 0                                             ⑫
      GOTO 999
    ENDIF
  ENDIF
  CALL LDLT(A,D,R,Q,MX,MY)                                ⑬
  CC3 = 0.0D0
  DO J = JSTA,JEND
    DO I = 1, MX
      CC3 = CC3 + R(I,J)*Q(I,J)                           ⑭
    ENDDO
  ENDDO
  CALL MPI_ALLREDUCE(CC3,C3,1,MPI_REAL8,                  [9]
&            MPI_SUM,MPI_COMM_WORLD,IERR)                 [9]
  BETA = C3/C1                                            ⑮
  C1 = C3                                                 ⑯
  DO J = JSTA,JEND
    DO I = 1, MX
      P(I,J) = Q(I,J) + BETA*P(I,J)                       ⑰
    ENDDO
  ENDDO
ENDDO
IER = 1
999 CONTINUE
  DO IRANK=0,NPROCS-1
    CALL MPI_BCAST(X,1,ITYPE(IRANK),IRANK,
&          MPI_COMM_WORLD,IERR)                           [10]
  ENDDO
  EPS = RES
  END
                                                          Ⓖ
```

```
SUBROUTINE DECOMP(A,D,MX,MY)
USE PARA
IMPLICIT REAL*8(A-H,O-Z)
DIMENSION A(0:MX+1,0:MY+1,3)
DIMENSION D(0:MX+1,0:MY+1)
DO J = 1, MY
  DO I = 1, MX
    D(I,J) = 1.0D0/( A(I,J,3)
&                 -D(I,J-1)*A(I,J,1)**2   3
&                 -D(I-1,J)*A(I,J,2)**2 )
  ENDDO
ENDDO
END
```

```
SUBROUTINE AXSUB(A,X,Y,MX,MY)
USE PARA
IMPLICIT REAL*8(A-H,O-Z)
DIMENSION A(0:MX+1,0:MY+1,3)
DIMENSION X(0:MX+1,0:MY+1),Y(0:MX+1,0:MY+1)
CALL MPI_ISEND(X(1,JEND),MX,MPI_REAL8,
&     IUP  ,1,MPI_COMM_WORLD,ISEND1,IERR)
CALL MPI_ISEND(X(1,JSTA),MX,MPI_REAL8,
&     IDOWN,1,MPI_COMM_WORLD,ISEND2,IERR)
CALL MPI_IRECV(X(1,JSTA-1),MX,MPI_REAL8,
&     IDOWN,1,MPI_COMM_WORLD,IRECV1,IERR)  11
CALL MPI_IRECV(X(1,JEND+1),MX,MPI_REAL8,
&     IUP  ,1,MPI_COMM_WORLD,IRECV2,IERR)
CALL MPI_WAIT(ISEND1,ISTATUS,IERR)
CALL MPI_WAIT(ISEND2,ISTATUS,IERR)
CALL MPI_WAIT(IRECV1,ISTATUS,IERR)
CALL MPI_WAIT(IRECV2,ISTATUS,IERR)
DO J = JSTA,JEND
  DO I = 1, MX
    Y(I,J) = A(I,J,1)*X(I,J-1)
&          + A(I,J,2)*X(I-1,J)
&          + A(I,J,3)*X(I,J)          5
&          + A(I+1,J,2)*X(I+1,J)
&          + A(I,J+1,1)*X(I,J+1)
  ENDDO
ENDDO
END
```

```
SUBROUTINE LDLT(A,D,X,Y,MX,MY)
USE PARA
IMPLICIT REAL*8(A-H,O-Z)
DIMENSION A(0:MX+1,0:MY+1,3)
DIMENSION D(0:MX+1,0:MY+1)
DIMENSION X(0:MX+1,0:MY+1),Y(0:MX+1,0:MY+1)
IBLOCK=2                                   12
DO II=1,MX,IBLOCK
  IBLKLEN = MIN(IBLOCK,MX-II+1)            13
  CALL MPI_IRECV(Y(II,JSTA-1),IBLKLEN,
&          MPI_REAL8,IDOWN,1,
&          MPI_COMM_WORLD,IREQR,IERR)      14
  CALL MPI_WAIT(IREQR,ISTATUS,IERR)
  DO J=JSTA,JEND
    DO I=II,II+IBLKLEN-1
      Y(I,J) = D(I,J)*( X(I,J)
&                  -A(I,J,1)*Y(I,J-1)      10
&                  -A(I,J,2)*Y(I-1,J) )
    ENDDO
  ENDDO
  CALL MPI_ISEND(Y(II,JEND),IBLKLEN,
&          MPI_REAL8,IUP,1,
&          MPI_COMM_WORLD,IREQS,IERR)      15
  CALL MPI_WAIT(IREQS,ISTATUS,IERR)
ENDDO
DO II=MX,1,-IBLOCK
  IBLKLEN = MIN(IBLOCK,II)                 16
  III = II-IBLKLEN+1
  CALL MPI_IRECV(Y(III,JEND+1),IBLKLEN,
&          MPI_REAL8,IUP,1,
&          MPI_COMM_WORLD,IREQR,IERR)      17
  CALL MPI_WAIT(IREQR,ISTATUS,IERR)
  DO J=JEND,JSTA,-1
    DO I=II,II-IBLKLEN+1,-1
      Y(I,J) = Y(I,J)-D(I,J)*(
&              A(I+1,J,2)*Y(I+1,J)         11
&             +A(I,J+1,1)*Y(I,J+1) )
    ENDDO
  ENDDO
  CALL MPI_ISEND(Y(III,JSTA),IBLKLEN,
&          MPI_REAL8,IDOWN,1,
&          MPI_COMM_WORLD,IREQS,IERR)      18
  CALL MPI_WAIT(IREQS,ISTATUS,IERR)
ENDDO
END
```

図 5-4-10

5-4-2　パラレル・ブロック・オーダリング法による並列化

本節では対称帯行列の ICCG 法を並列化する別の方法として、参考文献 [42] [39] で紹介されているパラレル・ブロック・オーダリング法 (以後 PBO 法) について説明します。

配列の分割方法

PBO 法でもパイプライン法の場合と同様に、図 5-4-1 (1) の各ベクトルを 2 次元配列（計算領域と同じ形状）として取り扱います。以後、計算領域として図 5-4-11 (1) の 12 × 12 の 2 次元配列を想定します (パイプライン法の場合と同じく、配列の周囲が 1 要素分大きくなっています)。PBO 法では並列化のための分割方法として、計算領域を 1 次元目と 2 次元目の両方でブロック分割します (5-1-3 節参照)。図 5-4-11 (1) の例では 1 次元目を 3 個、2 次元目を 3 個の計 9 プロセスで分割し、各プロセスは 4 × 4 の領域（以後ブロック）を担当します。

PBO 法の並列版のプログラムを図 5-4-16 に示します (図 5-4-1 (1) 内の番号と図 5-4-16 内の番号は対応しています)。まず [1] で 1 次元目と 2 次元目のプロセス数である IPROCS と JPROCS を指定し、次に各プロセスの上下および左右のプロセスのランクを知るため、[2] と [3] で図 5-4-11 (2) のようなテーブル ITABLE を作成し、[4] で自分の 1 次元と 2 次元方向のランクを MYRANKI, MYRANKJ に設定します。ランク 4 のプロセスの MYRANKI と MYRANKJ を図 5-4-11 (2) に示します。次に [5] で、各プロセスが担当する 1 次元方向と 2 次元方向の担当範囲を ISTA, IEND, JSTA, JEND に求めます (ランク 4 のプロセスの担当範囲を図 5-4-11 (1) に示します)。なお、説明を簡単にするため、配列の縮小は行なっていません。

図 5-4-11 (1)

図 5-4-11 (2)

ベクトル同志の演算と内積計算の並列化

図 5-4-1 (1) の④，⑥，⑧，⑩，⑪，⑭，⑰の並列化にともなって必要になる [6]〜[9] の通信と、最後に各プロセスが求めた解ベクトル x の値を全プロセスに送信する [10]，[11] の通信は、パイプライン法の場合と同じなので説明は省略します。

なお、[11] では 3-5-3-2 節で説明した自作のサブルーチン PARA_TYPE_BLOCK2 を使用しましたが、MPI-2 が使用できるマシン環境の場合は、MPI-2 で提供されている MPI_TYPE_CREATE_SUBARRAY (3-5-3-1 節参照) を使用して下さい。

サブルーチン DECOMP

PBO 法では、例えば図 5-4-12 (1) の計算領域を図 5-4-12 (2) のようにブロック分割した場合、要素を図 5-4-12 (2) 内の数字の順に並べて図 5-4-11 (3) の連立一次方程式を作成します (通常の ICCG 法の図 5-4-2 (4) に相当)。この連立一次方程式から前節の3，5，10，11式に相当する関係式を求めることができますが、導出過程は前節と同様なので省略し、以後の説明では導出した結果をいきなり使用します。

	1	2	3	MY
1	(1)	(5)	(9)	(13)
2	(2)	(6)	(10)	(14)
3	(3)	(7)	(11)	(15)
MX	(4)	(8)	(12)	(16)

図 5-4-12 (1)

	ランク0		ランク1	
	1	2	3	MY
1	1	9	3	11
2	5	13	7	15
3	2	10	4	12
MX	6	14	8	16
	ランク2		ランク3	

図 5-4-12 (2)

	(1)	(3)	(9)	(11)	(2)	(4)	(10)	(12)	(5)	(7)	(13)	(15)	(6)	(8)	(14)	(16)
(1)	a_1				b_2				c_5							
(3)		a_3			b_3	b_4				c_7						
(9)			a_9				b_{10}		c_9		c_{13}					
(11)				a_{11}			b_{11}	b_{12}		c_{11}		c_{15}				
(2)	b_2	b_3			a_2								c_6			
(4)		b_4				a_4								c_8		
(10)			b_{10}	b_{11}			a_{10}						c_{10}		c_{14}	
(12)				b_{12}				a_{12}						c_{12}		c_{16}
(5)	c_5		c_9						a_5				b_6			
(7)		c_7		c_{11}						a_7			b_7	b_8		
(13)			c_{13}								a_{13}				b_{14}	
(15)				c_{15}								a_{15}			b_{15}	b_{16}
(6)					c_6		c_{10}		b_6	b_7			a_6			
(8)						c_8		c_{12}		b_8				a_8		
(14)							c_{14}				b_{14}	b_{15}			a_{14}	
(16)								c_{16}				b_{16}				a_{16}

図 5-4-12 (3)

本サブルーチンでは $A \quad LDL^{\mathrm{T}}$ を行いますが、通常の ICCG 法と同様に実際には対角要素 $D_{i,j}$ のみを求めます。通常の ICCG 法では図 5-4-5 (2) のように「左と上の要素から自分を求める」という依存関係がありました。PBO 法でも図 5-4-13 (1) のように、基本的には「左と上の要素から自分を求める」という依存関係がありますが、各プロックの下の 1 行 (7, 8, 13) では下の要素からの依存関係 (↑) が、右の 1 列 (9, 10, 13) では右の要素からの依存関係 (←) があります。図中の□は同じ依存関係の要素を示します。このような依存関係があるので、□の数字の小さい要素から順に計算を行います (勿論、図からわかるように例えば2と3はどちらを先に計算してもかまいません)。

図 5-4-13 (1)

各要素の具体的な計算方法をまとめると図 5-4-13 (2) のようになります。基本的には前節の3式と同じですが、依存関係によって項の数が変わります。例えば13の要素の依存関係を (→ ↓ ↑ ←) に示しますが、このとき下線部の →, ↓, ↑, ← の項が必要になります。

1 ( )　$D_{i,j} = 1/(a_{i,j})$

2 (↓)　$D_{i,j} = 1/(a_{i,j} - D_{i-1,j}b_{i,j}^2)$

3 (→)　$D_{i,j} = 1/(a_{i,j} - D_{i,j-1}c_{i,j}^2)$

4 (→↓)　$D_{i,j} = 1/(a_{i,j} - D_{i,j-1}c_{i,j}^2 - D_{i-1,j}b_{i,j}^2)$

7 (↓↑)　$D_{i,j} = 1/(a_{i,j} - D_{i-1,j}b_{i,j}^2 - D_{i+1,j}b_{i+1,j}^2)$

8 (→↓↑)　$D_{i,j} = 1/(a_{i,j} - D_{i,j-1}c_{i,j}^2 - D_{i-1,j}b_{i,j}^2 - D_{i+1,j}b_{i+1,j}^2)$

9 (→←)　$D_{i,j} = 1/(a_{i,j} - D_{i,j-1}c_{i,j}^2 - D_{i,j+1}c_{i,j+1}^2)$

10 (→↓←)　$D_{i,j} = 1/(a_{i,j} - D_{i,j-1}c_{i,j}^2 - D_{i-1,j}b_{i,j}^2 - D_{i,j+1}c_{i,j+1}^2)$

13 (→↓↑←)　$D_{i,j} = 1/(a_{i,j} - \underbrace{D_{i,j-1}c_{i,j}^2}_{\rightarrow} - \underbrace{D_{i-1,j}b_{i,j}^2}_{\downarrow} - \underbrace{D_{i+1,j}b_{i+1,j}^2}_{\uparrow} - \underbrace{D_{i,j+1}c_{i,j+1}^2}_{\leftarrow})$

図 5-4-13 (2)

次に並列化した場合の動作について図 5-4-13 (3) で説明します。図 5-4-16 では配列の縮小を行っていませんが、図 5-4-13 (3) では紙面の関係で各プロセスが担当する部分の周辺のみを示します（図の周囲についている座標番号に注意して下さい）。例えば図 5-4-13 (3) でランク 4 のプロセスに着目すると、処理は以下のようになります。実際のプログラムは図 5-4-16 のサブルーチン DECOMP の対応する□を参照して下さい。

- 1〜4の要素をこの順に処理します。
- 5と6の要素を左と上のプロセスに対して送信し、右と下のプロセスから受信します。これによって7〜10の要素の処理が可能になります。
- 7〜10の要素をこの順に処理します。
- 11と12の要素を左と上のプロセスに対して送信し、右と下のプロセスから受信します。これによって13の要素の処理が可能になります。
- 13の要素を処理します。

図 5-4-13 (3)

サブルーチン AXSUB

本サブルーチンの計算方法は前節の5式と同じで以下のようになります。PBO 法では計算領域を 1 次元目と 2 次元目の両方で分割しているため、図 5-4-6 (3) に対応する境界のデータの通信は、図 5-4-14 の矢印のように上下左右のプロセスに対して行います。計算部分と通信部分を図 5-4-16 の28, 27に示します。

```
DO j = 1, MY
  DO i = 1, MX
    y_{i,j} = c_{i,j}x_{i,j-1} + b_{i,j}x_{i-1,j} + a_{i,j}x_{i,j} + b_{i+1,j}x_{i+1,j} + c_{i,j+1}x_{i,j+1}   ･･･ 28
  ENDDO
ENDDO
```

図 5-4-14

サブルーチン LDLT

前節で説明したように、本サブルーチンには前進消去と後退代入の 2 つのステップがあり、このうち前進消去のステップでの並列化の方法はサブルーチン DECOMP と全く同じです。図 5-4-8 (1) に対応する要素間の依存関係は前述の図 5-4-13 (1) のようになり、それを並列化した場合の動作は図 5-4-13 (3) のようになります。各要素の具体的な計算方法は図 5-4-15 (1) のようになります。実際のプログラムは図 5-4-16 のサブルーチン LDLT の対応する□を参照して下さい。

【前進消去】

1	( )	$y_{i,j} = D_{i,j}\cdot(x_{i,j})$
2	(↓)	$y_{i,j} = D_{i,j}\cdot(x_{i,j} - b_{i,j}y_{i-1,j})$
3	(→)	$y_{i,j} = D_{i,j}\cdot(x_{i,j} - c_{i,j}y_{i,j-1})$
4	(→↓)	$y_{i,j} = D_{i,j}\cdot(x_{i,j} - c_{i,j}y_{i,j-1} - b_{i,j}y_{i-1,j})$
7	(↓⬆)	$y_{i,j} = D_{i,j}\cdot(x_{i,j} - b_{i,j}y_{i-1,j} - b_{i+1,j}y_{i+1,j})$
8	(→↓⬆)	$y_{i,j} = D_{i,j}\cdot(x_{i,j} - c_{i,j}y_{i,j-1} - b_{i,j}y_{i-1,j} - b_{i+1,j}y_{i+1,j})$
9	(→ ⬅)	$y_{i,j} = D_{i,j}\cdot(x_{i,j} - c_{i,j}y_{i,j-1} - c_{i,j+1}y_{i,j+1})$
10	(→↓ ⬅)	$y_{i,j} = D_{i,j}\cdot(x_{i,j} - c_{i,j}y_{i,j-1} - b_{i,j}y_{i-1,j} - c_{i,j+1}y_{i,j+1})$
13	(→↓⬆⬅)	$y_{i,j} = D_{i,j}\cdot(x_{i,j} - \underbrace{c_{i,j}y_{i,j-1}}_{\rightarrow} - \underbrace{b_{i,j}y_{i-1,j}}_{\downarrow} - \underbrace{b_{i+1,j}y_{i+1,j}}_{\uparrow} - \underbrace{c_{i,j+1}y_{i,j+1}}_{\leftarrow})$

図 5-4-15 (1)

後退代入の動作は前進消去と逆で、並列化した場合の動作は図 5-4-15 (3) のようになります。各要素の具体的な計算方法をまとめると図 5-4-15 (2) のようになります。実際のプログラムは図 5-4-16 のサブルーチン LDLT の対応する□を参照して下さい。

【後退代入】

14	( )	$y_{i,j} = y_{i,j} - D_{i,j}\cdot(\quad)$
15	(↑)	$y_{i,j} = y_{i,j} - D_{i,j}\cdot(+ b_{i+1,j}y_{i+1,j})$
16	(←)	$y_{i,j} = y_{i,j} - D_{i,j}\cdot(+ c_{i,j+1}y_{i,j+1})$
17	(↑←)	$y_{i,j} = y_{i,j} - D_{i,j}\cdot(+ b_{i+1,j}y_{i+1,j} + c_{i,j+1}y_{i,j+1})$
20	(⬇↑)	$y_{i,j} = y_{i,j} - D_{i,j}\cdot(+ b_{i,j}y_{i-1,j} + b_{i+1,j}y_{i+1,j})$
21	(⬇↑←)	$y_{i,j} = y_{i,j} - D_{i,j}\cdot(+ b_{i,j}y_{i-1,j} + b_{i+1,j}y_{i+1,j} + c_{i,j+1}y_{i,j+1})$
22	(➡ ←)	$y_{i,j} = y_{i,j} - D_{i,j}\cdot(c_{i,j}y_{i,j-1} + c_{i,j+1}y_{i,j+1})$
23	(➡ ↑←)	$y_{i,j} = y_{i,j} - D_{i,j}\cdot(c_{i,j}y_{i,j-1} + b_{i+1,j}y_{i+1,j} + c_{i,j+1}y_{i,j+1})$
26	(➡⬇↑←)	$y_{i,j} = y_{i,j} - D_{i,j}\cdot(\underbrace{c_{i,j}y_{i,j-1}}_{➡} + \underbrace{b_{i,j}y_{i-1,j}}_{⬇} + \underbrace{b_{i+1,j}y_{i+1,j}}_{\uparrow} + \underbrace{c_{i,j+1}y_{i,j+1}}_{\leftarrow})$

図 5-4-15 (2)

図 5-4-15 (3)

パイプライン法と PBO 法の比較

まず収束性について検討します。ICCG 法のような反復解法では、収束までの反復回数が少ない方が収束性が高くパフォーマンスが向上します。パイプライン法は単体版の ICCG 法の前処理をそのまま使用して並列化しているので単体版の ICCG 法と収束性は変わりませんが、PBO 法ではやや不自然な前処理をしているので収束性が若干低下します。また PBO 法では分割方法 (1 次元目と 2 次元目のプロセス数) によっても収束性が変化します。

次に速度向上率について検討します。パイプライン法は完全な並列化ではないので速度向上率はそれほど高くありませんが、PBO 法は完全な並列処理を行っているので連立一次方程式の規模が大きくなればかなり高い速度向上率を得ることができます。

```
MODULE PARA
INCLUDE 'mpif.h'
PARAMETER(IPROCS=3,JPROCS=3)                    [1]
INTEGER ITABLE(-1:IPROCS,-1:JPROCS)
INTEGER ITYPE(0:IPROCS*JPROCS-1)
INTEGER NPROCS,MYRANK,ISTA,IEND,JSTA,JEND
INTEGER ISTATUS(MPI_STATUS_SIZE)
INTEGER IUP,IDOWN,JUP,JDOWN
END
```

```
PROGRAM MAIN
USE PARA
IMPLICIT REAL*8(A-H,O-Z)
PARAMETER (MX=12,MY=12)
DIMENSION A(0:MX+1,0:MY+1,3),B(MX,MY),
DIMENSION X(0:MX+1,0:MY+1)
DIMENSION D(0:MX+1,0:MY+1),P(0:MX+1,0:MY+1)
DIMENSION Q(0:MX+1,0:MY+1),R(0:MX+1,0:MY+1)
DIMENSION BUFS1(MY),BUFR1(MY)
DIMENSION BUFS2(MY),BUFR2(MY)
CALL MPI_INIT(IERR)
CALL MPI_COMM_SIZE(～,NPROCS,IERR)
CALL MPI_COMM_RANK(～,MYRANK,IERR)
IF (NPROCS/=IPROCS*JPROCS) THEN
  IF (MYRANK==0) PRINT *,'===ERROR==='
  CALL MPI_FINALIZE(IERR)
  STOP
ENDIF
DO I=-1,IPROCS
  DO J=-1,JPROCS
    ITABLE(I,J) = MPI_PROC_NULL                 [2]
  ENDDO
ENDDO
IRANK = 0
DO I=0,IPROCS-1
  DO J=0,JPROCS-1
    ITABLE(I,J) = IRANK                         [3]
    IF (MYRANK == IRANK) THEN
      MYRANKI = I
      MYRANKJ = J                               [4]
    ENDIF
    CALL PARA_RANGE
        (1,MX,IPROCS,I,ISTA,IEND)
    CALL PARA_RANGE
        (1,MY,JPROCS,J,JSTA,JEND)               [11]
    CALL PARA_TYPE_BLOCK2(0,MX+1,0,ISTA,
&   IEND,JSTA,JEND,MPI_REAL8,ITYPE(IRANK))
    IRANK = IRANK + 1
  ENDDO
ENDDO
```

Ⓐ

Ⓐ

```
CALL PARA_RANGE(1,MX,IPROCS,MYRANKI,
&                ISTA,IEND)
CALL PARA_RANGE(1,MY,JPROCS,MYRANKJ,            [5]
&                JSTA,JEND)
JUP   = ITABLE(MYRANKI,MYRANKJ+1)
JDOWN = ITABLE(MYRANKI,MYRANKJ-1)
IUP   = ITABLE(MYRANKI+1,MYRANKJ)
IDOWN = ITABLE(MYRANKI-1,MYRANKJ)
  :
配列AとBに値を設定します。
EPS=1.0D-7
ITR=1000
CALL ICCG(A,MX,MY,B,EPS,ITR,X,D,P,Q,R,IER,
&          BUFS1,BUFR1,BUFS2,BUFR2)
  :
CALL MPI_FINALIZE(IERR)
END
```

```
SUBROUTINE ICCG(A,MX,MY,B,EPS,ITR,X,D,P,Q,R,
&           IER,BUFS1,BUFR1,BUFS2,BUFR2)
USE PARA
IMPLICIT REAL*8(A-H,O-Z)
DIMENSION A(0:MX+1,0:MY+1,3),B(MX,MY)
DIMENSION X(0:MX+1,0:MY+1)
DIMENSION D(0:MX+1,0:MY+1),P(0:MX+1,0:MY+1)
DIMENSION Q(0:MX+1,0:MY+1),R(0:MX+1,0:MY+1)
DIMENSION BUFS1(MY),BUFR1(MY)
DIMENSION BUFS2(MY),BUFR2(MY)
DIMENSION WORKS(2),WORKR(2)
DO J = 0, MY+1
  DO I = 0, MX+1
    D(I,J) = 0.0D0
    X(I,J) = 0.0D0
    P(I,J) = 0.0D0                              ①
    Q(I,J) = 0.0D0
    R(I,J) = 0.0D0
  ENDDO
ENDDO
CALL DECOMP(A,D,MX,MY,BUFS1,BUFR1)              ②
CALL AXSUB(A,X,Q,MX,MY,                         ③
&           BUFS1,BUFR1,BUFS2,BUFR2)
DO J = JSTA,JEND
  DO I = ISTA,IEND
    R(I,J) = B(I,J) - Q(I,J)                    ④
  ENDDO
ENDDO
CALL LDLT(A,D,R,P,MX,MY,BUFS1,BUFR1)            ⑤
```

Ⓑ

Ⓑ

```
      CC1 = 0.0D0
      DO J = JSTA,JEND
        DO I = ISTA,IEND
          CC1 = CC1 + R(I,J)*P(I,J)                ⑥
        ENDDO
      ENDDO
      CALL MPI_ALLREDUCE(CC1,C1,1,MPI_REAL8,       [6]
     &          MPI_SUM,MPI_COMM_WORLD,IERR)       [6]
      DO K = 1, ITR     収束反復ループ
        CALL AXSUB(A,P,Q,MX,MY,                    ⑦
     &             BUFS1,BUFR1,BUFS2,BUFR2)
        CC2 = 0.0D0
        DO J = JSTA,JEND
          DO I = ISTA,IEND
            CC2 = CC2 + P(I,J)*Q(I,J)              ⑧
          ENDDO
        ENDDO
        CALL MPI_ALLREDUCE(CC2,C2,1,MPI_REAL8,     [7]
     &            MPI_SUM,MPI_COMM_WORLD,IERR)     [7]
        ALPHA = C1 / C2                            ⑨
        X1 = 0.0D0
        X2 = 0.0D0
        DO J = JSTA,JEND
          DO I = ISTA,IEND
            Y = X(I,J)
            X(I,J) = X(I,J) + ALPHA*P(I,J)         ⑩
            R(I,J) = R(I,J) - ALPHA*Q(I,J)         ⑪
            X1 = X1 + Y*Y
            X2 = X2 + (X(I,J)-Y)**2
          ENDDO
        ENDDO
        WORKS(1) = X1
        WORKS(2) = X2
        CALL MPI_ALLREDUCE(WORKS,WORKR,2,          [8]
     &    MPI_REAL8,MPI_SUM,MPI_COMM_WORLD,IERR)
        X1 = WORKR(1)
        X2 = WORKR(2)
        IF (X1 /= 0.0) THEN
          RES = DSQRT(X2/X1)
          IF (RES <= EPS) THEN
            ITR = K
            IER = 0                                ⑫
            GOTO 999
          ENDIF
        ENDIF
        CALL LDLT(A,D,R,Q,MX,MY,BUFS1,BUFR1)       ⑬
```

Ⓒ

Ⓒ

```
        CC3 = 0.0D0
        DO J = JSTA,JEND
          DO I = ISTA,IEND
            CC3 = CC3 + R(I,J)*Q(I,J)              ⑭
          ENDDO
        ENDDO
        CALL MPI_ALLREDUCE(CC3,C3,1,MPI_REAL8,     [9]
     &            MPI_SUM,MPI_COMM_WORLD,IERR)     [9]
        BETA = C3/C1                               ⑮
        C1 = C3                                    ⑯
        DO J = JSTA,JEND
          DO I = ISTA,IEND
            P(I,J) = Q(I,J) + BETA*P(I,J)          ⑰
          ENDDO
        ENDDO
      ENDDO
      IER = 1
999   CONTINUE
      DO IRANK=0,NPROCS-1
        CALL MPI_BCAST(X,1,ITYPE(IRANK),IRANK,
     &                 MPI_COMM_WORLD,IERR)        [10]
      ENDDO
      EPS = RES
      END
      SUBROUTINE DECOMP(A,D,MX,MY,BUFS1,BUFR1)
      USE PARA
      IMPLICIT REAL*8(A-H,O-Z)
      DIMENSION A(0:MX+1,0:MY+1,3)
      DIMENSION D(0:MX+1,0:MY+1)
      DIMENSION BUFS1(MY),BUFR1(MY)
      I = ISTA
      J = JSTA
      D(I,J) = 1.0D0/A(I,J,3)                      1
      BUFS1(J) = D(I,J)
      DO I = ISTA+1,IEND-1
        D(I,J) = 1.0D0/( A(I,J,3)
     &          -D(I-1,J  )*A(I  ,J  ,2)**2 )      2
      ENDDO
      DO J = JSTA+1,JEND-1
        I = ISTA
        D(I,J) = 1.0D0/( A(I,J,3)
     &          -D(I  ,J-1)*A(I  ,J  ,1)**2 )      3
        BUFS1(J) = D(I,J)
        DO I = ISTA+1,IEND-1
          D(I,J) = 1.0D0/( A(I,J,3)
     &           -D(I  ,J-1)*A(I  ,J  ,1)**2       4
     &           -D(I-1,J  )*A(I  ,J  ,2)**2 )
        ENDDO
      ENDDO
```

Ⓓ

Ⓓ

```
 CALL MPI_ISEND(D(ISTA,JSTA),IEND-ISTA,          5
&         MPI_REAL8,JDOWN,1,
&         MPI_COMM_WORLD,ISEND1,IERR)
 CALL MPI_ISEND(BUFS1(JSTA) ,JEND-JSTA,          6
&         MPI_REAL8,IDOWN,1,
&         MPI_COMM_WORLD,ISEND2,IERR)
 CALL MPI_IRECV(D(ISTA,JEND+1),IEND-ISTA,        5
&         MPI_REAL8,JUP,1,
&         MPI_COMM_WORLD,IRECV1,IERR)
 CALL MPI_IRECV(BUFR1(JSTA)   ,JEND-JSTA,        6
&         MPI_REAL8,IUP,1,
&         MPI_COMM_WORLD,IRECV2,IERR)
 CALL MPI_WAIT(ISEND1,ISTATUS,IERR)
 CALL MPI_WAIT(ISEND2,ISTATUS,IERR)
 CALL MPI_WAIT(IRECV1,ISTATUS,IERR)
 CALL MPI_WAIT(IRECV2,ISTATUS,IERR)
 DO J=JSTA,JEND-1
   D(IEND+1,J) = BUFR1(J)
 ENDDO
 I = IEND
 J = JSTA
 D(I,J) = 1.0D0/( A(I,J,3)                       7
&        -D(I-1,J  )*A(I  ,J  ,2)**2
&        -D(I+1,J  )*A(I+1,J  ,2)**2 )
 DO J = JSTA+1,JEND-1
   D(I,J) = 1.0D0/( A(I,J,3)
&          -D(I  ,J-1)*A(I  ,J  ,1)**2           8
&          -D(I-1,J  )*A(I  ,J  ,2)**2
&          -D(I+1,J  )*A(I+1,J  ,2)**2 )
 ENDDO
 I = ISTA
 J = JEND
 D(I,J) = 1.0D0/( A(I,J,3)                       9
&        -D(I  ,J-1)*A(I  ,J  ,1)**2
&        -D(I  ,J+1)*A(I  ,J+1,1)**2 )
 DO I = ISTA+1,IEND-1
   D(I,J) = 1.0D0/( A(I,J,3)
&          -D(I  ,J-1)*A(I  ,J  ,1)**2
&          -D(I-1,J  )*A(I  ,J  ,2)**2           10
&          -D(I  ,J+1)*A(I  ,J+1,1)**2 )
 ENDDO
 CALL MPI_ISEND(D(IEND,JSTA),1,MPI_REAL8,        11
&     JDOWN,1,MPI_COMM_WORLD,ISEND1,IERR)
 CALL MPI_ISEND(D(ISTA,JEND),1,MPI_REAL8,        12
&     IDOWN,1,MPI_COMM_WORLD,ISEND2,IERR)
 CALL MPI_IRECV(D(IEND,JEND+1),1,MPI_REAL8,      11
&     JUP,1,MPI_COMM_WORLD,IRECV1,IERR)
 CALL MPI_IRECV(D(IEND+1,JEND),1,MPI_REAL8,      12
&     IUP,1,MPI_COMM_WORLD,IRECV2,IERR)
```

Ⓔ

Ⓔ

```
 CALL MPI_WAIT(ISEND1,ISTATUS,IERR)
 CALL MPI_WAIT(ISEND2,ISTATUS,IERR)
 CALL MPI_WAIT(IRECV1,ISTATUS,IERR)
 CALL MPI_WAIT(IRECV2,ISTATUS,IERR)
 I = IEND
 J = JEND
 D(I,J) = 1.0D0/( A(I,J,3)
&        -D(I  ,J-1)*A(I  ,J  ,1)**2            13
&        -D(I-1,J  )*A(I  ,J  ,2)**2
&        -D(I+1,J  )*A(I+1,J  ,2)**2
&        -D(I  ,J+1)*A(I  ,J+1,1)**2 )
 END
 SUBROUTINE AXSUB(A,X,Y,MX,MY,
&                 BUFS1,BUFR1,BUFS2,BUFR2)
 USE PARA
 IMPLICIT REAL*8(A-H,O-Z)
 DIMENSION A(0:MX+1,0:MY+1,3)
 DIMENSION X(0:MX+1,0:MY+1),Y(0:MX+1,0:MY+1)
 DIMENSION BUFS1(MY),BUFR1(MY)
 DIMENSION BUFS2(MY),BUFR2(MY)
 DO J=JSTA,JEND
   BUFS1(J) = X(ISTA,J)
   BUFS2(J) = X(IEND,J)
 ENDDO
 CALL MPI_ISEND(X(ISTA,JSTA),IEND-ISTA+1,
&         MPI_REAL8,JDOWN,1,
&         MPI_COMM_WORLD,ISEND1,IERR)
 CALL MPI_ISEND(X(ISTA,JEND),IEND-ISTA+1,
&         MPI_REAL8,JUP  ,1,
&         MPI_COMM_WORLD,ISEND2,IERR)
 CALL MPI_ISEND(BUFS1(JSTA) ,JEND-JSTA+1,
&         MPI_REAL8,IDOWN,1,
&         MPI_COMM_WORLD,ISEND3,IERR)
 CALL MPI_ISEND(BUFS2(JSTA) ,JEND-JSTA+1,
&         MPI_REAL8,IUP  ,1,
&         MPI_COMM_WORLD,ISEND4,IERR)
 CALL MPI_IRECV(X(ISTA,JEND+1),IEND-ISTA+1,
&         MPI_REAL8,JUP  ,1,
&         MPI_COMM_WORLD,IRECV1,IERR)
 CALL MPI_IRECV(X(ISTA,JSTA-1),IEND-ISTA+1,
&         MPI_REAL8,JDOWN,1,                     27
&         MPI_COMM_WORLD,IRECV2,IERR)
 CALL MPI_IRECV(BUFR1(JSTA)   ,JEND-JSTA+1,
&         MPI_REAL8,IUP  ,1,
&         MPI_COMM_WORLD,IRECV3,IERR)
 CALL MPI_IRECV(BUFR2(JSTA)   ,JEND-JSTA+1,
&         MPI_REAL8,IDOWN,1,
&         MPI_COMM_WORLD,IRECV4,IERR)
```

Ⓕ

```
 CALL MPI_WAIT(ISEND1,ISTATUS,IERR)
 CALL MPI_WAIT(ISEND2,ISTATUS,IERR)
 CALL MPI_WAIT(ISEND3,ISTATUS,IERR)
 CALL MPI_WAIT(ISEND4,ISTATUS,IERR)
 CALL MPI_WAIT(IRECV1,ISTATUS,IERR)
 CALL MPI_WAIT(IRECV2,ISTATUS,IERR)
 CALL MPI_WAIT(IRECV3,ISTATUS,IERR)
 CALL MPI_WAIT(IRECV4,ISTATUS,IERR)
 DO J=JSTA,JEND
   X(IEND+1,J) = BUFR1(J)
   X(ISTA-1,J) = BUFR2(J)
 ENDDO
 DO J = JSTA,JEND
   DO I = ISTA,IEND
     Y(I,J) = A(I,J,1)*X(I,J-1)
&          + A(I,J,2)*X(I-1,J)
&          + A(I,J,3)*X(I,J)                    28
&          + A(I+1,J,2)*X(I+1,J)
&          + A(I,J+1,1)*X(I,J+1)
   ENDDO
 ENDDO
 END
```

```
 SUBROUTINE LDLT(A,D,X,Y,MX,MY,BUFS1,BUFR1)
 USE PARA
 IMPLICIT REAL*8(A-H,O-Z)
 DIMENSION A(0:MX+1,0:MY+1,3)
 DIMENSION D(0:MX+1,0:MY+1)
 DIMENSION X(0:MX+1,0:MY+1),Y(0:MX+1,0:MY+1)
 DIMENSION BUFS1(MY),BUFR1(MY)
 I = ISTA
 J = JSTA
 Y(I,J) = D(I,J)*X(I,J)                         1
 BUFS1(J) = Y(I,J)
 DO I = ISTA+1,IEND-1
   Y(I,J) = D(I,J)*( X(I,J)
&          -A(I  ,J  ,2)*Y(I-1,J  ) )           2
 ENDDO
 DO J = JSTA+1,JEND-1
   I = ISTA
   Y(I,J) = D(I,J)*( X(I,J)
&          -A(I  ,J  ,1)*Y(I  ,J-1) )           3
   BUFS1(J) = Y(I,J)
   DO I = ISTA+1,IEND-1
     Y(I,J) = D(I,J)*( X(I,J)
&          -A(I  ,J  ,1)*Y(I  ,J-1)             4
&          -A(I  ,J  ,2)*Y(I-1,J  ) )
   ENDDO
 ENDDO
```

```
 CALL MPI_ISEND(Y(ISTA,JSTA),IEND-ISTA,          5
&        MPI_REAL8,JDOWN,1,
&        MPI_COMM_WORLD,ISEND1,IERR)
 CALL MPI_ISEND(BUFS1(JSTA) ,JEND-JSTA,          6
&        MPI_REAL8,IDOWN,1,
&        MPI_COMM_WORLD,ISEND2,IERR)
 CALL MPI_IRECV(Y(ISTA,JEND+1),IEND-ISTA,        5
&        MPI_REAL8,JUP,1,
&        MPI_COMM_WORLD,IRECV1,IERR)
 CALL MPI_IRECV(BUFR1(JSTA)   ,JEND-JSTA,        6
&        MPI_REAL8,IUP,1,
&        MPI_COMM_WORLD,IRECV2,IERR)
 CALL MPI_WAIT(ISEND1,ISTATUS,IERR)
 CALL MPI_WAIT(ISEND2,ISTATUS,IERR)
 CALL MPI_WAIT(IRECV1,ISTATUS,IERR)
 CALL MPI_WAIT(IRECV2,ISTATUS,IERR)
 DO J=JSTA,JEND-1
   Y(IEND+1,J) = BUFR1(J)
 ENDDO
 I = IEND
 J = JSTA
 Y(I,J) = D(I,J)*( X(I,J)                        7
&        -A(I  ,J  ,2)*Y(I-1,J  )
&        -A(I+1,J  ,2)*Y(I+1,J  ) )
 DO J = JSTA+1,JEND-1
   Y(I,J) = D(I,J)*( X(I,J)
&        -A(I  ,J  ,1)*Y(I  ,J-1)
&        -A(I  ,J  ,2)*Y(I-1,J  )                8
&        -A(I+1,J  ,2)*Y(I+1,J  ) )
 ENDDO
 I = ISTA
 J = JEND
 Y(I,J) = D(I,J)*( X(I,J)                        9
&        -A(I  ,J  ,1)*Y(I  ,J-1)
&        -A(I  ,J+1,1)*Y(I  ,J+1) )
 DO I = ISTA+1,IEND-1
   Y(I,J) = D(I,J)*( X(I,J)
&          -A(I  ,J  ,1)*Y(I  ,J-1)              10
&          -A(I  ,J  ,2)*Y(I-1,J  )
&          -A(I  ,J+1,1)*Y(I  ,J+1) )
 ENDDO
 CALL MPI_ISEND(Y(IEND,JSTA),1,MPI_REAL8,        11
&     JDOWN,1,MPI_COMM_WORLD,ISEND1,IERR)
 CALL MPI_ISEND(Y(ISTA,JEND),1,MPI_REAL8,        12
&     IDOWN,1,MPI_COMM_WORLD,ISEND2,IERR)
 CALL MPI_IRECV(Y(IEND,JEND+1),1,MPI_REAL8,      11
&     JUP,1,MPI_COMM_WORLD,IRECV1,IERR)
 CALL MPI_IRECV(Y(IEND+1,JEND),1,MPI_REAL8,      12
&     IUP,1,MPI_COMM_WORLD,IRECV2,IERR)
```

```
                                   Ⓗ
 CALL MPI_WAIT(ISEND1,ISTATUS,IERR)
 CALL MPI_WAIT(ISEND2,ISTATUS,IERR)
 CALL MPI_WAIT(IRECV1,ISTATUS,IERR)
 CALL MPI_WAIT(IRECV2,ISTATUS,IERR)
 I = IEND
 J = JEND
 Y(I,J) = D(I,J)*( X(I,J)
&        -A(I  ,J  ,1)*Y(I  ,J-1)                13
&        -A(I  ,J  ,2)*Y(I-1,J  )
&        -A(I+1,J  ,2)*Y(I+1,J  )
&        -A(I  ,J+1,1)*Y(I  ,J+1) )
 I = IEND
 J = JEND                                          14
 BUFS1(J) = Y(I,J)
 DO I = IEND-1,ISTA+1,-1
   Y(I,J) = Y(I,J)-D(I,J)*(                        15
&          +A(I+1,J  ,2)*Y(I+1,J  ) )
 ENDDO
 DO J = JEND-1,JSTA+1,-1
   I = IEND
   Y(I,J) = Y(I,J)-D(I,J)*(                        16
&          +A(I  ,J+1,1)*Y(I  ,J+1) )
   BUFS1(J) = Y(I,J)
   DO I = IEND-1,ISTA+1,-1
     Y(I,J) = Y(I,J)-D(I,J)*(
&            +A(I+1,J  ,2)*Y(I+1,J  )              17
&            +A(I  ,J+1,1)*Y(I  ,J+1) )
   ENDDO
 ENDDO
 CALL MPI_ISEND(Y(ISTA+1,JEND),IEND-ISTA,          18
&         MPI_REAL8,JUP,1,
&         MPI_COMM_WORLD,ISEND1,IERR)
 CALL MPI_ISEND(BUFS1(JSTA+1) ,JEND-JSTA,          19
&         MPI_REAL8,IUP,1,
&         MPI_COMM_WORLD,ISEND2,IERR)
 CALL MPI_IRECV(Y(ISTA+1,JSTA-1),IEND-ISTA,        18
&         MPI_REAL8,JDOWN,1,
&         MPI_COMM_WORLD,IRECV1,IERR)
 CALL MPI_IRECV(BUFR1(JSTA+1)   ,JEND-JSTA,        19
&         MPI_REAL8,IDOWN,1,
&         MPI_COMM_WORLD,IRECV2,IERR)
 CALL MPI_WAIT(ISEND1,ISTATUS,IERR)
 CALL MPI_WAIT(ISEND2,ISTATUS,IERR)
 CALL MPI_WAIT(IRECV1,ISTATUS,IERR)
 CALL MPI_WAIT(IRECV2,ISTATUS,IERR)
 DO J=JSTA+1,JEND
   Y(ISTA-1,J) = BUFR1(J)
 ENDDO
                                   Ⓘ
```

```
                                   Ⓘ
 I = ISTA
 J = JEND
 Y(I,J) = Y(I,J)-D(I,J)*(                          20
&        +A(I  ,J  ,2)*Y(I-1,J  )
&        +A(I+1,J  ,2)*Y(I+1,J  ) )
 DO J = JEND-1,JSTA+1,-1
   Y(I,J) = Y(I,J)-D(I,J)*(
&          +A(I  ,J  ,2)*Y(I-1,J  )
&          +A(I+1,J  ,2)*Y(I+1,J  )                21
&          +A(I  ,J+1,1)*Y(I  ,J+1) )
 ENDDO
 I = IEND
 J = JSTA
 Y(I,J) = Y(I,J)-D(I,J)*(                          22
&         A(I  ,J  ,1)*Y(I  ,J-1)
&        +A(I  ,J+1,1)*Y(I  ,J+1) )
 DO I = IEND-1,ISTA+1,-1
   Y(I,J) = Y(I,J)-D(I,J)*(
&           A(I  ,J  ,1)*Y(I  ,J-1)
&          +A(I+1,J  ,2)*Y(I+1,J  )                23
&          +A(I  ,J+1,1)*Y(I  ,J+1) )
 ENDDO
 CALL MPI_ISEND(Y(ISTA,JEND),1,MPI_REAL8,          24
&     JUP,1,MPI_COMM_WORLD,ISEND1,IERR)
 CALL MPI_ISEND(Y(IEND,JSTA),1,MPI_REAL8,          25
&     IUP,1,MPI_COMM_WORLD,ISEND2,IERR)
 CALL MPI_IRECV(Y(ISTA,JSTA-1),1,MPI_REAL8,        24
&     JDOWN,1,MPI_COMM_WORLD,IRECV1,IERR)
 CALL MPI_IRECV(Y(ISTA-1,JSTA),1,MPI_REAL8,        25
&     IDOWN,1,MPI_COMM_WORLD,IRECV2,IERR)
 CALL MPI_WAIT(ISEND1,ISTATUS,IERR)
 CALL MPI_WAIT(ISEND2,ISTATUS,IERR)
 CALL MPI_WAIT(IRECV1,ISTATUS,IERR)
 CALL MPI_WAIT(IRECV2,ISTATUS,IERR)
 I = ISTA
 J = JSTA
 Y(I,J) = Y(I,J)-D(I,J)*(
&         A(I  ,J  ,1)*Y(I  ,J-1)                 26
&        +A(I  ,J  ,2)*Y(I-1,J  )
&        +A(I+1,J  ,2)*Y(I+1,J  )
&        +A(I  ,J+1,1)*Y(I  ,J+1) )
 END
```

図 5-4-16

5-5　マルチフロンタル法

不規則疎行列の連立一次方程式の直接法として、一般にスカイライン法が使用されます。図 5-5-1 (1) のような不規則疎行列（本例は対称行列を想定）の場合、スカイライン法では着色した部分のみをメモリー上に保持するので、行列全体を密行列として取り扱うよりも計算量が低減され、メモリーも節約できます。

ところが、図 5-5-1 (2) のようにバンド幅の広い不規則疎行列の場合、対角要素から遠く離れた要素が一つでもあると、その要素から対角要素までの全ての要素を、途中が 0 でも全て保持する必要があるため、計算量はあまり少なくならず、メモリーも節約できません。

近年マルチフロンタル法という解法が脚光を浴びており、市販の構造解析などのパッケージにも採用されています。マルチフロンタル法は、図 5-5-1 (3) に示すように、基本的に非ゼロ要素のみを必要とするため（これ以外に作業域が必要）、本例のようにバンド幅の広い不規則疎行列の場合、従来のスカイライン法と比べて計算時間が短縮でき、メモリーも少なくてすみます。

マルチフロンタル法の並列版は、第 6 章で紹介する数値計算ライブラリーの中に含まれている可能性がありますが、未調査です。

図 5-5-1 (1)　バンド幅が狭いスカイライン法

図 5-5-1 (2)　バンド幅が広いスカイライン法

図 5-5-1 (3)　バンド幅が広いマルチフロンタル法

5-6　SOR 法

5-6-1　SOR 法と並列性

差分法で使用されるガウスザイデル法やその改良版である SOR 法は、そのままでは並列性がありませんが、計算順序を変更することによって並列化することができます。まず、ガウスザイデル法はなぜ並列性がないのか、そして並列性を持たせるにはどのようにすればよいかについて図 5-6-1 で説明します。

- 図 5-6-1（次ページ）の (1) に示す、未知数が x_1〜x_4 の 1 次元の計算領域に対し、(2) に示す 1 次元ラプラス方程式を適用し、(3) のように差分化します。(3) を各要素に対して書き下すと (4) になり、これを上から順に計算したとすると、例えば 1 番目の式で求められた x_1 が 2 番目の式の右辺で参照されています。このような依存関係を ↘ で表わすことにします。すると (4) の各式には、i 番目の式の計算が終了しないと $i+1$ 番目の式の計算を始めることができないという依存関係があり、並列性のないことが分かります。なお、(4) の順序で計算を行うのが、(5) に示す通常のガウスザイデル法です。
- (4) を (6) のように書き表してみます。ここで例えば x_1 の計算前の値を①、計算後の値を$\boxed{1}$と表し、境界の定数を「境」と表します。例えば$\boxed{2}$の計算に$\boxed{1}$と③が使用されるので、これを ↙ と → で表します。例えば図のようにランク 0 と 1 の 2 つのプロセスで並列する場合、$\boxed{2}$から$\boxed{3}$へプロセス間にまたがる → があり、$\boxed{2}$の計算が終わらないと$\boxed{3}$を計算できないため、並列に実行することはできないことが分かります。
- (4) では上から順に計算しましたが、計算順序は必ずしもこの順である必要はありません。そこで、(4) の 2 式目と 3 式目を入れ替えた (7) を上から順に計算してみます。すると ↘ の矢印の状態は先ほどの (4) と変わりました。しかしこの図からでは、並列性があるのかどうかはっきりしません。
  そこで (7) を (8) で表し、さらに書き方を少し変えた (9) で検討してみます。すると$\boxed{1}$と$\boxed{3}$の計算、および$\boxed{2}$と$\boxed{4}$の計算を、ランク 0 と 1 で並列に行うことができることが分かります。ただし、$\boxed{1}$と$\boxed{3}$の計算が終了してから$\boxed{2}$と$\boxed{4}$の計算を行う必要があり、またランク 0 とランク 1 の間に矢印がまたがっているため、プロセス間で通信が必要となります。通信の部分をより明確にしたのが (10) で、→ と ← は通信を表します。
  なお、(4) と (7) は計算順序が異なるので、計算を 1 回行っただけでは計算結果は異なります。しかし計算を何度も反復させると次第に解は同じ値に収束していきます。ただし (4) と (7) で収束性は変わります。
- 以後、(10) を (11) のように表すことにします。ランク 0 は x_1 と x_2 を担当し、ランク 1 は x_3 と x_4 を担当します。
  [1] まず、プロセス間の境界のデータ②を〓方向へ通信します。
  [2] 各プロセスは奇数番号の要素① ③を計算します。
  [3] プロセス間の境界のデータ$\boxed{3}$を〓方向へ通信します。
  [4] 各プロセスは偶数番号の要素② ④を計算します。
- もう少し要素数が多い例を (12) に示します。(12) を上記の方法で 3 つのプロセスで並列に計算したときの動作を (13) に示します。以後の説明の都合上、要素の番号は① ② ③ ④ ... でなく⓪ ① ⓪ ① ... と表します。⓪と①は計算順序を意味します。[2] ではまず⓪の要素を計算し、[4] では①の要素を計算します。この方法を本書ではマルチカラー法と呼びます (計算領域が 2 つの異なる部分に互い違いに分かれるので)。なお (12) で各プロセス (ただしランク 2 を除く) の右側の境界の要素（★）は全て①になっています。理由は、右側の境界に①と⓪が混合していると、下図のようにプロセスによって通信の方向が反対になってしまい、並列化しにくいからです。

(1)（計算領域）
(2)（微分方程式）d²x/dy²=0
(3)（差分式）xi=(xi−1+xi+1)/2
(4)
(5)
do k=1,10000
do i=1,4
xi=(xi−1+xi+1)/2
enddo
収束したかチェック
enddo
(6)
(7)
順序
変更
(8)
(9)
(10)
(11)
ランク0
ランク1
ランク2
(12)
(13)

図 5-6-1

5-6-2　プログラム例（ゼブラ法）

本節では、2 次元のガウスザイデル法や SOR 法のプログラムを実際に並列化する方法を説明します。図 5-6-2 (1) に 2 次元 SOR 法のプログラムを示します。図 5-6-3 (1) が計算領域で、「境」は境界部分の要素を示します。計算は矢印の順に行われ、●は計算が終了している要素、○は現在計算している要素、○はこれから計算する要素を示します。また○の要素の計算には〓で囲んだ部分の要素を使用します (5 点差分)。

このプログラムを、まず前述のマルチカラー法の単体版に変更します。計算領域が 1 次元の場合、各要素を図 5-6-3 (2) の上段に示すように⓪と①に分割しましたが、2 次元の場合はこれを I 方向に引き伸ばしたと考え、図 5-6-3 (2) の下段のように分割します。⓪と①が列単位に互い違いになっているので、本書ではこれをゼブラ法と呼びます。計算順序はまず図 5-6-3 (2) で⓪を計算し、次に図 5-6-3 (3) で①を計算します。

ゼブラ法（単体版）のプログラムを図 5-6-2 (2) に示します。オリジナル版との変更点を下線で示します。[0] の DO ループが JJ = 0 のときに図 5-6-3 (2) の⓪を、JJ = 1 のとき図 5-6-3 (3) の①を計算します。

```
      :
  PARAMETER  (IMIN=1,IMAX=4,JMIN=1,JMAX=13)
  DIMENSION X(IMIN-1:IMAX+1,JMIN-1:JMAX+1)
      :
  DO K=1,10000 (収束反復)
    ERR = 0.0
    DO J=JMIN,JMAX
      DO I=IMIN,IMAX
        TEMP = 0.25*
&         (X(I+1,J  )+X(I  ,J+1)
&         +X(I  ,J-1)+X(I-1,J  ))-X(I,J)
        X(I,J) = X(I,J)+OMEGA*TEMP
        ERR = MAX(ERR,ABS(TEMP))
      ENDDO
    ENDDO
    IF (ERR<=EPS) EXIT (収束判定)
  ENDDO
      :
```

図 5-6-2 (1)　オリジナル

```
      :
  PARAMETER  (IMIN=1,IMAX=4,JMIN=1,JMAX=13)
  DIMENSION X(IMIN-1:IMAX+1,JMIN-1:JMAX+1)
      :
  DO K=1,10000 (収束反復)
    ERR = 0.0
    DO JJ=0,1                                  [0]
      DO J=JMIN+JJ,JMAX,2
        DO I=IMIN,IMAX
          TEMP = 0.25*
&           (X(I+1,J  )+X(I  ,J+1)
&           +X(I  ,J-1)+X(I-1,J  ))-X(I,J)
          X(I,J) = X(I,J)+OMEGA*TEMP
          ERR = MAX(ERR,ABS(TEMP))
        ENDDO
      ENDDO
    ENDDO
    IF (ERR<=EPS) EXIT (収束判定)
  ENDDO
      :
```

図 5-6-2 (2)　ゼブラ法（単体版）

図 5-6-3 (1)　SOR 法オリジナル

図 5-6-3 (2)　ゼブラ法（単体）⓪の計算

図 5-6-3 (3)　ゼブラ法（単体）①の計算

次にゼブラ法（単体版）を並列化する方法を説明します。まず図 5-6-4 に 3 プロセスで並列化したときの動作を示します。図 5-6-1 と同様に、分割方法は J 方向でブロック分割します。紙面の関係で「境」の部分は省略し、I 方向の要素数は 2 個とします。ゼブラ法で並列化したプログラムを図 5-6-6 に示し、説明を以下に示します。この方法は図 5-6-10 (1) に示すように、2 次元の 5，9 点差分、3 次元の 7，19 点差分の場合に使用できます。

- 図 5-6-6 の [1] で、プログラムの後半 ([9] と [10]) で行う通信（シフト）の準備をします (4-6-2 節参照)。
- [2]〜[6] で各プロセスの J 方向の担当範囲を求めます。このとき図 5-6-1 (12) の〓で説明したように、各プロセスが行う通信方向を一定にするため、各プロセス (ただしランク 2 を除く) の右側の境界を全て①にする必要があります。この方法を図 5-6-5 を使用して以下で説明します。
  - ⓪と①の 2 列を単位として分割するので、[2] で NBLOCK=2 を設定します。
  - [3] で、(2 列を 1 組とした) 列数 NTOTAL (本例では 7) を求めます。
  - [4] で、サブルーチン PARA_RANGE (4-5-4 節参照) を使用して、1〜NTOTAL のうちランクが IRANK のプロセスが担当する列番号の下限 JSTAX と上限 JENDX をブロック分割で求めます。
  - [5] で、ランクが IRANK のプロセスが担当する実際の列番号の下限 JJSTA(IRANK) と上限 JJEND(IRANK) を求めます。各プロセスは最低 2 列を担当する必要があるので、これを [6] でチェックし、2 列よりも少ないプロセスがある場合はエラーで終了します。
  - [7] で改めて、自プロセスが担当する J 方向の下限と上限を JSTA，JEND に設定します。
- [8] の JJ が 0 のとき、[9] で図 5-6-4 の〓の通信を行い、[11] で各プロセスは⓪を計算します。次に [8] の JJ が 1 のとき、[10] で〓の通信を行い、[11] で各プロセスは①を計算します。
- [12] で、各プロセスが求めた ERR の最大値を MPI_ALLREDUCE を使用して求めます。

図 5-6-4　ゼブラ法（並列版）の動作

図 5-6-5　ゼブラ法（並列版）の分割

```
    :
  INCLUDE 'mpif.h'
  PARAMETER  (IMIN=1,IMAX=4,JMIN=1,JMAX=13)
  DIMENSION X(IMIN-1:IMAX+1,JMIN-1:JMAX+1)
  INTEGER ISTATUS(MPI_STATUS_SIZE)
  INTEGER,ALLOCATABLE::JJSTA(:),JJEND(:)
  CALL MPI_INIT(IERR)
  CALL MPI_COMM_SIZE
&    (MPI_COMM_WORLD,NPROCS,IERR)
  CALL MPI_COMM_RANK
&    (MPI_COMM_WORLD,MYRANK,IERR)
  IUP   = MYRANK + 1
  IDOWN = MYRANK - 1                          [1]
  IF (MYRANK==NPROCS-1) IUP   = MPI_PROC_NULL
  IF (MYRANK==0       ) IDOWN = MPI_PROC_NULL
  ALLOCATE(JJSTA(0:NPROCS-1),
&          JJEND(0:NPROCS-1))
  NBLOCK = 2                                  [2]
  NTOTAL = (JMAX-JMIN)/NBLOCK + 1             [3]
  DO IRANK=0,NPROCS-1
    CALL PARA_RANGE(1,NTOTAL,NPROCS,          [4]
&        IRANK,JSTAX,JENDX)                   [4]
    JJSTA(IRANK) = JMIN+(JSTAX-1)*NBLOCK      [5]
    JJEND(IRANK)
&        = MIN(JMIN+JENDX*NBLOCK-1,JMAX)      [5]
    IF (JJEND(IRANK)-JJSTA(IRANK)+1
&                          < NBLOCK) THEN
      IF (MYRANK==0) PRINT *,'===ERROR==='
      CALL MPI_FINALIZE(IERR)                 [6]
      STOP
    ENDIF
  ENDDO
  JSTA = JJSTA(MYRANK)                        [7]
  JEND = JJEND(MYRANK)                        [7]
    :
```

```
  DO K=1,10000 (収束反復)
    ERR = 0.0
    DO JJ=0,1                                 [8]
      IF (JJ==0) THEN
        CALL MPI_ISEND(X(IMIN-1,JEND  ),
&            IMAX-IMIN+3,MPI_REAL,IUP  ,1,
&            MPI_COMM_WORLD,IREQS,IERR)       [9]
        CALL MPI_IRECV(X(IMIN-1,JSTA-1),
&            IMAX-IMIN+3,MPI_REAL,IDOWN,1,
&            MPI_COMM_WORLD,IREQR,IERR)
      ELSE
        CALL MPI_ISEND(X(IMIN-1,JSTA  ),
&            IMAX-IMIN+3,MPI_REAL,IDOWN,1,
&            MPI_COMM_WORLD,IREQS,IERR)
        CALL MPI_IRECV(X(IMIN-1,JEND+1),      [10]
&            IMAX-IMIN+3,MPI_REAL,IUP  ,1,
&            MPI_COMM_WORLD,IREQR,IERR)
      ENDIF
      CALL MPI_WAIT(IREQS,ISTATUS,IERR)
      CALL MPI_WAIT(IREQR,ISTATUS,IERR)
      DO J=JSTA+JJ,JEND,2
        DO I=IMIN,IMAX
          TEMP = 0.25*
&           (X(I+1,J  )+X(I  ,J+1)
&           +X(I  ,J-1)+X(I-1,J  ))-X(I,J)  [11]
          X(I,J) = X(I,J)+OMEGA*TEMP
          ERR = MAX(ERR,ABS(TEMP))
        ENDDO
      ENDDO
    ENDDO
    CALL MPI_ALLREDUCE(ERR,ERR2,1,MPI_REAL,
&        MPI_MAX,MPI_COMM_WORLD,IERR)          [12]
    ERR = ERR2
    IF (ERR<=EPS) EXIT (収束判定)
  ENDDO
  CALL MPI_FINALIZE(IERR)
    :
```

図 5-6-6　ゼブラ法（並列版）

5-6-3　その他の解法

本節ではゼブラ法以外の方法について紹介します。各解法のまとめを図 5-6-10 に示します。図 5-6-10 には計算領域が 3 次元の場合の図を描いてありますが、2 次元の場合は一番手前の面のみを考えて下さい。また、具体的な並列化の方法については 5-6-4 節で説明します。

マルチカラー法

- 図 5-6-7 (1) と (2) に示すオリジナルの SOR 法は、前述のように並列性がないため、(3) と (4) のように、J 方向の計算順序を「⓪ ① ⓪ ① ...」と互い違いにして並列性を持たせ、並列化しました（ゼブラ法）。
  ところでベクトル計算機の場合は、最も内側のループ (本例では DO I) がベクトル化の対象になります。(4) の〓で囲んだ各要素が独立に計算できれば DO I がベクトル化できますが、I 方向に関しては前述と同じ理由で依存関係があるためこのままではベクトル化できません。そこで (5) のように、I 方向の計算順序も「⓪ ① ⓪ ① ...」のようにすれば、前述と同じ理由で依存関係がなくなり、ベクトル化できます。(5) のように、I と J の両方向で「⓪ ① ⓪ ① ...」の計算順序になっている計算方法を本書では 2 色のマルチカラー法と呼びます。
  ところで (5) を並列化した場合、境界の 1 列のデータ (⓪または①) を通信しますが、例えば⓪を通信する場合、⓪がとびとびに入っていて通信しにくいので、実際には (6) (7) のように 4 色のマルチカラー法にした方が簡単になります（具体的な通信方法は後述します）。
  一方スカラー計算機で並列化する場合は (3) (4) のゼブラ法でも良いですが、I 方向と J 方向の計算を同じパターンにしたい場合、あるいは (8) のように I 方向と J 方向の両方でブロック分割する場合、マルチカラー法を使用します。ただし、メモリー上でデータの並ぶ I 方向の要素の計算順序がとびとびになるため、ゼブラ法よりもややキャッシュミスが多くなり、パフォーマンスが若干低下する可能性があります。

図 5-6-7

パラレル・ブロック・オーダリング法

1 次元のガウスザイデル法で要素数が 4 個の場合、ゼブラ法やマルチカラー法では図 5-6-8 (1) の実線のように「⓪ ① ⓪ ①」と分割しました。要素数が多い場合は「⓪ ① ⓪ ①」がそのまま続いたと考え、(2) のように分割しました。

これに対し、要素数が多い場合は「⓪ ① ⓪ ①」のうち⓪の部分だけを広げ、(5) のように分割する方法を本書ではパラレル・ブロック・オーダリング法と呼びます。(2) と比べ、⓪の計算がメモリー上で連続するため、キャッシュミスが若干減少すると思われます。3 プロセスで並列化した場合を (3) と (6) に示しますが、パラレル・ブロック・オーダリング法ではプロセス数によって計算パターンが変わるという短所があります。

図 5-6-8　ランク

パイプライン法

上記で説明した方法はいずれも、元のガウスザイデル法の計算順序を変更することによって並列性を持たせました。4-6-7 節で説明したパイプライン法を利用すると、以下のように、元の計算順序を変えることなく並列化できる場合があります（プログラム例は省略します）。

図 5-6-9 (1)　オリジナル

=

図 5-6-9 (2)　パイプライン法

計算領域 (2 次元と 3 次元) の相違

ゼブラ法とパラレル・ブロック・オーダリング法の場合は、図 5-6-10 の (1) (2) (3) (7) (8) に示すように、2 次元の計算領域を K 方向に引き伸ばすと 3 次元になり、計算方法は同じです。マルチカラー法の場合は、図 5-6-10 の (4) (6) に示すように、2 次元と 3 次元で色の数が変わります。

差分式による相違

以下の (a) のように、●の計算で 1 つ離れた値を参照する場合 (図 5-6-10 の左半分) と、(b) のように 2 つ離れた値も参照する場合 (図 5-6-10 の右半分) で使用する色数が変わり、また並列化したときの通信方法も変わります。具体的な方法については後述します。

x_{i-2} x_{i-1} x_i x_{i+1} x_{i+2}

(a) ○—●—○　$x_i = x_{i-1} + x_{i+1}$

(b) ○—○—●—○—○　$x_i = x_{i-2} + x_{i-1} + x_{i+1} + x_{i+2}$

【2次元/5,9点差分】
【3次元/7,19点差分】
【2次元/13点差分】
【3次元/25点差分】
I
J
K
(1)ゼブラ
(2色)
(2)ゼブラ
(2色2列)
(3)ゼブラ
(3色)
(4)マルチカラー
(2次元:4色)
(3次元:8色)
(5)マルチカラー(2色)
2次元は5点差分
3次元は7点差分
のみ
(6)マルチカラー
(2次元:9色)
(3次元:27色)
(7)パラレル・ブロック・オーダリング
(境界1列)
(8)パラレル・ブロック・オーダリング
(境界2列)
ランク0
ランク1
ランク2
KIJ→

図 5-6-10

5-6-4　各解法の具体的な並列化方法

本節では、図 5-6-10 の各方法 ((5) と図 5-6-6 で説明した (1) を除く) を並列化した場合のプログラム例とデータの動きについて説明します。

図 5-6-6 の 2 次元用プログラムのうち、各解法によって異なる部分は [2]〜[6] と [8]〜[11] なので、プログラム例ではその部分のみを示します。なお [6] のエラーチェック部分と [9] [10] の通信ルーチンは省略しました。また 3 次元用プログラムも省略しました。

ゼブラ (図 5-6-10 の (2) 参照)

- 2 次元の 13 点差分、3 次元の 25 点差分の場合に使用します。
- 各プロセス (ランク 2 を除く) が担当する列数は、4 列 (⓪ ⓪ ① ①) の倍数になっている必要があります。このため [1] で 4 を指定し、4 列を単位としてブロック分割します。
- 各プロセスが担当する列数は最低 4 列 (⓪ ⓪ ① ①) 必要なので、これを [2] でチェックします。

```
    :
  NBLOCK = 4                                          [1]
  NTOTAL = (JMAX-JMIN)/NBLOCK + 1
  DO IRANK=0,NPROCS-1
    CALL PARA_RANGE(1,NTOTAL,NPROCS,IRANK,JSTAX,JENDX)
    JJSTA(IRANK) = JMIN+(JSTAX-1)*NBLOCK
    JJEND(IRANK) = MIN(JMIN+JENDX*NBLOCK-1,JMAX)
    IF (JJEND(IRANK)-JJSTA(IRANK)+1<NBLOCK) THEN      [2]
      エラー処理
    ENDIF
  ENDDO
    :
```

```
    :
  DO JJJ=0,1
    JJJ=0のとき、↳の通信を行う。
    JJJ=1のとき、↲の通信を行う。
    DO JJ=JSTA+JJJ*2,JEND,4
      DO J=JJ,MIN(JJ+1,JMAX)
        DO I=IMIN,IMAX
          X(I,J) = ～
        ENDDO
      ENDDO
    ENDDO
  ENDDO
    :
```

ゼブラ (図 5-6-10 の (3) 参照)

- 2 次元の 13 点差分、3 次元の 25 点差分の場合に使用します。
- 各プロセス (ランク 2 を除く) が担当する列数は、3 列 (⓪ ① ②) の倍数になっている必要があります。このため [1] で 3 を指定し、3 列を単位としてブロック分割します。
- 各プロセスが担当する列数は最低 3 列 (⓪ ① ②) 必要なので、これを [2] でチェックします。

```
   :
 NBLOCK = 3                                          [1]
 NTOTAL = (JMAX-JMIN)/NBLOCK + 1
 DO IRANK=0,NPROCS-1
   CALL PARA_RANGE(1,NTOTAL,NPROCS,IRANK,JSTAX,JENDX)
   JJSTA(IRANK) = JMIN+(JSTAX-1)*NBLOCK
   JJEND(IRANK) = MIN(JMIN+JENDX*NBLOCK-1,JMAX)
   IF (JJEND(IRANK)-JJSTA(IRANK)+1<NBLOCK) THEN      [2]
     エラー処理
   ENDIF
 ENDDO
   :
```

```
   :
 DO JJ=0,2
   JJ=0のとき、↳の通信を行う。
   JJ=1のとき、↲の1回目の通信を行う。
   JJ=2のとき、↲の2回目の通信を行う。
   DO J=JSTA+JJ,JEND,3
     DO I=IMIN,IMAX
       X(I,J) = ～
     ENDDO
   ENDDO
 ENDDO
   :
```

マルチカラー (図 5-6-10 の (4) 参照)

- 2 次元の 5，9 点差分、3 次元の 7，19 点差分の場合に使用します。
- 2 次元では⓪〜③の (2 × 2 =) 4 色、3 次元では⓪〜⑦の (2 × 2 × 2 =) 8 色になります。ただし 3 次元で J 方向をマルチカラーにしなければ (2 × 2 =) 4 色です。
- 各プロセス (ランク 2 を除く) が担当する列数は、2 列 (⓪ ②) の倍数になっている必要があります。このため [1] で 2 を指定し、2 列を単位としてブロック分割します。
- 各プロセスが担当する列数は最低 2 列 (⓪ ②) 必要なので、これを [2] でチェックします。

```
   :
  NBLOCK = 2                                          [1]
  NTOTAL = (JMAX-JMIN)/NBLOCK + 1
  DO IRANK=0,NPROCS-1
    CALL PARA_RANGE(1,NTOTAL,NPROCS,IRANK,JSTAX,JENDX)
    JJSTA(IRANK) = JMIN+(JSTAX-1)*NBLOCK
    JJEND(IRANK) = MIN(JMIN+JENDX*NBLOCK-1,JMAX)
    IF (JJEND(IRANK)-JJSTA(IRANK)+1<NBLOCK) THEN      [2]
      エラー処理
    ENDIF
  ENDDO
   :
```

```
   :
  DO JJ=0,1
    JJ=0のとき、↳の通信を行う。
    JJ=1のとき、↵の通信を行う。
    DO II=0,1
      DO J=JSTA+JJ,JEND,2
        DO I=IMIN+II,IMAX,2
          X(I,J) = ～
        ENDDO
      ENDDO
    ENDDO
  ENDDO
   :
```

マルチカラー (図 5-6-10 の (6) 参照)

- 2 次元の 13 点差分、3 次元の 25 点差分の場合に使用します。
- 2 次元では⓪〜⑧の (3 × 3 =) 9 色、3 次元では⓪〜㉖の (3 × 3 × 3 =) 27 色になります。ただし 3 次元で J 方向をマルチカラーにしなければ (3 × 3 =) 9 色です。
- 各プロセス (ランク 2 を除く) が担当する列数は、3 列 (⓪ ③ ⑥) の倍数になっている必要があります。このため [1] で 3 を指定し、3 列を単位としてブロック分割します。
- 各プロセスが担当する列数は最低 3 列 (⓪ ③ ⑥) 必要なので、これを [2] でチェックします。

```
    :
  NBLOCK = 3                                          [1]
  NTOTAL = (JMAX-JMIN)/NBLOCK + 1
  DO IRANK=0,NPROCS-1
    CALL PARA_RANGE(1,NTOTAL,NPROCS,IRANK,JSTAX,JENDX)
    JJSTA(IRANK) = JMIN+(JSTAX-1)*NBLOCK
    JJEND(IRANK) = MIN(JMIN+JENDX*NBLOCK-1,JMAX)
    IF (JJEND(IRANK)-JJSTA(IRANK)+1<NBLOCK) THEN      [2]
      エラー処理
    ENDIF
  ENDDO
    :
```

```
    :
  DO JJ=0,2
    JJ=0のとき、↳の通信を行う。
    JJ=1のとき、↲の1回目の通信を行う。
    JJ=2のとき、↲の2回目の通信を行う。
    DO II=0,2
      DO J=JSTA+JJ,JEND,3
        DO I=IMIN+II,IMAX,3
          X(I,J) = ～
        ENDDO
      ENDDO
    ENDDO
  ENDDO
    :
```

パラレル・ブロック・オーダリング (図 5-6-10 の (7) 参照)

- 2 次元の 5，9 点差分、3 次元の 7，19 点差分の場合に使用します。
- 各プロセスが担当する列数は 2 列以上 (⓪が 1 列以上、①が 1 列) 必要なので、これを [1] [2] でチェックします。

```
    :
  NBLOCK = 2                                          [1]
  DO IRANK=0,NPROCS-1
    CALL PARA_RANGE(JMIN,JMAX,NPROCS,IRANK,
&                   JJSTA(IRANK),JJEND(IRANK))
    IF (JJEND(IRANK)-JJSTA(IRANK)+1<NBLOCK) THEN [2]
      エラー処理
    ENDIF
  ENDDO
    :
```

```
    :
  DO JJ=0,1
    JJ=0のとき、
      JSTAX=JSTA,JENDX=JEND-1
      ↳の通信を行う。
    JJ=1のとき、
      JSTAX=JEND,JENDX=JEND
      ↲の通信を行う。
    DO J=JSTAX,JENDX
      DO I=IMIN,IMAX
        X(I,J) = ～
      ENDDO
    ENDDO
  ENDDO
    :
```

パラレル・ブロック・オーダリング (図 5-6-10 の (8) 参照)

- 2 次元の 13 点差分、3 次元の 25 点差分の場合に使用します。
- 各プロセスが担当する列数は 4 列以上 (⓪が 2 列以上、①が 2 列) 必要なので、これを [1] [2] でチェックします。

```
    :
 NBLOCK = 4                                          [1]
 DO IRANK=0,NPROCS-1
   CALL PARA_RANGE(JMIN,JMAX,NPROCS,IRANK,
&                  JJSTA(IRANK),JJEND(IRANK))
   IF (JJEND(IRANK)-JJSTA(IRANK)+1<NBLOCK) THEN [2]
     エラー処理
   ENDIF
 ENDDO
    :
```

```
    :
 DO JJ=0,1
   JJ=0のとき、
     JSTAX=JSTA,JENDX=JEND-2
     ↳の通信を行う。
   JJ=1のとき、
     JSTAX=JEND-1,JENDX=JEND
     ↲の通信を行う。
   DO J=JSTAX,JENDX
     DO I=IMIN,IMAX
       X(I,J) = ～
     ENDDO
   ENDDO
 ENDDO
    :
```

5-7　モンテカルロ法

モンテカルロ法は、各粒子の挙動がバラバラなので、一般にベクトル計算機よりも並列計算機に向いています。また一般に各粒子間の動作に関係がないことから、計算量に比べて通信量の割合が少なく、並列化の効果が出ることが多いようです。本節ではモンテカルロ法モドキの簡単な例を紹介します。

図 5-7-1 は、粒子が原点から出発して 10 行程進んだとき、0〜9 のどのレンジに到達したかを調べるプログラムです (例えば図 5-7-1 の粒子はレンジ 6 に到達しています)。各粒子は 1 行程で「1 単位長さ」進み、進行方向 (360 度のいずれか) を乱数で決定します。これを 100000 個の粒子について調べ、0〜9 の各レンジに入った粒子の個数を求めます。

まず単体版のプログラムを図 5-7-2 に示します。なお、このプログラムでは 0〜1 の範囲の一様乱数を発生する乱数ルーチンを使用しますが、使用方法はマシン環境によって異なるので具体的には記述しません。

- ①で乱数の種（シード）の初期値を設定します。
- ②のループで 100000 個の粒子を 1 つずつ計算します。
- ③の粒子の X, Y 座標を「原点」に初期設定します。
- ④のループで粒子を 1 行程ずつ 10 行程まで進めます。
- ⑤で乱数を 1 個発生させ、⑥でそれを使用して粒子の進行方向を決定します。
- ⑦と⑧で粒子を「1 単位長さ」進め、X, Y 座標を更新します。
- 10 行程進んだ後の原点との距離を⑨で求め、そのレンジに含まれる個数を⑩で更新します。

図 5-7-1

```
PROGRAM MAIN
PARAMETER (N=100000, PAI=3.1415)
INTEGER ITOTAL(0:9)
DO I = 0, 9
  ITOTAL(I) = 0
ENDDO
乱数の種(シード)の初期値を設定 ①
DO I = 1, N                          ②
  X = 0.0; Y = 0.0                   ③
  DO ISTEP = 1, 10                   ④
    乱数を1個発生させる              ⑤
    ANGLE = 2.0*PAI*乱数             ⑥
    X = X + COS(ANGLE)               ⑦
    Y = Y + SIN(ANGLE)               ⑧
  ENDDO
  ITEMP = SQRT(X**2 + Y**2)          ⑨
  ITOTAL(ITEMP)=ITOTAL(ITEMP)+1⑩
ENDDO
PRINT *,'TOTAL =',ITOTAL
END
```

図 5-7-2

図 5-7-2 を並列化したプログラムを図 5-7-3 に示します。

- 図 5-7-3 の⑪で、サブルーチン PARA_RANGE (4-5-4 節参照) を使用して粒子数 (N) をブロック分割し、各プロセスが担当する粒子番号の下限を ISTA、上限を IEND とします。
- 12で乱数の種（シード）の初期値を設定します（詳細は後述します）。
- ⑬で各プロセスは自分の担当する粒子 (ISTA から IEND まで) を並列に計算します。
- 14で乱数を 1 個発生させます（詳細は後述します）。
- これ以後の動作は単体版 (図 5-7-2) の場合と同じなので、説明は省略します。
- ⑮で、各プロセスが求めた配列 ITOTAL の値を、MPI_REDUCE を使用して配列の各要素ごとに合計します。

```
PROGRAM MAIN
INCLUDE 'mpif.h'
PARAMETER (N=100000, PAI=3.1415)
INTEGER ITOTAL(0:9),IITOTAL(0:9)
CALL MPI_INIT(IERR)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,～)
CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,～)
CALL PARA_RANGE(1,N,NPROCS,MYRANK,ISTA,IEND)
DO I = 0, 9                                  ⑪
  ITOTAL(I) = 0
ENDDO
乱数の種(シード)の初期値を設定                 12
DO I = ISTA, IEND                            ⑬
  X = 0.0; Y = 0.0
  DO ISTEP = 1, 10
    乱数を1個発生させる                       14
    ANGLE = 2.0*PAI*乱数
    X = X + COS(ANGLE)
    Y = Y + SIN(ANGLE)
  ENDDO
  ITEMP = SQRT(X**2 + Y**2)
  ITOTAL(ITEMP) = ITOTAL(ITEMP) + 1
ENDDO
右上へ
```

図 5-7-3

```
左下より
CALL MPI_REDUCE(ITOTAL,IITOTAL,10,
& MPI_INTEGER,MPI_SUM,0,MPI_COMM_WORLD,IERR) ⑮
IF (MYRANK==0) PRINT *,'TOTAL = ',IITOTAL
CALL MPI_FINALIZE(IERR)
END
```

単体	(1) (2) (3) (4) (5) (6) (7) (8) (9)

図 5-7-4　単体版での乱数

ランク0	(1) (2) (3)
ランク1	(1) (2) (3) (4) (5) (6)
ランク2	(1) (2) (3) (4) (5) (6) (7) (8) (9)

図 5-7-5　並列版【方法 1】

ランク0	(1) (2) (3) (4) (5) (6) (7) (8) (9)
ランク1	(1) (2) (3) (4) (5) (6) (7) (8) (9) (10)
ランク2	(1) (2) (3) (4) (5) (6) (7) (8) (9) (10) (11)

図 5-7-6　並列版【方法 2】

ランク0	(1) (2) (3)
ランク1	[1] [2] [3]
ランク2	(A) (B) (C)

図 5-7-7　並列版【方法 3】

並列版で乱数を発生する方法を 3 種類紹介します (これ以外に、参考文献 [70] に記述があります。またネットや文献で調べれば、もっとよい方法があるかもしれません)。説明を簡単にするため、単体版では図 5-7-4 に示す (1)～(9) の 9 個の乱数を順番に使用し、並列版は 3 プロセスがそれぞれ 3 個の乱数を使用するとします。

- 【方法 1】(図 5-7-5) 各プロセスは単体版と同じ初期値の乱数を発生させ、そのうち下線部の乱数を使用します。例えばランク 2 のプロセスは、ランク 0 と 1 が乱数を 3 個ずつ使用することが分かっているので、(1)～(6) の乱数をダミーで発生させた後、(7)～(9) の乱数を使用します。図 5-7-3 では各粒子が 10 個の乱数を使用するので、12の後に (ISTA-1)*10 個の乱数をダミーで発生させた後、14で実際に使う乱数を発生させます。
  この方法は単体版と同じ乱数を同じ順番に使用するので、計算結果も同じになります。しかし図 5-7-5 で例えばランク 2 のプロセスは、単体版と同じ数の乱数を発生させる必要があるので、乱数を発生する部分に計算時間がかかっているプログラムの場合は、並列化の効果が出ません。また本例のように、各プロセスが使用する乱数の個数が計算前に分かっているプログラムは少ないと思われます。
- 【方法 2】(図 5-7-6) 各プロセスは単体版と同じ初期値の乱数を発生させ、それを各プロセスがサイクリックに使用します。まず12の後で MYRANK 個 (ランク 2 なら 2 個) の乱数をダミーで発生させます。14では NPROCS 個 (本例では 3 個) の乱数を発生させ、そのうち最初の乱数のみ (図 5-7-6 の下線の乱数) を使用します。
  この方法は単体版と同じ乱数を使用しますが、使用する順番が異なるので、計算結果は変わります。また各プロセスは単体版と同じ数の乱数を発生させる必要があるので、乱数を発生する部分に計算時間がかかっているプログラムの場合は、並列化の効果が出ません。
- 【方法 3】(図 5-7-7) 各プロセスはそれぞれ別の初期値の乱数を使用します。12で各プロセスが異なる種（シード）の初期値を設定します (初期値を乱数系列の 1 周期の 1/NPROCS 個おきに設定する方法、「MYRANK+100」のように MYRANK の値を使って設定する方法、その時点の時刻を使って設定する方法などがあります)。
  この方法は単体版と異なる乱数を使用して計算を行うので、単体版と計算結果が変わります。また 1 つの並列ジョブ内で異なる初期値の乱数を複数使用するので、乱数の性質が重要となるプログラムの場合は問題があると思われます。一方、例えば 3 プロセスで実行する場合、発生させる乱数の数は単体版の 1/3 に減少するので、乱数を発生する部分に計算時間がかかっているプログラムの場合は、並列化の効果が出ます。

5-8　個別要素法／分子動力学法

個別要素法や分子動力学法のプログラムは並列化に向いていると言われています。通常、これらのプログラムは粒子で反復する一部の DO ループのみがホットスポットになっています。4-3 節の『計算パターン 1』で述べたように、ホットスポットの DO ループのみを（本当の意味の）並列化し、ループ終了後に、各プロセスは自分が計算した部分のデータを他の全プロセスに送信する方法を取ると、並列化にともなう修正が簡単になり、またパフォーマンスもそこそこ出るようです。

ここでは簡単のため、一次元の個別要素法／分子動力学法モドキの例で並列化の方法を説明します。以下のように一次元の計算領域に粒子が 7 個存在し、粒子①の X 方向の座標を X_i とします。

全ての粒子間には粒子間の距離に反比例した引力が働くとします。例えば粒子③に働く粒子⑥からの引力は $f_{36} = 1.0/(X_6 - X_3)$ となり、相手の粒子⑥には反対向きの引力 $-f_{36}$ が働きます。粒子③に働く全ての粒子からの引力（以後合力と呼びます）を F_3 とすると、F_3 は以下のようになります。

$$F_3 = -f_{13} - f_{23} + f_{34} + f_{35} + f_{36} + f_{37}$$

各粒子の合力の計算が終了すると、その値を使用して各粒子の新しい座標を求めます。例えば粒子③に働く合力 F_3 と粒子③の現在の座標 $X_{3(\mathrm{OLD})}$ から、粒子③の新しい座標 $X_{3(\mathrm{NEW})}$ を次のように求めます。

$$X_{3(\mathrm{NEW})} = X_{3(\mathrm{OLD})} + F_3$$

以上の計算を行うプログラムを図 5-8-1 (1) に示します。各粒子の合力を配列 F に、座標を配列 X に入れます。

- (1) はタイムステップのループです。(2) で合力の配列 F をゼロクリアします。
- (3) と (4) で各粒子の合力を求めます。(3) の I は合力を求めたい粒子の番号①を、(4) の J は相手の粒子の番号ⓙを表します。重複を避けるため (4) のループは『DO J=I+1,N』とします。(5) で粒子①とⓙの間に働く引力を変数 FIJ に求め、それを (6) で粒子①の合力に加え、(7) で粒子ⓙの合力から引きます。
- (3) と (4) のループの計算が終了すると、合力の配列 F の中身は図 5-8-2 (1) のようになります。
- 合力の計算が終了した後、(8) で、各粒子の新しい座標 X を現在の座標 X と合力 F から求めます。

```
    :
  PARAMETER(N=7)
  DIMENSION F(N),X(N)
    :
  DO ITIME = 1, 100    (1)
    DO I = 1, N
      F(I) = 0.0       (2)
    ENDDO
    DO I = 1, N-1      (3)
      DO J = I+1, N    (4)
        FIJ = 1.0/(X(J)-X(I))  (5)
        F(I) = F(I) + FIJ      (6)
        F(J) = F(J) - FIJ      (7)
      ENDDO
    ENDDO
    DO I = 1, N
      X(I) = X(I) + F(I)       (8)
    ENDDO
  ENDDO
    :
```

図 5-8-1 (1)

図 5-8-1 (1) のプログラムの並列化について説明します。ここでは説明を簡単にするため、(3) と (4) のループのみ並列化し、(8) のループは並列化しないことにします。

まず分割方法ですが、通常個別要素法や分子動力学法のプログラムでは、各粒子の計算量が異なっていることが多いことと、(3) と (4) のループで実際に計算する部分が図 5-8-2 (1) の全体ではなく右上三角部分というブロック分割しにくい形状になっていることから、分割方法をサイクリック分割とします。

まず (3) のループでサイクリック分割して並列化したプログラムを図 5-8-1 (2) に示します。(3) をサイクリック分割して (9) に変更します。(3) と (4) のループが終了した時点では、各プロセスの合力の配列 F の中身は図 5-8-2 (2) の上図のようになります。(8) のループは並列化しないので、(8) のループに入る前に通信を行って全プロセスが正しい合力 F の値を持つ必要があります。これを (10) で、4-6-6 節で紹介した『重ね合せ』を使って行うと、合力 F は図 5-8-2 (2) の下図のようになります。結果が配列 FF に入るので、(8) の配列 F は (11) のように FF に修正します。

一方 (3) と (4) の 2 重ループの内側でサイクリック分割した場合、図 5-8-1 (3) のようになります。サイクリック分割に関係する部分を (12) に示します。この場合、(3) と (4) のループが終了した時点で各プロセスの合力の配列 F の中身は図 5-8-2 (3) のようになります。以後の処理は図 5-8-1 (2) の場合と同じです。

いくつか考慮点を以下に示します。

- 図 5-8-1 (2) と図 5-8-1 (3) のどちらがよいかはプロセス間のロードバランス、キャッシュミス、(12) の IF 文のオーバーヘッドなどが関係するのでケースバイケースです。
- (3) と (4) の部分の計算量が少ない場合、本例のように重ね合せのような派手な通信を行うと、通信のオーバーヘッドによってパフォーマンスが出ない場合もあります。(3) と (4) 部分のチューニング方法については参考文献 [16] を参照して下さい)。そのような場合、実際のプログラムでは全粒子間で引力が働くわけではない（近隣の粒子間のみ働く）ので、重ね合せではなく、必要最小限の要素のみを通信するようにします。ただしプログラムが面倒になります。
- (8) の部分も含めて並列化しないと効果が出ない場合もあります。その場合、タイムステップ内の全ループを検討し、どこで何をどのように通信すれば通信を少なくできるのかを検討する必要があります。

```
      :
      INCLUDE 'mpif.h'
      PARAMETER(N=7)
      DIMENSION F(N),X(N),FF(N)
      :
      DO ITIME = 1, 100
        DO I = 1, N
          F(I) = 0.0
        ENDDO
        DO I = 1+MYRANK, N-1, NPROCS                (9)
          DO J = I+1, N
            FIJ = 1.0/(X(J)-X(I))
            F(I) = F(I) + FIJ
            F(J) = F(J) - FIJ
          ENDDO
        ENDDO
        CALL MPI_ALLREDUCE(F,FF,N,MPI_REAL,         (10)
     &          MPI_SUM,MPI_COMM_WORLD,IERR)        (10)
        DO I = 1, N
          X(I) = X(I) + FF(I)                       (11)
        ENDDO
      ENDDO
      :
```

図 5-8-1 (2)

```
      :
      INCLUDE 'mpif.h'
      PARAMETER(N=7)
      DIMENSION F(N),X(N),FF(N)
      :
      DO ITIME = 1, 100
        DO I = 1, N
          F(I) = 0.0
        ENDDO
        IRANK = -1                                  (12)
        DO I = 1, N-1
          DO J = I+1, N
            IRANK = IRANK + 1                       (12)
            IF (IRANK == NPROCS) IRANK = 0          (12)
            IF (MYRANK == IRANK) THEN               (12)
              FIJ = 1.0/(X(J)-X(I))
              F(I) = F(I) + FIJ
              F(J) = F(J) - FIJ
            ENDIF                                   (12)
          ENDDO
        ENDDO
        CALL MPI_ALLREDUCE(F,FF,N,MPI_REAL,
     &          MPI_SUM,MPI_COMM_WORLD,IERR)
```

```
    DO I = 1, N
      X(I) = X(I) + FF(I)
    ENDDO
  ENDDO
    :
```

図 5-8-1 (3)

F(1)		$+f_{12}+f_{13}+f_{14}+f_{15}+f_{16}+f_{17}$					
F(2)	$-f_{12}$		$+f_{23}+f_{24}+f_{25}+f_{26}+f_{27}$				
F(3)	$-f_{13}$	$-f_{23}$		$+f_{34}+f_{35}+f_{36}+f_{37}$			
F(4)	$-f_{14}$	$-f_{24}$	$-f_{34}$		$+f_{45}+f_{46}+f_{47}$		
F(5)	$-f_{15}$	$-f_{25}$	$-f_{35}$	$-f_{45}$		$+f_{56}+f_{57}$	
F(6)	$-f_{16}$	$-f_{26}$	$-f_{36}$	$-f_{46}$	$-f_{56}$		$+f_{67}$
F(7)	$-f_{17}$	$-f_{27}$	$-f_{37}$	$-f_{47}$	$-f_{57}$	$-f_{67}$	

図 5-8-2 (1)

【ランク0】

F(1)		$+f_{12}+f_{13}+f_{14}+f_{15}+f_{16}+f_{17}$					
F(2)	$-f_{12}$						
F(3)	$-f_{13}$						
F(4)	$-f_{14}$				$+f_{45}+f_{46}+f_{47}$		
F(5)	$-f_{15}$			$-f_{45}$			
F(6)	$-f_{16}$			$-f_{46}$			
F(7)	$-f_{17}$			$-f_{47}$			

【ランク1】

			$+f_{23}+f_{24}+f_{25}+f_{26}+f_{27}$				
		$-f_{23}$					
		$-f_{24}$					
		$-f_{25}$				$+f_{56}+f_{57}$	
		$-f_{26}$			$-f_{56}$		
		$-f_{27}$			$-f_{57}$		

【ランク2】

				$+f_{34}+f_{35}+f_{36}+f_{37}$			
			$-f_{34}$				
			$-f_{35}$				
			$-f_{36}$				$+f_{67}$
			$-f_{37}$			$-f_{67}$	

FF(1)		$+f_{12}+f_{13}+f_{14}+f_{15}+f_{16}+f_{17}$					
FF(2)	$-f_{12}$		$+f_{23}+f_{24}+f_{25}+f_{26}+f_{27}$				
FF(3)	$-f_{13}$	$-f_{23}$		$+f_{34}+f_{35}+f_{36}+f_{37}$			
FF(4)	$-f_{14}$	$-f_{24}$	$-f_{34}$		$+f_{45}+f_{46}+f_{47}$		
FF(5)	$-f_{15}$	$-f_{25}$	$-f_{35}$	$-f_{45}$		$+f_{56}+f_{57}$	
FF(6)	$-f_{16}$	$-f_{26}$	$-f_{36}$	$-f_{46}$	$-f_{56}$		$+f_{67}$
FF(7)	$-f_{17}$	$-f_{27}$	$-f_{37}$	$-f_{47}$	$-f_{57}$	$-f_{67}$	

		$+f_{12}+f_{13}+f_{14}+f_{15}+f_{16}+f_{17}$					
$-f_{12}$			$+f_{23}+f_{24}+f_{25}+f_{26}+f_{27}$				
$-f_{13}$	$-f_{23}$			$+f_{34}+f_{35}+f_{36}+f_{37}$			
$-f_{14}$	$-f_{24}$	$-f_{34}$			$+f_{45}+f_{46}+f_{47}$		
$-f_{15}$	$-f_{25}$	$-f_{35}$	$-f_{45}$			$+f_{56}+f_{57}$	
$-f_{16}$	$-f_{26}$	$-f_{36}$	$-f_{46}$	$-f_{56}$			$+f_{67}$
$-f_{17}$	$-f_{27}$	$-f_{37}$	$-f_{47}$	$-f_{57}$	$-f_{67}$		

		$+f_{12}+f_{13}+f_{14}+f_{15}+f_{16}+f_{17}$					
$-f_{12}$			$+f_{23}+f_{24}+f_{25}+f_{26}+f_{27}$				
$-f_{13}$	$-f_{23}$			$+f_{34}+f_{35}+f_{36}+f_{37}$			
$-f_{14}$	$-f_{24}$	$-f_{34}$			$+f_{45}+f_{46}+f_{47}$		
$-f_{15}$	$-f_{25}$	$-f_{35}$	$-f_{45}$			$+f_{56}+f_{57}$	
$-f_{16}$	$-f_{26}$	$-f_{36}$	$-f_{46}$	$-f_{56}$			$+f_{67}$
$-f_{17}$	$-f_{27}$	$-f_{37}$	$-f_{47}$	$-f_{57}$	$-f_{67}$		

図 5-8-2 (2)

【ランク0】

F(1)		$+f_{12}$			$+f_{15}$		
F(2)	$-f_{12}$		$+f_{23}$			$+f_{26}$	
F(3)		$-f_{23}$			$+f_{35}$		
F(4)					$+f_{45}$		
F(5)	$-f_{15}$		$-f_{35}$	$-f_{45}$		$+f_{56}$	
F(6)		$-f_{26}$			$-f_{56}$		
F(7)							

【ランク1】

		$+f_{13}$			$+f_{16}$		
			$+f_{24}$			$+f_{27}$	
$-f_{13}$					$+f_{36}$		
	$-f_{24}$				$+f_{46}$		
						$+f_{57}$	
$-f_{16}$		$-f_{36}$	$-f_{46}$				
	$-f_{27}$			$-f_{57}$			

【ランク2】

			$+f_{14}$			$+f_{17}$	
				$+f_{25}$			
			$+f_{34}$			$+f_{37}$	
$-f_{14}$		$-f_{34}$				$+f_{47}$	
	$-f_{25}$						
						$+f_{67}$	
$-f_{17}$		$-f_{37}$	$-f_{47}$		$-f_{67}$		

図 5-8-2 (3)

第６章　並列版数値計算ライブラリー

本章では、並列版の数値計算サブルーチンライブラリーについて簡単に紹介します。

6-1　並列版数値計算ライブラリー

並列版（分散メモリー用）の数値計算ライブラリーには次の種類があります。(2) と (3) の URL は巻末の参考文献 [42] を参照して下さい。

(1) コンピュータメーカーが提供している数値計算ライブラリー（一般に、そのメーカーのマシン用にチューニングされているので高速です）。
(2) ソフトウェア会社が販売している数値計算ライブラリー (NAG，IMSL など)。
(3) フリーの数値計算ライブラリー (PCP，ScaLAPACK，PETSc，GeoFEM など)。

単体版の数値計算ライブラリーを使用した並列化

並列版（分散メモリー用）の数値計算ライブラリーは、各プロセスが配列を縮小して持つため、一般に使い方が面倒です。プログラムによっては、並列版でなく、単体版の数値計算ライブラリーを使用して並列化できる場合もあり、一般に後者の方が修正が簡単です。

【例 1】行列乗算の数値計算ライブラリーの単体版と並列版が提供されているとします。例えば $AB = C$ を計算する場合、各プロセスが図 6-1-1 の部分を計算するようにすれば、単体版のライブラリーを使用して並列化することができます。ただし全プロセスが配列 A の同じ内容を持つ必要があります。

図 6-1-1

【例 2】例えば 6 組の非対称密行列の連立一次方程式 $A_i x_i = b_i$ $(i = 1〜6)$ を解くプログラムがあり、それぞれの連立一次方程式は全く独立に解くことができるとします。連立一次方程式を LU 分解で解く数値計算ライブラリーの単体版と並列版が提供されているとします。

並列版のライブラリーでは、図 6-1-2 (1) のように、各 $A_i x_i = b_i$ ごとに、全プロセスが並列に計算を行います。このとき図の ↔ に示す通信が発生します。

一方、図 6-1-2 (2) のように、6 組の連立一次方程式を 2 組ずつ各プロセスに分配すれば、単体版のライブラリーを使用して並列化することができます。これは 4-3 節で述べた計算パターン 3（上位の部分の並列化）に相当します。

両者を比較すると、図 6-1-2 (1) では ↔ に示す通信が発生するなどの理由で、通常、図 6-1-2 (2) の方がよいパフォーマンスが得られます。また【例 1】と同様の理由で、一般に単体版の方が使用方法が簡単です。

図 6-1-2 (1)

図 6-1-2 (2)

第7章　本当に並列化の必要があるか？

本章もついに最後の章に到達しました。せっかく並列化について勉強してきたところに水をさすようで恐縮ですが、本章では『本当に並列化の必要があるか？』といういささか過激なタイトルで説明を行います。

7-1　単体チューニング

単体チューニングと並列化

本書の、しかも 7 章まで読み進んできた方ならば、何らかの形で並列プログラムを作成してみたいと考えておられると思います。そしてその主な目的は経過時間の短縮にあると思われます。

ところで、経過時間の短縮ということであれば、並列化以外に単体チューニング (参考文献 [6] 参照) という方法もあります。単体チューニングを行なっても全然速くならないプログラムも勿論ありますが、極端なプログラムになると 10 倍、あるいは 100 倍速くなる（言いかえると元のプログラムが悪過ぎる）場合があります。

並列化も単体チューニングも、経過時間の短縮という意味では同じことなのですが、根本的に異なる点があります。例えば図 7-1-1 で、もともと 100 分かかっているプログラムを単体チューニングし、その結果 50 分になったとします。この場合、CPU 時間が物理的に 50 分短縮されたわけですから、そのジョブの経過時間が短縮したのと同時に、他のジョブにとっても CPU 時間を使用する機会が増えます。つまり単体チューニングは並列計算機のシステム全体の処理効率を向上させます。

一方、単体チューニングをせずに並列化のみを行い、4 ノードで経過時間が 30 分になったとします。考えてみると、並列化というのは単に CPU 時間を各ノードにばらまいているだけであり、CPU 時間自体が減少するわけではありません。言いかえると他のジョブの CPU 時間を奪って経過時間を短縮させていることになります。さらに、並列化して 4 ノードで 4 倍のパフォーマンスを得ることは通常ありえないので、システム全体の CPU 時間の合計は単体で実行するよりも多くなります (図 7-1-1 の例では、単体版で 100 分だったのが並列版ではシステム全体で $30 \times 4 = 120$ 分)。このように、並列化というのは並列計算機のシステム全体の処理効率を低下させることになります。

図 7-1-1

勿論、単体チューニングだけではパフォーマンス向上に限界があり、それ以上のパフォーマンスを得る必要があるのであれば、やはり並列化するしかありません。いずれにしても、並列化ばかりに目を奪われず、まず単体チューニングを行なってみて、その後で、さらに経過時間を短縮させる必要があれば並列化を行うのがよいのではないかと思います。

チューニングの効果は、特に多くのノードを使用して並列計算する場合、顕著に現れます。図 7-1-2 に示すように、ある単体プログラムをチューニングして速度が 2 倍になったとします。このプログラムは 100% の部分が並列化でき、通信がほぼゼロだとすると、オリジナル版を 20 台で並列計算したときの経過時間と、チューニング版を 10 台で並列計算したときの経過時間が同じになり、チューニング版では 10 台分のノードが節約できることになります。

図 7-1-2

7-2　並列処理と分散処理

4-3 節で、プログラムには末端の部分での並列化と上位の部分の並列化があり、なるべく上位の部分で並列化した方がよいという説明をしました。ところで上位の部分の並列化をもっと上に進めていくとどうなるのでしょうか？

ジョブ単位の並列化

一人のユーザーが計算機を利用して仕事を進めていく場合、大きく次の 2 つの形態が考えられます。

(1)　ジョブ間に並列性がない場合

この形態では、データ 1 を入力としてジョブを実行し、実行結果 1 が得られます。次に、その結果に基づいてデータ 2 を作成します。そしてデータ 2 を入力として 2 本目のジョブを実行します。以後はこの繰り返しになります。この形態の場合、ジョブ間に並列性はなく、ジョブを 4 本同時に実行することはできません。

図 7-2-1 (1)

(2)　ジョブ間に並列性がある場合

この形態では、データ 1〜4 の 4 種類のデータ（例えば少しずつ条件の異なるデータ）を最初に作成し、その後でジョブを実行します。この場合はジョブ間に並列性があるので、4 本のジョブを同時に実行することができます。

図 7-2-1 (2)

図 7-2-1 (1) の形態の場合、結果 4 を得るまでの時間を短縮しようとすれば、やはりプログラム自体を並列化する必要があります。

一方図 7-2-1 (2) の形態の場合、ジョブ間に並列性があります。並列計算機には複数のノードがあるため、最終結果 4 を得るまでの時間を短縮したければ、プログラム自体を並列化しなくとも、4 つのジョブを並列計算機の複数のノードで同時に実行すればよいのです。図 7-2-1 (2) の形態が、まさにさきほど述べた、1 つのプログラムの上位の部分の並列化のさらに上位の並列化、すなわちジョブの単位の並列化であると言えます。通常これを分散処理と呼びます。

ジョブ単位の並列化では、当然ながらプログラムを並列化する必要はなく、また、4 ノードで 3 倍しかパフォーマンスが向上しないといった、並列化にともなうシステム全体の効率の低下もありません。

ジョブ単位の並列化をさらに上に進めていくとどうなるでしょうか？　そうです。最後は人間単位の並列化になります。一般に並列計算機は一人のユーザーだけでなく、多くの人が使用します。このとき、A さん、B さん、C さんのジョブ間には（通常）因果関係はなく、ジョブ間に並列性があります。人間単位の並列化もジョブ単位の並列化の一種なので、以後は両者を合わせてジョブ単位の並列化と呼びます。

負荷分散用ソフトウェア

以上のように、大勢の人が並列計算機を使用している環境、または単一ユーザーのジョブ間に並列性がある環境では、同時に実行可能な多くのジョブがあります。一方、並列計算機は多くのノードから構成されており、並列ジョブを実行するマシンであると同時に、多くの単体ジョブを処理する分散処理マシンであるということもできます。

ユーザーと並列計算機の間には、一般に負荷分散用ソフトウェアが介在します。その一例を図 7-2-2 に示します。この例では各ノードが単体ジョブ用と並列ジョブ用に分かれています。まずユーザーは単体ジョブや並列ジョブを負荷分散用ソフトウェアに対して投入します。負荷分散用ソフトウェアはジョブをいったん待ち行列（単体ジョブ用または並列ジョブ用）に入れ、そして待ち行列の中で優先順位の高いジョブを、その時点で最も負荷の軽いノードに割り当てます。

これによって、並列計算機のシステム全体の処理能力を最大にすることができ、また、どのノードが空いているかをユーザーがいちいち調べる手間も省けます。

図 7-2-2

並列計算機と並列ジョブ

いつの間にか、並列化から分散処理に話がそれてしまいました。それでは一体どのような場合にプログラムの並列化を行うのでしょうか？　それは主に以下の２つのケースになると思います。

- 何度も使用し、計算時間が非常にかかる特定のジョブの経過時間を、とにかく短縮したい場合。
- 計算領域のメッシュを多くしたいので、とにかく巨大なメモリーが欲しい場合。

次に、システム運用の観点から見た場合、並列ジョブと分散処理をどのように共存させたらよいのでしょうか？　並列ジョブは最も遅いノードに足にひっぱられるため、高いパフォーマンスを得るためには、他のジョブが流れていない占有環境で実行する必要があります。従って、多くのユーザーが多くのジョブを実行している環境では並列ジョブを実行するのは難しくなります。このような場合、並列ジョブの実行は夜間と休日のみにするなど時間帯によって制限する方法や、ノード数がたくさんあるのであれば、前述のように並列計算機の各ノードを並列ジョブ用と単体ジョブ用のノードに分ける方法などがあります。

7-3　並列化とワークロード

プログラムを並列化するのにどの程度ワークロードがかかるのかという質問がよく出ます。本節ではこれに対する回答および、並列化に関するその他のトピックについて説明します。

- 4-3 節で説明したように、まず、全体的にどのようなパターンで並列化を行うかを決定する必要がありますが、そのためにはプログラムの全体構造をある程度把握する必要があります。私の場合は他の人が作成した中身が全くわからないプログラムを並列化するので、実は並列化作業の大部分のワークロードはプログラムの全体構造の把握に費やされてしまいます。逆に、プログラムの中身を知っていれば、この時間はそれほどかからないでしょう。
- ベクトル計算機の場合も、ベクトル化に向いているプログラムとそうでないプログラムがありますが、並列化の場合も同様です。並列化に向いていないプログラムを無理に並列化しても意味はありません。
- 並列化できるプログラムの場合、修正箇所が少なく、10 分程度で完成するプログラムもあれば、修正箇所が多く、修正作業に時間がかかるプログラムもあります。一般に、4-3 節の「計算パターン 1」と「計算パターン 3」では簡単に並列化を行うことができます。
- 並列化を行うと通信が必要になります。どの変数（配列）をプログラム内のどこで誰に通信すれば、並列プログラムとして正しく動作し、しかも通信量を最小にできるのかを判断する必要がありますが、この判断も、プログラムの中身をある程度知っていない場合はワークロードがかかることがあります。
- ホットスポットの部分をそのまま並列版の数値計算ライブラリーに置き換えることができれば、並列化のワークロードがあまりかからず効率的で、しかも（一般に）高いパフォーマンスを得ることができます。
- 過去の経験では、並列化の効果がでるのは次のようなプログラムです。
  - 差分法…ソルバーが SOR 法の場合はレッドブラック（マルチカラー）SOR 法に修正します。ADI 法もパイプライン法またはツイスト分割法で並列化が可能です。ソルバーに ICCG 法を使用している場合はパイプライン法によって並列化できますが、メッシュ数が少ないとあまり効果は出ません。
  - モンテカルロ法、分子動力学法、個別要素法などの粒子を扱う計算は並列化の効果が出ます。
  - 境界要素法…ホットスポットが密行列連立一次方程式になるため、並列化の効果が出ます。

  逆に、並列化してもあまり効果が出ない解法は、
  - 有限要素法（陰解法）…疎行列の連立一次方程式をスカイライン法で解く場合、並列化は可能ですが、計算量が少ないために通信の割合が多くなってしまい、あまり効果がでません。また、ICCG 法の場合は、不規則疎行列なのでパイプライン法が使えません。一般に、有限要素法の場合は、連立一次方程式の部分を並列化するのでなく、もっと上位のレベルで、領域分割法と呼ばれる並列化を行うのがよいと言われています (参考文献 [17])。

以上、長々と説明をしてきましたがやっとゴールに到達しました。本書が分散メモリー型並列計算機でのプログラム並列化の一助になれば幸いです。一人でも多くの方がプログラムの並列化に成功することを願ってやみません。お疲れさまでした。

付 録　MPIサブルーチン一覧

本付録では、MPI で提供されている主なサブルーチンの使用方法について説明します。また付録内の「機能」の「関連する節」の所に、各サブルーチンが出てくる本書内の節を示しました。

なお、本書は正式マニュアルではありませんので、正式な使用方法は参考文献 [15] [27] を参照して下さい。

環境管理サブルーチン　MPI_INIT

機能 (関連する節：3-2 節)

MPI 環境の初期化処理を行います。ユーザーは、MPI_INIT が行う処理内容について意識する必要はありません。MPI_INIT は、全ての MPI ルーチンの一番最初に 1 回だけ、必ずコールする必要があります（必須）。MPI_INIT をコールする前にユーザープログラムが存在しても構いません。

使用法

```
CALL MPI_INIT(ierror)
```

- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

サンプルプログラム

```
PROGRAM MAIN
INCLUDE 'mpif.h'                          MPIが使用するインクルードファイルを指定します。
CALL MPI_INIT(IERR)                       MPI環境の初期化処理を行います。
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)  プロセス数NPROCSを取得します。
CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)  自分のランクMYRANKを取得します。
PRINT *,'NPROCS = ',NPROCS,'MYRANK = ',MYRANK   NPROCSとMYRANKの値を書き出します。
CALL MPI_FINALIZE(IERR)                   MPI環境の終了処理を行います。
END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 3 a.out ↵
  NPROCS = 3 MYRANK = 0
  NPROCS = 3 MYRANK = 1
  NPROCS = 3 MYRANK = 2
```

環境管理サブルーチン　MPI_FINALIZE

機能 (関連する節：3-2 節)

MPI 環境の終了処理を行います。ユーザーは、MPI_FANALIZE が行う処理内容について意識する必要はありません。MPI_FINALIZE は、全ての MPI ルーチンの一番最後に 1 回だけ、必ずコールする必要があります（必須)。MPI_FINALIZE をコールした後にユーザープログラムが存在しても構いません。

プログラムが終了する全ての箇所 (STOP 文または END 文) で、必ず 1 度だけ MPI_FINALIZE をコールしてから終了するようにして下さい。例えば図 1 では、MPI_FINALIZE を図 2 のように挿入します。

なお、あるプロセスが MPI_FINALZE をコールせずに STOP 文や END 文で終了した場合、マシン環境によっては誤動作する（実行中の他のプロセスを強制的に終了させてしまう、あるいは実行するたびに計算結果が異なる再現性のないエラーとなる）ことがありますので、必ずコールしてから終了するようにして下さい。

```
  :
IF(xxx) STOP
  :
CALL MPI_BCAST(～) 最後のMPIルーチン
  :
END
```

図 1　×

```
  :
IF(xxx) THEN
  CALL MPI_FINALIZE(IERR)
  STOP
ENDIF
  :
CALL MPI_BCAST(～) 最後のMPIルーチン
CALL MPI_FINALIZE(IERR)
  :
END
```

図 2　○

使用法

CALL MPI_FINALIZE(ierror)

- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

サンプルプログラム

```
PROGRAM MAIN
INCLUDE 'mpif.h'                                  MPIが使用するインクルードファイルを指定します。
CALL MPI_INIT(IERR)                               MPI環境の初期化処理を行います。
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)    プロセス数NPROCSを取得します。
CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)    自分のランクMYRANKを取得します。
PRINT *,'NPROCS = ',NPROCS,'MYRANK = ',MYRANK     NPROCSとMYRANKの値を書き出します。
CALL MPI_FINALIZE(IERR)                           MPI環境の終了処理を行います。
END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 3 a.out ↵
  NPROCS = 3 MYRANK = 0
  NPROCS = 3 MYRANK = 1
  NPROCS = 3 MYRANK = 2
```

コミュニケータに関するサブルーチン　MPI_COMM_SIZE

機能 (関連する節：3-2 節)

コミュニケータ comm で指定したグループに含まれるプロセスの数が size に戻ります。

使用法

CALL MPI_COMM_SIZE(comm, size, ierror)

- comm: 整数。コミュニケータを指定します。
- size: 整数。comm で指定したグループ内に含まれるプロセスの数が戻ります。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

サンプルプログラム

```
PROGRAM MAIN
INCLUDE 'mpif.h'                          MPIが使用するインクルードファイルを指定します。
CALL MPI_INIT(IERR)                       MPI環境の初期化処理を行います。
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)  プロセス数NPROCSを取得します。
CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)  自分のランクMYRANKを取得します。
PRINT *,'NPROCS = ',NPROCS,'MYRANK = ',MYRANK   NPROCSとMYRANKの値を書き出します。
CALL MPI_FINALIZE(IERR)                   MPI環境の終了処理を行います。
END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 3 a.out ↵
  NPROCS = 3 MYRANK = 0
  NPROCS = 3 MYRANK = 1
  NPROCS = 3 MYRANK = 2
```

コミュニケータに関するサブルーチン　MPI_COMM_RANK

機能 (関連する節：3-2 節)

コミュニケータ comm で指定したグループ内での自分（コールしたプロセス）のランクが rank に戻ります。

使用法

CALL MPI_COMM_RANK(comm, rank, ierror)

- comm: 整数。コミュニケータを指定します。
- rank: 整数。comm で指定したグループ内での自分（コールしたプロセス）のランクが戻ります。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

サンプルプログラム

```
PROGRAM MAIN
INCLUDE 'mpif.h'                                MPIが使用するインクルードファイルを指定します。
CALL MPI_INIT(IERR)                             MPI環境の初期化処理を行います。
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)  プロセス数NPROCSを取得します。
CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)  自分のランクMYRANKを取得します。
PRINT *,'NPROCS = ',NPROCS,'MYRANK = ',MYRANK   NPROCSとMYRANKの値を書き出します。
CALL MPI_FINALIZE(IERR)                         MPI環境の終了処理を行います。
END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 3 a.out ↵
  NPROCS = 3 MYRANK = 0
  NPROCS = 3 MYRANK = 1
  NPROCS = 3 MYRANK = 2
```

環境管理サブルーチン　MPI_ABORT

機能 (関連する節：4-4-1 節，4-8-1 節)

いずれかのプロセスが「MPI_ABORT」をコールすると、コミュニケータ comm に所属する全てのプロセスを強制的に異常終了させます。

使用法

CALL MPI_ABORT(comm, errorcode, ierror)

- comm: 整数。異常終了させたいプロセスが所属するコミュニケータを指定します。
- errorcode: 整数。ここに指定した値が OS に戻ります (Fortran で STOP 文とともに指定する数字に相当します)。この値が異常終了メッセージ内でどのように表示されるか（あるいは表示されないか）はマシン環境によって異なりますので、マニュアルを参照して下さい。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

サンプルプログラム

```
PROGRAM MAIN
INCLUDE 'mpif.h'                          MPIが使用するインクルードファイルを指定します。
CALL MPI_INIT(IERR)                       MPI環境の初期化処理を行います。
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)  プロセス数NPROCSを取得します。
CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)  自分のランクMYRANKを取得します。
  :
IF (エラーを発見したら)                     いずれかのプロセスがエラーを発見したら、
  PRINT *,'===ERROR=== MYRANK=',MYRANK      エラーメッセージとランク番号を表示し、
  CALL MPI_ABORT(MPI_COMM_WORLD,999,IERR)   全プロセスを強制的に異常終了させます。
  CALL MPI_FINALIZE(IERR)
  STOP
ENDIF
  :
CALL MPI_FINALIZE(IERR)                   MPI環境の終了処理を行います。
END
```

環境管理ファンクション　MPI_WTIME

機能 (関連する節：4-8-3 節)

『経過時間を測定したい部分』の前後で本ファンクションを実行すると、得られた値の差が、その部分の経過時間（単位は秒）となります。なお、得られた経過時間は、本ファンクションを実行したプロセスでの経過時間であり、他のプロセスとは関係ありません。

使用法

```
elp = MPI_WTIME()
```

- elp: 倍精度実数。ある過去の時点からの経過時間（単位は秒）が戻ります。

サンプルプログラム

以下の例で、『計算部分』の経過時間の測定を、②で開始し④で終了します。⑤に示すように、得られた時刻の差が経過時間（単位は秒）となります。本ファンクションで得られる値は倍精度なので、⑥を宣言して下さい。

なお、①（通常は必須）と③（任意）を指定する意味は 4-8-3 節を参照して下さい。

```
PROGRAM MAIN
INCLUDE 'mpif.h'                                  MPIが使用するインクルードファイルを指定します。
REAL*8 ELP1,ELP2                              ⑥  経過時間の測定に使用する変数を倍精度で宣言します。
CALL MPI_INIT(IERR)                               MPI環境の初期化処理を行います。
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)    プロセス数NPROCSを取得します。。
CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)    自分のランクMYRANKを取得します。
入力データの読み込みなど
CALL MPI_BARRIER(MPI_COMM_WORLD,IERR)         ①  プロセス間の同期をとります。
ELP1 = MPI_WTIME()                            ②  MPI_WTIME()を実行して測定を開始します。
計算部分
CALL MPI_BARRIER(MPI_COMM_WORLD,IERR)         ③ (任意)プロセス間の同期をとります。
ELP2 = MPI_WTIME()                            ④  MPI_WTIME()を実行して測定を終了します。
PRINT *,'ELAPSE = ',ELP2-ELP1                 ⑤  経過時間を秒で表します。
CALL MPI_FINALIZE(IERR)                           MPI環境の終了処理を行います。
END
```

集団通信サブルーチン　MPI_BCAST

機能 (関連する節：3-3-4 節，4-6-5 節，5-3 節)

コミュニケータ (comm) 内の、1 つの送信元プロセス (root) の送信バッファー (buffer) から、その他全てのプロセスの受信バッファー (buffer) にメッセージを送信します。

使用法

```
CALL MPI_BCAST(buffer, count, datatype, root, comm, ierror)
```

- buffer: 送信元プロセスでは、送信バッファーの先頭アドレスを指定します。
  宛先プロセスでは、受信バッファーの先頭アドレスを指定します。
- count: 整数。メッセージの要素数を指定します。
- datatype: 整数。メッセージのデータ型を指定します。
- root: 整数。送信元プロセスの comm 内でのランクを指定します。
  comm 内の全プロセスが同じ値を指定する必要があります。
- comm: 整数。送受信に参加する全てのプロセスを含むグループのコミュニケータを指定します。
  全プロセスが同じ値を指定する必要があります。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

注意

- 送信メッセージとそれに対応する受信メッセージ間のバイト数は一致している必要があります。この条件を満足していれば、各プロセスで count，datatype は異なっていても構いません (ただしデータ型 (INTEGER，REAL など) は一致している必要があります)。
- コミュニケータ (comm) 内の全プロセスが本サブルーチンをコールする必要があります。

【例 1】

【例 2】

サンプルプログラム

【例 1】のプログラム例です。

```
PROGRAM MAIN
INCLUDE 'mpif.h'                          MPIが使用するインクルードファイルを指定します。
CALL MPI_INIT(IERR)                       MPI環境の初期化処理を行います。
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)  プロセス数NPROCSを取得します。
CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)  自分のランクMYRANKを取得します。
IF (MYRANK == 0) THEN                     ランク0のプロセスは送信バッファーにデータを
  IBUF = 1                                セットします。
ENDIF
CALL MPI_BCAST(IBUF,1,MPI_INTEGER,0,MPI_COMM_WORLD,IERR)  通信を行います。
PRINT *,'MYRANK = ',MYRANK,'IBUF = ',IBUF   通信後のバッファーの内容を書き出します。
CALL MPI_FINALIZE(IERR)                   MPI環境の終了処理を行います。
END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 3 a.out ↵
  MYRANK = 0 IBUF = 1
  MYRANK = 1 IBUF = 1
  MYRANK = 2 IBUF = 1
```

集団通信サブルーチン　MPI_SCATTER

機能 (関連する節：3-3 節)

- コミュニケータ (comm) 内の、1 つの送信元プロセス (root) の送信バッファー (sendbuf) から、全プロセスの受信バッファー (recvbuf) にメッセージを送信します。
- 各宛先プロセスへの送信メッセージの長さは一定で、送信バッファー (sendbuf) の先頭から宛先プロセスのランクが小さい順に送信されます。
- 本サブルーチンは、MPI_GATHER と逆の通信になります。

使用法

```
  CALL MPI_SCATTER(sendbuf, sendcount, sendtype,
recvbuf, recvcount, recvtype, root, comm, ierror)
```

- sendbuf: 送信バッファーの先頭アドレスを指定します。root プロセスのみで意味を持ちます。
- sendcount: 整数。1 つのプロセスに送信する送信メッセージの要素数を指定します。
  root プロセスのみで意味を持ちます。
- sendtype: 整数。送信メッセージのデータ型を指定します。root プロセスのみで意味を持ちます。
- recvbuf: 受信バッファーの先頭アドレスを指定します。
  送信バッファーと受信バッファーの実際に使用する部分は、メモリー上で重なってはいけません (ただし MPI-2 の新機能 MPI_IN_PLACE で回避可能です。詳細は 3-8-1 節を参照)。
- recvcount: 整数。受信メッセージの要素数を指定します。
- recvtype: 整数。受信メッセージのデータ型を指定します。
- root: 整数。送信元プロセスの comm 内でのランクを指定します。
  comm 内の全プロセスが同じ値を指定する必要があります。
- comm: 整数。送受信に参加する全てのプロセスを含むグループのコミュニケータを指定します。
  全プロセスが同じ値を指定する必要があります。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

注意

- 各プロセスへの送信メッセージ間、および、送信メッセージとそれに対応する受信メッセージ間のバイト数は一致している必要があります。この条件を満足していれば、各プロセスで sendcount, sendtype, recvcount, recvtype は異なっていても構いません (ただしデータ型 (INTEGER, REAL など)) は一致している必要があります)。
- 送信元プロセス (root) 以外のプロセスでは、sendbuf, sendcount, sendtype は無視されますので、適当な変数を指定してもかまいません。
- コミュニケータ (comm) 内の全プロセスが本サブルーチンをコールする必要があります。

【例 1】

【例 2】

サンプルプログラム

【例 1】のプログラム例です。

```
  PROGRAM MAIN
  INCLUDE 'mpif.h'                          MPIが使用するインクルードファイルを指定します。
  INTEGER ISEND(3)                          送信バッファーを指定します。
  CALL MPI_INIT(IERR)                       MPI環境の初期化処理を行います。
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)  プロセス数NPROCSを取得します。
  CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)  自分のランクMYRANKを取得します。
  IF (MYRANK == 0) THEN
    DO I = 1, NPROCS                        ランク0のプロセスは送信バッファーにデータを
      ISEND(I) = I                          セットします。
    ENDDO
  ENDIF
  CALL MPI_SCATTER(ISEND,1,MPI_INTEGER,
&                  IRECV,1,MPI_INTEGER,0,MPI_COMM_WORLD,IERR)  通信を行います。
  PRINT *,'MYRANK = ',MYRANK,'IRECV = ',IRECV   通信後の受信バッファーの内容を書き出します。
  CALL MPI_FINALIZE(IERR)                   MPI環境の終了処理を行います。
  END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 3 a.out ↵
  MYRANK = 0 IRECV = 1
  MYRANK = 1 IRECV = 2
  MYRANK = 2 IRECV = 3
```

集団通信サブルーチン　MPI_SCATTERV

機能 (関連する節：3-3 節)

- コミュニケータ (comm) 内の、1 つの送信元プロセス (root) の送信バッファー (sendbuf) から、全プロセスの受信バッファー (recvbuf) にメッセージを送信します。
- MPI_SCATTER と異なり、送信メッセージの長さと送信バッファー (sendbuf) 内の位置（変位）を、宛先プロセスごとに変えることができます。

使用法

```
  CALL MPI_SCATTERV(sendbuf, sendcounts, displs, sendtype,
recvbuf, recvcount, recvtype, root, comm, ierror)
```

- sendbuf: 送信バッファーの先頭アドレスを指定します。root プロセスのみで意味を持ちます。
- sendcounts: 整数配列。ランク i のプロセスに送信する送信メッセージの要素数を、この配列の $i+1$ 番目の要素に指定します (サンプルプログラムに示すように、配列を 0 から開始した方が分かりやすくなります)。root プロセスのみで意味を持ちます。
- displs: 整数配列。ランク i のプロセスに送信する送信メッセージが置かれている送信バッファー sendbuf 内の位置 (変位；【例 1】参照) を、この配列の $i+1$ 番目の要素に指定します (サンプルプログラムに示すように、配列を 0 から開始した方が分かりやすくなります)。
  この値はタイプ sendtype を単位として表します。root プロセスのみで意味を持ちます。
- sendtype: 整数。送信メッセージのデータ型を指定します。root プロセスのみで意味を持ちます。
- recvbuf: 受信バッファーの先頭アドレスを指定します。
  送信バッファーと受信バッファーの実際に使用する部分は、メモリー上で重なってはいけません (ただし MPI-2 の新機能 MPI_IN_PLACE で回避可能です。詳細は 3-8-1 節を参照)。
- recvcount: 整数。受信メッセージの要素数を指定します。
- recvtype: 整数。受信メッセージのデータ型を指定します。
- root: 整数。送信元プロセスの comm 内でのランクを指定します。
  comm 内の全プロセスが同じ値を指定する必要があります。
- comm: 整数。送受信に参加する全てのプロセスを含むグループのコミュニケータを指定します。
  全プロセスが同じ値を指定する必要があります。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

注意

- 各プロセスへの送信メッセージ間でバイト数は異なっていても構いません。送信メッセージとそれに対応する受信メッセージ間のバイト数は一致している必要があります。この条件を満足していれば、各プロセスで sendcounts, sendtype, recvcount, recvtype は異なっていても構いません (ただしデータ型 (INTEGER, REAL など) は一致している必要があります)。
- 送信元プロセス (root) 以外のプロセスでは、sendbuf, sendcounts, displs, sendtype は無視されますので、適当な変数を指定してもかまいません。
- コミュニケータ (comm) 内の全プロセスが本サブルーチンをコールする必要があります。

【例 1】

サンプルプログラム

【例 1】のプログラム例です。

```
  PROGRAM MAIN
  INCLUDE 'mpif.h'                          MPIが使用するインクルードファイルを指定します。
  INTEGER ISEND(6),IRECV(3)                 送受信バッファーを指定します。
  INTEGER ISCNT(0:2),IDISP(0:2)
  DATA ISCNT/1,2,3/ IDISP/0,1,3/            送信メッセージの要素数と変位を指定します。
  CALL MPI_INIT(IERR)                       MPI環境の初期化処理を行います。
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)  プロセス数NPROCSを取得します。
  CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)  自分のランクMYRANKを取得します。
  IF (MYRANK == 0) THEN
    DO I = 1, 6                             ランク0のプロセスは送信バッファーにデータを
      ISEND(I) = I                          セットします。
    ENDDO
  ENDIF
  CALL MPI_SCATTERV(ISEND,ISCNT   ,IDISP,MPI_INTEGER,  通信を行います。
&                   IRECV,MYRANK+1      ,MPI_INTEGER,
&                   0,MPI_COMM_WORLD,IERR)
  PRINT *,'MYRANK = ',MYRANK,'IRECV = ',IRECV   通信後の受信バッファーの内容を書き出します。
  CALL MPI_FINALIZE(IERR)                   MPI環境の終了処理を行います。
  END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 3 a.out ↵
  MYRANK = 0 IRECV = 1 0 0
  MYRANK = 1 IRECV = 2 3 0
  MYRANK = 2 IRECV = 4 5 6
```

集団通信サブルーチン　MPI_GATHER

機能 (関連する節：3-3-5 節)

- コミュニケータ (comm) 内の、全プロセスの送信バッファー (sendbuf) から、1 つの宛先プロセス (root) の受信バッファー (recvbuf) にメッセージを送信します。
- 各送信元プロセスからの受信メッセージの長さは一定で、受信バッファー (recvbuf) の先頭から送信元プロセスのランクが小さい順に入ります。
- 本サブルーチンは MPI_SCATTER と逆の通信になります。

使用法

```
  CALL MPI_GATHER(sendbuf, sendcount, sendtype,
recvbuf, recvcount, recvtype, root, comm, ierror)
```

- sendbuf: 送信バッファーの先頭アドレスを指定します。
- sendcount: 整数。送信メッセージの要素数を指定します。
- sendtype: 整数。送信メッセージのデータ型を指定します。
- recvbuf: 受信バッファーの先頭アドレスを指定します。root プロセスのみで意味を持ちます。
  送信バッファーと受信バッファーの実際に使用する部分は、メモリー上で重なってはいけません (ただし MPI-2 の新機能 MPI_IN_PLACE で回避可能です。詳細は 3-8-1 節を参照)。
- recvcount: 整数。1 つのプロセスからの受信メッセージの要素数を指定します。
  root プロセスのみで意味を持ちます。
- recvtype: 整数。受信メッセージのデータ型を指定します。root プロセスのみで意味を持ちます。
- root: 整数。宛先プロセスの comm 内でのランクを指定します。
  comm 内の全プロセスが同じ値を指定する必要があります。
- comm: 整数。送受信に参加する全てのプロセスを含むグループのコミュニケータを指定します。
  全プロセスが同じ値を指定する必要があります。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

注意

- 各プロセスからの送信メッセージ間、および、送信メッセージとそれに対応する受信メッセージ間のバイト数は一致している必要があります。この条件を満足していれば、各プロセスで sendcount, sendtype, recvcount, recvtype は異なっていても構いません (ただしデータ型 (INTEGER, REAL など) は一致している必要があります)。
- 宛先プロセス (root) 以外のプロセスでは、recvbuf, recvcount, recvtype は無視されますので、適当な変数を指定してもかまいません。
- コミュニケータ (comm) 内の全プロセスが本サブルーチンをコールする必要があります。

【例 1】

【例 2】

サンプルプログラム

【例 1】のプログラム例です。

```
  PROGRAM MAIN
  INCLUDE 'mpif.h'                          MPIが使用するインクルードファイルを指定します。
  INTEGER IRECV(3)                          受信バッファーを指定します。
  CALL MPI_INIT(IERR)                       MPI環境の初期化処理を行います。
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)  プロセス数NPROCSを取得します。
  CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)  自分のランクMYRANKを取得します。
  ISEND = MYRANK + 1                        各プロセスは送信バッファーにデータをセットします。
  CALL MPI_GATHER(ISEND,1,MPI_INTEGER,
&                 IRECV,1,MPI_INTEGER,0,MPI_COMM_WORLD,IERR)  通信を行います。
  IF (MYRANK == 0) PRINT *,'IRECV = ',IRECV   通信後の受信バッファーの内容を書き出します。
  CALL MPI_FINALIZE(IERR)                   MPI環境の終了処理を行います。
  END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 3 a.out ↵
  IRECV = 1 2 3
```

集団通信サブルーチン　MPI_GATHERV

機能 (関連する節：4-6-3-1 節)

- コミュニケータ (comm) 内の、全プロセスの送信バッファー (sendbuf) から、1 つの宛先プロセス (root) の受信バッファー (recvbuf) にメッセージを送信します。
- MPI_GATHER と異なり、受信メッセージの長さと受信バッファー (recvbuf) 内の位置（変位）を、送信元プロセスごとに変えることができます。

使用法

```
  CALL MPI_GATHERV(sendbuf, sendcount, sendtype,
recvbuf, recvcounts, displs, recvtype, root, comm, ierror)
```

- sendbuf: 送信バッファーの先頭アドレスを指定します。
- sendcount: 整数。送信メッセージの要素数を指定します。
- sendtype: 整数。送信メッセージのデータ型を指定します。
- recvbuf: 受信バッファーの先頭アドレスを指定します。root プロセスのみで意味を持ちます。
  送信バッファーと受信バッファーの実際に使用する部分は、メモリー上で重なってはいけません (ただし MPI-2 の新機能 MPI_IN_PLACE で回避可能です。詳細は 3-8-1 節を参照)。
- recvcounts: 整数配列。ランク i のプロセスからの受信メッセージの要素数を、この配列の $i+1$ 番目の要素に指定します (サンプルプログラムに示すように、配列を 0 から開始した方が分かりやすくなります)。root プロセスのみで意味を持ちます。
- displs: 整数配列。ランク i のプロセスからの受信メッセージが置かれる受信バッファー recvbuf 内の位置 (変位；【例 1】参照) を、この配列の $i+1$ 番目の要素に指定します (サンプルプログラムに示すように、配列を 0 から開始した方が分かりやすくなります)。
  この値はタイプ recvtype を単位として表します。root プロセスのみで意味を持ちます。
- recvtype: 整数。受信メッセージのデータ型を指定します。root プロセスのみで意味を持ちます。
- root: 整数。宛先プロセスの comm 内でのランクを指定します。
  comm 内の全プロセスが同じ値を指定する必要があります。
- comm: 整数。送受信に参加する全てのプロセスを含むグループのコミュニケータを指定します。
  全プロセスが同じ値を指定する必要があります。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

注意

- 各プロセスからの送信メッセージ間でバイト数は異なっていても構いません。送信メッセージとそれに対応する受信メッセージ間のバイト数は一致している必要があります。この条件を満足していれば、各プロセスで sendcount, sendtype、recvcounts, recvtype は異なっていても構いません (ただしデータ型 (INTEGER, REAL など) は一致している必要があります)。
- 宛先プロセス (root) 以外のプロセスでは、recvbuf, recvcounts, displs, recvtype は無視されますので、適当な変数を指定してもかまいません。
- 1 回のコールで受信バッファーの同じ位置に複数の送信元プロセスからメッセージを受信しないで下さい。
- コミュニケータ (comm) 内の全プロセスが本サブルーチンをコールする必要があります。

【例 1】

サンプルプログラム

【例 1】のプログラム例です。

```
  PROGRAM MAIN
  INCLUDE 'mpif.h'                          MPIが使用するインクルードファイルを指定します。
  INTEGER ISEND(3),IRECV(6)                 送受信バッファーを指定します。
  INTEGER IRCNT(0:2),IDISP(0:2)
  DATA IRCNT/1,2,3/ IDISP/0,1,3/            受信メッセージの要素数と変位を指定します。
  CALL MPI_INIT(IERR)                       MPI環境の初期化処理を行います。
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)  プロセス数NPROCSを取得します。
  CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)  自分のランクMYRANKを取得します。
  DO I = 1, MYRANK+1                        各プロセスは送信バッファーにデータをセットします。
    ISEND(I) = I + MYRANK*10
  ENDDO
  CALL MPI_GATHERV(ISEND,MYRANK+1      ,MPI_INTEGER,  通信を行います。
&                  IRECV,IRCNT   ,IDISP,MPI_INTEGER,
&                  0,MPI_COMM_WORLD,IERR)
  IF (MYRANK == 0) PRINT *,'IRECV = ',IRECV   通信後の受信バッファーの内容を書き出します。
  CALL MPI_FINALIZE(IERR)                   MPI環境の終了処理を行います。
  END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 3 a.out ↵
  IRECV = 1 11 12 21 22 23
```

集団通信サブルーチン　MPI_ALLGATHER

機能 (関連する節：3-3 節)

- コミュニケータ (comm) 内の、全プロセスの送信バッファー (sendbuf) から、全プロセスの受信バッファー (recvbuf) にお互いにメッセージを送信します。
- 各送信元プロセスからの受信メッセージの長さは一定で、受信バッファー (recvbuf) の先頭から送信元プロセスのランクが小さい順に入ります。
- 本サブルーチンは、メッセージを MPI_GATHER で 1 つのプロセスに収集した後、それを MPI_BCAST で全プロセスに送信したのと機能的に等価です。使用法も『宛先プロセスのランク (root)』の引数がない点を除き、MPI_GATHER と同じです。

使用法

```
  CALL MPI_ALLGATHER(sendbuf, sendcount, sendtype,
recvbuf, recvcount, recvtype, comm, ierror)
```

- sendbuf: 送信バッファーの先頭アドレスを指定します。
- sendcount: 整数。送信メッセージの要素数を指定します。
- sendtype: 整数。送信メッセージのデータ型を指定します。
- recvbuf: 受信バッファーの先頭アドレスを指定します。
  送信バッファーと受信バッファーの実際に使用する部分は、メモリー上で重なってはいけません (ただし MPI-2 の新機能 MPI_IN_PLACE で回避可能です。詳細は 3-8-1 節を参照)。
- recvcount: 整数。1 つのプロセスからの受信メッセージの要素数を指定します。
- recvtype: 整数。受信メッセージのデータ型を指定します。
- comm: 整数。送受信に参加する全てのプロセスを含むグループのコミュニケータを指定します。
  全プロセスが同じ値を指定する必要があります。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

注意

- 各プロセスからの送信メッセージ間、および、送信メッセージとそれに対応する受信メッセージ間のバイト数は一致している必要があります。この条件を満足していれば、各プロセスで sendcount, sendtype, recvcount, recvtype は異なっていても構いません (ただしデータ型 (INTEGER, REAL など) は一致している必要があります)。
- コミュニケータ (comm) 内の全プロセスが本サブルーチンをコールする必要があります。

【例 1】

【例 2】

サンプルプログラム

【例 1】のプログラム例です。

```
  PROGRAM MAIN
  INCLUDE 'mpif.h'                          MPIが使用するインクルードファイルを指定します。
  INTEGER IRECV(3)                          受信バッファーを指定します。
  CALL MPI_INIT(IERR)                       MPI環境の初期化処理を行います。
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)  プロセス数NPROCSを取得します。
  CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)  自分のランクMYRANKを取得します。
  ISEND = MYRANK + 1                        各プロセスは送信バッファーにデータをセットします。
  CALL MPI_ALLGATHER(ISEND,1,MPI_INTEGER,
&                    IRECV,1,MPI_INTEGER,MPI_COMM_WORLD,IERR)  通信を行います。
  PRINT *,'MYRANK = ',MYRANK,'IRECV = ',IRECV   通信後の受信バッファーの内容を書き出します。
  CALL MPI_FINALIZE(IERR)                   MPI環境の終了処理を行います。
  END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 3 a.out ↵
  MYRANK = 0 IRECV = 1 2 3
  MYRANK = 1 IRECV = 1 2 3
  MYRANK = 2 IRECV = 1 2 3
```

集団通信サブルーチン　MPI_ALLGATHERV

機能 (関連する節：4-6-3-2 節)

- コミュニケータ (comm) 内の、全プロセスの送信バッファー (sendbuf) から、全プロセスの受信バッファー (recvbuf) にお互いにメッセージを送信します。
- MPI_ALLGATHER と異なり、受信メッセージの長さと受信バッファー (recvbuf) 内の位置（変位）を、送信元プロセスごとに変えることができます。
- 使用法は『宛先プロセスのランク (root)』の引数がない点を除き、MPI_GATHERV と同じです。

使用法

```
  CALL MPI_ALLGATHERV(sendbuf, sendcount, sendtype,
recvbuf, recvcounts, displs, recvtype, comm, ierror)
```

- sendbuf: 送信バッファーの先頭アドレスを指定します。
- sendcount: 整数。送信メッセージの要素数を指定します。
- sendtype: 整数。送信メッセージのデータ型を指定します。
- recvbuf: 受信バッファーの先頭アドレスを指定します。
  送信バッファーと受信バッファーの実際に使用する部分は、メモリー上で重なってはいけません (ただし MPI-2 の新機能 MPI_IN_PLACE で回避可能です。詳細は 3-8-1 節を参照)。
- recvcounts: 整数配列。ランク i のプロセスからの受信メッセージの要素数を、この配列の $i+1$ 番目の要素に指定します (サンプルプログラムに示すように、配列を 0 から開始した方が分かりやすくなります)。
- displs: 整数配列。ランク i のプロセスからの受信メッセージが置かれる受信バッファー recvbuf 内の位置 (変位；【例 1】参照) を、この配列の $i+1$ 番目の要素に指定します (サンプルプログラムに示すように、配列を 0 から開始した方が分かりやすくなります)。
  この値はタイプ recvtype を単位として表します。
- recvtype: 整数。受信メッセージのデータ型を指定します。
- comm: 整数。送受信に参加する全てのプロセスを含むグループのコミュニケータを指定します。
  全プロセスが同じ値を指定する必要があります。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

注意

- 各プロセスからの送信メッセージ間でバイト数は異なっていても構いません。送信メッセージとそれに対応する受信メッセージ間のバイト数は一致している必要があります。この条件を満足していれば、各プロセスで sendcount, sendtype、recvcounts, recvtype は異なっていても構いません (ただしデータ型 (INTEGER, REAL など) は一致している必要があります)。
- 1 回のコールで受信バッファーの同じ位置に複数の送信元プロセスからメッセージを受信しないで下さい。
- コミュニケータ (comm) 内の全プロセスが本サブルーチンをコールする必要があります。

【例 1】

サンプルプログラム

【例 1】のプログラム例です。

```
      PROGRAM MAIN
      INCLUDE 'mpif.h'                          MPIが使用するインクルードファイルを指定します。
      INTEGER ISEND(3),IRECV(6)                 送受信バッファーを指定します。
      INTEGER IRCNT(0:2),IDISP(0:2)
      DATA IRCNT/1,2,3/ IDISP/0,1,3/            受信メッセージの要素数と変位を指定します。
      CALL MPI_INIT(IERR)                       MPI環境の初期化処理を行います。
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)  プロセス数NPROCSを取得します。
      CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)  自分のランクMYRANKを取得します。
      DO I = 1, MYRANK+1                        各プロセスは送信バッファーにデータをセットします。
        ISEND(I) = I + MYRANK*10
      ENDDO
      CALL MPI_ALLGATHERV(ISEND,MYRANK+1      ,MPI_INTEGER,  通信を行います。
     &                    IRECV,IRCNT    ,IDISP,MPI_INTEGER,
     &                    MPI_COMM_WORLD,IERR)
      PRINT *,'MYRANK = ',MYRANK,'IRECV = ',IRECV   通信後の受信バッファーの内容を書き出します。
      CALL MPI_FINALIZE(IERR)                   MPI環境の終了処理を行います。
      END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 3 a.out ↵
  MYRANK = 0 IRECV = 1 11 12 21 22 23
  MYRANK = 1 IRECV = 1 11 12 21 22 23
  MYRANK = 2 IRECV = 1 11 12 21 22 23
```

集団通信サブルーチン　MPI_ALLTOALL

機能 (関連する節：4-6-10 節)

- コミュニケータ (comm) 内の、全プロセスの送信バッファー (sendbuf) から、全プロセスの受信バッファー (recvbuf) にお互いにメッセージを送信します。
- 各宛先プロセスへの送信メッセージの長さは一定で、送信バッファー (sendbuf) の先頭から宛先プロセスのランクが小さい順に送信されます。
- 各送信元プロセスからの受信メッセージの長さは一定で、受信バッファー (recvbuf) の先頭から送信元プロセスのランクが小さい順に入ります。
- 送信側に着目すると MPI_SCATTER を行い (例えば【例 1】で、ランク 0 の送信バッファーに入っている 1，2，3 が各プロセスに送られます)、受信側に着目すると MPI_GATHER を行っています (例えば【例 1】で、各プロセスから送られた 1，11，21 がランク 0 の受信バッファーに入ります)。

使用法

```
  CALL MPI_ALLTOALL(sendbuf, sendcount, sendtype,
recvbuf, recvcount, recvtype, comm, ierror)
```

- sendbuf: 送信バッファーの先頭アドレスを指定します。
- sendcount: 整数。1 つのプロセスに送信する送信メッセージの要素数を指定します。
- sendtype: 整数。送信メッセージのデータ型を指定します。
- recvbuf: 受信バッファーの先頭アドレスを指定します。
  送信バッファーと受信バッファーの実際に使用する部分は、メモリー上で重なってはいけません (MPI-2 の MPI_IN_PLACE を使用することはできません)。
- recvcount: 整数。1 つのプロセスからの受信メッセージの要素数を指定します。
- recvtype: 整数。受信メッセージのデータ型を指定します。
- comm: 整数。送受信に参加する全てのプロセスを含むグループのコミュニケータを指定します。
  全プロセスが同じ値を指定する必要があります。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

注意

- 各プロセスへの送信メッセージ間、および、送信メッセージとそれに対応する受信メッセージ間のバイト数は一致している必要があります。この条件を満足していれば、各プロセスで sendcount，sendtype，recvcount，recvtype は異なっていても構いません (ただしデータ型 (INTEGER，REAL など) は一致している必要があります)。
- コミュニケータ (comm) 内の全プロセスが本サブルーチンをコールする必要があります。

【例 1】

【例 2】

サンプルプログラム

【例 1】のプログラム例です。

```
  PROGRAM MAIN
  INCLUDE 'mpif.h'                          MPIが使用するインクルードファイルを指定します。
  INTEGER ISEND(3),IRECV(3)                 送受信バッファーを指定します。
  CALL MPI_INIT(IERR)                       MPI環境の初期化処理を行います。
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)  プロセス数NPROCSを取得します。
  CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)  自分のランクMYRANKを取得します。
  DO I = 1, NPROCS                          各プロセスは送信バッファーにデータをセットします。
    ISEND(I) = I + MYRANK*10
  ENDDO
  CALL MPI_ALLTOALL(ISEND,1,MPI_INTEGER,
&                   IRECV,1,MPI_INTEGER,MPI_COMM_WORLD,IERR)  通信を行います。
  PRINT *,'MYRANK = ',MYRANK,'IRECV = ',IRECV   通信後の受信バッファーの内容を書き出します。
  CALL MPI_FINALIZE(IERR)                   MPI環境の終了処理を行います。
  END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 3 a.out ↵
  MYRANK = 0 IRECV = 1 11 21
  MYRANK = 1 IRECV = 2 12 22
  MYRANK = 2 IRECV = 3 13 23
```

集団通信サブルーチン　MPI_ALLTOALLV

機能 (関連する節：4-6-3-3 節，4-6-10 節，5-2-3 節)

- コミュニケータ (comm) 内の、全プロセスの送信バッファー (sendbuf) から、全プロセスの受信バッファー (recvbuf) にお互いにメッセージを送信します。
- MPI_ALLTOALL とは異なり、送信メッセージの長さと送信バッファー (sendbuf) 内の位置（変位）を、宛先プロセスごとに変えることができます。
- MPI_ALLTOALL とは異なり、受信メッセージの長さと受信バッファー (recvbuf) 内の位置（変位）を、送信元プロセスごとに変えることができます。
- 【例 1】では、全プロセスの送信バッファーが同じ構造になっていますが、異なっていても構いません。

使用法

```
  CALL MPI_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype,
recvbuf, recvcounts, rdispls, recvtype, comm, ierror)
```

- sendbuf: 送信バッファーの先頭アドレスを指定します。
- sendcounts: 整数配列。ランク i のプロセスに送信する送信メッセージの要素数を、この配列の $i+1$ 番目の要素に指定します (サンプルプログラムに示すように、配列を 0 から開始した方が分かりやすくなります)。
- sdispls: 整数配列。ランク i のプロセスに送信する送信メッセージが置かれている送信バッファー sendbuf 内の位置 (変位；【例 1】参照) を、この配列の $i+1$ 番目の要素に指定します (サンプルプログラムに示すように、配列を 0 から開始した方が分かりやすくなります)。
  この値はタイプ sendtype を単位として表します。
- sendtype: 整数。送信メッセージのデータ型を指定します。
- recvbuf: 受信バッファーの先頭アドレスを指定します。
  送信バッファーと受信バッファーの実際に使用する部分は、メモリー上で重なってはいけません (MPI-2 の MPI_IN_PLACE を使用することはできません)。
- recvcounts: 整数配列。ランク i のプロセスからの受信メッセージの要素数を、この配列の $i+1$ 番目の要素に指定します (サンプルプログラムに示すように、配列を 0 から開始した方が分かりやすくなります)。
- rdispls: 整数配列。ランク i のプロセスからの受信メッセージが置かれる受信バッファー recvbuf 内の位置 (変位；【例 1】参照) を、この配列の $i+1$ 番目の要素に指定します (サンプルプログラムに示すように、配列を 0 から開始した方が分かりやすくなります)。
  この値はタイプ recvtype を単位として表します。
- recvtype: 整数。受信メッセージのデータ型を指定します。
- comm: 整数。送受信に参加する全てのプロセスを含むグループのコミュニケータを指定します。
  全プロセスが同じ値を指定する必要があります。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

注意

- 各プロセスへの送信メッセージ間でバイト数は異なっても構いません。送信メッセージとそれに対応する受信メッセージ間のバイト数は一致している必要があります。この条件を満足していれば、各プロセスで sendcounts，sendtype，recvcounts，recvtype は異なっていても構いません (ただしデータ型 (INTEGER，REAL など) は一致している必要があります)。
- 1 回のコールで受信バッファーの同じ位置に複数の送信元プロセスからメッセージを受信しないで下さい。
- コミュニケータ (comm) 内の全プロセスが本サブルーチンをコールする必要があります。

【例 1】

サンプルプログラム

【例 1】のプログラム例です。

```
      PROGRAM MAIN
      INCLUDE 'mpif.h'                          MPIが使用するインクルードファイルを指定します。
      INTEGER ISEND(6),IRECV(9)                 送受信バッファーを指定します。
      INTEGER ISCNT(0:2),ISDSP(0:2),IRCNT(0:2),IRDSP(0:2)
      DATA ISCNT/1,2,3/ ISDSP/0,1,3/            送信メッセージの要素数と変位を指定します。
      CALL MPI_INIT(IERR)                       MPI環境の初期化処理を行います。
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)  プロセス数NPROCSを取得します。
      CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)  自分のランクMYRANKを取得します。
      DO I = 0, NPROCS-1
        IRCNT(I) = 1 + MYRANK                   受信メッセージの要素数を指定します。
        IRDSP(I) = I*(MYRANK+1)                 受信メッセージの変位を指定します。
      ENDDO
      DO I = 1, 6
        ISEND(I) = I + MYRANK*10                各プロセスは送信バッファーにデータをセットします。
      ENDDO
      CALL MPI_ALLTOALLV(ISEND,ISCNT,ISDSP,MPI_INTEGER,  通信を行います。
     &                   IRECV,IRCNT,IRDSP,MPI_INTEGER,
     &                   MPI_COMM_WORLD,IERR)
      PRINT *,'MYRANK = ',MYRANK,'IRECV = ',IRECV   通信後の受信バッファーの内容を書き出します。
      CALL MPI_FINALIZE(IERR)                   MPI環境の終了処理を行います。
      END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 3 a.out ↵
  MYRANK = 0 IRECV = 1 11 21 0 0 0 0 0 0
  MYRANK = 1 IRECV = 2 3 12 13 22 23 0 0 0
  MYRANK = 2 IRECV = 4 5 6 14 15 16 24 25 26
```

【MPI-2】集団通信サブルーチン　MPI_ALLTOALLW

機能 (関連する節：4-6-3-3 節)

- 本サブルーチンは MPI_ALLTOAALV とほぼ同じで、「使用法」の波線の部分だけが異なります。
- コミュニケータ (comm) 内の、全プロセスの送信バッファー (sendbuf) から、全プロセスの受信バッファー (recvbuf) にお互いにメッセージを送信します。
- MPI_ALLTOALL とは異なり、送信メッセージの長さ、送信バッファー (sendbuf) 内の位置（変位)、およびデータ型を、宛先プロセスごとに変えることができます。
- MPI_ALLTOALL とは異なり、受信メッセージの長さ、受信バッファー (recvbuf) 内の位置（変位)、およびデータ型を、送信元プロセスごとに変えることができます。
- 【例 1】では、全プロセスの送信バッファーが同じ構造になっていますが、異なっていても構いません。

使用法

```
  CALL MPI_ALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes,
recvbuf, recvcounts, rdispls, recvtypes, comm, ierror)
```

- sendbuf: 送信バッファーの先頭アドレスを指定します。
- sendcounts: 整数配列。ランク i のプロセスに送信する送信メッセージの要素数を、この配列の $i+1$ 番目の要素に指定します (注 1 を参照して下さい)。
- sdispls: 整数配列。ランク i のプロセスに送信する送信メッセージが置かれている送信バッファー sendbuf 内の位置 (変位；【例 1】参照) を、この配列の $i+1$ 番目の要素に指定します (注 1 を参照して下さい)。この値はバイトを単位として表します。
- sendtypes: 整数配列。ランク i のプロセスに送信する送信メッセージのデータ型を、この配列の $i+1$ 番目の要素に指定します (注 1 を参照して下さい)。
- recvbuf: 受信バッファーの先頭アドレスを指定します。
  送信バッファーと受信バッファーの実際に使用する部分は、メモリー上で重なってはいけません (MPI-2 の MPI_IN_PLACE を使用することはできません)。
- recvcounts: 整数配列。ランク i のプロセスからの受信メッセージの要素数を、この配列の $i+1$ 番目の要素に指定します (注 1 を参照して下さい)。
- rdispls: 整数配列。ランク i のプロセスからの受信メッセージが置かれる受信バッファー recvbuf 内の位置 (変位；【例 1】参照) を、この配列の $i+1$ 番目の要素に指定します (注 1 を参照して下さい)。この値はバイトを単位として表します。
- recvtypes: 整数配列。ランク i のプロセスからの受信メッセージのデータ型を、この配列の $i+1$ 番目の要素に指定します (注 1 を参照して下さい)。
- comm: 整数。送受信に参加する全てのプロセスを含むグループのコミュニケータを指定します。
  全プロセスが同じ値を指定する必要があります。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

注意

- (注 1) サンプルプログラムに示すように、配列を 0 から開始した方が分かりやすくなります。
- 各プロセスへの送信メッセージ間でバイト数は異なっても構いません。送信メッセージとそれに対応する受信メッセージ間のバイト数は一致している必要があります。この条件を満足していれば、各プロセスで sendcounts, sendtypes, recvcounts, recvtypes は異なっていても構いません (ただしデータ型 (INTEGER, REAL など) は一致している必要があります)。
- 1 回のコールで受信バッファーの同じ位置に複数の送信元プロセスからメッセージを受信しないで下さい。
- コミュニケータ (comm) 内の全プロセスが本サブルーチンをコールする必要があります。

【例 1】

サンプルプログラム

【例 1】のプログラム例です。MPI_ALLTOALLV の例と異なる部分を波線で示します。

```
  PROGRAM MAIN
  INCLUDE 'mpif.h'                          MPIが使用するインクルードファイルを指定します。
  INTEGER ISEND(6),IRECV(9)                 送受信バッファーを指定します。
  INTEGER ISCNT(0:2),ISDSP(0:2),IRCNT(0:2),IRDSP(0:2)
  INTEGER ISTYPE(0:2),IRTYPE(0:2)
  DATA ISCNT/1,2,3/ ISDSP/0,4,12/           送信メッセージの要素数と変位を指定します。
  CALL MPI_INIT(IERR)                       MPI環境の初期化処理を行います。
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)  プロセス数NPROCSを取得します。
  CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)  自分のランクMYRANKを取得します。
  DO I = 0, NPROCS-1
    IRCNT(I) = 1 + MYRANK                   受信メッセージの要素数を指定します。
    IRDSP(I) = I*(MYRANK+1)*4               受信メッセージの変位を指定します。
    ISTYPE(I) = MPI_INTEGER                 送信メッセージのデータ型を指定します。
    IRTYPE(I) = MPI_INTEGER                 受信メッセージのデータ型を指定します。
  ENDDO
  DO I = 1, 6
    ISEND(I) = I + MYRANK*10                各プロセスは送信バッファーにデータをセットします。
  ENDDO
  CALL MPI_ALLTOALLW(ISEND,ISCNT,ISDSP,ISTYPE,  通信を行います。
&                    IRECV,IRCNT,IRDSP,IRTYPE,MPI_COMM_WORLD,IERR)
  PRINT *,'MYRANK = ',MYRANK,'IRECV = ',IRECV   通信後の受信バッファーの内容を書き出します。
  CALL MPI_FINALIZE(IERR)                   MPI環境の終了処理を行います。
  END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 3 a.out ↵
  MYRANK = 0 IRECV = 1 11 21 0 0 0 0 0 0
  MYRANK = 1 IRECV = 2 3 12 13 22 23 0 0 0
  MYRANK = 2 IRECV = 4 5 6 14 15 16 24 25 26
```

集団通信サブルーチン　MPI_REDUCE

機能 (関連する節：3-3-6 節，4-6-4 節，4-6-6-1 節)

- コミュニケータ (comm) 内の、全プロセスの送信バッファー (sendbuf) のメッセージが、通信しながら演算 (op) され、結果が 1 つの宛先プロセス (root) の受信バッファー (recvbuf) に入ります。
- 送信バッファーが配列の場合は、配列の対応する要素ごとに演算が行われます。
- MPI の基本データ型 (datatype) と MPI の定義済み演算 (op) の組み合わせを以下に示します。演算 (op) をユーザーが自分で定義することもできます。

MPI の基本データ型	MPI の定義済み演算
MPI_INTEGER (注 1) MPI_REAL (注 2) MPI_DOUBLE_PRECISION (注 3) MPI_COMPLEX (注 4)	MPI_SUM（合計） MPI_PROD（積）
MPI_INTEGER (注 1) MPI_REAL (注 2) MPI_DOUBLE_PRECISION (注 3)	MPI_MAX（最大） MPI_MIN（最小）
MPI_2INTEGER MPI_2REAL MPI_2DOUBLE_PRECISION (注 5)	MPI_MAXLOC（最大と位置） MPI_MINLOC（最小と位置）

MPI の基本データ型	MPI の定義済み演算
MPI_LOGICAL	MPI_LAND (論理 AND) MPI_LOR (論理 OR) MPI_LXOR (論理 XOR)
MPI_INTEGER (注 1) MPI_BYTE	MPI_BAND (ビット AND) MPI_BOR (ビット OR) MPI_BXOR (ビット XOR)

下記が使用できるマシン環境もあります。
(注 1) MPI_INTEGER4
(注 2) MPI_REAL4
(注 3) MPI_REAL8
(注 4) MPI_COMPLEX16, MPI_DOUBLE_COMPLEX
(注 5) MPI_2REAL8

使用法

```
CALL MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
```

- sendbuf: 送信バッファーの先頭アドレスを指定します。
- recvbuf: 受信バッファーの先頭アドレスを指定します。root プロセスのみで意味を持ちます。
  送信バッファーと受信バッファーの実際に使用する部分は、メモリー上で重なってはいけません (ただし MPI-2 の新機能 MPI_IN_PLACE で回避可能です。詳細は 3-8-1 節を参照)。
- count: 整数。送（受）信メッセージの要素数を指定します。
  comm 内の全プロセスが同じ値を指定する必要があります。
- datatype: 整数。送（受）信メッセージのデータ型を指定します。
  comm 内の全プロセスが同じ値を指定する必要があります。
- op: 整数。演算の種類を指定します。
  comm 内の全プロセスが同じ値を指定する必要があります。
- root: 整数。宛先プロセスの comm 内でのランクを指定します。
  comm 内の全プロセスが同じ値を指定する必要があります。
- comm: 整数。送受信に参加する全てのプロセスを含むグループのコミュニケータを指定します。
  全プロセスが同じ値を指定する必要があります。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

注意

- 宛先プロセス (root) 以外のプロセスでは、recvbuf は無視されますので、適当な変数を指定してもかまいません。
- コミュニケータ (comm) 内の全プロセスが本サブルーチンをコールする必要があります。

【例 1】

【例 2】

サンプルプログラム

【例 1】のプログラム例です。

```
   PROGRAM MAIN
   INCLUDE 'mpif.h'                          MPIが使用するインクルードファイルを指定します。
   CALL MPI_INIT(IERR)                       MPI環境の初期化処理を行います。
   CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)  プロセス数NPROCSを取得します。
   CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)  自分のランクMYRANKを取得します。
   ISEND = MYRANK + 1                        各プロセスは送信バッファーにデータをセットします。
   CALL MPI_REDUCE(ISEND,IRECV,1,MPI_INTEGER,
 &                 MPI_SUM,0,MPI_COMM_WORLD,IERR)  通信を行います。
   IF (MYRANK == 0) PRINT *,'IRECV = ',IRECV   通信後の受信バッファーの内容を書き出します。
   CALL MPI_FINALIZE(IERR)                   MPI環境の終了処理を行います。
   END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 3 a.out ↵
  IRECV = 6
```

集団通信サブルーチン　MPI_ALLREDUCE

機能 (関連する節：4-6-4 節，4-6-6-1 節，5-8 節)

- コミュニケータ (comm) 内の、全プロセスの送信バッファー (sendbuf) のメッセージが、通信しながら演算 (op) され、結果が全プロセスの受信バッファー (recvbuf) に入ります。
- 送信バッファーが配列の場合は、配列の対応する要素ごとに演算が行われます。
- 本サブルーチンは、メッセージを MPI_REDUCE で 1 つのプロセスに収集した後、それを MPI_BCAST で全プロセスに送信したのと機能的に等価です。使用法も『宛先プロセスのランク (root)』の引数がない点を除き、MPI_REDUCE と同じです。
- MPI の基本データ型 (datatype) と MPI の定義済み演算 (op) の組み合わせを以下に示します。演算 (op) をユーザーが自分で定義することもできます。

MPI の基本データ型	MPI の定義済み演算
MPI_INTEGER (注 1) MPI_REAL (注 2) MPI_DOUBLE_PRECISION (注 3) MPI_COMPLEX (注 4)	MPI_SUM（合計） MPI_PROD（積）
MPI_INTEGER (注 1) MPI_REAL (注 2) MPI_DOUBLE_PRECISION (注 3)	MPI_MAX（最大） MPI_MIN（最小）
MPI_2INTEGER MPI_2REAL MPI_2DOUBLE_PRECISION (注 5)	MPI_MAXLOC（最大と位置） MPI_MINLOC（最小と位置）

MPI の基本データ型	MPI の定義済み演算
MPI_LOGICAL	MPI_LAND (論理 AND) MPI_LOR (論理 OR) MPI_LXOR (論理 XOR)
MPI_INTEGER (注 1) MPI_BYTE	MPI_BAND (ビット AND) MPI_BOR (ビット OR) MPI_BXOR (ビット XOR)

下記が使用できるマシン環境もあります。
(注 1) MPI_INTEGER4
(注 2) MPI_REAL4
(注 3) MPI_REAL8
(注 4) MPI_COMPLEX16, MPI_DOUBLE_COMPLEX
(注 5) MPI_2REAL8

使用法

```
CALL MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm, ierror)
```

- sendbuf: 送信バッファーの先頭アドレスを指定します。
- recvbuf: 受信バッファーの先頭アドレスを指定します。
  送信バッファーと受信バッファーの実際に使用する部分は、メモリー上で重なってはいけません (ただし MPI-2 の新機能 MPI_IN_PLACE で回避可能です。詳細は 3-8-1 節を参照)。
- count: 整数。送（受）信メッセージの要素数を指定します。
  comm 内の全プロセスが同じ値を指定する必要があります。
- datatype: 整数。送（受）信メッセージのデータ型を指定します。
  comm 内の全プロセスが同じ値を指定する必要があります。
- op: 整数。演算の種類を指定します。
  comm 内の全プロセスが同じ値を指定する必要があります。
- comm: 整数。送受信に参加する全てのプロセスを含むグループのコミュニケータを指定します。
  全プロセスが同じ値を指定する必要があります。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

注意

- コミュニケータ (comm) 内の全プロセスが本サブルーチンをコールする必要があります。

【例 1】

【例 2】

サンプルプログラム

【例 1】のプログラム例です。

```
PROGRAM MAIN
INCLUDE 'mpif.h'                          MPIが使用するインクルードファイルを指定します。
CALL MPI_INIT(IERR)                       MPI環境の初期化処理を行います。
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)  プロセス数NPROCSを取得します。
CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)  自分のランクMYRANKを取得します。
ISEND = MYRANK + 1                        各プロセスは送信バッファーにデータをセットします。
CALL MPI_ALLREDUCE(ISEND,IRECV,1,MPI_INTEGER,
&                  MPI_SUM,MPI_COMM_WORLD,IERR)  通信を行います。
PRINT *,'MYRANK = ',MYRANK,'IRECV = ',IRECV   通信後の受信バッファーの内容を書き出します。
CALL MPI_FINALIZE(IERR)                   MPI環境の終了処理を行います。
END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 3 a.out ↵
MYRANK = 0 IRECV = 6
MYRANK = 1 IRECV = 6
MYRANK = 2 IRECV = 6
```

集団通信サブルーチン　MPI_SCAN

機能 (関連する節：4-6-9 節)

- コミュニケータ (comm) 内の、ランク 0〜ランク i までの各プロセスの送信バッファー (sendbuf) のメッセージが、通信しながら演算 (op) され、結果がランク i のプロセスの受信バッファー (recvbuf) に入ります。これがコミュニケータ (comm) 内の全プロセスに対して行われます。
- 送信バッファーが配列の場合は、配列の対応する要素ごとに演算が行われます。
- MPI の基本データ型 (datatype) と MPI の定義済み演算 (op) の組み合わせを以下に示します。演算 (op) をユーザーが自分で定義することもできます。

MPI の基本データ型	MPI の定義済み演算
MPI_INTEGER (注 1) MPI_REAL (注 2) MPI_DOUBLE_PRECISION (注 3) MPI_COMPLEX (注 4)	MPI_SUM（合計） MPI_PROD（積）
MPI_INTEGER (注 1) MPI_REAL (注 2) MPI_DOUBLE_PRECISION (注 3)	MPI_MAX（最大） MPI_MIN（最小）
MPI_2INTEGER MPI_2REAL MPI_2DOUBLE_PRECISION (注 5)	MPI_MAXLOC（最大と位置） MPI_MINLOC（最小と位置）

MPI の基本データ型	MPI の定義済み演算
MPI_LOGICAL	MPI_LAND (論理 AND) MPI_LOR (論理 OR) MPI_LXOR (論理 XOR)
MPI_INTEGER (注 1) MPI_BYTE	MPI_BAND (ビット AND) MPI_BOR (ビット OR) MPI_BXOR (ビット XOR)

下記が使用できるマシン環境もあります。
(注 1) MPI_INTEGER4
(注 2) MPI_REAL4
(注 3) MPI_REAL8
(注 4) MPI_COMPLEX16, MPI_DOUBLE_COMPLEX
(注 5) MPI_2REAL8

使用法

```
CALL MPI_SCAN(sendbuf, recvbuf, count, datatype, op, comm, ierror)
```

- sendbuf: 送信バッファーの先頭アドレスを指定します。
- recvbuf: 受信バッファーの先頭アドレスを指定します。
  送信バッファーと受信バッファーの実際に使用する部分は、メモリー上で重なってはいけません (ただし MPI-2 の新機能 MPI_IN_PLACE で回避可能です。詳細は 3-8-1 節を参照)。
- count: 整数。送（受）信メッセージの要素数を指定します。
  comm 内の全プロセスが同じ値を指定する必要があります。
- datatype: 整数。送（受）信メッセージのデータ型を指定します。
  comm 内の全プロセスが同じ値を指定する必要があります。
- op: 整数。演算の種類を指定します。
  comm 内の全プロセスが同じ値を指定する必要があります。
- comm: 整数。送受信に参加する全てのプロセスを含むグループのコミュニケータを指定します。
  全プロセスが同じ値を指定する必要があります。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

注意

- コミュニケータ (comm) 内の全プロセスが本サブルーチンをコールする必要があります。

【例 1】

【例 2】

サンプルプログラム

【例 1】のプログラム例です。

```
   PROGRAM MAIN
   INCLUDE 'mpif.h'                          MPIが使用するインクルードファイルを指定します。
   CALL MPI_INIT(IERR)                       MPI環境の初期化処理を行います。
   CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)  プロセス数NPROCSを取得します。
   CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)  自分のランクMYRANKを取得します。
   ISEND = MYRANK + 1                        各プロセスは送信バッファーにデータをセットします。
   CALL MPI_SCAN(ISEND,IRECV,1,MPI_INTEGER,
 &               MPI_SUM,MPI_COMM_WORLD,IERR)  通信を行います。
   PRINT *,'MYRANK = ',MYRANK,'IRECV = ',IRECV   通信後の受信バッファーの内容を書き出します。
   CALL MPI_FINALIZE(IERR)                   MPI環境の終了処理を行います。
   END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 3 a.out ↵
  MYRANK = 0 IRECV = 1
  MYRANK = 1 IRECV = 3
  MYRANK = 2 IRECV = 6
```

【MPI-2】集団通信サブルーチン　MPI_EXSCAN

機能 (関連する節：4-6-9 節)

- 通信に参加するプロセスのランクが 0〜k とします。コミュニケータ (comm) 内の、ランク 0〜ランク $i-1$ までの各プロセスの送信バッファー (sendbuf) のメッセージが、通信しながら演算 (op) され、結果がランク i のプロセスの受信バッファー (recvbuf) に入ります。これがコミュニケータ (comm) 内のランク 1〜ランク k のプロセスに対して行われます。ランク k のプロセスの送信バッファーの値は使用されません。またランク 0 のプロセスの受信バッファーに値は設定されません。
- 送信バッファーが配列の場合は、配列の対応する要素ごとに演算が行われます。
- MPI の基本データ型 (datatype) と MPI の定義済み演算 (op) の組み合わせを以下に示します。演算 (op) をユーザーが自分で定義することもできます。

MPI の基本データ型	MPI の定義済み演算
MPI_INTEGER (注 1) MPI_REAL (注 2) MPI_DOUBLE_PRECISION (注 3) MPI_COMPLEX (注 4)	MPI_SUM（合計） MPI_PROD（積）
MPI_INTEGER (注 1) MPI_REAL (注 2) MPI_DOUBLE_PRECISION (注 3)	MPI_MAX（最大） MPI_MIN（最小）
MPI_2INTEGER MPI_2REAL MPI_2DOUBLE_PRECISION (注 5)	MPI_MAXLOC（最大と位置） MPI_MINLOC（最小と位置）

MPI の基本データ型	MPI の定義済み演算
MPI_LOGICAL	MPI_LAND (論理 AND) MPI_LOR (論理 OR) MPI_LXOR (論理 XOR)
MPI_INTEGER (注 1) MPI_BYTE	MPI_BAND (ビット AND) MPI_BOR (ビット OR) MPI_BXOR (ビット XOR)

下記が使用できるマシン環境もあります。
(注 1) MPI_INTEGER4
(注 2) MPI_REAL4
(注 3) MPI_REAL8
(注 4) MPI_COMPLEX16, MPI_DOUBLE_COMPLEX
(注 5) MPI_2REAL8

使用法

```
CALL MPI_EXSCAN(sendbuf, recvbuf, count, datatype, op, comm, ierror)
```

- sendbuf: 送信バッファーの先頭アドレスを指定します。
- recvbuf: 受信バッファーの先頭アドレスを指定します。
  送信バッファーと受信バッファーの実際に使用する部分は、メモリー上で重なってはいけません ((MPI-2 の MPI_IN_PLACE を使用することはできません)。
- count: 整数。送（受）信メッセージの要素数を指定します。
  comm 内の全プロセスが同じ値を指定する必要があります。
- datatype: 整数。送（受）信メッセージのデータ型を指定します。
  comm 内の全プロセスが同じ値を指定する必要があります。
- op: 整数。演算の種類を指定します。
  comm 内の全プロセスが同じ値を指定する必要があります。
- comm: 整数。送受信に参加する全てのプロセスを含むグループのコミュニケータを指定します。
  全プロセスが同じ値を指定する必要があります。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

注意

- コミュニケータ (comm) 内の全プロセスが本サブルーチンをコールする必要があります。

【例 1】

【例 2】

サンプルプログラム

【例 1】のプログラム例です。

```
PROGRAM MAIN
INCLUDE 'mpif.h'                                MPIが使用するインクルードファイルを指定します。
CALL MPI_INIT(IERR)                             MPI環境の初期化処理を行います。
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)  プロセス数NPROCSを取得します。
CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)  自分のランクMYRANKを取得します。
ISEND = MYRANK + 1                              各プロセスは送信バッファーにデータをセットします。
CALL MPI_EXSCAN(ISEND,IRECV,1,MPI_INTEGER,
&               MPI_SUM,MPI_COMM_WORLD,IERR)    通信を行います。
PRINT *,'MYRANK = ',MYRANK,'IRECV = ',IRECV     通信後の受信バッファーの内容を書き出します。
CALL MPI_FINALIZE(IERR)                         MPI環境の終了処理を行います。
END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 4 a.out ↵
  MYRANK = 0 IRECV = (未設定)
  MYRANK = 1 IRECV = 1
  MYRANK = 2 IRECV = 3
  MYRANK = 3 IRECV = 6
```

集団通信サブルーチン　MPI_REDUCE_SCATTER

機能 (関連する節：4-6-6-2 節，5-2-2 節)

- コミュニケータ (comm) 内の、全プロセスの送信バッファー (sendbuf) のメッセージが、通信しながら演算 (op) されます。計算結果は、配列 recvcounts に指定した要素ずつ、ランクの小さい順に各プロセスの受信バッファー (recvbuf) に送信されます。例えば計算結果の最初から recvcounts(1) 個の要素はランク 0 のプロセスに、次の recvcounts(2) 個の要素はランク 1 のプロセスに…、のように送信されます。
- 送信バッファーが配列の場合は、配列の対応する要素ごとに演算が行われます。
- 本サブルーチンは、メッセージを MPI_REDUCE で 1 つのプロセスに収集した後、それを MPI_SCATTERV で全プロセスに送信したのと機能的に類似しています。
- MPI の基本データ型 (datatype) と MPI の定義済み演算 (op) の組み合わせを以下に示します。演算 (op) をユーザーが自分で定義することもできます。

MPI の基本データ型	MPI の定義済み演算
MPI_INTEGER (注 1) MPI_REAL (注 2) MPI_DOUBLE_PRECISION (注 3) MPI_COMPLEX (注 4)	MPI_SUM（合計） MPI_PROD（積）
MPI_INTEGER (注 1) MPI_REAL (注 2) MPI_DOUBLE_PRECISION (注 3)	MPI_MAX（最大） MPI_MIN（最小）
MPI_2INTEGER MPI_2REAL MPI_2DOUBLE_PRECISION (注 5)	MPI_MAXLOC（最大と位置） MPI_MINLOC（最小と位置）

MPI の基本データ型	MPI の定義済み演算
MPI_LOGICAL	MPI_LAND (論理 AND) MPI_LOR (論理 OR) MPI_LXOR (論理 XOR)
MPI_INTEGER (注 1) MPI_BYTE	MPI_BAND (ビット AND) MPI_BOR (ビット OR) MPI_BXOR (ビット XOR)

下記が使用できるマシン環境もあります。
(注 1) MPI_INTEGER4
(注 2) MPI_REAL4
(注 3) MPI_REAL8
(注 4) MPI_COMPLEX16, MPI_DOUBLE_COMPLEX
(注 5) MPI_2REAL8

使用法

```
CALL MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm, ierror)
```

- sendbuf: 送信バッファーの先頭アドレスを指定します。
- recvbuf: 受信バッファーの先頭アドレスを指定します。
  送信バッファーと受信バッファーの実際に使用する部分は、メモリー上で重なってはいけません (ただし MPI-2 の新機能 MPI_IN_PLACE で回避可能です。詳細は 3-8-1 節を参照)。
- recvcounts: 整数配列。ランク i のプロセスが受信するメッセージの要素数を、この配列の $i+1$ 番目の要素に指定します。comm 内の全プロセスが同じ値を指定する必要があります (サンプルプログラムに示すように、配列を 0 から開始した方が分かりやすくなります)。
- datatype: 整数。送（受）信メッセージのデータ型を指定します。
  comm 内の全プロセスが同じ値を指定する必要があります。
- op: 整数。演算の種類を指定します。
  comm 内の全プロセスが同じ値を指定する必要があります。
- comm: 整数。送受信に参加する全てのプロセスを含むグループのコミュニケータを指定します。
  全プロセスが同じ値を指定する必要があります。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

注意

- コミュニケータ (comm) 内の全プロセスが本サブルーチンをコールする必要があります。

【例 1】

サンプルプログラム

【例 1】のプログラム例です。

```
  PROGRAM MAIN
  INCLUDE 'mpif.h'                          MPIが使用するインクルードファイルを指定します。
  INTEGER ISEND(6),IRECV(3)                 送受信バッファーを指定します。
  INTEGER IRCNT(0:2)
  DATA IRCNT/1,2,3/                         各プロセスの受信メッセージの要素数を指定します。
  CALL MPI_INIT(IERR)                       MPI環境の初期化処理を行います。
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)   プロセス数NPROCSを取得します。
  CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)   自分のランクMYRANKを取得します。
  DO I = 1, 6
    ISEND(I) = I + MYRANK*10                各プロセスは送信バッファーにデータをセットします。
  ENDDO
  CALL MPI_REDUCE_SCATTER(ISEND,IRECV,IRCNT,MPI_INTEGER,
&                         MPI_SUM,MPI_COMM_WORLD,IERR)   通信を行います。
  PRINT *,'MYRANK = ',MYRANK,'IRECV = ',IRECV   通信後の受信バッファーの内容を書き出します。
  CALL MPI_FINALIZE(IERR)                   MPI環境の終了処理を行います。
  END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 3 a.out ↵
  MYRANK = 0 IRECV = 33 0 0
  MYRANK = 1 IRECV = 36 39 0
  MYRANK = 2 IRECV = 42 45 48
```

集団通信サブルーチン　MPI_OP_CREATE

機能 (関連する節：3-3-6-2 節)

MPI_REDUCE(V), MPI_ALLREDUCE(V), MPI_SCAN(V), MPI_REDUCE_SCATTER(V) で使用する演算 (op) をユーザーがサブルーチン function に作成した場合、本サブルーチンはその演算を登録し、演算の ID が op に戻ります。本サブルーチンを実行した後、op を MPI_REDUCE などの引数 op として使用することができます。なお、ユーザーが作成するサブルーチン (function) の指定方法については 3-3-6-2 節を参照して下さい。

使用法

CALL MPI_OP_CREATE(function, commute, op, ierror)

- function: ユーザーが演算を定義したサブルーチン名を指定します。MPI_OP_CREATE をコールしたサブルーチンでは、ここで指定したサブルーチン名を外部関数として EXTERNAL 文で宣言して下さい。
- commute: 論理。ユーザーが定義した演算は結合則を満たす必要がありますが、それに加えて交換則を満たす場合は .TRUE. を、満たさない場合は.FALSE. を指定して下さい。.TRUE. を指定した場合は演算（通信）順序が状況に応じて最適化され、.FALSE. を指定した場合にはプロセスのランクが 0 から昇順に演算（通信）が行われます (3-3-6 節の【例 1】参照)。
- op: 整数。登録された演算の ID が戻ります。この値を保存しておき、MPI_REDUCE などの引数 op で使用します。MPI_OP_CREATE をコールしたサブルーチン以外のサブルーチンで MPI_REDUCE などをコールするときは、この値を COMMON 文などで渡す必要があります。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

【例 1】倍精度複素数の加算の例です (3-3-6-2 節参照)。

サブルーチンfunction

サンプルプログラム

【例1】のプログラム例です。

```
  PROGRAM MAIN
  INCLUDE 'mpif.h'
  EXTERNAL PARA_SUM16                          PARA_SUM16を外部関数で宣言します。
  COMPLEX*16 C(2),CC(2)
  CALL MPI_INIT(IERR)                          MPI環境の初期化処理を行います。
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)     プロセス数NPROCSを取得します。
  CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)     自分のランクMYRANKを取得します。
  CALL MPI_OP_CREATE(PARA_SUM16,.TRUE.,ISUM16,IERR) 演算を登録し、演算のIDがISUM16に戻ります。
  C(1) = (MYRANK*10+1,MYRANK*10+1)             各プロセスは送信バッファーにデータをセットします。
  C(2) = (MYRANK*10+2,MYRANK*10+2)
  CALL MPI_REDUCE(C,CC,2,MPI_DOUBLE_COMPLEX,ISUM16,0,  ISUM16を使用して通信を行います。
&                 MPI_COMM_WORLD,IERR)
  IF (MYRANK == 0) PRINT *,'C =',CC            通信後の受信バッファーの内容を書き出します。
  CALL MPI_FINALIZE(IERR)                      MPI環境の終了処理を行います。
  END
```

```
  SUBROUTINE PARA_SUM16(CIN,CINOUT,LEN,ITYPE) ユーザーが定義した演算(倍精度複素数の加算)です。
  COMPLEX*16 CIN(*),CINOUT(*)
  DO I = 1, LEN
    CINOUT(I) = CIN(I) + CINOUT(I)
  ENDDO
  END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 3 a.out ↵
  C = (33.000･･･,33.000･･･) (36.000･･･,36.000･･･)
```

集団通信サブルーチン　MPI_BARRIER

機能 (関連する節：4-8-3 節、4-4-2 節)

コミュニケータ (comm) 内の全プロセス間で（何らかの理由で）周期をとりたい場合、本サブルーチンを使用します。

図 1 のように『処理 1』→『CALL MPI_BARRIER』→『処理 2』の順に処理を行うプログラムを 3 プロセスで並列に実行する場合を想定します。このとき『処理 1』の部分の進行が、図 2 のようにプロセスによって異なっているとします。

まず、ランク 1 のプロセスが『CALL MPI_BARRIER』に到達しますが、他のプロセスがまだ到達していないため、ランク 1 はウェイトします。次にランク 0 が『CALL MPI_BARRIER』に到達し、同様にウェイトします。そして最後にランク 2 が『CALL MPI_BARRIER』に到達すると、各プロセスはいっせいに実行を再開します。

図 1

図 2

使用法

```
CALL MPI_BARRIER(comm, ierror)
```

- comm: 整数。同期をとりたい全てのプロセスを含むグループのコミュニケータを指定します。
  全プロセスが同じ値を指定する必要があります。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

注意

- コミュニケータ (comm) 内の全プロセスが本サブルーチンをコールする必要があります。

サンプルプログラム

『CALL MPI_BARRIER』を、並列ジョブの経過時間測定の誤差を少なくするために用いる例を以下に示します。

以下の例で、『計算部分』の経過時間を測定するとします。ところが、『入力データの読み込みなど』の部分は、各プロセスで経過時間にばらつきがあるため（特に共有ディスクから読み込みを行う場合）、測定の開始時点 (②) がプロセスによって異なってしまいます。そこで①に『CALL MPI_BARRIER』を挿入し、プロセス間の同期をとります。

『計算部分』の終了後、④で測定を終了します。通常『計算部分』の経過時間はプロセス間でばらつきがありますが、最も長くかかったプロセス経過時間を知りたい場合は③にも『CALL MPI_BARRIER』を挿入してプロセス間の同期をとります。一方、各プロセスの『計算部分』の経過時間のばらつきを比較したい場合は③を指定しません。

```
PROGRAM MAIN
INCLUDE 'mpif.h'                          MPIが使用するインクルードファイルを指定します。
REAL*8 ELP1, ELP2                         経過時間の測定に使用する変数を倍精度で宣言します。
CALL MPI_INIT(IERR)                       MPI環境の初期化処理を行います。

入力データの読み込みなど

CALL MPI_BARRIER(MPI_COMM_WORLD,IERR) ①   プロセス間の同期をとります。
ELP1 = MPI_WTIME()                    ②   MPI_WTIME()を実行して測定を開始します。

計算部分

CALL MPI_BARRIER(MPI_COMM_WORLD,IERR) ③   プロセス間の同期をとります。
ELP2 = MPI_WTIME()                    ④   MPI_WTIME()を実行して測定を終了します。
PRINT *,'ELAPSE = ',ELP2-ELP1             経過時間を秒で表します。
CALL MPI_FINALIZE(IERR)                   MPI環境の終了処理を行います。
END
```

【MPI-2】MPI_IN_PLACE の使い方　MPI_SCATTER

以下では、当 MPI サブルーチンの機能のうち、MPI_IN_PLACE に関連する部分のみを説明します。

この機能は、使用するマシン環境が MPI-2 の MPI_IN_PLACE オプションをサポートしていない場合は、使用することができません (3-8-1 節参照)。

使用法

CALL MPI_SCATTER(sendbuf, sendcount, sendtype,
recvbuf, recvcount, recvtype, root, comm, ierror)

- 送信元プロセス (root) では、recvbuf に MPI_IN_PLACE を指定します。root プロセスでは recvcount と recvtype は無視されますので、適当な変数を指定してもかまいません。
- 下図の①の部分 (MPI_IN_PLACE を使用しない場合に root プロセスの送信データを入れる部分) は送信されません。

サンプルプログラム

上図のプログラム例です。

```
    :
  INTEGER ISEND(3)
    :
  IF (MYRANK==0) THEN                                    rootプロセス
    CALL MPI_SCATTER(ISEND       ,1     ,MPI_INTEGER,
&                  MPI_IN_PLACE,IDUMMY,IDUMMY     ,
&                  0,MPI_COMM_WORLD,IERR)
  ELSE                                                   その他のプロセス
    CALL MPI_SCATTER(IDUMMY,IDUMMY,IDUMMY     ,
&                  IRECV ,1     ,MPI_INTEGER,
&                  0,MPI_COMM_WORLD,IERR)
  ENDIF
    :
```

【MPI-2】MPI_IN_PLACE の使い方　MPI_SCATTERV

以下では、当 MPI サブルーチンの機能のうち、MPI_IN_PLACE に関連する部分のみを説明します。

この機能は、使用するマシン環境が MPI-2 の MPI_IN_PLACE オプションをサポートしていない場合は、使用することができません (3-8-1 節参照)。

使用法

CALL MPI_SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm, ierror)

- 送信元プロセス (root) では、recvbuf に MPI_IN_PLACE を指定します。root プロセスでは recvcount と recvtype は無視されますので、適当な変数を指定してもかまいません。
- 下図の①の部分 (MPI_IN_PLACE を使用しない場合に root プロセスの送信データを入れる部分) は送信されません。

サンプルプログラム

上図のプログラム例です。

```
    :
  INTEGER ISEND(6),IRECV(3),ISCNT(0:2),IDISP(0:2)
  DATA ISCNT/1,2,3/ IDISP/0,1,3/
    :
  IF (MYRANK==0) THEN                                    rootプロセス
    CALL MPI_SCATTERV(ISEND       ,ISCNT,IDISP,MPI_INTEGER,
&                   MPI_IN_PLACE,IDUMMY     ,IDUMMY     ,
&                   0,MPI_COMM_WORLD,IERR)
  ELSE                                                   その他のプロセス
    CALL MPI_SCATTERV(IDUMMY,IDUMMY,IDUMMY,IDUMMY     ,
&                   IRECV ,MYRANK+1     ,MPI_INTEGER,
&                   0,MPI_COMM_WORLD,IERR)
  ENDIF
    :
```

【MPI-2】MPI_IN_PLACE の使い方　MPI_GATHER

以下では、当 MPI サブルーチンの機能のうち、MPI_IN_PLACE に関連する部分のみを説明します。

この機能は、使用するマシン環境が MPI-2 の MPI_IN_PLACE オプションをサポートしていない場合は、使用することができません (3-8-1 節参照)。

使用法

CALL MPI_GATHER(sendbuf, sendcount, sendtype,
recvbuf, recvcount, recvtype, root, comm, ierror)

- 宛先プロセス (root) では、sendbuf に MPI_IN_PLACE を指定します。root プロセスでは sendcount と sendtype は無視されますので、適当な変数を指定してもかまいません。
- 下図の①の部分 (MPI_IN_PLACE を使用しない場合に root プロセスの受信データが入る部分) には受信されません。

サンプルプログラム

上図のプログラム例です。

```
    :
  INTEGER IRECV(3)
    :
  IF (MYRANK==0) THEN                                    rootプロセス
    CALL MPI_GATHER(MPI_IN_PLACE,IDUMMY,IDUMMY     ,
&                 IRECV       ,1      ,MPI_INTEGER,
&                 0,MPI_COMM_WORLD,IERR)
  ELSE                                                   その他のプロセス
    CALL MPI_GATHER(ISEND ,1      ,MPI_INTEGER,
&                 IDUMMY,IDUMMY,IDUMMY     ,
&                 0,MPI_COMM_WORLD,IERR)
  ENDIF
    :
```

【MPI-2】MPI_IN_PLACE の使い方　MPI_GATHERV

以下では、当 MPI サブルーチンの機能のうち、MPI_IN_PLACE に関連する部分のみを説明します。

この機能は、使用するマシン環境が MPI-2 の MPI_IN_PLACE オプションをサポートしていない場合は、使用することができません (3-8-1 節参照)。(関連する節：4-6-3-1 節)

使用法

CALL MPI_GATHERV(sendbuf, sendcount, sendtype,
recvbuf, recvcounts, displs, recvtype, root, comm, ierror)

- 宛先プロセス (root) では、sendbuf に MPI_IN_PLACE を指定します。root プロセスでは sendcount と sendtype は無視されますので、適当な変数を指定してもかまいません。
- 下図の①の部分 (MPI_IN_PLACE を使用しない場合に root プロセスの受信データが入る部分) には受信されません。

サンプルプログラム

上図のプログラム例です。

```
    :
  INTEGER ISEND(3),IRECV(6),IRCNT(0:2),IDISP(0:2)
  DATA IRCNT/1,2,3/ IDISP/0,1,3/
    :
  IF (MYRANK==0) THEN                                    rootプロセス
    CALL MPI_GATHERV(MPI_IN_PLACE,IDUMMY     ,IDUMMY     ,
&                  IRECV       ,IRCNT,IDISP,MPI_INTEGER,
&                  0,MPI_COMM_WORLD,IERR)
  ELSE                                                   その他のプロセス
    CALL MPI_GATHERV(ISEND ,MYRANK+1     ,MPI_INTEGER,
&                  IDUMMY,IDUMMY,IDUMMY,IDUMMY     ,
&                  0,MPI_COMM_WORLD,IERR)
  ENDIF
    :
```

【MPI-2】MPI_IN_PLACE の使い方　MPI_ALLGATHER

以下では、当 MPI サブルーチンの機能のうち、MPI_IN_PLACE に関連する部分のみを説明します。

この機能は、使用するマシン環境が MPI-2 の MPI_IN_PLACE オプションをサポートしていない場合は、使用することができません (3-8-1 節参照)。

使用法

```
CALL MPI_ALLGATHER(sendbuf, sendcount, sendtype,
recvbuf, recvcount, recvtype, comm, ierror)
```

- 全プロセスで、sendbuf に MPI_IN_PLACE を指定します。sendcount と sendtype は無視されますので、適当な変数を指定してもかまいません。
- 各プロセスは下図の①に送信データを設定します。通信終了後、下図の → のように変化します。

サンプルプログラム

上図のプログラム例です。

```
    :
  INTEGER IRECV(3)
    :
  CALL MPI_ALLGATHER(MPI_IN_PLACE,IDUMMY,IDUMMY     ,
&                    IRECV       ,1     ,MPI_INTEGER,
&                    MPI_COMM_WORLD,IERR)
    :
```

【MPI-2】MPI_IN_PLACE の使い方　MPI_ALLGATHERV

以下では、当 MPI サブルーチンの機能のうち、MPI_IN_PLACE に関連する部分のみを説明します。

この機能は、使用するマシン環境が MPI-2 の MPI_IN_PLACE オプションをサポートしていない場合は、使用することができません (3-8-1 節参照)。(関連する節：4-6-3-2 節)

使用法

CALL MPI_ALLGATHERV(sendbuf, sendcount, sendtype,
recvbuf, recvcounts, displs, recvtype, comm, ierror)

- 全プロセスで、sendbuf に MPI_IN_PLACE を指定します。sendcount と sendtype は無視されますので、適当な変数を指定してもかまいません。
- 各プロセスは下図の①に送信データを設定します。通信終了後、下図の → のように変化します。

サンプルプログラム

上図のプログラム例です。

```
    :
  INTEGER IRECV(6),IRCNT(0:2),IDISP(0:2)
  DATA IRCNT/1,2,3/ IDISP/0,1,3/
    :
  CALL MPI_ALLGATHERV(MPI_IN_PLACE,IDUMMY     ,IDUMMY     ,
&                     IRECV      ,IRCNT,IDISP,MPI_INTEGER,
&                     MPI_COMM_WORLD,IERR)
    :
```

【MPI-2】MPI_IN_PLACE の使い方　MPI_REDUCE

以下では、当 MPI サブルーチンの機能のうち、MPI_IN_PLACE に関連する部分のみを説明します。

この機能は、使用するマシン環境が MPI-2 の MPI_IN_PLACE オプションをサポートしていない場合は、使用することができません (3-8-1 節参照)。

使用法

```
CALL MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
```

- 宛先プロセス (root) では、sendbuf に MPI_IN_PLACE を指定します。
- 各プロセスは下図の①に送信データを設定します (ランク 0 の①の位置に注意して下さい)。通信終了後、下図の → のように変化します。

サンプルプログラム

上図のプログラム例です。

```
    :
  IF (MYRANK==0) THEN
    CALL MPI_REDUCE(MPI_IN_PLACE,IRECV,1,MPI_INTEGER,        rootプロセス
&                   MPI_SUM,0,MPI_COMM_WORLD,IERR)
  ELSE
    CALL MPI_REDUCE(ISEND,IDUMMY,1,MPI_INTEGER,              その他のプロセス
&                   MPI_SUM,0,MPI_COMM_WORLD,IERR)
  ENDIF
    :
```

【MPI-2】MPI_IN_PLACE の使い方　MPI_ALLREDUCE

以下では、当 MPI サブルーチンの機能のうち、MPI_IN_PLACE に関連する部分のみを説明します。

この機能は、使用するマシン環境が MPI-2 の MPI_IN_PLACE オプションをサポートしていない場合は、使用することができません (3-8-1 節参照)。

使用法

```
CALL MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm, ierror)
```

- 全プロセスで、sendbuf に MPI_IN_PLACE を指定します。
- 各プロセスは下図の①に送信データを設定します。通信終了後、下図の → のように変化します。

サンプルプログラム

上図のプログラム例です。

```
    :
  CALL MPI_ALLREDUCE(MPI_IN_PLACE,IRECV,1,MPI_INTEGER,
&                    MPI_SUM,MPI_COMM_WORLD,IERR)
    :
```

【MPI-2】MPI_IN_PLACE の使い方　MPI_SCAN

以下では、当 MPI サブルーチンの機能のうち、MPI_IN_PLACE に関連する部分のみを説明します。

この機能は、使用するマシン環境が MPI-2 の MPI_IN_PLACE オプションをサポートしていない場合は、使用することができません (3-8-1 節参照)。

使用法

```
CALL MPI_SCAN(sendbuf, recvbuf, count, datatype, op, comm, ierror)
```

- 全プロセスで、sendbuf に MPI_IN_PLACE を指定します。
- 各プロセスは下図の①に送信データを設定します。通信終了後、下図の → のように変化します。

サンプルプログラム

上図のプログラム例です。

```
      :
  CALL MPI_SCAN(MPI_IN_PLACE,IRECV,1,MPI_INTEGER,
&               MPI_SUM,MPI_COMM_WORLD,IERR)
      :
```

【MPI-2】MPI_IN_PLACE の使い方　MPI_REDUCE_SCATTER

以下では、当 MPI サブルーチンの機能のうち、MPI_IN_PLACE に関連する部分のみを説明します。

この機能は、使用するマシン環境が MPI-2 の MPI_IN_PLACE オプションをサポートしていない場合は、使用することができません (3-8-1 節参照)。

使用法

```
CALL MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm, ierror)
```

- 全プロセスで、sendbuf に MPI_IN_PLACE を指定します。
- 各プロセスは下図のように送信データを設定します。通信終了後、送信されたデータは (全プロセスで) 受信バッファー recvbuf の先頭から入り、図の → のように変化します。

サンプルプログラム

上図のプログラム例です。

```
      :
  INTEGER IRECV(6),IRCNT(0:2)
  DATA IRCNT/1,2,3/
      :
  CALL MPI_REDUCE_SCATTER(MPI_IN_PLACE,IRECV,IRCNT,MPI_INTEGER,
&                         MPI_SUM,MPI_COMM_WORLD,IERR)
      :
```

1対1ブロッキング通信サブルーチン　MPI_SEND

機能 (関連する節：3-4 節)

送信バッファー (buf) 内の、データ型が datatype である count 個の連続した要素から構成した送信メッセージを、タグ (tag) をつけて宛先プロセス (コミュニケータ comm 内でランクが dest) に送信します。MPI_SEND で送信したメッセージは、MPI_RECV と MPI_IRECV のどちらで受信してもかまいません。

MPI_SEND をコールした後、アプリケーションプログラムは、送信バッファーが再利用可能になるまで待ちの状態に入ります。

動作

- アプリケーションプログラムは、②で送信バッファー buf にデータ（メッセージ）をセットし、③で MPI_SEND をコールした後、④で待ちの状態に入ります。
- サブルーチン MPI_SEND は、1でメッセージを送信バッファー buf からシステムバッファーにコピーします。コピーが完了した後、2でアプリケーションプログラムの待ちを解除します。なお、待ちが解除されたということは、送信したメッセージが宛先プロセスに到着したことを意味するわけではありません。
- 3の時点で、宛先プロセスが MPI_RECV または MPI_IRECV をすでにコールしている場合は、システムバッファーのメッセージを宛先プロセスに送信します。宛先プロセスがまだコールしていない場合は、コールするまで3で待ちます。
- アプリケーションプログラムは、⑤で送信バッファー buf の再利用が可能になり、⑥で処理を再開します。
- ②で再び送信バッファー buf にデータ（メッセージ）をセットしても、前回の②で送信バッファー buf にセットしたメッセージはすでに2でシステムバッファーにコピーされているので、壊れることはありません。

使用法

CALL MPI_SEND(buf, count, datatype, dest, tag, comm, ierror)

- buf: 送信バッファーの先頭アドレスを指定します。
- count: 整数。送信バッファー内の送信メッセージの要素数を指定します。
- datatype: 整数。送信バッファー内の送信メッセージのデータ型を指定します。
- dest: 整数。宛先プロセスの comm 内でのランクを指定します。宛先プロセスに対してメッセージを送信しない場合には、MPI_PROC_NULL を指定します。
- tag: 整数。送信するメッセージの種類を区別したい場合に使用します。
  特に区別する必要がない場合には、適当な値 (例えば 1) を指定します。
  0〜MPI_TAG_UB までの整数を指定することができます。MPI_TAG_UB の値は mpif.h の中で設定されており、マシン環境によって異なります (PRINT 文で書き出せば値が分かります)。
- comm: 整数。自分のプロセスと宛先プロセスを含むグループのコミュニケータを指定します。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

サンプルプログラム

【例 1】のプログラム例です。ランク 0 からランク 1 にメッセージを送信します。

```
      PROGRAM MAIN
      INCLUDE 'mpif.h'                          MPIが使用するインクルードファイルを指定します。
      INTEGER ISTATUS(MPI_STATUS_SIZE)          MPI_RECVで使用する状況オブジェクトを指定します。
      CALL MPI_INIT(IERR)                       MPI環境の初期化処理を行います。
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)  プロセス数NPROCSを取得します。
      CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)  自分のランクMYRANKを取得します。
      IF (MYRANK == 0) ISBUF = 9                ランク0のプロセスは送信バッファーにデータを
                                                セットします。
      IF (MYRANK == 0) THEN
        CALL MPI_SEND(ISBUF,1,MPI_INTEGER,      ランク0はランク1にメッセージを送信します。
     &                1,1,MPI_COMM_WORLD,IERR)
      ELSEIF (MYRANK == 1) THEN
        CALL MPI_RECV(IRBUF,1,MPI_INTEGER,      ランク1はランク0からメッセージを受信します。
     &                0,1,MPI_COMM_WORLD,ISTATUS,IERR)
      ENDIF
      IF (MYRANK == 1) PRINT *,'IRBUF = ',IRBUF   通信後の受信バッファーの内容を書き出します。
      CALL MPI_FINALIZE(IERR)                   MPI環境の終了処理を行います。
      END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 2 a.out ↵
  IRBUF = 9
```

1対1ブロッキング通信サブルーチン　MPI_RECV

機能 (関連する節：3-4 節)

送信元プロセス (コミュニケータ comm 内でランクが source) から送信された、タグが tag のメッセージを、データ型が datatype である count 個の連続した要素から構成される受信バッファー (buf) に受信します。MPI_SEND と MPI_ISEND のどちらで送信したメッセージも、MPI_RECV で受信することができます。

MPI_RECV をコールした後、アプリケーションプログラムは、受信バッファーへのメッセージの受信が完了するまで待ちの状態に入ります。

動作

- アプリケーションプログラムは、②で MPI_RECV をコールした後、③で待ちの状態に入ります。
- サブルーチン MPI_RECV は、1の時点で送信元プロセスがすでに MPI_SEND または MPI_ISEND をコールしている場合は、送信元プロセスから送られたメッセージをシステムバッファーに受信します。送信元プロセスがまだコールしていない場合は、コールするまで1で待ちます。
- 2でメッセージをシステムバッファーから受信バッファー buf にコピーします。コピーが完了した後、3でアプリケーションプログラムの待ちを解除します。
- アプリケーションプログラムは、④で受信バッファー buf を参照可能になり、⑤で処理を再開します。
- ⑥で受信バッファー buf を参照したときには、受信メッセージが全て入っています。

使用法

CALL MPI_RECV(buf, count, datatype, source, tag, comm, status, ierror)

- buf: 受信バッファーの先頭アドレスを指定します。
- count: 整数。受信バッファーの要素数を指定します。受信メッセージの要素数と同じか、それ以上の大きさにして下さい。受信メッセージの要素数よりも少ない場合は、エラーになります（マシン環境によってはエラーにならない場合もあります）。
- datatype: 整数。受信バッファーに入る受信メッセージのデータ型を指定します。
- source: 整数。受信したいメッセージを送信するプロセスのcomm内でのランクを指定します。任意のプロセスからのメッセージを受信したい場合は、ワイルドカードMPI_ANY_SOURCEを指定します。送信元プロセスからのメッセージを受信しない場合には、MPI_PROC_NULLを指定します。
- tag: 整数。受信したいメッセージに付けられているタグの値を指定します。この値は送信元プロセスがそのメッセージを送信した際に引数tagとして付けた値です。
  任意のタグ値のメッセージを受信したい場合は、ワイルドカードMPI_ANY_TAGを指定します。
- comm: 整数。自分のプロセスと送信元プロセスを含むグループのコミュニケータを指定します。
- status: 状況オブジェクトの配列を指定します。サンプルプログラムに示すように、大きさがMPI_STATUS_SIZEの整数配列を、『INCLUDE 'mpif.h'』より後に指定して下さい。本サブルーチンが完了すると、受信したメッセージの送信元のランクがstatus(MPI_SOURCE)に、タグがstatus(MPI_TAG)に戻ります。これらの値は通常使用しません。使用するのは、ワイルドカードMPI_ANY_SOURCEやMPI_ANY_TAGを指定した際、受信したメッセージの送信元のランクやタグの値を知りたい場合です(4-7-1-3節参照)。その場合、受信したメッセージの要素数も知る必要があるならば、『CALL MPI_GET_COUNT』を使用して下さい。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

サンプルプログラム

【例1】のプログラム例です。ランク1はランク0からメッセージを受信します。

```
      PROGRAM MAIN
      INCLUDE 'mpif.h'                          MPIが使用するインクルードファイルを指定します。
      INTEGER ISTATUS(MPI_STATUS_SIZE)          MPI_RECVで使用する状況オブジェクトを指定します。
      CALL MPI_INIT(IERR)                       MPI環境の初期化処理を行います。
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)  プロセス数NPROCSを取得します。
      CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)  自分のランクMYRANKを取得します。
      IF (MYRANK == 0) ISBUF = 9                ランク0のプロセスは送信バッファーにデータを
                                                セットします。
      IF (MYRANK == 0) THEN
        CALL MPI_SEND(ISBUF,1,MPI_INTEGER,      ランク0はランク1にメッセージを送信します。
     &                1,1,MPI_COMM_WORLD,IERR)
      ELSEIF (MYRANK == 1) THEN
        CALL MPI_RECV(IRBUF,1,MPI_INTEGER,      ランク1はランク0からメッセージを受信します。
     &                0,1,MPI_COMM_WORLD,ISTATUS,IERR)
      ENDIF
      IF (MYRANK == 1) PRINT *,'IRBUF = ',IRBUF   通信後の受信バッファーの内容を書き出します。
      CALL MPI_FINALIZE(IERR)                   MPI環境の終了処理を行います。
      END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 2 a.out ↵
  IRBUF = 9
```

1対1ブロッキング通信サブルーチン　MPI_SENDRECV

機能 (関連する節：3-4 節，4-6-2 節)

MPI_SENDRECV は、MPI_SEND と MPI_RECV の両機能を 1 つのサブルーチンで実行します。【例 1】のように宛先 (dest) と送信元 (source) を別にすることも、【例 2】のように同一にすることも可能です。

- MPI_SENDRECV はブロッキング通信サブルーチンですが、デッドロックにはなりません。
- 送信バッファー (sendbuf) 内の、データ型が sendtype である sendcount 個の連続した要素から構成した送信メッセージを、タグ (sendtag) をつけて宛先プロセス (コミュニケータ comm 内でランクが dest) に送信します。
- 送信元プロセス (コミュニケータ comm 内でランクが source) から送信された、タグが recvtag のメッセージを、データ型が recvtype である recvcount 個の連続した要素から構成される受信バッファー (recvbuf) に受信します。
- MPI_SENDRECV をコールした後、アプリケーションプログラムは、送信バッファーが再利用可能になり、かつ受信バッファーへのメッセージの受信が完了するまで待ちの状態に入ります。

サンプルプログラム

【例 1】のプログラム例です。

```
   PROGRAM MAIN
   INCLUDE 'mpif.h'                          MPIが使用するインクルードファイルを指定します。
   INTEGER ISTATUS(MPI_STATUS_SIZE)          状況オブジェクトを指定します。
   CALL MPI_INIT(IERR)                       MPIの環境の初期化処理を行います。
   CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)  プロセス数NPROCSを取得します。
   CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)  自分のランクMYRANKを取得します。
   IUP   = MOD(MYRANK+1       ,NPROCS)       宛先のプロセスのランクを指定します。
   IDOWN = MOD(MYRANK-1+NPROCS,NPROCS)       送信元のプロセスのランクを指定します。
   ISBUF = MYRANK + 10                       各プロセスは送信バッファーにデータをセットします。
   CALL MPI_SENDRECV(ISBUF,1,MPI_INTEGER,IUP  ,1,  上記の【例1】に示す通信を行います。
 &                  IRBUF,1,MPI_INTEGER,IDOWN,1,
 &                  MPI_COMM_WORLD,ISTATUS,IERR)
   PRINT *,'MYRANK = ',MYRANK,' IRBUF = ',IRBUF    通信後の受信バッファーの内容を書き出します。
   CALL MPI_FINALIZE(IERR)                   MPI環境の終了処理を行います。
   END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 3 a.out ↵
  MYRANK = 0  IRBUF = 12
  MYRANK = 1  IRBUF = 10
  MYRANK = 2  IRBUF = 11
```

使用法

CALL MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype source, recvtag, comm, status, ierror)

- sendbuf: 送信バッファーの先頭アドレスを指定します。
- sendcount: 整数。送信バッファー内の送信メッセージの要素数を指定します。
- sendtype: 整数。送信バッファー内の送信メッセージのデータ型を指定します。
- dest: 整数。宛先プロセスの comm 内でのランクを指定します。宛先プロセスに対してメッセージを送信しない場合には、MPI_PROC_NULL を指定します。
- sendtag: 整数。送信するメッセージの種類を区別したい場合に使用します。
  特に区別する必要がない場合には、適当な値 (例えば 1) を指定します。
  0〜MPI_TAG_UB までの整数を指定することができます。MPI_TAG_UB の値は mpif.h の中で設定されており、マシン環境によって異なります (PRINT 文で書き出せば値が分かります)。
- recvbuf: 受信バッファーの先頭アドレスを指定します。送信バッファーと重複してはいけません。
- recvcount: 整数。受信バッファーの要素数を指定します。受信メッセージの要素数と同じか、それ以上の大きさにして下さい。受信メッセージの要素数よりも少ない場合は、エラーになります（マシン環境によってはエラーにならない場合もあります)。
- recvtype: 整数。受信バッファーに入る受信メッセージのデータ型を指定します。
- source: 整数。受信したいメッセージを送信するプロセスの comm 内でのランクを指定します。任意のプロセスからのメッセージを受信したい場合は、ワイルドカード MPI_ANY_SOURCE を指定します。送信元プロセスからのメッセージを受信しない場合には、MPI_PROC_NULL を指定します。
- recvtag: 整数。受信したいメッセージに付けられているタグの値を指定します。この値は送信元プロセスがそのメッセージを送信した際に引数 tag として付けた値です。
  任意のタグ値のメッセージを受信したい場合は、ワイルドカード MPI_ANY_TAG を指定します。
- comm: 整数。自分のプロセス、宛先プロセス、送信元プロセスを含むグループのコミュニケータを指定します。
- status: 状況オブジェクトの配列を指定します。サンプルプログラムに示すように、大きさが MPI_STATUS_SIZE の整数配列を、『INCLUDE 'mpif.h'』より後に指定して下さい。本サブルーチンが完了すると、受信したメッセージの送信元のランクが status(MPI_SOURCE) に、タグが status(MPI_TAG) に戻ります。これらの値は通常は使用しません。使用するのは、ワイルドカード MPI_ANY_SOURCE や MPI_ANY_TAG を指定した際、受信したメッセージの送信元のランクやタグの値を知りたい場合です (4-7-1-3 節参照)。その場合、受信したメッセージの要素数も知る必要があるならば、『CALL MPI_GET_COUNT』を使用して下さい。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

注意

- sendcount, sendtype と recvcount, recvtype は異なっていてもかまいません。

1対1非ブロッキング通信サブルーチン　MPI_ISEND

機能 (関連する節：3-4 節)

送信バッファー (buf) 内の、データ型が datatype である count 個の連続した要素から構成した送信メッセージを、タグ (tag) をつけて宛先プロセス (コミュニケータ comm 内でランクが dest) に送信します。MPI_ISEND で送信したメッセージは、MPI_RECV と MPI_IRECV のどちらで受信してもかまいません。

MPI_ISEND は MPI_WAIT とペアで使用します。MPI_ISEND をコールした後、アプリケーションプログラムは、送信バッファーが再利用可能になるのを待たず、ただちに次の処理を続行しますので、次に MPI_WAIT をコールするまでは送信バッファーの内容を更新しないで下さい。

MPI_ISEND を指定して MPI_WAIT を指定し忘れた場合、タイミングによる（再現性のない）エラーが発生します。また MPI_ISEND と MPI_WAIT の引数 request がスペルミスで異なっている場合も、当然ながら MPI_WAIT は働かず、タイミングによる（再現性のない）エラーが発生しますので、スペルミスには十分注意して下さい。

動作

- アプリケーションプログラムは、②で送信バッファー buf にデータ（メッセージ）をセットし、③で MPI_ISEND をコールした後、ただちに④で処理を続行します。③の送信メッセージに対し、識別子 (ID) が MPI_ISEND によって付けられ、引数 request に戻ります。
- アプリケーションプログラムは、②で再び送信バッファー buf を再利用するよりも前に、⑤で引数 request を指定して MPI_WAIT をコールし、その後⑥で待ちの状態に入ります。
- サブルーチン MPI_ISEND は、1でメッセージを送信バッファー buf からシステムバッファーにコピーします。コピーが完了した後、⑤が実行されている場合には2でアプリケーションプログラムの待ちを解除します。なお、待ちが解除されたということは、送信したメッセージが宛先プロセスに到着したことを意味するわけではありません。
- 3の時点で、宛先プロセスが MPI_RECV または MPI_IRECV をすでにコールしている場合は、システムバッファーのメッセージを宛先プロセスに送信します。宛先プロセスがまだコールしていない場合は、コールするまで3で待ちます。
- アプリケーションプログラムは、⑦で送信バッファー buf の再利用が可能になり、⑧で処理を再開します。
- ②で再び送信バッファー buf にデータ（メッセージ）をセットしても、前回の②で送信バッファー buf にセットしたメッセージはすでに2でシステムバッファーにコピーされているので、壊れることはありません。

使用法

CALL MPI_ISEND(buf, count, datatype, dest, tag, comm, request, ierror)

- buf: 送信バッファーの先頭アドレスを指定します。
- count: 整数。送信バッファー内の送信メッセージの要素数を指定します。
- datatype: 整数。送信バッファー内の送信メッセージのデータ型を指定します。
- dest: 整数。宛先プロセスの comm 内でのランクを指定します。宛先プロセスに対してメッセージを送信しない場合には、MPI_PROC_NULL を指定します。
- tag: 整数。送信するメッセージの種類を区別したい場合に使用します。
  特に区別する必要がない場合には、適当な値 (例えば 1) を指定します。
  0〜MPI_TAG_UB までの整数を指定することができます。MPI_TAG_UB の値は mpif.h の中で設定されており、マシン環境によって異なります (PRINT 文で書き出せば値が分かります)。
- comm: 整数。自分のプロセスと宛先プロセスを含むグループのコミュニケータを指定します。
- request: 整数。通信リクエスト（送信を要求したメッセージに付けられた識別子）が戻ります。
  後に MPI_WAIT の引数でこの値を指定します。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

サンプルプログラム

【例 1】のプログラム例です。ランク 0 からランク 1 にメッセージを送信します。

```
      PROGRAM MAIN
      INCLUDE 'mpif.h'                          MPIが使用するインクルードファイルを指定します。
      INTEGER ISTATUS(MPI_STATUS_SIZE)          MPI_WAITで使用する状況オブジェクトを指定します。
      CALL MPI_INIT(IERR)                       MPI環境の初期化処理を行います。
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)  プロセス数NPROCSを取得します。
      CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)  自分のランクMYRANKを取得します。
      IF (MYRANK == 0) ISBUF = 9                ランク0のプロセスは送信バッファーにデータを
                                                セットします。
      IF (MYRANK == 0) THEN
        CALL MPI_ISEND(ISBUF,1,MPI_INTEGER,     ランク0はランク1にメッセージを送信します。
     &                 1,1,MPI_COMM_WORLD,IREQ,IERR)
      ELSEIF (MYRANK == 1) THEN
        CALL MPI_IRECV(IRBUF,1,MPI_INTEGER,     ランク1はランク0からメッセージを受信します。
     &                 0,1,MPI_COMM_WORLD,IREQ,IERR)
      ENDIF
      CALL MPI_WAIT(IREQ,ISTATUS,IERR)          送信が完了するまで待ちます(受信も同様です)。
      IF (MYRANK == 1) PRINT *,'IRBUF = ',IRBUF   通信後の受信バッファーの内容を書き出します。
      CALL MPI_FINALIZE(IERR)                   MPI環境の終了処理を行います。
      END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 2 a.out ↵
  IRBUF = 9
```

1対1非ブロッキング通信サブルーチン　MPI_IRECV

機能 (関連する節：3-4 節)

送信元プロセス (コミュニケータ comm 内でランクが source) から送信された、タグが tag のメッセージを、データ型が datatype である count 個の連続した要素から構成される受信バッファー (buf) に受信します。MPI_SEND と MPI_ISEND のどちらで送信したメッセージも、MPI_IRECV で受信することができます。

MPI_IRECV は MPI_WAIT とペアで使用します。MPI_IRECV をコールした後、アプリケーションプログラムは、受信バッファーへのメッセージの受信が完了するのを待たず、ただちに処理を続行しますので、次に MPI_WAIT をコールするまでは受信バッファーの内容を参照しないで下さい。

MPI_IRECV を指定して MPI_WAIT を指定し忘れた場合、タイミングによる（再現性のない）エラーが発生します。また MPI_IRECV と MPI_WAIT の引数 request がスペルミスで異なっている場合も、当然ながら MPI_WAIT は働かず、タイミングによる（再現性のない）エラーが発生しますので、スペルミスには十分注意して下さい。

動作

- アプリケーションプログラムは、②で MPI_IRECV をコールした後、ただちに③で処理を続行します。②の受信メッセージに対し、識別子 (ID) が MPI_IRECV によって付けられ、引数 request に戻ります。
- アプリケーションプログラムは、⑧で受信バッファー buf を参照するよりも前に、④で引数 request を指定して MPI_WAIT をコールし、その後⑤で待ちの状態に入ります。
- サブルーチン MPI_IRECV は、1の時点で送信元プロセスがすでに MPI_SEND または MPI_ISEND をコールしている場合は、送信元プロセスから送られたメッセージをシステムバッファーに受信します。送信元プロセスがまだコールしていない場合は、コールするまで1で待ちます。
- 2でメッセージをシステムバッファーから受信バッファー buf にコピーします。コピーが完了した後、④が実行されている場合には3でアプリケーションプログラムの待ちを解除します。
- アプリケーションプログラムは、⑥で受信バッファー buf を参照可能になり、⑦で処理を再開します。
- ⑧で受信バッファー buf を参照したときには、受信メッセージが全て入っています。

使用法

CALL MPI_IRECV(buf, count, datatype, source, tag, comm, request, ierror)

- buf: 受信バッファーの先頭アドレスを指定します。
- count: 整数。受信バッファーの要素数を指定します。受信メッセージの要素数と同じか、それ以上の大きさにして下さい。受信メッセージの要素数よりも少ない場合は、エラーになります（マシン環境によってはエラーにならない場合もあります）。
- datatype: 整数。受信バッファーに入る受信メッセージのデータ型を指定します。
- source: 整数。受信したいメッセージを送信するプロセスの comm 内でのランクを指定します。任意のプロセスからのメッセージを受信したい場合は、ワイルドカード MPI_ANY_SOURCE を指定します。送信元プロセスからのメッセージを受信しない場合には、MPI_PROC_NULL を指定します。
- tag: 整数。受信したいメッセージに付けられているタグの値を指定します。この値は送信元プロセスがそのメッセージを送信した際に引数 tag として付けた値です。
  任意のタグ値のメッセージを受信したい場合は、ワイルドカード MPI_ANY_TAG を指定します。
- comm: 整数。自分のプロセスと送信元プロセスを含むグループのコミュニケータを指定します。
- request: 整数。通信リクエスト（受信を要求したメッセージに付けられた識別子）が戻ります。
  後に MPI_WAIT の引数でこの値を指定します。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

サンプルプログラム

【例1】のプログラム例です。ランク1はランク0からメッセージを受信します。

```
      PROGRAM MAIN
      INCLUDE 'mpif.h'                          MPIが使用するインクルードファイルを指定します。
      INTEGER ISTATUS(MPI_STATUS_SIZE)          MPI_WAITで使用する状況オブジェクトを指定します。
      CALL MPI_INIT(IERR)                       MPI環境の初期化処理を行います。
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)  プロセス数NPROCSを取得します。
      CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)  自分のランクMYRANKを取得します。
      IF (MYRANK == 0) ISBUF = 9                ランク0のプロセスは送信バッファーにデータを
                                                セットします。
      IF (MYRANK == 0) THEN
        CALL MPI_ISEND(ISBUF,1,MPI_INTEGER,     ランク0はランク1にメッセージを送信します。
     &                 1,1,MPI_COMM_WORLD,IREQ,IERR)
      ELSEIF (MYRANK == 1) THEN
        CALL MPI_IRECV(IRBUF,1,MPI_INTEGER,     ランク1はランク0からメッセージを受信します。
     &                 0,1,MPI_COMM_WORLD,IREQ,IERR)
      ENDIF
      CALL MPI_WAIT(IREQ,ISTATUS,IERR)          受信が完了するまで待ちます(送信も同様です)。
      IF (MYRANK == 1) PRINT *,'IRBUF = ',IRBUF   通信後の受信バッファーの内容を書き出します。
      CALL MPI_FINALIZE(IERR)                   MPI環境の終了処理を行います。
      END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 2 a.out ↵
  IRBUF = 9
```

1 対 1 非ブロッキング通信サブルーチン　MPI_WAIT

機能 (関連する節：3-4 節)

MPI_WAIT は、1 対 1 非ブロッキング通信サブルーチン (MPI_ISEND または MPI_IRECV) を使用した場合に用いますので、MPI_ISEND または MPI_IRECV の説明も併せて参照して下さい。

MPI_ISEND または MPI_IRECV を指定して MPI_WAIT を指定し忘れた場合、タイミングによる（再現性のない）エラーが発生します。また MPI_ISEND または MPI_IRECV と MPI_WAIT の引数 request がスペルミスで異なっている場合も、当然ながら MPI_WAIT は働かず、タイミングによる（再現性のない）エラーが発生しますので、スペルミスには十分注意して下さい。

使用法

CALL MPI_WAIT(request, status, ierror)

- request: 整数。通信リクエストを指定します。これは MPI_ISEND または MPI_IRECV をコールしたときにメッセージに付けられた識別子 request です。
  この引数の内容は、MPI_WAIT が完了すると MPI_REQUEST_NULL という値に変わります。
- status: 状況オブジェクトの配列を指定します。サンプルプログラムに示すように、大きさが MPI_STATUS_SIZE の整数配列を、『INCLUDE 'mpif.h'』より後に指定して下さい。
  MPI_IRECV に対して MPI_WAIT をコールした場合、MPI_WAIT が完了すると、受信したメッセージの送信元のランクの値が status(MPI_SOURCE) に、タグの値が status(MPI_TAG) に戻ります（サンプルプログラム参照)。
  これらの値は通常のプログラムでは使用しません。使用するのは、サンプルプログラムのように、MPI_IRECV の引数にワイルドカード MPI_ANY_SOURCE や MPI_ANY_TAG を指定し、受信した後にメッセージの送信元のランクやタグの値を知りたい場合です (4-7-1-3 節参照)。
  その場合、受信したメッセージの要素数も知る必要があるならば、『CALL MPI_GET_COUNT』を使用して下さい。
  MPI_ISEND に対して MPI_WAIT をコールした場合、MPI_WAIT が完了すると status(MPI_SOURCE) に値がセットされますが、この値は処理系によって異なり、またプログラムで使用することはありません。ただし、値がセットされるので配列 status(MPI_SOURCE) の宣言は必要です。
  また、4-7-1-3 節のマスタースレーブ方式などで MPI_ISEND と MPI_IRECV の両方を使用していて、前述のように MPI_IRECV に対する MPI_WAIT でセットされる status の値をプログラムで使用したい場合、MPI_ISEND に対する MPI_WAIT でセットされる status と別の配列名にしないと値が壊れてしまいますので注意して下さい。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

サンプルプログラム

ランク0からランク1にメッセージを送信します。

```
      PROGRAM MAIN
      INCLUDE 'mpif.h'                          MPIが使用するインクルードファイルを指定します。
      INTEGER ISTATUS(MPI_STATUS_SIZE)          MPI_WAITで使用する状況オブジェクトを指定します。
      CALL MPI_INIT(IERR)                       MPI環境の初期化処理を行います。
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)  プロセス数NPROCSを取得します。
      CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)  自分のランクMYRANKを取得します。
      IF (MYRANK == 0) THEN                     ランク0のプロセスは送信バッファーにデータを
        ISBUF = 9                               セットします。
      ENDIF
      IF (MYRANK == 0) THEN
        CALL MPI_ISEND(ISBUF,1,MPI_INTEGER,     ランク0はランク1にメッセージを送信します。
     &                 1,1,MPI_COMM_WORLD,IREQ,IERR)
      ELSEIF (MYRANK == 1) THEN
        CALL MPI_IRECV(IRBUF,1,MPI_INTEGER,     ランク1は任意のプロセスからの任意のタグの
     &       MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,IREQ,IERR)   メッセージを受信します。
      ENDIF
      CALL MPI_WAIT(IREQ,ISTATUS,IERR)          送信(受信)が完了するまで待ちます。
      IF (MYRANK == 1) THEN
        PRINT *,'IRBUF  = ',IRBUF               通信後の受信バッファーの内容を書き出します。
        PRINT *,'SOURCE = ',ISTATUS(MPI_SOURCE) 送信元プロセスのランクを書き出します。
        PRINT *,'TAG    = ',ISTATUS(MPI_TAG   ) 受信したメッセージのタグを書き出します。
      ENDIF
      CALL MPI_FINALIZE(IERR)                   MPI環境の終了処理を行います。
      END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 2 a.out ↵
  IRBUF  = 9
  SOURCE = 0
  TAG    = 1
```

1 対 1 通信サブルーチン　MPI_GET_COUNT

機能 (関連する節：4-6-10 節)

1 対 1 通信サブルーチン MPI_RECV (または MPI_IRECV) によって受信したメッセージの要素数を調べたい場合に本サブルーチンを使用します。MPI_RECV (または MPI_IRECV に対応する MPI_WAIT) が完了した後、MPI_RECV (または MPI_WAIT) の引数で指定した状況オブジェクト status に各種の情報が戻ります。この status と、MPI_RECV (または MPI_IRECV) の引数で指定した受信メッセージのデータ型 (datatype) を引数で指定して本サブルーチンをコールすると、MPI_RECV (または MPI_IRECV) で受信したメッセージの要素数が count に戻ります。

使用法

```
CALL MPI_GET_COUNT(status, datatype, count, ierror)
```

- status: 大きさが MPI_STATUS_SIZE の整数配列。調べたい MPI_RECV (または MPI_IRECV に対応する MPI_WAIT) の引数で指定した状況オブジェクト status を指定します。
- datatype: 整数。MPI_RECV (または MPI_IRECV) の引数で指定した受信メッセージのデータ型 (datatype) を指定します。
- count: 整数。MPI_RECV (または MPI_IRECV) によって受信したメッセージの要素数が戻ります。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

注意

- 本サブルーチンは、調べたい MPI_RECV (または MPI_IRECV に対応する MPI_WAIT) が完了した後でコールする必要があります。

サンプルプログラム

ランク0からランク1にメッセージを送信します。

```
      PROGRAM MAIN
      INCLUDE 'mpif.h'                          MPIが使用するインクルードファイルを指定します。
      INTEGER ISTATUS(MPI_STATUS_SIZE)          MPI_WAITで使用する状況オブジェクトを指定します。
      CALL MPI_INIT(IERR)                       MPI環境の初期化処理を行います。
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)  プロセス数NPROCSを取得します。
      CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)  自分のランクMYRANKを取得します。
      IF (MYRANK == 0) THEN                     ランク0のプロセスは送信バッファーにデータを
        ISBUF = 9                               セットします。
      ENDIF
      IF (MYRANK == 0) THEN
        CALL MPI_ISEND(ISBUF,1,MPI_INTEGER,     ランク0はランク1にメッセージを送信します。
     &                 1,1,MPI_COMM_WORLD,IREQ,IERR)
      ELSEIF (MYRANK == 1) THEN
        CALL MPI_IRECV(IRBUF,1,MPI_INTEGER,     ランク1は任意のプロセスからの任意のタグの
     &       MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,IREQ,IERR)   メッセージを受信します。
      ENDIF
      CALL MPI_WAIT(IREQ,ISTATUS,IERR)          送信(受信)が完了するまで待ちます。
      CALL MPI_GET_COUNT(ISTATUS,MPI_INTEGER,ICOUNT,IERR)  受信したメッセージの要素数を調べます。
      IF (MYRANK == 1) THEN
        PRINT *,'IRBUF  = ',IRBUF                 通信後の受信バッファーの内容を書き出します。
        PRINT *,'SOURCE = ',ISTATUS(MPI_SOURCE)   送信元プロセスのランクを書き出します。
        PRINT *,'TAG    = ',ISTATUS(MPI_TAG   )   受信したメッセージのタグを書き出します。
        PRINT *,'COUNT  = ',ICOUNT                受信したメッセージの要素数を書き出します。
      ENDIF
      CALL MPI_FINALIZE(IERR)                   MPI環境の終了処理を行います。
      END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 2 a.out ↵
  IRBUF  = 9
  SOURCE = 0
  TAG    = 1
  COUNT  = 1
```

派生データ型に関するサブルーチン　MPI_TYPE_SIZE

機能 (関連する節：3-5-2 節)

指定したデータ型 (datatype) の大きさが size に戻ります。C 言語の sizeof 関数に相当します。

使用法

CALL MPI_TYPE_SIZE(datatype, size, ierror)

- datatype: 整数。大きさを調べたいデータ型を指定します。
- size: 整数。指定したデータ型 (datatype) の大きさ（単位はバイト）が戻ります。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

【例 1】

size には 4（バイト）が戻ります。

【例 2】

size には 12（バイト）が戻ります。

サンプルプログラム

【例 1】のプログラム例です。

```
PROGRAM MAIN
INCLUDE 'mpif.h'                          MPIが使用するインクルードファイルを指定します。
CALL MPI_INIT(IERR)                       MPI環境の初期化処理を行います。
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)  プロセス数NPROCSを取得します。
CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)  自分のランクMYRANKを取得します。
CALL MPI_TYPE_SIZE(MPI_INTEGER,ISIZE,IERR)      MPI_INTEGERの大きさを調べます。
IF (MYRANK==0) PRINT *,'ISIZE = ',ISIZE   結果を書き出します。
CALL MPI_FINALIZE(IERR)                   MPI環境の終了処理を行います。
END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 1 a.out ↵
 ISIZE = 4
```

派生データ型に関するサブルーチン　MPI_TYPE_CONTIGUOUS

機能 (関連する節：3-5-2 節)

- count 個の古いデータ型 (oldtype) を連結して新しいデータ型 (newtype) を作成します。
- 作成した新しいデータ型 (newtype) は、MPI_TYPE_COMMIT で登録してから通信ルーチンで使用して下さい。

使用法

CALL MPI_TYPE_CONTIGUOUS(count, oldtype, newtype, ierror)

- count: 整数。古いデータ型 (oldtype) をいくつ連結するかを指定します。
- oldtype: 整数。古いデータ型を指定します。
- newtype: 整数。作成された新しいデータ型が戻ります。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

【例 1】

サンプルプログラム

【例 1】のプログラム例です。

```
PROGRAM MAIN
INCLUDE 'mpif.h'                          MPIが使用するインクルードファイルを指定します。
INTEGER IBUF(10)                          送(受)信バッファーを指定します。
CALL MPI_INIT(IERR)                       MPI環境の初期化処理を行います。
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)  プロセス数NPROCSを取得します。
CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)  自分のランクMYRANKを取得します。
IF (MYRANK == 0) THEN
  DO I = 1, 10                            ランク0のプロセスは送信バッファーに
    IBUF(I) = I                           データをセットします(ランク0以外のプロセスは
  ENDDO                                   受信バッファーをゼロクリアしますが、
ENDIF                                     スペースの都合で省略しました)。
CALL MPI_TYPE_CONTIGUOUS(3,MPI_INTEGER,INEWTYPE,IERR)   新しいデータ型を作成します。
CALL MPI_TYPE_COMMIT(INEWTYPE,IERR)                     新しいデータ型を登録します。
CALL MPI_BCAST(IBUF,1,INEWTYPE,0,MPI_COMM_WORLD,IERR)   新しいデータ型でメッセージを
PRINT *,'MYRANK = ',MYRANK,'IBUF = ',IBUF               1個通信し、書き出します。
CALL MPI_FINALIZE(IERR)                   MPI環境の終了処理を行います。
END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 2 a.out ↵
  MYRANK = 0 IBUF = 1 2 3 4 5 6 7 8 9 10
  MYRANK = 1 IBUF = 1 2 3 0 0 0 0 0 0 0
```

派生データ型に関するサブルーチン　MPI_TYPE_VECTOR/HVECTOR

機能 (関連する節：3-5-2 節)

- count 個のブロックから構成される新しいデータ型 (newtype) を作成します。
  - 各ブロックには、古いデータ型 (oldtype) の要素が blocklength 個含まれます。
  - 各ブロックの先頭間の間隔は stride です。
- 作成した新しいデータ型 (newtype) は、MPI_TYPE_COMMIT で登録してから通信ルーチンで使用して下さい。

使用法

```
CALL MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype, ierror)
CALL MPI_TYPE_HVECTOR(count, blocklength, stride, oldtype, newtype, ierror)
```

- count: 整数。ブロックの数を指定します。
- blocklength: 整数。各ブロックに含まれる古いデータ型の要素の数（全ブロックで同一）を指定します。
- stride: 整数。各ブロックの先頭間の間隔（一定）を指定します。
  単位は、MPI_TYPE_VECTOR では古いデータ型の要素の数、MPI_TYPE_HVECTOR ではバイトです。
- oldtype: 整数。古いデータ型（全ブロックで同一）を指定します。
- newtype: 整数。作成された新しいデータ型が戻ります。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

【例 1】

サンプルプログラム

【例1】のプログラム例です。

```
PROGRAM MAIN
INCLUDE 'mpif.h'                                  MPIが使用するインクルードファイルを指定します。
INTEGER IBUF1(20),IBUF2(20)                       送(受)信バッファーを指定します。
CALL MPI_INIT(IERR)                               MPI環境の初期化処理を行います。
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)    プロセス数NPROCSを取得します。
CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)    自分のランクMYRANKを取得します。
IF (MYRANK == 0) THEN
  DO I = 1, 20                                    ランク0のプロセスは送信バッファーに
    IBUF1(I) = I                                  データをセットします。
    IBUF2(I) = I
  ENDDO
ELSE
  DO I = 1, 20                                    ランク0以外のプロセスは受信バッファーを
    IBUF1(I) = 0                                  ゼロクリアします。
    IBUF2(I) = 0
  ENDDO
ENDIF
CALL MPI_TYPE_VECTOR(4,2,3,MPI_INTEGER,INEWTYPE1,IERR)     新しいデータ型を作成します。
CALL MPI_TYPE_COMMIT(INEWTYPE1,IERR)                       新しいデータ型を登録します。
CALL MPI_BCAST(IBUF1,1,INEWTYPE1,0,MPI_COMM_WORLD,IERR)    新しいデータ型でメッセージを
PRINT *,'MYRANK = ',MYRANK,'IBUF1 = ',IBUF1                1個通信し、書き出します。
CALL MPI_TYPE_HVECTOR(4,2,3*4,MPI_INTEGER,INEWTYPE2,IERR)  新しいデータ型を作成します。
CALL MPI_TYPE_COMMIT(INEWTYPE2,IERR)                       新しいデータ型を登録します。
CALL MPI_BCAST(IBUF2,1,INEWTYPE2,0,MPI_COMM_WORLD,IERR)    新しいデータ型でメッセージを
PRINT *,'MYRANK = ',MYRANK,'IBUF2 = ',IBUF2                1個通信し、書き出します。
CALL MPI_FINALIZE(IERR)                           MPI環境の終了処理を行います。
END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 2 a.out ↵
  MYRANK = 0 IBUF1 = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
  MYRANK = 1 IBUF1 = 1 2 0 4 5 0 7 8 0 10 11 0 0 0 0 0 0 0 0
  MYRANK = 0 IBUF2 = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
  MYRANK = 1 IBUF2 = 1 2 0 4 5 0 7 8 0 10 11 0 0 0 0 0 0 0 0
```

【MPI-2】派生データ型に関するサブルーチン　MPI_TYPE_CREATE_INDEXED_BLOCK

機能 (関連する節：3-5-2 節，4-6-6-1 節)

- 本ルーチンは MPI-2 のサブルーチンです。
- count 個のブロックから構成される新しいデータ型 (newtype) を作成します。
  - 各ブロックには、古いデータ型 (oldtype) の要素が blocklength 個含まれます。
  - ブロックごとに、新しいデータ型の先頭から各ブロックの先頭までの変位 (array_of_displacements) を指定します。
- 作成した新しいデータ型 (newtype) は、MPI_TYPE_COMMIT で登録してから通信ルーチンで使用して下さい。

使用法

```
CALL MPI_TYPE_CREATE_INDEXED_BLOCK(count, blocklength, array_of_displacements,
oldtype, newtype, ierror)
```

- count: 整数。ブロックの数を指定します。
  この値は同時に配列 array_of_displacements の要素数を表します。
- blocklength: 整数。各ブロックに含まれる古いデータ型の要素の数（全ブロックで同一）を指定します。
- array_of_displacements: 整数配列。新しいデータ型の先頭から i 個目のブロックの先頭までの変位を array_of_displacements(i) に指定します。単位は古いデータ型の要素の数です。
- oldtype: 整数。古いデータ型（全ブロックで同一）を指定します。
- newtype: 整数。作成された新しいデータ型が戻ります。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

注意

最初のブロックの前にブランクがあっても構いません。ただし、この派生データ型を使用してデータを 2 個以上通信した場合や、この派生データ型を MPI_TYPE_CONTIGUOUS で 2 個以上連続させた場合、2 個目以降の派生データ型では前のブランクが取れてしまいます。ブランクを残したい場合は、3-5-2 節の (7)〜(9) を参照して下さい。

【例 1】

サンプルプログラム

【例1】のプログラム例です。

```
PROGRAM MAIN
INCLUDE 'mpif.h'                          MPIが使用するインクルードファイルを指定します。
INTEGER IBUF(10)                          送(受)信バッファーを指定します。
INTEGER IDISP(4)                MPI_TYPE_CREATE_INDEXED_BLOCKで使用する引数の配列を指定します。
CALL MPI_INIT(IERR)                       MPI環境の初期化処理を行います。
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)  プロセス数NPROCSを取得します。
CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)  自分のランクMYRANKを取得します。
IF (MYRANK == 0) THEN
  DO I = 1, 10                            ランク0のプロセスは送信バッファーに
    IBUF(I) = I                           データをセットします
  ENDDO
ELSE
  DO I = 1, 10                            ランク0以外のプロセスは受信バッファーを
    IBUF(I) = 0                           ゼロクリアします。
  ENDDO
ENDIF
IDISP(1) = 1
IDISP(2) = 4                              新しいデータ型を作成するために
IDISP(3) = 5                              MPI_TYPE_CREATE_INDEXED_BLOCKの各引数を指定します。
IDISP(4) = 7
CALL MPI_TYPE_CREATE_INDEXED_BLOCK                          新しいデータ型を作成します。
&    (4,1,IDISP,MPI_INTEGER,INEWTYPE,IERR)
CALL MPI_TYPE_COMMIT(INEWTYPE,IERR)                         新しいデータ型を登録します。
CALL MPI_BCAST(IBUF,1,INEWTYPE,0,MPI_COMM_WORLD,IERR)       新しいデータ型でメッセージ
PRINT *,'MYRANK = ',MYRANK,'IBUF = ',IBUF                   を1個通信し、書き出します。
CALL MPI_FINALIZE(IERR)                   MPI環境の終了処理を行います。
END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 2 a.out ↵
  MYRANK = 0 IBUF = 1 2 3 4 5 6 7 8 9 10
  MYRANK = 1 IBUF = 0 2 0 0 5 6 0 8 0 0
```

派生データ型に関するサブルーチン　MPI_TYPE_INDEXED/HINDEXED

機能 (関連する節：3-5-2 節)

- count 個のブロックから構成される新しいデータ型 (newtype) を作成します。
  - 各ブロックには、古いデータ型 (oldtype) の要素が含まれます。
  - ブロックごとに、各ブロックに含まれる要素の数 (array_of_blocklengths)、新しいデータ型の先頭から各ブロックの先頭までの変位 (array_of_displacements) を指定します。
- 作成した新しいデータ型 (newtype) は、MPI_TYPE_COMMIT で登録してから通信ルーチンで使用して下さい。

使用法

```
  CALL MPI_TYPE_INDEXED(count, array_of_blocklengths, array_of_displacements,
oldtype, newtype, ierror)
  CALL MPI_TYPE_HINDEXED(count, array_of_blocklengths, array_of_displacements,
oldtype, newtype, ierror)
```

- count: 整数。ブロックの数を指定します。
  この値は同時に配列 array_of_blocklengths, array_of_displacements の要素数を表します。
- array_of_blocklengths: 整数配列。i 個目のブロック内に含まれる古いデータ型の要素の数を array_of_blocklengths(i) に指定します。
- array_of_displacements: 整数配列。新しいデータ型の先頭から i 個目のブロックの先頭までの変位を array_of_displacements(i) に指定します。
  単位は、MPI_TYPE_INDEXED では古いデータ型の要素の数、MPI_TYPE_HINDEXED ではバイトです。
- oldtype: 整数。古いデータ型（全ブロックで同一）を指定します。
- newtype: 整数。作成された新しいデータ型が戻ります。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

注意

最初のブロックの前にブランクがあっても構いません。ただし、この派生データ型を使用してデータを 2 個以上通信した場合や、この派生データ型を MPI_TYPE_CONTIGUOUS で 2 個以上連続させた場合、2 個目以降の派生データ型では前のブランクが取れてしまいます。ブランクを残したい場合は、3-5-2 節の (7)〜(9) を参照して下さい。

【例 1】

サンプルプログラム

【例1】のプログラム例です。

```
PROGRAM MAIN
INCLUDE 'mpif.h'                          MPIが使用するインクルードファイルを指定します。
INTEGER IBUF1(10),IBUF2(10)               送(受)信バッファーを指定します。
INTEGER IBLOCK(2),IDISP(2)               MPI_TYPE_(H)INDEXEDで使用する引数の配列を指定します。
CALL MPI_INIT(IERR)                       MPI環境の初期化処理を行います。
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)  プロセス数NPROCSを取得します。
CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)  自分のランクMYRANKを取得します。
IF (MYRANK == 0) THEN
  DO I = 1, 10                            ランク0のプロセスは送信バッファーに
    IBUF1(I) = I                          データをセットします
    IBUF2(I) = I
  ENDDO
ELSE
  DO I = 1, 10                            ランク0以外のプロセスは受信バッファーを
    IBUF1(I) = 0                          ゼロクリアします。
    IBUF2(I) = 0
  ENDDO
ENDIF
IBLOCK(1) = 2
IBLOCK(2) = 3                             新しいデータ型を作成するために
IDISP(1)  = 0                             MPI_TYPE_INDEXEDの各引数を指定します。
IDISP(2)  = 3
CALL MPI_TYPE_INDEXED(2,IBLOCK,IDISP,MPI_INTEGER,INEWTYPE1,IERR) 新しいデータ型を作成します。
CALL MPI_TYPE_COMMIT(INEWTYPE1,IERR)                             新しいデータ型を登録します。
CALL MPI_BCAST(IBUF1,1,INEWTYPE1,0,MPI_COMM_WORLD,IERR)          新しいデータ型でメッセージ
PRINT *,'MYRANK = ',MYRANK,'IBUF1 = ',IBUF1                      を1個通信し、書き出します。
IBLOCK(1) = 2
IBLOCK(2) = 3                             新しいデータ型を作成するために
IDISP(1)  = 0*4                           MPI_TYPE_HINDEXEDの各引数を指定します。
IDISP(2)  = 3*4
CALL MPI_TYPE_HINDEXED(2,IBLOCK,IDISP,MPI_INTEGER,INEWTYPE2,IERR)新しいデータ型を作成します。
CALL MPI_TYPE_COMMIT(INEWTYPE2,IERR)                             新しいデータ型を登録します。
CALL MPI_BCAST(IBUF2,1,INEWTYPE2,0,MPI_COMM_WORLD,IERR)          新しいデータ型でメッセージ
PRINT *,'MYRANK = ',MYRANK,'IBUF2 = ',IBUF2                      を1個通信し、書き出します。
CALL MPI_FINALIZE(IERR)                   MPI環境の終了処理を行います。
END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 2 a.out ↵
  MYRANK = 0 IBUF1 = 1 2 3 4 5 6 7 8 9 10
  MYRANK = 1 IBUF1 = 1 2 0 4 5 6 0 0 0 0
  MYRANK = 0 IBUF2 = 1 2 3 4 5 6 7 8 9 10
  MYRANK = 1 IBUF2 = 1 2 0 4 5 6 0 0 0 0
```

派生データ型に関するサブルーチン　MPI_TYPE_STRUCT

機能 (関連する節：3-5-2 節，3-5-4 節，3-3-6-2 節)

- count 個のブロックから構成される新しいデータ型 (newtype) を作成します。ブロックごとに、各ブロックに含まれる要素の数 (array_of_blocklengths)、新しいデータ型の先頭から各ブロックの先頭までの変位 (array_of_displacements)、各ブロックに含まれる要素の古いデータ型 (array_of_types) を指定します。
- 作成した新しいデータ型 (newtype) は、MPI_TYPE_COMMIT で登録してから通信ルーチンで使用して下さい。

使用法

```
CALL MPI_TYPE_STRUCT(count, array_of_blocklengths, array_of_displacements,
array_of_types, newtype, ierror)
```

- count: 整数。ブロックの数を指定します。
  この値は同時に配列 array_of_types, array_of_displacements, array_of_blocklengths の要素数を表します。
- array_of_blocklengths: 整数配列。i 個目のブロック内に含まれる古いデータ型の要素の数を array_of_blocklengths(i) に指定します。
- array_of_displacements: 整数配列。新しいデータ型の先頭から i 個目のブロックの先頭までの変位（単位はバイト）を array_of_displacements(i) に指定します。
- array_of_types: 整数配列。i 個目のブロック内に含まれる要素の古いデータ型を array_of_types(i) に指定します。
- newtype: 整数。作成された新しいデータ型が戻ります。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

注意

最初のブロックの前にブランクがあっても構いません。ただし、この派生データ型を使用してデータを 2 個以上通信した場合や、この派生データ型を MPI_TYPE_CONTIGUOUS で 2 個以上連続させた場合、2 個目以降の派生データ型では前のブランクが取れてしまいます (3-5-2 節の (7)〜(9) 参照)。

ブランクを残したい場合は、【例 2】のように、ブランクの前に要素数 1、大きさ 0、データ型が「MPI_LB」(LB は Lower Bound の略) のブロックを定義します。同様に、最後のブロックの後にブランクを入れたい場合は、ブランクの後に要素数 1、大きさ 0、データ型が「MPI_UB」(UB は Upper Bound の略) のブロックを定義します。なお、MPI_TYPE_CREATE_RESIZED（付録参照）を使用してブランクを入れることもできます (3-5-2 節の (7)〜(9) 参照)。

サンプルプログラム

【例1】と【例2】のプログラム例です。

```
PROGRAM MAIN
INCLUDE 'mpif.h'                          MPIが使用するインクルードファイルを指定します。
INTEGER IBUF1(10),IBUF2(10)               送(受)信バッファーを指定します。
INTEGER IBLOCK(2),IDISP(2),ITYPE(2)       MPI_TYPE_STRUCTで使用する引数の配列を指定します。
CALL MPI_INIT(IERR)                       MPI環境の初期化処理を行います。
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)  プロセス数NPROCSを取得します。
CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)  自分のランクMYRANKを取得します。
IF (MYRANK == 0) THEN
  DO I = 1, 10                            ランク0のプロセスは送信バッファーに
    IBUF1(I) = I                          データをセットします
    IBUF2(I) = I
  ENDDO
ELSE
  DO I = 1, 10                            ランク0以外のプロセスは受信バッファーを
    IBUF1(I) = 0                          ゼロクリアします。
    IBUF2(I) = 0
  ENDDO
ENDIF
IBLOCK(1) = 2
IBLOCK(2) = 3
IDISP(1)  = 0*4                           【例1】の新しいデータ型を作成するために
IDISP(2)  = 3*4                           MPI_TYPE_STRUCTの各引数を指定します。
ITYPE(1)  = MPI_INTEGER
ITYPE(2)  = MPI_INTEGER
CALL MPI_TYPE_STRUCT(2,IBLOCK,IDISP,ITYPE,INEWTYPE1,IERR)【例1】の新データ型を作成します。
CALL MPI_TYPE_COMMIT(INEWTYPE1,IERR)                     【例1】の新データ型を登録します。
CALL MPI_BCAST(IBUF1,1,INEWTYPE1,0,MPI_COMM_WORLD,IERR)  【例1】の新データ型でメッセージを
PRINT *,'MYRANK = ',MYRANK,'IBUF1 = ',IBUF1              1個通信し、書き出します。
IBLOCK(1) = 1
IBLOCK(2) = 3
IDISP(1)  = 0*4                           【例2】の新しいデータ型を作成するために
IDISP(2)  = 2*4                           MPI_TYPE_STRUCTの各引数を指定します。
ITYPE(1)  = MPI_LB
ITYPE(2)  = MPI_INTEGER
CALL MPI_TYPE_STRUCT(2,IBLOCK,IDISP,ITYPE,INEWTYPE2,IERR)【例2】の新データ型を作成します。
CALL MPI_TYPE_COMMIT(INEWTYPE2,IERR)                     【例2】の新データ型を登録します。
CALL MPI_BCAST(IBUF2,2,INEWTYPE2,0,MPI_COMM_WORLD,IERR)  【例2】の新データ型でメッセージを
PRINT *,'MYRANK = ',MYRANK,'IBUF2 = ',IBUF2              2個通信し、書き出します。
CALL MPI_FINALIZE(IERR)                   MPI環境の終了処理を行います。
END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 2 a.out ↵
  MYRANK = 0 IBUF1 = 1 2 3 4 5 6 7 8 9 10   【例1】の結果です。
  MYRANK = 1 IBUF1 = 1 2 0 4 5 6 0 0 0 0
  MYRANK = 0 IBUF2 = 1 2 3 4 5 6 7 8 9 10   【例2】の結果です。
  MYRANK = 1 IBUF2 = 0 0 3 4 5 0 0 8 9 10
```

【MPI-2】派生データ型に関するサブルーチン　MPI_TYPE_CREATE_SUBARRAY

機能 (関連する節：3-5-3-1 節，4-6-3-1 節，4-6-3-2 節)

- 【例 1】に示すように、全体配列内に含まれる部分配列を表す新しいデータ型 (newtype) を作成します。配列の次元数 (ndims) は何次元でも可能です。
- 作成した新しいデータ型 (newtype) は、MPI_TYPE_COMMIT で登録してから通信ルーチンで使用して下さい。
- 作成した新しいデータ型 (newtype) を通信ルーチンで使用する際、送／受信バッファーの先頭アドレスの引数には、全体配列の先頭アドレス（全体配列名でも可）を指定します。
- 3-5-3 節で紹介した派生データ型のユーティリティ・サブルーチン PARA_TYPE_BLOCK2、PARA_TYPE_BLOCK3 は、本ルーチンと同様の機能を持ちます。本ルーチンは MPI-2 に含まれているので、MPI-2 が使用できない環境の場合、代替として 3-5-3 節のルーチンを使用して下さい。

使用法

CALL MPI_TYPE_CREATE_SUBARRAY(ndims, array_of_sizes, array_of_subsizes, array_of_starts, order, oldtype, newtype, ierror)

- ndims: 整数。全体配列の次元数を指定します。
- array_of_sizes: 大きさ ndims の整数配列。各次元ごとに、全体配列の古いデータ型の要素の数を指定します。
- array_of_subsizeis: 大きさ ndims の整数配列。各次元ごとに、部分配列の古いデータ型の要素の数を指定します。
- array_of_starts: 大きさ ndims の整数配列。各次元ごとに、部分配列の最初の要素の座標を指定します。座標は【例 1】の［］に示す相対的な座標 (全体配列の最初の要素の座標を 0 としたときの座標) で指定します。
- order: 整数。Fortran の場合は MPI_ORDER_FORTRAN を、C の場合は MPI_ORDER_C を指定します。
- oldtype: 整数。古いデータ型を指定します。
- newtype: 整数。作成された新しいデータ型が戻ります。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

【例 1】

サンプルプログラム

【例1】のプログラム例です。

```
PROGRAM MAIN
INCLUDE(mpif.h)                                  MPIが使用するインクルードファイルを指定します。
INTEGER IBUF(3:7,-2:4)                           送(受)信バッファー(全体配列)を指定します。
INTEGER ISIZE(2),ISUBSIZE(2),ISTART(2)
CALL MPI_INIT(IERR)                              MPI環境の初期化処理を行います。
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)   プロセス数NPROCSを取得します。
CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)   自分のランクMYRANKを取得します。
IF(MYRANK==0) THEN
  DO J=-2,4
    DO I=3,7                                     ランク0のプロセスは送信バッファーに
      IBUF(I,J) = I*10+J                         データをセットします。
    ENDDO
  ENDDO
ELSE
  DO J=-2,4
    DO I=3,7                                     ランク0以外のプロセスは受信バッファーを
      IBUF(I,J) = 0                              ゼロクリアします。
    ENDDO
  ENDDO
ENDIF
ISIZE(1)=5; ISUBSIZE(1)=3; ISTART(1)=0           新しいデータ型(部分配列)を作成するために、
ISIZE(2)=7; ISUBSIZE(2)=4; ISTART(2)=2           MPI_TYPE_CREATE_SUBARRAYの各引数を指定します。
CALL MPI_TYPE_CREATE_SUBARRAY(2,ISIZE,ISUBSIZE,ISTART,   新しいデータ型を作成します。
&     MPI_ORDER_FORTRAN,MPI_INTEGER,INEWTYPE,IERR)
CALL MPI_TYPE_COMMIT(INEWTYPE,IERR)              新しいデータ型を登録します。
CALL MPI_BCAST(IBUF,1,INEWTYPE,0,MPI_COMM_WORLD,IERR)   新しいデータ型でメッセージを
IF (MYRANK==1) PRINT *,'IBUF = ',IBUF            1個通信し、書き出します。
CALL MPI_FINALIZE(IERR)                          MPI環境の終了処理を行います。
END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 2 a.out ↵
    IBUF =  0 0 0 0 0 0 0 0 0 0 30 40 50 0 0 31 41 51 0 0 32 42 52 0 0 33 43 53 0 0 0 0 0 0 0
```

派生データ型に関するサブルーチン　MPI_TYPE_COMMIT

機能 (関連する節：3-5-2 節)

MPI_TYPE_CONTIGUOUS、MPI_TYPE_(H)VECTOR、MPI_TYPE_CREATE_INDEXED_BLOCK, MPI_TYPE_(H)INDEXED、MPI_TYPE_STRUCT、MPI_TYPE_CREATE_SUBARRAY などで作成した派生データ型 (datatype) は、通信ルーチンで使用する前に、MPI_TYPE_COMMIT で一度だけ登録します。登録が完了すると、そのデータ型は通信ルーチンのデータ型引数に何度でも指定することができます。

使用法

CALL MPI_TYPE_COMMIT(datatype, ierror)

- datatype: 整数。登録したい、MPI_TYPE_CONTIGUOUS、MPI_TYPE_(H)VECTOR、MPI_TYPE_CREATE_INDEXED_BLOCK, MPI_TYPE_(H)INDEXED、MPI_TYPE_STRUCT、MPI_TYPE_CREATE_SUBARRAY などで作成した派生データ型を指定します。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

注意

- 新しいデータ型が、通信ルーチンで実際に使用する派生データ型でなく、途中で作られる一時的な派生データ型の場合、本サブルーチンで登録する必要はありません。

サンプルプログラム

MPI_TYPE_CONTIGUOUS、MPI_TYPE_(H)VECTOR、MPI_TYPE_CREATE_INDEXED_BLOCK, MPI_TYPE_(H)INDEXED、MPI_TYPE_STRUCT、MPI_TYPE_CREATE_SUBARRAY などの例を参照して下さい。

【MPI-2】派生データ型に関するサブルーチン　MPI_TYPE_CREATE_RESIZED

機能 (関連する節：3-5-2 節)

- 本ルーチンは MPI-2 のサブルーチンです。
- 本サブルーチンは、古いデータ型 (oldtype) の後にブランクを入れるときに使用します (3-5-2 節の (7)〜(9) 参照)。
- 古いデータ型 (oldtype) に対して下限の位置 (lb) と範囲 (extent) を設定し、新しい（派生）データ型 (newtype) を作成します。上限の位置は「lb + extent」になります。
- 作成した新しいデータ型 (newtype) は、MPI_TYPE_COMMIT で登録してから通信ルーチンで使用して下さい。

使用法

```
CALL MPI_TYPE_CREATE_RESIZED(oldtype, lb, extent, newtype, ierror)
```

- oldtype: 整数。古いデータ型を指定します。
- lb: 整数 (INTEGER(KIND=MPI_ADDRESS_KIND) で宣言して下さい)。新しいデータ型の下限の位置（先頭からの変位）を指定します（単位はバイト）。
- extent: 整数 (INTEGER(KIND=MPI_ADDRESS_KIND) で宣言して下さい)。新しいデータ型の範囲を指定します（単位はバイト）。
- newtype: 整数。作成された新しいデータ型が戻ります。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

【例 1】付録の MPI_TYPE_CONTIGUOUS の【例 1】で作成した派生データ型 (以下では IOLDTYPE) の後ろにブランクを 1 つ付けます。

サンプルプログラム

【例 1】のプログラム例です (付録の MPI_TYPE_CONTIGUOUS の【例 1】参照)。

```
  :
INTEGER(KIND=MPI_ADDRESS_KIND) ILB,IEXTENT
  :
CALL MPI_TYPE_CONTIGUOUS(3,MPI_INTEGER,IOLDTYPE,IERR)          IOLDTYPEを作成します。
ILB = 0*4
IEXTENT = 4*4
CALL MPI_TYPE_CREATE_RESIZED(IOLDTYPE,ILB,IEXTENT,INEWTYPE,IERR) INEWTYPEを作成します。
CALL MPI_TYPE_COMMIT(INEWTYPE,IERR)                            INEWTYPEを登録します。
CALL MPI_BCAST(IBUF,2,INEWTYPE,0,MPI_COMM_WORLD,IERR)          INEWTYPEでメッセージを
PRINT *,'MYRANK = ',MYRANK,'IBUF = ',IBUF                      2個通信し、書き出します。
  :
```

```
[aoyama@node01]/u/aoyama: mpirun -np 2 a.out ↵
  MYRANK = 0 IBUF = 1 2 3 4 5 6 7 8 9 10
  MYRANK = 1 IBUF = 1 2 3 0 5 6 7 0 0 0
```

コミュニケータに関するサブルーチン　MPI_COMM_SPLIT

機能 (関連する節：3-6 節)

旧グループ (コミュニケータが comm) に属する各プロセスを分割し、いくつかの新グループ (コミュニケータが newcomm) を作成します。実行が終了すると、新グループに付けられた値が newcomm に戻ります。

旧グループ (コミュニケータが comm) に属する全プロセスが、本サブルーチンをコールする必要があります。

color で指定した値が同一であるプロセスが同じ新グループに所属し、新グループ内のランク (0, 1, 2, ...) は key で指定した値の昇順に付けられます。

使用法

CALL MPI_COMM_SPLIT(comm, color, key, newcomm, ierror)

- comm: 整数。旧グループのコミュニケータ（グループ名）を指定します。
- color: 整数。ここで指定した値が同一であるプロセスが同じ新グループに所属します。ゼロ以上の整数値を指定して下さい。MPI_UNDEFINED を指定した場合には、そのプロセスは新グループに所属せず、newcomm には MPI_COMM_NULL が戻ります。
- key: 整数。ここで指定した値の昇順に、新グループ内のランクが 0, 1, 2, ... の順につけられます。key の値が同一のプロセスが複数ある場合は、旧グループのランクの順にランクがつけられます。
- newcomm: 整数。作成する新グループのコミュニケータ（グループ名）を指定します。実行が終了すると、新グループに付けられた値が戻ります。
- ierror: 整数。完了コードが戻ります。正常終了の場合は「MPI_SUCCESS」という値が戻ります。

【例 1】

【例 2】

サンプルプログラム

【例1】のプログラム例です。

```
PROGRAM MAIN
INCLUDE 'mpif.h'                                MPIが使用するインクルードファイルを指定します。
CALL MPI_INIT(IERR)                             MPI環境の初期化処理を行います。
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)  プロセス数NPROCSを取得します。
CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)  自分のランクMYRANKを取得します。
IF (MYRANK == 0) THEN                           各プロセスは、新グループでのcolorとkeyの値を
  ICOLOR = 1                                    設定します。
  IKEY   = 2
ELSEIF (MYRANK == 1) THEN
  ICOLOR = 1
  IKEY   = 1
ELSEIF (MYRANK == 2) THEN
  ICOLOR = 2
  IKEY   = 1
ELSEIF (MYRANK == 3) THEN
  ICOLOR = 2
  IKEY   = 2
ENDIF
CALL MPI_COMM_SPLIT(MPI_COMM_WORLD,ICOLOR,IKEY,NEW_WORLD,IERR)  新グループを作成します。
CALL MPI_COMM_SIZE(NEW_WORLD,NEWPROCS,IERR)  新グループのプロセス数NEWPROCSを取得します。
CALL MPI_COMM_RANK(NEW_WORLD,NEWRANK ,IERR)  新グループでのランクNEWRANKを取得します。
PRINT *,'MYRANK = ',MYRANK,'NEWPROCS = ',NEWPROCS,'NEWRANK = ',NEWRANK
CALL MPI_FINALIZE(IERR)                         MPI環境の終了処理を行います。
END
```

```
[aoyama@node01]/u/aoyama: mpirun -np 4 a.out ↵
 MYRANK = 0 NEWPROCS = 2 NEWRANK = 1
 MYRANK = 1 NEWPROCS = 2 NEWRANK = 0
 MYRANK = 2 NEWPROCS = 2 NEWRANK = 0
 MYRANK = 3 NEWPROCS = 2 NEWRANK = 1
```

付 録　参考文献

＊がついている資料は英語で書かれています。本書を改訂した影響で、文献の参照番号が不連続になっています。

[1] 本書は、http://accc.riken.jp/HPC/training.html に掲載されています。

[2] Linux で並列処理をしよう　第 2 版　石川裕他　著（共立出版）

[3] 反復法の数理　藤野清次、張紹良　著（朝倉書店）

[4] 並列プログラミングの基礎　大森健児　訳（丸善株式会社）

[5] 並列計算法入門　樫山和男、西村直志、牛島省　著（丸善）

[6] チューニング技法入門　青山幸也　著　http://accc.riken.jp/HPC/training/text.html

[15] *MPI: A Message-Passing Interface Standard Message Passing Interface Forum　MPI 標準そのもので、http://www.mpi-forum.org/ からダウンロードすることができます ([27] に日本語訳があります)。MPI-2.0 の第 3 章に MPI-1.1 の正誤表があります。またこのサイトの「How to make comments」で、MPI に関する質問（英語）を受け付けています。

[16] 超並列有限要素解析　矢川元基、塩谷隆二　著（朝倉書店）

[17] 計算力学と CAE シリーズ 7 パラレルコンピューティング　矢川元基、曽根田直樹　著（培風館）
有限要素法の領域分割法による並列化方法について説明されています。

[18] COM シリーズ超並列コンピュータ入門　野口正一　監修、村岡洋一、山名早人　著（オーム社）
ベクトル計算機と並列計算機の基本的な項目が説明されています。

[19] 計算力学 7—計算力学における超並列計算法　矢川元基、奥田洋司　共編（養賢堂）

[20] *USING MPI，Using MPI-2 (2 分冊になりました)
William Gropp，Ewing Lusk，Anthony Skjellum　著 (The MIT Press)

[21] 実践 MPI-2　ウィリアム・グロップ他　著、畑崎隆雄　訳（株式会社ピアソン・エデュケーション）
[20] の第 2 巻の日本語訳です。

[22] *MPI: The Complete Reference (2 分冊になりました) Marc Snir，Steve W. Otto，Steven Huss-Lederman，David W. Walker，Jack Dongarra　著 (The MIT Press)

[23] 行列計算ソフトウェア　WS，スーパーコン，並列計算機　小国力　編著、村田健郎、三好俊郎、ドンガラ，J. J.、長谷川秀彦　著（丸善）

[24] スーパーコンピュータ　科学技術計算への適用　村田健郎、小国力、唐木　幸比古　著（丸善）

[27] http://phase.hpcc.jp/phase/mpi-j/ml/　MPI 仕様書 [15] の日本語訳があります。MPI-2.0 の第 3 章に MPI-1.1 の正誤表があります。

[28] http://www.egroups.co.jp/group/mpi-j　MPI に関するメーリングリストです。

[31] Twisted Data Layout　進藤達也、岩下英俊、土肥実久、萩原純一、金城ショーン　著
「並列処理シンポジウム JSPP '94」ツイスト分割法についての論文です。

[36] 入門 Fortran 90 実践プログラミング　東田幸樹、山本芳人、熊沢友信　著 (SOFT BANK)

[37] *PARALLEL PROGRAMMING with MPI　Peter S. Pacheco　著 (Morgan Kaufmann Publishers, Inc.)
例題は全て C 言語で書かれていますが、Fortran 版も http://www.usfca.edu/mpi にあります。

[38] MPI 並列プログラミング　P. パチェコ　著、秋葉博　訳（培風館）
[37] の日本語訳です。

[39]「超並列コンピューティングの現状と動向調査」調査報告書（富士総合研究所）
現在稼動中の並列コンピュータの特性や市場動向、並列アルゴリズムなどが紹介されています。

[40] 富士総研技報 vol. 5 No. 1 1995 特集超並列コンピューティング（富士総合研究所）
現在稼動中の並列コンピュータの特性や市場動向、並列アルゴリズムなどが紹介されています。

[41] MPI に関連するインターネットのリンクを以下に示します (参考文献 [37] より転載)。
- フリーウェアの MPI
  `ftp://info.mcs.anl.gov` (mpich)
  `ftp://ftp.osc.edu/pub/lam` (LAM)
  `ftp://ftp.epcc.ed.ac.uk/pub/chimp/release` (CHIMP)
  `ftp://csftp.unomaha.edu/pub/rewini/WinMPI` (WinMPI)
  `http://pandora.uc.pt/w32mpi` (W32MPI)
  `http://www.open-mpi.org/` (Open MPI)
- MPI の FAQ (frequently asked questions) です。
  `http://www.erc.msstate.edu/mpi/mpi-faq.html`
- MPI Web Page
  `http://www.mcs.anl.gov/mpi`
  `http://www.erc.msstate.edu/mpi`
  `http://www.epm.ornl.gov/~walker/mpi`
- ニュースグループ
  `comp.parallel.mpi`
  `fj.comp.parallel`
- MPI-2，MPI-IO
  `http://www.mcs.anl.gov/Projects/mpi/mpi2/mpi2.html`
  `http://lovelace.nas.nasa.gov/MPI-IO/mpi-io.html`
- 参考文献 [37] のホームページとサンプルプログラム
  `http://www.mkp.com`
  `http://www.usfca.edu/mpi`

[42] 学会／メーリングリスト／リンク集など
- PHASE (Parallel and HPC Application Software Exchange)
  `http://phase.hpcc.jp/`
  HPC 関係の各種リンクやメーリングリストのリンクがあります。
- SWoPP メーリングリスト
  `http://www.hpcc.jp/swopp/ml_readme.html`
- SofTek HPC メーリングリスト
  `http://www.softek.co.jp/`
- SMPP（超並列計算研究会）
  `http://www.is.doshisha.ac.jp/SMPP/`
  「PC クラスタ超入門」というダウンロードできる資料が置いてあります。
- CFD ネット（数値流体力学関係のメーリングリスト）
  `majordomo@cfd.ritsumei.ac.jp` 宛てに『`subscribe cfdnet` 登録したいメールアドレス』だけを書いたメールを送付すると、管理者が後で登録します。
- HPC 関係のニュースを集めたサイト
  `http://ytaka.at.infoseek.co.jp/`
- 日本計算工学会
  `http://www2.kajima.co.jp/jsces/wwwjsces/jsces.htm`
- 情報処理学会 HPC 研究会
  `http://phase.etl.go.jp/sighpc/`
- 新情報処理開発機構（並列処理シンポジウムへのリンクあり）
  `http://www.rwcp.or.jp/`
- 日本原子力研究開発機構システム計算科学センター
  `http://ccse.jaea.go.jp/ja/index.html`

- SWoPP（並列／分散／協調処理に関するサマー・ワークショップ）
  http://www.hpcc.jp/swopp/

[43] 数値計算ライブラリー
- PCP（離散化並列解法のための並列計算プラットフォーム）
  http://www.ibase.aist.go.jp/infobase/pcp/index.html
- SSI（大規模シミュレーション向け基盤ソフトウェア）
  http://www.ssisc.org/
- ScaLAPACK
  http://www.netlib.org/scalapack/index.html
  以下に、LAPACK と ScaLAPACK に関する日本語のドキュメントがあります。
  http://phase.hpcc.jp/phase/lapack-j/index.html
  http://phase.hpcc.jp/phase/lapack-j/ScaLAPACK/index.htm
  http://spring.cc.kyushu-u.ac.jp/scp/system/manual/ScaLAPACK/scalapack.pdf
- PETSc http://www-unix.mcs.anl.gov/petsc/petsc-2/
- Lis http://www.ssisc.org/lis/
- PARCEL http://guide.tokai.jaeri.go.jp/program/software/list/pg1/index.html
- GeoFEM http://geofem.tokyo.rist.or.jp/
- FFTW http://www.fftw.org/ (高速フーリエ変換を高速に行う FFTW のサイト)
- ICCG 法の並列版（京都大学大型計算機センター）
  http://fem.kuee.kyoto-u.ac.jp/~take/parasolver/piccg/index.html (~ は波線です)
- NAG http://www.nag-j.co.jp/（日本ニューメリカルアルゴリズムズグループ（株））
- IMSL http://www.vnij.co.jp/（日本ビジュアルニューメリックス（株））
- SMS（行列計算ライブラリー）(MPI 版は開発中？)
  http://www.vinas.com/seihin/sms/index.html（(株）ヴァイナス）

[44] *On parallelism and convergence of incomplete LU factorizations, Shun Doi, Applied Numerical Mathematics 7 (1991) 417–436, North-Holland

[45] *Introduction to Parallel Computing, Design and Analysis of Algorithms, Vipin Kumar, Ananth Grama, Anshul Gupta, George Karypis, The Benjamin/Cummings Publishing Company, Inc.

[46] bit 別冊　はじめての並列プログラミング　湯淺太一、安村通晃、中田登志之　編（共立出版）

[47] たのしくできる並列処理コンピュータ　小畑正貴　著（東京電機大学出版局）

[48] つくる並列処理コンピュータ　小畑正貴　著（東京電機大学出版局）

[49] *SOLVING PARALLEL DIFFERENTIAL EQUATIONS ON PARALLEL COMPUTERS
JIANPING ZHU　著 (World Scientific)　差分法の並列アルゴリズムに関する良書です。

[50] NUMERICAL RECIPES in C［日本語版］（技術評論社）

[51] ICCG 法 (SCG 法) の説明　http://www.ppc.nec.co.jp/ で、左欄の「登録ソフト」を選択し、ソフト名「psc98」を選択して下さい。

[52] *PARALLEL PROGRAMMING　BARRY WILKINSON，MICHAEL ALLEN　著 (PRENTICE HALL)

[53] 並列プログラミング入門　飯塚肇、緑川博子　共訳（丸善）
[52] の第 I 部の日本語訳です。

[54] http://guide.tokai.jaeri.go.jp/program/　日本原子力研究所計算科学技術推進センターが公開している並列関係の資料（論文や並列ライブラリーなど）です。

[55] 並列有限要素解析 [I]　クラスタコンピューティング　奥田洋司、中島研吾　共編（培風館）

[56] 並列有限要素解析 [II]　並列構造解析ソフトウェア FrontSTR を使いこなす　奥田洋司編著（培風館）

[57] *Parallel Programming in OpenMP　Rohit Chandra 他　著 (MORGAN KAUFMANN PUBLISHERS)

[58] *Parallel Programming in C with MPI and OpenMP　Michael J. Quinn　著 (McGraw-Hill)

[59] *High Performance LINUX CLUSTERS with OSCAR, Rocks, openMosix, & MPI　Joseph D. Sloan　著

[60] OpenMP の仕様書（英語）　http://openmp.org/wp/openmp-specifications/

[61] OpenMP の仕様書 ([60] の日本語訳)　[60] で「Version 3.0, Japanese」を選択して下さい。

[62] OpenMP による並列プログラミングと数値計算法　牛島省　著（丸善）

[63] OpenMP 入門　南里豪志　著　http://www.cc.kyushu-u.ac.jp/scp/system/library/OpenMP/OpenMP.html

[64] OpenMP 入門 (1)〜(4)　計算工学 (2006 No. 4、2007 No. 1, 2, 3)　南里豪志　著（日本計算工学会）

[65] マルチコア CPU のための並列プログラミング並列処理 & マルチスレッド入門
安田絹子、小林林広、飯塚博道、阿部貴之、青柳信吾　著（秀和システム）

[66] C/C++ プログラマーのための OpenMP 並列プログラミング　菅原清文　著（カットシステム）

[67] OpenMP 入門　北山洋幸　著（秀和システム）

[68] *Using OpenMP: Portable Shared Memory Parallel Programming　Barbara Chapman 他　著 (The MIT Press)

[69]「並列コンピューティング技法」千住　治郎　訳（オライリー・ジャパン）

[70]「並列数値処理ー高速化と性能向上のためにー」金田　康正　編著（コロナ社）

[71]「コンピュータ設計の基礎」Hisa Ando　著（マイコミジャーナルブックス）

[72]「プロセッサを支える技術」Hisa Ando　著（技術評論社）

[73]「高性能コンピュータ技術の基礎」Hisa Ando　著（マイコミジャーナルブックス）