MPI: メッセージ通信インターフェース標準

(日本語訳ドラフト)

MPI フォーラム

MPI 日本語訳プロジェクト訳

1996 年 7 月 16 日

原書題名

MPI: A Message-Passing Interface Standard

Message Passing Interface Forum

June 12, 1995 This work was supported in part by ARPA and NSF under grant

ASC-9310330, the National Science Foundation Science and Technology Center Cooperative Agreement No. CCR-8809615, and by the Commission of the European Community through Esprit project P6643 (PPPE).

原書改版記録、著作権表示

第 1.1 版: 1995 年 6 月 1995 年 3 月から、 Message Passing Interface Forum は再び会合を開くようになった。その目的は、 1994 年 5 月 5 日付けの MPI 文書、すなわち下記に第 1.0 版として言及されているものの間違いを修正し、不明瞭な点を明らかにすることであった。そこでの議論の成果が本文書、第 1.1 版である。第 1.0 版からの変更点は比較的小さなものである。本文書の、すべての変更点に印をつけた版も入手できる。この段落は、変更箇所の一例である。

第 1.0 版: 1994 年 6 月 Message Passing Interface Forum (MPIF) は、 40 を越える団体の参加を受けて、 1993 年 1 月から会合を繰り返し、メッセージ通信のためのライブラリインターフェースの標準に関する議論と定義を行った。 MPIF はいかなる公式標準制定組織からの認可も援助も受けていない。

Message Passing Interface の目標は、簡単に言うと、メッセージ通信を行うプログラムを書くための、広く一般に使われる標準を作り出すことである。この目標を実現するには、 MPI は実際的で、ポータブルで、効率がよくかつ融通の効くメッセージ通信の標準を確立しなければならない。

本文書は Message Passing Interface Forum の最終報告第 1.0 版である。本文書には MPI インターフェースのために提案されたすべての技術的項目が盛り込まれている。この草稿の版組は LaTeX により 1995 年 6 月 12 日に行った。

MPI に関する意見は `mpi-comments@cs.utk.edu` に送ってくださるようお願いする。送ってくださった意見は MPIF 委員会のメンバーに転送され、彼らは返事を書こうと努めるはずである。

日本語版著作権表示

もくじ

原書謝辞 x

日本語版謝辞 xii

1 MPI の紹介 1

1.1 概要と目標 1
1.2 この標準を利用する人々 3
1.3 この標準の実現対象となるプラットフォーム 3
1.4 標準に含まれるもの 4
1.5 標準に含まれないもの 4
1.6 この文書の構成 5

2 MPI の用語と規則 7

2.1 文書の表記 7
2.2 手続きの仕様 7
2.3 意味に関する用語 9
2.4 データ型 9
2.4.1 不透明オブジェクト 9
2.4.2 配列引数 11
2.4.3 状態型 12
2.4.4 名前付き定数 12
2.4.5 選択型 12
2.4.6 アドレス型 12
2.5 言語の呼び出し形式 13
2.5.1 Fortran 77 言語の呼び出し形式に関する事項 13
2.5.2 C 言語呼び出し形式に関する事項 14
2.6 プロセス 15
2.7 エラー処理 16

2.8 実装に関する事項 18
2.8.1 基本ランタイム・ルーチンの独立性 18
2.8.2 POSIX シグナルとの相互作用 19
2.9 プログラム例 19

3 1 対 1 通信 **20**
3.1 概要 20
3.2 ブロッキング送信関数および受信関数 21
3.2.1 ブロッキング送信 21
3.2.2 メッセージ・データ 22
3.2.3 メッセージ・エンベロープ 24
3.2.4 ブロッキング受信 25
3.2.5 返却ステータス 27
3.3 データ型の一致とデータ変換 29
3.3.1 型一致規則 29
3.3.2 データ変換 33
3.4 通信モード 34
3.5 1 対 1 通信の意味 39
3.6 バッファのアロケーションと使用法 43
3.6.1 バッファモードのモデル実装 45
3.7 ノンブロッキング通信 46
3.7.1 通信オブジェクト 48
3.7.2 通信の起動 48
3.7.3 通信の完了 51
3.7.4 ノンブロッキング通信の意味論 56
3.7.5 多重完了 57
3.8 Probe および キャンセル 64
3.9 持続的な通信要求 70
3.10 送受信 75
3.11 ヌルプロセス 77
3.12 派生データ型 77
3.12.1 データ型コンストラクタ 79
3.12.2 アドレス関数と範囲関数 87
3.12.3 下限マーカと上限マーカ 89
3.12.4 記憶と解放 91

3.12.5 通信時の汎用データ型の利用 92
3.12.6 アドレスの正しい利用 96
3.12.7 例 97
3.13 パックとアンパック 107

4 集団通信 116
4.1 概論と概要 116
4.2 コミュニケータ引数 119
4.3 バリア同期 119
4.4 ブロードキャスト 120
4.4.1 MPI_BCAST の使用例 120
4.5 Gather 121
4.5.1 MPI_GATHER、 MPI_GATHERV の使用例 124
4.6 スキャッタ 132
4.6.1 MPI_SCATTER, MPI_SCATTER の使用例 135
4.7 全プロセスへのギャザー 138
4.7.1 MPI_ALLGATHER, MPI_ALLGATHERV の使用例 140
4.8 全対全スキャッタ／ギャザ 141
4.9 大域的なリダクション操作 143
4.9.1 Reduce 144
4.9.2 定義済みリデュース操作 145
4.9.3 MINLOC と MAXLOC 148
4.9.4 ユーザー定義操作 153
4.9.5 ユーザー定義リデュースの例 156
4.9.6 All-Reduce 157
4.10 Reduce-Scatter 通信 158
4.11 Scan 160
4.11.1 MPI_SCAN の使用例 160
4.12 正当性 162

5 グループ、 コンテキスト、 コミュニケータ 167
5.1 はじめに 167
5.1.1 ライブラリをサポートするのに必要な機能 167
5.1.2 MPI のライブラリ・サポート 168
5.2 基本概念 170
5.2.1 グループ 170

5.2.2 コンテキスト . . . 171
5.2.3 グループ内コミュニケータ . . . 172
5.2.4 定義済みグループ内コミュニケータ . . . 172
5.3 グループ管理 . . . 173
5.3.1 グループ参照関数 . . . 173
5.3.2 グループコンストラクタ . . . 175
5.3.3 グループデストラクタ . . . 180
5.4 コミュニケータ管理 . . . 181
5.4.1 コミュニケータ・アクセッサ . . . 181
5.4.2 コミュニケータコンストラクタ . . . 183
5.4.3 コミニュケータデストラクタ . . . 187
5.5 例題 . . . 188
5.5.1 一般的な慣例 #1 . . . 188
5.5.2 一般的な慣例 #2 . . . 189
5.5.3 (おおむね) 一般的な慣例 #3 . . . 190
5.5.4 例 #4 . . . 191
5.5.5 Library Example #1 . . . 192
5.5.6 ライブラリの例 #2 . . . 194
5.6 グループ間通信 . . . 197
5.6.1 グループ間コミュニケータのアクセッサ . . . 200
5.6.2 グループ間コミュニケータの操作 . . . 201
5.6.3 グループ間コミュニケータの使用例 . . . 204
5.7 キャッシング . . . 214
5.7.1 機能説明 . . . 214
5.7.2 属性の例 . . . 220
5.7.3 基本説明 . . . 223
5.7.4 実行モデル . . . 223

6 プロセス・トポロジー **225**

6.1 はじめに . . . 225
6.2 仮想トポロジー . . . 226
6.3 MPI への埋め込み . . . 227
6.4 トポロジー関数の概要 . . . 227
6.5 トポロジー・コンストラクタ . . . 228
6.5.1 カルテシアン・コンストラクタ . . . 228

6.5.2 カルテシアン支援関数: MPI_DIMS_CREATE 229
6.5.3 一般 (グラフ)・コンストラクタ 230
6.5.4 トポロジー問い合わせ関数 232
6.5.5 カルテシアン座標のシフト 237
6.5.6 カルテシアン構造の分割 239
6.5.7 低レベル・トポロジー関数 240
6.6 アプリケーション例 241

7 MPI 環境管理 244
7.1 実装情報 244
7.1.1 環境の問い合わせ 244
7.2 エラー処理 247
7.3 エラー・コードおよびクラス 251
7.4 時刻関数と同期 252
7.5 起動 253

8 プロファイリング・インターフェース 256
8.1 要求仕様 256
8.2 議論 257
8.3 設計の論理 257
8.3.1 プロファイリングの各種制御 258
8.4 例 259
8.4.1 プロファイラの実装 259
8.4.2 MPI ライブラリの実装 259
8.4.3 厄介な問題 261
8.5 複数レベルの横取り 262

Bibliography 263

A 言語からの呼び出し形式 266
A.1 はじめに 266
A.2 C 言語および Fortran 言語における定義済み定数 266
A.3 一対一通信のための C 言語呼び出し形式 271
A.4 集団通信のための C 言語呼び出し形式 276
A.5 グループ、コンテクスト、コミュニケータのための C 言語呼び出し形式 277
A.6 プロセストポロジーのための C 言語呼び出し形式 278
A.7 環境問い合わせのための C 言語呼び出し形式 279

A.8 プロファイリングのための C 言語呼び出し形式 280
A.9 一対一通信のための Fortran 言語呼び出し形式 280
A.10 集団通信のための Fortran 言語呼び出し形式 285
A.11 グループ、コンテクストその他のための Fortran 言語呼び出し形式 287
A.12 プロセストポロジーのための Fortran 言語呼び出し形式 290
A.13 環境問い合わせのための Fortran 言語呼び出し形式 291
A.14 プロファイリングのための Fortran 言語呼び出し形式 292

MPI Function Index **293**

原書謝辞

技術的内容の作成はいくつかの部会で行い、その成果を全体委員会において検証した。 Message Passing Interface (MPI) の作成期間中には、多くの人が責任ある地位に就いた。以下にそれらの人々の一覧を示す。

- Jack Dongarra, David Walker 主宰および会合議長
- Ewing Lusk, Bob Knighten 議事録
- Marc Snir, William Gropp, Ewing Lusk 一対一通信
- Al Geist, Marc Snir, Steve Otto 集団通信
- Steve Otto 編集
- Rolf Hempel プロセストポロジー
- Ewing Lusk 言語からの呼び出し形式
- William Gropp 環境管理
- James Cownie プロファイリング
- Tony Skjellum, Lyndon Clarke, Marc Snir, Richard Littlefield, Mark Sears グループ、コンテキスト、およびコミュニケータ
- Steven Huss-Lederman 初期実装用サブセット

上述した人以外で、 MPI の活動に活発に参加した人々の一部を以下に示す。

Ed Anderson	Robert Babb	Joe Baron	Eric Barszcz
Scott Berryman	Rob Bjornson	Nathan Doss	Anne Elster
Jim Feeney	Vince Fernando	Sam Fineberg	Jon Flower
Daniel Frye	Ian Glendinning	Adam Greenberg	Robert Harrison
Leslie Hart	Tom Haupt	Don Heller	Tom Henderson
Alex Ho	C.T. Howard Ho	Gary Howell	John Kapenga
James Kohl	Susan Krauss	Bob Leary	Arthur Maccabe
Peter Madams	Alan Mainwaring	Oliver McBryan	Phil McKinley
Charles Mosher	Dan Nessett	Peter Pacheco	Howard Palmer
Paul Pierce	Sanjay Ranka	Peter Rigsbee	Arch Robison
Erich Schikuta	Ambuj Singh	Alan Sussman	Robert Tomlinson
Robert G. Voigt	Dennis Weeks	Stephen Wheat	Steven Zenith

テネシー大学とオークリッジ国立研究所は、 anonymous FTP メールサーバによるドラフトの公開を行い、当文書の配布の手段を提供した。

MPI は非常に厳しい予算の下で運営された (実際、最初の会合の案内が出された時点では予算はまったくなかった)。 ARPA および NSF はさまざまな組織における研究を支援しており、それらの組織が米国の学術関係者の旅費に対する貢献を行った。何人かのヨーロッパからの参加者は ESPRIT からの支援を受けた。

日本語版謝辞

日本語版は、有志の参加者からなるプロジェクトにより行った。以下に参加者一覧 (あいうえお順) を示す。

翻訳担当者

荒木 宏之 (Hiroyuki Araki) NEC C&C 研究所 (第 3 章 12,13 節責任者)

大塚 寛 (Hiroshi Ohtsuka) 九州大学大学院数理学研究科 (第 6,7,8 章責任者)

小川 宏高 (Hirotaka Ogawa) 東大情報工学

勝田 宇一 (Uichi Katsuta) 日本電気 (株) コンピュータソフトウェア事業本部

隈本 覚 (Satoru Kumamoto) 北九州大学 経済学部

小西 弘一 (Koichi Konishi) NEC Research Institute (第 1,2 章責任者)

佐川 暢俊 (Nobutoshi Sagawa) 日立中央研究所

柴田 良一 (Ryoichi Shibata) 岐阜工業高等専門学校建築学科

清水 尚彦 (Naohiko Shimizu) 東海大学工学部通信工学科 (第 5 章責任者)

正代 隆義 (Takayoshi Shoudai) 九州大学 理学部

田村 正典 (Masanori Tamura) 日本電気 (株) コンピュータソフトウェア事業本部 (第 4 章責任者)

丹下 利雄 (Toshio Tange) NEC 情報システムズ技術システム事業部 第一技術部

中村 健太 (Kenta Nakamura) (株) 日立情報システムズ

中村 純 (Atsushi Nakamura) 山形大学教育学部

早坂 武 (Takeshi Hayasaka) 日本電気 (株) コンピュータソフトウェア事業本部

原田 敬 (Kei Harada) (株) 日立製作所ソフトウェア開発本部

日置 慎治 (Shinji Hioki) 広島大学理学部物理学科 (第 3 章 1-11 節責任者)

溝口 佳寛 (Yoshihiro Mizoguchi) 九州工業大学 情報工学部

森 雅生 (Maso Mori) 九州大学総合理工学研究科

山内 斉 (Hitoshi Yamauchi) 東北大学情報科学研究科中村研究室

レビュー参加者

金山 二郎 (Jiro Kanayama) 成蹊大学大学院工学研究科博士後期過程 2 年

中野 淳 (Jun Nakano) 日本 IBM

村田 英明 (Hideaki Murata) 三菱重工業株式会社 エレクトロニクス技術部

矢吹 洋一 (Youichi Yabuki) (株)SRA

NEC は全章に対する独自の訳を叩き台として無償で提供した。ドラフトの配布には、 PHASE(電総研) および NEC Research Institute の ftp サーバを利用した。

Chapter1

MPI の紹介

1.1 概要と目標

メッセージ通信は、ある種の、特に分散メモリを備える並列マシンで広く用いられているパラダイムである。そのバリエーションはいろいろあるが、メッセージによって通信をおこなうプロセスという基本概念は良く理解されている。この十年間で、重要なアプリケーションをこのパラダイムに当てはめることに関して、非常に大きな進歩が成し遂げられた。各ベンダはそれぞれに異なる独自のシステムを実現した。さらに最近では、いくつかのシステムによって、メッセージ通信システムが効率よくかつポータブルに実現できるということが実証された。したがって、今や、広範囲のユーザーにとって役に立ち、なおかつ広範囲のコンピュータ上で効率的に実現できるライブラリルーチンの中核について、その構文と意味の定義を試みるに適当な時期に来ていると言えるだろう。

MPI の設計に際しては、いくつかの既存のメッセージ通信システムが持つもっとも魅力的な機能の数々を採りいれるよう努めた。既存のシステムの中から 1 つを選択し、それを標準として採用するというやり方は採らなかった。 MPI は、 IBM の T. J. Watson Research Center [1, 2] での研究や、 Intel の NX/2 [23]、 Express [22]、 nCUBE の Vertex [21]、 p4 [7, 6]、 PARMACS [5, 8] から多大な影響を受けてきた。他に、 Zipcode [24, 25]、 Chimp [14, 15]、 PVM [4, 11]、 Chameleon [19]、 PICL [18] などからも重要な貢献があった。

MPI の標準化の事業には、主に米国、ヨーロッパの 40 の組織から約 60 人が参加した。並列コンピュータの主要なベンダのほとんどが MPI に参加し、大学、国立研究所、そして企業からの研究者もこれに加わった。標準化の作業が始まったのは、 「分散メモリ環境におけるメッセージ通信のための標準に関するワークショップ [29] (Workshop on Standards for Message Passing in a Distributed Memory Environment)」においてである。このワークショップは 1992 年 4 月 29 日 -30 日、バージニア州ウィリアムズバーグで「並列コンピューティング研究センタ」 (Center for Research on Parallel Computing) の主催により開催された。このワーク

ショップにおいて、標準的なメッセージ通信インタフェースの中心となる基本機能が討議され、標準化のプロセスを継続するため、ワーキンググループが設立された。

1992 年 11 月に、 Dongarra、 Hempel、 Hey, Walker により、 MPI1 として知られる予備的な草案が提示され、修正版が 1993 年 2 月に完成した [12]。 MPI1 には、ウィリアムスバーグのワークショップにおいて、メッセージ通信標準に必要と指摘された主要な機能が具体化されていた。 MPI1 はもともと、論議を促進するためのたたき台だったので、主に 1 対 1 通信に焦点を当てている。 MPI1 は、標準化における多くの重要な課題を明らかにしたが、これには集団通信ルーチンは含まれておらず、またマルチスレッド環境には対応していなかった。

1992 年 11 月に、 MPI ワーキンググループのミーティングがミネアポリスで開かれ、そこで標準化をより形式に則ったプロセスにし、 High Performance Fortran Forum の手順と組織構成をほぼそのまま採用することが決定された。標準の主要な構成分野ごとに小委員会が作られ、それぞれに電子メール討議サービスが設立された。さらに、 1993 年秋までに MPI 標準の草案を作成するという目標が設定された。この目標を達成するため、 MPI ワーキンググループは、 1993 年の最初の 9 カ月間、 6 週間ごとに 2 日間の会合を開き、 1993 年 11 月の Supercomputing 93 会議で MPI 標準の草案を発表するに至った。これらの会合と電子メールによる議論が MPI フォーラムを構成したものであり、その会員資格はハイパフォーマンスコンピューティングに関わるすべての人に対して常に開放されていた。

メッセージ通信の標準を確立する主な利点は、ポータビリティと使いやすさにある。分散メモリ通信環境の中でも特に、高レベルのルーチンやアブストラクションが低レベルのメッセージ通信ルーチンの上に構築されているものでは、標準化の利点は明白である。さらに、メッセージ通信標準の定義、たとえばここに提示するようなものは、ベンダに対して、明確に定義された、基礎となる一群のルーチンを提供する。このようなルーチン群があれば、ベンダは、それを効率よく実現したり、あるいは場合によってはそれ専用のハードウェアを用意したりすることができ、それによりスケーラビリティを高めることができる。

すべての目標のリストを以下に示す。

- アプリケーションプログラムから呼ぶためのインタフェースを設計する（必ずしもコンパイラや、システムの一部を実現するライブラリだけが使うためのものではないということ）。
- 効率のよい通信を可能にすること。メモリ間コピーを避け、計算と通信を並行して進めることができ、通信コプロセッサがあれば処理の一部をそれに任せるようなもの。
- 異機種環境でも使用できる処理系に備える。
- C 言語や Fortran77 言語のための使い易い呼び出し形式を可能にする。
- 信頼できる通信インタフェースを想定する。ユーザーは通信障害に対処する必要がない。そのような障害は下位の通信サブシステムが処理する。

- PVM、 NX、 Express、 p4 などの既存のインタフェースとそれほど違わないインタフェースで、なおかつより融通の効く拡張機能を可能にするインタフェースを定義する。
- 多くのベンダのプラットフォーム上に実現でき、その際、下位の通信およびシステムソフトウェアに大きな変更を加えずに済むインタフェースを定義する。
- このインターフェースの意味は言語に依存するべきでない。
- このインタフェースは、マルチスレッド環境への対応が可能なように設計するべきである。

1.2 この標準を利用する人々

この標準が意図する利用者は、ポータブルなメッセージ通信プログラムを Fortran77 言語と C 言語で記述しようとするすべての人である。この中には、個々のアプリケーションのプログラマや、並列マシンで動作するよう設計されたソフトウェアの開発者、環境やツールの作成者を含む。この標準が、これらの広範囲に渡る人々にとって魅力的なものであるためには、初歩のユーザーには簡単で使いやすいインタフェースを提供する一方で、先進的マシンに備わる高性能のメッセージ通信操作を意味的に妨げないものでなければならない。

1.3 この標準の実現対象となるプラットフォーム

メッセージ通信パラダイムの魅力は、少なくとも部分的には、その広いポータビリティから生じている。この方法で表現されたプログラムは、分散メモリのマルチプロセッサ、ワークステーションのネットワーク、そしてこれらの組み合わせの上でも動作しうる。さらに、共有メモリによる実現も可能である。このパラダイムは、共有メモリと分散メモリの観点を組み合わせたアーキテクチャが出現しても、またネットワークが高速化しても時代遅れにはならないだろう。したがって、この標準を実装することが可能であり有用であるマシンの範囲は非常に広いはずであり、そこには通信ネットワークによって接続された他の（並列機か逐次機かを問わない）マシンの集まりで構成されるような「マシン群」まで含まれるだろう。

このインタフェースは、まったく一般的な MIMD プログラムでの使用にも、より制限された SPMD のスタイルで書かれたプログラムでの使用にも同じように適している。このインタフェースでは、スレッドに対する明確なサポートはないが、スレッドの使用を妨げることがないように設計されている。このバージョンの MPI では、タスクの動的な生成はサポートされていない。

MPI には、専用のプロセッサ間通信ハードウェアを備えるスケーラブルな並列コンピュータでの性能を高めることを狙いとした多くの機能がある。したがって、このようなマシンには高性能な専用 MPI 処理系が提供されることが期待される。また一方では、 Unix プロセッサ間通信プロトコルを用いる MPI 処理系は、ワークステーションクラスタやワークステーションの異

機種間ネットワークの間でのポータビリティを提供するだろう。ベンダ独自の専用 MPI 処理系いくつかと、パブリックドメインのポータブルな MPI 処理系一つの開発が、本文の記述時点において進行している [17, 13]。

1.4 標準に含まれるもの

標準は以下の事項を含む。

- 1 対 1 通信
- 集団操作
- プロセスグループ
- 通信コンテキスト
- プロセストポロジー
- Fortran77 言語と C 言語の呼び出し形式
- 環境管理と問い合わせ
- プロファイリングインタフェース

1.5 標準に含まれないもの

標準は以下の事項を指定しない。

- 明示的な共有メモリ操作
- 現在標準となっているもの以上のオペレーティングシステムのサポートを必要とする操作。たとえば割り込み駆動の受信、リモート実行、アクティブメッセージなど。
- プログラム構築ツール
- デバッグ機能
- スレッドに対する明示的サポート
- タスク管理のサポート
- 入出力機能

一応考慮された上で、この標準には含まれなかった機能が多数ある。これはさまざまな理由によるが、そのうちの１つは、標準の仕上げに際してつきまとう時間的制約によるものである。含まれていない機能は、特定の実装により拡張機能としていつでも提供され得る。おそらく MPI の将来のバージョンでは、これらの事項のうちいくつかについて扱うことになるであろう。

1.6 この文書の構成

以下は、この文書の残りの章についてのリストで、それぞれに簡単な説明を付けた。

- 第 2章、MPI の用語と規則。この章では、この MPI 文書全体で使用される表記上の用語と規則について説明する。
- 第 3章、１対１通信。この章では、MPI の基本的な二者間通信に関するサブセットについて定義する。基本操作である send と receive はここで扱う。加えて、基本的な通信を強力で効率のよいものにするよう設計された、多くの関連する関数についても述べる。
- 第 4章、集団通信。この章では、プロセスグループ内での集団通信動作を定義する。良く知られている例としては、あるプロセスのグループ（必ずしも全プロセスではない）内でのバリアおよびブロードキャストがある。
- 第 5章、グループ、コンテキスト、コミュニケータ。この章では、プロセスのグループを形成して、それらを操作する方法、固有の通信コンテキストを取得する方法、その 2 つを結合してコミュニケータを作る方法が説明されている。
- 第 6章、プロセストポロジー。この章では、プロセスグループ（線形順序集合）を多次元グリッドのような、より複雑なトポロジー構造に対応づける時に役に立つよう用意された一群のユーティリティ関数について説明する。
- 第 7章、MPI 環境管理。この章では、プログラマが現在の MPI 環境について管理、問い合わせをおこなう方法について説明する。これらをおこなう関数は、正しい頑健なプログラムの記述に必要であり、とりわけ、非常にポータブルなメッセージ通信プログラムを構築するために重要である。
- 第 8章、プロファイリングインタフェース。この章では、すべての MPI 処理系がサポートしなければならない、簡単な、名前のシフトの規則について説明する。このシフトの狙いの一つは、性能のプロファイリングのための呼び出しを、MPI のソースコードに手を加える必要なしに MPI に入れることができるようにすることである。名前シフトは、単なるインタフェースに過ぎず、実際にプロファイリングで何をどうするかということについては何も指示しない。実際、名前シフトを他の目的で有効に使用することも可能である。

- 付録 A、言語からの呼び出し形式。ここでは、 Fortran77 言語と C 言語における具体的な構文を、すべての MPI 関数、定数、型について示す。
- MPI 関数の索引、 これは簡単な索引で、それぞれの MPI 関数の正確な定義と共に C 言語および Fortran 言語の呼び出し形式が併記されている場所を示す。

Chapter2

MPI の用語と規則

この章では、 MPI 文書全体で使用される表記上の用語と規則、行われた選択のいくつか、およびそれらの選択の背後にある根拠について説明する。

2.1　文書の表記

根拠 インタフェースの仕様の中でおこなわれた設計上の選択の根拠は、この文書全体を通して、この形式で提示される。読者によっては、これらの節はとばすことにしようと思うかもしれないし、インタフェースのデザインに興味のある読者は念入りに読もうと思うかもしれない。（根拠の終わり）

ユーザへのアドバイス ユーザーに訴えたいことや使用法の説明については、この文書全体を通して、この形式で提示される。読者によっては、これらの節はとばすことにしようと思うかもしれないし、 MPI のプログラミングに興味のある読者は念入りに読もうと思うかもしれない。（ユーザへのアドバイスの終わり）

実装者へのアドバイス　主に実装者に対する注釈となることについては、この文書全体を通して、この形式で提示される。読者によっては、これらの節はとばすことにしようと思うかもしれないし、 MPI の実装に興味のある読者は念入りに読もうと思うかもしれない。（実装者へのアドバイスの終わり）

2.2　手続きの仕様

MPI 手続きの仕様は、言語に依存しない表記を使用して記述する。手続き呼び出しの引数には、入力、 出力、 入出力の区別を示す。これらの意味は以下の通りである。

- 入力と示した引数は、呼び出しにより使用されるが、更新されることはない。

- 出力と示した引数は、呼び出しにより更新されるかもしれない。
- 入出力と示した引数は、呼び出しにより使用、更新の両方が行われる。

特別な場合が一つある。引数が不透明オブジェクトのハンドル（これらの用語については 2.4.1 節で定義）であって、そのオブジェクトがその手続き呼び出しで更新される場合、その引数は 出力と示す。ハンドル自身は修正されないにも関わらずこのように示す。つまり、 出力属性は、ハンドルの参照するものが更新されるということを示すために使うことがある。

MPI の定義は、 入出力引数の使用をできる限り避けている。このような使い方は、特にスカラー引数の場合、間違いの元になるからである。

MPI の関数では、ある引数が一つのプロセスでは 入力として使用され、別のプロセスでは出力として使用されることがよくある。このような引数は、文法的には 入出力引数であり、 入出力と示す。しかし、意味的には、1 つの呼び出しで入力と出力の両方のために使用されることはない。

また、ある引数の値が一群のプロセスのうちの幾つかにとってのみ必要な場合もよくある。あるプロセスにおいてある引数が意味をもたない場合は、任意の値を引数として渡して構わない。

特に指定しない限り、 出力または 入出力の引数について、同じ MPI 手続きに渡す他の引数を用いて別名をつけてはならない。以下に示すのは C 言語において引数に別名をつけた例である。次のような C 言語の手続きを定義すると、

```
void copyIntBuffer( int *pin, int *pout, int len )
{   int i;
    for (i=0; i<len; ++i) *pout++ = *pin++;
}
```

この手続きの以下のコードにおける呼び出しは、別名つきの引数を持つ。

```
int a[10];
copyIntBuffer ( a, a+3, 7);
```

C 言語はこれを許可しているが、 MPI 手続きのこのような使い方は特に指定しない限り、禁止する。なお Fortran 言語は、引数に別名をつけることを禁止している。

すべての MPI 関数の仕様は、最初に言語に依存しない表記によって示す。そのすぐ下に、ANSI C 言語版の関数を、その下に Fortran77 言語における同じ関数を表示する。

2.3 意味に関する用語

MPI 手続きについて説明する際、以下に示す、意味に関する用語を使用する。最初の 2 つは通常、通信操作に対して用いる。

ノンブロッキング 操作が完了する前で、なおかつ呼び出しにおいて指定された (バッファなどの) リソースをユーザーが再使用できるようになる前に、手続きから戻ることがありうる場合。

ブロッキング 手続きから戻り次第、呼び出しにおいて指定されたリソースをユーザーが再使用してよい場合。

ローカル 手続きの完了がそれを実行しているプロセスのみに依存する場合。このような操作は、別のユーザープロセスとの通信を必要としない。

ノンローカル 操作を完了するために、別のプロセスでのなんらかの MPI 手続きの実行が必要かもしれない場合。このような操作は、別のユーザープロセスとの通信を必要とするかもしれれない。

集団的 あるプロセスグループ中のすべてのプロセスが、その手続きを起動しなければならない場合。

2.4 データ型

2.4.1 不透明オブジェクト

MPI は、システムメモリを管理する。システムメモリとは、ここでは、メッセージのバッファリングや、さまざまな MPI オブジェクト (グループ、コミュニケータ、データ型など) の内部表現の格納に使われるメモリを指す。このメモリにはユーザーが直接アクセスすることができず、したがってここに格納されるオブジェクトは不透明である。つまり、そのサイズと形状はユーザーには見えない。不透明オブジェクトは、ユーザー領域にあるハンドルを通してアクセスする。不透明オブジェクトを操作する MPI 手続きには、そのオブジェクトをアクセスするためのハンドル引数を渡す。ハンドルは、 MPI 呼び出しでオブジェクトのアクセスに使うだけでなく、そのまま代入したり比較したりすることもできる。

Fortran 言語の場合、すべてのハンドルは整数型を持つ。 C 言語の場合、オブジェクトの種類ごとに、異なるハンドル型が定義される。これらは、代入および等価演算子をサポートする型でなければならない。

Fortran 言語の場合、ハンドルは、システムテーブル中の不透明オブジェクトのテーブルに対する添字として実現できる。 C 言語の場合は、同様の添字にするか、あるいはオブジェクトへのポインタにすることができる。他にも変わったやり方はあるだろう。

不透明オブジェクトの割り当てと解放は各オブジェクト型に固有の呼び出しによって行う。これらは各オブジェクトについて述べた節に挙がっている。これらの呼び出しは、対応する型のハンドル引数を受け取る。割り当て呼び出しの場合、この引数はオブジェクトへの有効な参照値を返す 出力引数である。解放のための呼び出しでは、この引数はヌルハンドルを値として戻る入出力引数である。 MPI はヌルハンドル定数を各オブジェクト型ごとに一つ設ける。この定数と比較することにより、ハンドルが有効かどうかを調べることができる。

解放処理の呼び出しは、ハンドルを無効にし、オブジェクトには、解放するべきものであることを示す印をつける。そのオブジェクトは、この呼び出しの後、ユーザーからはアクセスできなくなる。しかしながら MPI はそのオブジェクトをすぐに解放する必要はない。このオブジェクトに関わる（その解放処理の時点で）保留中の操作は、正常に終了し、オブジェクトはその後に解放される。

MPI 呼び出しは、ハンドルの値を変えることはない。例外として、オブジェクトの割り当てや解放をおこなう呼び出しの場合、および 3.12.4 節の MPI_TYPE_COMMIT 呼び出しの場合がある。

ヌルハンドル引数を MPI 呼び出しにおける 入力引数とするのは、関数を定義する文中で例外が明示されていない限り、間違った使い方である。そのような例外は、 Wait および Test 呼び出しにおけるリクエストオブジェクトのハンドル（3.7.3 節および 3.7.5 節）のために設けてある。それ以外でヌルハンドルを渡してよいのは、新しいオブジェクトを割り当て、そのオブジェクトへの参照値をハンドルとして返す関数の場合に限られる。

不透明オブジェクトとそのハンドルは、オブジェクトを生成したプロセスでのみ意味を持ち、別のプロセスに転送することはできない。

MPI では、いくつかのあらかじめ定義された不透明オブジェクト、およびこれらのオブジェクトに対するあらかじめ定義された静的ハンドルが用意されている。このようなオブジェクトは消去してはならない。

根拠 この設計では、 MPI データ構造のために使う内部表現を隠してあるので、 C 言語でも Fortran 言語でも同じような呼び出しの形式を使うことができる。またこれらの言語における型に関する規則との矛盾も回避されており、将来の機能拡張も簡単に行えるようになっている。ここで使用されている不透明オブジェクトのメカニズムは、 POSIX の Fortran 言語用呼び出し形式の標準にゆるやかに沿ったものになっている。

ユーザー領域でハンドル、システム領域でオブジェクトという具合に明示的に区別すると、メモリ領域を回収する解放呼び出しを、ユーザープログラムの適切な場所で行えるようになる。仮に、不透明オブジェクトがユーザー領域にあったならば、そのオブジェクトを必要とする保留中の動作が完了する前に、そのユーザ領域が有効である範囲の外に出ないよう非常に気を付けなければならないだろう。ここに示した設計では、オブジェクトに解放

すべきものとして印をつけることができ、ユーザープログラムが上述の範囲外に出ることも可能で、それでもオブジェクト自体は、保留中の動作が完了するまで解放されることはない。

ハンドルが代入や比較をサポートしなければならないと定めたのは、そのような処理がよく行われるからである。これにより、可能な実現方法の幅は狭くなった。これに替えて、ハンドルが任意の不透明な型であることを許すという選択もありえた。この場合、代入と比較を行うためのルーチンを導入しなければならなくなり、より複雑になるので、この規定は採用されなかった。（根拠の終わり）

ユーザへのアドバイス 参照先のない参照値が間違ってできることがある。これは、ユーザーがあるハンドルに別のハンドルの値を代入し、そのあとこれらのハンドルに対応するオブジェクトを解放することによりできる。逆に、あるハンドル変数を、それに対応するオブジェクトを解放する前に、解放してしまうと、そのオブジェクトにはアクセスできなくなる（これは、たとえばハンドルが、あるサブルーチン内のローカル変数であって、関連するオブジェクトが解放される前にサブルーチンが終了する場合に起きる）。不透明オブジェクトへの参照値の追加や削除は、そのようなオブジェクトの割り当て、解放をおこなう呼び出しに伴う場合を除き、ユーザーの責任で避けなければならない。（ユーザへのアドバイスの終わり）

実装者へのアドバイス　不透明オブジェクトの定義が意図するところでは、各不透明オブジェクトは一つ一つ別のオブジェクトであり、このようなオブジェクトを割り当てる呼び出しは、そのオブジェクトに必要なすべての情報をコピーする。実装は、必要以上のコピーを避けるため、コピーを参照で置き換えてもよい。たとえば、ユーザー定義データ型には、その構成要素のコピーを持たせる代わりに、その構成要素への参照値を持たせてもよい。MPI_COMM_GROUP の呼び出しは、コミュニケータに対応するグループのコピーの代わりに、そのグループに対する参照値を返してもよい。そのような場合、実装は、参照カウントを管理する必要があり、またオブジェクトの割り当てと解放を、見かけ上オブジェクトがコピーされているように見えるように行わなければならない。（実装者へのアドバイスの終わり）

2.4.2 配列引数

MPI 呼び出しで、不透明オブジェクトの配列またはハンドルの配列の引数が必要になる場合もある。ハンドルの配列は普通の配列であって、そのエントリは同じ型のオブジェクトに対するハンドルであり、その配列の中の連続した位置に並んでいる。このような配列を使う場合は常に、有効なエントリの数を示すため長さを示す len 引数を添える必要がある（この数値が他から求め

られる場合は除く）。有効なエントリは配列の前方寄りにあり、 len はその個数を示すが、これは、配列全体のサイズと同じでなくてもよい。他の配列引数でも同じアプローチがとられる。

2.4.3 状態型

MPI 手続きでは、さまざまな場所で「状態型」の引数を使用する。このようなデータ型の値はすべて名前で識別され、それについての操作は定義されていない。たとえば MPI_ERRHANDLER_SET ルーチンは状態型の引数を一つ取り、これは MPI_ERRORS_ARE_FATAL、 MPI_ERRORS_RETURN などの値を持つ

2.4.4 名前付き定数

MPI 手続きは時に基本的な型の引数の特別な値に、特別な意味を割り当てている。たとえばタグは 1 対 1 通信操作の整数値を取る引数であるが、これには特別なワイルドカード値として MPI_ANY_TAG がある。このような引数は、ある値域を通常値としてとる。この値域は対応する基本データ型の値域の一部分である。特殊な値（たとえば MPI_ANY_TAG）は、この通常値域の外になる。通常値域は、環境問い合わせ関数を使用して問い合わせすることができる（第 7節）。

MPI はまたあらかじめ定義された名前付き定数としてハンドルも提供する。たとえば、 MPI_COMM_WORLD は、起動時に存在していて互いに通信が可能な全プロセスを表すオブジェクトへのハンドルである。

すべての名前付き定数は、 Fortran 言語における MPI_BOTTOM を除き、初期化式および代入の中で使用することができる。これらの定数は実行時に値が変わることはない。定数ハンドルによってアクセスされる不透明オブジェクトは MPI の初期化（MPI_INIT() 呼び出し) からの MPI 終了処理（MPI_FINALIZE() 呼び出し）までの間存在し、その間、値が変わることはない。

2.4.5 選択型

MPI 関数は時に「選択 (または共用）」データ型の引数を使用する。いくつかの異なる呼び出しが、同じルーチンに対するものであるにも関わらず、参照渡しによって、それぞれ違う型の実引数を渡すことがある。このような引数を与えるためのメカニズムは、言語によって異なる。 Fortran 言語の場合、本文書では <type> を使用して選択型変数を表現し、 C 言語の場合は、 (void *) を使用する。

2.4.6 アドレス型

いくつかの MPI 手続きは、呼び出し側のプログラム内の絶対アドレスを示すアドレス型引数を使用する。このような引数のデータ型は整数型で、実行環境における任意の有効なアドレスを保持するのに必要なサイズを持っている。

2.5 言語の呼び出し形式

この節では、 MPI における言語の呼び出し形式一般についての規則と、 Fortran 77 言語と ANSI C 言語についての個別の規則を定義する。ここではさまざまなオブジェクトの表現、およびこの標準の表現に使用する命名規則を定義する。実際の呼び出し方法については別の箇所で定義する。

Fortran 90 言語用および C++ 言語用の実装では、それぞれ Fortran 77 言語と ANSI C 言語の呼び出し形式を使用するものと想定している。他の呼び出し形式を Fortran 90 言語と C++ 言語について定義するには時期尚早と判断したものの、現在の呼び出し形式をこのように設計したのは、将来採用し得るよりよい呼び出し形式を定める試みを促すためであって、そのような試みをやめさせるためではない。

PARAMETER(引数) という言葉は、 Fortran 言語ではキーワードなので、ここでは argument (引数) という言葉を使ってサブルーチンに渡す引数を表現する。 C 言語では普通これらは parameter と呼ばれるが、 C 言語のプログラマは argument という言葉 (C 言語ではこれには特定の意味はない) を理解できるであろうから、それを当てにして、 Fortran 言語プログラマを不必要に混乱させるのは避けることにした。

言語呼び出し形式に関する重要な問題で、この標準では扱っていないものがいくつかある。この標準では、複数の言語間でのメッセージ通信の操作の可否については論じていない。多くの実装がそのような機能を持つこと、およびそのような機能がその実装の品質の証しになることが強く期待されている。

2.5.1 Fortran 77 言語の呼び出し形式に関する事項

すべての MPI 名は `MPI_` という接頭辞を持ち、またすべての文字は大文字である。プログラムは、この接頭辞 `MPI_` で始まる変数や関数を宣言してはならない。この制約は名前の衝突を避けるために必要である。

すべての MPI Fortran 言語のサブルーチンには、最後の引数に返却コードを付ける。いくつかの MPI 操作は関数であり、これらには返却コード引数を持たない。成功した場合の返却コード値は MPI_SUCCESS になる。その他の場合のエラーコードは実装によって異なる。第 7 章を参照すること。

Fortran 言語では、ハンドルは整数型で表現される。二値変数は、論理型を持つ。

配列引数の添字は 1 から始まる。

特に明示されていない限り、 MPI F77 呼び出し形式は、 ANSI 標準の Fortran77 言語と整合性を持つ。いくつかの点ではこの標準は ANSI Fortran77 言語標準から外れている。これらの例外は Fortran 言語コミュニティにおける普通の慣習に沿ったものである。以下にそれらを挙げる。

```
double precision a
integer b
...
call MPI_send(a,...)
call MPI_send(b,...)
```

図 2.1: 仮引数と実引数が一致しないルーチンの呼び出し例

- MPI の識別子の表現文字としては、6 文字ではなく、30 文字を上限とする。
- MPI の識別子では、最初の文字以降に下線を含んでいてよい。
- 選択型引数を持つ MPI サブルーチンは、異なる型の引数で呼び出すことができる。図 2.1に例を示す。これは、Fortran 言語標準が述べるところには違反しているが、このような違反はごく普通に行われている。さもなければ、それぞれのデータ型に対する個別の MPI_SEND を設けることになったであろう。
- 必須事項ではないが、名前付き MPI 定数（FORTRAN の PARAMETER）を mpif.h というインクルードファイルの中に入れて提供することを強く奨める。インクルードファイルをサポートしていないシステムでは、実装は名前付き定数の値を明らかにするべきである。
- ベンダーは、ユーザー定義型をサポートする Fortran 言語システムでは、なるべく mpif.h ファイルの中に型宣言を提供するよう努めてほしい。できれば、MPI_ADDRESS 型は定義するべきである。これはその実行環境においてアドレスを格納するのに必要なサイズを持った整数型である。型の定義がサポートされていないシステムの場合、アドレスを表現するための正しい種類の整数型を使用するのはユーザーの責任になる（つまり 32 ビットマシンの場合 4 バイト長整数型 (INTEGER*4)、64 ビットマシンの場合 8 バイト長整数型 (INTEGER*8)、など）。

すべての MPI の名前付き定数は、Fortran 言語のパラメータ属性で宣言された実体を使用できる、あらゆる場面で、使用することができる。この規則には、1 つだけ例外があり、MPI 定数 MPI_BOTTOM （3.12.2 節）はバッファ引数としてのみ使用することができる。

2.5.2 C 言語呼び出し形式に関する事項

ANSI C 言語宣言形式を使用する。すべての MPI 名は MPI_ という接頭辞を持ち、あらかじめ定義された定数は全て大文字からなる名前を持ち。あらかじめ定義された型や関数では、接頭辞の

あとの文字が 1 文字だけ大文字になる。名前が MPI_ という接頭辞で始まる変数や関数を、プログラムで宣言してはならない。この制約は名前の衝突を避けるために必要である。

名前付き定数の定義、関数プロトタイプ、型定義は、 sf mpi.h という名のインクルードファイルの中にいれて提供しなければならない。

ほとんどすべての C 言語関数はエラーコードを返す。成功した場合の返却コードは MPI_SUCCESS になるが、エラーが起こった場合の返却コードは、実装によって異なる。いくつかの C 言語関数は値を返さず[1]、そのためマクロとして実現することができる。

不透明オブジェクトの各種類に対応するハンドルに対しては、それぞれ別の型宣言が用意される。ポインタ型または整数型が使用される。

配列引数の添字は 0 から始まる。

論理フラグは整数で、値が 0 の場合「偽」、 0 以外の場合「真」を意味する。

選択型引数は、 void* 型のポインタである。

アドレス引数は、 MPI で定義する型 MPI_Aint である。この型は、実装が対象とするアーキテクチャ上の任意の有効なアドレスを保持するのに必要なサイズを持つ int として定義する。

すべての名前付き MPI 定数は、 C 言語定数のように初期化の式または代入で使用することができる。

2.6 プロセス

MPI プログラムは、自律的なプロセスで構成される。これらのプロセスは独自のコードを MIMD スタイルで実行する。各プロセスによって実行されるコードは、同一である必要はない。これらのプロセスは、 MPI の通信プリミティブを呼び出すことにより通信する。典型的には、各プロセスは独自のアドレス空間で実行されるが、共有メモリに基づく MPI の実装もありえる。この文書で規定する、並列プログラムの動作は、 MPI 呼び出しのみが通信に使用されることを前提としている。 MPI プログラムと他の通信手段（たとえば共有メモリ）との相互作用については規定していない。

MPI は、それぞれのプロセスの実行モデルについては規定しない。プロセスは逐次的かもしれないし、マルチスレッドになっていて、中で複数のスレッドが並行して動作しているかもしれない。 MPI を「スレッドセーフな (マルチスレッド環境に対応した)」ものにするため、十分な注意が払われており、明示されていない状態への依存を避けることによりこれを達成している。

MPI とスレッド群の間の相互作用は、以下のようになっていることが望ましい。並行して

[1] 訳注: この、「値を返さない C の関数」が、何を指しているのか分かりません。実際にある例外的な関数は MPI_Wtime と MPI_Wtick ですが、これらはエラーコードの代わりに double の値を返します。「返却コードを返さない」の間違いかもしれません。

実行するスレッド群はすべて MPI 呼び出しを行うことが許され、呼び出しはリエントラントであること。ブロッキング MPI 呼び出しは、これを呼び出したスレッドのみをブロックし、これにより別のスレッドのスケジューリングを可能にすること。

MPI は、 MPI を用いる計算のためにプロセスを最初に割り当てること、およびそれらのプロセスと物理プロセッサの対応づけを指定することのためのメカニズムを提供しない。このような処理をロード時や実行時に行うためのメカニズムはベンダが提供すると期待されている。このようなメカニズムは以下の事項の指定を可能にする。すなわち、必要なプロセスの最初の個数、最初のプロセスがそれぞれ実行するコード、およびプロセッサに対するプロセスの割り当てである。また現提案では、プログラム実行時におけるプロセスの動的な作成や削除は提供されていない（プロセスの総数が固定されている）。ただし、そのような拡張をおこなっても、矛盾が生じないことを意図してはいる。

最後に、プロセスは常に一グループ内での相対的なランク、つまり 0 ... グループサイズ - 1 までの範囲の連続する整数によって識別する。

2.7 エラー処理

MPI は、信頼性のあるメッセージ転送をユーザーに提供する。送られるメッセージは、常に正しく受け取られるため、ユーザーが転送エラー、タイムアウトその他のエラー状態についてチェックをおこなう必要はない。いいかえれば、 MPI には、通信システム内で発生するエラーを扱うメカニズムを提供しない。 MPI の実装を信頼性のない下位のメカニズムの上に構築する場合、ユーザをその信頼性のなさから隔離したり、回復不能なエラーを障害として通知するのは、その MPI サブシステムの実装者の仕事である。可能な限り、このような障害は関連する通信呼び出しでエラーとして通知される。同様に MPI それ自身にはプロセッサの障害を処理するメカニズムはない。 7.2節で記述しているエラー処理機能は、回復不能なエラーの範囲を制限し、またアプリケーション・レベルでのエラーからの回復処理を設計するために使用できる。

とはいえ、 MPI プログラムが間違っている可能性は当然ある。プログラムエラーは、 MPI 呼び出しの引数が正しくない（送信操作のときに送信先が存在しない、受信操作のときにバッファが小さすぎるなど）ときに発生しうる。この種のエラーは、どのような実装においても起きるだろう。また、リソースエラーは、使用可能なシステム・リソース（保留中のメッセージ数、システム・バッファ数など）の量をプログラムが超えたときに発生しうる。この種のエラーが起きるかどうかは、システム内の使用可能なリソースの量およびリソース割当てに使うメカニズムに依存する。品質の高い実装は、重要なリソースについては十分ゆとりのある上限を提供して、これに代表される移植性の問題を軽減しようとするだろう。

ほとんど全ての MPI 呼び出しでは、操作が正しく終了したことを示すコードを返す。エラーが呼び出しの最中に発生した場合、可能な限り、 MPI 呼び出しはエラーコードを返す。ある状

況下、すなわち MPI 関数が複数の別々の操作を完了する可能性があって、したがっていくつかの独立したエラーを発生し得る場合、 MPI 関数は複数のエラー・コードを返すことがありうる。特に指定しなければ、 MPI ライブラリの実行中にエラーが見つかった場合、その並列計算は異常終了する。しかし MPI は、ユーザーがこのデフォルト時の振舞いを変更して、回復可能なエラーを処理するためのメカニズムを提供する。ユーザーは、エラーはすべて致命的でないと指定して、 MPI 呼び出しが返すエラーコードをユーザー自身で処理することができる。またユーザーはユーザー独自のエラー処理ルーチンを用意することもできる。このルーチンは、 MPI 呼び出しが異常な状態で戻ってくる場合には常に起動される。 MPI エラー処理機能については、 7.2節で説明している。

エラーが発生したとき、 MPI 呼び出しが意味のあるエラーコードを返す機能は、いくつかの要因によって制限される。 MPI はある種のエラーは検出できないことがある。また他の種のエラーについては、通常の実行モードで検出するにはコストがかかりすぎることもある。最後に、ある種のエラーでは「天変地異」的に被害が大きいため、 MPI が制御を整合のとれた状態で呼び出し側に返すことができないこともある。

その他の微妙な問題として、非同期通信の性質によって発生するものがある。つまり、 MPI 呼び出しは、その呼び出しが戻った後にも非同期的に継続する処理を起動することがある。このため、その操作は正しく終了したことを示すコードを返すが、そのあとでエラー例外を発生する、という可能性がある。同じ操作に関連する呼び出し（たとえば、非同期操作が終了したことを確認する呼び出し）がそのあとにある場合、この呼び出しに対応するエラー引数をエラーの性質を示すために使用する。場合によっては、操作に関連するすべての呼び出しが終了したあとにエラーが発生することがあり、そのためエラーの性質を示すために使えるエラー値がないこともある（例えば、レディモードによる送信でのエラー）。このようなエラーは、致命的なものとして扱わなければならない。その状態からユーザーが復旧するための情報を返すことができないからである。

この文書では、 MPI 呼び出しでエラーが発生したあとの計算処理の状態を規定していない。望ましい操作は、しかるべきエラーコードが返され、エラーの影響が可能な限り狭い範囲に留められることである。例えば、エラーを起こした受信呼び出しが、メッセージ受信用に指定された領域を越える、受信側のメモリのいかなる部分をも書き換えることがない、という動作は非常に望ましい。

実装時には、この文書での仕様の範囲を超えて、 MPI 呼び出しのエラー定義を意味のある方法でサポートしてもよい。たとえば MPI では、対応する送信操作と受信操作の間で厳密な型一致規則が規定されている。つまり浮動小数点変数を送り、これを整数として受け取るのは間違いである。実装時には、これらの型一致規則を越えて、このような場合に自動型変換を行ってもよい。このような、規則に則らない操作に対しては、警告を出すのも役に立つであろう。

2.8 実装に関する事項

いくつかの領域において、 MPI の実装が動作環境やシステムとやりとりすることがある。 MPI はいかなるサービス（入出力やシグナル処理など）の提供も義務づけてはいないが、これらのサービスが使用できるのであれば、このような動作を提供するよう強く提唱している。これは、同じサービスを提供するプラットフォーム間の移植性を実現する上で、非常に重要な事項である。

2.8.1 基本ランタイム・ルーチンの独立性

MPI プログラムでは、以下のことが必要である。基本的な言語環境の一部であるライブラリルーチン（たとえば Fortran 言語の `date` や `write`、 ANSI C 言語の `printf` や `malloc` など）が、 MPI_INIT と MPI_FINALIZE の間に実行される場合、それらがそれぞれ独立に動作すること、およびそれらが完了するかどうかは同じ MPI プログラム内の他のプロセスの動作には依存しないこと。

これは、操作が集団的な並列サービスを提供するライブラリルーチンの作成を妨げるものではまったくない。しかしながら以下のプログラムは、 MPI_COMM_WORLD のサイズに関係なく、 ANSI C 言語環境で最後まで動作するものと期待されている（実行ノードで入出力が行えることが前提）。

```
int rank;
MPI_Init( argc, argv );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
if (rank == 0) printf( "Starting program\n" );
MPI_Finalize();
```

同様の Fortran 77 言語プログラムも最後まで動作するものと期待される。

逆に、要求されないものの例としては、複数のタスクから呼び出されるとき、これらのルーチンが特定の順序で動作すること、がある。たとえば MPI では、以下のプログラムからの出力がどうなるべきかもどうなるのが望ましいかも定めない（ここでも実行ノードで入出力が行えることが前提）。

```
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
printf( "Output from task rank %d\n", rank );
```

さらに、リソースの枯渇その他のエラーによって失敗した呼び出しは、ここで要求している事項に違反するとは見なされない（ただしこれらは、正常終了しなくてもよいだけで、終了する必要はある）。

2.8.2 POSIX シグナルとの相互作用

MPI は、UNIX 環境で、シグナルを使用したプロセス間のやりとりについては指定していないし、また MPI 通信と関係しない他のイベントを使用するやりとりについても規定しない。つまり、シグナルは MPI から見て特別な意味を持たず、実装者は、シグナルが透過的になるように MPI を実現するよう試みるべきである。シグナルによって保留された MPI 呼び出しは、シグナルが処理されたあと、処理を再開して完了するべきである。一般的に、MPI から見て検知可能であるか意味を持つかする処理状態は、MPI 呼び出しによってのみ影響を受けるべきである。

MPI をスレッドやシグナルと共に安全に使えるものにしようとすると、微妙な影響がいろいろ出てくる。例えば UNIX システムでは、SIGALRM のような捕捉可能なシグナルのせいで、MPI ルーチンがそのシグナルのないときと違う動作をすることがあってはならない。もちろん、シグナルハンドラが MPI 呼び出しを発行したり、MPI ルーチンが動作している環境を変更したりする（たとえばすべての使用可能メモリ領域を消費するなど）場合、MPI ルーチンは、その状況に対して適切に動作する必要がある（この場合で言えば、複数スレッド向け MPI 実装と同じ動作をするべきである）。

二つめの影響として、MPI 呼び出しを実行するシグナルハンドラが、MPI の動作を妨げることがあってはならないということがある。たとえば、シグナルハンドラの中で行われるいかなる種類の MPI 受信も MPI 実装の誤動作を引き起こしてはならない。ただし、実装はシグナルハンドラ内での MPI 呼び出しを禁止しても構わず、そのような使用を検出する必要はない。

MPI が `SIGALRM`、`SIGFPE`、`SIGIO` を使用しないことが非常に望ましい。実装はその MPI 実装が使用する全てのシグナルを明らかに文書で必ず示さなければならない。この情報提供に適切な場所の一例は、MPI についての Unix `man` ページである。

2.9 プログラム例

この文書中のプログラム例は、解説のみを目的としている。標準を規定することを意図したものではない。またプログラム例については、注意深いチェックや検証がおこなわれているわけではない。

Chapter3

1 対 1 通信

3.1 概要

プロセスによるメッセージの送信と受信が基本的な MPI 通信メカニズムである。1 対 1 通信の基本的な操作は、送信 (**send**) と受信 (**receive**) である。その使用法を以下の例で説明する。

```
#include "mpi.h"
main( argc, argv )
int argc;
char **argv;
{
    char message[20];
    int myrank;
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    if (myrank == 0)    /* プロセス 0 のコード */
    {
        strcpy(message,"Hello, there");
        MPI_Send(message, strlen(message), MPI_CHAR, 1, 99, MPI_COMM_WORLD);
    }
    else                /* プロセス 1 のコード */
    {
        MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
        printf("received :%s:\n", message);
    }
```

```
    MPI_Finalize();
}
```

この例では、プロセス 0 (myrank = 0) は送信関数 MPI_SEND を使用して、プロセス 1 にメッセージを送っている。この関数では、メッセージデータを取出すための送信側のメモリ内の送信バッファを指定する。上の例の場合、送信バッファは変数 message を含む、プロセス 0 のメモリ上の記憶領域である。送信バッファの位置、サイズ、型は、送信関数の先頭 3 つの引数により指定される。送信されるメッセージは、 13 文字からなる変数 message である。またこの送信関数は、エンベロープ (訳注: データを包む封筒,**3.2.3**参照) をメッセージに付加する。エンベロープ中には、メッセージ送信先と、受信関数が特定のメッセージを選択するときに使用出来る区別情報がある。送信関数の最後の 3 つの引数によって、送信されるメッセージのエンベロープが特定される。

プロセス 1(myrank = 1) が、受信関数 MPI_RECV を使用して、このメッセージを受信する。エンベロープの値に従って受信されるメッセージが選択され、メッセージ・データが受信バッファに格納される。上の例では、受信バッファは、プロセス 1 のメモリ上の変数 message を含む記憶領域から構成される。受信関数の先頭 3 つの引数により受信バッファの位置、サイズ、型が指定される。最後の引数は、受け取ったメッセージの情報の返却の為に使用される。

次の節では、ブロッキング送信関数および受信関数について説明する。送信、受信、ブロッキング通信の意味、型一致の条件、異機種環境における型変換、一般的な通信モードについて説明する。ノンブロッキング通信については次に説明し、そのあとにチャネル風の構成、送受信操作などを続ける。さらにそのあと、異機種間のデータ転送や不連続データを効率的に転送するための一般的なデータの型について考察する。最後に、メッセージの明示的なパッキング、アンパッキングの関数について説明する。

3.2 ブロッキング送信関数および受信関数

3.2.1 ブロッキング送信

ブロッキング送信関数の構文は、以下のようになる。

MPI_SEND(buf, count, datatype, dest, tag, comm)

入力	buf	送信バッファの先頭アドレス (選択型)
入力	count	送信バッファ内の要素数 (非負の整数型)
入力	datatype	送信バッファの各要素のデータ型 (ハンドル)
入力	dest	送信先のランク (整数型)
入力	tag	メッセージタグ (整数型)
入力	comm	コミュニケータ (ハンドル)

```
int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
```

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

この呼び出しのブロッキングの意味については、3.4節で記述する。

3.2.2 メッセージ・データ

MPI_SEND 関数によって指定される送信バッファは、datatype で示される型の、連続した count 個のエントリによって構成される。このエントリの開始アドレスは buf である。メッセージの長さは、バイト数ではなく要素数で指定することに注意すること。後者はマシン非依存になっており、アプリケーション・レベルに近くなる。

メッセージのデータ部分は、datatype で示される型の連続した count 個の値で構成される。count の値は 0 でも良く、その場合、メッセージのデータ部分が空になる。メッセージのデータ値に指定できる基本的なデータ型は、ホスト言語の基本的なデータ型に対応している。Fortran 言語で利用可能この引数の型および対応する Fortran 言語のデータ型を以下に示す。

MPI datatype	Fortran datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

C 言語で利用可能なこの引数の型および対応する C 言語のデータ型を以下に示す。 types are listed below.

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

データ型 MPI_BYTE と MPI_PACKED は、 Fortran 言語や C 言語のデータ型に対応するものがない。データ型 MPI_BYTE は、バイト（8 桁の 2 進数）で構成される。バイトは解釈されていないもので、文字とは異なるものである。異なるマシンでは異なる文字表現になることがあり、また文字を表現するときに複数バイトを使用することもある。一方、バイトはすべてのマシンで同じ 2 進値を持つ。データ型 MPI_PACKED の使用については、 3.13節で説明する。

MPI は上に列挙した、Fortran77 言語と ANSI C 言語の基本データ型と対応するデータ型のサポートを要求する。ホスト言語がその他のデータ型を持つ場合、MPI データ型にも追加のものが必要になる。つまり longlong int 型として宣言された C 言語の整数値（64 ビット）に対する MPI_LONG_LONG_INT、DOUBLE PRECISION 型として宣言された Fortran 言語の倍精度複素数に対する MPI_DOUBLE_COMPLEX、REAL*2、REAL*4、REAL*8 型としてそれぞれ宣言された Fortran 言語の実数に対する MPI_REAL2、MPI_REAL4、MPI_REAL8、INTEGER*1、INTEGER*2、INTEGER*4 としてそれぞれ宣言された Fortran の整数に対する MPI_INTEGER1、MPI_INTEGER2、MPI_INTEGER4 などがそれにあたる。

> 根拠 MPI のデザインにおける目標の 1 つは、追加的なプリプロセシングやコンパイルをおこなうことなく、MPI をライブラリとして実装できるようにすることである。そのため、通信呼び出しが通信バッファに変数のデータ型についての情報を持つことを前提とすることはできない。この情報は、明示的な引数によって提供しなければならない。このようなデータ型の情報の必要性は、3.3.2節で説明する。（根拠の終わり）

3.2.3 メッセージ・エンベロープ

データ部分の他に、メッセージは、メッセージを区別したり選択的に受信したりするときに使用できる情報を伝える。この情報は、一定の数のフィールドによって構成されるもので、集合的にメッセージ・エンベロープと呼ばれる。これらのフィールドは次のようになっている。

送信元
送信先
タグ
コミュニケータ

メッセージの送信元は、メッセージの送信側の ID によって暗黙的に決定される。他のフィールドは、送信関数の引数によって指定される。

メッセージの送信先は、dest 引数で指定される。

メッセージ・タグは整数値であり、tag 引数で指定される。この整数は、プログラムがメッセージのタイプを区別する為に使用される。有効なタグの値の範囲は、0,...,UB であるが、ここで言う UB は実装依存である。この値は、第 7章で述べているように、属性 MPI_TAG_UB の値を問い合わせることで得ることが出来る。MPI では、UB の値が 32767 より大きいことを要求している。

comm 引数により、送信関数で使用されるコミュニケータが指定される。コミュニケータについては第 5章で説明するが、以下にその使用法を簡単にまとめる。

コミュニケータは、通信操作に対応する通信コンテキストを指定する。それぞれの通信コンテキストに応じて、別々の “通信世界” が提供され、メッセージは常にそれらが送られるコンテキストの中で受信されて、異なるコンテキストで送られたメッセージが干渉することはない。

コミュニケータはまた、この通信コンテキストを共有するプロセスの集合を特定する。このプロセス・グループ内ではプロセスは順序付けられ、グループ内でのランクで識別される。このため、送信先を示す引数 dest の有効な値の範囲は、0, ... , n-1 となる。ただし n はグループ内のプロセスの数である。（コミュニケータがグループ間コミュニケータの場合、送信先はリモート・グループ内のランクで識別される。第 5章参照のこと。）

MPI は定義済みのコミュニケータ MPI_COMM_WORLD を提供する。これを使用すると、 MPI 初期化のあとアクセスできるすべてのプロセスとの通信が可能になり、プロセスは、 MPI_COMM_WORLD 内のランクで識別される。

ユーザへのアドバイス ユーザーが、既存のほとんどの通信ライブラリで提供されているような、フラットな名前空間および 1 つの通信コンテキストで満足していれば、 comm 引数として使用するのは定義済みの変数 MPI_COMM_WORLD で十分である。これを使用することによって、すべてのプロセスとの通信が初期化時に使用できるようになる。

ユーザーは、第 5章で説明するように新しいコミュニケータを定義することができる。コミュニケータは、ライブラリとモジュールのための有効なカプセル化メカニズムを提供する。これらを使用することにより、モジュールは、独立の通信世界と独自のプロセス番号体系を持つことができる。（ユーザへのアドバイスの終わり）

実装者へのアドバイス メッセージ・エンベロープは通常固定長のメッセージ・ヘッダとして符号化される。しかしながら、実際の符号化は、実装依存である。いくつかの情報（たとえば送信元または送信先）は暗黙的にもすることができ、メッセージによって明示的に転送される必要はない。またプロセスは、相対的なランクや、絶対的な ID などにより識別できる。（実装者へのアドバイスの終わり）

3.2.4 ブロッキング受信

ブロッキング受信関数の構文は、以下のようになる。

MPI_RECV (buf, count, datatype, source, tag, comm, status)

出力	buf	受信バッファの先頭アドレス (選択型)
入力	count	受信バッファ内の要素数 (整数型)
入力	datatype	受信バッファの各要素のデータ型 (ハンドル)
入力	source	送信元のランク (整数型)
入力	tag	メッセージタグ (整数型)
入力	comm	コミュニケータ (ハンドル)
出力	status	ステータスオブジェクト (ステータス)

```
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm, MPI_Status *status)
```

```
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),
    IERROR
```

この呼び出しのブロッキングの意味については、3.4節で記述する。

受信バッファは、datatype で指定される型の連続した count 個の要素を含む記憶領域によって構成される。この記憶領域の開始アドレスは buf である。受信されるメッセージの長さは、受信バッファの長さ以下でなければならない。オーバーフロー・エラーは、送られてくるデータが切り詰めなくては受信バッファに収まらない場合に発生する。

受信バッファより短いメッセージが到着した場合、(短い) メッセージに対応する範囲内の領域のみで更新が行われる。

ユーザへのアドバイス 3.8節で説明している MPI_PROBE 関数を使用すると、長さの不明なメッセージを受信することができる。(ユーザへのアドバイスの終わり)

実装者へのアドバイス プログラムエラーの場合の特定の動作が MPI によって規定されていない場合でも、オーバーフローの処理方法として推奨される方法は、入ってくるメッセージの送信元とタグについての情報を status に入れて戻ることである。その (オーバーフローが発生した) 受信関数はエラー・コードを返す。高品質の実装ではまた、受信バッファの範囲外のメモリは上書きされない事を保証すべきである。

メッセージが受信バッファより短い場合、MPI は非常に厳格に他の範囲の更新を禁止する。ある種の最適化のために、もっと寛大な処置は可能であろうが、これは許されていな

い。実装では、たとえそれが奇数アドレスでも、受信バッファの終了地点で、受信メモリに対するコピーを終了させる準備をしておかなくてはならない。（実装者へのアドバイスの終わり）

受信関数は、メッセージ・エンベロープの値によってメッセージを選択する。そのエンベロープが、受信関数によって指定された source、 tag、 comm のそれぞれの値と一致する場合にメッセージは受信関数によって受信される。受信側は、 source の値としてワイルドカード MPI_ANY_SOURCE、及び / 又は、 tag の値としてワイルドカード MPI_ANY_TAG を指定することができ、これによって任意の送信元、及び / 又は、タグが受け入れ可能であることを示す。 comm の値としてはワイルドカードを指定することはできない。このように、メッセージが受信プロセスに送られ、コミュニケータが一致し、送信元が一致（source= MPI_ANY_SOURCE パターンになっていない場合）、タグが一致（tag= MPI_ANY_TAG パターンになっていない場合）する場合に限り、メッセージは受信関数で受信される。

メッセージ・タグは、受信関数の tag 引数によって指定される。引数 source が、 MPI_ANY_SOURCE と異なる場合は、同じコミュニケータに付随するプロセス・グループ（グループ間コミュニケータの場合はリモート・プロセス・グループ）内のランクとして指定される。そのため、 source 引数に対応する有効な値の範囲は、 {0,...,n-1}∪{MPI_ANY_SOURCE} になる（ただし n はこのグループ内のプロセス数）。

送信関数と受信関数の間の次の違いに注意しなければならない：受信関数では、任意の送信元からメッセージを受けつけるが、送信関数では、特定の受信先を指定しなければならない。このことは、データ転送が送信側によってなされる “プッシュ型” 通信メカニズムに対応していることを示す（データ転送が受信側によってなされる “プル型” メカニズムとは異なる）。

送信元 = 送信先は許される、つまりプロセスがメッセージを自分自身に送ることは可能である（ただし、上記で説明したブロッキング送受信操作の場合、デッドロックを引き起こす可能性があるため安全ではない。 3.5節を参照のこと）。

実装者へのアドバイス メッセージ・コンテキストや、コミュニケータのその他の情報を、追加のタグ・フィールドとして実装することができる。このフィールドではワイルドカード一致が許可されていないことや、このフィールドの値の設定がコミュニケータ操作関数で制御されるなどの点で通常のメッセージ・タグとは異なっている。（実装者へのアドバイスの終わり）

3.2.5 返却ステータス

受信関数でワイルドカード値が使用された場合、受信したメッセージの送信元やタグはわからないことがある。また一つの MPI 関数で複数のリクエストが実行された場合（3.7.5節を参照）、それぞれのリクエストごとに、別々のエラーコードを返さなければならない場合もあり得る。こ

れらの情報は MPI_RECV 関数の status 引数で返される。 status の型は、 MPI 定義である。ステータス変数は、ユーザーによって明示的に割り当てられなければならない。つまりこれらはシステム・オブジェクトではない。

C 言語の場合、 status は、 MPI_SOURCE、 MPI_TAG、 MPI_ERROR という名前の 3 つのフィールドを含む構造体になる。その構造体は追加されたフィールドを含んでいるかもしれない。 status.MPI_SOURCE、 status.MPI_TAG、 status.MPI_ERROR には、それぞれ受信メッセージの送信元、タグ、エラー・コードが含まれる。

Fortran 言語の場合、 status は、 MPI_STATUS_SIZE のサイズを持つ INTEGER の配列になる。 3 つの定数 MPI_SOURCE、 MPI_TAG、 MPI_ERROR はそれぞれ送信元、タグ、エラーが格納されているエントリを示す添字である。それ故、 status(MPI_SOURCE)、 status(MPI_TAG)、 status(MPI_ERROR) には、それぞれ受信メッセージの送信元、タグ、エラー・コードが含まれる。

一般的に、メッセージ通信の呼び出しは、ステータス変数のエラー・コード・フィールドを変更しない。このフィールドは、3.7.5節で説明する複数のステータスを返す関数でのみ更新される可能性がある。このフィールドは、それらの関数が MPI_ERR_IN_STATUS というエラー・コードを返す場合のみ更新される。

> 根拠 ステータスのエラー・フィールドは、 MPI_WAIT のように、 1 つのステータスのみを返す関数の場合必要ない。なぜなら、この場合、関数自体から返された情報と同一だからである。現在の MPI の設計では、このような場合にエラー・フィールドをセットするという余分なオーバーヘッドを避けている。このフィールドは、複数のステータスを返す関数でのみ必要になる。なぜなら、それぞれの要求で異なるエラーが発生する可能性があるためである。（根拠の終わり）

ステータス引数はまた、受信したメッセージの長さについての情報を返す。ただし、この情報はステータス変数のフィールドとして直接得られるわけではなく、この情報を “解読” するために MPI_GET_COUNT の呼び出しが必要になる。

MPI_GET_COUNT(status, datatype, count)

入力	status	受信関数の返却ステータス (ステータス)
入力	datatype	受信バッファの各エントリのデータ型 (ハンドル)
出力	count	受信されたエントリの数 (整数型)

```
int MPI_Get_count(MPI_Status status, MPI_Datatype datatype, int *count)

MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
```

この関数は受信したエントリの数を返す（ここでもバイトではなく、それぞれの *datatype* が示す型でのエントリの数を数える）。 datatype 引数は、 status 変数をセットした受信関数に渡されたデータ型と一致しなければならない（後程、 3.12.5節で、 MPI_GET_COUNT がある特定の状況で MPI_UNDEFINED 値を返す場合を考察する）。

根拠 メッセージ通信ライブラリによっては、入出力の count、 tag、 source 引数を使用するものがある。つまりこれらの引数を、入ってくるメッセージの選択基準の指定と、受信メッセージの実際のエンベロープ値の返却の両方で使用する。ステータスを示す異なった引数を使用することにより、入出力引数を用いる場合に発生することの多いエラーを回避することができる（例えば MPI_ANY_TAG 定数を受信側のタグとして使用する場合など。）。ライブラリによっては、暗黙的に “最後の受信メッセージ” を用いる呼び出しを使用するものがあるが、これはスレッド・セーフではない。

性能を向上させるために、 datatype 引数が MPI_GET_COUNT に渡される。メッセージ中に含まれる要素の数を勘定せずにメッセージを受信することもあり、要素数の値は必要でないことが多い。また、こうすることにより、同じ関数を MPI_PROBE の呼び出し後に使用することができる。（根拠の終わり）

全ての送受信関数で、 buf、 count、 datatype、 source、 dest、 tag、 comm、 status 引数が、この節で説明したブロッキング関数 MPI_SEND、 MPI_RECV と同じ様に使用される。

3.3 データ型の一致とデータ変換

3.3.1 型一致規則

メッセージ転送を、次の 3 つのフェーズで構成されるものと考えることができる。

1. データが送信バッファから引き出され、メッセージが組立られる。
2. メッセージが送信元から受信側へ転送される。
3. 入ってくるメッセージからデータが取りだされ、分解されて受信バッファに入れられる。

この 3 つのフェーズで型一致を守らなければならない。送信バッファのそれぞれの変数の型が、送信関数でのエントリで指定された型と一致しなければならない。また送信関数で指定された型は、受信関数で指定された型と一致しなければならない。さらに、受信バッファのそれぞれの変数の型も、受信関数のエントリで指定された型と一致しなければならない。これらの 3 つの規則が守られていないプログラムは、間違いである。

型一致をもっと正確に定義するためには、ホスト言語の型と通信動作で指定した型との一致、および送信側と受信側の型の一致という、 2 つの問題を扱う必要がある。

送信と受信の型は、両方の関数の型の名前が同一であれば一致する（フェーズ 2）。つまり MPI_INTEGER は、 MPI_INTEGER と一致し、 MPI_REAL は MPI_REAL と一致するということである。この規則には 1 つだけ例外があり、 3.13節で説明した MPI_PACKED 型は他のどんな型とも一致させることができる。

ホスト・プログラムの変数の型は、通信関数で使用されているデータ型名が、ホスト・プログラム変数の基本型と対応する場合、その関数で指定した型と一致する。たとえば、型名 MPI_INTEGER を持つエントリは、 INTEGER 型の Fortran 言語変数と一致する。 Fortran 言語と C 言語におけるこの対応を示す表が 3.2.2節にある。この規則にも例外が 2 つある。型名 MPI_BYTE や MPI_PACKED は、データ記憶領域の任意のバイトと対応するために使用されるものであり（バイトによるアドレス指定可能マシンの場合）、このバイトを含む変数のデータ型とは関係ない。 3.13に示すように、 MPI_PACKED 型は、明示的にパックするデータを送信したり、明示的にアンパックされるデータを受信したりするときに使用される。 MPI_BYTE 型は、メモリ内の任意のバイトのバイナリ値を変更なしに転送するときに使用される。

以上をまとめると、型一致規則は以下の 3 つのカテゴリに分類される。

- 型の指定された値の通信 (すなわち MPI_BYTE データ型以外を使用する場合) では、送信側のプログラム、送信関数、受信関数、受信側のプログラムでの対応するエントリのデータ型は、全て一致しなければならない。
- 型の指定されない値の通信 (すなわち MPI_BYTE データ型を使用する場合) 、すなわち送信側と受信側の両方が MPI_BYTE データ型を使用する場合では、送信側プログラムと受信側プログラムでの、対応するエントリの型になんらの要件もなく、それらが同じである必要もない。
- MPI_PACKED が使用されている、パックされたデータを含む通信。

以下の例で、最初の 2 つの場合を説明する。

例 **3.1** 送信側と受信側が一致した型を指定する場合

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank.EQ.0) THEN
    CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
ELSE
    CALL MPI_RECV(b(1), 15, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

a と b の両方が、 10 以上の大きさの実数型配列の場合、このプログラムは正しい。 (Fortran 言語の場合、たとえば a(1) が 10 個の実数要素を持つ配列に記憶領域を共有させる (equiv-

alence される) ことができる場合など、a と b のいずれかのサイズが 10 より小さくても、このプログラムを使用することは正しい時がある。)

例 **3.2** 送信側と受信側の型が一致しない場合

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank.EQ.0) THEN
    CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
ELSE
    CALL MPI_RECV(b(1), 40, MPI_BYTE, 0, tag, comm, status, ierr)
END IF
```

このプログラムは、送信側と受信側が一致するデータ型引数を指定しないため、エラーになる。

例 **3.3** 送信側と受信側が型指定されない値での通信をした場合

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank.EQ.0) THEN
    CALL MPI_SEND(a(1), 40, MPI_BYTE, 1, tag, comm, ierr)
ELSE
    CALL MPI_RECV(b(1), 60, MPI_BYTE, 0, tag, comm, status, ierr)
END IF
```

このプログラムは、a と b の型とサイズにかかわらず正しい（この結果がメモリ・アクセスの制限を越えない場合）。

ユーザへのアドバイス　MPI_BYTE 型のバッファが MPI_SEND 関数の引数として渡された場合、MPI は buf 引数によって示されたアドレスから始まる、連続して格納されたデータを送る。これを行うと、通常のユーザーが期待するものとデータ・レイアウトが違っている場合、予期しない結果になることがある。たとえば、Fortran 言語コンパイラの中には、CHARACTER 型の変数を、文字の長さと、実際の文字列に対するポインタを含む構造体として実装するものがある。このような環境では、MPI_BYTE 型を使用して、Fortran 言語の CHARACTER 型の変数を送受信すると、文字列を転送したときの予想結果と違う結果になることがある。このためユーザーに対しては、可能な限り型指定の通信を使用するように奨める。（ユーザへのアドバイスの終わり）

MPI_CHARACTER 型

MPI_CHARACTER 型は、 CHARACTER 型の Fortran 言語変数に格納された文字列全体ではなく、その変数の中の 1 文字と対応する。 CHARACTER 型 Fortran 言語の変数または部分文字列は、文字の配列であるかのように転送される。これについては以下の例で説明する。

例 **3.4** Fortran 言語の CHARACTER の転送

```
CHARACTER*10 a
CHARACTER*10 b

CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank.EQ.0) THEN
    CALL MPI_SEND(a, 5, MPI_CHARACTER, 1, tag, comm, ierr)
ELSE
    CALL MPI_RECV(b(6:10), 5, MPI_CHARACTER, 0, tag, comm, status, ierr)
END IF
```

プロセス 1 における文字列 b の最後の 5 文字が、プロセス 0 における文字列 a の最初の 5 文字で置換される。

根拠 もう 1 つの選択肢は、 MPI_CHARACTER を任意の長さの文字列と一致させることである。こうした場合には問題が発生する。

Fortran 言語の文字変数は、一定の長さの文字列で、特殊な終端記号がない。文字を表現するときの決まった規則はなく、またどのようにしてその長さを格納するかに関する規則もない。コンパイラによっては、文字引数を一対の引数としてルーチンに渡すものもある。この場合、 1 つには文字列のアドレスが含まれ、もう 1 つには文字列の長さが含まれる。派生データ型で定義された型 (3.12節) を含む通信バッファが、ある MPI 通信関数に渡される場合を想定してみよう。もし、この通信バッファに CHARACTER 型が含まれている場合、これらの文字列の長さの情報が MPI ルーチンに渡されることはない。

この問題があるために、 MPI 関数では、文字列の長さに関する明示的な情報を指定しなければならない。 MPI_CHARACTER 型に長さのパラメータを追加することはできるが、これはあまり便利ではなく、新たなデータ型を定義することによって同じ機能を実現することができる。(根拠の終わり)

実装者へのアドバイス コンパイラによっては、 Fortran 言語の CHARACTER 引数を、長さと実際の文字列を示すポインタを持つ構造体として渡すものもある。このような環境では、文字列に至るために、 MPI 呼び出しでポインタの参照解決をおこなう必要がある。(実装者へのアドバイスの終わり)

3.3.2 データ変換

MPI の目標の 1 つは、異機種環境で並列計算をサポートすることである。異機種環境で通信を行うときは、データ変換が必要になる場合がある。ここでは以下の用語を使用する。

型変換 値のデータ型を変更すること。例えば、 REAL を丸めて INTEGER に変換すること。

表現変換 値の 2 進表現を変更すること。例えば、 16 進浮動小数から IEEE 浮動小数に変換すること。

型一致規則により、 MPI 通信では型変換は伴わない。それに対して、ある型の表現形式が異なるような環境の間でこの型のデータが転送される場合には、 MPI 通信での表現変換が必要になる。 MPI では表現変換の規則を規定していない。そのような変換では、整数、論理、文字としての値を維持することと、浮動小数としての値を、相手側のシステムで表現できる最も近い値に変換することが必要である。

浮動小数変換の場合、オーバーフローやアンダーフローの例外が発生することもある。整数や文字の変換でも、あるシステムで表現できる値が別のシステムで表現できないとき、例外が発生することがある。表現変換のときに発生する例外は、結果的に通信時のエラーになる。エラーは、送信操作と受信操作のいずれか、またはその両方で発生する。

メッセージで送信されるデータが、型の指定されない値の場合（つまり MPI_BYTE 型）、受信側に格納されるそのバイトの 2 進表現は、送信側でロードされるバイトの 2 進表現と同一である。送信側と受信側が同じ環境で動作していても、あるいは違う環境で動作していても、これは真になる。表現変換は必要ない（ MPI_CHARACTER 型や MPI_CHAR 型の値が転送されるときは、表現変換が発生し得ることに注意、例えば EBCDIC 符号から ASCII 符号への変換 ）。

すべてのプロセスが同じ環境で動作するような、同機種環境で MPI プログラムが実行されるとき、変換を行う必要はない。

3.1から 3.3までの 3 つの例題を考えてみる。 a と b が 10 以上のサイズを持つ REAL の配列である場合、最初のプログラムは正しい。送信側と受信側が異なる環境で実行する場合、送信バッファから取り出される 10 個の実数値が、受信バッファに格納される前に、受信側の実数表現に変換される。送信バッファから取り出される実数の要素の数が、受信バッファに格納される実数の要素の数と等しくても、格納されるバイトの数はロードされるバイトの数と同じである必要はない。例えば、送信側が実数に 4 バイトの表現を使用し、受信側が 8 バイトの表現を使用することもある。

2 番目のプログラムはエラーになり、その動作は保証されない。

3 番目のプログラムは正しい。送信側と受信側が異なる環境で動作しても、送信バッファからロードされるのとまったく同じ 40 バイトの列が受信バッファに格納される。送られるメッセージは、受信されるメッセージとまったく同じ長さ（バイトで）と同じ 2 進表現を持っている。 a

と b が異なる型の場合、または同じ型であっても異なるデータ表現が使用されている場合、受信バッファに格納されたビット列は、送信バッファで符号化された値とは異なる値を符号化するかもしれない。

データの表現変換は、メッセージのエンベロープにも適用される。送信元、送信先、タグは、全て整数値であり、変換の必要があり得る。

実装者へのアドバイス　現在の定義では、メッセージにデータ型の情報を持つことを要求していない。送信側と受信側の両方が、データ型に関する完全な情報を持っている。異機種環境では、XDR のようなマシンに依存しないコード変換を使用するか、受信側で送信側の表現を独自の表現に変換するか、さらには送信側で変換を行うかのいずれかになる。

システムで送信側と受信側のデータ型の間の不一致を検出するようにするため、その他の型情報をメッセージに追加することができる。これは、遅いがより安全という意味で、デバッグ時に特に便利である。（実装者へのアドバイスの終わり）

MPI では、言語間通信のサポートは必須ではない。メッセージが C 言語プロセスから Fortran 言語プロセスへ送られる場合、またはその逆の場合、プログラムの動作は保証しない。

根拠　C 言語の型と Fortran 言語の型との間での対応関係については合意された標準が無いため、MPI では言語間通信を取り扱わない。そのため、言語を混在させる MPI プログラムはポータブルではない。（根拠の終わり）

実装者へのアドバイス　MPI の実装者は、Fortran 言語プログラムで、MPI_INT、MPI_CHAR などのような “C 言語の MPI 型” を使用できるようにし、C 言語プログラムで Fortran 言語の型を使用できるようにすることによって、言語間通信をサポートできるようにすることができる。（実装者へのアドバイスの終わり）

3.4　通信モード

3.2.1節で記述された送信呼び出しは、ブロッキングである: つまり、メッセージデータとエンベロープが安全に格納されて、送信側が送信バッファを自由にアクセスしたり上書きできるようになるまでは、この呼び出しは戻らない。

メッセージは、対応する受信バッファに直接コピーされるか、もしくは、一時的なシステムバッファにコピーされる。メッセージのバッファリングは、送信操作と受信操作を独立なものとする。ブロッキング送信は、たとえ対応する受信が受信側で実行されていなくても、メッセージがバッファリングされるとすぐに完了することができる。一方で、メッセージバッファリングは、新たなメモリ間コピーを伴い、また、バッファリングのためのメモリアロケーションを必要とす

るのでコストが高くなるかもしれない。 MPI はいくつかの通信モードの選択肢を提供しており、これにより、通信プロトコルの選択を制御することができる。

3.2.1節で記述された送信呼び出しは、標準 送信モードを使用した。このモードでは、送出メッセージがバッファリングされるかどうかを決定するのは MPI に任せている。 MPI は、送出メッセージをバッファリングするかもしれない。このような場合には、送信呼び出しは、対応する受信が起動される前に完了することができる。一方で、バッファ領域が使用できないかもしれないし、 MPI が性能上の理由から、送出メッセージをバッファリングしないことを選択するかもしれない。この場合には、送信呼び出しは、対応する受信が発行され、そして、データが受信側に移動してしまうまでは完了しない。

このように、標準モードでの送信は、対応する受信が発行されているかどうかにかかわらず、開始することができる。これは、対応する受信が発行される前に完了することもある。標準モード送信は ノンローカルである: つまり、送信操作が成功完了するかどうかは、対応する受信が発生するかどうかに依存する。

> 根拠　 MPI において、標準送信がバッファリングするかどうかを指定しないのは、可搬性のあるプログラムを作るという要求に起因している。メッセージサイズが増えるに従って、どのシステムもバッファ資源を使い尽くすため、また、ある実装では、バッファリングをほとんど提供しないために、 MPI は正しい（従って可搬性のある）プログラムは標準モードでシステムバッファリングに依存しないという立場をとる。バッファリングは正しいプログラムの性能を向上させるかもしれない。しかし、プログラムの結果には影響を与えない。もし、ユーザーがある量のバッファリングを保証したければ、バッファモードの送信と共に、3.6節のユーザー提供バッファシステムを使用するべきである。（根拠の終わり）

他に、３つの追加的な通信モードが存在する。

バッファ モード送信操作は、対応する受信が発行されているかどうかにかかわらず、開始することができる。これは、対応する受信が発行される前に完了するかもしれない。しかしながら、標準送信とは異なり、この操作はローカルである。そして、その完了は、対応する受信が発生するかどうかに依存しない。従って、送信が実行され、対応する受信が発行されない場合には、 MPI は、送信呼び出しが完了できるように、送出メッセージをバッファリングしなければならない。バッファ領域が不十分な場合には、エラーが発生する。利用可能なバッファ領域の量はユーザーが制御する。（3.6節参照）バッファモードが有効に働くためには、ユーザーによるバッファアロケーションが必要になる。

同期モードを使用する送信は、対応する受信が発行されているかどうかにかかわらず、開始することができる。しかしながら、送信は、対応する受信が発行され、そして、受信操作が同期送信によって送信されたメッセージを受信開始した場合にのみ成功完了する。このように、同期

送信の完了は、送信バッファが再利用できることを示すだけでなく、受信側が実行におけるあるポイントに到達したこと、つまり、対応する受信の実行が開始されたことを示す。送信および受信の両操作がブロッキング操作の場合、同期モードの使用は、同期通信を意味する。つまり通信は、両方のプロセスが通信位置で通信を取り交わす前にはどちら側も完了しない。このモードで実行される送信は、ノンローカルである。

レディ通信モードを使用する送信は、対応する受信がすでに発行されている場合にのみ、開始することができる。それ以外の場合は、この操作はエラーになり、その結果は保証されない。システムによっては、レディ通信は、他の通信で必要なハンドシェーク操作をなくすことが可能となり、性能改善をもたらす。送信操作の完了は、対応する受信の状態には依存せず、単に送信バッファが再使用できることを意味する。レディモードを使用する通信は、標準送信操作あるいは同期送信操作と意味的には同じである。単に送信側が（対応する受信が既に発行されているという）追加情報をシステムに提供し、ある種のオーバーヘッドを抑えるということにすぎない。従って、正しいプログラムにおいては、性能を除くプログラムの動作に影響を与えることなく、レディ送信を標準送信で置き換えることができる。

これら 3 つの追加的な通信モードに対して、3 つの追加的な送信関数が提供されている。通信モードは 1 文字の接頭語で表される。 B はバッファ、 S は同期、そして R はレディである。

MPI_BSEND (buf, count, datatype, dest, tag, comm)

入力	buf	送信バッファの先頭アドレス (選択型)
入力	count	送信バッファ内の要素数 (整数型)
入力	datatype	各々の送信バッファ要素のデータ型 (ハンドル)
入力	dest	送信先のランク (整数型)
入力	tag	メッセージタグ (整数型)
入力	comm	コミュニケータ (ハンドル)

```
int MPI_Bsend(void* buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm)
```

```
MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

バッファモードの送信。

MPI_SSEND (buf, count, datatype, dest, tag, comm)

入力	buf	送信バッファの先頭アドレス (選択型)
入力	count	送信バッファ内の要素数 (整数型)
入力	datatype	各々の送信バッファ要素のデータ型 (ハンドル)
入力	dest	送信先のランク (整数型)
入力	tag	メッセージタグ (整数型)
入力	comm	コミュニケータ (ハンドル)

```
int MPI_Ssend(void* buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm)

MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

同期モードの送信。

MPI_RSEND (buf, count, datatype, dest, tag, comm)

入力	buf	送信バッファの先頭アドレス (選択型)
入力	count	送信バッファ内の要素数 (整数型)
入力	datatype	各々の送信バッファ要素のデータ型 (ハンドル)
入力	dest	送信先のランク (整数型)
入力	tag	メッセージタグ (整数型)
入力	comm	コミュニケータ (ハンドル)

```
int MPI_Rsend(void* buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm)

MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

レディモードの送信。

受信操作は一つだけであり、どの送信モードにも対応することができる。前節で記述された受信操作はブロッキングである。つまり、受信バッファが新しく受信したメッセージを取り込ん

だ後にのみ戻る。受信は、対応する送信が完了する前に、完了することができる。（もちろん、それは対応する送信が開始した後にのみ戻ることができる。）

MPI のマルチスレッド実装では、システムは、送信や受信操作でブロックされたスレッドのスケジューリング解除を行い、そして、実行のために同じアドレス空間を用いて別のスレッドのスケジューリングを行うことができる。このような場合には、通信が完了するまで、通信バッファをアクセスしたり変更したりしないことは、ユーザーの責任で行う。さもないと、計算の結果は保証されない。

根拠 たとえ送信操作が送信バッファの内容を変更しないとしても、送信バッファが使用されている間はこれに対する読み込みアクセスを禁止している。これは必要以上に厳しいように見えるかもしれない。しかし、この追加的な制限はほとんど機能損失にはならないし、また、システムによってはパフォーマンスの向上をもたらす。（データ転送が、主プロセッサとキャッシュコヒーレントでないようなＤＭＡエンジンによって行われるような場合を考えてみよ。）（根拠の終わり）

実装者へのアドバイス 同期送信は、対応する受信が発行される前には完了できないので、そのような操作に対しては通常バッファリングは行われない。

標準送信に対しては、できるだけ、送信側をブロッキングするよりも、バッファリングを選択することが推奨される。プログラマは、同期送信モードを使用することによって、対応する受信が発生するまで送信側をブロックしたい意向を明示することができる。

様々な通信モードに対する可能な通信プロトコルを以下に概説する。

レディ送信: メッセージはできるだけ早く送られる。

同期送信: 送信側は送信要求メッセージを送信する。受信側はこの要求を保存する。対応する受信が発行されたときに、受信側は送信許可メッセージを送り返し、そして、その時、送信側はメッセージを送信する。

標準送信: 短いメッセージに対しては、レディ送信プロトコルが使用され、長いメッセージに対しては、同期通信プロトコルが使用される。

バッファ送信: 送信側はメッセージをバッファにコピーし、そしてそれを、（標準送信と同じプロトコルを使用して）ノンブロッキング送信により送信する。

フロー制御とエラー回復のためには、追加的な制御メッセージが必要になると考えられる。もちろんその他にも、可能なプロトコルはたくさんある。

レディ送信は、標準送信として実装可能である。この場合には、レディ送信を使用することによる性能上の利点（あるいは欠点）はない。

標準送信は、同期送信として実装可能である。この場合には、データバッファリングは必要ない。しかしながら、多くの（ほとんどの？）ユーザーはバッファリングを期待している。

マルチスレッド環境では、ブロッキング通信の実行は、スレッドスケジューラが実行スレッドをスケジュール解除したり、実行のために別のスレッドをスケジュールすることを許可しながら、実行スレッドのみをブロックするべきである。（実装者へのアドバイスの終わり）

3.5 1 対 1 通信の意味

正しい MPI 実装は、1 対 1 通信のある種の一般的な性格を保証しており、この節ではこれについて説明する。

順序 メッセージは追越し禁止である: 送信側が 2 つのメッセージを続けて同じ送信先に送信し、両方の送信が同じ受信と対応する場合には、第 1 メッセージがまだ保留状態にある間は、この受信操作は第 2 メッセージを受信することができない。受信側が 2 つの受信を続けて発行し、両方が同じメッセージに対応する場合には、第 1 受信がまだ保留状態にある間は、第 2 受信操作はこのメッセージに対応できない。この要求により、送信と受信の対応が容易になる。このことは、プロセスがシングルスレッドでしかも受信側においてワイルドカード MPI_ANY_SOURCE が使用されていない場合に、メッセージ通信コードの確実性を保証する。（後で述べる、 MPI_CANCEL や MPI_WAITANY のようなある種の呼び出しは、非確実性のもととなる。）

プロセスがシングルスレッドの実行の場合には、このプロセスによって実行されるどの 2 つの通信も順序が決まっている。一方、プロセスがマルチスレッドの場合には、スレッド実行が意味するところでは、2 つの違ったスレッドにより実行された 2 つの送信操作の間の相対的な順序は不定となる。たとえ、1 つのスレッドが物理的に他よりも先行していたとしても、それらの操作は論理的には並行している。このような場合には、送信された 2 つのメッセージは任意の順序で受信されうる。同様に、論理的には並行している 2 つの受信操作が続いて送信された 2 つのメッセージを受信する場合には、これらの 2 つのメッセージは 2 つの受信と任意の順序で対応することができる。

例 **3.5** 追越し禁止メッセージの例

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag, comm, ierr)
    CALL MPI_BSEND(buf2, count, MPI_REAL, 1, tag, comm, ierr)
```

```
ELSE    ! ランクが 1 の場合
    CALL MPI_RECV(buf1, count, MPI_REAL, 0, MPI_ANY_TAG, comm, status, ierr)
    CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

第 1 送信により送信されたメッセージは、第 1 受信により受信されなければならない、また、第 2 送信により送信されたメッセージは、第 2 受信により受信されなければならない。

進行　対応する送信と受信の 1 対が 2 つのプロセスで起動された場合には、システムの他の動作とは独立に、これら 2 つの操作のうち少なくとも 1 つは完了する。つまり、受信が別のメッセージと対応せず完了しない場合には、送信操作が完了し、送信されたメッセージが同じ受信先で発行された対応する別の受信により消費されない場合には、受信操作が完了する。

例 **3.6** 交差して対応する 2 つの対の例

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag1, comm, ierr)
    CALL MPI_SSEND(buf2, count, MPI_REAL, 1, tag2, comm, ierr)
ELSE    ! ランクが 1 の場合
    CALL MPI_RECV(buf1, count, MPI_REAL, 0, tag2, comm, status, ierr)
    CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag1, comm, status, ierr)
END IF
```

両プロセスが、第 1 通信呼び出しを起動する。プロセス 0 の第 1 送信は、バッファモードを使用しているため、プロセス 1 の状態とは無関係に、完了しなければならない。対応する受信が発行されないので、メッセージはバッファ領域にコピーされる。（充分なバッファ領域がない場合には、このプログラムはエラーとなる。）次に、第 2 送信が起動される。この時点において、対応する 1 対の送信操作と受信操作が可能となり、両操作は完了しなければならない。次にプロセス 1 は、第 2 受信呼び出しを起動し、これはバッファリングされたメッセージと対応する。プロセス 1 が、送信されたのとは逆順にメッセージを受信したことに注意すること。

公平さ　MPI は通信のハンドリングの 公平さ をなんら保証しない。送信が発行された場合を考えてみよう。この時、送信先のプロセスが、この送信に対応する受信を繰り返し発行するのだが、その度に別の送信元から送信された別のメッセージにより追い越されるために、このメッセージが決して受信されないといったことが起こりうる。また、この受信に対応するメッセージが繰り返し受信されるのだが、このノード（で実行中の他のスレッド）で発行された他の受信に

より追い越されるために、この受信が決して完了しないといったことも起こりうる。このような状況を回避するのは、プログラマの責任である。

資源制限　保留中の通信操作は限りあるシステム資源を消費している。資源の不足により、 MPI 呼び出しの実行ができない場合には、エラーが起きる。質の高い実装では、それぞれのレディモードや同期モードでの保留中の送信及び保留中の受信に対して、（小さな）一定量の資源を使用するだろう。しかしながら、バッファ領域は、対応する受信がない場合に標準モードで送信されたメッセージを格納するために消費されうるし、バッファモードで送られてきたメッセージを格納するためには必ず消費される。多くのシステムにおいては、バッファリングに利用できる領域量は、プログラムデータメモリに比べて随分小さくなっている。そのため、利用可能バッファ領域を使い尽くすプログラムを書くことは容易である。

MPI では、バッファモードで送られるメッセージのためのバッファメモリを、ユーザーが用意することができる。さらに MPI は、このバッファ使用に対する詳細な操作モデルを指定している。 MPI の実装では、このモデルで示される以上の事が要求される。このことにより、ユーザーがバッファ送信を使ったときには、バッファオーバーフローを避けることができる。バッファのアロケーションと使用法は 3.6節で説明される。

バッファ領域の不足により完了できないバッファ送信は間違いである。その様な状況が検知されれば、プログラムが異常終了するかもしれないというエラーが出される。一方、バッファ領域の不足により完了できない標準送信操作は、バッファ領域が利用可能になるか、対応する受信が発行されるのを待ちながら、単にブロックする。この動作は、多くの状況において、望ましいものである。作成側が、繰り返し新しい値を作成して、それらを消費側に送信している状況を考えてみよう。作成側が、消費側の消費できるより速く新しい値を作成すると仮定する。バッファ送信が使用された場合には、バッファのオーバーフローが生じるだろう。これが発生するのを防ぐためには、他の同期をプログラムに加える必要がある。標準送信が使用される場合には、バッファ領域が使用できなければ、送信操作がブロックとなり、作成側は自動的に抑制される。

状況によっては、バッファ領域の不足はデッドロック状況を導く。これについては、以下の例で説明される。

例 **3.7** メッセージの交換

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
ELSE    ! ランクが 1 の場合
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
```

```
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

このプログラムは、たとえデータのためのバッファ領域が使用できなくても、成功する。この例では、標準送信操作は同期送信で置き換えることができる。

例 **3.8** メッセージ交換の試み

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
ELSE    ! ランクが1の場合
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

第1プロセスの受信操作は、その送信の前に完了しなければならず、これは第2プロセスの対応する送信が実行された場合にのみ、完了することができる。第2プロセスの受信操作は、その送信の前に完了しなければならず、これは第1プロセスの対応する送信が実行された場合にのみ、完了することができる。このプログラムは常にデッドロックとなる。他の送信モードに対しても、同じ事が起こる。

例 **3.9** バッファリングに依存した交換

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
ELSE    ! ランクが1の場合
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

各プロセスにより送信されるメッセージは、送信操作が戻り、また、受信操作が開始する前にコピーされなければならない。プログラムが完了するためには、送信される2つのメッセージのうち少なくとも1つはバッファリングされることが必要である。従って、このプログラムは、通信システムが少なくともcount個のワードデータをバッファリングできる場合にのみ、成功することができる。

ユーザへのアドバイス 標準送信が使用されるときには、バッファ領域が使用できないために両方のプロセスがブロックされる場合に、デッドロック状況が発生する。同期モードが使用される場合にも、同様のことが必ずおこる。バッファモードが使用され、十分なバッファ領域が利用できない場合には、プログラムは完了しない。しかしながら、デッドロック状況は起こらず、バッファオーバーフローになる。

プログラムが完了するために、メッセージのバッファリングが必要ない場合には、プログラムは “安全” である。そのようなプログラムでは、全ての送信を同期送信に置き換えることができ、しかも依然としてプログラムは正しく走る。この保守的なプログラミングスタイルは、最良のポータビリティを提供する。これは、プログラムの完了が、使用可能なバッファ領域の量や、使用される通信プロトコルに依存しないからである。

多くのプログラマは活動の余地が広いことを好み、例 3.9に示されるような、 “危険な” プログラミングスタイルが使用できることを好む。そのような場合には、標準送信の使用は、性能と頑健さの最良の妥協点を提供するだろう。つまり、質の高い実装では、 “通常の” プログラムがデッドロックしないように、十分なバッファリングが提供される。バッファ送信モードは、より多くのバッファリングを必要とするプログラムに対して、また、プログラマがもっと制御したい状況において使用される。このモードは、バッファオーバーフロー状況の方が、デッドロック状況よりも診断しやすいため、デバッギングの目的のためにも使用される。

3.7節で記述されるように、送出メッセージのバッファリングの必要を避けるために、ノンブロッキングメッセージ通信操作を使用することができる。このことにより、バッファ領域の不足によるデッドロックを回避でき、計算と通信との重ね合わせを許し、また、バッファのアロケーションとメッセージをバッファへコピーするオーバーヘッドを回避することにより、性能を改善する。（ユーザへのアドバイスの終わり）

3.6 バッファのアロケーションと使用法

ユーザーは、バッファモードで送られるメッセージのバッファリングに使用されるバッファを指定することができる。

MPI_BUFFER_ATTACH(buffer, size)

入力	buffer	バッファの先頭アドレス（選択型）
入力	size	バッファサイズ（バイト）（整数型）

```
int MPI_Buffer_attach( void* buffer, int size)
```

```
MPI_BUFFER_ATTACH( BUFFER, SIZE, IERROR)
    <type> BUFFER(*)
    INTEGER SIZE, IERROR
```

送出メッセージのバッファリングに使用するユーザメモリ内のバッファを MPI に与える。このバッファは、バッファモードで送られるメッセージによってのみ使用される。同時には、ただ 1 つのバッファのみが、1 つのプロセスに貼り付けられる。

MPI_BUFFER_DETACH(buffer, size)

出力	buffer	バッファの先頭アドレス (選択型)
出力	size	バッファサイズ (バイト) (整数型)

```
int MPI_Buffer_detach( void** buffer, int* size)
```

```
MPI_BUFFER_DETACH( BUFFER, SIZE, IERROR)
    <type> BUFFER(*)
    INTEGER SIZE, IERROR
```

現在 MPI に付随しているバッファを切り離す。この呼び出しは、切り離されたバッファのアドレスとサイズを返す。現在バッファにある全てのメッセージが転送されるまで、この操作はブロックする。この関数が戻った上で、ユーザーはこのバッファにより占められた領域を再利用したり、非アロケートしたりできる。

例 **3.10** バッファリングに依存した交換

```
#define BUFFSIZE 10000
int size
char *buff;
MPI_Buffer_attach( malloc(BUFFSIZE),BUFFSIZE);
/* 10000 バイトのバッファは今や MPI_Bsend で使用可能である */
MPI_Buffer_detach( &buff, &size);
/* バッファサイズは 0 になる */
MPI_Buffer_attach( buff, size);
/* 10000 バイトのバッファは再び利用可能となる */
```

ユーザへのアドバイス C 言語関数 MPI_Buffer_attach と MPI_Buffer_detach は両方とも第 1 引数に void* 型を持つが、これらの引数は異なった方法で使われる。つまり、この呼び出しがポインタ値を返せるように、 MPI_Buffer_attach へはバッファのポインタが渡され、

MPI_Buffer_detach へはポインタのアドレスが渡される。（ユーザへのアドバイスの終わり）

根拠 複雑な型変換を避けるため、両引数は void* 型として定義されている。（各々が void* と void** となるのではなく）例えば、最後の例では、 char** 型をもつ &buffは、型変換される事なく引数として MPI_Buffer_detach に渡される。正式な引数が void** を持つ場合には、呼び出しの前後で型変換を行う必要がある。（根拠の終わり）

この節の記述は、バッファモード送信における MPI の動作について説明する。現在バッファが付随していない場合、 MPI はプロセスにサイズ 0 のバッファが付随しているかのように動作する。

MPI は、送信メッセージデータが、送信プロセスにより指定されたバッファ領域へ巡回的な連続領域アロケーションの原則に従ってバッファリングされるために必要な量のバッファを、送出メッセージに対して、提供しなければならない。以下では、この原則を定義するモデル実装について概説する。 MPI は、より多くのバッファリングを提供するかもしれないし、以下に記述されるものより優れたバッファアロケーションアルゴリズムを使用するかもしれない。一方、MPI は、以下に記述した単純バッファアロケータが領域を使いきった時にはいつでもエラーシグナルを返すかもしれない。特に、プロセスに明示的に付随されたバッファがない時には、バッファ送信はエラーを引き起こすかもしれない。

MPI は、標準モード送信で実行されるバッファリングについて、問い合わせや制御のためのメカニズムを提供しない。そのような情報は、ベンダーによる実装で提供されることが望ましい。

根拠 バッファ通信の可能な実装は、多岐にわたる。バッファリングは、送信側、受信側あるいは両方で行うことができるし、バッファは 1 つの送信・受信対専用になったり、全ての通信で共有することもできるし、バッファリングは実メモリかあるいは仮想メモリで行うこともできるし、専用のメモリを使ったり、他のプロセスとの共有メモリを使うこともできるし、バッファは静的にアロケートしたり、動的にアロケートしたりもできる。これら全ての選択に互換性をもつような、バッファリングについての問い合わせや制御を行うポータブルなメカニズムを提供することは容易ではないが、これらは有益な情報を提供している。（根拠の終わり）

3.6.1 バッファモードのモデル実装

モデル実装は、 3.13節で記述されるパッキング関数とアンパッキング関数、そして 3.7節で記述されるノンブロッキング通信関数を使用する。

保留中のメッセージエントリ（pending message entries=PME）の循環キューが整備されると仮定する。各々のエントリは、保留中のノンブロッキング送信を識別する通信要求ハンド

ル、次のエントリへのポインタおよびパックされたメッセージデータなどを含んでいる。エントリは、バッファ内の連続した位置に格納される。キューの末尾とキューの先頭の間には、空き領域があってもよい。

バッファ送信呼び出しは、以下のコードのように実行されることになる。

- 先頭から末尾に向かって、まだ完了していない要求の最初のエントリに達するまで、すでに完了した通信の全てのエントリを消去しながら、PME キューを順に移動する。つまり、そのエントリを指すようにキューの先頭を更新する。
- 新しいメッセージのエントリを格納するために必要なバイト数 n を計算する。 n の上限は、以下のようにして計算できる。 MPI_BSEND で使用される count,datatype,comm 引数を持つ MPI_PACK_SIZE(count, datatype, comm, size) 関数の呼び出しは、メッセージデータのバッファリングに必要な領域の上限を返す。（3.13節参照） MPI 定数 MPI_BSEND_OVERHEAD は、このエントリ（例えば、ポインタやエンベロープ情報）で消費される追加領域の上限を与える。
- バッファ内（キューの末尾に続く領域、あるいは、キューの末尾がバッファの最後に近すぎる場合にはバッファの最初の領域）で次の連続する n バイトの空き領域を探す。
- 領域が見つからない場合には、バッファオーバーフローエラーを出す。
- PME キューの最後に、要求ハンドル、次のポインタ、パックされたメッセージデータを含む新しいエントリを（連続した領域で）追加する。データのパックには、 MPI_PACK が使用される。
- パックされたデータに対し、ノンブロッキング送信（標準モード）を発行する。
- 戻る。

3.7 ノンブロッキング通信

多くのシステムにおいて、通信と計算を重ね合わせることによって性能を向上させることができる。通信が知的な通信コントローラーによって自動的に実行できるようなシステムにこのことは特にあてはまる。軽負荷スレッドはこのような重ね合わせを達成するための一つの機構である。これよりよい性能がしばしば実現されるもう一つの機構はノンブロッキング通信を用いることである。ノンブロッキング通信開始が呼ばれると、送信操作が起動されるが、完了はしない。送信バッファからメッセージがコピーされる前に送信開始の呼出しは戻ってくる。通信を完了させるため、すなわちデータが送信バッファからコピーされたことを確認するためには、別の送信完了の呼出しが必要となる。適当なハードウエアを使えば、送信が初期化されたが完了はしていない

時点で、送信側での計算と並行させて送信側のメモリーからのデータの転送を行うことができるであろう。同様に、ノンブロッキング受信開始呼出しは受信作業を起動するが完了はしない。この呼出しはメッセージが受信バッファに格納される前に戻される。受信操作を完了させ、データが受信バッファに受け取られたことを確認するためには、別の受信完了の呼出しが必要である。適当なハードウエアを使えば、受信が起動され終了する以前に、受信側のメモリーへのデータの転送を計算と並行させて行えるであろう。ノンブロッキング受信を使用すると、情報は早い時点で受信バッファの位置で提供されているので、システムのバッファリングやメモリーからメモリーへのコピーを避けることもできるであろう。

ノンブロッキング送信開始の呼出しは、ブロック送信と同じ標準、バッファ、同期、レディの４つのモードを使用することができる。これらはブロック通信のときと同じ意味を持っている。レディ以外のすべてのモードでは、対応する受信が発行されたかどうかにかかわらず送信を開始することができる。ノンブロッキングのレディ・モードでの送信は、対応する受信が発行されたときにのみ開始することができる。すべての場合において、送信開始呼出しはローカルであり、他のプロセスの状態によらずただちに戻ってくる。もしその呼出しがシステムの資源の何かを使い果たしてしまったときは、失敗となりエラーコードが返される。 MPI の高品質な実装においては、このようなことは “病的” な場合にのみ起こるようになっていなければならない。すなわち MPI の実装は、多くの保留になっているノンブロッキング操作をサポートすることができなければならない。

送信完了の呼出しは、データが送信バッファからコピーされ終わったときに戻ってくる。このことは送信モードによっては付加的な意味をもつことがある。

送信モードが同期なら、送信は対応する受信が開始したとき、すなわち受信が発行され送信と対応されたときにのみ完了することができる。この場合には送信完了はノンローカルである。同期ノンブロッキング送信は、もしノンブロッキング受信と対応がとられれば、受信完了呼出しが発生する前に完了することができることに注意されたい。（送信側が転送が完了することを “知れ” ば、受信側が転送が完了することを “知る” 以前であってもただちに完了する。）

送信モードがバッファのときは、もし保留中の受信がなければメッセージはバッファリングされなければならない。この場合には送信完了呼出しはローカルであり、対応する受信の状態によらず継続されなければならない。

送信モードが標準のときは、もしメッセージがバッファリングされているなら、送信完了呼出しは対応する受信が起こる前に戻されることがある。一方、対応する受信が発生し、メッセージが受信バッファにコピーされるまで送信完了は終了しないこともありうる。

ノンブロッキング送信はブロッキング受信と対応させることができ、逆も可能である。

ユーザへのアドバイス 送信操作の完了は、標準モードに対しては遅延することがあり、同期モードに対しては対応する受信が発行されるまで遅延しなければならない。この２つの場合においては、ノンブロッキング送信を使用することにより、送信側は受信側より先に

進むことができ、２つのプロセスの速度の揺らぎに対し、計算がより強い耐性を持つ。

バッファ・モードとレディ・モードにおけるノンブロッキング送信はより限定された効果しか持たない。ノンブロッキング送信は可能な限りすぐに戻るが、ブロック送信はデータが送信側のメモリーからコピーされた後に戻る。この場合にはデータのコピーを計算と平行して行うことができるときにのみ、ノンブロッキング送信を使用することが有利になる。

メッセージパッシングモデルは、通信は送信側によって起動されるということを意味している。もし送信側が通信を起動したときに受信がすでに発行されているならば、通信は一般的により小さなオーバーヘッドを持つ。（データを受信側のバッファに直接移動させることができ、保留中の送信要求を待つ必要がない。）しかしながら、受信操作は対応する送信が起こった後にのみ完了することができる。ノンブロッキング受信を使用することにより、送信を待っている間も受信側をブロックすることなく、より低い通信オーバーヘッドが実現できる。（ユーザへのアドバイスの終わり）

3.7.1 通信オブジェクト

ノンブロッキング通信は、通信操作を識別するために不透明な要求オブジェクトを使用し、通信を起動する操作と終了させる操作を対応させる。これらはハンドルを通してアクセスされるシステムオブジェクトである。要求オブジェクトは、送信モード、それに対応した通信バッファ、そのコンテキスト、送信のために使用されるタグや送信先引数、受信のために使用されるタグや送信元引数などのさまざまな通信操作の性質を識別する。さらに、このオブジェクトは保留中の通信操作の状態についての情報を記憶する。

3.7.2 通信の起動

我々はブロック通信に対するのと同じ名前付けの規約を使用する。すなわちバッファ、同期、レディモードに対し B、S、R とい接頭辞を使う。さらに I(即座 immediate）という接頭辞が付くと呼出しがノンブロッキングであることを意味する。

MPI_ISEND(buf, count, datatype, dest, tag, comm, request)

入力	buf	送信バッファの先頭アドレス（選択型）
入力	count	送信バッファの中の要素の数（整数型）
入力	datatype	各送信バッファの要素のデータ型（ハンドル）
入力	dest	送信先のランク（整数型）
入力	tag	メッセージタグ（整数型）
入力	comm	コミュニケータ（ハンドル）
出力	request	通信要求（ハンドル）

```
int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm, MPI_Request *request)
MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

標準モードのノンブロッキング送信を開始する。

MPI_IBSEND(buf, count, datatype, dest, tag, comm, request)

入力	buf	送信バッファの先頭アドレス（選択型）
入力	count	送信バッファの中の要素の数（整数型）
入力	datatype	各送信バッファの要素のデータ型（ハンドル）
入力	dest	送信先のランク（整数型）
入力	tag	メッセージタグ（整数型）
入力	comm	コミュニケータ（ハンドル）
出力	request	通信要求（ハンドル）

```
int MPI_Ibsend(void* buf, int count, MPI_Datatype datatype, int dest,
               int tag, MPI_Comm comm, MPI_Request *request)
MPI_IBSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

バッファ・モードのノンブロッキング送信を開始する。

MPI_ISSEND(buf, count, datatype, dest, tag, comm, request)

入力	buf	送信バッファの先頭アドレス（選択型）
入力	count	送信バッファの中の要素の数（整数型）
入力	datatype	各送信バッファの要素のデータ型（ハンドル）
入力	dest	送信先のランク（整数型）
入力	tag	メッセージタグ（整数型）
入力	comm	コミュニケータ（ハンドル）
出力	request	通信要求（ハンドル）

```
int MPI_Issend(void* buf, int count, MPI_Datatype datatype, int dest,
               int tag, MPI_Comm comm, MPI_Request *request)
MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

同期モードのノンブロッキング送信を開始する。

MPI_IRSEND(buf, count, datatype, dest, tag, comm, request)

入力	buf	送信バッファの先頭アドレス（選択型）
入力	count	送信バッファの中の要素の数（整数型）
入力	datatype	各送信バッファの要素のデータ型（ハンドル）
入力	dest	送信先のランク（整数型）
入力	tag	メッセージタグ（整数型）
入力	comm	コミュニケータ（ハンドル）
出力	request	通信要求（ハンドル）

```
int MPI_Irsend(void* buf, int count, MPI_Datatype datatype, int dest,
               int tag, MPI_Comm comm, MPI_Request *request)
MPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
```

```
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

レディモードのノンブロッキング送信を開始する。

MPI_IRECV (buf, count, datatype, source, tag, comm, request)

入力	buf	受信バッファの先頭アドレス（選択型）
入力	count	受信バッファの中の要素の数（整数型）
入力	datatype	各受信バッファの要素のデータ型（ハンドル）
入力	dest	送信元のランク（整数型）
入力	tag	メッセージタグ（整数型）
入力	comm	コミュニケータ（ハンドル）
出力	request	通信要求（ハンドル）

```
int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source,
              int tag, MPI_Comm comm, MPI_Request *request)
MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
```

ノンブロッキング受信を開始する。

これらの呼出しは通信要求オブジェクトを割り当て、要求ハンドル（request という引数）と対応付ける。この要求は後で通信の状態を問い合わせたり、その完了を待つために使用することができる。

ノンブロッキング送信呼出しは、システムが送信バッファからデータをコピーすることを開始してもよいということを意味する。送信側は、ノンブロッキング送信操作が呼ばれた後は、送信が完了するまで送信バッファのどの部分にアクセスすることも許されない。

ノンブロッキング受信の呼出しは、システムが受信バッファにデータを書き込むことを始めてもよいということを意味する。受信側は、ノンブロッキング受信操作が呼ばれた後は、受信が完了するまで受信バッファのどの部分にアクセスすることも許されない。

3.7.3 通信の完了

関数 MPI_WAIT と MPI_TEST はノンブロッキング通信を完了させるために使われる。送信操作の完了は送信側がその時点で自由に送信バッファの記憶領域を更新できることを意味する。（送

信操作自身は送信バッファの内容を変更しない。）送信の完了はメッセージが受信されたことを意味するのではなく、通信サブシステムによって既にバッファリングされた可能性があることを意味している。しかし、同期モードの送信が使用されたときは、送信の完了は対応する受信が起動され、その結果この対応する受信によってメッセージが受信されたことを意味する。

受信操作の完了は、受信バッファが受信メッセージを取り込んでおり、受信側はそれに自由にアクセスでき、状態オブジェクトが設定されているということを意味する。対応する送信操作が完了したことを意味するわけではない。（もちろん送信が起動されたことは意味する。）

以下のような用語を使うことにしよう：ヌルハンドルとは MPI_REQUEST_NULL という値をもったハンドルである。持続的要求とそれに対するハンドルは、もしその要求が現在実行中の通信と関連していなければ非アクティブ状態である。（3.9節参照）ハンドルはヌルでも非アクティブ状態でもなければアクティブ状態である。空のステータスとは、tag = MPI_ANY_TAG と source = MPI_ANY_SOURCE を返すように設定されたステータスであり、MPI_GET- _COUNT や MPI_GET_ELEMENTS の呼出しに対して count = 0 を返すように内部的に設定されるステータスでもある。返される値が重要でない時には、状態変数に空を設定する。古くなってしまった情報をアクセスすることによるエラーを防ぐために、このような方法でステータスが設定される。

MPI_WAIT(request, status)

入出力	request	要求（ハンドル）
出力	status	ステータスオブジェクト（ステータス）

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)

MPI_WAIT(REQUEST, STATUS, IERROR)
    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
```

MPI_WAIT の呼出しは request によって識別される操作が完了したとき戻ってくる。この要求に対応する通信オブジェクトがノンブロッキング送信やノンブロッキング受信の呼出しによって作られた場合には、MPI_WAIT を呼び出すことによってオブジェクトは解放され、要求ハンドルは MPI_REQUEST_NULL に設定される。MPI_WAIT はノンローカルな操作である。

呼出しは完了した操作についての情報を status に返す。受信操作に対するステータスオブジェクトは 3.2.5節に記述したような方法でアクセスすることができる。送信操作に対するステータスオブジェクトは MPI_TEST_CANCELLED を呼び出すことによって照会することができる。（3.8節参照）

ヌルや非アクティブ状態の request 引数で MPI_WAIT を呼び出すことが許されている。この場合には操作は空の status を持ってただちに戻る。

ユーザへのアドバイス　MPI_IBSEND の後、MPI_WAIT が無事に戻ってきたということは、ユーザーが送信バッファを再使用することができるということを意味している。すなわち、データはすでに MPI_BUFFER_ATTACH によって加えられたバッファにコピーされたか、外部に送られたということである。この時点ではもはや送信を取り消すことはできないということに注意されたい（3.8節参照）。もし対応する受信が発行されなければ、バッファを解放することはできない。これは公表された MPI_CANCEL の目的（通信サブシステムに関与したプログラム領域を常に解放することができる）に少々反する。（ユーザへのアドバイスの終わり）

実装者へのアドバイス　多重スレッド環境では、 MPI_WAIT の呼出しは呼び出されたスレッドのみをブロックするようにしなければならず、スレッド・スケジューラが他のスレッドを実行するようスケジュールできるようにしなければならない。（実装者へのアドバイスの終わり）

MPI_TEST(request, flag, status)

入出力	request	通信要求（ハンドル）
出力	flag	もし操作が完了すれば真（論理型）
出力	status	ステータスオブジェクト（ステータス）

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)

MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
    LOGICAL FLAG
    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
```

MPI_TEST の呼出しでは、もし request によって識別される操作が終了していれば flag = true を返す。このような場合、ステータスオブジェクトは完了した操作についての情報を持つように設定される。通信オブジェクトがノンブロッキング送信やノンブロッキング受信によって生成された場合には、このオブジェクトは解放され、要求ハンドルには MPI_REQUEST_NULL が設定される。それ以外の場合は、呼出しは flag = false を返す。この場合にはステータスオブジェクトの値は定義されない。 MPI_TEST はローカルな操作である。

受信操作に対して返されるステータスオブジェクトは、3.2.5節で記述されたような方法でアクセスできる情報を持っている。送信操作に対するステータスオブジェクトは MPI_TEST_CANCELLED を呼び出すことによってアクセスできる情報を持っている。（3.8節参照）

MPI_TEST は、ヌルあるいは非アクティブ状態の request によって呼び出すことができる。このような場合には flag = true かつ status が空で返される。

関数 MPI_WAIT と MPI_TEST は受信と送信の両者を完了させるために使うことができる。

ユーザへのアドバイス ノンブロッキング MPI_TEST 呼出しを利用することにより、ユーザーは単一の実行スレッド内で別の作業をスケジューリングすることができる。 MPI_TEST を定期的に呼び出すことによってイベント駆動型のスレッドスケジューラをエミュレートすることができる。（ユーザへのアドバイスの終わり）

根拠 関数 MPI_TEST は関数 MPI_WAIT が返されるときと正確に同じ状態で flag = true で戻ってくる。この場合、この２つの関数は status に同じ値を返してくる。それゆえ、ブロック Wait は容易にノンブロッキング Test に置き換えることができる。（根拠の終わり）

例 **3.11** ノンブロッキング操作と MPI_WAIT の簡単な使用法

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank.EQ.0) THEN
    CALL MPI_ISEND(a(1), 10, MPI_REAL, 1, tag, comm, request, ierr)
    **** 待ち時間をマスクするために何か計算をする ****
    CALL MPI_WAIT(request, status, ierr)
ELSE
    CALL MPI_IRECV(a(1), 15, MPI_REAL, 0, tag, comm, request, ierr)
    **** 待ち時間をマスクするために何か計算をする ****
    CALL MPI_WAIT(request, status, ierr)
END IF
```

以下の操作を使用すれば、対応する通信が完了するのを待たずに要求オブジェクトを解放することができる。

MPI_REQUEST_FREE(request)

入出力	request	通信要求（ハンドル型）

```
int MPI_Request_free(MPI_Request *request)

MPI_REQUEST_FREE(REQUEST, IERROR)
    INTEGER REQUEST, IERROR
```

解放のための要求オブジェクトにマークを付け、 request を MPI_REQUEST_NULL に設定する。この要求に対応する継続中の通信は完了することができる。これが完了した後にのみ要求は解放される。

根拠　MPI_REQUEST_FREE という機構は、送信側の性能向上と便利さのために用意されている。（根拠の終わり）

ユーザへのアドバイス いったん MPI_REQUEST_FREE を呼び出すことにより要求が解放されると、 MPI_WAIT や MPI_TEST の呼出にし関連する通信が正常に完了したことをチェックすることはできない。また、もし通信中にエラーが起こると、エラーコードをユーザーに返すことができなくなる。このようなエラーは致命的なものとして取り扱うべきである。MPI_REQUEST_FREE を使用する場合に、操作が完了した時をどのように知るのかという疑問が起こる。プログラムの論理によっては、ある操作が完了したことをプログラムが知る別の方法があるかもしれない。このことは、 MPI_REQUEST_FREE の使用法を実際的なものにする。たとえば、プログラムの論理が受信側が送られたメッセージに応答を送るようなものであれば、応答の到着は、送信が完了し送信バッファが再利用可能であることを送信側に知らせることになり、アクティブな送信要求を解放することができる。受信側は、受信が完了し受信バッファが再利用可能であることを確認する手だてがなくなるので、アクティブな受信要求を解放すべきではない。（ユーザへのアドバイスの終わり）

例 **3.12** MPI_REQUEST_FREE を使用した例

```
CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank)
IF(rank.EQ.0) THEN
    DO i=1, n
      CALL MPI_ISEND(outval, 1, MPI_REAL, 1, 0, req, ierr)
      CALL MPI_REQUEST_FREE(req, ierr)
      CALL MPI_IRECV(inval, 1, MPI_REAL, 1, 0, req, ierr)
      CALL MPI_WAIT(req, status, ierr)
    END DO
ELSE    ! ランクが 1 の場合
    CALL MPI_IRECV(inval, 1, MPI_REAL, 0, 0, req, ierr)
    CALL MPI_WAIT(req, status)
    DO I=1, n-1
       CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0, req, ierr)
       CALL MPI_REQUEST_FREE(req, ierr)
       CALL MPI_IRECV(inval, 1, MPI_REAL, 0, 0, req, ierr)
       CALL MPI_WAIT(req, status, ierr)
    END DO
    CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0, req, ierr)
    CALL MPI_WAIT(req, status)
```

```
END IF
```

3.7.4 ノンブロッキング通信の意味論

ノンブロッキング通信の意味論は、3.5節の定義をノンブロッキングの場合に適した形に拡張することによって定義される。

順序 ノンブロッキング通信操作は通信を起動する呼出しの実行順に従って順序づけられる。3.5節の追越し禁止要求は、順序についてのこの定義によってノンブロッキング通信へ拡張される。

例 **3.13** ノンブロッキング操作に対するメッセージの順序

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (RANK.EQ.0) THEN
      CALL MPI_ISEND(a, 1, MPI_REAL, 1, 0, comm, r1, ierr)
      CALL MPI_ISEND(b, 1, MPI_REAL, 1, 0, comm, r2, ierr)
ELSE    ! ランクが 1 の場合
      CALL MPI_IRECV(a, 1, MPI_REAL, 0, MPI_ANY_TAG, comm, r1, ierr)
      CALL MPI_IRECV(b, 1, MPI_REAL, 0, 0, comm, r2, ierr)
END IF
CALL MPI_WAIT(r1,status)
CALL MPI_WAIT(r2,status)
```

たとえプロセス 1 がいずれかの受信を実行する前に両方のメッセージが送られていたとしても、プロセス 0 の最初の送信は、プロセス 1 の最初の受信に対応する。

進行 受信を完了させる MPI_WAIT の呼出しは、もし対応する送信が開始され、その送信に他の受信が応じなければ、最終的に終了して戻ってくる。特に対応する送信がノンブロッキングである時には、たとえ送信を完了させるための呼出しが送信側によって実行されなくても、受信は完了しなければならない。同様に、送信を完了させる MPI_WAIT は、もし対応する受信が開始されていれば、たとえその受信を完了させるための呼出しが実行されなくても、最終的に戻ってくる。

例 **3.14** 進行の意味論の実例

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (RANK.EQ.0) THEN
      CALL MPI_SSEND(a, 1, MPI_REAL, 1, 0, comm, ierr)
```

```
      CALL MPI_SEND(b, 1, MPI_REAL, 1, 1, comm, ierr)
ELSE    ! ランクが 1 の場合
      CALL MPI_IRECV(a, 1, MPI_REAL, 0, 0, comm, r, ierr)
      CALL MPI_RECV(b, 1, MPI_REAL, 0, 1, comm, ierr)
      CALL MPI_WAIT(r, status, ierr)
END IF
```

正しい MPI の実装においては、このコードはデッドロックを起こしてはならない。プロセス 0 の最初の同期送信は、たとえプロセス 1 がまだウエイトの呼出しの終了に到達していなくても、プロセス 1 が対応する（ノンブロッキング）受信を発行した後には完了しなければならない。そうして、プロセス 0 は継続して 2 番目の送信を実行し、プロセス 1 は実行を終了する。

もし受信を完了させる MPI_TEST が繰り返し同じ引数で呼ばれ、対応する送信が開始されていると、その送信に他の受信が応じなければ、この呼出しは最終的に flag = true を返す。もし送信を完了させる MPI_TEST が繰り返し同じ引数で呼ばれ、対応する受信が開始されていると、その受信に他の送信が応じなければ、この呼出しは最終的に flag = true を返す。

3.7.5 多重完了

特定のメッセージを待つのではなく、リストの中のすべての、あるいはいくつかの、あるいは任意の操作の完了を待つことができれば便利である。いくつかの操作のうちの 1 つの完了を待つために、 MPI_WAITANY あるいは MPI_TESTANY の呼出しを使用することができる。 MPI_WAITALL や MPI_TESTALL の呼出しを、リストの中のすべての保留中の操作を待つために使用することができる。 MPI_WAITSOME や MPI_TESTSOME の呼出しは、リストの中のすべての可能な操作を完了させるために使用することができる。

MPI_WAITANY (count, array_of_requests, index, status)

入力	count	リストの長さ（整数型）
入出力	array_of_requests	要求の配列（ハンドルの配列）
出力	index	完了した操作に対するハンドルの添字（整数型）
出力	status	ステータスオブジェクト（ステータス）

```
int MPI_Waitany(int count, MPI_Request *array_of_requests, int *index,
                MPI_Status *status)
MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR)
```

```
    INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
    IERROR
```

配列の中のアクティブな要求に対応する操作の 1 つが完了するまでブロックする。もし 1 つ以上の操作が可能で終了できるなら、その 1 つを任意に選ぶことができる。配列の中のその要求の添字が index に返され、完了した通信の状態が status に返される。（配列は C 言語においては 0 から、Fortran 言語においては 1 から添字が付けられる。）もしノンブロッキング通信操作によって割り当てられた要求ならば、それは解放され要求ハンドルには MPI_REQUEST_NULL が設定される。

リスト array_of_requests はヌルや非アクティブハンドルを含むことができる。もしリストがアクティブなハンドルを含んでいなければ（リストの長さがゼロかすべての要素がヌルか非アクティブ）、呼出しは index = MPI_UNDEFINED、status は空でただちに戻ってくる。

MPI_WAITANY(count, array_of_requests, index, status) の実行は MPI_WAIT(&array_of_requests[i], status) の実行と同じ結果になる。ここで i は（index の値が MPI_UNDEFINED でなければ）index によって返される値である。1 つのアクティブな要素を含む配列を持つ MPI_WAITANY は MPI_WAIT と同等である。

MPI_TESTANY(count, array_of_requests, index, flag, status)

入力	count	リストの長さ（整数型）
入出力	array_of_requests	requests の配列（ハンドルの配列）
出力	index	完了した操作の添字、もし何も完了しなかった時は MPI_UNDEFINED（整数型）
出力	flag	操作のうちの 1 つが完了したときは真（論理型）
出力	status	ステータスオブジェクト（ステータス）

```
int MPI_Testany(int count, MPI_Request *array_of_requests, int *index,
              int *flag, MPI_Status *status)

MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS, IERROR)
    LOGICAL FLAG
    INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
    IERROR
```

アクティブなハンドルに対応する操作の 1 つが完了しているか、あるいはどれも完了していないかのテスト。前者の場合には、flag = true が返され、配列の中のこの要求の添字が index に返され、その操作の状態が status に返される。もし要求がノンブロッキング通信の呼出しによっ

て割り当てられたものであれば、その要求は解放され、ハンドルには MPI_REQUEST_NULL が設定される。（配列はC言語の場合は0から、Fortran 言語の場合は1から添字が始まる。）後者の場合（完了した操作がない場合）、 flag = false が返され、 MPI_UNDEFINED という値が index に返され、 status は未定義となる。

配列はヌルあるいは非アクティブなハンドルを含むことができる。配列がアクティブなハンドルを含んでいなければ、呼出しは flag = false、 index = MPI_UNDEFINED でただちに戻り、status は未定義になる。

もし要求の配列がアクティブなハンドルを含んでいれば、 MPI_TESTANY(count, array_of_requests, index, status) の実行は、任意のある順序の i=0, 1 ,..., count-1 に対して、 1 つの呼出しが flag = true を返すか、すべてが失敗するまで MPI_TEST(&array_of_requests[i], flag, status) が実行された場合と同じ効果を持つ。前者の場合には index には最後の i の値が、後者の場合には MPI_UNDEFINED が設定される。 1 つのアクティブな要素を含む配列を持った MPI_TESTANY は MPI_TEST と同等である。

根拠 flag = true で戻る関数 MPI_TESTANY は、関数 MPI_WAITANY が戻るときとまったく同じ状態にある。この場合、２つの関数は残りのパラメータにも同じ値を返す。したがって、ブロック MPI_WAITANY はノンブロッキング MPI_TESTANY に用意に置き換えることができる。同じ関係がこの節で定義した他の Wait 関数と Test 関数の対に対しても成り立つ。（根拠の終わり）

MPI_WAITALL(count, array_of_requests, array_of_statuses)

入力	count	リストの長さ（整数型）
入出力	array_of_requests	要求の配列（ハンドルの配列）
出力	array_of_statuses	ステータスオブジェクトの配列（ステータスの配列）

```
int MPI_Waitall(int count, MPI_Request *array_of_requests,
              MPI_Status *array_of_statuses)

MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR)
    INTEGER COUNT, ARRAY_OF_REQUESTS(*)
    INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
```

リストの中のアクティブなハンドルに対応するすべての通信操作が完了までブロックし、これらすべての操作のステータスを返す。（このことは、リストの中のどのハンドルもアクティブでない場合も含む。）両方の配列は同じ数の有効なエントリーを持つ。 array_of_statuses の i 番

目のエントリーには i 番目の操作の返却ステータスが設定される。ノンブロッキング通信操作によって生成された要求は解放され、配列の中の対応するハンドルには MPI_REQUEST_NULL が設定される。リストはヌルや非アクティブなハンドルを含むこともできる。そのようなエントリーは、この呼出しによってそれぞれ空に設定される。

MPI_WAITALL(count, array_of_requests, array_of_statuses) が誤りなく実行されたときは、任意のある順序の i=0 ,..., count-1 に対して MPI_WAIT(&array_of_request[i], &array_of_statuses[i]) が実行されたときと同じ効果を待つ。長さ 1 の配列を持つ MPI_WAITALL は MPI_WAIT と同等である。

MPI_WAITALL の呼出しによって完了した 1 つあるいはそれ以上の通信が失敗した時には、それぞれの通信についての特定の情報を返すことが望ましい。このような場合、関数 MPI_WAITALL はエラーコード MPI_ERR_IN_STATUS を返し、それぞれのステータスの誤りフィールドに特定のエラーコードを返す。特定の通信が完了した場合はこのコードは MPI_SUCCESS であり、失敗した場合は他の特定のエラーコードである。あるいは、失敗でも完了でもない場合は MPI_ERR_PENDING であってもよい。関数 MPI_WAITALL はどの要求も誤りを含まなかったときは MPI_SUCCESS を返し、（不正な引数等の）他の理由で失敗したときは、他のエラーコードを返す。このような場合、ステータスの誤りフィールドは更新されない。

> 根拠 この設計は、アプリケーションにおける誤りの取扱いを簡素化してくれる。アプリケーションプログラムのコードは、誤りが起きたかどうか判定するために（1 つの）関数の結果をテストするだけでよい。誤りが起きたときにのみ個々のステータスを確かめる必要がある。（根拠の終わり）

MPI_TESTALL(count, array_of_requests, flag, array_of_statuses)

入力	count	リストの長さ（整数型）
入出力	array_of_requests	要求の配列（ハンドルの配列）
出力	flag	（論理型）
出力	array_of_statuses	ステータスオブジェクトの配列（ステータスの配列）

```
int MPI_Testall(int count, MPI_Request *array_of_requests, int *flag,
              MPI_Status *array_of_statuses)
MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, FLAG, ARRAY_OF_STATUSES, IERROR)
    LOGICAL FLAG
    INTEGER COUNT, ARRAY_OF_REQUESTS(*),
    ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
```

配列の中のアクティブなハンドルに対応するすべての通信が完了したときに flag = true を返す。（これはリストの中のハンドルがどれもアクティブでない場合を含む。）この場合、アクティブなハンドル要求に対応するステータスの各エントリーには対応する通信のステータスが設定される。もしノンブロッキング通信の呼出しによって要求が割り当てられていたときは、要求の割当は解放され、ハンドルには MPI_REQUEST_NULL が設定される。ヌルあるいは非アクティブなハンドルに対応するステータスの各エントリーには空が設定される。

それ以外の場合には、 flag = false が返され、どの要求も変更されず、ステータスのエントリーの値は定義されない。これはローカルな操作である。

MPI_TESTALL を実行中に起こった誤りは MPI_WAITALL の中の誤りとして処理される。

MPI_WAITSOME(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)

入力	incount	array_of_requests の長さ（整数型）
入出力	array_of_requests	要求の配列（ハンドルの配列）
出力	outcount	完了した要求の数（整数型）
出力	array_of_indices	完了した操作の添字の配列（整数の配列）
出力	array_of_statuses	完了した操作に対するステータスオブジェクトの配列（ステータスの配列）

```
int MPI_Waitsome(int incount, MPI_Request *array_of_requests, int *outcount,
              int *array_of_indices, MPI_Status *array_of_statuses)

MPI_WAITSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,
              ARRAY_OF_STATUSES, IERROR)
    INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
    ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
```

リストの中のハンドルに対応する操作のうち、少なくとも 1 つが終了するまで待つ。リスト array_of_requests からの要求のうち完了したものの数を outcount に返す。配列 array_of_requests の最初の outcount 個の位置にこれらの操作の添字を返す。（配列 array_of_requests 内の添字。配列は C 言語の場合は 0 から、 Fortran 言語の場合は 1 から添字が始まる。）配列 array_of_status の最初の outcount 個の位置に、完了したこれら操作に対するステータスを返す。完了した要求がノンブロッキング通信の呼出しによって割り当てられた場合は、その要求は解放され、対応するハンドルには MPI_REQUEST_NULL が設定される。

リストがアクティブなハンドルを含まない場合は、 outcount=MPI_UNDEFINED で呼出しはただちに戻ってくる。

もし MPI_WAITSOME によって完了した 1 つあるいはそれ以上の通信が失敗した時は、それぞれの通信についての特定の情報を返すことが望ましい。引数 outcount、 array_of_indices、 array_of_statuses は成功、あるいは失敗したすべての通信の完了を示すように調節される。呼出しはエラーコード MPI_ERR_IN_STATUS を返し、それぞれのステータスの返ってきた誤りフィールドは成功を示すように設定されるか、生じた特定の誤りを示すように設定される。この呼出しは、失敗に終わった要求がなかったときは MPI_SUCCESS を返し、他の理由（不正な引数など）によって失敗したときは他のエラーコードを返す。このような場合には、ステータスの誤りフィールドは更新されない。

MPI_TESTSOME(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)

入力	incount	array_of_requests の長さ（整数型）
入出力	array_of_requests	要求の配列（ハンドルの配列）
出力	outcount	完了した要求の数（整数型）
出力	array_of_indices	完了した操作の添字の配列（整数の配列）
出力	array_of_statuses	完了した操作に対するステータスオブジェクトの配列（ステータスの配列）

```
int MPI_Testsome(int incount, MPI_Request *array_of_requests, int *outcount,
             int *array_of_indices, MPI_Status *array_of_statuses)

MPI_TESTSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,
             ARRAY_OF_STATUSES, IERROR)
    INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
    ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
```

ただちに戻る場合を除いて、 MPI_WAITSOME のように振る舞う。操作が 1 つも完了していないときは outcount = 0 を返す。リストの中にアクティブなハンドルが無いときは outcount=MPI_UNDEFIN を返す。

MPI_TESTSOME はローカルな操作であり、 MPI_WAIT- SOME が 通信が完了するまでブロックされていたのに対し、少なくとも 1 つのアクティブなハンドルを含むリストを渡されると、ただちに戻ってくる。この呼出しはどちらも公平の要求を満たす。すなわち、 MPI_WAITSOME あるいは MPI_TESTSOME に渡された要求のリストに受信要求が繰り返し現れ、対応する送信が発行されると、送信が他の受信によって満たされなければ、受信は最終的には成功する。送信要求についても同様である。

MPI_TESTSOME の実行中に起こった誤りは、 MPI_WAITSOME に対するのと同じように取り扱われる。

ユーザへのアドバイス　MPI_TESTSOME の使用は MPI_TESTANY の使用よりおそらくもっと有効である。前者は完了したすべての通信についての情報を返すが、後者では完了したそれぞれの通信について新たな呼出しが必要となる。

複数のクライアントを持つサーバでは、クライアントに待ちぼうけを食わせることがないように MPI_WAITSOME を利用することができる。クライアントはサーバにサービス要求のメッセージを送る。サーバはそれぞれのクライアントに対する受信要求を付けて MPI_WAITSOME を呼び、完了したすべての受信を取り扱う。もし MPI_WAITANY の呼出しを代わりに使うと、他のクライアントからの要求が常に最初に割り込んでしまい、あるクライアントが死んでしまうということが起こりうる。（ユーザへのアドバイスの終わり）

実装者へのアドバイス　MPI_TESTSOME は保留中の通信を可能な限りたくさん完了させなければならない。（実装者へのアドバイスの終わり）

例 **3.15** クライアントサーバ・コード （要求処理欠落が起こりうる）

```
CALL MPI_COMM_SIZE(comm, size, ierr)
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank > 0) THEN          ! クライアント・コード
    DO WHILE(.TRUE.)
       CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
       CALL MPI_WAIT(request, status, ierr)
    END DO
ELSE          ! ランク 0 -- サーバ・コード
       DO i=1, size-1
          CALL MPI_IRECV(a(1,i), n, MPI_REAL, 0, tag,
                    comm, request_list(i), ierr)
       END DO
       DO WHILE(.TRUE.)
          CALL MPI_WAITANY(size-1, request_list, index, status, ierr)
          CALL DO_SERVICE(a(1,index))  ! 1つのメッセージを取り扱う
          CALL MPI_IRECV(a(1, index), n, MPI_REAL, 0, tag,
                     comm, request_list(index), ierr)
       END DO
```

```
END IF
```

例 **3.16** 同じコード。ただし MPI_WAITSOME を使用。

```
CALL MPI_COMM_SIZE(comm, size, ierr)
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank > 0) THEN          ! クライアント・コード
    DO WHILE(.TRUE.)
       CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
       CALL MPI_WAIT(request, status, ierr)
    END DO
ELSE          ! ランク 0 -- サーバ・コード
    DO i=1, size-1
       CALL MPI_IRECV(a(1,i), n, MPI_REAL, 0, tag,
                      comm, request_list(i), ierr)
    END DO
    DO WHILE(.TRUE.)
       CALL MPI_WAITSOME(size, request_list, numdone,
                         index_list, status_list, ierr)
       DO i=1, numdone
          CALL DO_SERVICE(a(1, index_list(i)))
          CALL MPI_IRECV(a(1, index_list(i)), n, MPI_REAL, 0, tag,
                         comm, request_list(i), ierr)
       END DO
    END DO
END IF
```

3.8 Probe および キャンセル

MPI_PROBE および MPI_IPROBE の操作により、実際にメッセージを受け取ることなく、送信されたメッセージが確認される。したがって、ユーザは、 probe 操作により得られた情報に基づいて、メッセージを受け取る方法を決定することが可能である。（基本的には、この情報は、status により返されるものである）特に、あらかじめ確認されたメッセージの長さに応じて、受信バッファの記憶領域を配置することができる。

MPI_CANCEL の操作により、保留中の通信がキャンセルされる。この操作は、これら保留中の通信を一掃するために必要である。送信又は受信の実行は、ユーザーリソース（送受信バッファ）の消費に直接関連している。このキャンセルの操作は、効果的にこれらのリソースを解放するために、必要とされるものである。

MPI_IPROBE(source, tag, comm, flag, status)

入力	source	送信元のランク、または MPI_ANY_SOURCE(整数型)
入力	tag	タグ値、または MPI_ANY_TAG (整数型)
入力	comm	コミュニケータ (ハンドル)
出力	flag	(論理型)
出力	status	ステータスオブジェクト (ステータス)

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,
               MPI_Status *status)

MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)
    LOGICAL FLAG
    INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
```

MPI_IPROBE(source, tag, comm, flag, status) は、受信されたメッセージの中に、引数 source や tag,comm により指定されたパターンに対応するものがあった場合、flag = true を返す。この呼び出しによるメッセージは、プログラムの中で同じ時点で実行される MPI_RECV(..., source, tag, comm, status) によって受信されるメッセージと一致し、MPI_RECV() によって返される場合と同じ値を status に返す。そうでない場合、この呼び出しは flag = false を返し、status を未定義のままにする。

3.2.5節で説明しているように、MPI_IPROBE が、flag = true を返した場合、probe されたメッセージの送信元やタグ、長さを見つけるため、ステータス・オブジェクトの内容が、続いてアクセスされる。

probe 操作の後、他のメッセージが受信されていない場合、MPI_IPROBE の操作による status の中で返された送信元やタグを持ち、続いて同じ状態で実行された受信では、probe されたものと同じメッセージを受け取る。また、受信プロセスが複数スレッドの場合、最終の状態を保持することは、ユーザの責任である。

MPI_PROBE の source 引数は、MPI_ANY_SOURCE とする設定が可能であり、tag 引数にも同様に MPI_ANY_TAG が設定可能である。そのため、プログラムは、任意の送信元および（または）タグによって、複数のメッセージについて probe することが出来る。ただし、これらの実

行においては、 comm 引数に対して、特定された通信条件が与えられなければならない。

一旦メッセージを probe した後では、あらためてすぐにメッセージを受信する必要はなく、また受信する以前に、何度でも同じメッセージを probe することが可能である。

MPI_PROBE(source, tag, comm, status)

入力	source	送信元のランク、または MPI_ANY_SOURCE (整数型)
入力	tag	タグ値、または MPI_ANY_TAG (整数型)
入力	comm	コミュニケータ (ハンドル)
出力	status	ステータスオブジェクト (ステータス)

```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)

MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)
    INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
```

対応するメッセージを発見した場合にのみ、ブロッキング呼出しとして戻る点を除いて、MPI_PROBE の操作は、 MPI_IPROBE と同様に作用する。

MPI における MPI_PROBE や MPI_IPROBE の実装では、以下の動作を保証する必要がある。あるプロセスにより MPI_PROBE の呼出しが実行され、複数のプロセスにより probe されたメッセージに対応する送信が起動された場合、その送信メッセージが、もう 1 つの同時に動作する受信操作（これは、 probe のプロセスにおいて他の別のスレッドにより実行されるものである。）によって受信されない場合に限り、 MPI_PROBE に対する呼出しが戻る。同様に、プロセスが MPI_IPROBE の状態で待機し、対応するメッセージが起動された場合には、その送信メッセージがもう 1 つの同時に動作する受信操作によって受信されない場合に限り、 MPI_IPROBE の呼出しに対していかなる場合にも flag = true が返される。

例 **3.17** 送信されたメッセージを待つためのブロッキング probe の使用

```
      CALL MPI_COMM_RANK(comm, rank, ierr)
      IF (rank.EQ.0) THEN
           CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
      ELSE IF(rank.EQ.1) THEN
           CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
      ELSE   ! ランクが 2 の場合
          DO i=1, 2
             CALL MPI_PROBE(MPI_ANY_SOURCE, 0,
```

```
                              comm, status, ierr)
               IF (status(MPI_SOURCE) = 0) THEN
100                 CALL MPI_RECV(i, 1, MPI_INTEGER, 0, 0, status, ierr)
               ELSE
200                 CALL MPI_RECV(x, 1, MPI_REAL, 1, 0, status, ierr)
               END IF
            END DO
        END IF
```

それぞれのメッセージは、正しい型で受信されるものとする。

例 **3.18** 前の例と同様のプログラム（ただしこの例では問題がある）。 has a problem.

```
        CALL MPI_COMM_RANK(comm, rank, ierr)
        IF (rank.EQ.0) THEN
             CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
        ELSE IF(rank.EQ.1) THEN
             CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
        ELSE
            DO i=1, 2
               CALL MPI_PROBE(MPI_ANY_SOURCE, 0,
                              comm, status, ierr)
               IF (status(MPI_SOURCE) = 0) THEN
100                 CALL MPI_RECV(i, 1, MPI_INTEGER, MPI_ANY_SOURCE,
                                  0, status, ierr)
               ELSE
200                 CALL MPI_RECV(x, 1, MPI_REAL, MPI_ANY_SOURCE,
                                  0, status, ierr)
               END IF
            END DO
        END IF
```

例では、 100 および 200 のラベルを持つ文にある２つの受信呼び出しにおいて、 source 引数に MPI_ANY_SOURCE を使用するように、例 3.17に若干修正が加えられている。このプログラムはこのままでは、正しく動作しない。すなわち、これらの受信操作が、あらかじめ MPI_PROBE の呼び出しによって probe されたメッセージとは異なるメッセージを受け取る可能性がある。

実装者へのアドバイス MPI_PROBE(source, tag, comm, status) の呼び出しは、同じ地点で実行する MPI_RECV(..., source, tag, comm, status) の呼び出しによって受け取られるであろうメッセージと一致するものである。このメッセージにおいて、送信元に s、タグに t、コミュニケータに c を持つものとする。このとき、 probe 呼び出しにおける タグ引数が、値 MPI_ANY_TAG をもつ場合には、 probe されるメッセージは、任意のタグとコミュニケータ c を持つ 、送信元 s からの最も早く受信した保留中のメッセージである。いかなる場合でも、 probe されたメッセージは、タグ t とコミュニケータ c を持つ送信元 s からの最も早く受信した保留中のメッセージである。（これは、メッセージの順序を保持するように、受信メッセージである）。（実装者へのアドバイスの終わり）

MPI_CANCEL(request)

入力 request 通信要求 (ハンドル)

```
int MPI_Cancel(MPI_Request *request)

MPI_CANCEL(REQUEST, IERROR)
    INTEGER REQUEST, IERROR
```

MPI_CANCEL の呼出しは、取消しを行う保留中のノンブロッキング通信操作（送信または受信）を選択する。取消しの呼び出しは、ローカルに実行される。この呼出しは、目的とする通信が実際に取り消される以前に、可能な限りすばやく戻る。取消しを選択するための通信は、MPI_REQUEST_FREE、MPI_WAIT 又は MPI_TEST （または任意の対応する操作）の呼出しを利用して完了されている必要がある。

ある通信が取消しの対象とされた場合、他のプロセスの状態に関わらず、選択された通信に対する MPI_WAIT の呼出しが戻ることが保証される。（したがって、 MPI_WAIT はローカル関数として動作することになる）同様に、 busy wait ループにおいて、取消しの対象とされた通信に対して MPI_TEST を繰り返し呼び出した場合、この呼出しは、最終的には成功することになる。

MPI_CANCEL は、非持続的な要求を用いる場合と同様な方法で、持続的な要求（第 3.9節参照）を用いた通信を取り消すためにも、利用することが可能である。取消し操作の成功は、動作中の通信を取り消すことになるが、要求自身を取り消すわけではない。 MPI_CANCEL が呼び出され、続いて MPI_WAIT や MPI_TEST が呼び出された場合、取消し要求は停止し、新しい通信を起動することが可能になる。取消し操作の成功は、動作中の通信を取り消すことになるが、要求自身を取り消すわけではない。

バッファ送信の取消しが成功すると、保留中のメッセージによって確保されていたバッファ領域は解放される。

通信に対しては、取消しが成功するか、動作が成功するかのどちらかであり、いずれもが成功することはない。ある送信が取消しの対象として選択された場合、送信が正常に完了する場合（このときは、送信されたメッセージは、送信先のプロセスに受信される）または、送信の取消しが成功する場合（このときは、送信先においてメッセージの全てが受信されない）、いずれかになる。したがって、取り消された送信に対応する受信に対しては、別の送信が対応する必要がある。また、ある受信が取消しの対象として選択された場合、送信が正常に完了する場合、または、受信の取消しが成功する場合（このときは、受信バッファの全てが更新されない）、いずれかになる。

操作が取り消された場合、通信を完了させる操作のステータス引数に対して、取消しの結果についての情報が返される。

MPI_TEST_CANCELLED(status, flag)

入力	status	ステータスオブジェクト（ステータス）
出力	flag	(論理型)

```
int MPI_Test_cancelled(MPI_Status status, int *flag)

MPI_TEST_CANCELLED(STATUS, FLAG, IERROR)
    LOGICAL FLAG
    INTEGER STATUS(MPI_STATUS_SIZE), IERROR
```

ステータスオブジェクトに関連する通信の取消しが成功した場合には、flag = true が返される。この場合、count や tag 等の status 中の他のすべてのフィールドは、未定義になる。そうでない場合には、flag = false が返される。受信操作が取り消された可能性がある場合、返却ステータスでの他の項目の情報について確認する前に、受信操作が取り消されたかどうかを調べるために、まずはじめに MPI_TEST_CANCELLED を呼び出す必要がある。

ユーザへのアドバイス 取消しは、コストのかかる操作になる可能性があるので、例外的な場合にのみ利用すべきである。（ユーザへのアドバイスの終わり）

実装者へのアドバイス　送信操作で“eager”プロトコル（対応する受信が実行される前に、受信側に対してデータを転送する）を用いた場合、この送信の取消しを行うには、確保されていたバッファの領域を解放するために、対象とされる受信側との通信が必要になる場合がある。また、システムによっては、この通信が、対象とされる受信側に対して、割り込みを行う必要がある。なお、実装に当たり、以下に示す点を注意する必要がある。まず、通信が MPI_CANCEL の実行を必要としている状態では、この操作が他のプロセスにより実行されたコードに依存していないため、ローカルな操作と見なすことが出来る。

次に、あるプロセスが他のプロセスを要求している場合、通信はアプリケーションに対して透過的になっている必要がある。（そのため、割り込みや割り込みハンドラが必要とされる）（実装者へのアドバイスの終わり）

3.9 持続的な通信要求

並列計算のループの内部において、同一の引数並びを持つ通信が、しばしば繰り返し実行される。このような場合、通信での引数並びを持続的な通信要求と統合すること、すなわち、メッセージの起動と完了の要求を繰り返し行うことによって、これらの通信の効率を最適化することが可能である。つまり、このようにして生成された持続的な通信要求は、通信ポートまたは “ハーフチャネル” と見なすことが出来る。これらの通信では、送信ポートと受信ポートとの間に呼びだし形式が完成していないため、標準的なチャネルのような完全な機能が提供されているわけではない。このような通信を行うことにより、プロセスと通信制御機構との間に行われる通信に関して、オーバーヘッドの軽減が可能である。一方、ある通信制御機構と他の通信制御機構との通信に関しては、このようなオーバーヘッドの軽減は期待できない。持続的な通信要求による送信させるメッセージは、必ずしも、持続的な要求による受信操作によって受信される必要はなく、また、逆の状況であった場合にも、送信側や受信側に制限を与えるものではない。

１つの持続的な通信要求は、以下の４つの呼出しのうちの１つを利用して生成される。なお、これらの呼出しでは、通信は実行されない。

MPI_SEND_INIT(buf, count, datatype, dest, tag, comm, request)

入力	buf	送信バッファの先頭アドレス (選択型)
入力	count	送信される要素の数 (整数型)
入力	datatype	各要素の型 (ハンドル)
入力	dest	送信先のランク (整数型)
入力	tag	メッセージタグ (整数型)
入力	comm	コミュニケータ (ハンドル)
出力	request	通信要求 (ハンドル)

```
int MPI_Send_init(void* buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm, MPI_Request *request)

MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
```

```
    INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

標準モードでの送信操作のための、持続的な通信要求を生成し、送信操作での全ての引数をそれに割り当てる。

MPI_BSEND_INIT(buf, count, datatype, dest, tag, comm, request)

入力	buf	送信バッファの先頭アドレス (選択型)
入力	count	送信される要素の数 (整数型)
入力	datatype	各要素の型 (ハンドル)
入力	dest	送信先のランク (整数型)
入力	tag	メッセージタグ (整数型)
入力	comm	コミュニケータ (ハンドル)
出力	request	通信要求 (ハンドル)

```
int MPI_Bsend_init(void* buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm, MPI_Request *request)
MPI_BSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

バッファモードでの送信操作のための、持続的な通信要求を生成する。

MPI_SSEND_INIT(buf, count, datatype, dest, tag, comm, request)

入力	buf	送信バッファの先頭アドレス (選択型)
入力	count	送信される要素の数 (整数型)
入力	datatype	各要素の型 (ハンドル)
入力	dest	送信先のランク (整数型)
入力	tag	メッセージタグ (整数型)
入力	comm	コミュニケータ (ハンドル)
出力	request	通信要求 (ハンドル)

```
int MPI_Ssend_init(void* buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm, MPI_Request *request)
```

```
MPI_SSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

同期モード送信操作のための持続的な通信オブジェクトを生成する。

MPI_RSEND_INIT(buf, count, datatype, dest, tag, comm, request)

入力	buf	送信バッファの先頭アドレス (選択型)
入力	count	送信される要素の数 (整数型)
入力	datatype	各要素の型 (ハンドル)
入力	dest	送信先のランク (整数型)
入力	tag	メッセージタグ (整数型)
入力	comm	コミュニケータ (ハンドル)
出力	request	通信要求 (ハンドル)

```
int MPI_Rsend_init(void* buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm, MPI_Request *request)
MPI_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

レディモード送信操作のための持続的な通信オブジェクトを生成する。

MPI_RECV_INIT(buf, count, datatype, source, tag, comm, request)

出力	buf	受信バッファの先頭アドレス (選択型)
入力	count	受信された要素の数 (整数型)
入力	datatype	各要素の型 (ハンドル)
入力	source	送信元のランクまたは MPI_ANY_SOURCE (整数型)
入力	tag	メッセージタグまたは MPI_ANY_TAG (整数型)
入力	comm	コミュニケータ (ハンドル)
出力	request	通信要求 (ハンドル)

```
int MPI_Recv_init(void* buf, int count, MPI_Datatype datatype, int source,
```

```
              int tag, MPI_Comm comm, MPI_Request *request)

MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
```

受信操作のための持続的な通信要求を生成する。ユーザが MPI_RECV_INIT に対して引数を渡すことにより、受信バッファに書き込み許可を与えるために、送信される引数 buf は、出力という値を持つ

1 つの持続的な通信要求は、それが生成された後では動作を停止する。つまり、動作している通信は、持続的な通信要求に対応づけられることはない。

持続的な通信要求を用いた通信（送信又は受信）は、 MPI_START 関数により起動される。

MPI_START(request)

入出力	request	通信要求（ハンドル）

```
int MPI_Start(MPI_Request *request)

MPI_START(REQUEST, IERROR)
    INTEGER REQUEST, IERROR
```

ここでの引数 request は、上記の 5 種類の呼出しのうちで実行されたものより返されるハンドルである。なお、対応する要求は、この状態では無効である。ただし、一度呼出しが実行された後では、要求は有効になる。

この要求が、レディモードの状態での送信であった場合、呼出しが実行される前に、対応する受信が準備されている必要がある。呼出しの後や、この操作が完了するまでは、目的の通信バッファは、参照される必要はない。

第 3.7節で説明したような、ノンブロッキング型の通信操作と同様な意味を持つような呼出しは、ローカルな通信である。つまり、 MPI_SEND_INIT により生じた要求を伴った MPI_START の呼出しは、 MPI_ISEND の呼出しと同様な方法によって、通信が開始される。また、 MPI_BSEND_INIT により生じた要求を伴った MPI_START の呼出しは、 MPI_IBSEND の呼出しと同様な方法によって、通信が開始される。

MPI_STARTALL(count, array_of_requests)

入力	count	リストの長さ (整数型)
入出力	array_of_requests	要求の配列 (ハンドルの配列)

```
int MPI_Startall(int count, MPI_Request *array_of_requests)

MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERROR)
    INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERROR
```

上記の呼出しにより、 array_of_requests における要求に対応する全ての通信は、開始される。 MPI_STARTALL(count, array_of_requests) の呼出しは、任意の順序で i=0 ,..., count-1 に渡って繰り返し実行される MPI_START (&array_of_requests[i]), の呼出しと同じ機能を持っている。

MPI_START や MPI_STARTALL の呼出しにより開始された通信は、 MPI_WAIT や MPI_TEST の呼出し、または 3.7.5節で説明された構造型関数の呼出しによって完了する。これらの通信の要求は、上記の呼出しが完了したあとで動作を停止する。また、これらの要求では、領域の割り当てが解放されることはない。そして、 MPI_START や MPI_STARTALL の呼出しによって新たに起動することが可能である。

なお、持続的な要求では、 MPI_REQUEST_FREE の呼出し（3.7.3節）によって、領域の割り当てが解放される。

MPI_REQUEST_FREE の呼出しは、持続的な要求が生成された後であれば、プログラムの任意の時点で、実行することが可能である。ただし、この要求は、動作が停止した後でのみ割り当て解放が実行される。つまり、動作中の受信要求に対しては、領域解放は実行されない。それ以外の場合、受信要求が完了したかどうかを確認することは、実行できない。一般的に、これらの受信要求の動作が完了した場合に、要求を解放することが望ましい。この規則に従っている場合には、この節で説明した関数は、以下の形式の記述で起動される。

Create (**Start Complete**)* **Free** , ここで $*$ は 0 回または 1 回以上の繰り返しを示す。

同じ通信オブジェクトが、複数の同時実行されるスレッド内で使用される場合、正しい手順に従うようにするための調整は、ユーザーにより管理される。

MPI_START により起動された送信操作は、任意の受信操作と対応づけることが可能であり、同様に、 MPI_START により起動された受信操作は、任意の送信操作によって生成されるメッセージを受信することが可能である。

3.10 送受信

送受信操作は、特定の送信先に対するメッセージの送信動作と、他のプロセスからの別のメッセージを受信する動作を 1 つに組み合わせたものである。なお、送信側と受信側の両者は、同一にすることも可能である。この送受信操作は、プロセスの連鎖に従って連続したシフト操作を実行するときに、大変有効である。ブロック化された送信や受信が、このようなシフト操作に利用される場合、デッドロックを引き起こすような周期的な依存関係を避けるためにも、送信と受信の順序を正しく保つ必要がある。（例えば、偶数のプロセスは送信をしてから受信を行い、奇数のプロセスであれば、はじめに受信を行ってから送信を行う。）このような送受信操作が利用される場合、これらの問題は通信サブシステムにより、調節される。様々な論理的なトポロジー上でシフト操作を実行するためには、第 6 章で説明するような関数と結合する形式において、送受信操作が利用可能になる。さらに、送受信操作は、リモートプロシージャ呼出しを実装する場合において、非常に有効である。

送受信操作により送信されたメッセージは、通常の受信操作によって受信されるか、または、 probe 操作によって、事前に確認される。同様に、送受信操作は、通常の送信操作によって送信されたメッセージを受信することも可能である。

MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)

入力	sendbuf	送信バッファの先頭アドレス (選択型)
入力	sendcount	送信バッファ内の要素数 (整数型)
入力	sendtype	送信バッファ内の要素の型 (ハンドル)
入力	dest	送信先のランク (整数型)
入力	sendtag	送信タグ (整数型)
出力	recvbuf	受信バッファの先頭アドレス (選択型)
入力	recvcount	受信バッファ内の要素数 (整数型)
入力	recvtype	受信バッファ内の要素の型 (ハンドル)
入力	source	送信元のランク (整数型)
入力	recvtag	受信タグ (整数型)
入力	comm	コミュニケータ (ハンドル)
出力	status	ステータスオブジェクト (ステータス)

```
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
```

```
              int dest, int sendtag, void *recvbuf, int recvcount,
              MPI_Datatype recvtype, int source, MPI_Datatype recvtag,
              MPI_Comm comm, MPI_Status *status)

MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF,
              RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE,
    SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
```

上記の操作により、ブロック化された送受信操作が実行される。なお、送信と受信の両方で、同じコミュニケータが使われているが、タグは異なったものすることが望ましい。ただし、送信バッファと受信バッファは独立したものであり、異なった長さとデータ型を持つことが出来る。

MPI_SENDRECV_REPLACE(buf, count, datatype, dest, sendtag, source, recvtag, comm, status)

入出力	buf	送信兼受信バッファの先頭アドレス (選択型)
入力	count	送信兼受信バッファ内の要素数 (整数型)
入力	datatype	送信兼受信バッファ内の要素の型 (ハンドル)
入力	dest	送信先のランク (整数型)
入力	sendtag	送信メッセージタグ (整数型)
入力	source	送信元のランク (整数型)
入力	recvtag	受信メッセージタグ (整数型)
入力	comm	コミュニケータ (ハンドル)
出力	status	ステータスオブジェクト (ステータス)

```
int MPI_Sendrecv_replace(void* buf, int count, MPI_Datatype datatype,
              int dest, int sendtag, int source, int recvtag, MPI_Comm comm,
              MPI_Status *status)

MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,
              COMM, STATUS, IERROR)
    <type> BUF(*)
```

```
INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM,
STATUS(MPI_STATUS_SIZE), IERROR
```

上記の操作により、ブロック化された送受信が実行される。ここでは、送信操作と受信操作の両方に対して同じバッファが利用される。そのため、送信されたメッセージは受信されたメッセージに置換されてしまう。

送受信操作の機能は、同時に動作している、送信を実行するスレッドと受信を実行するスレッドの２つが呼び出される様な場合に対応して、これら２つのスレッドを結合した場合に同じような情報を受け取ることである。

実装者へのアドバイス　異なるメッセージの “置換” に対しては、中間的なバッファ操作の追加が必要になる。（実装者へのアドバイスの終わり）

3.11　ヌルプロセス

多くの場合、通信を行う場合に、 “ダミー” の送信元と送信先を指定することは、有用である。これによって、非循環型のシフト操作を送受信操作の呼出しで事項する場合などにおいて、境界を取り扱うためのプログラムを単純にすることが出来る。

呼出しにおいて、送信元と送信先の引数が必要な場合には、どのような場合であっても、特別な値 MPI_PROC_NULL をランクの変わりに用いることが可能である。プロセス MPI_PROC_NULL による通信では、何も実行されない。この場合、 MPI_PROC_NULL に対する送信は成功し、ただちに戻る。また、 MPI_PROC_NULL による受信も成功し、受信バッファに対しての修正が行われることなく、ただちに戻る。 source = MPI_PROC_NULL として受信が実行される場合には、ステータスオブジェクトは、 source = MPI_PROC_NULL、 tag = MPI_ANY_TAG そして count = 0 を返す。

3.12　派生データ型

ここまでの一対一通信は全て同じ型の要素列を含む連続バッファのみを扱って来た。このことは二つの大きな制限を含んでいた。ひとつは異なるデータ型 (例えば、整数で要素数を持ちその後に実数列が続くようなデータ) を含むメッセージを送ろうとする場合と、今ひとつは、不連続なデータ (例えば行列の一部) を送ろうとする場合である。解のひとつは、送信側で不連続なデータを連続なバッファにパックし、受信側でアンパックすることである。これは通信サブシステムがスキャッタ - ギャザの機能を持っている場合ですら余分なメモリコピーが送信側、受信側双方で必要になるという欠点を持つ。代わりに、 MPI はより汎用的な、データ型の混在や、不連続な通信バッファを指定できる機構を提供する。データを転送する前に連続バッファにまずパックする必要があるのか、或は直接集めることが出来るのかは実装依存である。

ここで提供される機構によって、コピーせずに、様々な形や大きさを持つオブジェクトを送ることが出来る。 MPI ライブラリはホスト言語で宣言されたオブジェクトを認識できるとは限らない。従って、構造体や配列の一部を転送しようとすると、その構造体や配列の一部の定義を模倣した定義を MPI に与える必要がある。これらの機能はライブラリのデザイナがホスト言語で定義されたオブジェクトを、それらの定義をシンボル・テーブルやドープ・ベクトルの形に解読することで、転送する通信関数を定義するために用いることが出来る。この様な高レベルの通信関数は MPI にはない。

この節で説明するコンストラクタで基本データ型から作られた派生データ型で、これまで用いられて来た基本データ型を置き換えることで、より汎用的な通信バッファを指定できる。派生データ型は再帰的に定義することが出来る。

A 汎用データ型 は次の二つの事項を指定する不透明オブジェクトである。

- 基本データ型の並び。
- 整数 (バイト) 変位の並び。

変位は正数や異なる値や昇順である必要はない。そのため、項目の順序は項目が格納された順序と一致する必要はなく、また、ひとつのアイテムが繰り返し現れてもよい。この様な並びの対 (或は対の並び) を型マップと呼ぶ。基本データ型の並び (変位は考慮しない) は、そのデータ型の型仕様である。

$$Typemap = \{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

この様な型マップで、 $type_i$ は基本データ型で $disp_i$ は変位である。

$$Typesig = \{type_0, ..., type_{n-1}\}$$

これは上記の型マップに対応する型仕様である。この型マップは、基底アドレス buf と共に用いて、通信バッファを指定する。この例では n 個のエントリを持ち、 i 番目のエントリのアドレスは $buf + disp_i$ で型は $type_i$ になる。この様なバッファから作られたメッセージは、 n 個の値を持ち、 $Typesig$ で表された型からなる。

汎用データ型のハンドルは基本データ型の代わりに送受信操作の引数として用いることが出来る。 MPI_SEND(buf, 1, datatype,...) は基底アドレス buf と datatype で表される汎用データ型とで定義される送信バッファを使う。このとき datatype 引数で決まる型仕様を持つメッセージが生成される。 MPI_RECV(buf, 1, datatype,...) は基底アドレス buf と datatype で表される汎用データ型とで定義される受信バッファを使う。

汎用データ型は全ての送受信操作で利用できる。 3.12.5節で第二引数 count が 1 より大きな場合について論じる。

3.2.2節に示される基本データ型は汎用データ型の特殊なものである。つまり、 MPI_INT は {(int, 0)}(int,0)、 1 つの int 型のエントリを持ちその変位が 0 であるような、型マップを持つ定義済みのデータ型である。他の基本データ型についても同様である。

データ型の範囲はこのデータ型のエントリによって占められる最初の 1 バイトから最後の 1 バイトまでの区間によって定義され、アラインメントを満たすように丸められる。

$$Typemap = \{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

の様な型マップは、

$$\begin{aligned} lb(Typemap) &= \min_j disp_j, \\ ub(Typemap) &= \max_j (disp_j + sizeof(type_j)), \text{ かつ} \\ extent(Typemap) &= ub(Typemap) - lb(Typemap) + \epsilon. \end{aligned} \tag{3.1}$$

の様な範囲を持つ。もし $type_i$ が k_i の倍数バイトのアラインメントを必要とするなら、ϵ は $extent(Typemap)$ を一番近い $max_i k_i$ の倍数バイトに丸めるのに必要な正数値となる。

例 **3.19** $Type = \{(\text{double}, 0), (\text{char}, 8)\}$ (変位 0 で double がひとつ、続けて変位 8 で char がある。) を仮定する。さらに double は 8 の倍数のアラインメントが必要であるとする。すると、このデータ型の範囲は 16 になる。 (9 が次の 8 の倍数 (16) に切り上げられる。) 文字の直後に double が続くようなデータ型の範囲も 16 である。

根拠　範囲の定義は構造体の配列のアラインメントをあわせるのに必要なパディングを前提としている。より厳密な範囲の管理は 3.12.3節に示す。このような厳密な管理は、例えば union 型が用いられるようなこの前提が成り立たない様な場合に必要となる。（根拠の終わり）

3.12.1 データ型コンストラクタ

Contiguous　最も単純なデータ型コンストラクタは連続なデータ型配置が出来る MPI_TYPE_CONTIGUOUS である。

MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)

入力	count	繰り返し回数 (非負の整数型)
入力	oldtype	旧データ型 (ハンドル)
出力	newtype	新データ型 (ハンドル)

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
              MPI_Datatype *newtype)
```

```
MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
```

newtype は連続する oldtype の count 個のコピーからなるデータ型である。「連続性」は連続したコピーの大きさである範囲によって定義される。

例 **3.20** oldtype の型マップが $\{(\text{double}, 0), (\text{char}, 8)\}$, で、範囲が 16、そして count $= 3$ であるとする。このとき newtype の型マップは次のようになる。

$$\{(\text{double}, 0), (\text{char}, 8), (\text{double}, 16), (\text{char}, 24), (\text{double}, 32), (\text{char}, 40)\};$$

即ち、 double と char が交互に、それぞれ $0, 8, 16, 24, 32, 40$ の変位で現れる。

一般的に、 oldtype 型マップが次のようで、

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

範囲が ex であるならば、 newtype は count · n このエントリを持つ次のような型マップを持つ。

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1}), (type_0, disp_0 + ex), ..., (type_{n-1}, disp_{n-1} + ex),$$

$$..., (type_0, disp_0 + ex \cdot (\text{count} - 1)), ..., (type_{n-1}, disp_{n-1} + ex \cdot (\text{count} - 1))\}.$$

Vector MPI_TYPE_VECTOR は等間隔に並んだデータ型の複製でデータ型を作ることが出来るより汎用なコンストラクタである。個々のブロックは同じ数の元のデータ型の連なりからなる。ブロックの間隔は元のデータ型の範囲の倍数となる。

MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype)

入力	count	繰り返し回数 (非負の整数型)
入力	blocklength	個々のブロックの要素数 (非負の整数型)
入力	stride	個々のブロックの先頭の間隔の要素数 (整数型)
入力	oldtype	旧データ型 (ハンドル)
出力	newtype	新データ型 (ハンドル)

```
int MPI_Type_vector(int count, int blocklength, int stride,
            MPI_Datatype oldtype, MPI_Datatype *newtype)
```

```
MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR
```

例 **3.21** 再び型マップが $\{(\mathsf{double}, 0), (\mathsf{char}, 8)\}$, で、範囲が 16 であるような oldtype を仮定する。MPI_TYPE_VECTOR(2, 3, 4, oldtype, newtype) の呼び出しは、次のような型マップを持つデータ型を作る。

$$\{(\mathsf{double}, 0), (\mathsf{char}, 8), (\mathsf{double}, 16), (\mathsf{char}, 24), (\mathsf{double}, 32), (\mathsf{char}, 40),$$

$$(\mathsf{double}, 64), (\mathsf{char}, 72), (\mathsf{double}, 80), (\mathsf{char}, 88), (\mathsf{double}, 96), (\mathsf{char}, 104)\}.$$

つまり、元のデータ型の 3 つのコピーからなるブロックが 2 つ、要素 4 つ分の間隔 ($4 \cdot 16$ バイト) で並ぶ。

例 **3.22** MPI_TYPE_VECTOR(3, 1, -2, oldtype, newtype) の呼び出しは次のようなデータ型を作る。

$$\{(\mathsf{double}, 0), (\mathsf{char}, 8), (\mathsf{double}, -32), (\mathsf{char}, -24), (\mathsf{double}, -64), (\mathsf{char}, -56)\}.$$

一般的に、 oldtype 型マップが次のようで、

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

範囲が ex であり、 blocklength が bl であるならば、新たに生成されるデータ型は count $\cdot$ bl $\cdot n$ 個のエントリを持つ次のような型マップを持つ。

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1}),$$

$$(type_0, disp_0 + ex), ..., (type_{n-1}, disp_{n-1} + ex), ...,$$

$$(type_0, disp_0 + (\mathsf{bl} - 1) \cdot ex), ..., (type_{n-1}, disp_{n-1} + (\mathsf{bl} - 1) \cdot ex),$$

$$(type_0, disp_0 + \mathsf{stride} \cdot ex), ..., (type_{n-1}, disp_{n-1} + \mathsf{stride} \cdot ex), ...,$$

$$(type_0, disp_0 + (\mathsf{stride} + \mathsf{bl} - 1) \cdot ex), ..., (type_{n-1}, disp_{n-1} + (\mathsf{stride} + \mathsf{bl} - 1) \cdot ex), ...,$$

$$(type_0, disp_0 + \mathsf{stride} \cdot (\mathsf{count} - 1) \cdot ex), ...,$$

$$(type_{n-1}, disp_{n-1} + \mathsf{stride} \cdot (\mathsf{count} - 1) \cdot ex), ...,$$

$$(type_0, disp_0 + (\mathsf{stride} \cdot (\mathsf{count} - 1) + \mathsf{bl} - 1) \cdot ex), ...,$$

$(type_{n-1}, disp_{n-1} + (\mathsf{stride} \cdot (\mathsf{count} - 1) + \mathsf{bl} - 1) \cdot ex)\}$.

MPI_TYPE_CONTIGUOUS(count, oldtype, newtype) は MPI_TYPE_VECTOR(count, 1, 1, oldtype, newtype) 或は MPI_TYPE_VECTOR(1, count, n, oldtype, newtype), n は任意) と同じである。

Hvector MPI_TYPE_HVECTOR は、 stride が要素数ではなくバイト単位であることを除けば、MPI_TYPE_VECTOR と同じである。両方のベクトル型コンストラクタの使い方は 3.12.7 節で説明する。 (H は"heterogeneous"(異種の) の意味)

MPI_TYPE_HVECTOR(count, blocklength, stride, oldtype, newtype)

入力	count	繰り返し回数 (非負の整数型)
入力	blocklength	個々のブロックの要素数 (非負の整数型)
入力	stride	個々のブロックの先頭の間隔のバイト数 (整数型)
入力	oldtype	旧データ型 (ハンドル)
出力	newtype	新データ型 (ハンドル)

```
int MPI_Type_hvector(int count, int blocklength, MPI_Aint stride,
             MPI_Datatype oldtype, MPI_Datatype *newtype)
MPI_TYPE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR
```

oldtype 型マップが次のようで、

$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\}$,

範囲が ex であり、 blocklength が bl であるならば、新たに生成されるデータ型は count·bl·n 個のエントリを持つ次のような型マップを持つ。

$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1}),$

$(type_0, disp_0 + ex), ..., (type_{n-1}, disp_{n-1} + ex), ...,$

$(type_0, disp_0 + (\mathsf{bl} - 1) \cdot ex), ..., (type_{n-1}, disp_{n-1} + (\mathsf{bl} - 1) \cdot ex),$

$(type_0, disp_0 + \mathsf{stride}), ..., (type_{n-1}, disp_{n-1} + \mathsf{stride}), ...,$

$(type_0, disp_0 + \mathsf{stride} + (\mathsf{bl} - 1) \cdot ex), ...,$

$(type_{n-1}, disp_{n-1} + \mathsf{stride} + (\mathsf{bl} - 1) \cdot ex), ...,$

$(type_0, disp_0 + \mathsf{stride} \cdot (\mathsf{count} - 1)), ..., (type_{n-1}, disp_{n-1} + \mathsf{stride} \cdot (\mathsf{count} - 1)), ...,$

$(type_0, disp_0 + \mathsf{stride} \cdot (\mathsf{count} - 1) + (\mathsf{bl} - 1) \cdot ex), ...,$

$(type_{n-1}, disp_{n-1} + \mathsf{stride} \cdot (\mathsf{count} - 1) + (\mathsf{bl} - 1) \cdot ex)\}.$

INDEXED　MPI_TYPE_INDEXED は元のデータ型の複製でブロックの並びを作ることが出来る。(個々のブロックは元のデータ型の連なりである。) 個々のブロックは異なる数のコピーを持つことが出来、また異なる変位を持つことが出来る。全てのブロックの変位は元の型の範囲の倍数である。

MPI_TYPE_INDEXED(count, array_of_blocklengths, array_of_displacements, oldtype, newtype)

入力	count	ブロック数 − または array_of_displacements と array_of_blocklengths のエントリ数 (非負の整数型)
入力	array_of_blocklengths	ブロック毎の要素数 (非負の整数の配列)
入力	array_of_displacements	個々のブロックの変位、oldtype の範囲の倍数 (整数の配列)
入力	oldtype	旧データ型 (ハンドル)
出力	newtype	新データ型 (ハンドル)

```
int MPI_Type_indexed(int count, int *array_of_blocklengths,
             int *array_of_displacements, MPI_Datatype oldtype,
             MPI_Datatype *newtype)

MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
             OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
    OLDTYPE, NEWTYPE, IERROR
```

例 3.23 型マップが $\{(\mathsf{double}, 0), (\mathsf{char}, 8)\}$, で、範囲が 16 であるような oldtype を仮定する。 B = (3, 1)、 D = (4, 0) としたときの MPI_TYPE_INDEXED(2, B, D, oldtype, newtype) は次のような型マップを持つデータ型を返す。

$$\{(\mathsf{double}, 64), (\mathsf{char}, 72), (\mathsf{double}, 80), (\mathsf{char}, 88), (\mathsf{double}, 96), (\mathsf{char}, 104),$$

$$(\mathsf{double}, 0), (\mathsf{char}, 8)\}.$$

つまり、変位 64 から 3 つ、元のデータ型のコピーが、変位 0 から 1 つのコピーが並ぶ。

一般的に、 oldtype 型マップが次のようで、

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

範囲が ex であり、 B が array_of_blocklength 引数で、 D が array_of_displacements 引数である時、新たに生成されるデータ型は $n \cdot \sum_{i=0}^{\mathsf{count}-1} \mathsf{B[i]}$ 個のエントリを持つ。

$$\{(type_0, disp_0 + \mathsf{D[0]} \cdot ex), ..., (type_{n-1}, disp_{n-1} + \mathsf{D[0]} \cdot ex), ...,$$

$$(type_0, disp_0 + (\mathsf{D[0]} + \mathsf{B[0]} - 1) \cdot ex), ..., (type_{n-1}, disp_{n-1} + (\mathsf{D[0]} + \mathsf{B[0]} - 1) \cdot ex), ...,$$

$$(type_0, disp_0 + \mathsf{D[count} - 1] \cdot ex), ..., (type_{n-1}, disp_{n-1} + \mathsf{D[count} - 1] \cdot ex), ...,$$

$$(type_0, disp_0 + (\mathsf{D[count} - 1] + \mathsf{B[count} - 1] - 1) \cdot ex), ...,$$

$$(type_{n-1}, disp_{n-1} + (\mathsf{D[count} - 1] + \mathsf{B[count} - 1] - 1) \cdot ex)\}.$$

MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype) は MPI_TYPE_INDEXED(count, B, D, oldtype, newtype) と同じである。但し、

$$\mathsf{D[j]} = j \cdot \mathsf{stride},\ j = 0, ..., \mathsf{count} - 1,$$

かつ

$$\mathsf{B[j]} = \mathsf{blocklength},\ j = 0, ..., \mathsf{count} - 1.$$

である。

Hindexed　MPI_TYPE_HINDEXED は、 array_of_displacements でのブロックの変位の指定が、 oldtype の範囲の倍数ではなく、バイト数であることを除いて、 MPI_TYPE_INDEXED と同じである。

MPI_TYPE_HINDEXED(count, array_of_blocklengths, array_of_displacements, oldtype, newtype)

入力	count	ブロック数 – または array_of_displacements と array_of_blocklengths のエントリ数 (非負の整数型)
入力	array_of_blocklengths	ブロック毎の要素数 (非負の整数の配列)
入力	array_of_displacements	個々のブロックのバイト単位の変位 (整数の配列)
入力	oldtype	旧データ型 (ハンドル)
出力	newtype	新データ型 (ハンドル)

```
int MPI_Type_hindexed(int count, int *array_of_blocklengths,
              MPI_Aint *array_of_displacements, MPI_Datatype oldtype,
              MPI_Datatype *newtype)

MPI_TYPE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
              OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
    OLDTYPE, NEWTYPE, IERROR
```

一般的に、 oldtype 型マップが次のようで、

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

範囲が ex であり、 B が array_of_blocklength 引数で、 D が array_of_displacements 引数である時、新たに生成されるデータ型は $n \cdot \sum_{i=0}^{\mathsf{count}-1} \mathsf{B[i]}$ 個のエントリを持つ。

$$\{(type_0, disp_0 + \mathsf{D[0]}), ..., (type_{n-1}, disp_{n-1} + \mathsf{D[0]}), ...,$$

$$(type_0, disp_0 + \mathsf{D[0]} + (\mathsf{B[0]} - 1) \cdot ex), ...,$$

$$(type_{n-1}, disp_{n-1} + \mathsf{D[0]} + (\mathsf{B[0]} - 1) \cdot ex), ...,$$

$$(type_0, disp_0 + \mathsf{D[count} - 1]), ..., (type_{n-1}, disp_{n-1} + \mathsf{D[count} - 1]), ...,$$

$$(type_0, disp_0 + \mathsf{D[count-1]} + (\mathsf{B[count-1]} - 1) \cdot ex), ...,$$

$$(type_{n-1}, disp_{n-1} + \mathsf{D[count-1]} + (\mathsf{B[count-1]} - 1) \cdot ex)\}.$$

Struct MPI_TYPE_STRUCT は最も汎用的なコンストラクタである。これまでのものを更に一般化して、各々のブロックに異なるデータ型の複製を持たせることができる。

MPI_TYPE_STRUCT(count, array_of_blocklengths, array_of_displacements, array_of_types, newtype)

入力	count	ブロック数 – または array_of_displacements 、 array_of_blocklengths 及び array_of_types のエントリ数 (非負の整数型)
入力	array_of_blocklength	ブロック毎の要素数 (非負の整数の配列)
入力	array_of_displacements	個々のブロックのバイト単位の変位 (整数の配列)
入力	array_of_types	個々のブロックのデータ型 (データ型オブジェクトのハンドルの配列)
出力	newtype	新データ型 (ハンドル)

```
int MPI_Type_struct(int count, int *array_of_blocklengths,
              MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types,
              MPI_Datatype *newtype)

MPI_TYPE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
              ARRAY_OF_TYPES, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
    ARRAY_OF_TYPES(*), NEWTYPE, IERROR
```

例 **3.24** type1 は次のような型マップを持ち、その範囲は 16 である。

$$\{(\mathsf{double}, 0), (\mathsf{char}, 8)\},$$

B = (2, 1, 3)、 D = (0, 16, 26)

そして T = (MPI_FLOAT, type1, MPI_CHAR) としたときの MPI_TYPE_STRUCT(3, B, D, T, newtype) は次のような型マップを持つデータ型を返す。

$$\{(\mathsf{float}, 0), (\mathsf{float}, 4), (\mathsf{double}, 16), (\mathsf{char}, 24), (\mathsf{char}, 26), (\mathsf{char}, 27), (\mathsf{char}, 28)\}.$$

つまり、2 つの MPI_FLOAT のコピーが変位 0 から並び、続いて 1 つの type1 のコピーが変位 16 から並び、3 つの MPI_CHAR のコピーが変位 26 からそれぞれ並ぶ。(float は 4 バイトを占めるものとする。)

一般的に、 array_of_types 引数が argT であり、 T[i] が次のようで、

$$typemap_i = \{(type^i_0, disp^i_0), ..., (type^i_{n_i-1}, disp^i_{n_i-1})\},$$

範囲が ex_i であり、 B が array_of_blocklength 引数で、 D が array_of_displacements 引数で c が count 引数である時、新たに生成されるデータ型は $\sum_{i=0}^{c-1}$ B[i] · n_i 個のエントリを持つ。

$$\{(type^0_0, disp^0_0 + D[0]), ..., (type^0_{n_0}, disp^0_{n_0} + D[0]), ...,$$

$$(type^0_0, disp^0_0 + D[0] + (B[0] - 1) \cdot ex_0), ..., (type^0_{n_0}, disp^0_{n_0} + D[0] + (B[0] - 1) \cdot ex_0), ...,$$

$$(type^{c-1}_0, disp^{c-1}_0 + D[c - 1]), ..., (type^{c-1}_{n_{c-1}-1}, disp^{c-1}_{n_{c-1}-1} + D[c - 1]), ...,$$

$$(type^{c-1}_0, disp^{c-1}_0 + D[c - 1] + (B[c - 1] - 1) \cdot ex_{c-1}), ...,$$

$$(type^{c-1}_{n_{c-1}-1}, disp^{c-1}_{n_{c-1}-1} + D[c - 1] + (B[c - 1] - 1) \cdot ex_{c-1})\}.$$

MPI_TYPE_HINDEXED(count, B, D, oldtype, newtype) は MPI_TYPE_STRUCT(count, B, D, T, newtype) と同じである。但し T の個々の要素は oldtype と等しいとする。

3.12.2 アドレス関数と範囲関数

汎用データ型の変位はなんらかのバッファの先頭アドレスからの相対的なものである。 絶対アドレスをこれらの変位の代わりに用いることが出来る。絶対アドレスはアドレス空間の始点である「アドレス 0」からの相対的な変位として扱う。この初期アドレス 0 は定数 MPI_BOTTOM で表される。従って、 buf 引数には MPI_BOTTOM を渡すことで、データ型の定義に通信バッファ内のエントリとして絶対アドレスを指定できる。

メモリ内での位置に対するアドレスは MPI_ADDRESS によって得られる。

MPI_ADDRESS(location, address)

入力	location	呼び出し元メモリ内の位置 (選択型)
出力	address	位置のアドレス (整数型)

```
int MPI_Address(void* location, MPI_Aint *address)
```

```
MPI_ADDRESS(LOCATION, ADDRESS, IERROR)
    <type> LOCATION(*)
    INTEGER ADDRESS, IERROR
```

位置に対する (バイト) アドレスを返す。

例 **3.25** MPI_ADDRESS を配列に使う。

```
   REAL A(100,100)
   INTEGER I1, I2, DIFF
   CALL MPI_ADDRESS(A(1,1), I1, IERROR)
   CALL MPI_ADDRESS(A(10,10), I2, IERROR)
   DIFF = I2 - I1
! DIFF の値は 909*REAL のサイズ である。 I1, I2 の値は実装による。
```

ユーザへのアドバイス C の利用者は MPI_ADDRESS の使用を避け、アドレス演算子 & を使用したいと思うかも知れない。しかしながら、 & 変換式はアドレスではなくポインタであることに注意しなければならない。 ANSI C ではポインタ (或はポインタを変換した整数値) がオブジェクトの指し示す絶対アドレスを示すことを要求しない — が、これが一般的ではある。さらに、セグメント化されたアドレス空間を持つマシン上では、参照に対して、唯一の定義が存在しないかも知れない。 MPI_ADDRESS を C の変数への参照に利用することで、この様なマシン上での移植性が保証される。(ユーザへのアドバイスの終わり)

以下の補助関数は派生データ型に対する有用な情報を与えるものである。

MPI_TYPE_EXTENT(datatype, extent)

入力	datatype	データ型 (ハンドル)
出力	extent	データ型の範囲 (整数型)

```
int MPI_Type_extent(MPI_Datatype datatype, int *extent)
MPI_TYPE_EXTENT(DATATYPE, EXTENT, IERROR)
    INTEGER DATATYPE, EXTENT, IERROR
```

datatype の範囲を返す。範囲は 79ページの 3.1式で定義されている。

MPI_TYPE_SIZE(datatype, size)

入力	datatype	データ型 (ハンドル)
出力	size	データ型のサイズ (整数型)

```
int MPI_Type_size(MPI_Datatype datatype, int *size)

MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)
    INTEGER DATATYPE, SIZE, IERROR
```

MPI_TYPE_SIZE は datatype の型仕様のエントリの合計サイズをバイト単位で返す。つまり、このデータ型で作成されるメッセージデータの合計サイズである。 datatype 内で複数回現れるエントリは複数回数えられる。

MPI_TYPE_COUNT(datatype, count)

入力	datatype	データ型（ハンドル）
出力	count	データ型のカウント（整数型）

```
int MPI_Type_count(MPI_Datatype datatype, int *count)

MPI_TYPE_COUNT(DATATYPE, COUNT, IERROR)
    INTEGER DATATYPE, COUNT, IERROR
```

データ型の「最上層」のエントリーの数を返す。

3.12.3 下限マーカと上限マーカ

型マップの上限と下限を明示的に指定し、 79ページの 3.1式の定義と置き換えると便利な場合がある。これによって、最初や最後に「穴」を持つデータ型や、上限の後ろや下限の前に拡張されたエントリを持つような、データ型を定義することが出来る。この様な使い方の例を 3.12.7節に示す。また、利用者が上限や範囲の計算に利用されるアラインメントの規則を変更することができる。例えば、ある C コンパイラでは利用者がプログラム中の構造体の幾つかの標準のアラインメントを変更することが可能かも知れない。利用者はこれらの構造体に合致するようにデータ型の範囲を明示する必要がある。 このために、二つの疑似データ型、 MPI_LB と MPI_UB、を追加し、個々に、データ型の上限と下限を示すのに用いることが出来るようにする。これらの疑似データ型は範囲を持たない。 $(extent(\mathsf{MPI\_LB}) = extent(\mathsf{MPI\_UB}) = 0)$ また、データ型の大きさやエントリ数、また作成されるメッセージの内容にも影響しない。しかし、データ型の範囲の定義には関係するので、データ型コンストラクタによる、このデータ型の複製結果には影響す

る。

例 **3.26** D = (-3, 0, 6); T = (MPI_LB, MPI_INT, MPI_UB)、かつ B = (1, 1, 1) とする。このとき、 MPI_TYPE_STRUCT(3, B, D, T, type1) は範囲が 9(-3 から 5 まで (5 を含む)) で、整数値を変位 0 に持つような、データ型を作る。これは、 {(lb, -3), (int, 0), (ub, 6)} のような並びで表すことができる。もし、 MPI_TYPE_CONTIGUOUS(2, type1, type2) によって、このデータ型が 2 回繰り返されたら、新たに生成されるデータ型は、 {(lb, -3), (int, 0), (int,9), (ub, 15)} で表すことが出来る。 (ub の上位に ub があればその ub は削除でき、 lb の下位に lb があればその lb は削除できる。)

一般的に、

$$Typemap = \{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

の場合、 $Typemap$ の下限は

$$lb(Typemap) = \begin{cases} \min_j disp_j & \text{lb が含まれない場合} \\ \min_j\{disp_j \text{ such that } type_j = \text{lb}\} & \text{otherwise} \end{cases}$$

Similarly, the **upper bound** of $Typemap$ is defined to be

$$ub(Typemap) = \begin{cases} \max_j disp_j + sizeof(type_j) & \text{ub が含まれない場合} \\ \max_j\{disp_j \text{ such that } type_j = \text{ub}\} & \text{otherwise} \end{cases}$$

のように定義される。したがって、範囲は、

$$extent(Typemap) = ub(Typemap) - lb(Typemap) + \epsilon$$

となる。 $type_i$ が k_i の倍数に配置されなければならないならば、 ϵ は $extent(Typemap)$ を $\max_i k_i$ の倍数のうち最も小さなものに丸める非負の値になる。

この範囲の定義の修正で、様々なデータ型コンストラクタの正式な定義がなされた。

以下の二つの関数は、データ型の上限と下限を返す。

MPI_TYPE_LB(datatype, displacement)

入力	datatype	データ型 (ハンドル)
出力	displacement	原点からの下限のバイト変位 (整数型)

```
int MPI_Type_lb(MPI_Datatype datatype, int* displacement)

MPI_TYPE_LB( DATATYPE, DISPLACEMENT, IERROR)
```

```
    INTEGER DATATYPE, DISPLACEMENT, IERROR
```

MPI_TYPE_UB(datatype, displacement)

入力	datatype	データ型 (ハンドル)
出力	displacement	原点からの上限のバイト変位 (整数型)

```
int MPI_Type_ub(MPI_Datatype datatype, int* displacement)

MPI_TYPE_UB( DATATYPE, DISPLACEMENT, IERROR)
    INTEGER DATATYPE, DISPLACEMENT, IERROR
```

3.12.4 記憶と解放

データ型オブジェクトは通信で利用される前に記憶されていなければならない。記憶されたデータ型はデータ型コンストラクタの引数としても用いることが出来る。基本データ型は「予め記憶されている」ので、記憶する必要はない。

MPI_TYPE_COMMIT(datatype)

入出力	datatype	記憶するデータ型 (ハンドル)

```
int MPI_Type_commit(MPI_Datatype *datatype)

MPI_TYPE_COMMIT(DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR
```

記憶操作はデータ型、つまり通信バッファの内容ではなく、通信バッファの記述を記憶する。従って、ひとたびデータ型を記憶すれば、バッファの内容を変更しながらの通信や、異なるアドレスから始まる異なるバッファを用いた通信に繰り返し利用することが出来る。

> 実装者へのアドバイス　システムは、通信を容易にするために記憶時にデータ型を内部表現へコンパイルしてもよい。例えば、圧縮された表現からフラットな表現へ変え、最も有利な転送機能を選ぶことが出来る。（実装者へのアドバイスの終わり）

MPI_TYPE_FREE(datatype)

入出力 datatype 解放されるデータ型 (ハンドル)

```
int MPI_Type_free(MPI_Datatype *datatype)

MPI_TYPE_FREE(DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR
```

datatype で表されているデータ型オブジェクトに削除するためのマークをつけ、 datatype を MPI_DATATYPE_NULL にする。 datatype を使っている通信があれば、その通信は通常通り完了する。解放されたデータ型から派生したデータ型は無効である。

例 **3.27** 以下のコードの一部は MPI_TYPE_COMMIT の使い方である。

```
INTEGER type1, type2
CALL MPI_TYPE_CONTIGUOUS(5, MPI_REAL, type1, ierr)
              ! 新しいデータ型オブジェクトを作成。
CALL MPI_TYPE_COMMIT(type1, ierr)
              ! type1 は通信に利用可能。
type2 = type1
              ! type2 は通信に利用可能。
              ! (type1 と同じオブジェクトへのハンドルである。)
CALL MPI_TYPE_VECTOR(3, 5, 4, MPI_REAL, type1, ierr)
              ! 新しい記憶されていないデータ型を作成。
CALL MPI_TYPE_COMMIT(type1, ierr)
              ! type1 は新たに通信可能になった。
```

解放操作は、解放しようとするデータ型から派生したデータ型には作用しない。システムは派生データ型コンストラクタへ渡されたデータ型引数は値渡しされたものとして振舞う。

> 実装者へのアドバイス 実装者は、データ型を解放する時期を決定するために、そのデータ型を使っている通信の参照カウンタを保持してもよい。またある実装では、派生データ型コンストラクタへのデータ型引数ををコピーする代わりに、そのポインタを保持するようにしてもよい。この場合、データ型オブジェクトを解放する時期を知るために、有効なデータ型定義への参照を追跡する必要がある。(実装者へのアドバイスの終わり)

3.12.5 通信時の汎用データ型の利用

派生データ型のハンドルは、どこであれデータ型引数が要求される通信関数に渡すことが出来る。 MPI_SEND(buf, count, datatype , ...) 型の呼出、 count $>$ 1 の様な場合、は count 回

datatype が連結された新たなデータ型が渡されたものと考えることができる。故に、 MPI_SEND(buf, count, datatype, dest, tag, comm) は以下と等価である。

```
MPI_TYPE_CONTIGUOUS(count, datatype, newtype)
MPI_TYPE_COMMIT(newtype)
MPI_SEND(buf, 1, newtype, dest, tag, comm).
```

他の count と datatype を引数に持つ通信関数全てに同様のことがいえる。

datatype が次のような型マップを持ち、範囲が $extent$ である場合、 MPI_SEND(buf, count,datatype, dest, tag, comm) を仮定する。 (空のエントリ、 MPI_UB 及び MPI_LB の「疑似データ型」は型マップ中にはないものとするが、 $extent$ に関係している。)

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

この送信操作は n·count エントリを送信する。この時、 $i{\cdot}n{+}j$ 番目のエントリが $addr_{i,j} =$ buf $+ extent \cdot i + disp_j$ に配置され、型が $type_j$ であり、かつ、 $i = 0, ...,$ count $- 1$ 及び $j = 0, ..., n-1$ である。これらのエントリは連続である必要もなければ別々である必要もない。順序も任意である。

呼出側のプログラム中で $addr_{i,j}$ に格納されている変数は $type_j$ に一致する型でなければならない。型の一致については 3.3.1節で定義している。送信されるメッセージは $n \cdot$ count 個のエントリからなり、 $i \cdot n + j$ 番目のエントリは $type_j$ の型である。

同様に、 datatype が次のような型マップを持ち範囲が $extent$ であるような、受信操作 MPI_RECV(buf, count, datatype, source, tag, comm, status) を実行すると仮定する。 (同様に、空のエントリ、 MPI_UB 及び MPI_LB の「疑似データ型」は型マップ中にはないものとするが、 $extent$ に関係している。)

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

この受信操作は $n \cdot$ count 個のエントリを受信する。この時 $i \cdot n + j$ 番目のエントリが buf $+ extent \cdot i + disp_j$ に配置されていて、型が $type_j$ であるとする。届いたメッセージが k 個の要素を持っている場合、 k は $k \leq n \cdot$ count でなければならず、メッセージの $i \cdot n + j$ 番目の要素は $type_j$ と合致する型でなければならない。

型の一致は対応するデータ型の型仕様、即ち、基本データ型で表される構成要素の並びによって定義される。型の一致はデータ型定義の幾つかの様相、変位 (メモリ中の位置) や中間型の利用など、には依存しない。

例 **3.28** この例は、型の一致が派生データ型の構成要素である基本データ型によって定義されることを示す。

```
...
CALL MPI_TYPE_CONTIGUOUS( 2, MPI_REAL, type2, ...)
CALL MPI_TYPE_CONTIGUOUS( 4, MPI_REAL, type4, ...)
CALL MPI_TYPE_CONTIGUOUS( 2, type2, type22, ...)
...
CALL MPI_SEND( a, 4, MPI_REAL, ...)
CALL MPI_SEND( a, 2, type2, ...)
CALL MPI_SEND( a, 1, type22, ...)
CALL MPI_SEND( a, 1, type4, ...)
...
CALL MPI_RECV( a, 4, MPI_REAL, ...)
CALL MPI_RECV( a, 2, type2, ...)
CALL MPI_RECV( a, 1, type22, ...)
CALL MPI_RECV( a, 1, type4, ...)
```

各々の送信は、どの受信とでも適合する。

データ型は重複するエントリを含んでいるかもしれない。この様なデータ型を受信に使うのは誤りである。(これは実際に受信されたメッセージが、どのエントリも上書きしない位に短くても誤りである。)

datatype が以下のような型マップを持つ、 MPI_RECV(buf, count, datatype, dest, tag, comm, status) の実行を仮定する。

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\}.$$

受信されたメッセージは受信バッファを全て満たす必要はなく、また n の倍数個の場所を満たす必要もない。任意の数 k 個の基本要素を受信可能である。但し、 $0 \leq k \leq$ count $\cdot\, n$ である。受信することが出来る基本要素の数は、問い合わせ関数 MPI_GET_ELEMENTS を使って status から取り出すことが出来る。

MPI_GET_ELEMENTS(status, datatype, count)

入力	status	受信操作の結果 (ステータス)
入力	datatype	受信操作に使ったデータ型 (ハンドル)
出力	count	受信された基本要素数 (整数型)

```
int MPI_Get_elements(MPI_Status status, MPI_Datatype datatype, int *count)
```

```
MPI_GET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
```

既に定義した関数、MPI_GET_COUNT(3.2.5節)、はこれとは異なる振舞いをする。それは受信された「上層のエントリ数」、即ち datatype 型の「コピー数」、を返す。前の例で、MPI_GET_COUNT は $0 \leq k \leq$ count となる k を返す。このとき、受信された基本要素数 (MPI_GET_ELEMENTS が返す値) は $n \cdot k$ である。受信された基本要素数が n の倍数ではない場合、即ち、受信操作によって datatype の整数個のコピーが受信できなかった場合 MPI_GET_COUNT は MPI_UNDEFINED を返す。

例 **3.29** MPI_GET_COUNT 及び MPI_GET_ELEMENT の使い方。

```
...
CALL MPI_TYPE_CONTIGUOUS(2, MPI_REAL, Type2, ierr)
CALL MPI_TYPE_COMMIT(Type2, ierr)
...
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank.EQ.0) THEN
      CALL MPI_SEND(a, 2, MPI_REAL, 1, 0, comm, ierr)
      CALL MPI_SEND(a, 3, MPI_REAL, 1, 0, comm, ierr)
ELSE
      CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
      CALL MPI_GET_COUNT(stat, Type2, i, ierr)     ! i=1   を返す
      CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr)  ! i=2   を返す
      CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
      CALL MPI_GET_COUNT(stat, Type2, i, ierr)     ! i=MPI_UNDEFINED   を返す
      CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr)  ! i=3   を返す
END IF
```

関数 MPI_GET_ELEMENTS も探査後に探査されたメッセージの要素数を調べるために用いることが可能である。これら二つの関数、MPI_GET_COUNT と MPI_GET_ELEMENTS は基本データ型を用いた場合同じ値を返すことに注意。

根拠　この MPI_GET_COUNT の定義に対する拡張は自然なものである。受信バッファが一杯であるなら、この関数は count 引数を返すことが期待されるだろう。時には datatype が転送したいデータの基本単位を表す。例えば、レコード (構造) の配列中のレコードの様に。各々の構成要素に於いて、幾つの要素が受信されたのかを割算の手間をかけずに得ることが出来るべきである。しかしながら、場合によっては、datatype が受信側のメモリ中

の複雑なデータは位置を定義するために用いられているために、転送の基本単位を表すことが出来ない場合がある。この様な場合に MPI_GET_ELEMENTS が必要となる。（根拠の終わり）

実装者へのアドバイス　この定義は、受信によって、通信バッファを構成するエントリ以外の記憶領域は変更できないことを含んでいる。特に、構造体中のパディング領域は、この様な構造体があるプロセスから別のプロセスへ複写される時に変更されてはならないことを含む。このため、パディング領域を含めてひとつの連続したブロックと見立てる、単純な構造体の複写の最適化は行なうことが出来ない。実装の際には計算結果に影響のない範囲でこの最適化を自由に行なってよい。利用者はパディングをメッセージの一部として明示的に加えることで、この最適化を強制することが出来る。（実装者へのアドバイスの終わり）

3.12.6 アドレスの正しい利用

C や FORTRAN で続けて宣言された変数が連続した位置に配置されるとは限らない。従って、変位がある変数から他の変数へと跨らない様に注意して用いなければならない。同様に、セグメント化されたアドレス空間を持つ計算機の場合は、アドレスは一意ではなく、アドレスの計算には特別な方法がある。それゆえアドレス、即ち、始点 MPI_BOTTOM に対する相対変位、の利用は制限されなければならない。

同じ配列に属する変数、 FORTRAN の同じ COMMON ブロックに属する変数や、 C の同じ構造体に属する変数は、同じ連続領域に属する変数である。有効なアドレスは次のように再帰的に定義される。

1. 呼出側プログラムの変数を引数として渡した時、関数 MPI_ADDRESS は有効なアドレスを返す。
2. 呼出側プログラムの変数を引数として渡した時、通信関数は引数 buf を有効なアドレスとして評価する。
3. v が有効なアドレスであり、 i が整数であるとき、 v と v+i が同じ記憶領域にあれば、 v+i は有効なアドレスである。
4. v が有効なアドレスなら、 MPI_BOTTOM + v は有効なアドレスである。

正しいプログラムは、通信バッファ内のエントリの位置を特定するために有効なアドレスのみを利用する。さらに、 u と v が有効なアドレスである時、 (整数値の) 差分 u - v は u と v が同じ連続した領域にある場合に限り計算できる。アドレスに関して他の算術演算は意味を持たない。

上記の規則は、派生データ型が同じ記憶領域内に全て含まれる通信バッファを定義するために用いられる限り、何ら制約を課すものではない。しかしながら、同じ記憶領域内にない変数を使って通信バッファを作る場合には制約に従わねばならない。基本的に、異なる記憶領域内にある変数からなる通信バッファは、buf = MPI_BOTTOM、count = 1 かつ、全て有効な (絶対) アドレスを変位として持つ datatype を引数として指定した時のみ通信に用いることが出来る。

ユーザへのアドバイス　MPI はホストプログラム中の配列やレコードの大きさを知ることが出来ないかも知れないので、MPI の実装が、利用者のアドレス空間を越えない様に、“範囲外” 変位の検出をすることを、期待してはならない。(ユーザへのアドバイスの終わり)

実装者へのアドバイス　連続したアドレス空間を持つ計算機上では (絶対) アドレスと、(相対) 変位を区別する必要はない。MPI_BOTTOM は 0 で、アドレスと変位の双方を整数とみなすことが出来る。区別を必要とする計算機上で、アドレスは MPI_BOTTOM を含む式と見なすことが出来る。(実装者へのアドバイスの終わり)

3.12.7 例

以下の例は派生データ型の使用法を説明している。

例 **3.30** 3次元配列の一部を送受信する。

```
      REAL a(100,100,100), e(9,9,9)
      INTEGER oneslice, twoslice, threeslice, sizeofreal, myrank, ierr
      INTEGER status(MPI_STATUS_SIZE)

C      配列の一部 a(1:17:2, 3:11, 2:10) を取り出し、
C      e(:,:,:) へ格納する。

      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank)

      CALL MPI_TYPE_EXTENT( MPI_REAL, sizeofreal, ierr)

C      1次元部分に対するデータ型を作成する
      CALL MPI_TYPE_VECTOR( 9, 1, 2, MPI_REAL, oneslice, ierr)

C      2次元部分に対するデータ型を作成する
```

```
      CALL MPI_TYPE_HVECTOR(9, 1, 100*sizeofreal, oneslice, twoslice, ierr)

C     全部分に対するデータ型を作成する
      CALL MPI_TYPE_HVECTOR( 9, 1, 100*100*sizeofreal, twoslice, 1,
                             threeslice, ierr)

      CALL MPI_TYPE_COMMIT( threeslice, ierr)
      CALL MPI_SENDRECV(a(1,3,2), 1, threeslice, myrank, 0, e, 9*9*9,
                        MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
```

例 **3.31** 行列の下三角部分を（厳密に）コピーする。

```
      REAL a(100,100), b(100,100)
      INTEGER  disp(100), blocklen(100), ltype, myrank, ierr
      INTEGER status(MPI_STATUS_SIZE)

C     配列 a の下三角部分を
C     配列 b の下三角部分にコピーする

      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank)

C     各々の列の先頭とサイズを計算する
      DO i=1, 100
        disp(i) = 100*(i-1) + i
        block(i) = 100-i
      END DO

C     下三角部分に対するデータ型を作成する
      CALL MPI_TYPE_INDEXED( 100, block, disp, MPI_REAL, ltype, ierr)

      CALL MPI_TYPE_COMMIT(ltype, ierr)
      CALL MPI_SENDRECV( a, 1, ltype, myrank, 0, b, 1,
                         ltype, myrank, 0, MPI_COMM_WORLD, status, ierr)
```

例 **3.32** 行列を転置する。

```
      REAL a(100,100), b(100,100)
```

```
      INTEGER row, xpose, sizeofreal, myrank, ierr
      INTEGER status(MPI_STATUS_SIZE)

C     行列 a を b に転置する

      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank)

      CALL MPI_TYPE_EXTENT( MPI_REAL, sizeofreal, ierr)

C     一つの行に対するデータ型を作成する
      CALL MPI_TYPE_VECTOR( 100, 1, 100, MPI_REAL, row, ierr)

C     create datatype for matrix in row-major order
      CALL MPI_TYPE_HVECTOR( 100, 1, sizeofreal, row, xpose, ierr)

      CALL MPI_TYPE_COMMIT( xpose, ierr)

C     send matrix in row-major order and receive in column major order
      CALL MPI_SENDRECV( a, 1, xpose, myrank, 0, b, 100*100,
                 MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
```

例 **3.33** 転置問題への違ったアプローチ：

```
      REAL a(100,100), b(100,100)
      INTEGER  disp(2), blocklen(2), type(2), row, row1, sizeofreal
      INTEGER  myrank, ierr
      INTEGER status(MPI_STATUS_SIZE)

      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank)

C     transpose matrix a onto b

      CALL MPI_TYPE_EXTENT( MPI_REAL, sizeofreal, ierr)

C     create datatype for one row
      CALL MPI_TYPE_VECTOR( 100, 1, 100, MPI_REAL, row, ierr)
```

```
C     create datatype for one row, with the extent of one real number
      disp(1) = 0
      disp(2) = sizeofreal
      type(1)  = row
      type(2)  = MPI_UB
      blocklen(1)  = 1
      blocklen(2)  = 1
      CALL MPI_TYPE_STRUCT( 2, blocklen, disp, type, row1, ierr)

      CALL MPI_TYPE_COMMIT( row1, ierr)

C     send 100 rows and receive in column major order
      CALL MPI_SENDRECV( a, 100, row1, myrank, 0, b, 100*100,
                MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
```

例 **3.34** We manipulate an array of structures.

```
struct Partstruct
   {
   int    class;  /* particle class */
   double d[6];   /* particle coordinates */
   char   b[7];   /* some additional information */
   };

struct Partstruct    particle[1000];

int                  i, dest, rank;
MPI_Comm     comm;



/* build datatype describing structure */

MPI_Datatype Particletype;
MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
int          blocklen[3] = {1, 6, 7};
```

```
MPI_Aint     disp[3];
int          base;



/* compute displacements of structure components */

MPI_Address( particle, disp);
MPI_Address( particle[0].d, disp+1);
MPI_Address( particle[0].b, disp+2);
base = disp[0];
for (i=0; i <3; i++) disp[i] -= base;

MPI_Type_struct( 3, blocklen, disp, type, &Particletype);

   /* If compiler does padding in mysterious ways,
   the following may be safer */

MPI_Datatype type1[4] = {MPI_INT, MPI_DOUBLE, MPI_CHAR, MPI_UB};
int          blocklen1[4] = {1, 6, 7, 1};
MPI_Aint     disp1[4];

/* compute displacements of structure components */

MPI_Address( particle, disp1);
MPI_Address( particle[0].d, disp1+1);
MPI_Address( particle[0].b, disp1+2);
MPI_Address( particle+1, disp1+3);
base = disp1[0];
for (i=0; i <4; i++) disp1[i] -= base;

/* build datatype describing structure */

MPI_Type_struct( 4, blocklen1, disp1, type1, &Particletype);



```

```
             /* 4.1:
        send the entire array */

MPI_Type_commit( &Particletype);
MPI_Send( particle, 1000, Particletype, dest, tag, comm);



             /* 4.2:
        send only the entries of class zero particles,
        preceded by the number of such entries */

MPI_Datatype Zparticles;   /* datatype describing all particles
                              with class zero (needs to be recomputed
                              if classes change) */
MPI_Datatype Ztype;

MPI_Aint     zdisp[1000];
int zblock[1000], j, k;
int zzblock[2] = {1,1};
MPI_Aint     zzdisp[2];
MPI_Datatype zztype[2];

/* compute displacements of class zero particles */
j = 0;
for(i=0; i < 1000; i++)
  if (particle[i].class==0)
     {
     zdisp[j] = i;
     zblock[j] = 1;
     j++;
     }

/* create datatype for class zero particles  */
MPI_Type_indexed( j, zblock, zdisp, Particletype, &Zparticles);
```

```
/* prepend particle count */
MPI_Address(&j, zzdisp);

MPI_Address(particle, zzdisp+1);
zztype[0] = MPI_INT;
zztype[1] = Zparticles;
MPI_Type_struct(2, zzblock, zzdisp, zztype, &Ztype);

MPI_Type_commit( &Ztype);
MPI_Send( MPI_BOTTOM, 1, Ztype, dest, tag, comm);



       /* A probably more efficient way of defining Zparticles */

/* consecutive particles with index zero are handled as one block */
j=0;
for (i=0; i < 1000; i++)
  if (particle[i].index==0)
    {
    for (k=i+1; (k < 1000)&&(particle[k].index == 0) ; k++);
    zdisp[j] = i;
    zblock[j] = k-i;
    j++;
    i = k;
    }
MPI_Type_indexed( j, zblock, zdisp, Particletype, &Zparticles);


               /* 4.3:
          send the first two coordinates of all entries */

MPI_Datatype Allpairs;      /* datatype for all pairs of coordinates */

MPI_Aint sizeofentry;

MPI_Type_extent( Particletype, &sizeofentry);

```

```
     /* sizeofentry can also be computed by subtracting the address
        of particle[0] from the address of particle[1] */

MPI_Type_hvector( 1000, 2, sizeofentry, MPI_DOUBLE, &Allpairs);
MPI_Type_commit( &Allpairs);
MPI_Send( particle[0].d, 1, Allpairs, dest, tag, comm);

      /* an alternative solution to 4.3 */

MPI_Datatype Onepair;   /* datatype for one pair of coordinates, with
                           the extent of one particle entry */
MPI_Aint disp2[3];
MPI_Datatype type2[3] = {MPI_LB, MPI_DOUBLE, MPI_UB};
int blocklen2[3] = {1, 2, 1};

MPI_Address( particle, disp2);
MPI_Address( particle[0].d, disp2+1);
MPI_Address( particle+1, disp2+2);
base = disp2[0];
for (i=0; i<2; i++) disp2[i] -= base;

MPI_Type_struct( 3, blocklen2, disp2, type2, &Onepair);
MPI_Type_commit( &Onepair);
MPI_Send( particle[0].d, 1000, Onepair, dest, tag, comm);
```

例 **3.35** The same manipulations as in the previous example, but use absolute addresses in datatypes.

```
struct Partstruct
   {
   int class;
   double d[6];
   char b[7];
   };
```

```
1
2   struct Partstruct particle[1000];
3
4
5              /* build datatype describing first array entry */
6
7
8   MPI_Datatype Particletype;
9   MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
10  int          block[3] = {1, 6, 7};
11
12  MPI_Aint     disp[3];
13
14
15  MPI_Address( particle, disp);
16  MPI_Address( particle[0].d, disp+1);
17  MPI_Address( particle[0].b, disp+2);
18
19  MPI_Type_struct( 3, block, disp, type, &Particletype);
20
21  /* Particletype describes first array entry -- using absolute
22
23     addresses */
24
25
26                   /* 5.1:
27              send the entire array */
28
29
30  MPI_Type_commit( &Particletype);
31  MPI_Send( MPI_BOTTOM, 1000, Particletype, dest, tag, comm);
32
33
34
35                  /* 5.2:
36           send the entries of class zero,
37
38           preceded by the number of such entries */
39
40
41  MPI_Datatype Zparticles, Ztype;
42
43  MPI_Aint zdisp[1000]
44
45  int zblock[1000], i, j, k;
46  int zzblock[2] = {1,1};
47
48  MPI_Datatype zztype[2];
```

```
MPI_Aint     zzdisp[2];

j=0;
for (i=0; i < 1000; i++)
  if (particle[i].index==0)
    {
    for (k=i+1; (k < 1000)&&(particle[k].index = 0) ; k++);
    zdisp[j] = i;
    zblock[j] = k-i;
    j++;
    i = k;
    }
MPI_Type_indexed( j, zblock, zdisp, Particletype, &Zparticles);
/* Zparticles describe particles with class zero, using
   their absolute addresses*/

/* prepend particle count */
MPI_Address(&j, zzdisp);
zzdisp[1] = MPI_BOTTOM;
zztype[0] = MPI_INT;
zztype[1] = Zparticles;
MPI_Type_struct(2, zzblock, zzdisp, zztype, &Ztype);

MPI_Type_commit( &Ztype);
MPI_Send( MPI_BOTTOM, 1, Ztype, dest, tag, comm);
```

例 **3.36** Handling of unions.

```
union {
   int     ival;
   float   fval;
      } u[1000]

int     utype;
```

```
/* All entries of u have identical type; variable
   utype keeps track of their current type */

MPI_Datatype   type[2];
int            blocklen[2] = {1,1};
MPI_Aint       disp[2];
MPI_Datatype   mpi_utype[2];
MPI_Aint       i,j;

/* compute an MPI datatype for each possible union type;
   assume values are left-aligned in union storage. */

MPI_Address( u, &i);
MPI_Address( u+1, &j);
disp[0] = 0; disp[1] = j-i;
type[1] = MPI_UB;

type[0] = MPI_INT;
MPI_Type_struct(2, blocklen, disp, type, &mpi_utype[0]);

type[0] = MPI_FLOAT;
MPI_Type_struct(2, blocklen, disp, type, &mpi_utype[1]);

for(i=0; i<2; i++) MPI_Type_commit(&mpi_utype[i]);

/* actual communication */

MPI_Send(u, 1000, mpi_utype[utype], dest, tag, comm);
```

3.13 パックとアンパック

既存の幾つかの通信ライブラリは不連続なデータを送信するためにパック / アンパック関数を提供している。この場合、利用者は送信前に連続したバッファに明示的にパックし、受信後に連続したバッファからアンパックする。利用者は 3.12節で説明したような、派生データ型を、ほとんどの場合、明示的にパック、アンパックしなくてよい。利用者は送信する、または受信するデー

タの配置を指定し、通信ライブラリが直接不連続バッファにアクセスする。パック、アンパック機能は既存のライブラリとの互換性のために提供されている。また、 MPI では実現できない様ないくつかの機能を提供する。例えば、受信内容が既に受信された部分に依存するような、複数に分かれたメッセージを受信することが出来る。他には、送出するメッセージを利用者が明示的に指定した場所にバッファリングすることで、システムのバッファリング規則を置き換えることが出来る。つまり、パック、アンパックを利用することで MPI 上に別の通信ライブラリを容易に開発できる。

MPI_PACK(inbuf, incount, datatype, outbuf, outcount, position, comm)

入力	inbuf	入力バッファ始点 (選択型)
入力	incount	入力項目数 (整数型)
入力	datatype	個々の入力項目のデータ型 (ハンドル)
出力	outbuf	出力バッファ始点 (選択型)
入力	outcount	出力バッファ・バイト長 (整数型)
入出力	position	バッファ中での現在のバイト位置 (整数型)
入力	comm	コミュニケータ (ハンドル)

```
int MPI_Pack(void* inbuf, int incount, MPI_Datatype datatype, void *outbuf,
             int outcount, int *position, MPI_Comm comm)

MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTCOUNT, POSITION, COMM,
             IERROR)
    <type> INBUF(*), OUTBUF(*)
    INTEGER INCOUNT, DATATYPE, OUTCOUNT, POSITION, COMM, IERROR
```

inbuf, incount, datatype で指定される送信バッファ内のメッセージを outbuf, outcount で指定されるバッファへパックする。入力バッファには MPI_SEND で使うことが出来るバッファを指定できる。出力バッファは、 outbuf から始まる outcount バイトの領域を含む、連続領域である。 (長さは、 MPI_PACKED 型メッセージの通信バッファであるかのごとく、要素数ではなく、バイトで表される。)

入力値 position は出力バッファ中のパッキングの開始位置である。 position はパックされたメッセージの大きさ分増加し、出力値 position はパックされたメッセージに続く領域の先頭である。 comm はパックされたメッセージを次に送信するために利用するコミュニケータである。

MPI_UNPACK(inbuf, insize, position, outbuf, outcount, datatype, comm)

入力	inbuf	入力バッファ始点 (選択型)
入力	insize	入力バッファ・バイト長 (整数型)
入出力	position	バッファ中での現在のバイト位置 (整数型)
出力	outbuf	出力バッファ始点 (選択型)
入力	outcount	出力項目数 (整数型)
入力	datatype	個々の入力項目のデータ型 (ハンドル)
入力	comm	コミュニケータ (ハンドル)

```
int MPI_Unpack(void* inbuf, int insize, int *position, void *outbuf,
              int outcount, MPI_Datatype datatype, MPI_Comm comm)

MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM,
              IERROR)
    <type> INBUF(*), OUTBUF(*)
    INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR
```

outbuf, outcount, datatype で指定される受信バッファへ inbuf, insize で指定されるバッファ空間からデータをアンパックする。出力バッファには MPI_RECV で利用できる、どの通信バッファでも指定できる。入力バッファは、 inbuf で始まる insize バイトの領域を含む連続した記憶領域である。入力値 position は入力バッファ中のパックされたメッセージの先頭位置である。 position はパックされたメッセージの大きさの分増加するので、出力値 position は入力バッファ中でアンパックされたメッセージの占めていた位置に続く領域の先頭である。 comm はパックされたメッセージを受信するのに使ったコミュニケータである。

> ユーザへのアドバイス MPI_RECV と MPI_UNPACK の違いに注意。 MPI_RECV では count 引数は受信できる最大項目数を指定する。実際に受信された項目数は受信されたメッセージ長で決まる。 MPI_UNPACK では count 引数は実際にアンパックされた項目数である。メッセージの “サイズ” は position の増分に一致する。この “受信メッセージサイズ” の違いは、利用者がどれだけアンパックするかを決定するため、事前に決まらないためである。アンパックされる項目数から “メッセージサイズ” を決定するのは容易ではない。実際、異機種混在のシステムではこの数は経験的に決められない。 (ユーザへのアドバイスの終わり)

パックとアンパックの動作は、そのメッセージで送られる値を連結して得られた並びと考えることが。パック操作は、バッファに送信するかのように、この並びを格納する。アンパック操

作は、バッファから受信されるかのように、この並びを取り出す。(この方法は FORTRAN 言語の内部ファイルや、C 言語の sscanf 等の同様の関数を理解する助けにもなる。)

いくつかの連続した関連する MPI_PACK の呼出によって、いくつかのメッセージを続けてひとつのパッキング単位にパックすることが出来る。最初の呼出は position = 0 で行なわれ、続くそれぞれの呼出は直前の position の出力を入力値として与え、同じ outbuf, outcount 及び comm を使う。このパッキング単位は、個々の送信バッファを "連結" したバッファをひとつの送信操作によって送られたメッセージと等価な情報を含んでいる。

パッキング単位は MPI_PACKED 型として送信できる。あらゆる一対一通信で、パッキング単位の形状をしたバイトの並びを、あるプロセッサから他へ移動することが出来る。この時点で、このパッキング単位はどのようなデータ型を用いてどのような受信操作ででも受信できる。型一致規則は MPI_PACKED 型の場合は緩められる。

どんなデータ型 (MPI_PACKED を含む) で送られたメッセージでも、MPI_PACKED 型で受信することが出来る。この様なメッセージは MPI_UNPACK によってアンパックできる。

連続した MPI_UNPACK の呼出によって、パッキング単位 (或は通常の "型つき" 送信で生成されたメッセージ) を幾つかの連続したメッセージにアンパック出来る。最初は position = 0 で呼び出され、続く呼び出しは、それぞれの直前の呼び出しによって得られる position を引数とし、同じ inbuf, insize 及び comm で呼び出される。

二つのパッキング単位の連結は必ずしもひとつのパッキング単位になるとは限らない。またパッキング単位の一部分がひとつのパッキング単位になるとは限らない。従って、二つのパッキング単位を連結して、ひとつのパッキング単位としてアンパックすることは出来ない。また、パッキング単位の一部分をアンパックして、別々のパッキング単位としてアンパックすることは出来ない。一連のパックによって、或は通常の送信によって作られた、個々のパッキング単位は、一連の関連するアンパック操作によって、ひとまとまりとしてアンパックされる必要がある。

> 根拠 パッキング単位の "分割できない" パックとアンパックの制限によって、パッキング単位の頭に送信側のアーキテクチャ情報を付加することが出来る。(異機種環境での型変換に使用するために。) (根拠の終わり)

以下の関数によって利用者はメッセージをパックするのにどれだけの領域が必要か得ることが出来る。それ故、バッファ領域を管理することが出来る。

MPI_PACK_SIZE(incount, datatype, comm, size)

入力	incount	パック操作に渡す incount 引数 (整数型)
入力	datatype	パック操作に渡す datatype 引数 (ハンドル)
入力	comm	パック操作に渡す comm 引数 (ハンドル)
出力	size	パックされたメッセージの上限値 (バイト単位) (整数型)

```
int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm,
              int *size)

MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERROR)
    INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERROR
```

MPI_PACK_SIZE(incount, datatype, comm, size) は MPI_PACK(inbuf, incount, datatype, outbuf, outcount, position, comm) の結果 position 引数の増加分の上限を size に返す。

根拠 厳密な範囲ではなく、上限を返すのは、メッセージをパックするのに要する領域の大きさは状況に依存する可能性があるからである。(例えば、パッキング単位に最初にパックされたメッセージはより多くの領域を占めるかも知れない。) (根拠の終わり)

例 **3.37** MPI_PACK を使用する例

```
int position, i, j, a[2];
char buff[1000];


....

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0)
{
   / * 送信側コード */

  position = 0;
  MPI_Pack(&i, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
  MPI_Pack(&j, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
  MPI_Send( buff, position, MPI_PACKED, 1, 0, MPI_COMM_WORLD);
}
else  /* 受信側コード */
```

```
  MPI_Recv( a, 2, MPI_INT, 0, 0, MPI_COMM_WORLD)

}
```

例 **3.38** A elaborate example.

```
int position, i;
float a[1000];
char buff[1000]

....

MPI_Comm_rank(MPI_Comm_world, &myrank);
if (myrank == 0)
{
  / * SENDER CODE */

  int len[2];
  MPI_Aint disp[2];
  MPI_Datatype type[2], newtype;

  /* build datatype for i followed by a[0]...a[i-1] */

  len[0] = 1;
  len[1] = i;
  MPI_Address( &i, disp);
  MPI_Address( a, disp+1);
  type[0] = MPI_INT;
  type[1] = MPI_FLOAT;
  MPI_Type_struct( 2, len, disp, type, &newtype);
  MPI_Type_commit( &newtype);

  /* Pack i followed by a[0]...a[i-1]*/

  position = 0;
  MPI_Pack( MPI_BOTTOM, 1, newtype, buff, 1000, &position, MPI_COMM_WORLD);
```

```
  /* Send */

  MPI_Send( buff, position, MPI_PACKED, 1, 0,
            MPI_COMM_WORLD)

/* *****
   One can replace the last three lines with
   MPI_Send( MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD);
   ***** */
}
else /* myrank == 1 */
{
   /* RECEIVER CODE */

  MPI_Status status;

  /* Receive */

  MPI_Recv( buff, 1000, MPI_PACKED, 0, 0, &status);

  /* Unpack i */

 position = 0;
 MPI_Unpack(buff, 1000, &position, &i, 1, MPI_INT, MPI_COMM_WORLD);

 /* Unpack a[0]...a[i-1] */
 MPI_Unpack(buff, 1000, &position, a, i, MPI_FLOAT, MPI_COMM_WORLD);
}
```

例 **3.39** Each process sends a count, followed by count characters to the root; the root concatenate all characters into one string.

```
int count, gsize, counts[64], totalcount, k1, k2, k,
    displs[64], position, concat_pos;
char chr[100], *lbuf, *rbuf, *cbuf;
```

```
...
MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);

      /* allocate local pack buffer */
MPI_Pack_size(1, MPI_INT, comm, &k1);
MPI_Pack_size(count, MPI_CHAR, &k2);
k = k1+k2;
lbuf = (char *)malloc(k);

      /* pack count, followed by count characters */
position = 0;
MPI_Pack(&count, 1, MPI_INT, &lbuf, k, &position, comm);
MPI_Pack(chr, count, MPI_CHAR, &lbuf, k, &position, comm);

if (myrank != root)
      /* gather at root sizes of all packed messages */
   MPI_Gather( &position, 1, MPI_INT, NULL, NULL,
               NULL, root, comm);

      /* gather at root packed messages */
  MPI_Gatherv( &buf, position, MPI_PACKED, NULL,
            NULL, NULL, NULL, root, comm);

else {   /* root code */
      /* gather sizes of all packed messages */
   MPI_Gather( &position, 1, MPI_INT, counts, 1,
               MPI_INT, root, comm);

      /* gather all packed messages */
  displs[0] = 0;
  for (i=1; i < gsize; i++)
    displs[i] = displs[i-1] + counts[i-1];
  totalcount = dipls[gsize-1] + counts[gsize-1];
  rbuf = (char *)malloc(totalcount);
```

```
cbuf = (char *)malloc(totalcount);
MPI_Gatherv( lbuf, position, MPI_PACKED, rbuf,
          counts, displs, MPI_PACKED, root, comm);

    /* unpack all messages and concatenate strings */
concat_pos = 0;
for (i=0; i < gsize; i++) {
  position = 0;
  MPI_Unpack( rbuf+displs[i], totalcount-displs[i],
        &position, &count, 1, MPI_INT, comm);
  MPI_Unpack( rbuf+displs[i], totalcount-displs[i],
        &position, cbuf+concat_pos, count, MPI_CHAR, comm);
  concat_pos += count;
  }
cbuf[concat_pos] = '\0';
}
```

Chapter4

集団通信

4.1 概論と概要

集団通信は、複数のプロセスから成るグループに関連した通信として定義される。 MPI で規定するこのタイプの機能としては次のものがある。

- 全グループ・メンバに跨るバリア同期（4.3節）
- グループ内のあるメンバから全メンバへのブロードキャスト（4.4節）。これは図 4.1に示されている。
- 全グループ・メンバからあるメンバへのデータの **gather** （4.5節）。これは図 4.1に示されている。
- グループ内のあるメンバから全メンバへのデータのスキャッタ **scatter** （4.6 節）。これは、図 4.1に示されている。
- グループ内の全メンバがギャザの結果を受信する **gather** のバリエーション（4.7 節）。これは、図 4.1で “allgather” として示されている。
- グループ内の全メンバから全メンバへのデータの **scatter/gather** （完全交換または **all-to-all** とも呼ばれる）（4.8 節）。これは、図 4.1で “alltoall” として示されている。
- sum、 max、 min、またはユーザー定義関数などの結果を全グループ・メンバへ返却したり、あるメンバだけに返却したりするバリエーションをもつグローバル・リダクション操作（4.9 節）。
- 組み合わせリダクションおよびスキャッタ通信（4.10 節）。
- グループ内の全メンバにまたがるスキャン（プレフィックスとも呼ばれる）（4.11 節）。

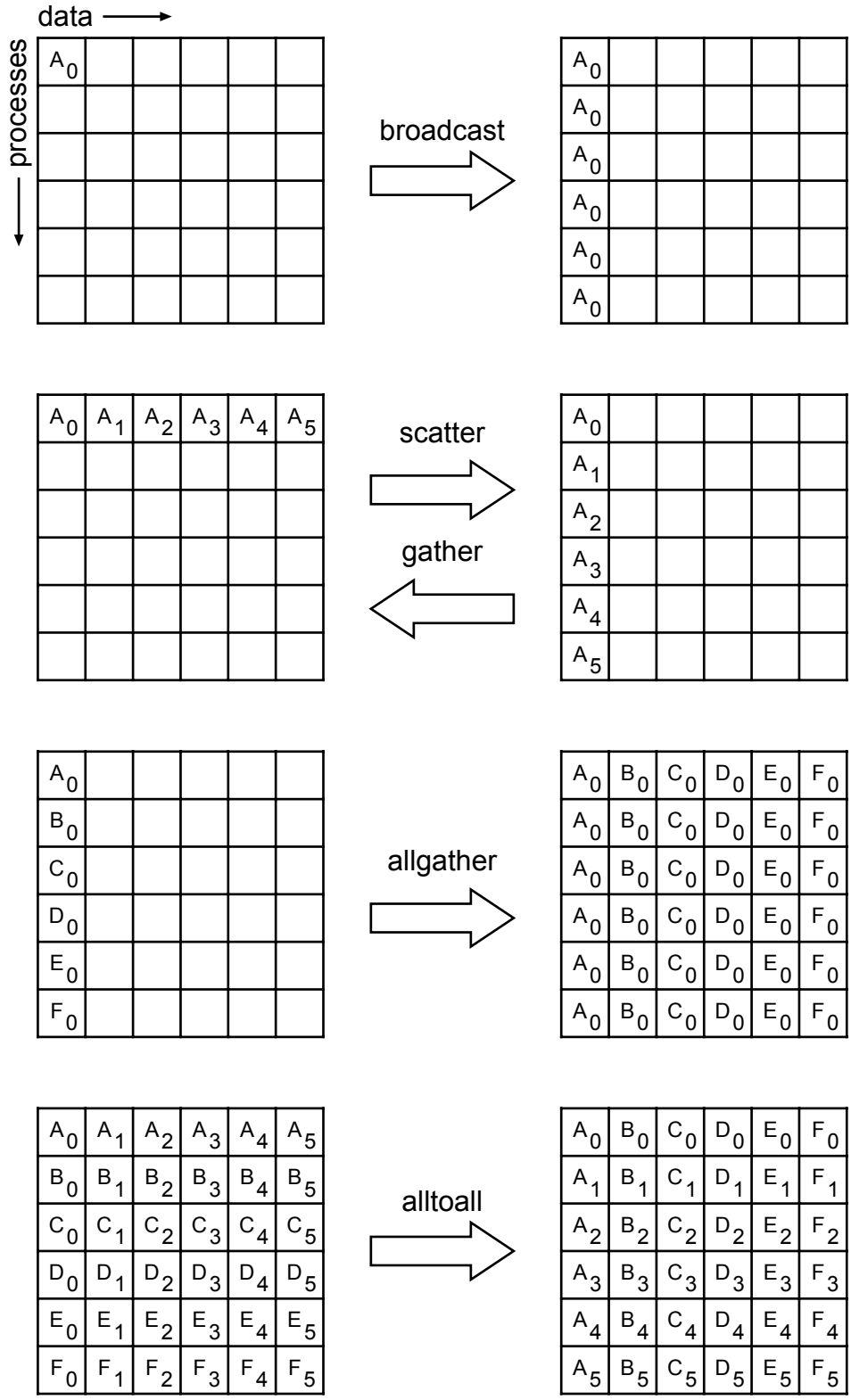

図 4.1: 6 プロセスのグループの集団通信関数の図解。それぞれの場合において、各行はあるプロセスの中のデータ位置を表している。すなわち、ブロードキャストでは、はじめに第 1 プロセスだけがデータ A_0 を持っているが、ブロードキャスト後、全プロセスがそのデータ A_0 を持つ。

集団通信は、グループの中の全プロセスが同一の引数をもって通信ルーチンを呼び出すことで実行される。集団通信の構文および意味は、1 対 1 通信の構文および意味と同じように定義される。したがって、一般のデータタイプが利用可能であり、第 3 章で記述されているように、このタイプは送信側プロセスと受信側プロセスとで一致しなければならない。重要な引数の 1 つに、通信に参加するプロセスのグループを定義し、通信のためのコンテキストを与えるコミュニケータがある。ブロードキャストやギャザといった幾つかの集団通信ルーチンでは、1 つのプロセスだけがメッセージを発信したり受信したりする。そのプロセスはルートと呼ばれる。集団通信関数の引数には、“ルートでのみ意味を持つ” ものがあり、ルートを除く全参加プロセスでは無視される。通信バッファ、一般のデータタイプ、および型一致規則の詳細については第 3 章を、グループの定義の仕方およびコミュニケータの生成の仕方については第 5 章を参照のこと。

集団通信の型一致条件は、1 対 1 通信における送信側と受信側との間の対応条件よりも厳しい。つまり、集団通信では、送信されるデータ量は受信側が指定するデータ量と厳密に一致していなければならない。しかし送信側と受信側とで型マップ（メモリ内のレイアウト、3.12 節を参照）が違っていても許される。

集団通信ルーチンの呼び出しは集団通信への参加が完了するとすぐに戻ることができる（しかし、戻ることを強制するわけではない）。呼び出しの完了は、呼び出しプロセスが自由に通信バッファの領域をアクセスすることを示す。しかし、それはグループ内の他のプロセスが集団通信が完了していることや、あるいは通信操作が開始されているということを示すものではない（集団通信の説明で特記している場合を除く。）。すなわち、集団通信の呼び出しは、全呼び出しプロセスの同期の効果をもたらす場合もあれば、そうでない場合もある。もちろん、この文の記述はバリア関数には適用されない。

集団通信呼び出しでは、1 対 1 通信と同じコミュニケータを使用することができる。MPI では、集団通信呼び出しで作られたメッセージが 1 対 1 通信で作られたメッセージと混同されないことを保証している。集団ルーチンの正しい使用法のより詳細な検討は、4.12 節で記述している。

根拠 データ一致制限（型の対応で）によって、送信データの量を確認するために MPI_RECV の引数 status の解析機能を設けるという煩わしいことを避けるようにした。集団ルーチンのいくつかは、status 値の配列を必要としていた。

集団機能の様々な実装を可能にするため、同期型の文とした。

集団通信関数は、メッセージ・タグの引数を受け入れない。MPI の将来の改訂で非ブロッキング集団通信関数を定義するとすれば、タグ（または類似のメカニズム）を追加し、多重、保留、集団通信のあいまいさを除去できるようにする必要がある。（根拠の終わり）

ユーザへのアドバイス 集団操作の副次作用である同期に頼ることは、正しいプログラムにとって危険である。例えば、一部の実装で同期という副次作用をブロードキャスト・ルー

チンにもたらす場合があるとしても、 MPI の規格ではこの作用を要求していないし、これに依存するプログラムには移植性がない。

その一方、正しく、移植性のあるプログラムは、集団通信関数の呼び出しでは、同期をとっているかもしれないという事を考慮しなければならない。同期によるいかなる副次作用をも頼ることはできないが、その副次作用を考慮してプログラムしなければならない。こうした問題点について 4.12 節で詳述する。（ユーザへのアドバイスの終わり）

実装者へのアドバイス　ベンダーは自社のアーキテクチャに合った最適な集団通信ルーチンを書くことができるし、一方 MPI 1 対 1 通信機能や、いくつかの補助機能を使用するだけによって完璧な集団通信ルーチンのライブラリを書くことができる。1 対 1 通信関数の最初の実装で、集団操作用に隠蔽した専用コミュニケータを作成しておけば、集団操作の呼び出しの時点で、その時実行中の 1 対 1 通信の処理の妨げを回避する。この点についてはさらに 4.12 節で詳述する。（実装者へのアドバイスの終わり）

4.2 コミュニケータ引数

集団機能の重要なコンセプトは、操作に参加するプロセスから成る “グループ” を持つということである。ルーチンは、明示的な引数としてグループの識別子を持たない。その代わりに、コミュニケータ引数がある。本章でのその用途として、コミュニケータはコンテキストと結びついたグループの識別子と考えることができる。 inter-communicator、2 つのグループをつなぐこのコミュニケータは、集団機能の引数としては許されない。

4.3 バリア同期

MPI_BARRIER(comm)

入力	comm		コミュニケータ (handle)

```
int MPI_Barrier(MPI_Comm comm )
```

```
MPI_BARRIER(COMM, IERROR)
    INTEGER COMM, IERROR
```

MPI_BARRIER は全グループ・メンバが呼び出すまで呼び出し側をブロックする。この呼び出しは、全グループ・メンバがその呼び出した後でなければ戻らない。

4.4 ブロードキャスト

MPI_BCAST(buffer, count, datatype, root, comm)

入出力	buffer	バッファの開始アドレス (choice)
入力	count	バッファ内の要素の数 (integer)
入力	datatype	バッファのデータタイプ (handle)
入力	root	ブロードキャスト・ルートのランク (integer)
入力	comm	コミュニケータ (handle)

```
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, M
              PI_Comm comm )
```

```
MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
    <type> BUFFER(*)
    INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
```

MPI_BCAST は、ランクが root であるプロセスからそのプロセスを含むグループ内の全プロセスへメッセージをブロードキャストする。 comm および root として同じ引数を使って全グループ・メンバにより呼び出される。戻った時点で、 root の通信バッファの内容が全プロセスへコピーされている。

一般に datatype として、派生データタイプが許される。どのプロセスの count、 datatype の値もルートの count、 datatype のそれと等しくなければならない。

このことは、各プロセスとルートの間で、送信データの量と受け取るデータの量が等しくなければならないということを意味している。 MPI_BCAST および他のすべてのデータ通信集団ルーチンはこの制約を課している。ただし送信側と受信側とで型マップの異なりだけは許される。

4.4.1 MPI_BCAST の使用例

例 **4.1** プロセス 0 からグループ内のすべてのプロセスへ 100 個の整数をブロードキャストする。

```
MPI\_Comm comm;
int array[100];
int root=0;
...
MPI_Bcast( array, 100, MPI_INT, root, comm);
```

ここで取りあげる一部分のコードの多くは、変数（上記の comm など）に適切な値が代入されているものと仮定している。

4.5 Gather

MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

入力	sendbuf	送信バッファの開始アドレス (choice)
入力	sendcount	送信バッファの要素の数 (integer)
入力	sendtype	送信バッファの要素のデータタイプ (handle)
出力	recvbuf	受信バッファのアドレス (choice, ルートでのみ意味をもつ)
入力	recvcount	number of elements for any single receive (integer, ルートでのみ意味をもつ)
入力	recvtype	受信バッファ要素のデータタイプ (ルートでのみ意味をもつ) (handle)
入力	root	受信プロセスのランク (integer)
入力	comm	コミュニケータ (handle)

```
int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void*
              recvbuf, int recvcount, MPI_Datatype recvtype, int root,
              MPI_Comm comm)

MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE ,
              ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
```

各プロセス（ルートプロセスを含む）は、送信バッファの内容をルートプロセスへ送信する。ルートプロセスはメッセージを受信し、ランクの順番に格納する。その結果は、グループの中の n 個のプロセス（ルートプロセスを含む）が次のルーチンの呼び出しを実行し

$$\texttt{MPI\_Send}(\texttt{sendbuf}, \texttt{sendcount}, \texttt{sendtype}, \texttt{root}, ...),$$

さらに次の呼び出しを n 回実行した結果と同じである。

$$\texttt{MPI\_Recv}(\texttt{recvbuf} + \texttt{i} \cdot \texttt{recvcount} \cdot \texttt{extent}(\texttt{recvtype}), \texttt{recvcount}, \texttt{recvtype}, \texttt{i}, ...),$$

ここで、 extent(recvtype) は MPI_Type_extent() を呼び出して得られる型の大きさである。

言い換えると、グループ内のプロセスが送信した n 個のメッセージをランク順に連結し、得られたメッセージを MPI_RECV(recvbuf, recvcount·n, recvtype, ...) を呼び出して受け取ったかのようにルートが受信する。

ルート以外の全てのプロセスについては受信バッファは無視される。

通常、 sendtype と recvtype には派生データタイプが許される。プロセス i の sendcount、 sendtype は、ルートの recvcount、 recvtype と等しくなければならない。このことは、各プロセスとルートの間で、送信データの量と受け取るデータの量が等しくなければならないということを意味する。ただし送信側と受信側とで型マップの異なりは許される。

root のプロセスでは関数への全ての引数は意味を持つが、他のプロセスでは引数 sendbuf、 sendcount、 sendtype、 root、 comm のみが意味を持つ。引数 root および comm は全てのプロセスで同じ値でなければならない。

個数と型の指定は、ルートプロセス上の同じ位置に複数回書き込まれることがあってはならない。そのような呼び出しはエラーである。

ルートプロセスの引数 recvcount は各プロセスから受信する要素数を示しているのであって、受信する要素の総数を示しているのではないことに注意すること。

MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm)

入力	sendbuf	送信バッファの開始アドレス (choice)
入力	sendcount	送信バッファの要素の数 (integer)
入力	sendtype	送信バッファの要素のデータタイプ (handle)
出力	recvbuf	受信バッファのアドレス (choice, ルートでのみ意味をもつ)
入力	recvcounts	(グループサイズの長さの) 整数配列各プロセスから受け取る要素の数を含んでいる (ルートでのみ意味をもつ)
入力	displs	(グループサイズの長さの) 整数配列。プロセス i から送れて来るデータを置く場所を recvbuf からの相対位置として i 番目の要素に指定 (ルートでのみ意味をもつ)
入力	recvtype	受信バッファ要素のデータタイプ (ルートでのみ意味をもつ) (handle)
入力	root	データを受信するプロセスのランク (integer)
入力	comm	コミュニケータ (handle)

```
int MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype, void
              * recvbuf, int *recvcounts, int *displs,
              MPI_Datatype recvtype, int root, MPI_ Comm comm)

MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS ,
              RECVTYPE, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
    COMM, IERROR
```

MPI_GATHERV は MPI_GATHER の機能を拡張したもので、 recvcounts が配列になっており、各プロセスから可変個のデータを受け取れるようになっている。さらに、新しい引数として displs を提供することにより、ルート上のデータの配置に関して自由度が増している。

結果としては、ルートプロセスを含む各プロセスがメッセージをルートへ送り、

MPI_Send(sendbuf, sendcount, sendtype, root, ...),

ルートプロセスが受信を n 繰り返した場合と同じである。

MPI_Recv(recvbuf + disp[i].extent(recvtype), recvcounts[i], recvtype, i, ...).

メッセージはルートプロセスの受信バッファの中にランク順に配置される。つまり、プロセス j から送られたデータはルートプロセスの受信バッファ recvbuf の j 番目の部分に配置される。recvbuf の j 番目の部分は recvbuf をベースにしたオフセット displs[j] 要素（recvtype の表現で）から始まる。

ルート以外の全てのプロセスでは、受信バッファは無視される。

プロセス i の sendcount、 sendtype は、ルートの recvcounts[i]、 recvtype と等しくなければならない。このことは、各プロセスとルートとの間で、送信データの量が受け取るデータの量と等しくなければならないということを意味する。ただし、例 4.6に示されているように、送信側と受信側とで型マップの違いは許される。

root プロセスでは、全ての引数が意味を持つが、それ以外のプロセスでは引数 sendbuf、 sendcount、 sendtype、 root、 comm のみが意味をもつ。引数 root および comm は全てのプロセスで値が同一でなければならない。

個数と型の指定により、ルートプロセス上の同じ位置に複数回書き込まれることがあってはならない。そのような呼び出しはエラーである。

4.5.1 MPI_GATHER、 MPI_GATHERV の使用例

例 **4.2** グループ内のすべてのプロセスからルートへ 100 個の整数を収集する。図 4.2を参照のこと。

```
MPI_Comm comm;
int gsize,sendarray[100];
int root, *rbuf;
...
MPI_Comm_size( comm, &gsize)
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Gather( sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

例 **4.3** 前の例の手直し－ルートだけが受信バッファ用のメモリを割り当てる。

```
MPI_Comm_rank( comm, myrank);
if ( myrank == root) {
   MPI_Comm_size( comm, &gsize);
   rbuf = (int *)malloc(gsize*100*sizeof(int));
   }
MPI_Gather( sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

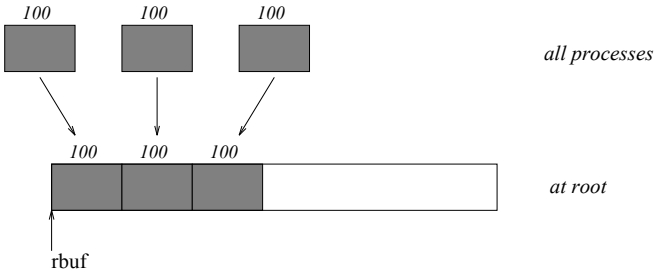

図 4.2: ルートプロセスがグループ内の各プロセスから 100 個の整数を収集する。

例 4.4 前の例と同じことをしているが、派生データタイプを使用している。 gather ではルートプロセスと各プロセスとの間で型対応が行われているので、その型は gsize*100 個の整数の集まりと一致しない。

```
MPI_Comm comm;
int gsize,sendarray[100];
int root, *rbuf;
MPI_Daratype rtype;
...
MPI_Comm_size( comm, &gsize);
MPI_Type_contiguous( 100, MPI_INT, &rtype );
MPI_Type_commit( &rtype );
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Gather( sendarray, 100, MPI_INT, rbuf, 1k rtype, root, comm);
```

例 4.5 各プロセスはルートへ 100 個の整数を送信するが、それぞれを、整数 stride 個分だけの間隔をおいて配置する。 MPI_GATHERV 関数および displs 引数を使用してこの効果を得ることが出来る。 stride ≧ 100 と仮定する。図 4.3 を参照のこと。

```
MPI_Comm comm;
int gsize,sendarray[100];
int root, *rbuf, stride;
int *displs,i,*rcounts;


...

MPI_Comm_size( comm, &gsize);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
```

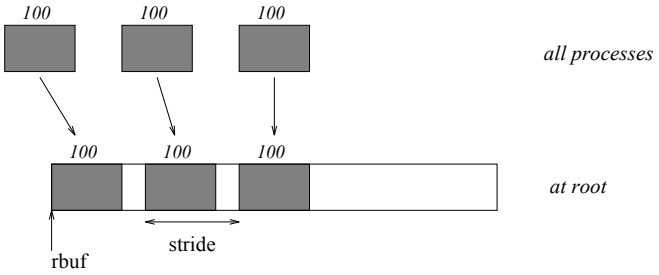

図 4.3: ルートプロセスは、グループ内の各プロセスから 100 個の整数を集め（gather）、各々を整数 stride 個分だけ間隔をおいて配置する。

```
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    rcounts[i] = 100;
}
MPI_Gatherv( sendarray, 100, MPI_INT, rbuf, rcounts, displs, MPI_INT,
                                                          root, comm);
```

stride¡100 だと、プログラムは誤りであることに注意。

例 **4.6** 受信側についての例 4.5と同じ。ただし、 C 言語における 100×150 の整数配列の第 0 列から 100 個の整数を送信する。図 4.4を参照のこと。

```
MPI_Comm comm;
int gsize,sendarray[100][150];
int root, *rbuf, stride;
MPI_Datatype stype;
int *displs,i,*rcounts;

...

MPI_Comm_size( comm, &gsize);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
```

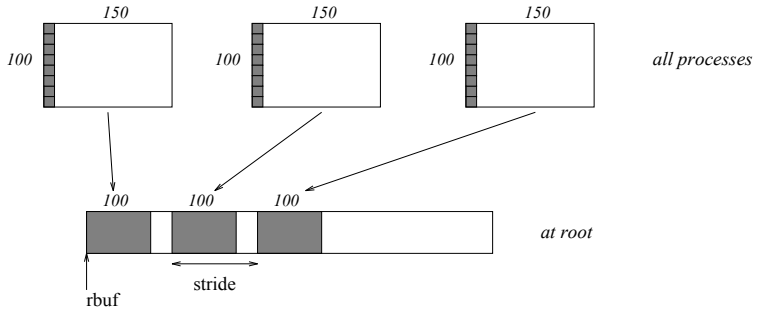

図 4.4: ルート・プロセスは C 言語における 100 × 150 配列の第 0 列を集め, 各々を整数 stride 個分だけ間隔をおいて配置する。

```
        rcounts[i] = 100;
    }
    /* Create datatype for 1 column of array
     */
    MPI_Type_vector( 100, 1, 150, MPI_INT, &stype);
    MPI_Type_commit( &stype );
    MPI_Gatherv( sendarray, 1, stype, rbuf, rcounts, displs, MPI_INT,
                                                          root, comm);
```

例 **4.7** プロセス i は C 言語における 100 × 150 の整数配列の第 i 列から (100-i) 個の整数を送信する。上記２つの例と同様にして、stride 間隔でバッファの中へ読み込む。図 4.5を参照のこと。

```
    MPI_Comm comm;
    int gsize,sendarray[100][150],*sptr;
    int root, *rbuf, stride, myrank;
    MPI_Datatype stype;
    int *displs,i,*rcounts;

    ...

    MPI_Comm_size( comm, &gsize);
    MPI_Comm_rank( comm, &myrank );
    rbuf = (int *)malloc(gsize*stride*sizeof(int));
    displs = (int *)malloc(gsize*sizeof(int));
    rcounts = (int *)malloc(gsize*sizeof(int));
```

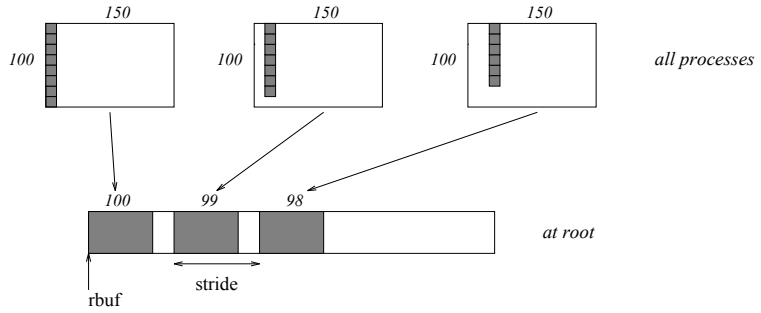

図 4.5: ルートプロセスは 100-i 個の整数を C 言語における 100×100 配列の第 i 列から集め、各々を stride 個の整数だけ離して配置する。

```
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    rcounts[i] = 100-i;     /* note change from previous example */
}
/* Create datatype for the column we are sending
 */
MPI_Type_vector( 100-myrank, 1, 150, MPI_INT, &stype);
MPI_Type_commit( &stype );
/* sptr is the address of start of "myrank" column
 */
sptr = &sendarray[0][myrank];
MPI_Gatherv( sptr, 1, stype, rbuf, rcounts, displs, MPI_INT,
                                                    root, comm);
```

例 **4.8** 例 4.7と同じ。ただし、送信側では別の方法で行われている。送信側で正しいストライドとなるようなデータタイプを生成し、C 言語における配列から 1 列読み込む。**??** 節、例 3.33で行ったのと同じである。

```
MPI_Comm comm;
int gsize,sendarray[100][150],*sptr;
int root, *rbuf, stride, myrank, disp[2], blocklen[2];
MPI_Datatype stype,type[2];
int *displs,i,*rcounts;

...
```

```
    MPI_Comm_size( comm, &gsize);
    MPI_Comm_rank( comm, &myrank );
    rbuf = (int *)malloc(gsize*stride*sizeof(int));
    displs = (int *)malloc(gsize*sizeof(int));
    rcounts = (int *)malloc(gsize*sizeof(int));
    for (i=0; i<gsize; ++i) {
        displs[i] = i*stride;
        rcounts[i] = 100-i;
    }
    /* Create datatype for one int, with extent of entire row
     */
    disp[0] = 0;       disp[1] = 150*sizeof(int);
    type[0] = MPI_INT; type[1] = MPI_UB;
    blocklen[0] = 1;   blocklen[1] = 1;
    MPI_Type_struct( 2, blocklen, disp, type, &stype );
    MPI_Type_commit( &stype );
    sptr = &sendarray[0][myrank];
    MPI_Gatherv( sptr, 100-myrank, stype, rbuf, rcounts, displs, MPI_INT,
                                                          root, comm);
```

例 **4.9** 送信側は例 4.7と同じ。ただし、受信側では受信したブロックの間のストライドはブロックごとに異なる。図 4.6を参照のこと。

```
    MPI_Comm comm;
    int gsize,sendarray[100][150],*sptr;
    int root, *rbuf, *stride, myrank, bufsize;
    MPI_Datatype stype;
    int *displs,i,*rcounts,offset;


    ...

    MPI_Comm_size( comm, &gsize);
    MPI_Comm_rank( comm, &myrank );

    stride = (int *)malloc(gsize*sizeof(int));
    ...
```

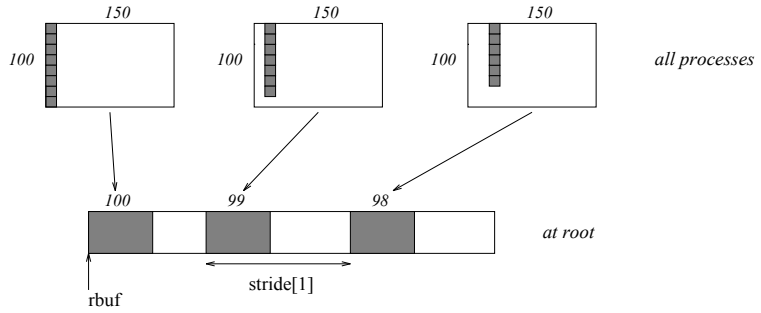

図 4.6: ルート・プロセスは、C 言語における 100×150 配列の第 i 列から 100-i 個の整数を集め、各々整数個 stride[i] 分だけ離して配置する（可変ストライド）。

```
/* stride[i] for i = 0 to gsize-1 is set somehow
 */

/* set up displs and rcounts vectors first
 */
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
offset = 0;
for (i=0; i<gsize; ++i) {
    displs[i] = offset;
    offset += stride[i];
    rcounts[i] = 100-i;
}
/* the required buffer size for rbuf is now easily obtained
 */
bufsize = displs[gsize-1]+rcounts[gsize-1];
rbuf = (int *)malloc(bufsize*sizeof(int));
/* Create datatype for the column we are sending
 */
MPI_Type_vector( 100-myrank, 1, 150, MPI_INT, &stype);
MPI_Type_commit( &stype );
sptr = &sendarray[0][myrank];
MPI_Gatherv( sptr, 1, stype, rbuf, rcounts, displs, MPI_INT,
                                                    root, comm);
```

例 4.10 プロセス i は、C 言語における 100 × 150 の整数配列の第 i 列から num 個の整数を送信する。しかしルートプロセス num の値がわからないので、実際にデータを送信する前に gather を実行して、その値を獲得する。データは、受信側で連続した領域に配置される。

```
MPI_Comm comm;
int gsize,sendarray[100][150],*sptr;
int root, *rbuf, stride, myrank, disp[2], blocklen[2];
MPI_Datatype stype,types[2];
int *displs,i,*rcounts,num;


...

MPI_Comm_size( comm, &gsize);
MPI_Comm_rank( comm, &myrank );

/* First, gather nums to root
 */
rcounts = (int *)malloc(gsize*sizeof(int));
MPI_Gather( &num, 1, MPI_INT, rcounts, 1, MPI_INT, root, comm);
/* root now has correct rcounts, using these we set displs[] so
 * that data is placed contiguously (or concatenated) at receive end
 */
displs = (int *)malloc(gsize*sizeof(int));
displs[0] = 0;
for (i=1; i<gsize; ++i) {
    displs[i] = displs[i-1]+rcounts[i-1];
}
/* And, create receive buffer
 */
rbuf = (int *)malloc(gsize*(displs[gsize-1]+rcounts[gsize-1])
                                                    *sizeof(int));
/* Create datatype for one int, with extent of entire row
 */
disp[0] = 0;       disp[1] = 150*sizeof(int);
type[0] = MPI_INT; type[1] = MPI_UB;
blocklen[0] = 1;   blocklen[1] = 1;
```

```
    MPI_Type_struct( 2, blocklen, disp, type, &stype );
    MPI_Type_commit( &stype );
    sptr = &sendarray[0][myrank];
    MPI_Gatherv( sptr, num, stype, rbuf, rcounts, displs, MPI_INT,
                                                          root, comm);
```

4.6 スキャッタ

MPI_SCATTER (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN	sendbuf	送信バッファのアドレス (選択型、ルートでのみ意味を持つ)
IN	sendcount	各プロセスに送られる要素数 (整数型、ルートでのみ意味を持つ)
IN	sendtype	送信バッファ要素のデータ型 (ルートでのみ意味を持つ)(ハンドル)
OUT	recvbuf	受信バッファのアドレス (選択型)
IN	recvcount	受信バッファ内の要素数 (整数型)
IN	recvtype	受信バッファ要素のデータ型 (ハンドル)
IN	root	送信側プロセスのランク (整数型)
IN	comm	コミュニケータ (ハンドル)

```
int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype,
             void*  recvbuf, int recvcount, MPI_Datatype recvtype,
             int root, MPI_Comm comm)

MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
             ROOT, COMM, IERROR0)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
```

MPI_SCATTER は MPI_GATHER の逆操作である． MPI_SCATTER の効果はルートが次の操作

$$\texttt{MPI\_Send}(\texttt{sendbuf} + \texttt{i} \cdot \texttt{sendcount.extent}(\texttt{sendtype}), \texttt{sendcount}, \texttt{sendtype}, \texttt{i}, \cdots),$$

を n 回繰返し，各プロセスが次の操作を行うのと同じ結果である．

MPI_Recv(recvbuf, recvcount, recvtype, i, ···)

ルートが MPI_Send(sendbuf,sendcount◦n,sendtype,...) によってメッセージを送ることであるとも言い換られる．このメッセージは n 個の等しいセグメントに分割されて i 番目のセグメントが i 番目のプロセスに送られる．各プロセスはこのメッセージを上で述べた通りに受け取る．

全ての非ルートプロセスによる送信バッファへの操作は無効である．

ルートに於ける sendcount, sendtype と全てのプロセスに於ける recvcount, recvtype は型仕様に関してそれぞれ一致していなければならない (しかし，型マップは異なっていてもよい)．各プロセスとルートごとに，送られるデータ群と受け取られるデータ群とが一致しなければならないことを意味している．送信側と受信側の異なった型マップは今のところ許される．

関数の全ての引数はルートに於いて意味を持ち，その他のプロセスで意味を持つのは recvbuf, recvcount, recvtype, root, comm のみである．引数 root, comm については，全てのプロセスが同一の値を持っていなければならない．

count と type の指定によりルートプロセス上の同じ位置から複数回読み出すことがあってはならない．

根拠　必要ではないが，MPI_GATHER との対称性を成すため上述の制約が課されている．対応する MPI_GATHER での制約（多重書き込み制約）は必要である．（根拠の終わり）

MPI_SCATTERV (sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm)

入力	sendbuf	送信バッファのアドレス (選択型、ルートでのみ意味を持つ)
入力	sendcounts	各プロセッサに送る要素の数を指定する (長さグループサイズの) 整数配列
入力	displs	(長さグループサイズの) 整数配列。i 番めのエントリはプロセス i 宛に送るデータを取り出すべき場所の (sendbuf からの) 変位を指定する。
入力	sendtype	送信バッファ要素のデータ型 (ハンドル)
出力	recvbuf	受信バッファのアドレス (選択型)
入力	recvcount	受信バッファの要素数 (整数型)
入力	recvtype	受信バッファ要素のデータ型 (ハンドル)
入力	root	送信側プロセスのランク (整数型)
入力	comm	コミュニケータ (ハンドル)

```
int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs,
             MPI_Datatype sendtype, void* recvbuf, int recvcount,
             MPI_Datatype recvtype, int root, MPI_comm comm)
```

```
MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
             RECVTYPE, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), DISPLS(*),
    SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
```

MPI_SCATTERV は MPI_GATHERV の逆通信である。

MPI_SCATTERV は， MPI_SCATTER の機能を拡張したものであり， sendcounts を配列にして各プロセスに送信されるデータの count を可変にするができる．また，新たな引数 displs の働きにより，ルートプロセス上でのデータの取り出し先に柔軟性を持たせている．

MPI_SCATTERV の効果はルートが次の操作

$$\texttt{MPI\_Send}(\texttt{sendbuf} + \texttt{displs[i]} \cdot \texttt{extent}(\texttt{sendtype}), \texttt{sendcounts[i]}, \texttt{sendtype}, \texttt{i}, \cdots),$$

を n 回繰返し，各プロセスが次の受信を行うのと同じ結果である．

$$\texttt{MPI\_Recv}(\texttt{recvbuf}, \texttt{recvcount}, \texttt{recvtype}, \texttt{i}, \cdots)$$

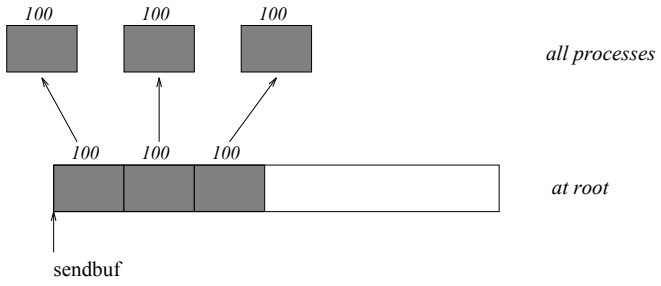

図 4.7: 100 の整数をルートプロセスからグループ内の各プロセスに分散する.

全ての非ルートプロセスによる送信バッファへの操作は無効である.

ルートに於ける sendcounts[i], sendtype と全てのプロセスに於ける recvcount, recvtype は型仕様に関してそれぞれ一致していなければならない (しかし, 型マップは異なっていてもよい). 各プロセスとルートごとに, 送られるデータ群と受け取られるデータ群とが一致しなければならないことを意味している. 送信側と受信側の異なった型マップは今のところ許される.

関数の全ての引数はルートに於いて意味を持ち, その他のプロセスで意味を持つのは recvbuf, recvcount, recvtype, root, comm のみである. 引数 root, comm については, 全てのプロセスが同一の値を持っていなければならない.

count と type の指定によりルートプロセス上の同じ位置から複数回読み出すことがあってはならない.

4.6.1 MPI_SCATTER, MPI_SCATTER の使用例

例 **4.11** Example 4.2 の逆の例. 100 の整数をルートプロセスからグループ内の各プロセスに分散する.

```
MPI_Comm comm;
int gsize,*sendbuf;
int root, rbuf[100];
...
MPI_Comm_size( comm, &gsize);
sendbuf = (int *)malloc(gsize*100*sizeof(int));
...
MPI_Scatter( sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

例 **4.12** Example 4.5 の逆の例. ルートプロセスは 100 の整数を他のプロセスに分散するが, 整数は送信バッファに散在している. このような時は MPI_SCATTERV の使用が求められる. $stride \geq 100$ を仮定している. 図 4.8を見よ.

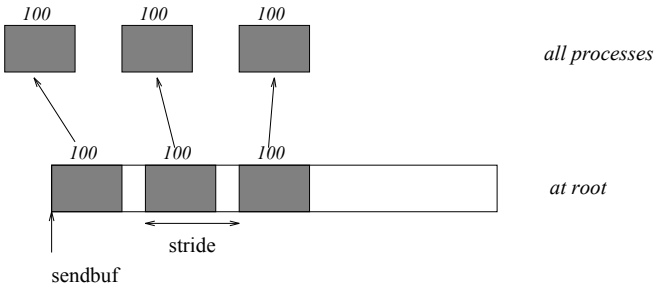

図 4.8: ルートプロセスが scatter 内で変数 stride ごとに移動しながらスキャッタ送信する.

```
MPI_Comm comm;
int gsize,*sendbuf;
int root, rbuf[100], i, *displs, *scounts;

...

MPI_Comm_size( comm, &gsize);
sendbuf = (int *)malloc(gsize*stride*sizeof(int));
...
displs = (int *)malloc(gsize*sizeof(int));
scounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    scounts[i] = 100;
}
MPI_Scatterv( sendbuf, scounts, displs, MPI_INT, rbuf, 100, MPI_INT,
                root, comm);
```

例 **4.13** Example 4.9 の逆の例. 送信側（ルートプロセス）では可変 stride でデータが散在しており，受信側では 100×150 の C 配列の第 i 列に受けとる. 図 4.9を見よ.

```
MPI_Comm comm;
int gsize,recvarray[100][150],*rptr;
into root, *sendbuf, myrank, bufsize, *stride;

MPI_Datatype rtype;
```

```
int i, *displs, *scounts, offset;
...
MPI_Comm_size( comm, &gsize);
MPI_Comm_rank( comm, &myrank );

stride = (int *)malloc(gsize*sizeof(int));
...
/* stride[i] for i = 0 to gsize-1 is set somehow
 * sendbuf comes from elsewhere
 */
...
displs = (int *)malloc(gsize*sizeof(int));
scounts = (int *)malloc(gsize*sizeof(int));
offset = 0;
for (i=0, i<gsize; ++i) {
    displs[i] = offset;
    offset += stride[i];
    scounts[i] = 100 - i;
}
/* Create datatype for the column we are receiving
 */
MPI_Type_vector( 100-myrank, 1, 150, MPI_INT, &rtype);
MPI_Type_commit( &rtype );
rptr = &recvarry[0][myrank];
MPI_Scatterv( sendbuf, scounts, displs, MPI_INT, rptr, 1, rtype, root, comm);
```

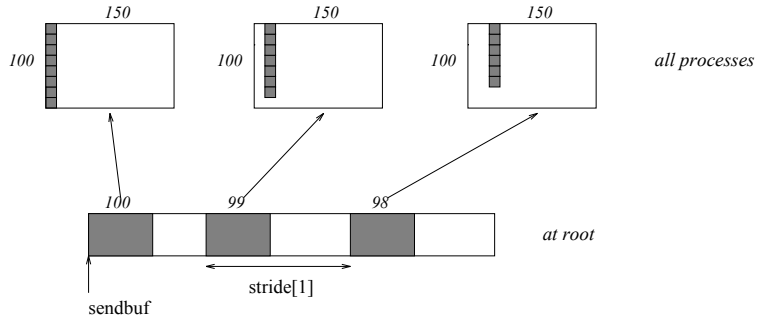

図 4.9: ルートは， 100-i 個の整数ブロックを 100×150 の C 配列の第 i 列へ分散させる．送信側では，ブロックが各 stride[i] 分で散在している。

4.7 全プロセスへのギャザー

MPI_ALLGATHER (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

入力	sendbuf	送信バッファの先頭アドレス (選択型)
入力	sendcount	送信バッファ内の要素数 (整数型)
入力	sendtype	送信バッファ要素のデータ型 (ハンドル)
出力	recvbuf	受信バッファのアドレス (選択型)
入力	recvcount	一プロセスから受信する要素の数 (整数型)
入力	recvtype	受信バッファ要素のデータ型 (ハンドル)
入力	comm	コミュニケータ (ハンドル)

```
int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
              void* recvbuf, int recvcount,
              MPI_Datatype recvtype,MPI_Comm comm)

MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
              COMM, IERROR) ¡type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT,
              SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
```

MPI_ALLGATHER は，ルートプロセスの代わりにすべてのプロセスが結果を受けとる MPI_GATHER と考えて良い．第 j 番目のプロセスから送信されるデータのブロックがすべてのプロセスで受信され，バッファ recvbuf の第 j 番目のブロックとして格納される．

送信元プロセスの sendcount, sendtype と全てのプロセスに於ける recvcount, recvtype は型仕様に関してそれぞれ一致していなければならない．

MPI_ALLGATHER の効果は， root=0,...,n-1 ごとに全プロセスが次の関数呼びだしを n 回行なうのと同じ．

MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm),

MPI_ALLGATHER の正しい使用規則は MPI_GATHER で対応する規則から容易に見い出せる．

MPI_ALLGATHERV (sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm)

入力	sendbuf	送信バッファの先頭アドレス (選択型)
入力	sendcount	送信バッファ内の要素数 (整数型)
入力	sendtype	送信バッファ要素のデータ型 (ハンドル)
出力	recvbuf	受信バッファのアドレス (選択型)
入力	recvcounts	各プロセスから受信する要素数を保持する (長さグループサイズの) 整数配列
入力	displs	(長さグループサイズの) 整数配列。 i 番めのエントリはプロセス i から送られてくるデータを置くべき場所の (recvbuf からの) 変位を指定する。
入力	recvtype	受信バッファ要素のデータ型 (ハンドル)
入力	comm	コミュニケータ (ハンドル)

```
int MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,
              void* recvbuf, int *recvcounts, int *displs,
              MPI_Datatype recvtype, MPI_Comm comm)

MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
              RECVTYPE, COMM, IERROR) ¡type> SENDBUF(*), RECVBUF(*) INTEGER
              SENDCOUNT, SENDTYPE,RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
              IERROR
```

MPI_ALLGATHERV は，ルートプロセスの代わりにすべてのプロセスが結果を受けとる MPI_GATHERV と考えて良い．第 j 番目のプロセスから送信されるデータのブロックがすべてのプロセスで受信され，バッファ recvbuf の第 j 番目のブロックとして格納される．但し，これらのブロックは全て同サイズでなくても良い．

第 j プロセスの sendcount, sendtype と全てのプロセスに於ける recvcounts[j], recvtype は型仕様に関してそれぞれ一致していなければならない.

MPI_ALLGATHERV の効果は， root=0,...,n-1 ごとに全プロセスが次の関数呼びだしを n 回行なうのと同じである.

MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm),

MPI_ALLGATHERV の正しい使用規則は MPI_GATHERV で対応する規則から容易に見い出せる.

4.7.1 MPI_ALLGATHER, MPI_ALLGATHERV の使用例

例 **4.14** 例 *4.2* のオールギャザ版． MPI_ALLGATHER を使って，各プロセスがグループ中の全てのプロセスから *100* の整数を集める.

```
MPI_Comm comm;
int gsize,sendarray[100];
int *rbuf;
...
MPI_Comm_size( comm, &gsize);
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Allgather( sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, comm);
```

呼び出しの後、すべてのプロセスはグループ内のプロセスからの連結したデータを保持する。

4.8 全対全スキャッタ／ギャザ

MPI_ALLTOALL (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

入力	sendbuf	送信バッファの先頭アドレス (選択型)
入力	sendcount	各プロセスに送る要素数 (整数型)
入力	sendtype	送信バッファ要素のデータ型 (ハンドル)
出力	recvbuf	受信バッファのアドレス (選択型)
入力	recvcount	一プロセスから受信する要素数 (整数型)
入力	recvtype	受信バッファ要素のデータタイプ (ハンドル)
入力	comm	コミュニケータ (ハンドル)

```
int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype,
              void* recvbuf, int recvcount, MPI_Datatype recvtype,
              MPI_Comm comm)

MPI_ALLTOTAL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
              COMM, IERROR) ¡type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT,
              SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
```

MPI_ALLTOALL は， MPI_ALLGATHER の機能を，各プロセスが異なるデータを受信プロセスに送ることのできるように拡張したものである．プロセス i から送られる j 番目のブロックは，プロセス j の受信バッファ recvbuf の i 番目のブロックに格納される．

送信元プロセスの sendcount, sendtype と全てのプロセスに於ける recvcount, recvtype は型仕様に関してそれぞれ一致していなければならない．全プロセス組合せごとに，送られるデータ群と受け取られるデータ群とが一致しなければならないことを意味している．しかし，通常は型マップが異なっていても良い．

MPI_ALLTOALL の効果は，次のルーチンの呼び出しで各プロセス（それ自身を含む）への送信を実行し，

$$\texttt{MPI\_Send}(\texttt{sendbuf} + \texttt{i} \circ \texttt{sendcoun} \circ \texttt{extent}(\texttt{sendtype}), \texttt{sendcount}, \texttt{sendtype}, \texttt{i}, \cdots);$$

次のルーチンの呼び出しで他のすべてのプロセスからの受信を実行した場合と同じである．

$$\texttt{MPI\_Recv}(\texttt{recvbuf} + \texttt{i} \circ \texttt{recvcount} \circ \texttt{extent}(\texttt{recvtype}), \texttt{recvcount}, \texttt{i}, \cdots);$$

すべてのプロセスにおいて引数はすべて意味を持つ．引数 comm はすべてのプロセスで同じ値でなければならない．

MPI_ALLTOALLV (sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm)

入力	sendbuf	送信バッファの先頭アドレス (選択型)
入力	sendcounts	各プロセスに送る要素数を指定する、グループサイズに等しい整数配列
入力	sdispls	(長さグループサイズの) 整数配列。j 番めのエントリはプロセス j 宛に送られるデータを取り出す位置の (sendbuf からの) 変位を指定する。
入力	sendtype	送信バッファ要素のデータ型 (ハンドル)
出力	recvbuf	受信バッファアドレス (選択型)
入力	recvcounts	各プロセッサから受信できる要素数を指定する、グループサイズに等しい整数配列
入力	rdispls	(長さグループサイズの) 整数配列。i 番めのエントリはプロセス i から送られてくるデータを置く位置の (recvbuf からの) 変位を指定する。
入力	recvtype	受信バッファ要素のデータ型 (ハンドル)
入力	comm	コミュニケータ (ハンドル)

```
int MPI_Alltotallv(void* sendbuf, int *sendcounts, int *sdisples,
              MPI_Datatype sendtype, void* recvbuf, int *recvcounts,
              int *rdispls, MPI_Datatype recvtype, MPI_Comm comm)

MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,
              RDISPLS, RECVTYPE, COMM, IERROR) ¡type> SENDBUF(*), RECVBUF(*)
              INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*),
              RDISPLS(*), RECVTYPE, COMM, IERROR
```

MPI_ALLTOALLV には，送信されるデータの位置を sdispls によって表し，受信側が受信したデータを格納する位置を rdispls によって表す自由度を付加した MPI_ALLTOALL である．

プロセス i から送られるデータの第 j 番目のブロックは，プロセス j により受信され，受信バッファ recvbuf の第 j 番目のブロックに格納される．これらのブロック全てが同じサイズである必要はない．

プロセス i の sendcount[j], sendtype とプロセス j の recvcount[i], recvtype は型仕様に関してそれぞれ一致していなければならない．全プロセス組合せごとに，送られるデータ群と受け

取られるデータ群とが一致しなければならないことを意味している．しかし，通常は型マップが異なっていても良い．

効果は，各プロセスが次のルーチンで他のすべてのプロセスへメッセージを送信し，

MPI_Send(sendbuf + displs[i] · extent(sendtype), sendcounts[i], sendtype, i, ···),

次のルーチンの呼び出しにより他のすべてのプロセスからメッセージを受信したのと同じものである．

MPI_Recv(recvbuf + displs[i] · extent(recvtype), recvcounts[i], recvtype, i, ···).

すべてのプロセスにおいて，引数はすべて意味を持つ．引数 comm はすべてのプロセスで同じ値でなければならない．

根拠 MPI_ALLTOALL と MPI_ALLTOALLV の定義は，1 対 1 通信を n 回独立に行なうことと同じ効果を示しているが，2 点の例外がある．全てのメッセージは同じ型を持ち，スキャッタ（あるいはギャザ）されるデータの格納先（元）は逐次的である点である．（根拠の終わり）

実装者へのアドバイス　集合通信を 1 対 1 通信の観点から見れば，メッセージは送信側から受信側に直接送る形をとるが，通信経路に木構造を持ったものを実装しても良い．効率が良くなるのであれば，分岐（スキャッタの場合）または合流（ギャザの場合）する点を中継してメッセージをやりとりすることは可能である．（実装者へのアドバイスの終わり）

4.9 大域的なリダクション操作

このセクションの関数は、グループの全メンバに対し大域的なリデュース操作（sum、 max、論理、積など）を実行する。リダクション操作は、定義済み操作の 1 つまたはユーザ定義操作である。大域的なリダクション関数にはいくつかバリエーションがある。 1 つのノードでリダクションの結果を返すリデュース (reduce)、全ノードでこの結果を返すオールリデュース (all-reduce)、およびスキャン（scan: 並列・プレフィックス）操作の三つである。さらに、リデュース - スキャッタ操作はリデュース操作の機能とスキャッタ操作の機能を組み合わせたものである。

4.9.1 Reduce

MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)

入力	sendbuf	送信バッファの先頭アドレス (choice)
出力	recvbuf	受信バッファの先頭アドレス (choice、ルートでのみ意味を持つ)
入力	count	送信バッファ中の要素数 (integer)
入力	datatype	送信バッファ中の要素のデータ型 (handle)
入力	op	演算 (handle)
入力	root	ルートプロセスのランク (integer)
入力	comm	コミュニケータ (handle)

```
int MPI_Reduce(void* sendbuf, void* recvbuf,
               int count,MPI_Datatype datatype, MPI_Op op, into root,
               MPI_Comm comm)

MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
```

MPI_REDUCE は、演算 op を使用してグループ内の各プロセスの入力バッファの中に与えられる要素に対して通信を行ない、その結果をランク root のプロセスの出力バッファの中に返す。入力バッファは、引数 sendbuf、 count、および datatype で定義される。出力バッファは、引数 recvbuf、 count、 datatype で定義される。両者とも同じ型の要素を同じ個数だけ持つ。このルーチンは、 count、 datatype、 op、 root、 comm について同じ引数を使用してすべてのグループ・メンバから呼び出される。したがってすべてのプロセスは同じ型の要素を持つ同じ長さの入力バッファと出力バッファを用意する。各プロセスは、 1 つの要素、または要素の列を与えることができ、要素の列の場合には、要素ごとに操作を行う。例えば、 MPI_MAX という操作で、送信バッファに浮動小数点数の 2 つの要素 (count $=$ 2、 datatype $=$ MPI_FLOAT) が入っている場合、 recvbuf$(1) = \mathrm{globalmax}($sendbuf$(1))$ および、 recvbuf$(2) = \mathrm{globalmax}($sendbuf$(2))$ となる。

4.9.2 節に、 MPI で用意している定義済み操作のリストを掲載した。さらに、各操作を適用できるデータタイプもまとめた。そのほかに、ユーザーは複数のデータタイプについての操作で基本型または派生型のいずれでもオーバーロードできるユーザー用の操作も定義できる。これに

ついては 4.9.4項で詳述する。

演算 op は常に結合則が成立すると仮定する。定義済み操作はすべて可換則が成立すると仮定する。ユーザは結合的であるが可換ではないと仮定された操作を定義することもできる。リダクションの「標準的」評価順序はグループ内のプロセスのランクによって決まる。しかし、実装では結合則、または結合則と可換則を利用して、評価順を変更することもできる。浮動小数点数加算など厳密には結合的、可換的でない操作についてはこの操作によってリダクションの結果が変わることもある。

実装者へのアドバイス　MPI_REDUCE を実装する場合には、同じ順序で現れる同じ引数で関数を適用する際には、必ず同じ結果が得られるようにすることがを強く求められる。このためプロセッサの物理的配置を利用する最適化はできない場合があることに注意されたい。
（実装者へのアドバイスの終わり）

MPI_REDUCE の datatype 引数は op と結合していなければならない。定義済み演算子は 4.9.2節と 4.9.3節であげられている MPI の型でしか機能しない。ユーザー定義演算子は一般的な派生データ型でも機能する。この場合、リデュース操作が適用される各引数はそのような datatype によって記述される 1 つの要素であり、それは複数の基本値を含んでいてもよい。これについてはさらに、 4.9.4項で詳述する。

4.9.2　定義済みリデュース操作

以下の定義済み操作は MPI_REDUCE と関連する関数 MPI_ALLREDUCE、 MPI_REDUCE_SCATTER、MPI_SCAN について用意されている。これらの操作は、 op に次の値を入れることで呼び出される。

名前	意味
MPI_MAX	最大値
MPI_MIN	最小値
MPI_SUM	和
MPI_PROD	積
MPI_LAND	論理積
MPI_BAND	ビット演算の積
MPI_LOR	論理和
MPI_BOR	ビット演算の和

MPI_LXOR	排他的論理和
MPI_BXOR	ビット演算の排他的論理和
MPI_MAXLOC	最大値と位置
MPI_MINLOC	最小値と位置

2 つの操作、 MPI_MINLOC および MPI_MAXLOC については 4.9.3で別に説明する。他の定義済み操作については、引数 op と datatype の組み合わせのうち、許されるものを以下にリストアップする。最初に、次のように MPI 基本データ型のグループを定義する。

C 整数型:	MPI_INT, MPI_LONG,MPI_SHORT MPI_UNSIGNED_SHORT,MPI_UNSIGNED, MPI_UNSIGNED_LONG
Fortran 整数型:	MPI_INTEGER
実数型:	MPI_FLOAT,MPI_DOUBLE, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_LONG_DOUBLE
論理型:	MPI_LOGICAL
複素数型:	MPI_COMPLEX
バイト型:	MPI_BYTE

各操作に対し、有効なデータタイプを以下に示す。

操作	データ型
MPI_MAX, MPI_MIN	C 整数型, Fortran 整数型, 実数型, 複素数
MPI_SUM, MPI_PROD	C 整数型, Fortran 整数型, 実数型, 複素数
MPI_LAND, MPI_LOR, MPI_LXOR	C 整数型, 論理型
MPI_BAND, MPI_BOR, MPI_BXOR	C 整数型, Fortran 整数型, バイト型

例 **4.15** あるグループ中のプロセスに分配される 2 つのベクトルの内積を計算し、ノード 0 に結果を返すルーチン。

```
SUBROUTINE PAR_BLAS1(m, a, b, c, comm)
REAL a(m), b(m)        ! ベクトルの断片
REAL c                 ! 結果（0 番ノード）
REAL sum
INTEGER m, comm, i, ierr
```

```
! 領域の和
sum = 0.0
DO i = 1, m
   sum = sum + a(i)*b(i)
END DO

! 総和
CALL MPI_REDUCE(sum, c, 1, MPI_REAL, MPI_SUM, 0, comm, ierr)
RETURN
```

例 **4.16**　あるグループの中のプロセスに分配されるベクトルと配列の積を計算し、ノード 0 に結果を返すルーチン。

```
SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
REAL a(m), b(m,n)     ! ベクトルの断片
REAL c(n)             ! 結果
REAL sum(n)
INTEGER n, comm, i, j, ierr

! 領域の和
DO j= 1, n
  sum(j) = 0.0
  DO i = 1, m
    sum(j) = sum(j) + a(i)*b(i,j)
  END DO
END DO

! 総和
CALL MPI_REDUCE(sum, c, n, MPI_REAL, MPI_SUM, 0, comm, ierr)

! 0 番ノードに結果が返される（他のノードにはゴミが入っている）
RETURN
```

4.9.3 MINLOC と MAXLOC

演算子 MPI_MINLOC と MAXLOC は、大域的な最小値とこの最小値に位置するインデックスを計算する場合に使用する。 MPI_MAXLOC もこれと同様に、大域的な最大値とそのインデックスを計算する場合に使用する。これの応用の 1 つとして、大域的な最小値（最大値）とこの値を持つプロセスのランクを計算する例をとりあげる。

MPI_MAXLOC を定義する操作は次のとおりである。

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix}$$

ただし、

$$w = \max(u, v)$$

さらに、

$$k = \begin{cases} i & \text{if } u > v \\ min(i, j) & \text{if } u = v \\ j & \text{if } u < v \end{cases}$$

MPI_MINLOC も同様に定義する。

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ l \end{pmatrix}$$

ただし、

$$w = min(u, v)$$

さらに、

$$k = \begin{cases} i & \text{if } u > v \\ min(i, j) & \text{if } u = v \\ j & \text{if } u < v \end{cases}$$

この操作は両者とも結合則と可換則が成立する。 MPI_MAXLOC を $(u_0、0), (u_1, 1), \ldots, (un-1;n-1)$ のペアの列に対するリデュースを行なうと、戻り値は (u, r) となることに注意されたい。ここで、 $u = \max_i u_i$ と r はこの列の中の最初の大域的な最大値のインデックスである。したがって、各プロセスが値とそのグループ内におけるランクを与えるとすると、 op=MPI_MAXLOC としたリデュース操作は最大値とその値を持つ最初のプロセスのランクを返す。同様に、 MPI_MINLOC を使用して最小値とそのインデックスを返すことができる。より一般的には、 MPI_MINLOC は

辞書順での最小値を求めるということであり、各ペアの最初の要素に応じて順序づけられた状態では 2 番目の成分により決定される。

リデュース操作は、値とインデックスの組からなる引数に対する操作と定義される。 Fortran 言語と C 言語とでは、型はこのペアを記述するように与えられる。 Fortran 言語では、引数がもともと混合型であるという性質が問題となる。 Fortran 言語の場合この問題は、インデックスを値と同じ型に強制的に変換し、値と同じ型を持つペアから成る型を MPI が提供することで回避されている。 C では、 MPI の提供する型は別々の型のペアであり、インデックスは int 型である。 リデュース操作の中で MPI_MINLOC および MPI_MAXLOC を使用するためには、ペア（値とインデックス）を表す datatype 引数を与えなければならない。 MPI はこのような定義済みデータタイプを 7 個用意している。 MPI_MINLOC と MPI_MAXLOC の操作を次のデータタイプとともに使用することができる。

Fortran:

データタイプ	記述
MPI_2REAL	REAL 型の対
MPI_2DOUBLE_PRECISION	DOUBLE 型の対
MPI_2INTEGER	INTEGER 型の対

C:

データタイプ	記述
MPI_FOLAT_INT	float と int
MPI_DOUBLE_INT	double と int
MPI_LONG_INT	long と int
MPI_2INT	pair と int
MPI_SHORT_INT	short と int
MPI_LONG_DOUBLE_INT	long double と int

データタイプ MPI_2REAL は次のように定義されているのと同様である（ 3.12節を参照）。

```
MPI_TYPE_CONTIGUOUS(2, MPI_REAL, MPI_2REAL)
```

MPI_2INTEGER、 MPI_2DOUBLE_PRECISION、 MPI_2INT についても同様のことがいえる。データタイプ MPI_FLOAT_INT は、次の命令群によって定義されているのと同じである。

```
type[0] = MPI_FLOAT
type[1] = MPI_INT
disp[0] = 0
```

```
disp[1] = sizeof(float)
block[0] = 1
block[1] = 1
MPI_TYPE_STRUCT(2, block, disp, type, MPI_FLOAT_INT)
```

MPI_LONG_INT および MPI_DOUBLE_INT についても同様のことがいえる。

例 **4.17** 各プロセスは C 言語における 30 個の double の配列を持つ。30 個の double のそれぞれについて、最大値とその値を持つプロセスのランクを計算する。

```
/* 各プロセスは 30 個の double: ain[30]
 */
 double ain[30], aout[30];
 int  ind[30];
 struct {
     double val;
     int   rank;
 } in[30], out[30];
 int i, myrank, root;

 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
 for (i=0; i<30; ++i) {
     in[i].val = ain[i];
     in[i].rank = myrank;
 }
 MPI_Reduce( in, out, 30, MPI_DOUBLE_INT, MPI_MAXLOC, root, comm );
 /* ここで、結果はルートプロセスにある。
  */
 if (myrank == root) {
     /* ランクを読み出す
      */
     for (i=0; i<30; ++i) {
         aout[i] = out[i].val;
             ind[i] = out[i].rank;
         }
    }
```

例 **4.18** Fortran での同様の例

```
! 各プロセスは長さ 30 の DOUBLE 型配列: ain(30)


DOUBLE PRECISION ain(30), aout(30)
INTEGER ind(30);
DOUBLE PRECISION in(2,30), out(2,30)
INTEGER i, myrank, root, ierr;

MPI_COMM_RANK(MPI_COMM_WORLD, myrank);
DO I=1, 30
    in(1,i) = ain(i)
    in(2,i) = myrank    ! 自身のランクを強制的に DOUBLE 型にする
END DO

MPI_REDUCE( in, out, 30, MPI_2DOUBLE_PRECISION, MPI_MAXLOC, root, comm, ierr );

! ここで、結果はルートプロセスにある。


IF (myrank .EQ. root) THEN
    ! read ranks out
    DO I= 1, 30
        aout(i) = out(1, i)
        ind(i) = out(2,i)   ! INTEGER 型に戻す。
    END DO
END IF
```

例 **4.19** 各プロセスは値の空でない配列を持つ。大域的な最小値、その値を保持するプロセスのランク、このプロセス上でのインデックスを見つける。

```
#define  LEN   1000

float val[LEN];         /* ローカルな配列 */
int count;              /* ローカルな配列の値の数 */
int myrank, minrank, minindex;
float minval;

```

```
struct {
    float value;
    int   index;
} in, out;

    /*  局所的最小値と、その位置 */
in.value = val[0];
in.index = 0;
for (i-1; i < count; i++)
    if (in.value > val[i]) {
        in.value = val[i];
        in.index = i;
    }

    /* 大域的最小値と、その位置 */
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
in.index = myrank*LEN + in.index;
MPI_Reduce( in, out, 1, MPI_FLOAT_INT, MPI_MINLOC, root, comm );
    /* ここで、結果はルートプロセスにある。
     */
if (myrank == root) {
    /* 結果を読みだす。

     */
    minval = out.value; /* 最小値 */
    minrank = out.index / LEN; /* 最小値が存在するノードのランク */
    minindex = out.index % LEN; /* 最小値が存在する配列のインデックス */
}
```

根拠　ここで与えた MPI_MINLOC および MPI_MAXLOC の定義には、これら 2 つの操作を特別に扱う必要がないという利点がある。他のリデュース操作のように処理すればよいのである。プログラマは MPI_MAXLOC および MPI_MINLOC の独自の定義を必要に応じて与えることができる。欠点としては、値とインデックスのペアを最初に作って置かなければならないという点と、更に Fortran 言語の場合に限り、インデックスと値を同じ型に強制変換しなければならないという点があげられる。

（根拠の終わり）

4.9.4 ユーザー定義操作

MPI_OP_CREATE(function, commute, op)

入力	function	ユーザ定義関数 (関数)
入力	commute	もし true なら可換則、false ならそうでない
出力	op	演算 (handle)

```
int MPI_Op_create(MPI_User_function *function, int commute, MPI_Op *op)

MPI_OP_CREATE( FUNCTION, COMMUTE, OP, IERROR)
    EXTERNAL FUNCTION
    LOGICAL COMMUTE
    INTEGER OP, IERROR
```

MPI_OP_CREATE は、ユーザー定義の大域的操作を op ハンドルにバインドし、その後、これを MPI_REDUCE、MPI_ALLREDUCE、MPI_REDUCE_SCATTER、MPI_SCAN で使用することができる。ユーザー定義操作は結合的であると仮定される。commute=true であれば、この操作は可換かつ結合的でなければならない。commute=false であれば、操作順序は固定され、プロセスランクの昇順でプロセス 0 から始まると定義される。操作の結合的性質から操作の実行順序を変更することが可能である。また、commute=true の場合は、さらに可換的性質を利用して実行順序を変更することが可能である。function は、ユーザー定義関数であり、invec、inoutvec、len、および datatype の四つの引数を持たなければならない。
この関数の ANSI-C プロトタイプは次のとおりである。

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

ユーザー定義関数の Fortran 言語での宣言は次のとおりである。

```
FUNCTION USER_FUNCTION( INVEC(*), INOUTVEC(*), LEN, TYPE)
<type> INVEC(LEN), INOUTVEC(LEN)
INTEGER LEN, TYPE
```

datatype 引数は、MPI_REDUCE 関数に渡されるデータ型のハンドルである。ユーザー定義のリデュース関数を作成する場合には、次のことが成立するようにしなければならない。u[0], ... ,u[len-1] をこの関数を呼び出すときの引数 invec、len、datatype で記述される通信バッファの中の len 個の要素とする。v[0],...,v[len-1] をこの関数が呼び出されるときの引数 inoutvec、len、

datatype で記述される通信バッファの中の len 個の要素とする。 w[0],...,w[len-1] をこの関数から戻るときの引数 inoutvec、 len、 datatype で記述される通信バッファの中の len 個の要素とする。 i=0,...,len-1 について w[i]=u[i]∘v[i] である。ここで、 ∘ は関数が計算するリデュース操作である。

invec、 inoutvec を function が演算する len 個の要素の配列と考えることもできる。リダクションの結果、 inoutvec の内容を上書きする。この関数の呼び出しにより、 len 個の各要素についてリデュース操作が行われる。つまり、関数は i=0,...,count-1 について inoutvec[i] に値 invec[i]∘ inoutvec[i] を返す。ここで、 ∘ はこの関数が行う演算である。

根拠　 len 引数を使用することで、 MPI_REDUCE が入力バッファ中の要素ごとに操作を行う関数を呼び出さずに済むようになる。さらに言えば、システム側で入力データに対する関数の適用方法を選べるということである。 C 言語では、 Fortran 言語との互換性から参照渡しとしている。 datatype 引数の値と既知の大域的なハンドルとを内部で比較することで、 1 つのユーザー定義関数を複数の異なるデータ型についてオーバーロードして使用することが可能である。
（根拠の終わり）

一般データタイプをユーザー定義関数にわたすことができる。ただし、連続していないデータタイプを使用すると、効率が低下するおそれがある。

ユーザー定義関数内部で MPI 操作関数を呼び出すことはできないが、エラーが発生した場合関数の内部で MPI_ABORT を呼び出すことができる。

ユーザへのアドバイス　 オーバロードされるユーザー定義リデュース関数のライブラリを定義するとする。 datatype 引数は、オペランド型に応じて正しい関数を選択するのに利用される。ユーザー定義リデュース関数は渡される datatype 引数が何を表しているか判断することはできず、またデータタイプ・ハンドルとそれが表すデータタイプとの対応関係を識別することもできない。この対応関係は、データタイプを生成した時点で確定している。ライブラリを使用する前に、ライブラリを初期化しなければならない。この初期化ルーチンよって、ライブラリが使用するデータタイプを定義し、これらのデータタイプのハンドルをユーザー・コードとライブラリ・コードで共有するスタティックな大域変数に格納する。

MPI_REDUCE の Fortran バージョンは、 Fortran の呼び出し規約を使用してユーザー定義リデュース関数を呼び出し、 Fortran タイプのデータタイプ引数をわたす。 C バージョンでは C の呼び出し規約でデータタイプ・ハンドルの C 言語の表現を使用する。両方を同時に使用することをを予定しているユーザーの場合には、それに応じたリダクション関数の定義を行わなければならない。
（ユーザへのアドバイスの終わり）

実装者へのアドバイス　MPI_REDUCE の単純なミスによる不効率な実装を以下に示す。

```
if (rank > 0) {
    RECV(tempbuf, count, datatype, rank-1,...)
    User_reduce( tempbuf, sendbuf, count, datatype)
}
if (rank < groupsize-1) {
    SEND( sendbuf, count, datatype, rank+1, ...)
}
/* 結果は groupsize-1 番のノードにある ... 直ちにルートに送る。
 */
if (rank == groupsize-1) {
    SEND( sendbuf, count, datatype, root, ...)
}
if (rank == root) {
    RECV(recvbuf, count, datatype, groupsize-1,...)
}
```

リダクション演算はプロセス 0 からプロセス group- size-1 まで順番に進行する。この順序は、User_reduce() 関数によって定義される演算子が非可換でない場合を考慮して順序を選択する。

結合性を利用し、さらにトリー的にリダクションを行うことにより効率のよい実装を行うことができる。MPI_OP_CREATE への commute 引数が真の場合、結合性を利用できる。さらに、サイズ len<count の単位で要素を転送、リデュースすることにより必要な一時バッファの大きさをリデュースし、操作を計算とパイプライン化することができる。

定義済みリダクション操作はユーザー定義操作のライブラリとして実装できる。しかし、MPI_REDUCE でこれらの関数を特別な場合として処理するようにすれば性能が向上する可能性もある。

（実装者へのアドバイスの終わり）

MPI_OP_FREE(op)

入力	op	演算 (handle)

```
int MPI_op_free( MPI_Op *op)

MPI_OP_FREE( OP, IERROR)
```

```
    INTEGER OP, IERROR
```

解放するユーザー定義リダクション操作をマークし、 op を MPI_OP_NULL に設定する。

4.9.5 ユーザー定義リデュースの例

ここでユーザー定義リダクション操作の例を取りあげる。

例 **4.20** C 言語で複素数の配列の積を計算する。

```
typedef struct {
    double real,imag;
} Complex;

/* ユーザ定義関数
 */
void myProd( Complex *in, Complex *inout, int *len, MPI_Datatype *dptr )
{
    int i;
    Complex c;

    for (i=0; i< *len; ++i) {
        c.real = inout->real*in->real-
                    inout->imag*in->imag.
        c.imag = inout->real*in->imag +
                    inout->imag*in->real;
        *inout = c;
        in++; inout++;
    }
}

/* そして，それを呼びだす...
 */
...
    /* 各プロセスは長さ 100 の復素数配列を持つ
     */
    Complex a[100], answer[100];
```

```
    MPI_Op myOp;
    MPI_Datatype ctype;

    /* MPI に対して Complex 型とはどのような型かを知らせる。
     */
    MPI_Type_contiguous( 2, MPI_DOUBLE, &ctype );
    MPI_Type_commit( &ctype );
    /* ユーザ定義で復素数の積を定義する。
     */
    MPI_Op_create( myProd, True, &myOp );

MPI_Reduce( a, answer, 100, ctype, myOp, root, comm );

/* ここで、結果は 100 個の復素数でルートプロセスにある。
 */
```

4.9.6 All-Reduce

MPI では、演算結果がグループのすべてのプロセスへ返されるリデュース操作のバリエーションが用意されている。 MPI では、これらの操作に関わるすべてのプロセスが同一の結果を受け取ることが要求されている。

MPI_ALLREDUCE (sendbuf, recvbuf, count, datatype, op, comm)

入力	sendbuf	送信バッファの先頭アドレス (choice)
出力	recvbuf	受信バッファの先頭アドレス (choice)
入力	count	送信バッファ中の要素数 (integer)
入力	datatype	送信バッファ中の要素のデータ型 (handle)
入力	op	演算 (handle)
入力	comm	コミュニケータ (handle)

```
int MPI_Allreduce(void *sendbuf, void *recvbuf, int count,
              MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
    <type> SENDBUF(8), RECVBUF(*)
```

```
INTEGER COUNT, DATA TYPE, OP, COMM, IERROR
```

結果が全グループ・メンバの受信バッファに格納される点を除き、MPI_REDUCE と同じ。

実装者へのアドバイス　オールリデュース操作は、リダクションの後にブロードキャストが続くものとして実装することができる。しかし、直接実装したほうが効率がよいと思われる。
（実装者へのアドバイスの終わり）

例 **4.21**　ベクトルと、グループ内のプロセスに分散した配列の積を計算し、全ノードに結果を返すルーチン（4.16節を参照）。

```
SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
REAL a(m), b(m,n)     ! ベクトルの断片
REAL c(n)             ! 結果
REAL sum(n)
INTEGER n, comm, i, j, ierr

! 領域の和
DO j=1, n
   sum(j) = 0.0
   DO i = 1, m
     sum(j) = sum(j) + a(i)*b(i,j)
    END DO
END DO

! 総和
CALL MPI_ALLREDUCE(sum, c, n, MPI_REAL, MPI_SUM, 0, comm, ierr)

! 結果は全てのノードに返される
RETURN
```

4.10　Reduce-Scatter 通信

MPI では、reduce 演算のバリエーションが用意されている。そこでは、戻る際に演算結果がグループ内の全プロセスに scatter される。

MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm)

入力	sendbuf	送信バッファの開始アドレス (choice)
出力	recvbuf	受信バッファの開始アドレス (choice)
入力	recvcounts	各々のプロセスに分配される結果の要素数を指定する整数型配列。配列は、すべての呼び出しプロセスについて同じでなければならない。
入力	datatype	入力バッファの要素のデータ型 (handle)
入力	op	演算 (handle)
入力	comm	コミュニケータ (handle)

```
int MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int* recvcounts,
              MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
               IERROR)
    <type> SENDBUF(*), RECBBUF(*)
    INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR
```

MPI_REDUCE_SCATTER はまず、 sendbuf、 count、および datatype で定義される送信バッファの中の count $=\sum_i$ recvcounts[i] 個の要素からなるベクトルに対し要素ごとのリダクションを実行する。次に、得られた結果のベクトルを n 個の非連結セグメントに分割する。ここで n はグループ内のメンバ数である。セグメント i には、 recvcounts[i] 個の要素が入る。第 i セグメントがプロセス i に送信され、 recvbuf、 recvcounts[i]、および datatype で定義される受信バッファの中に格納される。

実装者へのアドバイス　MPI_REDUCE_SCATTER ルーチンは機能的には、 count が recvcounts[i] の総和に等しい MPI_REDUCE 演算関数の後に sendcounts が recvcounts に等しい MPI_SCATTERV が続くものと同じである。
ただし、直接実装したほうが高速になる場合がある。
（実装者へのアドバイスの終わり）

4.11 Scan

MPI_SCAN (sendbuf, recvbuf, count, datatype, op, comm)

入力	sendbuf	送信バッファの開始アドレス (choice)
出力	recvbuf	受信バッファの開始アドレス (choice)
入力	count	入力バッファの要素数 (integer)
入力	datatype	入力バッファの要素のデータ型 (handle)
入力	op	演算 (handle)
入力	comm	コミュニケータ (handle)

```
int MPI_Scan(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm )
```

```
MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR
```

MPI_SCAN は、グループ中に分配されたデータに対しプレフィックスリダクションを実行する場合に使用する。この演算は、ランク i のプロセスの受信バッファの中に、ランク 0,...,i までのプロセスの送信バッファの中の値のリダクションを返す。サポートされている演算の種類や意味、および送信、受信バッファに対する制約は MPI_REDUCE の場合と同様である。

根拠 包括的 scan を定義した。つまり、プロセス i に対するプレフィックスリダクションはプロセス i からのデータも含む。代わりに排他的に scan を定義することも考えられる。その場合、i に対する結果はプロセス i-1 までのデータを含むだけである。この 2 つの定義はともに有用である。後者にはいくつか利点がある。包括的 scan は常に、通信を付加しないで排他的 scan から計算することができる。 max や min などの不可逆な演算の場合、包括的 scan から排他的 scan を計算するのに通信が必要となる。しかし、排他的 scan の場合、「単位」要素を定義しなければならないという問題が生じる。つまり、プロセス 0 に対し何が発生するかを明示しなければならないのである。このことは、利用者定義演算には複雑だと考えられたため、排他的 scan は採用されなかった。
（根拠の終わり）

4.11.1 MPI_SCAN の使用例

例 **4.22** この例では、利用者定義演算を使用して部分 *(segment) scan* を行う。部分 scan は、入力として値の集合および論理値の集合をとり、その論理値により scan の様々なセグメントを区分けする。例えば、

値	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
論理値	0	0	1	1	1	0	0	1
結果	v_1	$v_1 + v_2$	v_3	$v_3 + v_4$	$v_3 + v_4 + v_5$	v_6	$v_6 + v_7$	v_8

この結果をもたらす演算子は次のとおりである。

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ j \end{pmatrix},$$

ここで、

$$w = \begin{cases} u + v & \text{if } i = j \\ v & \text{if } i \neq j \end{cases}.$$

これは非可換演算であることに注意されたい。これを実装する C 言語によるコードは次のとおりである。

```
typedef struct {
    double val;
    int log;
} SegScanPair;

/* the user-defined function
 */
void segScan( SegScanPair *in, SegScanPair *inout, int *len,
MPI_Datatype *dptr )
{
    int i;
    SegScanPair c;

    for (i=0; i< *len; ++i) {
        if ( in->log == inout->log )
            c.val = in->val + inout->val;
        else
            c.val = inout->val;
```

```
        c.log = inout->log;
        *inout = c;
        in++; inout++;
    }
}
```

利用者定義関数の inout 引数は演算子の右辺に対応することに注意。この演算子を使用する場合には、次のように非可換であることを指定するのに注意しなければならない。

```
int i, base;
SeqScanPair  a, answer;
MPI_Op       myOp;
MPI_Datatype type[2] = {MPI_DOUBLE, MPI_INT};
MPI_Aint     disp[2];
int          blocklen[2] = { 1, 1};
MPI_Datatype sspair;

/* explain to MPI how type SegScanPair is defined
 */

MPI_Address( a, disp);
MPI_Address( a.log, disp+1);
base = disp[0];
for (i=0; i<2; ++i) disp[i] -= base;
MPI_Type_struct( 2, blocklen, disp, type, *sspair );
MPI_Type_commit( &sspair );
/* create the segmented-scan user-op
 */
MPI_Op_create( segScan, False, &myOp );
...
MPI_Scan( a, answer, 1, sspair, myOp, root, comm );
```

4.12 正当性

正しくかつ移植性のあるプログラムは、集団通信を同期的に行うか否かにかかわらず、デッドロックが発生しないように集団通信を呼び出さなければならない。次の例は、集団通信ルーチン

の危険な使用例である。

例 **4.23** 以下の例は誤りである。

```
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Bcast(buf2, count, type, 1, comm);
        break;
    case 1:
        MPI_Bcast(buf2, count, type, 1, comm);
        MPI_Bcast(buf1, count, type, 0, comm);
        break;
}
```

comm のグループは $\{0,1\}$ であると仮定する。2 つのプロセスは逆順で 2 つのブロードキャスト演算を実行する。演算が同期していると、デッドロックが発生する。

集団通信は通信を行うグループのすべてのメンバで同じ順序で実行されなければならない。

例 **4.24** 以下の例は誤りである。

```
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm0);
        MPI_Bcast(buf2, count, type, 2, comm2);
        break;
    case 1:
        MPI_Bcast(buf1, count, type, 1, comm1);
        MPI_Bcast(buf2, count, type, 0, comm0);
        break;
    case 2:

        MPI_Bcast(buf1, count, type, 2, comm2);
        MPI_Bcast(buf2, count, type, 1, comm1);
        break;
}
```

comm0 のグループは {0,1}、comm1 のグループは {1,2}、comm2 のグループは {2,0} であると仮定する。ブロードキャストが同期演算であれば、巡回依存性がある。つまり、comm2 のブロードキャストは comm0 のブロードキャスト後でしか完了せず、comm0 のブロードキャストは comm1 のブロードキャストの後でしか完了せず、comm1 のブロードキャストは comm2 のブロードキャストの後でしか完了しない。このように、このコードはデッドロックを発生することになる。

集団通信は、巡回依存性が発生しない順序で実行しなければならない。

例 **4.25** 以下の例は誤りである。

```
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Send(buf2, count, type, 1, tag, comm);
        break;
    case 1:
        MPI_Recv(buf2, count, type, 0, tag, comm);
        MPI_Bcast(buf1, count, type, 0, comm);
        break;
}
```

プロセス 0 は、ブロードキャストを実行し、その後ブロッキング送信操作を行う。プロセス 1 はまず送信と一致するブロッキング受信を実行し、その後プロセス 0 のブロードキャストと一致するブロードキャスト呼び出しを実行する。このプログラムもデッドロックを発生する可能性がある。プロセス 0 でのブロードキャスト呼び出しは、プロセス 1 が一致するブロードキャスト呼び出しを実行するまでブロックする場合があるので、送信は実行されない。プロセス 1 は明らかに受信でブロックするので、この場合ブロードキャストを実行することはない。

集団演算と 1 対 1 演算の実行の相対的な順序はこのようにすべきである。それは集団通信と 1 対 1 通信とが同期しても、デッドロックを引き起こさないようにするためである。

例 **4.26** 正しいけれども非決定性のプログラム。

```
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Send(buf2, count, type, 1, tag, comm);
        break;
```

```
    case 1:
        MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm);
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm);
        break;
    case 2:
        MPI_Send(buff2, count, type, 1, tag, comm);
        MPI_Bcast(buf1, count, type, 0, comm);
        break;
}
```

3 つのプロセスはすべてブロードキャストに参加している。プロセス 0 はブロードキャスト後、メッセージをプロセス 1 に送信し、プロセス 2 はブロードキャストの前にメッセージをプロセス 1 に送信する。プロセス 1 は引数にワイルドカードで送信元を指定して、ブロードキャストの前後に受信する。

このプログラムには送受信の一致方法の違いで 2 通りの実行の仕方がある。これを 4.10に示す。ここで、2 番目の実行は、ブロードキャスト後の送信が、他のノードにおいてブロードキャスト前に受信されるという、特殊な結果になる。この例は、特定の同期効果を持つ集団通信関数に依存すべきではないという事実を証明している。1 番目の実行が起こる時のみ（ブロードキャストが同期をとる時のみ）正常に動作するプログラムは誤りである。

最後に、マルチスレッド実装では、1 つのプロセスで 1 つ以上の同時に実行される集団通信呼び出しがあるかもしれない。このような状況では利用者側の責任で同じプロセスで 2 つの異なる集団通信呼び出しが同時に同じコミュニケータを使用しないようにする必要がある。

実装者へのアドバイス　1 対 1 通信を使用してブロードキャストを実装していると仮定する。この時は、以下の 2 つの規則に従わなければならない。

1. すべての受信でその送信元を明示する（ワイルドカードを使用しない）。
2. 各プロセスは後の集団通信に関連するメッセージを送信する前に、一つの集団通信に関連するすべてのメッセージを送信する。

これで、1 対 1 メッセージの順序が保存されているので、後続のブロードキャストに属するメッセージが混同することはありえない。

1 対 1 通信のメッセージと集団通信のメッセージを混同しないようにするのは実装者の責

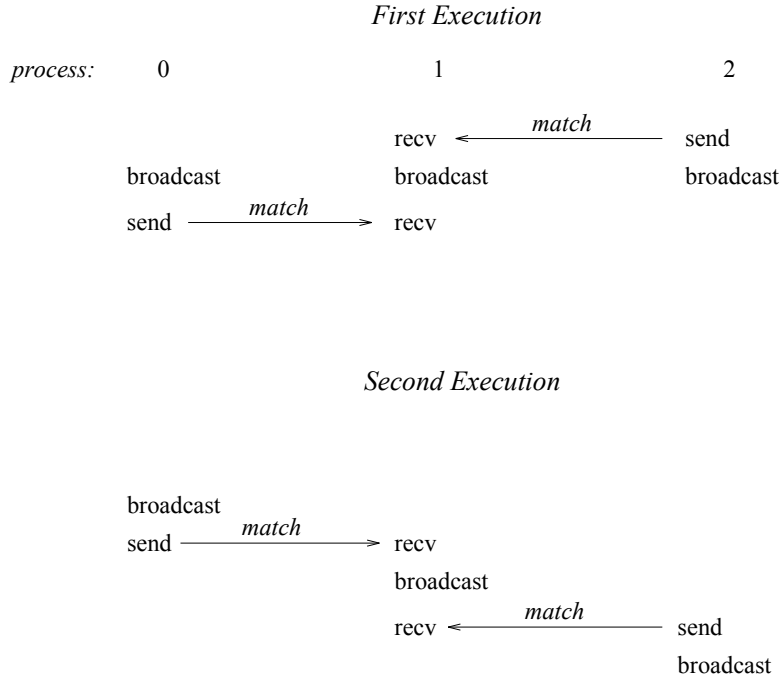

図 4.10: 競合条件が発生すると送信と受信の非決定性の一致が生じる。プログラムを決定性にするためにブロードキャストからの同期に依存することはできない。

任である。これを達成するための 一つの方法は、コミュニケータを生成するたびに、集団通信用の「隠されたコミュニケータ」を生成するということである。 また、例えば隠されたタグまたはコンテキスト・ビットを使用してコミュニケータが 1 対 1 通信用か集団通信用かを指定することにより、同様の効果をより安価に実現できる。

（実装者へのアドバイスの終わり）

Chapter5

グループ、 コンテキスト、 コミュニケータ

5.1 はじめに

この章では、 並列ライブラリの開発を支援する MPI の諸機能を紹介する。並列ライブラリはアルゴリズムの並列化における複雑さを緩和するために必要とされる。並列ライブラリの利用により、 並列化の過程における一貫した正当性を確保でき、 MPI 自体で実現できる以上の高いレベルのポータビリティを実現できる。同様に、 ライブラリを利用することで、 プログラマは（行列演算などの）アルゴリズムを実装するにあたりデータ構造、 データ配置、演算手順をそのたびに定義する作業を繰り返さずに済む。良い並列ライブラリには（システムや問題の規模あるいは浮動小数点の型に応じてデータ配置や戦略の異なる）複数のバリエーションがあるので、これもまたユーザーから隠蔽する必要がある。

本章で説明する機能を使用し、 MPI でライブラリを記述する際の詳しい情報については [26]、 [3] を参照されたい。

5.1.1 ライブラリをサポートするのに必要な機能

頑健な並列ライブラリを作成するため次のような機能が必要となる。

- ライブラリ外の通信によって邪魔される事のない通信を保証する安全な通信空間。
- ライブラリと関与していない（おそらく無関連のコードを実行している）プロセスとの不必要な同期を避ける為の集団操作のスコープ。
- ライブラリ独自のデータ構造およびアルゴリズムに適した用語での通信の記述を可能にするプロセス命名の抽象化。
- 通信プロセス群に対した新たな集団演算などに対応する新しいユーザー定義属性での「修飾」。 この機構によってユーザーやライブラリ作成者はメッセージ通信の記法を拡張することができる。

また、 通信コンテキスト、 通信プロセスのグループを簡潔に表し、プロセス命名の抽象化を扱い、 修飾を格納するため統一された機構ないしはオブジェクトが必要である。

5.1.2 MPI のライブラリ・サポート

堅牢なライブラリをサポートするために MPI が提供する概念には次のものがある。

- 通信の「コンテキスト」
- プロセスの「グループ」
- 「仮想トポロジー」
- 「属性キャッシング」
- 「コミュニケータ」

「コミュニケータ」([16, 24, 27] を参照) は上記の概念すべてを含み、 MPI におけるすべての通信操作の適切なスコープを定める。

コミュニケータは、 単一のプロセスグループ内の操作を行うグループ内コミュニケータと、2 つのプロセスグループ間で 1 対 1 通信を行うグループ間コミュニケータの 2 種類に分けられる。

キャッシング　コミュニケータ（下記参照）は新しい属性をコミュニケータに付与するための「キャッシング」メカニズムを提供する。この新しい属性は MPI に組み込まれた属性と同等に扱われる。この機能は、 上級ユーザーがコミュニケータをさらに修飾する場合や、 MPI 自身がコミュニケータの機能を実装する為に使用する。例えば、 第 6 章で説明している仮想トポロジー関数はこの方法でサポートされるであろう。

グループ　グループは、 順序付されたプロセスの一群であり、各プロセスはそれぞれランクを与えられる。プロセス間通信における低レベルの名前はグループによって定義される (ランクが送受信に使用される)。このように、 グループは 1 対 1 通信におけるプロセス名のスコープを定義する。さらに、 グループは集団操作のスコープも定義する。 MPI においてグループはコミュニケータと別個に扱われる場合があるが、通信操作にはコミュニケータしか使用できない。

グループ内コミュニケータ　MPI のメッセージ通信においてもっとも一般的に使用される手法はグループ内コミュニケータによるものである。グループ内コミュニケータはグループのインスタンス、1 対 1 通信および集団通信のコンテキスト、 仮想トポロジーおよびそのほかの属性を含む。これらの機能は次のように働く。

- MPI の「コンテキスト」は、 メッセージ通信が独立した安全な「空間」で実行されることを可能とするものである。コンテキストは、 メッセージを区別する付加タグに類似している。メッセージはシステムによって区別される。異なるライブラリ（または異なるライブラリ呼び出し）で別々の通信コンテキストを使用することにより、ライブラリ実行の内部通信を外部通信から隔離する。これによって、 「他の」コミュニケータ上に保留された通信があってもライブラリ呼び出しが可能となり、かつ、 ライブラリコードの実行の前後で同期を取る必要がない。また、 保留された 1 対 1 通信は単一のコミュニケータ内の集団通信を妨害しないことが保証されている。

- 「グループ」は、 ひとつのコミュニケータの通信（上記参照）への参加者を定義する。

- 「仮想トポロジー」は、 単一のグループ内のランクとトポロジーからあるいはトポロジーへの特別な写像を定義する。この機能を実現するためコミュニケータ用の特別なコンストラクタを第 6章に定義する。本章で説明しているグループ内コミュニケータはトポロジーを持たない。

- 「属性」は、 ユーザーやライブラリが後の参照のため、 コミュニケータに追加したローカル情報を定義する。

ユーザへのアドバイス

現在の通信ライブラリの多くの実装では、並列プログラムを起動したときに利用可能なすべてのプロセスを含む唯一の定義済み通信空間が用意され、これらのプロセスには連続したランクが割り当てられている。 1 対 1 通信への参加者はそのランクで識別され、（ブロードキャストなどの）集団通信は常にすべてのプロセスを呼び出す。 MPI では、 定義済みコミュニケータ MPI_COMM_WORLD を使用することでこの実装にしたがうことができる。この実装に満足しているユーザーは、 コミュニケータ引数が必要な場所に MPI_COMM_WORLD を設定すればよく、 本章の残り部分を無視可能である。

（ユーザへのアドバイスの終わり）

グループ間コミュニケータ　これまでに説明は「グループ内通信」について述べてきた。 MPI はさらに 2 つの重ならない「グループ間の通信」もサポートする。いくつかの並列モジュールで構成されるアプリケーションを作成する場合、あるモジュールが他のモジュールと当該モジュール内のランクを使用して通信を行えるようにすると便利である。これは特に、クライアントまたはサーバのいずれかが並列動作するクライアント - サーバパラダイムで重宝するものである。グループ間通信によって、すべてのプロセスが初期化時に割り当てられていない動的モデルに MPI を拡張するための機構が提供される。このような場合には、 「空間」をまたぐ通信が必要になる。グループ間通信は、 「グループ間コミュニケータ」と呼ぶオブジェクトによりサポートさ

れる。グループ間コミュニケータは 2 つのグループをこの両グループで共有される通信コンテキストと結合する。グループ間コミュニケータでは、 これらの機能は次のように作用する。

- コンテキストは、 2 つのグループの間でメッセージ通信の独立した安全な「空間」を持つことを可能とする。自グループにおける送信は常に他グループにおける受信であり、またその逆も真である。メッセージはシステムによって区別される。異なるライブラリ（または異なるライブラリ呼び出し）で別々の通信コンテキストを使用することにより、ライブラリ実行の内部通信を外部通信から隔離する。これによって、 「他の」コミュニケータ上に保留された通信があってもライブラリ呼び出しが可能となり、かつ、 ライブラリコードの実行の前後で同期を取る必要がない。グループ間コミュニケータには汎用の集団通信は提供されないので、コンテキストは 1 対 1 通信を隔離するためにのみ使用される。
- 自グループおよび他グループは、 グループ間コミュニケータの受信側と送信先を指定する。
- 仮想トポロジーは、 グループ間コミュニケータに関しては定義されない。
- 属性キャッシュは、 ユーザーまたはライブラリが後の参照のためにコミュニケータに追加したローカル情報を定義する。

MPI は、 グループ間コミュニケータを生成し操作するためのメカニズムを提供する。これは、 グループ内コミュニケータと同様に 1 対 1 通信に使用される。グループ間通信を必要としないユーザーはこの拡張機能を無視してもよい。グループ間コミュニケータを介した集団通信を必要とするユーザーはこの拡張を MPI 上に実装しなければならない。重なり合うグループ間でグループ間通信を必要とするユーザーは、同様に当該機能を MPI 上に実装しなければならない。

5.2 基本概念

この節では、 前節で紹介した概念の形式的な定義を与える。

5.2.1 グループ

「グループ」とは、 順序付けされたプロセス識別子 (以下プロセスと呼ぶ) の集合のことである。プロセスは実装依存オブジェクトである。グループ内の各プロセスは整数の「ランク」が付加されている。ランクは連続で、 0 から始まる。グループは不透明な「グループオブジェクト」なので、 プロセスからプロセスへ直接引き渡す事はできない。コミュニケータにおいてグループは通信「空間」への参加者を記述しまた参加者の順序付けのために使用される (そこで、 参加者には通信「空間」の中で固有の名前が与えられることになる)。

MPI_GROUP_EMPTY は、メンバを持たない特別なグループとして定義されている。定数 MPI_GROUP_NULL は、無効なグループハンドルの値として使用される。

ユーザへのアドバイス　MPI_GROUP_EMPTY は、 空のグループへの有効なハンドルであり、 無効なハンドルである MPI_GROUP_NULL と混同してはならない。前者はグループ操作の引数として使用されるが、後者はグループを解放するときに返されるもので、 引数として使用できない。

（ユーザへのアドバイスの終わり）

実装者へのアドバイス

グループは仮想 - 実プロセス・アドレス変換テーブルで表すことができる。各コミュニケータ・オブジェクト（下記参照）はこのようなテーブルを指すポインタを持つであろう。

MPI の単純な実装では、 テーブル形式にグループを列挙するであろう。しかし多数のプロセスがある場合、 スケーラビリティとメモリ利用度を高めるために高度なデータ構造が意味を持つ。 MPI ではこのような実装が可能である。

（実装者へのアドバイスの終わり）

5.2.2 コンテキスト

「コンテキスト」とは、 通信空間の分割を行う（次に定義する）コミュニケータの特性である。あるコンテキストで送信されるメッセージは他のコンテキストでは受信できない。さらに、 集団操作が可能な場合では集団操作は保留されている 1 対 1 通信操作と無関係に行える。コンテキストは明示的な MPI オブジェクトではなく、コミュニケータの実現の一部である（以下参照）。

実装者へのアドバイス

同じプロセス内の異なるコミュニケータは異なるコンテキストを持つ。コンテキストは、コミュニケータを 1 対 1 通信および MPI で定義する集団通信に対し安全なものとするために必要なシステム管理タグである。

ここで安全という意味は、 1 つのコミュニケータ内の集団通信および 1 対 1 通信が干渉しない、 かつ、 異なるコミュニケータ上の通信が互いに干渉しないことである。

コンテキストの一つの実装方法としては、 送信時メッセージへ付加し、合致するメッセージを受信する補助タグを使用する方法がある。各グループ内コミュニケータは、 2 つのタグの値（ 1 対 1 通信に 1 つ、 集団通信に 1 つ）を持つ。コミュニケータ生成関数は新しいグループ固有のコンテキストに関し、各プロセス間で合意を得るために集団通信を使用する。

同様に、 グループ間通信（必ず 1 対 1 通信である）では、コミュニケータに 2 つのコンテキスト・タグが格納される。一つはグループ A が送信にグループ B が受信に使用し、他の一つはグループ B が送信にグループ A が受信に使用するものである。

コンテキストは明示されたオブジェクトではないので、 他の実装も可能である。

（実装者へのアドバイスの終わり）

5.2.3 グループ内コミュニケータ

グループ内コミュニケータはグループとコンテキストの概念からなる。実装固有の最適化およびアプリケーション・トポロジー（次の第 6 章で定義する）をサポートするためにコミュニケータは追加情報を「キャッシュ」することができる（5.7節を参照）。 MPI の通信操作は 1 対 1 通信および集団操作のスコープと「通信空間」を決定するためにコミュニケータを参照する。

各コミュニケータは有効な参加者からなるグループを持ち、このグループにはローカル・プロセスを含んでいる。メッセージの送信元と送信先のプロセスはそのグループにおけるランクにより指定する。

集団通信の場合、 グループ内コミュニケータは集団操作に加わるプロセスの集合を（必要であれば、 その順序も）指定する。したがって、 コミュニケータは通信の「空間的」スコープを制限し、ランクによって機械独立なプロセス指定方法を提供する。

グループ内コミュニケータは、不透明な「グループ内コミュニケータオブジェクト」によって表され、したがってプロセスからプロセスへ直接には転送できない。

5.2.4 定義済みグループ内コミュニケータ

MPI_INIT を呼び出すと（自分自身を含め）初期化後通信できるすべてのプロセスを含む初期グループ内コミュニケータである MPI_COMM_WORLD が定義される。さらに、 そのプロセスのみが含まれるコミュニケータ MPI_COMM_SELF が提供される。

定義済み定数 MPI_COMM_NULL は無効なコミュニケータ・ハンドルに使用する値である。

MPI の静的なプロセス・モデルの実装では、 初期化後計算に関係するすべてのプロセスが利用可能となる。この場合、 MPI_COMM_WORLD はその計算に使用可能なすべてのプロセスのコミュニケータであり、 すべてのプロセスで同じ値を持つ。プロセスが動的に MPI の実行に参加できる実装においては、他のすべてのプロセスが参加する以前に計算を開始する場合があり得る。この状況において、 MPI_COMM_WORLD は参加プロセスが即座に通信できるプロセスすべてを含むコミュニケータである。したがって、 MPI_COMM_WORLD は同時に異なるプロセスにおいて異なる値を持つことがある。

すべての MPI の実装は MPI_COMM_WORLD コミュニケータを提供することが要求される。このコミュニケータはプロセスの実行中に解放することはできない。このコミュニケータに対応

するグループは定義済み定数とはなっていないが、MPI_COMM_GROUP を使用してアクセスされる場合がある（下記参照）。MPI は MPI_COMM_WORLD のプロセスのランクとその（機械依存）絶対アドレスとの間の対応関係を指定しない。また、MPI はホストプロセスの関数を指定することもない。他の実装依存定義済みコミュニケータも提供されるかもしれない。

5.3 グループ管理

この節では、MPI におけるプロセス・グループの操作について説明する。

これらの操作はローカルなものであり、その実行にはプロセス間通信を必要としない。

5.3.1 グループ参照関数

MPI_GROUP_SIZE(group, size)

入力	group	グループ (ハンドル)
出力	size	グループ中のプロセス数 (整数型)

```
int MPI_Group_size(MPI_Group group, int *size)

MPI_GROUP_SIZE(GROUP, SIZE, IERROR)
    INTEGER GROUP, SIZE, IERROR
```

MPI_GROUP_RANK(group, rank)

入力	group	グループ (ハンドル)
出力	rank	呼び出しプロセスのグループ内のランク、もしくはプロセスがグループのメンバーでない場合 MPI_UNDEFINED (整数型)

```
int MPI_Group_rank(MPI_Group group, int *rank)

MPI_GROUP_RANK(GROUP, RANK, IERROR)
    INTEGER GROUP, RANK, IERROR
```

MPI_GROUP_TRANSLATE_RANKS (group1, n, ranks1, group2, ranks2)

入力	group1	グループ 1 (ハンドル)
入力	n	配列 ranks1 と ranks2 の次元 (整数型)
入力	ranks1	グループ 1 中の 0 以上の有効なランクの配列
入力	group2	グループ 2 (ハンドル)
出力	ranks2	グループ 2 において対応するランクの配列、対応するものがない場合には MPI_UNDEFINED

```
int MPI_Group_translate_ranks (MPI_Group group1, int n, int *ranks1,
              MPI_Group group2, int *ranks2)

MPI_GROUP_TRANSLATE_RANKS(GROUP1, N, RANKS1, GROUP2, RANKS2, IERROR)
    INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERROR
```

この関数は、 2 つの異なるグループに所属する同一のプロセスの順序付けの対応を決定するために使用する。例えば、 MPI_COMM_WORLD のグループ中のプロセスのランクを知っている場合に、その部分集合のグループにおける同じプロセスのランクを知りたいと思うかもしれない。

MPI_GROUP_COMPARE(group1, group2, result)

入力	group1	第一グループ (ハンドル)
入力	group2	第二グループ (ハンドル)
出力	result	結果 (整数型)

```
int MPI_Group_compare(MPI_Group group1,MPI_Group group2, int *result)

MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERROR)
    INTEGER GROUP1, GROUP2, RESULT, IERROR
```

グループのメンバと順序が二つのグループで同一ならば結果は MPI_IDENT となる。これは、例えば group1 および group2 が同じハンドルの場合に起こる。グループメンバは同じであるが順序が異なる場合には MPI_SIMILAR となる。それ以外の場合には MPI_UNEQUAL となる。

5.3.2 グループコンストラクタ

グループコンストラクタは、既存のグループのサブセットやスーパーセットを生成する場合に使用する。これらのコンストラクタは、 既存のグループから新規グループを生成する。これらはローカルな操作であり、異なるグループを異なるプロセスで定義する場合がある。またプロセスはそれ自身を含まないグループを定義する場合もある。コミュニケータを作る関数の引数としてグループを使用する場合にはグループ内の各プロセスが同一のグループ定義を持つ事が必要とされる。 MPI は、 新規にグループを構築するためのメカニズムを提供せず、他のすでに定義されているグループからのみグループを作ることができる。初期コミュニケータ MPI_COMM_WORLD に付随する（関数 MPI_COMM_GROUP によりアクセス可能な）グループが他のすべてのグループ定義のベースとなる。

根拠

後述する MPI_COMM_DUP に類似したグループ複製関数はない。グループがいったん作成されたら、 ハンドルのコピーを作成することでそのグループへの複数の参照を持つことができるのでグループ複製子は必要ない。本節に示すコンストラクタは既存のグループのサブセットおよびスーパーセットが必要な場合に対処するものである。

（根拠の終わり）

実装者へのアドバイス

各グループコンストラクタは新しいグループ・オブジェクトを返すかのように働く。この新規グループが既存グループのコピーの場合には、参照回数を管理することによって新しいオブジェクトを作成しないようにすることができる。

（実装者へのアドバイスの終わり）

MPI_COMM_GROUP(comm, group)

入力	comm	コミュニケータ (ハンドル)
出力	group	comm に対応するグループ (ハンドル)

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
MPI_COMM_GROUP(COMM, GROUP, IERROR)
    INTEGER COMM, GROUP, IERROR
```

MPI_COMM_GROUP は変数 group に comm のグループハンドルを返す。

MPI_GROUP_UNION(group1, group2, newgroup)

入力	group1	第一グループ (ハンドル)
入力	group2	第二グループ (ハンドル)
出力	newgroup	和グループ (ハンドル)

```
int MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)
```

```
MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
```

MPI_GROUP_INTERSECTION(group1, group2, newgroup)

入力	group1	第一グループ (ハンドル)
入力	group2	第二グループ (ハンドル)
出力	newgroup	積グループ (ハンドル)

```
int MPI_Group_intersection(MPI_Group group1, MPI_Group group2,
              MPI_Group *newgroup)
```

```
MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
```

MPI_GROUP_DIFFERENCE(group1, group2, newgroup)

入力	group1	第一グループ (ハンドル)
入力	group2	第二グループ (ハンドル)
出力	newgroup	差グループ (ハンドル)

```
int MPI_Group_difference(MPI_Group group1, MPI_Group group2,
              MPI_Group *newgroup)
```

```
MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
```

集合風演算を次のように定義する。

和 第一グループ (group1) のすべての要素の後に、第一グループにない第二グループ (group2) のすべての要素を続ける。

積 第一グループの要素で第二グループの要素でもあるすべての要素。順序は第一グループのとおりとする。

差 第二グループにない第一グループの要素のすべての要素で、 順序は第一グループのとおりとする。

これらの演算では、 出力グループのプロセス順序は可能ならば主として第一グループの順序で、ついで必要であれば第二グループの順序で決定される。和も積も可換ではないが、 両方とも結合的である。

新しいグループは空の場合があり、 MPI_GROUP_EMPTY に等しくなる。

MPI_GROUP_INCL(group, n, ranks, newgroup)

入力	group	グループ (ハンドル)
入力	n	配列 ranks の要素数 (および newgroup の大きさ) (整数型)
入力	ranks	newgroup に出力される group 中のプロセスのランクの配列 (整数配列)
出力	newgroup	生成された新グループ、 順序は ranks によって決まる。(ハンドル)

```
int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)

MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR
```

関数 MPI_GROUP_INCL は、グループ group 中のランク rank[0],...,rank[n-1] の n 個のプロセスからなるグループ newgroup を生成する。 newgroup の中のランク i のプロセスは group の中のランク ranks[i] を持つプロセスである。 ranks の n 個の要素のそれぞれは group の中の有効なランクでなければならず、 すべての要素は異なっていなければならない。そうでない場合には、 プログラムはエラーになる。 $n = 0$ の場合、 newgroup は MPI_GROUP_EMPTY である。例えば、 この関数はグループの要素の順序を変更する場合に使用できる。 MPI_GROUP_COMPARE も参照のこと。

MPI_GROUP_EXCL(group, n, ranks, newgroup)

入力	group	グループ (ハンドル)
入力	n	配列 ranks の要素数 (整数型)
入力	ranks	newgroup に出力されない group 中のプロセスのランクの配列
出力	newgroup	生成された新グループ、 順序は group 内の順序を保つ。(ハンドル)

```
int MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)

MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR
```

関数 MPI_GROUP_EXCL は、 group からランク ranks[0],...,ranks[n-1] のプロセスを削除することで得られるプロセスのグループ newgroup を作成する。

newgroup の中のプロセスの順序づけは group における順序づけと同じである。

ranks の n 個の要素のそれぞれは group 内で有効なランクでなければならず、すべての要素は異なっていなければならない。そうでない場合には、 プログラムエラーになる。 $n = 0$ であれば、 newgroup は group と同一である。

MPI_GROUP_RANGE_INCL(group, n, ranges, newgroup)

入力	group	グループ (ハンドル)
入力	n	配列 ranges 中の三つ組みの数 (整数型)
入力	ranges	newgroup に含まれるべき group 中のプロセスのランクを示す (最初のランク、 最後のランク、 ストライド) の三つ組みの整数の配列
出力	newgroup	生成される新しいグループ。プロセスの順序は配列 ranges によって決まる。 (ハンドル)

```
int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3],
              MPI_Group *newgroup)

MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR
```

ranges が次のような 3 組だとすると、

$$(first_1, last_1, stride_1), ..., (first_n, last_n, stride_n)$$

newgroup は次のようなランクの group 内のプロセスの列である。

$$first_1, first_1 + stride_1, ..., first_1 + \left\lfloor \frac{last_1 - first_1}{stride_1} \right\rfloor stride_1, ...$$

$$first_n, first_n + stride_n, ..., first_n + \left\lfloor \frac{last_n - first_n}{stride_n} \right\rfloor stride_n.$$

計算で求めたそれぞれのランクは group 内で有効なランクでなければならず、 計算で求めたすべてのランクは異なっていなければならない。そうでない場合、 プログラムエラーになる。 $first_i > last_i$ や、 $stride_i$ は負の場合もあるが、 0 にはならないことに注意すること。

このルーチンの動作は、 配列 ranges をその配列によって指定されたランクを含む単一の配列へ拡張し、 結果の配列およびその他の引数を MPI_GROUP_INCL に渡した動作と同等である。 MPI_GROUP_INCL の動作は、 ranks の中の各ランク i を引数 ranges の中の三組 (i,i,1) で置き換えた MPI_GROUP_RANGE_INCL の動作と同等である。

MPI_GROUP_RANGE_EXCL(group, n, ranges, newgroup)

入力	group	グループ (ハンドル)
入力	n	配列 ranks の要素数 (整数型)
入力	ranges	newgroup から排除される group 中のプロセスのランクを示す (最初のランク, 最後のランク, ストライド) の三つ組みの整数の一次元配列
出力	newgroup	生成される新しいグループ。 group の順序は保たれる。 (ハンドル)

```
int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3],
              MPI_Group *newgroup)

MPI_GROUP_RANGE_EXCL(GROUP, N, RANGES, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR
```

計算で求めたそれぞれのランクは group の中で有効なランクでなければならず、 計算で求めたランクはすべて異なっていなければならない。そうでない場合、 プログラムはエラーになる。

このルーチンの動作は、 配列 ranges を除外されているランクを含む単一の配列へ拡張し、 結果の配列およびその他の引数を MPI_GROUP_EXCL に渡した動作と同等である。 MPI_GROUP_EXCL

の動作は、 ranks の中の各ランク i を引数 ranges の中の三組 (i,i,1) で置き換えた MPI_GROUP_RANGE_EXCL の動作と同等である。

ユーザへのアドバイス 範囲操作はランクを明示的に列挙しないので、 効率よく実装されていればよりスケーラブルである。高品質の実装ではこの利点を得られるので MPI プログラマはできるかぎり範囲操作を使用するよう推奨する。

(ユーザへのアドバイスの終わり)

実装者へのアドバイス

範囲操作は (時間と空間の) よりよいスケーラビリティを得るためできればグループ・メンバを列挙しない形で実装すべきである。

(実装者へのアドバイスの終わり)

5.3.3 グループデストラクタ

MPI_GROUP_FREE(group)

入出力	group		グループ (ハンドル)

```
int MPI_Group_free(MPI_Group *group)

MPI_GROUP_FREE(GROUP, IERROR)
    INTEGER GROUP, IERROR
```

この関数は、 グループ・オブジェクトに解放マークをつける。ハンドル group は、 呼び出しにより MPI_GROUP_NULL に設定される。このグループを使用する実行中の操作はすべて正常に完了する。

実装者へのアドバイス

MPI_COMM_CREATE および MPI_COMM_DUP を呼び出すごとにインクリメントされ、MPI_GROUP_FREE あるいは MPI_COMM_FREE を呼び出すごとにデクリメントされる参照カウントを持つことができる。グループ・オブジェクトは、 参照カウントが 0 になると最終的に解放される。

(実装者へのアドバイスの終わり)

5.4 コミュニケータ管理

この節では、 MPI におけるコミュニケータの操作について説明する。コミュニケータを参照する操作はローカルであり、 実行のためにプロセス間通信を必要としない。コミュニケータを生成する操作は集団操作であり、プロセス間通信を必要とする場合がある。

実装者へのアドバイス

高品質な実装では複数回にわたる（同じグループ、 またはそのサブセットの）コミュニケータの生成に関連するオーバーヘッドを 1 つの集団通信で複数のコンテキストを割り当てることにより低減しなければならない。

（実装者へのアドバイスの終わり）

5.4.1 コミュニケータ・アクセッサ

次のものはすべてローカル通信操作である。

MPI_COMM_SIZE(comm, size)

入力	comm	コミュニケータ (ハンドル)
出力	size	comm のグループ中のプロセス数 (整数型)

```
int MPI_Comm_size(MPI_Comm comm, int *size)

MPI_COMM_SIZE(COMM, SIZE, IERROR)
    INTEGER COMM, SIZE, IERROR
```

根拠

この関数は、 後述するように MPI_COMM_GROUP でコミュニケータのグループをアクセスし、 MPI_GROUP_SIZE を使用してサイズを計算し、 MPI_GROUP_FREE によって一時的なグループを解放する操作と等価である。しかし、 この関数は非常によく使用されるのでショートカットが導入された。

（根拠の終わり）

ユーザへのアドバイス

この関数は、 コミュニケータに関わるプロセスの個数を返す。 MPI_COMM_WORLD の場合、 これは利用可能なプロセスの総数を示している（MPI の本レポートのバージョンでは、 初期化後にプロセスの個数を変更する標準的な手段は用意されていない）。

この関数は、 特定のライブラリやプログラムで利用可能な並列度を決定するために次の MPI_COMM_RANK とともに使用されることが多い。関数 MPI_COMM_RANK は、 0 から size−1 の範囲で呼び出したプロセスのランクを示す。ただし、 size は MPI_COMM_SIZE の返却値である。

（ユーザへのアドバイスの終わり）

MPI_COMM_RANK(comm, rank)

入力	comm	コミュニケータ (ハンドル)
出力	rank	グループ comm における呼び出しプロセスのランク (整数型)

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

```
MPI_COMM_RANK(COMM, RANK, IERROR)
    INTEGER COMM, RANK, IERROR
```

根拠

この関数は、 MPI_COMM_GROUP でコミュニケータのグループをアクセスし（上記参照）、 MPI_GROUP_RANK を使用してランクを計算し、 MPI_GROUP_FREE によって一時的なグループを解放する操作と等価である。しかし、 この関数は非常によく使用されるのでショートカットが導入された。

（根拠の終わり）

ユーザへのアドバイス

この関数は、 特定のコミュニケータ・グループの中の自プロセスのランクを与える。上述のように、 MPI_COMM_SIZE とともに使用すると便利である。

多くのプログラムは、 マスタ・スレーブ・モデルに基づいて作成される。つまり、 一つのプロセス（ランク 0 のプロセスなど）の管理下に、他のプロセス群が計算ノードとしてサービスするのである。このモデルでは、 前出の 2 つの関数がコミュニケータの個々のプロセスの役割を決定するために使用される。

（ユーザへのアドバイスの終わり）

入力 comm1 第一コミュニケータ (ハンドル) 入力 comm2 第二コミュニケータ (ハンドル) 出力 result 結果 (整数型)

MPI_COMM_COMPARE(comm1, comm2, result)

入力	comm1	第一コミュニケータ (ハンドル)
入力	comm2	第二コミュニケータ (ハンドル)
出力	result	結果 (整数型)

```
int MPI_Comm_compare(MPI_Comm comm1,MPI_Comm comm2, int *result)

MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR)
    INTEGER COMM1, COMM2, RESULT, IERROR
```

comm1 と comm2 が同じオブジェクト（同一グループ、 同一コンテキスト）のハンドルである場合、 かつその場合にかぎり MPI_IDENT が返される。対応するグループのメンバおよびランク順序が同一の場合には MPI_CONGRUENT が返される。この場合二つのコミュニケータはコンテキストのみが異なる。両方のコミュニケータのグループのメンバは同一であるが、ランク順が異なる場合に MPI_SIMILAR となる。これら以外の場合は、 MPI_UNEQUAL となる。

5.4.2 コミュニケータコンストラクタ

次に示す関数は、 コミュニケータ comm に付随するグループ内のすべてのプロセスにより呼び出される集団操作関数である。

根拠

MPI では新しいコミュニケータを生成する場合にコミュニケータが必要とされるという点で、鶏が先か卵が先かという議論があり得ることに注意されたい。すべての MPI コミュニケータを生成するための基本コミュニケータは MPI の外であらかじめ MPI_COMM_WORLD として定義されている。

このモデルは、 多くの議論の後に MPI で書いたプログラムの「安全性」を高めるために選択された。

（根拠の終わり）

入力 comm コミュニケータ (ハンドル) 出力 newcomm comm の複製 (ハンドル)

MPI_COMM_DUP(comm, newcomm)

入力	comm	コミュニケータ (ハンドル)
出力	newcomm	comm の複製 (ハンドル)

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
```

```
MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
    INTEGER COMM, NEWCOMM, IERROR
```

MPI_COMM_DUP は、 付随するキーとともに既存のコミュニケータ comm を複製する。付随する各々のキーに対応する複製コールバック関数は新しいコミュニケータに付随する属性値を決定する。複製コールバック関数の利用方法として新規コミュニケータから属性を削除する操作があげられる。返り値 newcomm には同じグループ、 コピーされたキャッシュ情報を持つ新しいコンテキストの新規コミュニケータが与えられる。（5.7.1 節を参照のこと）。

ユーザへのアドバイス

この操作は、 元のコミュニケータと同じ特性を持つ複製された通信空間を使用する並列ライブラリ関数を提供するためにある。この特性には、 属性（下記参照）とトポロジー（第6章参照）が含まれる。この呼び出しは、 コミュニケータ comm に保留された 1 対 1 通信がある場合でも有効である。典型的な呼び出しでは、 並列関数のはじめに MPI_COMM_DUP が呼ばれそれによって複製されたコミュニケータは関数の終わりで MPI_COMM_FREE によって解放されるだろう。他のコミュニケータ管理の方法も可能である。この呼び出しは、グループ内とグループ間コミュニケータの両方に適用される。

（ユーザへのアドバイスの終わり）

実装者へのアドバイス

実際にはグループ情報をコピーする必要はなく、 新しい参照を追加し、 参照カウントをインクリメントすればよい。コピーオンライトの手法をキャッシュ情報に使用することもできる。

（実装者へのアドバイスの終わり）

MPI_COMM_CREATE(comm, group, newcomm)

入力	comm	コミュニケータ (ハンドル)
入力	group	comm のサブセットであるグループ (ハンドル)
出力	newcomm	新しいコミュニケータ (ハンドル)

```
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)

MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
    INTEGER COMM, GROUP, NEWCOMM, IERROR
```

この関数は、 group によって定義された通信グループと新しいコンテキストを持つ新規コミュニケータ newcomm を生成する。キャッシュ情報は comm から newcomm へは継承されない。この関数は、 group に属さないプロセスに MPI_COMM_NULL を返す。この関数の呼び出しに置いて全ての group 引数が同じ値を持たない場合や group が comm のグループのサブセットでない場合にはエラーとなる。この呼び出しは、 新規グループに属す属さないにかかわらず comm のすべてのプロセスによって実行されるべきであることに注意されたい。この呼び出しはグループ内コミュニケータにのみ適用される。

根拠

comm のグループ全体が呼び出しを行うという要求条件は次の考察から生じた。

- 通常の集団通信を使用して MPI_COMM_CREATE を実装可能とする。
- 特に一部重なり合うグループ同士が新規コミュニケータを生成する場合を含め、 さらなる安全性を提供する。
- コンテキスト生成時に、 可能な場合には通信を避ける実装を許容する。

（根拠の終わり）

ユーザへのアドバイス

MPI_COMM_CREATE は、別の通信空間で別の MIMD 計算を行うためにプロセスグループのサブセットを提供するための手段を提供する。 MPI_COMM_CREATE で得られたコミュニケータ newcomm にさらに MPI_COMM_CREATE （または他のコミュニケータコンストラクタ）を適用し、計算を並列化された副計算に再分割することができる。より一般的なサービスは後述の MPI_COMM_SPLIT が提供する。

（ユーザへのアドバイスの終わり）

実装者へのアドバイス

MPI_COMM_DUP または MPI_COMM_CREATE を呼び出すプロセスはすべて同じ group 引数を持つので、 理論的には通信なしでグループ固有のコンテキストを提供することが可能である。しかし、 これらの関数をローカルに実行するにはコンテキスト名を保持するために大きな空間が必要となりまたエラー検査の機会を減らすことにもなる。実装に際してはこれらの矛盾する目標に対して 1 つの集団通信で複数のコンテキストを一括して割り当てるなどの種々の妥協案を打つことができる。

重要：関与するプロセスと同期せずに新しいコミュニケータを生成する場合、 通信システムは受信側プロセスでまだ割り当てられていないコンテキストに到達するメッセージに対処できなければならない。

（実装者へのアドバイスの終わり）

MPI_COMM_SPLIT(comm, color, key, newcomm)

入力	comm	コミュニケータ (ハンドル)
入力	color	サブセット割当の制御 (整数型)
入力	key	ランク割当の制御 (整数型)
出力	newcomm	新しいコミュニケータ (ハンドル)

```
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)

MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)
    INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR
```

この関数は、 comm に付随するグループを color の値に対応してそれぞれ重ならないサブグループに分割する。各サブグループは同一 color のプロセスをすべて含む。各サブグループ内で、 プロセスは引数 key の値の順序でランクが付けられ、同一の key のプロセスは旧グループのランク順に従ってランクが決められる。各サブグループに対応した新しいコミュニケータが生成され、newcomm に返される。プロセスは color の値として MPI_UNDEFINED を供給することができ、その場合、 newcomm には MPI_COMM_NULL が返る。この関数は、 集団呼び出しであるが、 各プロセスは異なる color と key で呼び出しすることが許される。 group のすべてのメンバが color $= 0$ および key としてグループのランクを与え、 group のメンバでないすべてのプロセスは color $=$ MPI_UNDEFINED を与えるとすれば、 MPI_COMM_CREATE(comm, group, newcomm) の呼び出しは、 MPI_COMM_SPLIT(comm, color,key, newcomm) の呼び出しと同等である。関数 MPI_COMM_SPLIT は、 サブグループにおける順序付の変更に対応したグループ分割のより汎用的な方法を提供する。この呼び出しはグループ内コミュニケータにのみ適用される。

color の値は負であってはならない。

ユーザへのアドバイス これはプロセスグループを k 個のサブグループに分割するためのきわめて強力なメカニズムである。ここで k は、 プロセス全体に対して設定したカラーの個数で決まり、ユーザーが暗黙のうちに選択した値である。生成したコミュニケータは互いに重なり合うことはない。このような分割は、マルチグリッド法や、 線形代数などの計算処理の階層構造を定義する際に有用であろう。

MPI_COMM_SPLIT を何回呼び出しても、各プロセスは呼び出し 1 回につき 1 つの color しか持たないため、それぞれの呼出においては生成されたコミュニケータ同士は重なり合わない。このような複数回の MPI_COMM_SPLIT の呼出によって、多重に重なり合う通信構造を生成することができる。分割操作における color および key の独創的な使用方法を推奨する。

ある color に対応するプロセス群のキーがそれぞれユニークである必要はないことに注意が必要である。 MPI_COMM_SPLIT 側で、 key にしたがって昇順でプロセスをソートし、key が一致する場合にも一貫した方法でランクを付与する必要がある。すべてのキーに同じ値が指定された場合、 指定された color のすべてのプロセスはその親グループと同じ相対ランク順序を持つことになる。（一般に、 異なるランクを持つことになる）

与えられた color のすべてのプロセスについてキー値を 0 とすることは、 実質的に新しいコミュニケータにおけるプロセスのランク順序を気にしないということをを意味する。

（ユーザへのアドバイスの終わり）

根拠　color は、 負でない値に制限されている。これは、 MPI_UNDEFINED に割り当てられている値と衝突しないようにするためである。（根拠の終わり）

5.4.3　コミニュケータデストラクタ

MPI_COMM_FREE(comm)

入出力	comm	壊されるコミュニケータ (ハンドル)

```
int MPI_Comm_free(MPI_Comm *comm)

MPI_COMM_FREE(COMM, IERROR)
    INTEGER COMM, IERROR
```

この集団通信操作関数は、 通信オブジェクトに解放のマークを付ける。

ハンドルは MPI_COMM_NULL にセットされる。このコミュニケータを使用中の保留通信操作はすべて正常終了する。なぜなら、 オブジェクトは、 アクティブな参照がほかにない場合にかぎり実際に解放されるからである。この呼び出しはグループ内とグループ間コミュニケータに適用される。すべてのキャッシュ属性に対する削除コールバック関数（5.7節を参照）は任意の順序で呼び出される。

実装者へのアドバイス

MPI_COMM_DUP の呼び出しによってインクリメントされ、 MPI_COMM_FREE を呼び出しによってデクリメントされるような参照カウント・メカニズムを使用することができる。オブジェクトはカウントが 0 になると最終的に解放される。

集団通信操作ではあるが、 この操作は通常ローカルに実装されることが期待される。ただし、 MPI ライブラリのデバッグ・バージョンにおいては同期操作としての実装を選択することができる。（実装者へのアドバイスの終わり）

5.5 例題

5.5.1 一般的な慣例 #1

例 #1a

```
main(int argc, char **argv)
{
  int me, size;
  ...
  MPI_Init ( &argc, &argv );
  MPI_Comm_rank (MPI_COMM_WORLD, &me);
  MPI_Comm_size (MPI_COMM_WORLD, &size);

  (void)printf ("Process %d size %d\n", me, size);
  ...
  MPI_Finalize();
}
```

例 #1a は「何もしない」プログラムで、 プログラム自身を正しく初期化し、「全プロセスを含む」コミュニケータを参照し、 メッセージを出力する。また、 このプログラムは正しく終了している。この例は、 MPI が printf 風の通信自体をサポートしていることを意味していない。

例 #1b （size が偶数であると仮定している）

```
main(int argc, char **argv)
{
   int me, size;
   int SOME_TAG = 0;
   ...
   MPI_Init(&argc, &argv);

   MPI_Comm_rank(MPI_COMM_WORLD, &me);   /* ローカル */
   MPI_Comm_size(MPI_COMM_WORLD, &size); /* ローカル */

   if((me % 2) == 0)
   {
      /* 最上位のプロセス以外は send を行う。 */
```

```
        if((me + 1) < size)
           MPI_Send(..., me + 1, SOME_TAG, MPI_COMM_WORLD);
     }
     else
        MPI_Recv(..., me - 1, SOME_TAG, MPI_COMM_WORLD);


     ...
     MPI_Finalize();
  }
```

例 #1b は、「全プロセスを含む」コミュニケータにおける「偶数」プロセスと「奇数」プロセスとのメッセージ交換を概略的に示している。

5.5.2 一般的な慣例 #2

```
main(int argc, char **argv)
{
  int me, count;
  void *data;
  ...

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &me);

  if(me == 0)
  {
       /* 入力を受け付け、 バッファ "data" を作成する */
      ...
  }

  MPI_Bcast(data, count, MPI_BYTE, 0, MPI_COMM_WORLD);

  ...

  MPI_Finalize();
}
```

この例は集団通信の使用法を説明している。

5.5.3 (おおむね) 一般的な慣例 #3

```
main(int argc, char **argv)
{
  int me, count, count2;
  void *send_buf, *recv_buf, *send_buf2, *recv_buf2;
  MPI_Group MPI_GROUP_WORLD, grprem;
  MPI_Comm commslave;
  static int ranks[] = {0};
  ...
  MPI_Init(&argc, &argv);
  MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
  MPI_Comm_rank(MPI_COMM_WORLD, &me);  /* ローカル */

  MPI_Group_excl(MPI_GROUP_WORLD, 1, ranks, &grprem);  /* ローカル */
  MPI_Comm_create(MPI_COMM_WORLD, grprem, &commslave);

  if(me != 0)
  {
    /* スレーブ側の計算 */
    ...
    MPI_Reduce(send_buf,recv_buff,count, MPI_INT, MPI_SUM, 1, commslave);
    ...
  }
  /* ランク 0 のプロセスは直接この reduce 関数を実行するが、
     他のプロセスは後から到着する */
  MPI_Reduce(send_buf2, recv_buff2, count2,
             MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

  MPI_Comm_free(&commslave);
  MPI_Group_free(&MPI_GROUP_WORLD);
  MPI_Group_free(&grprem);
  MPI_Finalize();
}
```

この例は、 0 番目のプロセス以外のすべてのプロセスからなるグループを (全プロセスからなるグループから) どのように生成するか、その新しいグループに対応したコミュニケータ (commslave) をどのように形成するかを示している。新しいコミュニケータは集団呼び出しで使用され、 すべてのプロセスは MPI_COMM_WORLD コンテキストで別の集団呼び出しを実行する。この例は、それぞれ異なるコンテキストを持つ二つのコミュニケータがどのように通信を保護しているかを示している。つまり、 MPI_COMM_WORLD の通信は commslave の通信から隔離されるし、またその逆もあるということである。

ここで示したように、 どのプロセスにおいても異なるコミュニケータ内のコンテキストは互いに隔離されるので「グループ安全性」はコミュニケータを介して実現される。

5.5.4 例 #4

次の例は、 1 対 1 通信と集団通信との間の「安全性」を示すものである。 MPI は、 一つのコミュニケータが安全な 1 対 1 および集団通信を実行できることを保証している。

```
#define TAG_ARBITRARY 12345
#define SOME_COUNT       50

main(int argc, char **argv)
{
  int me;
  MPI_Request request[2];
  MPI_Status status[2];
  MPI_Group MPI_GROUP_WORLD, subgroup;
  int ranks[] = {2, 4, 6, 8};
  MPI_Comm the_comm;
  ...
  MPI_Init(&argc, &argv);
  MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);

  MPI_Group_incl(MPI_GROUP_WORLD, 4, ranks, &subgroup); /* ローカル */
  MPI_Group_rank(subgroup, &me);     /* ローカル */

  MPI_Comm_create(MPI_COMM_WORLD, subgroup, &the_comm);

  if(me != MPI_UNDEFINED)
```

```
    {
        MPI_Irecv(buff1, count, MPI_DOUBLE, MPI_ANY_SOURCE, TAG_ARBITRARY,
                          the_comm, request);
        MPI_Isend(buff2, count, MPI_DOUBLE, (me+1)%4, TAG_ARBITRARY,
                          the_comm, request+1);
    }

    for(i = 0; i < SOME_COUNT, i++)
      MPI_Reduce(..., the_comm);
    MPI_Waitall(2, request, status);

    MPI_Comm_free(t&he_comm);
    MPI_Group_free(&MPI_GROUP_WORLD);
    MPI_Group_free(&subgroup);
    MPI_Finalize();
  }
```

5.5.5 Library Example #1

主プログラム：

```
  main(int argc, char **argv)
  {
    int done = 0;
    user_lib_t *libh_a, *libh_b;
    void *dataset1, *dataset2;
    ...
    MPI_Init(&argc, &argv);
    ...
    init_user_lib(MPI_COMM_WORLD, &libh_a);
    init_user_lib(MPI_COMM_WORLD, &libh_b);
    ...
    user_start_op(libh_a, dataset1);
    user_start_op(libh_b, dataset2);
    ...
    while(!done)
```

```
    {
       /* 処理 */
       ...
       MPI_Reduce(..., MPI_COMM_WORLD);
       ...
       /* 終了の確認 */
       ...
    }
    user_end_op(libh_a);
    user_end_op(libh_b);

    uninit_user_lib(libh_a);
    uninit_user_lib(libh_b);
    MPI_Finalize();
  }
```

ユーザー・ライブラリの初期化コード：

```
  void init_user_lib(MPI_Comm comm, user_lib_t **handle)
  {
    user_lib_t *save;

    user_lib_initsave(&save); /* ローカル */
    MPI_Comm_dup(comm, &(save -> comm));

    /* その他の初期化 */
    ...

    *handle = save;
  }
```

ユーザー・スタートアップ・コード：

```
  void user_start_op(user_lib_t *handle, void *data)
  {
    MPI_Irecv( ..., handle->comm, &(handle -> irecv_handle) );
    MPI_Isend( ..., handle->comm, &(handle -> isend_handle) );
  }
```

ユーザー通信クリーンアップ・コード：

```
void user_end_op(user_lib_t *handle)
{
  MPI_Status *status;
  MPI_Wait(handle -> isend_handle, status);
  MPI_Wait(handle -> irecv_handle, status);
}
```

ユーザー・オブジェクト・クリーンアップ・コード

```
void uninit_user_lib(user_lib_t *handle)
{
  MPI_Comm_free(&(handle -> comm));
  free(handle);
}
```

5.5.6 ライブラリの例 #2

主プログラム：

```
main(int argc, char **argv)
{
  int ma, mb;
  MPI_Group MPI_GROUP_WORLD, group_a, group_b;
  MPI_Comm comm_a, comm_b;

  static int list_a[] = {0, 1};
#if  defined(EXAMPLE_2B) | defined(EXAMPLE_2C)
  static int list_b[] = {0, 2 ,3};
#else/* EXAMPLE_2A */
  static int list_b[] = {0, 2};
#endif
  int size_list_a = sizeof(list_a)/sizeof(int);
  int size_list_b = sizeof(list_b)/sizeof(int);

  ...
  MPI_Init(&argc, &argv);
```

```
    MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);

    MPI_Group_incl(MPI_GROUP_WORLD, size_list_a, list_a, &group_a);
    MPI_Group_incl(MPI_GROUP_WORLD, size_list_b, list_b, &group_b);

    MPI_Comm_create(MPI_COMM_WORLD, group_a, &comm_a);
    MPI_Comm_create(MPI_COMM_WORLD, group_b, &comm_b);

    if(comm_a != MPI_COMM_NULL)
       MPI_Comm_rank(comm_a, &ma);
    if(comm_a != MPI_COMM_NULL)
       MPI_Comm_rank(comm_b, &mb);

    if(comm_a != MPI_COMM_NULL)
       lib_call(comm_a);

    if(comm_b != MPI_COMM_NULL)
    {
      lib_call(comm_b);
      lib_call(comm_b);
    }

    if(comm_a != MPI_COMM_NULL)
      MPI_Comm_free(&comm_a);
    if(comm_b != MPI_COMM_NULL)
      MPI_Comm_free(&comm_b);
    MPI_Group_free(&group_a);
    MPI_Group_free(&group_b);
    MPI_Group_free(&MPI_GROUP_WORLD);
    MPI_Finalize();
  }
```

ライブラリ：

```
  void lib_call(MPI_Comm comm)
  {
```

```
     int me, done = 0;
     MPI_Comm_rank(comm, &me);
     if(me == 0)
        while(!done)
        {
           MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, comm);
           ...
        }
     else
     {
       /* 処理 */
       MPI_Send(..., 0, ARBITRARY_TAG, comm);
       ....
     }
#ifdef EXAMPLE_2C
     /* include (resp, exclude) for safety (resp, no safety): */
     MPI_Barrier(comm);
#endif
   }
```

上記の例は, list_b にランク 3 を含むかどうか、 そして同期が lib_call に含まれるかどうかにより、 実際には三つの例となっている。この例は、 コンテキストを使用したとしても、 同じコンテキストで複数回の lib_call の呼び出しを行うとこれらの呼び出しは必ずしも互いに安全な状態にないということを示している（いわゆる、 「バック・マスキング」現象が発生し得る）。MPI_Barrier が追加された場合に安全な通信が実現する。この例は、 コンテキストを使用したとしてもライブラリは慎重に作成しなければならないということを示している。ランク 3 が無い場合には、 バック・マスキングを避けて安全性を確保するための同期は必要ない。

「リデュース」や「オールリデュース」などのアルゴリズムでは、 MPI が提供する基本性質によって十分に強力な発信元選択性を持つので、本質的にバック・マスキングの問題は発生しない。同じ根または異なる根の一般的なツリー型ブロードキャスト・アルゴリズムを複数回呼び出す ([28] を参照のこと) 場合も同様である。ここで、同じコンテキストにおけるプロセス間メッセージの二者間での順序保証、およびソース選択性の MPI の二つの性質が前提となる。いずれかの特徴を削除すると、 バック・マスキングが発生としないという保証がなくなる。

非決定論的ブロードキャストや、 ワイルドカード操作を含む他の呼び出しを実行しようとするアルゴリズムは一般には、 「リデュース」、 「オールリデュース」、 「ブロードキャスト」の決定論的実装が持つ、 よい特性を持たない。このようなアルゴリズムにおいて、 正しい

処理を行うためにはコミュニケータのスコープ内で単調増加するタグを利用しなくてはならないこともある。

前述の議論はすべて 1 対 1 通信で実装された「集団呼び出し」を仮定している。 MPI では集団呼び出しは 1 対 1 通信を使用して実装してもいいし、 しなくてもよい。これらのアルゴリズムは、 MPI での集団呼び出しの実装方法とは独立に正確さと安全性の議論のために例示された。 5.7.2節も参照のこと。

5.6 グループ間通信

本節では、グループ間通信の概念を導入し、それをサポートする MPI の仕様について説明を加える。さらに、利用者レベルのサーバを含むプログラムを作成するためのサポートについても述べる。

これまでに述べてきた一対一通信は、プロセスが同じグループのメンバである場合のみを考慮の対象としてきた。本章の前の部分で述べたように、このようなタイプの通信は「グループ内通信」と呼ばれ、そのような通信で使用されるコミュニケータは「グループ内コミュニケータ」と呼ばれる。

モジュール化され、複数分野にまたがるアプリケーションでは、異なるプロセスグループがそれぞれ違ったモジュールを実行したり、異なるモジュールに含まれるプロセスが互いにパイプライン的な、あるいはより一般的なグラフ状の通信を行なったりする。このようなアプリケーションでは、あるプロセスが通信の相手プロセスを特定する最も自然な方法は、相手グループ中での相手プロセスのランクを指定することである。一方、内部に利用者レベルのサーバを含むようなアプリケーションでは、各サーバが複数のクライアントに対してサービスを提供するようなプロセスグループであったり、各クライアントが複数のサーバからのサービスを受けるプロセスグループであるような状況が生ずる。このようなアプリケーションにおいても、相手プロセスを相手グループにおけるランクを用いて指定することがきわめて自然だと考えられる。先にも述べたように、このようなタイプの通信は「グループ間通信」と呼ばれ、ここで使われるコミュニケータは「グループ間コミュニケータ」と呼ばれる。

訳注

ここで、「複数分野にまたがるアプリケーション」とは、例えば海洋シミュレーションと気流シミュレーションとを連成させて気象シミュレーションを行なうような場合を指す。

訳注終り

グループ間通信は、異なるグループに属するプロセス間での一対一通信である。グループ間通信を開始するプロセスを含むグループは「自グループ」と呼ばれる。送信操作における送信プロセス、あるいは受信操作における受信プロセスがこれに相当する。一方、通信の相手プロセスを含むグループは「他グループ」と呼ばれる。送信操作における受信プロセス、受信操作におけ

る送信プロセスがこれに相当する。グループ内コミュニケーションの場合と同様、相手プロセスは（コミュニケータ、ランク）のペアを用いて指定される。しかし、グループ内通信と異なり、ランクはリモートグループに対して相対的に定義される。

グループ間コミュニケータに関するすべてのコンストラクタの動作はブロッキングである。また、デッドロックを避けるために、自グループと他グループは重なりを持ってはならない。

以下に、グループ間通信とグループ間コミュニケータの特性をまとめる。

- 一対一通信のシンタックスは、グループ間、グループ内双方の通信に関して同様である。同じコミュニケータを送信と受信の双方に用いることができる。
- 相手プロセスは、送信と受信の双方に関して、他グループでのランクによって指定される。
- グループ間コミュニケータを用いた通信は、異なるコミュニケータを用いた他のいかなる通信とも干渉しないことが保証される。
- グループ間コミュニケータは集団通信には用いることができない。
- 一つのコミュニケータはグループ内、グループ間のいずれかの通信にのみ使用でき、両方に用いることはできない。

あるコミュニケータがグループ内、グループ間のいずれであるかを知るためには、関数 MPI_COMM_TEST_INTER を用いることができる。グループ間コミュニケータは、コミュニケータの情報を参照するためのいくつかの関数に対して、引数として与えることができる。一方、グループ内コミュニケータのためのコンストラクタ関数のうちいくつかは、グループ間コミュニケータを入力引数としてとることができないものがある。（たとえば MPI_COMM_CREATE）

実装者へのアドバイス　一対一通信を実現するために、コミュニケータは各プロセスにおいて以下のような項目の組によって表現することができる。

グループ (**group**)

送信コンテクスト

受信コンテクスト

ソース (**source**)

グループ間コミュニケータに対しては、 **group** は他グループを表わし、 **source** は自グループでのランクを表わす。グループ内コミュニケータに対しては、 **group** はコミュニケータグループ（= 他グループ = 自グループ）を、 **source** はそのグループにおけるプロセスのランクを表わす。送信コンテクストと受信コンテクストは同一である。 **group** は、ランクからプロセスの絶対位置への変換テーブルとして表現される。

訳注

ランクはグループに付随する属性であり、コミュニケータを表すための必須の要素ではない。しかし、コミュニケータを実装する上でランク情報が有用であるので、記載されていると思われる。

訳注終わり

グループ間コミュニケータを論ずるには、自グループと他グループそれぞれにおける 2 つのプロセスを考える必要がある。いま、グループ $\mathcal{P}$ におけるプロセス **P** がグループ間コミュニケータ $\mathbf{C}_{\mathcal{P}}$ を持ち、グループ $\mathcal{Q}$ におけるプロセス **Q** がグループ間コミュニケータ $\mathbf{C}_{\mathcal{Q}}$ を持つとする。このとき、

- $\mathbf{C}_{\mathcal{P}}$**.group** はグループ $\mathcal{Q}$ を表わし、 $\mathbf{C}_{\mathcal{Q}}$**.group** はグループ $\mathcal{P}$ を表わす。
- $\mathbf{C}_{\mathcal{P}}$**.send_context** $=$ $\mathrm{C}_{\mathcal{Q}}$**.receive_context** であり、そのコンテキストはグループ $\mathcal{Q}$ に関してユニークである。同様に、 $\mathbf{C}_{\mathcal{P}}$**.receive_context** $=$ $\mathbf{C}_{\mathcal{Q}}$**.send_context** であり、そのコンテキストは $\mathcal{P}$ に関してユニークである。
- $\mathbf{C}_{\mathcal{P}}$**.source** はグループ $\mathcal{P}$ におけるプロセス **P** のランクであり、 $\mathbf{C}_{\mathcal{Q}}$**.source** はグループ $\mathcal{Q}$ におけるプロセス **Q** のランクである。

プロセス **P** がグループ間コミュニケータを用いてプロセス **Q** にメッセージを送る場合を考える。このとき、 **P** は **group** の変換テーブルを用いて **Q** の絶対位置を見出す。 **source** と **send_context** はメッセージに付加される。

プロセス **Q** がソースを明示的に指定してグループ間コミュニケータを用いて受信をポストする場合を考える。このとき、 **Q** は **receive_context** をメッセージのコンテクストと照合し、 **source** をメッセージに付加されたソース情報と照合する。

同様なアルゴリズムは、グループ内コミュニケータに関しても適用できる。

グループ間コミュニケータのアクセッサとコンストラクタをサポートするためには、上に述べたモデルにさらに構造体を追加することが必要である。この構造体は、自通信グループと付加的なコンテキストに関する情報を格納するために用いる。（実装者へのアドバイスの終わり）

5.6.1 グループ間コミュニケータのアクセッサ

MPI_COMM_TEST_INTER(comm, flag)

入力	comm	コミュニケータ（ハンドル）
出力	flag	（論理型）

```
int MPI_Comm_test_inter(MPI_Comm comm, int *flag)

MPI_COMM_TEST_INTER(COMM, FLAG, IERROR)
    INTEGER COMM, IERROR
    LOGICAL FLAG
```

このローカル関数によって、呼び出し元プロセスは、あるコミュニケータがグループ間コミュニケータかグループ内コミュニケータかを知ることができる。本関数は、前者であれば true を、後者であれば false を返す。

これまでに挙げられたグループ内コミュニケータに関するアクセッサに対し、入力引数としてグループ間コミュニケータを与えた場合、それらの関数は次の表に示すように振る舞う。

MPI_COMM_* 関数の振舞い (グループ間コミュニケータモード)	
MPI_COMM_SIZE	自グループのサイズを返す。
MPI_COMM_GROUP	自グループを返す。
MPI_COMM_RANK	自グループ中でのランクを返す。

さらに、比較関数 MPI_COMM_COMPARE はグループ間コミュニケータに関しても用いることができる。比較が成立するためには双方のコミュニケータはプロセス内かプロセス間かのいずれか一方である必要があり、そうでない場合には MPI_UNEQUAL が返される。 MPI_CONGRUENT と MPI_SIMILAR の結果を得るためには、対応する自グループと他グループがそれぞれ正しく比較できなくてはならない。特に、自グループ、他グループの一方でも要素の並び順が異なる場合には、 MPI_SIMILAR が返されることに注意。

次に挙げる関数は、つねにグループ間コミュニケータの他グループに関するアクセス手段を提供する。

これらはすべてローカル関数である。

MPI_COMM_REMOTE_SIZE(comm, size)

入力	comm	グループ間コミュニケータ（ハンドル）
出力	size	comm 中の他グループに属するプロセスの個数（整数型）

```
int MPI_Comm_remote_size(MPI_Comm comm, int *size)

MPI_COMM_REMOTE_SIZE(COMM, SIZE, IERROR)
    INTEGER COMM, SIZE, IERROR
```

MPI_COMM_REMOTE_GROUP(comm, group)

入力	comm	グループ間コミュニケータ（ハンドル）
出力	group	comm に対応する他グループ（ハンドル）

```
int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)

MPI_COMM_REMOTE_GROUP(COMM, GROUP, IERROR)
    INTEGER COMM, GROUP, IERROR
```

根拠 グループ間コミュニケータを構成する自グループと他グループに対称な方法でアクセスできることは重要である。そのために、本関数および MPI_COMM_REMOTE_SIZE が提供されている。（根拠の終わり）

5.6.2 グループ間コミュニケータの操作

本節では、グループ間コミュニケータに関する 4 つのブロッキング操作を導入する。 MPI_INTERCOMM_CREATE は 2 つのグループ内コミュニケータを結合してグループ間コミュニケータとするのに用いられる。 MPI_INTERCOMM_MERGE はグループ間コミュニケータを構成する自グループと他グループを結合して 1 つのグループ内コミュニケータを生成する。以前にも述べた MPI_COMM_DUP と MPI_COMM_FREE はそれぞれ、グループ間コミュニケータを複製、解放する。

グループ間コミュニケータの構成要素となる自グループと他グループとの間の重複は認められない。重複がある場合、そのプログラムは誤りであり、デッドロックに陥る可能性が高い。（プロセスがマルチスレッド構成であり、 MPI の関数呼び出しがプロセス全体ではなく 1 つのスレッドしかブロックしないのであれば、プロセスが重複してグループに属することも可能である。その場合には、利用者の責任において、そのプロセスの 2 つの役割がそれぞれ別のスレッドによって実行されることを保証する必要がある。）

MPI_INTERCOMM_CREATE は、次に述べる状況下で、2 つの既存のグループ内コミュニケータからグループ間コミュニケータを生成するのに用いられる。まず、それぞれのグループ中で、選ばれた少なくとも 1 つのメンバ（グループリーダ）が他方のグループのグループリーダと通信できなくてはならない。すなわち、2 つのグループリーダが属する「仲介」コミュニケータが存在し、かつ各リーダは他方のリーダの仲介コミュニケータ内でのランクを知っている必要がある。（2 つのリーダは同一プロセスである可能性もある。）さらに、各グループのメンバは、それぞれのリーダのランクを知っていなくてはならない。

訳注

自グループと他グループとの重複は認められず、2 つのリーダが同一のプロセスである可能性はないことから、上の記述は誤りだと思われる。

訳注終わり

2 つのグループ内コミュニケータからのグループ間コミュニケータの生成時には、自グループと他グループでの独立した集団操作と、2 つのリーダの間での一対一通信が必要となる。

標準的な MPI の実装（初期化時に静的にプロセスが割り当てられる）では、MPI_COMM_WORLD コミュニケータ（より好ましくは、専用にそこから作られた複製）が仲介コミュニケータとなりうる。MPI 実行中にプロセスが新しい子プロセスを生成できるような動的な実装においては、もとの通信プロセス空間と、親子双方のプロセスを含む新しい通信プロセス空間との橋渡しとして、親プロセスを利用することが可能であろう。

chapter 6にて述べたアプリケーショントポロジーに関する関数は、グループ間コミュニケータに対しては適用することはできない。このような機能が必要な利用者は、MPI_INTERCOMM_MERGE を用いてグループ内コミュニケータを生成し、それに対してグラフあるいはカルテシアントポロジーを当てはめて適当なトポロジー属性を持たせるのが良い。さもなければ、一般性を失わないような自分自身のアプリケーショントポロジー機構を考案することも可能である。

MPI_INTERCOMM_CREATE(local_comm, local_leader, peer_comm, remote_leader, tag, newintercomm)

入力	local_comm	自グループ内コミュニケータ。（ハンドル）
入力	local_leader	local_comm 中の自グループリーダのランク。（整数型）
入力	peer_comm	「仲介」コミュニケータ。local_leader プロセスのみで意味を持つ。（ハンドル）
入力	remote_leader	peer_comm 中での他グループリーダのランク。local_leader プロセスのみで意味を持つ。（整数型）
入力	tag	「安全のための」タグ（整数型）
出力	newintercomm	新しく生成されたグループ間コミュニケータ。（ハンドル）

```
int MPI_Intercomm_create (MPI_Comm local_comm, int local_leader,
              MPI_Comm peer_comm, int remote_leader, int tag,
              MPI_Comm  *newintercomm)

MPI_INTERCOMM_CREATE(LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,
              NEWINTERCOMM, IERROR)
    INTEGER LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,
    NEWINTERCOMM, IERROR
```

本関数はグループ間コミュニケータを生成する。動作は、自グループと他グループの和集合について集団的である。プロセスは各グループ内で同一の local_comm と local_leader を指定する必要がある。 remote_leader、 local_leader、 tag に関してワイルドカードを用いることはできない。本関数はリーダ間の一対一通信に関して、コミュニケータ peer_comm とタグ tag を用いる。したがって、 peer_comm 上でこの通信と干渉する可能性のある未完了の通信が存在しないように注意を払う必要がある。

ユーザへのアドバイス 仲介コミュニケータの使用にともなう問題を避けるために、例えば MPI_COMM_WORLD の複製のような専用のコミュニケータを利用することを推奨する。（ユーザへのアドバイスの終わり）

図 5.1: 3 つのグループによる「パイプライン」通信

MPI_INTERCOMM_MERGE(intercomm, high, newintracomm)

入力	intercomm	グループ間コミュニケータ（ハンドル）
入力	high	（論理型）
出力	newintracomm	新しく生成されたグループ内コミュニケータ（ハンドル）

```
int MPI_Intercomm_merge(MPI_Comm intercomm, int high,
              MPI_Comm *newintracomm)

MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, INTRACOMM, IERROR)
    INTEGER INTERCOMM, INTRACOMM, IERROR
    LOGICAL HIGH
```

本関数は、 intercomm を構成する 2 つのグループの和集合から 1 つのグループ内コミュニケータを生成する。すべてのプロセスは、各グループ内で同一の high の値を設定しなくてはならない。一方のグループに属するプロセスが high=false を設定し、他方のグループに属するプロセスが high=true を設定したならば、和集合内でのランクは「low」のグループが「high」のグループより前になるように決定される。すべてのプロセスが同じ high の値を設定したならば、和集合内でのランクは任意となる。この関数は 2 つのグループの和集合に関して集団的であり、かつブロッキングである。

> 実装者へのアドバイス　MPI_INTERCOMM_MERGE, MPI_COMM_FREE, MPI_COMM_DUP の実装は MPI_INTERCOMM_CREATE の実装と似ている。ただし前 3 者では、グループリーダ間の通信において、仲介コミュニケータのコンテクストではなく入力引数に与えられたグループ間コミュニケータのコンテクストを用いることに留意。（実装者へのアドバイスの終わり）

5.6.3 グループ間コミュニケータの使用例

例 1: 3 つのグループによる「パイプライン」通信

グループ 0 と 1 とが通信を行ない、グループ 1 と 2 とが通信を行なう。したがって、グループ 1 は 1 つのグループ間コミュニケータを、グループ 2 は 2 つのグループ間コミュニケータを、グループ 3 は 1 つのグループ間コミュニケータを必要とする。

```
main(int argc, char **argv)
{
  MPI_Comm   myComm;       /* ローカルサブグループに対する */
                           /* グループ内コミュニケータ     */
  MPI_Comm   myFirstComm;  /* グループ間コミュニケータ     */
  MPI_Comm   mySecondComm; /* 第 2 のグループ間コミュニケータ（グループ 1 のみ）*/
  int membershipKey;
  int rank;

  MPI_Init(&argc, &argv);
  Mpi_Comm_rank(MPI_COMM_WORLD, &rank);

  /* 利用者コードは [0, 1, 2] の範囲で membershipKey を生成 */
  /* しなくてはならない。                                   */
  membershipKey = rank % 3;

  /* ローカルサブグループ用にグループ内コミュニケータを作成する。 */
  MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);

  /* グループ間コミュニケータを作成する。タグはハードコードされる。 */
  if (membershipKey == 0)
  {    /* グループ 0 はグループ 1 と通信する。 */
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
         1, &myFirstComm);
  }
  else if (membershipKey == 1)
  {    /* グループ 1 はグループ 0 および 2 と通信する。 */
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0,
                          1, &myFirstComm);
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2,
                          12, &mySecondComm);
  }
```

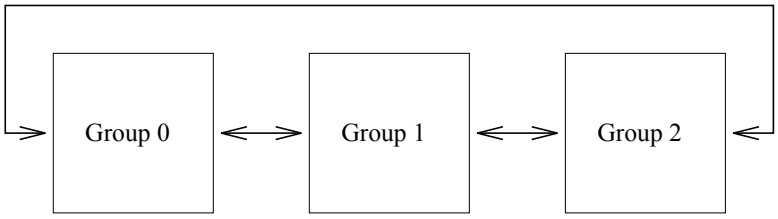

図 5.2: 3 つのグループによる「リング」通信

```
  else if (membershipKey == 2)
  {    /* グループ 2 はグループ 1 と通信する。 */
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                          12, &myFirstComm);
  }

  /* ここで必要な仕事をする... */

  switch(membershipKey)  /* コミュニケータを適切に解放する。 */
  {
  case 1:
    MPI_Comm_free(&mySecondComm);
  case 0:
  case 2:
    MPI_Comm_free(&myFirstComm);
    break;
  }

  MPI_Finalize();
}
```

例 2: 3 つのグループによる「リング」通信

グループ 0 と 1、 1 と 2、 2 と 0 がそれぞれ通信を行なう。したがって、それぞれが 2 つのグループ間コミュニケータを必要とする。

```
main(int argc, char **argv)
{
  MPI_Comm    myComm;       /* ローカルサブグループに対する */
                            /* グループ内コミュニケータ      */
```

```
MPI_Comm   myFirstComm; /* グループ間コミュニケータ */
MPI_Comm   mySecondComm;
MPI_Status status;
int membershipKey;
int rank;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
...

/* 利用者コードは [0, 1, 2] の範囲で membershipKey を生成 */
/* しなくてはならない。                                    */
membershipKey = rank % 3;

/* ローカルサブグループ用にグループ内コミュニケータを作成する。 */
MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);

/* グループ間コミュニケータを作成する。タグはハードコードされる。 */
if (membershipKey == 0)
{    /* グループ 0 はグループ 1 および 2 と通信する。 */
  MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                        1, &myFirstComm);
  MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2,
                        12, &mySecondComm);
}
else if (membershipKey == 1)
{    /* グループ 1 はグループ 0 および 2 と通信する。 */
  MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0,
                        1, &myFirstComm);
  MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2,
                        12, &mySecondComm);
}
else if (membershipKey == 2)
{    /* グループ 2 はグループ 0 および 1 と通信する。 */
  MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0,
```

```
                        2, &myFirstComm);
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                        12, &mySecondComm);
  }

  /* ここで必要な仕事をする ... */

  /* 終了前にコミュニケータを解放する... */
  MPI_Comm_free(&myFirstComm);
  MPI_Comm_free(&mySecondComm);
  MPI_Comm_free(&myComm);
  MPI_Finalize();
}
```

例 3: グループ間通信のためのネームサービスの作成

以下のプログラムは、グループ間コミュニケータ作成用のネームサービスを利用者が作成する際の手順を示す一例である。グループ間コミュニケータは、2 つのグループによって選ばれた名前（タグ）を、サーバコミュニケータを用いて照合することにより作られる。

すべての MPI プロセスが MPI_INIT をコールした後で、各プロセスは下の例に示す関数 Init_server() をコールする。そこから返される new_world が NULL であったならば、そのプロセスは要求待ちループ形式のサーバー関数 Do_server() を実装しなくてはならない。その他の全プロセスは、 new_world を実質的に新しい「グローバル」コミュニケータとみなして必要な仕事を行なう。サーバーが不要になった場合には、ある特定のプロセスが Undo_server() をコールし、サーバーを終了させる。

このアプローチは、以下のような特徴を持つ。

- 複数のネームサーバーのサポート
- ネームサーバーのスコープを指定されたプロセスに限定する機能
- そのようなサーバーを要求通りに起動、終了させる機能

```
#define INIT_SERVER_TAG_1  666
#define UNDO_SERVER_TAG_1  777

static int server_key_val;
```

```
/* server_comm のアトリビュート管理のためのコピーコールバック */
void handle_copy_fn(MPI_Comm *oldcomm, int *keyval, void *extra_state,
            void *attribute_val_in, void **attribute_val_out, int *flag)
{
  /* ハンドルをコピーする。 */
  *attribute_val_out = attribute_val_in;
  *flag = 1; /* コピーが起きることを示す。 */
}

int Init_server(peer_comm, rank_of_server, server_comm, new_world)
MPI_Comm peer_comm;
int rank_of_server;
MPI_Comm *server_comm;
MPI_Comm *new_world;  /* 実質上の新しいワールド。サーバ以外を含む。 */
{
  MPI_Comm temp_comm, lone_comm;
  MPI_Group peer_group, temp_group;
  int rank_in_peer_comm, size, color, key = 0;
  int peer_leader, peer_leader_rank_in_temp_comm;

  MPI_Comm_rank(peer_comm, &rank_in_peer_comm);
  MPI_Comm_size(peer_comm, &size);

  if ((size < 2) || (0 > rank_of_server) || (rank_of_server >= size))
    return (MPI_ERR_OTHER);

  /* peer_comm をサーバープロセスとその他に分割することにより */
  /* 2 つのコミュニケータを生成する。                          */

  peer_leader = (rank_of_server + 1) % size;  /* 選択は任意。 */

  if ((color = (rank_in_peer_comm == rank_of_server)))
  {
    MPI_Comm_split(peer_comm, color, key, &lone_comm);
```

```
  MPI_Intercomm_create(lone_comm, 0, peer_comm, peer_leader,
      INIT_SERVER_TAG_1, server_comm);

  MPI_Comm_free(&lone_comm);
  *new_world = MPI_COMM_NULL;
}
else
{
  MPI_Comm_Split(peer_comm, color, key, &temp_comm);

  MPI_Comm_group(peer_comm, &peer_group);
  MPI_Comm_group(temp_comm, &temp_group);
  MPI_Group_translate_ranks(peer_group, 1, &peer_leader,
      temp_group, &peer_leader_rank_in_temp_comm);

  MPI_Intercomm_create(temp_comm, peer_leader_rank_in_temp_comm,
      peer_comm, rank_of_server,
      INIT_SERVER_TAG_1, server_comm);

  /* new_world のアトリビュートを server_comm に付加する。 */

  /* マルチスレッドのためのクリティカルセクション */
  if(server_keyval == MPI_KEYVAL_INVALID)
  {
    /* server_keyval に対してプロセスローカルな名前を取得する。 */
    MPI_keyval_create(handle_copy_fn, NULL,
    &server_keyval, NULL);
  }

  *new_world = temp_comm;

  /* グループ内コミュニケータのハンドルをグループ間コミュニケータに */
  /* キャッシングする。                                              */
  MPI_Attr_put(server_comm, server_keyval, (void *)(*new_world));
}
```

```
  return (MPI_SUCCESS);
}
```

サーバプロセスは、次のコードを実行する必要がある。

```
int Do_server(server_comm)
MPI_Comm server_comm;
{
  void init_queue();
  int en_queue(), de_queue();  /* 正数の 3 つ組を後のマッチング用に */
                               /* 保持する。 (関数の内容は省略)     */

  MPI_Comm comm;
  MPI_Status status;
  int client_tag, client_source;
  int client_rank_in_new_world, pairs_rank_in_new_world;
  int buffer[10], count = 1;

  void *queue;
  init_queue(&queue);

  for (;;)
  {
    MPI_Recv(buffer, count, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
        server_comm, &status); /* どのクライアントからの要求も受け付ける。 */

    /* クライアントを決定する。 */
    client_tag = status.MPI_TAG;
    client_source = status.MPI_SOURCE;
    client_rank_in_new_world = buffer[0];

    if (client_tag == UNDO_SERVER_TAG_1)
    /* サーバを終了させるクライアントの場合 */
    {
      while (de_queue(queue, MPI_ANY_TAG, &pairs_rank_in_new_world,
```

```
        &pairs_rank_in_server))
      ;

      MPI_Intercomm_free(&server_comm);
      break;
    }

    if (de_queue(queue, client_tag, &pairs_rank_in_new_world,
        &pairs_rank_in_server))
    {
    /* 同じタグでプロセスのペアが対応づけられたので、 */
    /* 各々のプロセスに相手を知らせる。                */
    buffer[0] = pairs_rank_in_new_world;
    MPI_Send(buffer, 1, MPI_INT, client_src, client_tag,
        server_comm);

    buffer[0] = client_rank_in_new_world;
    MPI_Send(buffer, 1, MPI_INT, pairs_rank_in_server, client_tag,
    server_comm);
  }
  else
    en_queue(queue, client_tag, client_source,
    client_rank_in_new_world);

  }
}
```

サーバーが不必要になった場合には、ある特定のプロセスがそれを終了させる。 Undo_server() 関数をコールすることでそのようなサーバの終了を行なうことができる。

```
int Undo_server(server_comm)  /* サーバーを終了させるクライアントの例
*/
MPI_Comm *server_comm;
{
  int buffer = 0;
  MPI_Send(&buffer, 1, MPI_INT, 0, UNDO_SERVER_TAG_1, *server_comm);
```

```
  MPI_Intercomm_free(server_comm);
}
```

次に挙げる関数は、グループ間コミュニケータのためのネームサービスを行なう。本関数はブロッキング動作である。 MPI_Intercomm_create と同様なセマンティック上の制約に従うが、使い方はより簡便である。ネームサービスを行なうために定義された機能のみを用いている。

```
int Intercomm_name_create(local_comm, server_comm, tag, comm)
MPI_Comm local_comm, server_comm;
int tag;
MPI_Comm *comm;
{
  int error;
  int found;   /* new_world のためのアトリビュート取得管理用 */
  /* server_comm 内キャッシングされているコミュニケータ */
  void *val;

  MPI_Comm new_world;

  int buffer[10], rank;
  int local_leader = 0;

  MPI_Attr_get(server_comm, server_keyval, &val, &found);
  new_world = (MPI_Comm)val; /* キャッシングされたハンドルを取得。 */

  MPI_Comm_rank(server_comm, &rank);  /* 自グループ内のランク */

  if (rank == local_leader)
  {
    buffer[0] = rank;
    MPI_Send(&buffer, 1, MPI_INT, 0, tag, server_comm);
    MPI_Recv(&buffer, 1, MPI_INT, 0, tag, server_comm);
  }

  error = MPI_Intercomm_create(local_comm, local_leader, new_world,
  buffer[0], tag, comm);
```

```
  return(error);
}
```

5.7 キャッシング

MPI はコミュニケータに対して「属性」 と呼ばれる任意の情報をアプリケーションに付加するため、 “キャッシング” というファシリティを用意している。より正確に言えば、キャッシングファシリティを用いれば、次のような機能をポータブルなライブラリとして実装できるようになる。

- MPI のコミュニケータ内、あるいはコミュニケータ間でキャッシングファシリティと協力して呼び出し間の情報を伝え、
- その情報を高速に回復し、
- 古くなった情報が再利用されないことを保証する。これは例えコミュニケータが開放され、その後、 MPI によって同一のコミュニケータへのハンドルが再利用された場合にも保証される。

集団的通信やアプリケーショントポロジーを扱うような MPI の組み込みルーチンでキャッシングの能力が必要とされる。これらの機能へのインターフェイスを MPI の標準の一部として定義することには重要である。それは集団的通信とアプリケーショントポロジーを扱うようなルーチンがポータブルになるからである。そしてまた、これにより MPI をユーザが拡張した時にも標準の MPI の呼び出し手順を用いることができる。

> ユーザへのアドバイス　コミュニケータ MPI_COMM_SELF は、この属性キャッシング機構を通してプロセスにローカルな属性を設定するのに適している。 (ユーザへのアドバイスの終わり)

5.7.1 機能説明

属性はコミュニケータに附属する。属性はそれらが附属されたプロセスとそれぞれのコミュニケータにローカルである。属性は、 MPI_COMM_DUP によって複製された場合を除いてコミュニケータからコミュニケータへと伝播することはない (そして、アプリケーションはコピー時に属性をコピーする特定のコールバック関数に許可を与える必要がある。)。

> 実装者へのアドバイス　属性は C 言語のポインタのサイズと同じ大きさかあるいはそれより大きなサイズのスカラ値である。属性は、 MPI のハンドルを常に保持することができる。 (実装者へのアドバイスの終わり)

ここで定義されるキャッシングインターフェイスは、 MPI に対してストアされる属性が、コミュニケータに対して不透明であることを示している。アクセッサ関数は次のものを含んでいる。

- (属性を識別するための) キー値を得る; コミュニケータが破壊あるいはコピーされた時、MPI がそのアプリケーションを知らせるので、それに従ってユーザは呼び出すコールバック関数を指定する。
- 属性の値を保持し回復する。

実装者へのアドバイス　キャッシングとコールバック関数は、アプリケーションの明示的なリクエストに対して、常に同期して呼び出される。これによりユーザ領域とシステム領域間で繰り返される交錯の問題を避けることができる。 (この同期呼び出し規則は、 MPI の一般的な特性である。)

キー値の選択は MPI に一任されている。これは、 MPI が属性の組の最適化を可能にするためである。また、これにより同一のコミュニケータを使用する独立したモジュールでのキャッシュ情報の衝突を MPI 側で避けることができる。

コールバックのファシリティからだけなるような非常に小さなインターフェイスを用いれば、キャッシングファシリティ全般にわたってポータブルなコードを書くことができるだろう。しかしながら、インターフェイスに最小のものを採用した場合には、いくつかの表検索手法を任意のコミュニケータから利用するしくみを提供する必要がある。一方ここではより複雑なインターフェイスを定義している。これは属性への高速なアクセスが可能を可能にするためである。ここでは、 (属性テーブルをみつけるため) コミュニケータのポインタを用いて賢くキー値を選択することによって (独立な属性を検索することで) 高速なアクセスを達成する。最小のインターフェイスの持つ固有の効果的な点を考慮しても、ここで実現されるより複雑なインターフェイスの方が優れているように見える。

(実装者へのアドバイスの終わり)

MPI はキャッシングに関して次のようなサービスを提供する。これらは全てプロセスにローカルである。

MPI_KEYVAL_CREATE(copy_fn, delete_fn, keyval, extra_state)

入力	copy_fn	keyval をコピーするためのコールバック関数
入力	delete_fn	keyval を削除するためのコールバック関数
出力	keyval	今後のアクセスのためのキー値 (整数値)
入力	extra_state	コールバック関数のためのその他の状態

```
int MPI_Keyval_create(MPI_Copy_function *copy_fn, MPI_Delete_function
              *delete_fn, int *keyval, void* extra_state)

MPI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)
    EXTERNAL COPY_FN, DELETE_FN, INTEGER KEYVAL, EXTRA_STATE, IERROR
```

は新規の属性を生成する。キー値はプロセスにローカルで、ユーザからは見えなくなっており、明示的に整数値としてストアされる。ひとたび属性にキー値がわりつけられると、キー値はローカルに定義されたあらゆるコミュニケータ上で属性を識別するために利用され、それらのコミュニケータにアクセスする際にも利用される。 copy_fn 関数は、コミュニケータが MPI_COMM_DUP によって複製された時に呼び出される。 copy_fn は MPI_Copy_function 型の関数でなくてはならず、次のように定義されている。

```
typedef int MPI_Copy_function(MPI_Comm *oldcomm, int *keyval,
                              void *extra_state, void *attribute_val_in,
                              void **attribute_val_out, int *flag)

FUNCTION COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
              ATTRIBUTE_VAL_OUT, FLAG, IERR)
    INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
    ATTRIBUTE_VAL_OUT IERR_LOGICAL FLAG
```

コピーコールバック関数は、 oldcomm 中のそれぞれのキー値に対して任意の順に呼び出される。それぞれのコピーコールバック関数の呼び出しは、キー値とそれに一致する属性に対して行なわれる。もし、 flag = 0 が返された場合には、複製されたコミュニケータ中の属性は削除されている。そうでない場合 (flag = 1) には新しい属性値が attribute_val_out を通じてセットされる。この関数が MPI_SUCCESS を返した場合には呼び出しは成功であり、エラー時にはエラーコードを返す。 (エラー時には、 MPI_COMM_DUP は失敗するであろう。)

copy_fn は C あるいは FORTRAN の MPI_NULL_COPY_FN あるいは MPI_DUP_FN とされることもある。その時には、 keyval に対するコールバックは発生しない。 MPI_NULL_COPY_FN は flag = 0 にセットし、 MPI_SUCCESS を返す以外はなにもしない関数である。単純なコピー関

数として MPI_DUP_FN 関数が用意されている。この関数は flag = 1 として attribute_val_in 中の値を attribute_val_out とし、 MPI_SUCCESS を返却値とする。

ユーザへのアドバイス　attribute_val_in と attribute_val_out は両方とも void * 型であるが、その使い方は異なっている。C のコピー関数は、 attribute_val_in 中の属性の値を MPI を通じでコピーし、 attribute_val_out に属性のアドレスを入れる。これはこの関数が、(新しい) 属性値を返すことができるようにするためである。 void * 型を利用するのは、みにくいキャストを避けるためである。

正しいとされるコピー関数には二種類ある。一つは属性を含んだデータ構造まるごとを複製するものである。もう一つは、他のデータ構造への参照をリファレンスカウント方式で持つだけの方法である。後者の方法では属性のその他の型はまったくコピーされない (それらは oldcomm だけで指定されていたかもしれない)。

(ユーザへのアドバイスの終わり)

実装者へのアドバイス　C のインターフェイスは C で作成されたキー値によるコピーと削除の関数を仮定している。 FORTRAN では FORTRAN で作成されたキー値を仮定している。(実装者へのアドバイスの終わり)

copy_fn に対するアナロジーは次のように定義される削除コールバック関数である。 delete_fn 関数は、コミュニケータが MPI_COMM_FREE によって削除された場合か、 MPI_ATTR_DELETE によって削除されなくてはならなくなった場合にはいつでも呼び出される。 delete_fn 関数は、 MPI_Delete_function 型でなくてはならず、それは次のように定義される。

```
typedef int MPI_Delete_function(MPI_Comm *comm, int *keyval,
                                void *attribute_val, void *extra_state);
```

この関数の FORTRAN 方式の宣言は次のようになる。

```
SUBROUTIONE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR
```

この関数は、 MPI_COMM_FREE と MPI_ATTR_DELETE により属性が削除されなくてはならなくなった時には常に呼び出される。この関数は成功時には MPI_SUCCESS を返し、失敗時にはエラーコードを返す (この時には、 MPI_COMM_FREE は失敗している)。

delete_fn は C 及び FORTRAN の両方で MPI_NULL_DELETE_FN である。この関数は MPI_SUCCESS を返す以外には何もしない。

特殊なキー値、 MPI_KEYVAL_INVALID は MPI_KEYVAL_CREATE ではけして返されない。なぜならその関数は、キー値の静的な初期化を行うために使われるからである。

MPI_KEYVAL_FREE(keyval)

入出力	keyval	整数値のキー値を解放する

```
int MPI_Keyval_free(int *keyval)

MPI_KEYVAL_FREE(KEYVAL, IERROR)
    INTEGER KEYVAL, IERROR
```

これらの関数は、現存の属性キーを解放する。この関数は keyval の値を MPI_KEYVAL_INVALID にセットする。使用中の属性キーを解放することは間違いではないことに注意。なぜなら実際の解放は、(そのプロセスでの他のコミュニケータによる) 全てのキーへの参照が解放されるまで行われないからである。これらの参照は、 MPI_ATTR_DELETE を呼んで一つの属性のインスタンスを解放するか、あるいは MPI_COMM_FREE を呼ぶことで解放されたコミュニケータに関係する全ての属性を解放し、プログラムで明示的に解放される必要がある。

MPI_ATTR_PUT(comm, keyval, attribute_val)

入力	comm	属性が付加されるコミュニケータ (ハンドル)
入力	keyval	MPI_KEYVAL_CREATE によって返されるキー値 (整数値)
入力	attribute_val	属性値

```
int MPI_Attr_put(MPI_Comm comm, int keyval, void* attribute_val)

MPI_ATTR_PUT(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
```

この関数は、 MPI_ATTR_GET によって次に取り出される正しいと保証された属性値 attribute_val をストアする。もし値が既に存在していた場合には、その結果は、あたかも MPI_ATTR_DELETE が呼び出されて以前の値が削除された (そしてコールバック関数 delete_fn が実行される) のちに新しい値がストアされたようにふるまう。値 keyval 無しの呼び出しは間違いであり、特に MPI_KEYVAL_INVALID は間違ったキー値である。 delete_fn 関数が、 MPI_SUCCESS と異なるエラーコードを返した時は失敗である。

MPI_ATTR_GET(comm, keyval, attribute_val, flag)

入力	comm	属性を付加するコミュニケータ (ハンドル)
入力	keyval	キー値 (整数値)
出力	attribute_val	属性値、 flag = false の時を除く
出力	flag	もし属性値が存在する場合には true、キーに関連した属性値が無い場合には false

```
int MPI_Attr_get(MPI_Comm comm, int keyval, void **attribute_val, int *flag)

MPI_ATTR_GET(COMM, KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR LOGICAL FLAG
```

この関数は、キーをもとに属性を検索する。値 keyval の無いキーでの呼び出しは誤りである。一方、キー値は存在するが、そのキーの comm に属性が存在しない場合にはその呼び出しは正しい。そのような場合には、呼び出しによって flag = false が返される。特に MPI_KEYVAL_INVALID は誤ったキー値である。

ユーザへのアドバイス MPI_Attr_get の呼び出しでは、 attribute_val 中の属性の値が渡される。 MPI_Attr_put は属性値の返されるアドレスが渡される。したがって、属性値それ自体は、 void * 型のポインタである。 MPI_Attr_put の引数 attribute_val は void * 型であり MPI_Attr_put(訳注 MPI_Attr_get の誤りか) の引数 attribute_val は void ** 型である。 (ユーザへのアドバイスの終わり)

MPI_ATTR_DELETE(comm, keyval)

入力	comm	属性を付加するコミュニケータ (ハンドル)
入力	keyval	削除された属性のキー値 (integer)

```
int MPI_Attr_delete(MPI_Comm comm, int keyval)

MPI_ATTR_DELETE(COMM, KEYVAL, IERROR)
    INTEGER COMM, KEYVAL, IERROR
```

この関数は、キーに基づいてキャッシュから属性を削除する。この関数は、 keyval の作成時に指定された属性を削除する関数 delete_fn を呼び出す。 delete_fn が MPI_SUCCESS でない値を返した場合はエラーである。

コミュニケータが関数 MPI_COMM_DUP を用いて複製される時は常に現在セットされている属性の全てのコールバックコピー関数が (任意の順で) 呼び出される。コミュニケータが関数 MPI_COMM_FREE を用いて削除された時には常に、現在セットされている属性の全てのコールバック削除関数が呼び出される。

5.7.2 属性の例

ユーザへのアドバイス この例では二度目以降の集団的通信操作が、効果的にキャッシングを利用するにはどのように書けばよいかを示している。コーディングスタイルは、 MPI 関数がエラーステータスのみを返すことを想定している。 (ユーザへのアドバイスの終わり)

```
/* key for this module's stuff: このモジュールのスタッフへのキー */
static int gop_key = MPI_KEYVAL_INVALID;
typedef struct
{
  int ref_count;                /* リファアレンスカウント (reference count) */

  /* 他のスタッフ (stuff), 欲しいもの以外はなんでも */
} gop_stuff_type;

Efficient_Collective_Op (comm, ...)
     MPI_Comm comm;
{
  gop_stuff_type *gop_stuff;
  MPI_Group group;
  int foundflag;

  MPI_Comm_group(comm, &group);
  if (gop_key == MPI_KEYVAL_INVALID) /* 最初の呼び出しによってキーを得る */
  {
    if ( ! MPI_keyval_create( gop_stuff_copier,
                              gop_stuff_destructor,
                             &gop_key, (void *)0));
    /* コピーと削除のコールバックを割り当てた上でキーを得る */
```

```
    MPI_Abort (comm, 99);
  }

  MPI_Attr_get (comm, gop_key, &gop_stuff, &foundflag);
  if (foundflag)
  {
    /* このモジュールはこのグループで既に実行された。
       よってキャッシュされた情報を利用する。 */
  }
  else
  {
    /* まだ何もキャッシュされていないグループである。そのためのコード。 */

    /* 最初に必要なスタッフの領域を確保し、リファレンスカウントを初期
       化する。  */
    gop_stuff = (gop_stuff_type *) malloc (sizeof(gop_stuff_type));

    if (gop_stuff == NULL) {    /* abort on out-of-memory error */ }

    gop_stuff -> ref_count = 1;
    /* 次に、必要なだけ *gop_stuff を埋める。この部分はここでは現れな
       い */

    /* 3 番目に、属性値として gop_stuff をストアする */
    MPI_Attr_put ( comm, gop_key, gop_stuff);
  }
  /* *gop_stuff の内容は常に大域的に操作される... */
}

/* 次のルーチンは、グループが解放される時に MPI によって呼び出される */

gop_stuff_destructor (comm, keyval, gop_stuff, extra)
     MPI_Comm comm;
     int keyval;
     gop_stuff_type *gop_stuff;
```

```
     void *extra;
{
  if (keyval != gop_key) {      /* abort -- programming error */ }

  /* グループが解放される時は、 gop_stuff への参照を 1 つ減らす */
  gop_stuff -> ref_count -= 1;

  /* 参照がなくなった場合には、領域を解放する。 */
  if (gop_stuff -> ref_count == 0) {
    free((void *)gop_stuff);
  }
}

/* 次のルーチンはグループがコピーされる場合に MPI によって呼び出される
   ルーチンである */
gop_stuff_copier (comm, keyval, extra, gop_stuff_in, gop_stuff_out, flag)
     MPI_Comm comm;
     int keyval;
     gop_stuff_type *gop_stuff_in, *gop_stuff_out;
     void *extra;
{
  if (keyval != gop_key) {      /* abort -- programming error */ }
  /* 新しいグループがこの gop_stuff を参照するのでリファアレンスカウン
     トを増加する */

  gop_stuff -> ref_count += 1;
  gop_stuff_out = gop_stuff_in;
}
```

section ゆるやかな同期モデルの定式化

本節ではゆるやかな同期モデルについて特にグループ内コミュニケーションに注意をはらって説明する。

5.7.3 基本説明

呼び出し側がコミュニケータ (コンテキストとグループを含む) を呼び出される側に渡す時、副プログラムの実行によってそのコミュニケータが副作用の影響を受けることのないようにしなくてはならない。つまり、プロセスを必要とするかもしれないコミュニケータ上ではアクティブな操作があってはならない。このモデルに従ってライブラリを記述すれば、“安全” な動作が実現される。このように設計されたライブラリにより、呼び出される側は、コミュニケータを介する通信をしていればどのような通信も他の通信に邪魔されることはない。したがって、(コミュニケータ上にあらかじめ割り付けられたコンテキストによるような場合でも) 同期なしで新しくコミュニケータを作成するようなものをうまく実装することが可能になり、これによって性能に重大な影響を与えるようなオーバーヘッドを必要としなくなる。

このような形態によってもたらされる安全は、例えば、ライブラリルーチンへのディスクリプタの配列を通して利用するような一般の計算機科学で利用されているもののアナロジーである。ライブラリルーチンは、正当でかつ変更可能なディスクリプタのようなものとして考えられている。

5.7.4 実行モデル

ゆるやかな同期モデルでは、実行中のプロセスがそれぞれの手続きを呼び出し、並列手続きへ制御をうつすことで効率をあげることができる。この呼び出しは集団的操作である。それは実行グループの全てのプロセスで実行され、呼び出しは全てのプロセスでほぼ同じ順に実行される。しかしながら、呼び出しには同期がいらない。

並列手続きがあるプロセス中でアクティブであるとは、そのプロセスが集団的手続きを実行しているグループに属しており、グループのメンバのいくつかが現在手続きのコードを実行中であるということである。もし並列手続きがあるプロセス中でアクティブならば、たとえ現在、この手続きのコードを実行していない場合でも、このプロセスはこの手続きに関するメッセージを受けとることができるであろう。

静的なコミュニケータ割り当て

任意の時点において、どのプロセスにおいてもアクティブな並列手続きが高々一つであり、その手続きを実行しているプロセスのグループが固定されているような場合には、コミュニケータを静的に割り当てることができる。例えば、並列手続きの呼び出しがすべて、全プロセスによって行なわれ、プロセスはみなシングルスレッドであり、再帰呼び出しがないような場合が挙げられる

そのような場合には、コミュニケータはそれぞれの手続きへ静的に割り付けることができる。つまり静的な割り付けが、初期化のコードの部分であらかじめ可能になる。もしも並列手続

きをライブラリ中に作成することができれば、それぞれのライブラリ中でただ一つの手続きがそれぞれのプロセスで並行してアクティブになる。したがって 1 つのライブラリには 1 つのコミュニケータだけを割り当てることができる。

動的なコミュニケータ割り当て

もし新しい並列手続きが同一の並列手続きを実行するグループの一部から常に呼び出される場合には、並列手続きの呼び出しは深くネストしたものとなる。したがって、同一の並列手続きを実行するプロセスは、同一の実行スタックを持つ。

そのような場合には、それぞれの新しい並列手続きごとに、新しいコミュニケータが動的に割り当てられる必要がある。この割り当ては呼び出し側で行われる。新しいコミュニケータは MPI_COMM_DUP を呼び出すことで作成され、呼び出される側の実行グループは、呼び出し側の実行グループと区別される。あるいは、呼び出し側の実行グループが、個々の並列ルーチンを実行するいくつかの副グループに分解される場合には、 MPI_COMM_SPLIT が呼ばれ、新しいコミュニケータが作成される。新しいコミュニケータは、呼び出されるルーチンへ引数として渡される。

それぞれの呼び出しでは、新しいコミュニケータの作成を減らすか、ある場合には避けることもできる。たとえば、もしも実行グループが分割されない場合には、コミュニケータのスタックをあらかじめ割り当てておき、それを再帰呼び出しのスタックとみなして次回から再利用することで呼び出しの負荷を軽減できる。

たとえ呼び出し側と呼び出される側で同一のコミュニケータを利用した場合でも、通信が順序付けられているという特性を利用することで、呼び出し側と呼び出される側の通信の混乱を避けることができる。そのような場合には次の 2 つの規則を守る必要がある。

- 手続き呼び出しの前 (あるいは手続きから戻る前) に送ったメッセージは、その手続きの呼び出し (あるいはリターン) の前に受信を終了すること。
- メッセージは常に送信元によって選択されること。 (MPI_ANY_SOURCE によって作成されたもの使用しない。)

一般的な場合

一般に、同一グループ中に、同じ並列手続きを複数並行してアクティブに呼び出す場合が存在するであろう。例えば呼び出しがあまりネストしない場合である。このとき、新しいコミュニケータをそれぞれの呼び出しで作成する必要がある。 2 つの異った並列手続きがオーバーラップするプロセスの集合上で並行して呼び出された場合のコミュニケータの作成を調整することは、ユーザの責任である。

Chapter6

プロセス・トポロジー

6.1 はじめに

本章では MPI のトポロジーのメカニズムについて解説する。トポロジーとはグループ内通信で使用できる特別なオプション属性であり、グループ間通信では使用できない。トポロジーは (コミュニケータ内で) グループをなすプロセス群に便利な命名メカニズムを提供し、さらに、ランタイム・システムがプロセス群をハードウェアにマッピングするのを補助することもある。

第 5章で説明したように、 MPI のプロセス・グループを n 個のプロセスの集まりとすると、グループ内の各プロセスには 0 から n-1 までの間のランクが割り当てられる。多くの並列アプリケーションでは、プロセス群への線形のランク付けは、プロセスの論理的通信パターン (これは通常、もとになる問題の幾何学的配置と利用する数値アルゴリズムによって決定される) を十分には反映していない。多くの場合、プロセス群は二次元あるいは三次元格子のようなトポロジカル・パターンに配置される。より一般的には、論理的プロセス配置はグラフで表現される。この章では、この論理的プロセス配置を「仮想トポロジー」と呼ぶことにする。

仮想的プロセス・トポロジーは、もとになる物理的ハードウェアのトポロジーとはまったく異なる。与えられたマシン上での通信性能を改善できる場合には、システムは仮想トポロジーを利用して、プロセス群を物理的プロセッサに割り当ててもよい。しかしながら、この割り当て (以下ではマッピングと呼ぶ) をどのように行うかということについては、 MPI の対象外である。他方で、仮想トポロジーを記述することは、アプリケーションにのみ依存し、マシンとは独立している。この章で提案する関数とは、マシンとは独立したマッピングのみを扱う関数である。

> 根拠 物理的なマッピングについては取りあげないが、ランタイム・システムは仮想トポロジーに関する情報があれば、これをアドバイスとして使用しうる。格子 / トーラス構造を超立方体や格子などのハードウェア・トポロジーにマッピングする手法は十分確立されている。より複雑なグラフ構造については、適切な発見的手法によりほとんど最適な結果が

得られることがよくある [20]。他方、ユーザーが論理的プロセス配置を「仮想トポロジー」として指定する方法がなければ、ランダム・マッピングが最もありうる配置となる。一部のマシンでは、これは相互接続ネットワーク内で不必要な競合を引き起こす。最近のワームホール・ルーティング・アーキテクチャ上での適切なプロセス - プロセッサ間マッピングから得られた、性能改善の予測値及び測定値に関する詳細が [10, 9] に記載されている。

性能面での利点のほかに、仮想トポロジーにはメッセージ通信プログラミングにおいてプログラムの可読性や表記能力を著しく向上させるという、便利なプロセス命名機構としての面もある。（根拠の終わり）

6.2 仮想トポロジー

プロセスの集合の通信パターンはグラフで表現できる。ノードでプロセスを表し、相互に通信するプロセスをエッジで接続する。 MPI はグループ内のプロセスの任意のペアの間のメッセージ通信を提供する。チャネルを明示的に開く必要はない。したがって、ユーザーが定義するプロセス・グラフに「ミッシング・リンク」があっても、対応するプロセス間でメッセージを交換できないということはない。これはむしろ、この接続が仮想トポロジーでは無視されるということを意味する。この戦略は、このトポロジーではこの通信の経路を指名するうまい方法がないということを示唆している。ほかにも、自動マッピング・ツールが (もしこれがランタイム環境に存在すればであるが)、マッピングの際にこのエッジを考慮しないということも考えられる。通信グラフ内のエッジは重み付きではないので、プロセスは単に接続されているか、そうでないかのいずれかである。

根拠　PARMACS [5, 8] での同様の手法による経験から、通常はこの情報があれば、十分よいマッピングができることがわかっている。更に、より正確な仕様を求めれば、ユーザーがそれをセットアップするのがより難しくなり、インターフェース関数が実質的により複雑なものとなるであろう。（根拠の終わり）

どのようなアプリケーションでも、仮想トポロジーをグラフによって指定するだけで十分である。しかし、多くのアプリケーションではグラフ構造は規則的であり、グラフを詳細にセットアップすることはユーザーにとっては不便なもので、実行時に効率を落とすことも考えられる。並列アプリケーションの大部分は環、二次あるいは高次元の格子、またはトーラスのようなプロセス・トポロジーを使用する。これらは次元数と各座標方向におけるプロセスの個数を指定すれば完全に定義できる、デカルト座標で表される直積構造である。また一般的にいって、格子とトーラスのマッピングは一般グラフのマッピングよりは簡単な問題である。そこでこれら直積で表されるケース、以下ではこれらをカルテシアンと呼ぶことにするが、これらは改めて別個に取り扱うのが望ましい。

カルテシアン構造のプロセス座標には 0 から番号をつける。カルテシアン構造を持つプロセス群には常に行優先番号付けを使用する。このことは、例えば (2×2) 格子における 4 個のプロセスのグループ・ランクと座標には次の関係がある事を意味する。

座標 (0,0) : ランク 0
座標 (0,1) : ランク 1
座標 (1,0) : ランク 2
座標 (1,1) : ランク 3

6.3 MPI への埋め込み

この章で定義されている仮想トポロジーを支援する機能は、 MPI の他の部分と整合がとれており、また可能な限り、他の箇所で定義された関数を利用している。トポロジー情報はコミュニケータに付加される。この情報は第 5章で説明したキャッシング機構を用いてコミュニケータに追加される。

6.4 トポロジー関数の概要

関数 MPI_GRAPH_CREATE および MPI_CART_CREATE は、一般的な (グラフ) 仮想トポロジーおよびカルテシアン・トポロジーを生成するのに使う。これらのトポロジー生成関数は集団的である。他の集団呼び出しと同様、呼び出しが同期するしないに関わらず、正しく動作するようにプログラムを書かなければならない。

トポロジー生成関数は入力として既存のコミュニケータ comm_old をとる。これはトポロジーをマッピングするプロセスの集合を定義している。これらの関数はトポロジー構造をキャッシュ情報に持つ新規コミュニケータ comm_topol を生成する (第 5章を参照)。関数 MPI_COMM_CREATE の場合と同様に、 comm_old が持つキャッシュ情報は comm_topol へは伝播しない。

MPI_CART_CREATE を使用すると、任意の次元のカルテシアン構造が記述できる。この関数では、各座標方向についてプロセス構造が周期的か否かを指定する。 n 次元超立方体は各座標方向につき 2 個のプロセスを持つ n 次元トーラスであることに注意されたい。したがって、超立方体構造に対する特別なサポートは不要である。ローカルな補助関数 MPI_DIMS_CREATE を使用すると、与えられた次元数に対して、バランスのとれたプロセス配置を求めることができる。

根拠 EXPRESS [22] と PARMACS にも同様の関数が含まれている。(根拠の終わり)

関数 MPI_TOPO_TEST を使用すると、コミュニケータに付加されたトポロジーについて問い合わせできる。トポロジー情報は、一般グラフについては関数 MPI_GRAPHDIMS_GET およ

び MPI_GRAPH_GET を使用して、カルテシアン・トポロジーについては MPI_CARTDIM_GET および MPI_CART_GET を使用して、コミュニケータから抽出できる。カルテシアン・トポロジーを操作するために、いくつかの追加関数が用意されている。関数 MPI_CART_RANK 及び MPI_CART_COORDS は、カルテシアン座標をグループ・ランクへ、逆にグループ・ランクをカルテシアン座標へ変換する。関数 MPI_CART_SUB を使用すると、(MPI_COMM_SPLIT と同様に) 部分カルテシアン空間を抽出できる。関数 MPI_CART_SHIFT はプロセスが一つの次元におけるその近傍と通信するために必要な情報を提供する。2 つの関数 MPI_GRAPH_NEIGHBORS_COUNT 及び MPI_GRAPH_NEIGHBORS を使用すると、グラフの中のノードの近傍を抽出できる。関数 MPI_CART_SUB は、入力コミュニケータのグループ上で集団的である。他のすべての関数はローカルである。

2 つの追加関数 MPI_GRAPH_MAP 及び MPI_CART_MAP については最後の節で紹介する。一般にこれらの関数はユーザーが直接呼び出す関数ではない。しかし、第 5章で説明したコミュニケータ操作関数と一緒に使用すれば、他のすべてのトポロジー関数を実装できる。このような実装については 第?? 節で概説する。

6.5 トポロジー・コンストラクタ

6.5.1 カルテシアン・コンストラクタ

MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart)

入力	comm_old	入力コミュニケータ (ハンドル)
入力	ndims	カルテシアン格子の次元数 (整数型)
入力	dims	各次元に対しそのプロセス数を指定したサイズ ndims の整数型配列
入力	periods	各次元に対しその格子が周期的 (true) か否か (false) を指定したサイズ ndims の論理型配列
入力	reorder	ランク付けを変更してよい (true) か否か (false) (論理型)
出力	comm_cart	新しく生成されたカルテシアン・トポロジーを持つコミュニケータ (ハンドル)

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods,
              int reorder, MPI_Comm *comm_cart)

MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)
```

```
    INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
    LOGICAL PERIODS(*), REORDER
```

MPI_CART_CREATE は、カルテシアン・トポロジー情報を付加した新しいコミュニケータのハンドルを返す。 reorder = false であれば、新規グループの中の各プロセスのランクは旧グループでのそれと同じである。そうでない場合は、(仮想トポロジーを物理的マシンへうまく埋め込めるように) この関数はプロセスを並べ替える場合がある。カルテシアン格子の総サイズがグループ comm_old のサイズよりも小さければ、 MPI_COMM_SPLIT の場合と同様に、いくつかのプロセスには MPI_COMM_NULL を返す。この関数をグループ・サイズよりも大きな格子を指定して呼び出すことは間違いである。

6.5.2 カルテシアン支援関数: MPI_DIMS_CREATE

カルテシアン・トポロジーでは関数 MPI_DIMS_CREATE を使用して、バランスをとるべきグループ内のプロセスの個数と、ユーザーが指定できるオプショナルな制約に応じて、座標方向ごとにバランスのとれたプロセス配置を選ぶことができる。これには (グループ MPI_COMM_WORLD のサイズの) すべてのプロセスを n 次元トポロジーに分割するという用途がある。

MPI_DIMS_CREATE(nnodes, ndims, dims)

入力	nnodes	格子中のノード数 (整数型)
入力	ndims	カルテシアン・トポロジーの次元数 (整数型)
入出力	dims	各次元に対しそのノード数を指定したサイズ ndims の整数型配列

```
int MPI_Dims_create(int nnodes, int ndims, int *dims)
```

```
MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)
    INTEGER NNODES, NDIMS, DIMS(*), IERROR
```

配列 dims のエントリは、 ndims 次元でかつ全部で nnodes 個のノードのカルテシアン格子を記述するように設定される。次元は適切な分割アルゴリズムを使用して、できるだけ互いに近い値になるように設定される。呼び出し側は更に配列 dims の要素を指定することで、このルーチンの操作を制限することができる。 dims[i] が正数に設定されている場合、このルーチンは次元 i のノードの個数を (その値のままにして) 変更しない。 dims[i] = 0 となっているエントリのみが、このルーチンの呼び出しにより変更される。

dims[i] の入力値が負の呼び出しは間違いである。 nnodes が $\prod_{i,dims[i]\neq 0} dims[i]$ の倍数でない場合も間違いである。

呼び出しにより設定された dims[i] は非増加順に並べられる。配列 dims はルーチン MPI_CART_CREATE への入力として使用するのに適している。 MPI_DIMS_CREATE はローカルである。

例 **6.1**

呼び出し前の dims	関数呼び出し	戻り時の dims
(0,0)	MPI_DIMS_CREATE(6, 2, dims)	(3,2)
(0,0)	MPI_DIMS_CREATE(7, 2, dims)	(7,1)
(0,3,0)	MPI_DIMS_CREATE(6, 3, dims)	(2,3,1)
(0,3,0)	MPI_DIMS_CREATE(7, 3, dims)	間違った呼び出し

6.5.3 一般 (グラフ)・コンストラクタ

MPI_GRAPH_CREATE(comm_old, nnodes, index, edges, reorder, comm_graph)

入力	comm_old	入力コミュニケータ (ハンドル)
入力	nnodes	グラフのノードの個数 (整数型)
入力	index	ノードの次数を表す整数型配列 (下記参照)
入力	edges	グラフのエッジを表す整数型配列 (下記参照)
入力	reorder	ランク付けを変更してよい (true) か否か (false) (論理型)
出力	comm_graph	グラフ・トポロジーを付け加えたコミュニケータ (ハンドル型)

```
int MPI_Graph_create(MPI_Comm comm_old, int nnodes, int *index, int *edges,
            int reorder, MPI_Comm *comm_graph)
```

```
MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES, REORDER, COMM_GRAPH,
            IERROR)
    INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERROR
    LOGICAL REORDER
```

MPI_GRAPH_CREATE は、グラフ・トポロジー情報が付加された新しいコミュニケータのハンドルを返す。 reorder = false であれば、新規グループの中の各プロセスのランクは旧グループでのそれと同一である。そうでない場合には、この関数はプロセスを並べ替える場合がある。グラフのサイズ nnodes が comm_old のグループのサイズよりも小さければ、 MPI_CART_CREATE

および MPI_COMM_SPLIT と同様に、いくつかのプロセスには MPI_COMM_NULL を返す。この関数を入力コミュニケータのグループ・サイズよりも大きなグラフを指定して呼び出すことは間違いである。

3 つのパラメータ nnodes, index および edges でグラフ構造を定義する。 nnodes はグラフのノードの個数である。ノードには 0 から nnodes-1 までの番号が付けられる。配列 index の i 番目のエントリには、最初の i 個のグラフのノードの近傍の総数が格納される。ノード 0, 1, ..., nnodes-1 の近傍のリストは、配列 edges の中の連続した位置に格納される。配列 edges はエッジ・リストを平坦化した表現である。 index のエントリの総数は nnodes で、 edges のエントリの総数はグラフのエッジの本数に等しい。

引数 nnodes, index および edges の定義については、以下の簡単な例で説明する。

例 **6.2** 以下の隣接行列を持つ 4 個のプロセス 0, 1, 2, 3 があるとする。

プロセス	近傍
0	1, 3
1	0
2	3
3	0, 2

すると、入力引数は次のとおりである。

nnodes = 4
index = 2, 3, 4, 6
edges = 1, 3, 0, 3, 0, 2

したがって C 言語では、 index[0] はノード 0 の次数であり、 index[i]-index[i-1] はノード i, i = 1, ..., nnodes-1 の次数である。ノード 0 の近傍のリストは 0 ≤ j ≤ index[0]−1 の各 j について edges[j] に格納され、ノード i, i > 0 の近傍のリストは index[i − 1] ≤ j ≤ index[i] − 1 の各 j について edges[j] に格納される。

Fortran 言語では、 index(1) はノード 0 の次数であり、 index(i+1)-index(i) はノード i, i=1, ..., nnodes-1 の次数である。ノード 0 の近傍のリストは 1 ≤ j ≤ index(1) の各 j について edges(j) に格納され、ノード i, i > 0 の近傍のリストは index(i) + 1 ≤ j ≤ index[i + 1] の各 j について edges(j) に格納される。

実装者へのアドバイス 以下のトポロジー情報がコミュニケータに格納されるであろう。

- トポロジーの型 (カルテシアン / グラフ)
- カルテシアン・トポロジーの場合:

1. ndims (次元数),
2. dims (各座標方向についてのプロセスの個数),
3. periods (周期情報),
4. own_position (格子内での自位置、 rank と dims から計算で求めることも可能)

- グラフ・トポロジーの場合:

1. index,
2. edges,

これらはグラフ構造を定義するベクトルである。

グラフ構造については、ノードの個数はグループ内のプロセスの個数に等しい。したがって、ノードの個数は明示的に格納しなくてもよい。配列 index の最初に 0 を挿入すると、トポロジー情報へのアクセスが簡単になる。(実装者へのアドバイスの終わり)

6.5.4 トポロジー問い合わせ関数

上記関数のいずれか一つでトポロジーを定義している場合、問い合わせ関数を使用してトポロジー情報を調べることができる。これらはすべてローカルな呼び出しである。

MPI_TOPO_TEST(comm, status)

入力	comm	コミュニケータ (ハンドル)
出力	status	コミュニケータ comm のトポロジーの型 (選択型)

```
int MPI_Topo_test(MPI_Comm comm, int *status)
```

```
MPI_TOPO_TEST(COMM, STATUS, IERROR)
    INTEGER COMM, STATUS, IERROR
```

関数 MPI_TOPO_TEST はコミュニケータに割り当てられているトポロジーの型を返す。出力値 status は次のうちのいずれか一つである。

MPI_GRAPH	グラフ・トポロジー
MPI_CART	カルテシアン・トポロジー
MPI_UNDEFINED	トポロジーなし

MPI_GRAPHDIMS_GET(comm, nnodes, nedges)

入力	comm	グラフ構造を持つグループのコミュニケータ (ハンドル)
出力	nnodes	グラフのノードの個数 (整数型) (グループの中のプロセスの数と等しい)
出力	nedges	グラフのエッジの個数 (整数型)

```
int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)

MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, IERROR)
    INTEGER COMM, NNODES, NEDGES, IERROR
```

関数 MPI_GRAPHDIMS_GET 及び MPI_GRAPH_GET は MPI_GRAPH_CREATE によってコミュニケータに付加されたグラフ・トポロジー情報を検索する関数である。

MPI_GRAPHDIMS_GET から得られる情報を使用すると、 MPI_GRAPH_GET の以下の呼び出しで、ベクトル index および edges のサイズを正確に決めることができる。

MPI_GRAPH_GET(comm, maxindex, maxedges, index, edges)

入力	comm	グラフ構造を持つコミュニケータ (ハンドル)
入力	maxindex	呼び出し側プログラムのベクトル index のサイズ (整数型)
入力	maxedges	呼び出し側プログラムのベクトル edges のサイズ (整数型)
出力	index	グラフ構造を格納した整数型配列 (詳細については MPI_GRAPH_CREATE の定義を参照)
出力	edges	グラフ構造を格納した整数型配列

```
int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int *index,
              int *edges)

MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR)
    INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR
```

MPI_CARTDIM_GET(comm, ndims)

入力	comm	カルテシアン構造を持つコミュニケータ (ハンドル)
出力	ndims	カルテシアン構造の次元数 (整数型)

```
int MPI_Cartdim_get(MPI_Comm comm, int *ndims)

MPI_CARTDIM_GET(COMM, NDIMS, IERROR)
    INTEGER COMM, NDIMS, IERROR
```

関数 MPI_CARTDIM_GET 及び MPI_CART_GET は MPI_CART_CREATE によってコミュニケータに付加されたカルテシアン・トポロジー情報を返す。

MPI_CART_GET(comm, maxdims, dims, periods, coords)

入力	comm	カルテシアン構造を持つコミュニケータ (ハンドル)
入力	maxdims	呼び出し側プログラムのベクトル dims, periods および coords のサイズ (整数型)
出力	dims	各次元ごとのプロセスの数 (整数型配列)
出力	periods	各次元について周期的か否か (true/false) (論理型配列)
出力	coords	カルテシアン構造中の呼び出しプロセスの座標 (整数型配列)

```
int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int *periods,
             int *coords)

MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
    INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
    LOGICAL PERIODS(*)
```

MPI_CART_RANK(comm, coords, rank)

入力	comm	カルテシアン構造を持つコミュニケータ (ハンドル)
入力	coords	プロセスのカルテシアン座標を指定した (サイズ ndims の) 整数型配列
出力	rank	指定したプロセスのランク (整数型)

```
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)

MPI_CART_RANK(COMM, COORDS, RANK, IERROR)
    INTEGER COMM, COORDS(*), RANK, IERROR
```

関数 MPI_CART_RANK は、カルテシアン構造を持つプロセス・グループに対して、論理的プロセス座標を 1 対 1 ルーチンが使用するプロセス・ランクに変換する。

periods(i) = true となる次元 i について、座標 coords(i) が範囲外である、つまり coords(i) $<$ 0 または coords(i) $\geq$ dims(i) であれば、自動的に区間 0 $\leq$ coords(i) $<$ dims(i) へシフトされる。非周期的次元の場合には、範囲外の座標を指定することは間違いである。

MPI_CART_COORDS(comm, rank, maxdims, coords)

入力	comm	カルテシアン構造を持つコミュニケータ (ハンドル)
入力	rank	グループ comm の中でのプロセスのランク (整数型)
入力	maxdims	呼び出し側プログラムのベクトル coords のサイズ (整数型)
出力	coords	指定したプロセスのカルテシアン座標を格納する (サイズ ndims の) 整数型配列

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)

MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)
    INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR
```

MPI_CART_COORDS は、逆のマッピングであるランクから座標への変換を行う。

MPI_GRAPH_NEIGHBORS_COUNT(comm, rank, nneighbors)

入力	comm	グラフ・トポロジーを持つコミュニケータ (ハンドル)
入力	rank	グループ comm の中でのプロセスのランク (整数型)
出力	nneighbors	指定したプロセスの近傍の数 (整数型)

```
int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)
```

```
MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERROR)
    INTEGER COMM, RANK, NNEIGHBORS, IERROR
```

MPI_GRAPH_NEIGHBORS_COUNT 及び MPI_GRAPH_NEIGHBORS は一般的なグラフ・トポロジーの隣接情報を提供する。

MPI_GRAPH_NEIGHBORS(comm, rank, maxneighbors, neighbors)

入力	comm	グラフ・トポロジーを持つコミュニケータ (ハンドル)
入力	rank	グループ comm の中でのプロセスのランク (整数型)
入力	maxneighbors	配列 neighbors のサイズ (整数型)
出力	neighbors	指定したプロセスの近傍にあたるプロセスのランク (整数型配列)

```
int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors,
              int *neighbors)
```

```
MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR)
    INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR
```

例 6.3 comm はシャッフル交換トポロジーを持つコミュニケータと仮定する。このグループは 2^n 個のプロセスを持つ。各プロセスには $a_1, \ldots, a_n$ $(a_i \in \{0, 1\})$ から成るラベルが付けられており、各々 3 個の近傍を持つ。交換とは $\mathrm{exchange}(\mathrm{a_1}, \ldots, \mathrm{a_n}) = \mathrm{a_1}, \ldots, \mathrm{a_{n-1}}, \bar{\mathrm{a}}_\mathrm{n}$ $(\bar{\mathrm{a}} = 1 - \mathrm{a})$、シャッフルとは $\mathrm{shuffle}(\mathrm{a_1}, \ldots, \mathrm{a_n}) = \mathrm{a_2}, \ldots, \mathrm{a_n}, \mathrm{a_1}$、アンシャッフルとは $\mathrm{unshuffle}(\mathrm{a_1}, \ldots, \mathrm{a_n}) = \mathrm{a_n}, \mathrm{a_1}, \ldots, \mathrm{a_{n-1}}$ であるとする。$n = 3$ ではグラフの隣接リストは以下の通りである。

ノード	交換	シャッフル	アンシャッフル
	neighbors(1)	neighbors(2)	neighbors(3)
0 (000)	1	0	0
1 (001)	0	2	4
2 (010)	3	4	1
3 (011)	2	6	5
4 (100)	5	1	2
5 (101)	4	3	6
6 (110)	7	5	3
7 (111)	6	7	7

コミュニケータ comm には上記トポロジーが付加されていると仮定する。以下のコードはこの３種類の近傍を順繰りに巡り、それぞれに適切な置換を実行する。

```
C  仮定: 各プロセスは実数 A を保持しているとする。
C  近傍の情報を抽出する
      CALL MPI_COMM_RANK(comm, myrank, ierr)
      CALL MPI_GRAPH_NEIGHBORS(comm, myrank, 3, neighbors, ierr)
C  交換による置換を実行
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(1), 0,
     +     neighbors(1), 0, comm, status, ierr)
C  シャッフルによる置換を実行
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(2), 0,
     +     neighbors(3), 0, comm, status, ierr)
C  アンシャッフルによる置換を実行
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(3), 0,
     +     neighbors(2), 0, comm, status, ierr)
```

6.5.5 カルテシアン座標のシフト

プロセス・トポロジーがカルテシアン構造であれば、座標方向に沿った MPI_SENDRECV 操作を使用して、データのシフトを実行したい場合がある。 MPI_SENDRECV は入力として、受信については送信元プロセスのランクを、送信については送信先プロセスのランクをとる。カルテシアン・プロセス・グループに対して関数 MPI_CART_SHIFT が呼ばれると、呼び出しプロセスに上記識別子を与える。これは MPI_SENDRECV に渡すことができる。ユーザーは、座標方向と (正または負の) シフトするステップ数を指定する。この関数はローカルである。

MPI_CART_SHIFT(comm, direction, disp, rank_source, rank_dest)

入力	comm	カルテシアン構造を持つコミュニケータ (ハンドル)
入力	direction	シフトを行なう座標の次元 (整数型)
入力	disp	変位 (> 0: 上方へのシフト、< 0: 下方へのシフト) (整数型)
出力	rank_source	送信元プロセスのランク (整数型)
出力	rank_dest	送信先プロセスのランク (整数型)

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source,
              int *rank_dest)

MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR)
    INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR
```

引数 direction はシフトの次元、つまりシフトで値が変更される座標を示している。 ndims が次元数の時、座標には 0 から ndims-1 までの番号が付けられている。

MPI_CART_SHIFT は、指定された座標方向におけるカルテシアン・グループが周期的か否かに応じて、循環シフトかまたはエンド - オフ・シフトの識別子を提供する。エンド - オフ・シフトの場合、値 MPI_PROC_NULL を rank_source または rank_dest に返すことがある。これは、シフトの送信元または送信先が範囲外であることを示している。

例 **6.4** コミュニケータ comm には二次元周期的カルテシアン・トポロジーが付加されているとする。また、 REAL 型の二次元配列が 1 プロセスにつき 1 要素という形で変数 A に格納されているとする。この配列を列 i を i ステップだけシフトするように (垂直に、つまり列に沿って) ずらしたい。

```
....
C プロセスランクを求める
      CALL MPI_COMM_RANK(comm, rank, ierr)
C カルテシアン座標を求める
      CALL MPI_CART_COORDS(comm, rank, maxdims, coords, ierr)
C シフトの送信元と送信先を計算する
      CALL MPI_CHART_SHIFT(comm, 1, coords(2), source, dest, ierr)
C 配列をずらす
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, dest, 0, source, 0, comm,
     +                          status, ierr)
```

ユーザへのアドバイス Fortran 言語において、 DIRECTION = i で示される次元は DIMS(i+1) 個のノードを持つ。ここで DIMS は格子を生成するのに使用した配列である。 C 言語においては、 direction = i で示される次元は dims[i] で指定される次元である。 (ユーザへのアドバイスの終わり)

6.5.6 カルテシアン構造の分割

MPI_CART_SUB(comm, remain_dims, newcomm)

入力	comm	カルテシアン構造を持つコミュニケータ (ハンドル)
入力	remain_dims	remain_dims の i 番目のエントリが i 番目の次元が部分格子に残る (true) か否か (false) を指定する (論理型ベクトル)
出力	newcomm	呼び出しプロセスを含む部分格子を持つコミュニケータ (ハンドル)

```
int MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm *newcomm)

MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, IERROR)
    INTEGER COMM, NEWCOMM, IERROR
    LOGICAL REMAIN_DIMS(*)
```

カルテシアン・トポロジーを MPI_CART_CREATE で生成した場合には、関数 MPI_CART_SUB を使用すると、コミュニケータ・グループを低次元部分カルテシアン格子を形成する部分グループに分割し、各部分グループごとに部分格子のなすカルテシアン・トポロジーを付加されたコミュニケータを生成することができる。(この関数は、 MPI_COMM_SPLIT と密接に関係している。)

例 **6.5** MPI_CART_CREATE(..., comm) によって $(2 \times 3 \times 4)$ 格子が定義されていると仮定する。 remain_dims = (true, false, true) とする。このとき、次の関数呼び出しにより、

```
MPI_CART_SUB(comm, remain_dims, comm_new)
```

2×4 カルテシアン・トポロジーを成す 8 個のプロセスを持つ 3 個のコミュニケータが生成される。 remain_dims = (false, false, true) とすると、関数 MPI_CART_SUB(comm, remain_dims, comm_new) の呼び出しで、 1 次元カルテシアン・トポロジーを成す 4 個のプロセスを持つ、重なり合わない 6 個のコミュニケータが生成される。

6.5.7 低レベル・トポロジー関数

この節で紹介する 2 つの追加関数を使用すると、他のすべてのトポロジー関数を実装することができる。一般に、これらは MPI が提供している機能に加えて仮想トポロジー機能を作成したい場合でないかぎり、ユーザーが直接呼び出すような関数ではない。

MPI_CART_MAP(comm, ndims, dims, periods, newrank)

入力	comm	入力コミュニケータ (ハンドル)
入力	ndims	カルテシアン構造の次元数 (整数型)
入力	dims	各座標方向に対しそのプロセス数を指定したサイズ ndims の整数型配列
入力	periods	各座標方向に対しその格子が周期的か否かを指定したサイズ ndims の論理型配列
出力	newrank	呼び出しプロセスの再順序付けられたランク。呼び出しプロセスが格子に属さなければ MPI_UNDEFINED となる。 (整数型)

```
int MPI_Cart_map(MPI_Comm comm, int ndims, int *dims, int *periods,
              int *newrank)

MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR)
    INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR
    LOGICAL PERIODS(*)
```

MPI_CART_MAP は呼び出しプロセスの物理マシン上での「最適な」配置を計算する。この関数の実装としては、常に呼び出しプロセスのランクを返す、つまり順序を変更しないものが考えられる。

> 実装者へのアドバイス 関数 MPI_CART_CREATE(comm, ndims, dims, periods, reorder, comm_cart) は、 reorder = true の場合には、 MPI_CART_MAP(comm, ndims, dims, periods, newrank) を呼び出し、次に newrank ≠ MPI_UNDEFINED であれば color = 0 とし、それ以外の場合には color = MPI_UNDEFINED、そして key = newrank として MPI_COMM_SPLIT(comm, color, key, comm_cart) を呼び出すことで実装できる。
>
> 関数 MPI_CART_SUB(comm, remain_dims, comm_new) は、 color として破棄された次元をまとめて一つの数にエンコーディングしたものと、 key として残された次元をまとめ

て一つの数にエンコーディングしたものを用いて、 MPI_COMM_SPLIT(comm, color, key, comm_new) を呼び出すことで実装できる。

他のすべてのカルテシアン・トポロジー関数は、コミュニケータにキャッシュされているトポロジー情報を使用し、ローカルに実装できる。 (実装者へのアドバイスの終わり)

上の関数に対応する、一般的グラフ構造のための新しい関数は次のとおりである。

MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank)

入力	comm	入力コミュニケータ (ハンドル)
入力	nnodes	グラフのノードの個数 (整数型)
入力	index	グラフ構造を指定する整数型配列、 MPI_GRAPH_CREATE を参照
入力	edges	グラフ構造を指定する整数型配列
出力	newrank	しプロセスの再順序付けられたランク。呼び出しプロセスがグラフに属さなければ MPI_UNDEFINED となる。 (整数型)

```
int MPI_Graph_map(MPI_Comm comm, int nnodes, int *index, int *edges,
              int *newrank)

MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR)
    INTEGER COMM, NNODES, INDEX(*), EDGES(*), NEWRANK, IERROR
```

実装者へのアドバイス　関数 MPI_GRAPH_CREATE(comm, nnodes, index, edges, reorder, comm_graph) は、 reorder = true の場合には、 MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank) を呼び出し、次に newrank $\neq$ MPI_UNDEFINED であれば color = 0 とし、それ以外の場合には color = MPI_UNDEFINED、そして key = newrank として MPI_COMM_SPLIT(comm, color, key, comm_graph) を呼び出すことで実装できる。

他のすべてのグラフ・トポロジー関数は、コミュニケータにキャッシュされているトポロジー情報を使用し、ローカルに実装できる。 (実装者へのアドバイスの終わり)

6.6 アプリケーション例

例 **6.6** 図 6.1の例は、格子定義と問い合わせ関数をアプリケーション・プログラムでどのように使用するかを示している。偏微分方程式、例えばポアソン方程式、を矩形領域について解いてみる。まず、プロセス群を二次元構造に組織する。各プロセスは 4 方向 (上下左右) の近傍にそのランクを問い合わせる。数値問題は反復法で解く。詳細はサブルーチン `relax` の中に隠蔽されている。

各緩和ステップで、各プロセスは自身が所有しているすべての点で解格子関数の新しい値を計算する。次に、プロセス間の境界の値を近傍プロセスと交換しなければならない。例えばサブルーチン `exchange` が、更新された値を左側近傍 (`i-1,j`) に送るために MPI_SEND(...,neigh_rank(1),...) のような呼び出しを行うだろう。

```
1
2         integer ndims, num_neigh
3         logical reorder
4
5         parameter (ndims=2, num_neigh=4, reorder=.true.)
6         integer comm, comm_cart, dims(ndims), neigh_def(ndims), ierr
7
8         integer neigh_rank(num_neigh), own_position(ndims), i, j
9         logical periods(ndims)
10        real*8 u(0:101,0:101), f(0:101,0:101)
11
12        data dims / ndims * 0 /
13        comm = MPI_COMM_WORLD
14    C     プロセス格子のサイズと周期的か否かを指定する
15
16        call MPI_DIMS_CREATE(comm, ndims, dims,ierr)
17        periods(1) = .TRUE.
18        periods(2) = .TRUE.
19
20    C     グループ WORLD に格子構造を生成し、自身の位置を問い合わせる
21        call MPI_CART_CREATE (comm, ndims, dims, periods, reorder, comm_cart,ierr)
22        call MPI_CART_GET (comm_cart, ndims, dims, periods, own_position,ierr)
23
24    C     近傍プロセスのランクを調べる。ここに自身のプロセス座標は (i,j) であり、
25    C     近傍のそれは (i-1,j), (i+1,j), (i,j-1), (i,j+1) となる。
26
27        i = own_position(1)
28        j = own_position(2)
29        neigh_def(1) = i-1
30
31        neigh_def(2) = j
32        call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(1),ierr)
33
34        neigh_def(1) = i+1
35        neigh_def(2) = j
36        call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(2),ierr)
37
38        neigh_def(1) = i
39        neigh_def(2) = j-1
40        call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(3),ierr)
41
42        neigh_def(1) = i
43        neigh_def(2) = j+1
44        call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(4),ierr)
45
46    C     格子関数を初期化し、反復を開始する。
47        call init (u, f)
48
          do 10 it=1,100
            call relax (u, f)
    C     近傍プロセスと値を交換する
            call exchange (u, comm_cart, neigh_rank, num_neigh)
    10    continue
          call output (u)
          end
```

Chapter7

MPI 環境管理

本章では、 MPI の実装と実行環境 (エラー処理など) に関連する各種パラメータの取得、および該当する場合にはその設定を行うルーチンについて論じる。 MPI 実行環境へ出入りするための手続きについてもここで説明する。

7.1 実装情報

7.1.1 環境の問い合わせ

MPI が初期化される時、実行環境を記述する属性の集合がコミュニケータ MPI_COMM_WORLD に付加される。これらの属性の値を問い合わせるには、第 5章で説明した関数 MPI_ATTR_GET を使用する。これらの属性の削除、キーの解放、および値の変更をしてはいけない。

定義済み属性キーとしては次のものがある。

MPI_TAG_UB タグ値の上限

MPI_HOST ホスト・プロセスが存在していればそのランク、そうでなければ MPI_PROC_NULL

MPI_IO 正規の I/O 機能を持つノードのランク (場合によっては myrank)。同じコミュニケータでもノード毎に異なる値を返す場合がある。

MPI_WTIME_IS_GLOBAL クロックを同期させるかどうかを示す論理型変数。

ベンダは実装依存のパラメータ (ノード番号、実メモリ・サイズ、仮想メモリ・サイズ、など) を追加してよい。

これらの定義済み属性の値は MPI 初期化 (MPI_INIT) から MPI 完了 (MPI_FINALIZE) まで変化せず、またユーザーが更新したり削除したりすることはできない。

ユーザへのアドバイス　C 言語の呼び出し形式では、これらの属性の返す値は要求された値を格納する変数 (整数型) へのポインタである。 (ユーザへのアドバイスの終わり)

必要なパラメータの値については以下で詳しく論じる。

タグ値

タグ値は 0 から MPI_TAG_UB の返す値 (MPI_TAG_UB を含む) までの範囲の値である。これらの値は MPI プログラムの実行中に変化しないことが保証されている。さらに、タグ上限値は 32767 以上でなければならない。 MPI の実装では、 MPI_TAG_UB の値はこの値よりも大きければよい。例えば、値 $2^{30} - 1$ は MPI_TAG_UB の妥当な値でもある。

属性 MPI_TAG_UB は MPI_COMM_WORLD に属す全てのプロセスで同じ値を持つ。

ホスト・ランク

MPI_HOST の返す値は、コミュニケータ MPI_COMM_WORLD と関連するグループ中の `HOST` プロセス (もしあれば) のランクである。ホストがなければ MPI_PROC_NULL が返される。 MPI では、プロセスが `HOST` であるということの意味を指定しないし、また `HOST` の存在さえ要求しない。

属性 MPI_HOST は MPI_COMM_WORLD に属す全てのプロセスで同じ値を持つ。

IO ランク

MPI_IO の返す値は、言語標準 I/O 機能を持っているプロセッサのランクである。 Fortran 言語では、 Fortran 言語 I/O 制御のすべてがサポートされていることを意味する (例えば、 `OPEN`、`REWIND`、 `WRITE`)。 C 言語では、 ANSI-C I/O 制御のすべてがサポートされていることを意味する (例えば、 `fopen`、 `fprintf`、 `lseek`)。

すべてのプロセスが言語標準 I/O 機能を持っていれば、値 MPI_ANY_SOURCE が返される。そうでない場合、呼び出しプロセスが言語標準 I/O 機能を持っていれば、自分自身のランクが返される。そうでない場合には、言語標準 I/O 機能を提供できるプロセスがあれば、そのようなプロセスのうち 1 つのプロセスのランクが返される。すべてのプロセスが同じ値を返す必要はない。どのプロセスも言語標準 I/O 機能を提供できない場合には、値 MPI_PROC_NULL が返される。

> ユーザへのアドバイス 入力は集団的ではなく、また、この属性はどのプロセスが入力を提供できるか、あるいは入力を提供しているのかを「示してはいない」ことに注意せよ。
> (ユーザへのアドバイスの終わり)

クロック同期

MPI_WTIME_IS_GLOBAL の返す値は、 MPI_COMM_WORLD に属す全てのプロセスのクロックが同期していれば 1 であり、そうでなければ 0 である。同期させるための明示的な操作が行われている場合に、クロックが同期していると考える。 MPI_WTIME の呼出しで測定される時間の変化

は長さ 0 の MPI メッセージが往復するのに要する時間の半分未満となることが期待される。あるプロセスの送信直前と、別のプロセスの対応する受信直後に時間を測定した場合、受信側での時間は送信側での時間よりも常に後の時間を示さなければならない。

属性 MPI_WTIME_IS_GLOBAL は、クロックが同期していない場合には存在する必要はない (ただし、属性キー MPI_WTIME_IS_GLOBAL は常に有効である)。この属性は MPI_COMM_WORLD 以外のコミュニケータに付加してもよい。

属性 MPI_WTIME_IS_GLOBAL は MPI_COMM_WORLD に属す全てのプロセスで同じ値を持つ。

MPI_GET_PROCESSOR_NAME(name, resultlen)

出力	name	(仮想ノードではなく) 実ノードに対する唯一の識別子
出力	resultlen	name に返される結果の (印字可能文字の) 長さ

```
int MPI_Get_processor_name(char *name, int *resultlen)
```

```
MPI_GET_PROCESSOR_NAME( NAME, RESULTLEN, IERROR)
    CHARACTER*(*) NAME
    INTEGER RESULTLEN,IERROR
```

このルーチンは、ルーチンを呼び出したプロセッサの名前を返す。この名前は長さに制限のない文字列である。この値から、ハードウェアの特定の一部分が識別できなければならない; 可能な値としては、「mpp.cs.org の 4 番目のラックの 9 番目のプロセッサ」や「231」 (231 は稼働中の同機種システムの実際のプロセッサ番号である) がある。引数 name は、文字列長 MPI_MAX_PROCESSOR_ 以上の記憶域でなければならない。 MPI_GET_PROCESSOR_NAME はこの文字数まで name の中に書き込める。

実際に書き込まれた文字数は出力引数 resultlen に返される。

根拠 この関数を使用することで、現在のプロセッサへ復帰するようにプロセスマイグレーションを行うことが可能である。プロセスマイグレーションの必要性や定義は MPI に含まれないことに注意せよ; MPI_GET_PROCESSOR_NAME のこの定義により、そのような実装を許すだけである。(根拠の終わり)

ユーザへのアドバイス ユーザーは少なくとも MPI_MAX_PROCESSOR_NAME の大きさの記憶域をプロセッサ名を書き込むために用意しなければならない－プロセッサ名はこの長さまで許される。ユーザーは名前の実際の長さを調べるために、出力引数 resultlen を調べるべきである。(ユーザへのアドバイスの終わり)

定数 MPI_BSEND_OVERHEAD は、 MPI_BSEND の呼出しによりバッファリングされたメッセージごとの固定されたオーバーヘッドに上限を定める。 (第 3.6.1 章を参照せよ)

7.2 エラー処理

MPI の実装では、MPI 呼出し中に発生するいくつかのエラーを処理できないか、または処理しないようにするかもしれない。このようなエラーとしては、浮動小数点エラーやアクセス違反など、例外やトラップを発生するエラーなどがある。MPI で処理されるエラーは実装依存である。このようなエラーにより **MPI** 例外が発生する。

上記の文は、本書の中のエラー処理に関する記述に優先する。特に、エラー処理されるという記述は、エラー処理されるかもしれないと読み替えなければならない。

ユーザーはエラー・ハンドラをコミュニケータに付加することができる。このコミュニケータとの通信のための MPI 呼出し中に起こったどのような MPI 例外に対してもその指定されたエラー処理ルーチンが使用される。どのコミュニケータとも関連しない MPI 呼出しはコミュニケータ MPI_COMM_WORLD に付加されていると考える。コミュニケータにエラー・ハンドラを付加することは純粋にローカルである: 異なるプロセスでは異なるエラー・ハンドラを同じコミュニケータに付加してもよい。

新規に生成されたコミュニケータは「親」コミュニケータに対応するエラー・ハンドラを継承する。特に、ユーザーは初期化直後にコミュニケータ MPI_COMM_WORLD にハンドラを付加することにより、すべてのコミュニケータについて「大域的な」エラー・ハンドラを指定することができる。

MPI ではいくつかの定義済みエラー・ハンドラが利用できる。

MPI_ERRORS_ARE_FATAL このハンドラは、呼び出されると、実行中のすべてのプロセスを終了させる。これはこのハンドラを呼び出したプロセスで MPI_ABORT が呼ばれたのと同じ効果を持つ。

MPI_ERRORS_RETURN このハンドラはエラー・コードをユーザーへ返すだけである。

MPI の実装では、さらに定義済みエラー・ハンドラを提供してもよいし、またプログラマが独自のエラー・ハンドラをコーディングすることもできる。

デフォルトでは初期化後にエラー・ハンドラ MPI_ERRORS_ARE_FATAL が、MPI_COMM_WORLD に付加されている。したがって、ユーザーがエラー処理を制御しないようにしている場合、MPI が処理するエラーはすべて致命的エラーとして取り扱われる。（ほとんど）すべての MPI 呼出しはエラー・コードを返すので、ユーザーは MPI 呼出しの返却コードを調べ、呼び出しが失敗であれば適切な復旧コードを実行し、エラーを処理することができる。この場合、エラー・ハンドラ MPI_ERRORS_RETURN が使用される。通常は、MPI 呼出し毎にエラーのテストはせずに、そのようなエラーを自明でない MPI エラー・ハンドラで処理したほうが都合が良いし、また効率もよい。

エラーが検出された後、MPI の状態は未定義である。つまり、ユーザー定義エラー・ハン

ドラ、または MPI_ERRORS_RETURN を使用しても、必ずしも、エラー検出後にユーザーが MPI を使用し続けられるとはかぎらない。これらのエラー・ハンドラの目的は、プログラムが終了する前に、ユーザーがユーザー定義エラー・メッセージを発行し、 MPI に関連しない処置 (I/O バッファのフラッシュなど) を実行することである。エラーの後 MPI が続行できるように実装してもよいが、そうする必要はない。

実装者へのアドバイス　高品質な実装では、可能な限り最大限、エラーの影響を抑え、エラー・ハンドラが呼び出された後も通常処理が続行できるようにすることが望まれる。その実装解説書にはエラーの各クラスの起こりうる影響について情報を提供することが望まれる。（実装者へのアドバイスの終わり）

MPI エラー・ハンドラはハンドルによってアクセスされる不透明なオブジェクトでる。新しいエラー・ハンドラを作成するための MPI 呼出し、コニュニケータにエラー・ハンドラを付加するための MPI 呼出し、および、どのエラーハンドラがコミュニケータに付加されているかをテストするための MPI 呼出しが用意されている。

MPI_ERRHANDLER_CREATE(function, errhandler)

入力	function	ユーザー定義のエラー処理手続き
出力	errhandler	MPI エラー・ハンドラ (ハンドル)

```
int MPI_Errhandler_create(MPI_Handler_function *function,
              MPI_Errhandler *errhandler)
```

```
MPI_ERRHANDLER_CREATE(FUNCTION, HANDLER, IERROR)
    EXTERNAL FUNCTION
    INTEGER ERRHANDLER, IERROR
```

このルーチンは、 MPI 例外処理ハンドラとして使うためにユーザー・ルーチン function を登録する。その登録された例外ハンドラへのハンドルを errhandler に返す。

C 言語では、ユーザー・ルーチンは MPI_Handler_function 型の C 関数でなければならない。これは次のように定義される:

```
typedef void (MPI_Handler_function)(MPI_Comm *, int *, ...);
```

最初の引数は、使用するコミュニケータである。二番目の引数は、エラーを起こした MPI ルーチンが返すエラー・コードである。このルーチンの返却値が MPI_ERR_IN_STATUS である場合、これはエラー・ハンドラが呼び出される原因となった要求が生じたステータス中で返されるエ

ラー・コードである。残りの引数は「stdargs」引数であり、その数と意味は実装依存である。実装の際はこれらの引数について明確に文書化しなければならなず、 Fortran 言語でハンドラを書けるようにアドレスを使用する。

根拠 可変引数リストを用いたのは、エラー・ハンドラに付加情報を提供するための ANSI 規格のフックを利用するためである; ANSI C ではこのフックを使用しないで、引数を追加することは禁止されている。（根拠の終わり）

MPI_ERRHANDLER_SET(comm, errhandler)

入力	comm	エラー・ハンドラを付加するコミュニケータ (ハンドル)
入力	errhandler	コミュニケータに対する新しい MPI エラーハンドラ (ハンドル)

```
int MPI_Errhandler_set(MPI_Comm comm, MPI_Errhandler errhandler)

MPI_ERRHANDLER_SET(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERR HANDLER, IERROR
```

このルーチンは、呼び出されると、新しいエラー・ハンドラ errorhandler をコミュニケータ comm に付加する。エラー・ハンドラは常にコミュニケータへ付加されているに注意せよ。

MPI_ERRHANDLER_GET(comm, errhandler)

入力	comm	エラー・ハンドラを取り出すコミュニケータ (ハンドル)
出力	errhandler	コミュニケータに現在付加されている MPI エラー・ハンドラ (ハンドル)

```
int MPI_Errhandler_get(MPI_Comm comm, MPI_Errhandler *errhandler)

MPI_ERRHANDLER_GET(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERR HANDLER, IERROR
```

このルーチンは現在コミュニケータ comm へ付加されているエラー・ハンドラ (のハンドル) を errhandler に返す。

例 **7.1** ライブラリ関数はそのエントリポイントで、コミュニケータに現在付加されているエラー・ハンドラを登録しておき、このコミュニケータについてライブラリ関数内でのみ有効なエラー・ハンドラを付加し、ライブラリ終了時にもとのハンドラに戻すことができる。

MPI_ERRHANDLER_FREE(errhandler)

入力	errhandler	MPI エラー・ハンドラ (ハンドル)

```
int MPI_Errhandler_free(MPI_Errhandler *errhandler)

MPI_ERRHANDLER_FREE(ERRHANDLER, IERROR)
    INTEGER ERRHANDLER, IERROR
```

このルーチンは errhandler に関連したエラー・ハンドラを解放のためにマークし、 errhandler を MPI_ERRHANDLER_NULL に設定する。このエラー・ハンドラは、これを付加されたすべてのコミュニケータの解放後に解放される。

MPI_ERROR_STRING(errorcode, string, resultlen)

入力	errorcode	MPI ルーチンによって返されるエラー・コード
出力	string	errorcode に対応する文
出力	resultlen	string に返される結果の (印字可能文字の) 長さ

```
int MPI_Error_string(int errorcode, char *string, int *resultlen)

MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR)
    INTEGER ERRORCODE, RESULTLEN, IERROR
    CHARACTER*(*) STRING
```

このルーチンはエラー・コードやエラー・クラスと関連するエラー文字列を返す。引数 string は文字列長 MPI_MAX_ERROR_STRING 以上の記憶域でなければならない。

実際に書き込まれた文字数は、引数 resultlen に返される。

根拠 この関数の形式は、 Fortran 言語と C 言語の呼び出し形式が同様なものとなるように決められている。文字列へのポインタを返す場合は難しい点が 2 点ある。第 1 に、返却文字列領域は静的に割り付けられていなければならず、 (連続する MPI_ERROR_STRING への呼び出しによって返されるポインタが正しいメッセージを指すようにするには) 各エラー・メッセージごとに領域が異なっていなければならないということである。第 2 に、 Fortran では、 CHARACTER*(*) を返すように宣言された関数は、例えば PRINT 文などの文中で参照することができないということである。 (根拠の終わり)

7.3 エラー・コードおよびクラス

MPI で返されるエラー・コードは、(MPI_SUCCESS を除いて) 完全に実装に依存する。これは、実装時に (MPI_ERROR_STRING で使われる) エラー・コードにできるだけ多くの情報を入れることができるようとするためである。

アプリケーションがエラー・コードを解釈出来るように、ルーチン MPI_ERROR_CLASS は、エラーコードをエラー・クラスと呼ばれる標準エラーコードの小さな集合のうちの 1 つに変換する。有効なエラー・クラスには次のものがある。

MPI_SUCCESS	エラーなし
MPI_ERR_BUFFER	無効なバッファ・ポインタ
MPI_ERR_COUNT	無効なカウント引数
MPI_ERR_TYPE	無効なデータタイプ引数
MPI_ERR_TAG	無効なタグ引数
MPI_ERR_COMM	無効なコミュニケータ
MPI_ERR_RANK	無効なランク
MPI_ERR_REQUEST	無効な要求 (ハンドル)
MPI_ERR_ROOT	無効なルート
MPI_ERR_GROUP	無効なグループ
MPI_ERR_OP	無効な操作
MPI_ERR_TOPOLOGY	無効なトポロジー
MPI_ERR_DIMS	無効な次元引数
MPI_ERR_ARG	その他の無効な引数
MPI_ERR_UNKNOWN	未知のエラー
MPI_ERR_TRUNCATE	受信時にメッセージが切り捨てられた
MPI_ERR_OTHER	このリストのなかに載っていない既知のエラー
MPI_ERR_INTERN	MPI 内部 (実装) エラー
MPI_ERR_IN_STATUS	ステータス中のエラー・コード
MPI_ERR_PENDING	保留要求
MPI_ERR_LASTCODE	最後のエラー・コード

エラー・クラスはエラー・コードの部分集合である: MPI 関数はエラー・クラス番号を返してもよい; また、関数 MPI_ERROR_STRING を使用してエラー・クラスと関連するエラー文字列を取り出すことができる。

エラー・クラスは次の不等式を満たす。

$$0 = \text{MPI\_SUCCESS} < \text{MPI\_ERR\_...} \leq \text{MPI\_ERR\_LASTCODE}.$$

根拠 MPI_ERR_UNKNOWN と MPI_ERR_OTHER との違いは、MPI_ERR_OTHER に対しては MPI_ERROR_STRING が有用な情報を返すことができるという点にある。

C 言語での慣例と一致させるために MPI_SUCCESS = 0 が必要であることに注意せよ; エラー・クラスとエラー・コードを分けることで、上のようにエラー・クラスを定義することができる。既知の LASTCODE があることは、多くの場合エラー・クラスの値の正確な判断基準になる。(根拠の終わり)

MPI_ERROR_CLASS(errorcode, errorclass)

入力	errorcode	MPI ルーチンによって返されるエラー・コード
出力	errorclass	errorcode に関連するエラー・クラス

```
int MPI_Error_class(int errorcode, int *errorclass)
```

```
MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR)
    INTEGER ERRORCODE, ERRORCLASS, IERROR
```

MPI_ERROR_CLASS 関数は標準エラー・コード (エラー・クラス) をエラー・クラスに変換する。

7.4 時刻関数と同期

MPI は時刻関数を定義する。時刻関数そのものは「メッセージ通信」の範疇ではないが, MPI ではこれを規格化する。なぜなら、「性能デバッギング」では並列プログラムの時間を計ることが重要であり、かつ (POSIX 1003.1-1988 と 1003.4D14.1 や Fortran 90 中の) 既存の時刻関数は不便であったり、高精度な時刻へ適切にアクセスできる関数として提供されてないからである。

MPI_WTIME()

```
double MPI_Wtime(void)
```

```
DOUBLE PRECISION MPI_WTIME()
```

MPI_WTIME は、ある過去の時刻からの経過時間を表す秒数を浮動小数点数で返す。

前文の「過去の時刻」は、プロセスの存続期間中変更されないことが保証されている。ユーザーは、大きな秒数を他の単位に変換したい場合には自分で変換を行わなければならない。

この関数は、移植性があり (「時計の刻み数」ではなく秒数を返す)、高精度も可能であり、不要な処理も必要ない。次のように使用する。

```
{
    double starttime, endtime;
    starttime = double MPI_Wtime();
      ....  時間が計られる部分  ...
    endtime   = double MPI_Wtime();
    printf("That took %f seconds\n" ,endtime-starttime);
}
```

返される時刻はそれを呼び出したノードにローカルなものである。異なるノードが「同じ時刻」を返す必要はない (MPI_WTIME_IS_GLOBAL の議論も参照のこと)。

MPI_WTICK()

```
double MPI_Wtick(void)
```

```
DOUBLE PRECISION MPI_WTICK()
```

MPI_WTICK は、 MPI_WTIME の精度を秒数で返す。すなわち、時計の刻み幅の秒数を倍精度値として返す。例えば、時計が 1 ミリ秒ごとに増加するカウンターとしてハードウェアにより実装されている場合、 MPI_WTICK が返す値は 10^{-3} でなければならない。

7.5 起動

MPI の一つの目標はソース・コードの移植性を実現することである。これは、 MPI を使って適切な言語標準に準拠して作成されたプログラムはそのままで移植でき、あるシステムから別のシステムへ移したときにソース・コードの変更を必要としてはいけないということを意味する。このことは、コマンド行から MPI プログラムを起動する方法や MPI プログラムの実行環境をセットアップするためにユーザーがしなければならないことについては何も明確に述べてはいない。しかし、実装では他の MPI ルーチンが呼ばれる前に何らかのセットアップを実行する必要があるかもしれない。この場合に備えて、 MPI には初期化ルーチン MPI_INIT がある。

MPI_INIT()

```
int MPI_Init(int *argc, char ***argv)
```

```
MPI_INIT(IERROR)
    INTEGER IERROR
```

このルーチンは、他の MPI ルーチンの前に 1 回だけ呼び出されなければならない; 2 回目以降の呼び出しは間違いである (MPI_INITIALIZED を参照)。

MPI プログラムはすべて、 MPI_INIT の呼び出しを含まなければならない; このルーチンは、他の (MPI_INITIALIZED を除く)MPI ルーチンを呼び出す前に呼ばなければならない。 ANSI C の場合は、 main 関数の引数によって渡される argc と argv を受け付ける。

```
MPI_init( argc, argv );

int main(argc, argv)
int argc;
char **argv;
{
     MPI_Init(&argc, &argv);

     /* 引数をパージングする部分 */
     /* メイン・プログラム */

     MPI_Finalize();     /* 以下を参照せよ */
}
```

Fortran 言語の場合は IERROR のみを引数とする。

C の呼び出し形式の引数が main 関数への引数でなければならないとすることは実装に依存する。

根拠 コマンド行引数は、 MPI の実装が MPI 環境の初期化に使用するために、 MPI_Init に渡される。コマンド行引数が main に渡されない環境でも MPI の実装にコマンド行引数を提供できるようにそれらは参照渡しである。 (根拠の終わり)

MPI_FINALIZE()

```
int MPI_Finalize(void)

MPI_FINALIZE(IERROR)
    INTEGER IERROR
```

このルーチンはすべての MPI 状態を終了させる。このルーチンがいったん呼び出された後は、どんな MPI ルーチンも (MPI_INIT でさえも) 呼び出すことはできない。

ユーザーは、プロセスが MPI_FINALIZE を呼び出す前にそのプロセスと関連するすべての保留中の通信を完了するようにしなければならない。

MPI_INITIALIZED(flag)

出力	flag	MPI_INIT が呼び出されていたら flag が true、そうでなければ false

```
int MPI_Initialized(int *flag)

MPI_INITIALIZED(FLAG, IERROR)
    LOGICAL FLAG
    INTEGER IERROR
```

このルーチンは、 MPI_INIT が呼び出されたかどうかを調べる場合に使用できる。これは MPI_INIT が呼び出される前に使用できる唯一のルーチンである。

MPI_ABORT(comm, errorcode)

入力	comm	中断するタスクのコミュニケータ
入力	errorcode	呼び出し環境へ返すエラー・コード

```
int MPI_Abort(MPI_Comm comm, int errorcode)

MPI_ABORT(COMM, ERRORCODE, IERROR)
    INTEGER COMM, ERRORCODE, IERROR
```

このルーチンは、コニュニケータ comm のグループ中のすべてのタスクを「できるかぎり」中断しようとする。この関数は呼び出し環境でエラー・コードに応じた処理を要求しない。しかし、 Unix や POSIX 環境では、これをメイン・プログラムにおける `return errorcode` や `abort(errorcode)` として処理すべきである。 MPI の実装では、 MPI_ABORT の挙動を少なくとも MPI_COMM_WORLD の comm については定義する必要がある。 MPI の実装では、 comm 引数を無視し、 comm が MPI_COMM_WORLD であるかのように扱ってよい。

Chapter8

プロファイリング・インターフェース

8.1 要求仕様

MPI プロファイリング・インターフェースの条件を満たすために、 MPI 関数の実装は次のとおりでなければならない。

1. MPI 定義関数すべてを名前シフトでアクセスするためのメカニズムを用意する。これにより (通常はプレフィックス「MPI_」で始まる) MPI 関数はすべてプレフィックス「PMPI_」でもアクセスできなければいけない。

2. 置き換えない MPI 関数があっても、名前衝突を引き起こさずに実行可能イメージにリンクできることを保証する。

3. 異なる言語での MPI インターフェースの呼びだし形式が、それぞれの上層に置かれている場合、その実装を文書化するようにして、プロファイラ開発者が各呼び出し形式に対するプロファイルインターフェース実装をしなければいけないのか、低レベルに位置するルーチンのみを実装する事で簡略化できるのか、分かるようにする。

4. 階層化手法で、異なる言語の呼び出し形式を実装する場合 (例えば、 Fortran 言語の呼び出し形式が C 言語の実装を呼び出す「ラッパー」関数の集合である場合)、これらのラッパー関数をライブラリの他の部分から分離可能なようにする。

   これは、分離したプロファイリング・ライブラリを正しく実装するために必要である。なぜなら (少なくとも Unix のリンカにおいては) プロファイリング・ライブラリが期待通りに動作するためにはラッパー関数を含んでいなければならないからである。この要求仕様により、プロファイリング・ライブラリを構築する人は、オリジナルの MPI ライブラリからこれらの関数を抽出したり、それらを他の不要なコードを持ち込むことなくプロファイリング・ライブラリに追加することができる。

5. MPI ライブラリに op なしルーチン MPI_PCONTROL を用意する。

8.2 議論

MPI プロファイリング・インターフェースの目的は、プロファイリング (および他の類似の) ツールの作成者が、異なるマシン上で MPI 実装とのインタフェースを比較的容易に作成できるようにすることである。

MPI には多くの異なる実装があるがマシンに依存しない標準であるので、 MPI 用プロファイリング・ツールの作成者が、ある特定のマシンに実装された MPI のソース・コードを参照できるわけではない。したがって、そのようなツールの実装者が実装の下層部分を参照することなく、期待される性能情報を集められるようにするメカニズムを用意する必要がある。

このようなインターフェースは、 MPI がエンドユーザーにとって魅力的であるためには重要であると思われる。様々なツールが利用できることは、ユーザーが MPI 規格を魅力的に思うようになるための大きな要素だからである。

プロファイリング・インターフェースは、インターフェースにすぎない。使用法については何も述べられていない。したがって、インターフェースを通じてどのような情報を集めるのか、また集められた情報をどのように保存し、フィルタにかけ、あるいは表示するかなどということについては規定していない。

このインターフェースを開発することになった最初の動機はプロファイリング・ツールの実装を可能にしたいということであったが、上で規定したようなインターフェースが他の目的、例えば複数の MPI 実装間を「ネットワークで接続すること」にも有用になる可能性がある。定義したものはすべてインターフェースなので、有用であるならばどこに使われても異論はない。

ここで取り扱っている問題は実行可能イメージを構築する方法と密接に関わっており、それはマシンが異なれば大きく異なる可能性がある。それゆえ、以下に挙げる例は、すべて MPI プロファイリング・インターフェースの目的を実装する手段の一つとして取り扱うべきである。実装に対する実際の要求については、上の「要求仕様」の節で詳述しており、本章の残りの部分ではこれらの要求仕様の論理が正しいことを説明し、議論するのみである。

以下の例では、 Unix システム上で要求を満たす実装を構築する一方法を紹介する (同じように有効なものがほかにもあることは疑いない)。

8.3 設計の論理

MPI の実装が上記の要求仕様を満たしている場合、プロファイリング・システムの実装者はユーザープログラムからの MPI 関数呼び出しをすべて横取りすることができる。そこで、下層の MPI 関数を (名前シフト・エントリ・ポイントを通じて) 呼び出す前に必要な情報を集めて、所

望の結果を得ることができる。

8.3.1 プロファイリングの各種制御

ユーザー・コードからプロファイラを実行時に動的に制御したいという要求がある。これは通常、(少なくとも) 次の目的のためである。

- 計算の状態に応じてプロファイリングを許可、禁止する
- 際どい計算処理を外した時点でトレース・バッファをフラッシュする
- ユーザ・イベントをトレース・ファイルに追加する

これらの要求は、 MPI_PCONTROL を使用することで満たされる。

MPI_PCONTROL(level, ...)

入力 level プロファイルを行なうレベル

```
int MPI_Pcontrol(const int level, ...)

MPI_PCONTROL(level)
    INTEGER LEVEL, ...
```

MPI ライブラリ自体はこのルーチンを使用せず、ユーザ・コードへ即座に戻るだけである。しかし、このルーチンの呼び出しが存在していれば、ユーザ側でプロファイリング・パッケージを明示的に呼び出すことができる。

MPI はプロファイリング・コードの実装についてはなにも規定しないので、 MPI_PCONTROL 呼び出しの意味を正確に指示することはできない。この曖昧さは関数へ渡す引数の個数やそのデータの型にも及んでいる。

しかし、異なるプロファイリング・ライブラリ間でユーザー・コードにある程度の移植性を持たせるために、 level の値のあるものには以下の意味を持たせる。

- level==0 プロファイリングが利用不可
- level==1 通常デフォルトレベルでプロファイリングが利用可能である
- level==2 プロファイル・バッファをフラッシュする。(これはいくつかのプロファイラでは op なしの場合がある)
- 上記以外の値 level の他の値ではすべて、プロファイル・ライブラリで定義される効果を生じ、付加的な引数を持つ。

さらに、MPI_INIT を呼び出した後のデフォルトの状態が、通常のデフォルトレベルでプロファイリングを利用できるということも必要となる。(つまり、MPI_PCONTROL が引数 1 で呼び出されたのと同様である)。これにより、ユーザはソース・コードをまったく修正する必要なしにプロファイリング・ライブラリとリンクし、プロファイル出力を得ることができる。

標準 MPI ライブラリでは op なしの MPI_PCONTROL を規定しているため、そのソース・コードを修正してより詳細なプロファイリング情報を取得するようにできるが、それでも全く同じコードを標準 MPI ライブラリに対してリンクすることができる。

8.4 例

8.4.1 プロファイラの実装

プロファイラを使って MPI_SEND 関数が送信したデータの総量と、この関数が消費した総時間経過の累計をとりたいと仮定する。これは次のようにして得られることは明らかである。

```
static int totalBytes;
static double totalTime;

int MPI_SEND(void * buffer, const int count, MPI_Datatype datatype,
             int dest, int tag, MPI_comm comm)
{
   double tstart = MPI_Wtime();         /* すべての引数を引き渡す */
   int extent;
   int result    = PMPI_Send(buffer,count,datatype,dest,tag,comm);

   MPI_Type_size(datatype, &extent);    /* 送信したデータのバイト数を加える */
   totalBytes += count * extent;

   totalTime  += MPI_Wtime() - tstart; /* 時間経過を加える */

   rturn result;
}
```

8.4.2 MPI ライブラリの実装

Unix システムで MPI ライブラリが C 言語で実装されているものは、いろいろなオプションがあり、そこでもっとも明白なもののうち二つをここに紹介する。どちらがよいかはリンカとコン

パイラが weak シンボルをサポートしているかどうかによる。

weak シンボルのあるシステム

コンパイラとリンカが weak 外部シンボルをサポートしている場合（例えば、Solaris 2.x、その他の system V.4 マシン）、次のように #pragma weak を使用することで一つのライブラリしか必要としない。

```
#pragma weak MPI_Example = PMPI_Example

int PMPI_Example(/* 適当な引数 */)
{
    /* 有益なコード */
}
```

この #pragma の効果として、外部シンボル MPI_Example が weak 定義として定義される。このことは、リンカでは（例えば、プロファイリング・ライブラリで）シンボルの定義がほかにあってもエラーにならず、他の定義が存在しない場合には、リンカは weak 定義を使用するということを意味する。

weak シンボルのないシステム

weak シンボルが存在しない場合には、解決策の一つとして、次のように C のマクロ・プリプロセッサを使用するという方法があるだろう。

```
#ifdef PROFILELIB
#    ifdef __STDC__
#        define FUNCTION(name) P##name
#    else
#        define FUNCTION(name) P/**/name
#    endif
#else
#    define FUNCTION(name) name
#endif
```

この時、ライブラリ内のユーザーに見える関数はそれぞれ、次のように宣言し得るだろう。

```
int FUNCTION(MPI_Example)(/* 適当な引数 */)
{
```

```
    /* 有益なコード */
}
```

PROFILELIB マクロ・シンボルの状態に従って、両方のバージョンのライブラリが同じソース・ファイルをコンパイルして作成することができる。

MPI 関数が一度に一つしか含まれないように標準 MPI ライブラリを構築する必要がある。これは、それぞれの外部関数を別々のファイルからコンパイルしなければならないため、いくぶん面白くない要求である。しかし、プロファイリング・ライブラリの作成者が、横取りしたい MPI ライブラリとそれ以外の標準 MPI ライブラリで済ませられる関数への参照さえ定義すればよいようにするために必要である。そのため、リンク段階ではこのようになる。

```
% cc ... -lmyprof -lpmpi -lmpi
```

ここで、libmyprof.a にはいくつかの MPI 関数を横取りするプロファイラ関数が含まれている。libpmpi.a には、「名前シフト」した MPI 関数が入っており、libmpi.a には MPI 関数の標準の定義が入っている。

8.4.3 厄介な問題

多重カウント

MPI ライブラリの一部は、それ自身より基本的な MPI 関数を使用して実装できるので (例えば、1 対 1 通信を使用して実装された集団通信の移植性のある実装)、プロファイリング関数から呼ばれた MPI 関数の中からプロファイリング関数を呼び出す可能性がある。このようなことがあると、内側のルーチンで費やされる時間を「二重カウント」することになる可能性もある。この効果は状況によっては実際に有用な場合もあるので (例えば、「集団関数から呼び出されたときに 1 対 1 ルーチンでどれだけの時間が費やされるか」という疑問に答えることもできる)、MPI ライブラリの作成者にこれを抑制するという制約を強制しないようにした。したがって、プロファイリング・ライブラリの作成者はこの問題に注意し、自身でこれを防ぐようにしなければならない。単一スレッドの世界では、すでにプロファイリング・ルーチンに入っているかどうかを示すスタティック変数を使用して簡単に実現できる。複数スレッド環境では、これはより複雑なものになる (記録された回数の意味を考えるとよい)。

リンカの奇妙なところ

Unix のリンカは伝統的に 1 パスで実行する。この結果、ライブラリ内を走査している時に必要であれば、ライブラリ中の関数がイメージの中に取り込まれるのみである。weak シンボルや同じ関数の複数の定義があった場合、これは奇妙な (そして期待していない) 結果を生じる場合がある。

例えば、C 言語用の実装の上にラッパー関数をかぶせて実現した Fortran 言語の呼び出し形式の MPI 関数を考える。プロファイル・ライブラリの作成者は、C 言語の呼び出し形式のプロファイル関数を用意するだけで十分と仮定する。それは、Fortran 言語は最終的にこれらの関数を呼び出し、またラッパー関数のコストは微々たるものと仮定しているからである。しかし、ラッパー関数がプロファイリング・ライブラリの中にない場合でも、プロファイリング・ライブラリを呼び出したときにプロファイル用のエントリ・ポイントはどれも未定義にはならない。したがって、プロファイリング・コードはどれもイメージの中には取り込まれない。標準 MPI ライブラリを走査するときに、Fortran 言語のラッパー関数は解決され、MPI 関数の基本バージョンの中の関数が利用される。全体として、コードは正常にリンクされるが、プロファイルされないという結果になる。

この問題を解決するために、Fortran 言語用ラッパー関数をプロファイル用ライブラリの中に取り込むようにしなければならない。これは、ラッパー関数を基本 MPI ライブラリの残り部分から分離できるように要求することで実現できる。こうすることで、`ar` を使用して基本ライブラリから抽出し、プロファイリング・ライブラリの中に取り込むことができる。

8.5 複数レベルの横取り

ここで述べた方法は、各 MPI 関数について単一の代替名しか用意していないため、プロファイリング関数の入れ子を直接にはサポートしていない。複数レベルの呼び出し横取りを可能にする実装が検討されたが、以下の欠点を持たない実装を構築することはできなかった。

- 特定の実装言語を仮定する
- プロファイリングを行わないときでも実行時コストがかかる

MPI の目的の一つは効率がよく待ち時間の短い実装を可能にすることであり、特定の実装言語を要求することは標準の役目でないことから、上述の方式を受け入れることに決定した。

しかし、ユーザが呼び出す関数は MPI の下層の関数を呼び出す前に様々な異なったプロファイリング用関数を呼び出すので、上記の方式を使用して複数レベル・システムを実装することも可能である。

残念なことに、このような実装では、上述の単一レベル実装で必要とする以上に、異なるプロファイリング・ライブラリ同士で密接に協調することが求められる。

参考文献

[1] V. Bala and S. Kipnis. Process groups: a mechanism for the coordination of and communication among processes in the Venus collective communication library. Technical report, IBM T. J. Watson Research Center, October 1992. Preprint.

[2] V. Bala, S. Kipnis, L. Rudolph, and Marc Snir. Designing efficient, scalable, and portable collective communication libraries. Technical report, IBM T. J. Watson Research Center, October 1992. Preprint.

[3] Purushotham V. Bangalore, Nathan E. Doss, and Anthony Skjellum. MPI++: Issues and Features. In *OON-SKI '94*, page in press, 1994.

[4] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Visualization and debugging in a heterogeneous environment. *IEEE Computer*, 26(6):88–95, June 1993.

[5] Luc Bomans and Rolf Hempel. The Argonne/GMD macros in FORTRAN for portable parallel programming and their implementation on the Intel iPSC/2. *Parallel Computing*, 15:119–132, 1990.

[6] R. Butler and E. Lusk. User's guide to the p4 programming system. Technical Report TM-ANL–92/17, Argonne National Laboratory, 1992.

[7] Ralph Butler and Ewing Lusk. Monitors, messages, and clusters: the p4 parallel programming system. *Journal of Parallel Computing*, 1994. to appear (Also Argonne National Laboratory Mathematics and Computer Science Division preprint P362-0493).

[8] Robin Calkin, Rolf Hempel, Hans-Christian Hoppe, and Peter Wypior. Portable programming with the parmacs message–passing library. *Parallel Computing, Special issue on message–passing interfaces*, to appear.

[9] S. Chittor and R. J. Enbody. Performance evaluation of mesh–connected wormhole–routed networks for interprocessor communication in multicomputers. In *Proceedings of the 1990 Supercomputing Conference*, pages 647–656, 1990.

[10] S. Chittor and R. J. Enbody. Predicting the effect of mapping on the communication performance of large multicomputers. In *Proceedings of the 1991 International Conference on Parallel Processing, vol. II (Software)*, pages II–1 – II–4, 1991.

[11] J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Integrated PVM framework supports heterogeneous network computing. *Computers in Physics*, 7(2):166–75, April 1993.

[12] J. J. Dongarra, R. Hempel, A. J. G. Hey, and D. W. Walker. A proposal for a user-level, message passing interface in a distributed memory environment. Technical Report TM-12231, Oak Ridge National Laboratory, February 1993.

[13] Nathan Doss, William Gropp, Ewing Lusk, and Anthony Skjellum. A model implementation of MPI. Technical report, Argonne National Laboratory, 1993.

[14] Edinburgh Parallel Computing Centre, University of Edinburgh. *CHIMP Concepts*, June 1991.

[15] Edinburgh Parallel Computing Centre, University of Edinburgh. *CHIMP Version 1.0 Interface*, May 1992.

[16] D. Feitelson. Communicators: Object-based multiparty interactions for parallel programming. Technical Report 91-12, Dept. Computer Science, The Hebrew University of Jerusalem, November 1991.

[17] Hubertus Franke, Peter Hochschild, Pratap Pattnaik, and Marc Snir. An efficient implementation of MPI. In *1994 International Conference on Parallel Processing*, 1994.

[18] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A user's guide to PICL: a portable instrumented communication library. Technical Report TM-11616, Oak Ridge National Laboratory, October 1990.

[19] William D. Gropp and Barry Smith. Chameleon parallel programming tools users manual. Technical Report ANL-93/23, Argonne National Laboratory, March 1993.

[20] O. Krämer and H. Mühlenbein. Mapping strategies in message–based multiprocessor systems. *Parallel Computing*, 9:213–225, 1989.

[21] nCUBE Corporation. *nCUBE 2 Programmers Guide, r2.0*, December 1990.

[22] Parasoft Corporation, Pasadena, CA. *Express User's Guide*, version 3.2.5 edition, 1992.

[23] Paul Pierce. The NX/2 operating system. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 384–390. ACM Press, 1988.

[24] A. Skjellum and A. Leung. Zipcode: a portable multicomputer communication library atop the reactive kernel. In D. W. Walker and Q. F. Stout, editors, *Proceedings of the Fifth Distributed Memory Concurrent Computing Conference*, pages 767–776. IEEE Press, 1990.

[25] A. Skjellum, S. Smith, C. Still, A. Leung, and M. Morari. The Zipcode message passing system. Technical report, Lawrence Livermore National Laboratory, September 1992.

[26] Anthony Skjellum, Nathan E. Doss, and Purushotham V. Bangalore. Writing Libraries in MPI. In Anthony Skjellum and Donna S. Reese, editors, *Proceedings of the Scalable Parallel Libraries Conference*, pages 166–173. IEEE Computer Society Press, October 1993.

[27] Anthony Skjellum, Steven G. Smith, Nathan E. Doss, Alvin P . Leung, and Manfred Morari. The Design and Evolution of Zipcode. *Parallel Computing*, 1994. (Invited Paper, to appear in Special Issue on Message Passing).

[28] Anthony Skjellum, Steven G. Smith, Nathan E. Doss, Charles H. Still, Alvin P. Leung, and Manfred Morari. Zipcode: A Portable Communication Layer for High Performance Multicomputing. Technical Report UCRL-JC-106725 (revised 9/92, 12/93, 4/94), Lawrence Livermore National Laboratory, March 1991. To appear in Concurrency: Practice & Experience.

Annex A

言語からの呼び出し形式

A.1 はじめに

本節では、 Fortran および C 両言語のための特定の呼び出し形式をまとめて示す。最初に C 言語の呼び出し形式を、次に Fortran 言語の呼び出し形式を示す。表示順序は章ごとのアルファベット順である。

A.2 C 言語および Fortran 言語における定義済み定数

以下に示すのは、必須の定義済み定数であり、ファイル `mpi.h`(C 言語の場合) および `mpif.h`(Fortran 言語の場合) の中に定義されるべきものである。

```
/* 返し値 (C 言語 Fortran 言語共通) */
MPI_SUCCESS
MPI_ERR_BUFFER
MPI_ERR_COUNT
MPI_ERR_TYPE
MPI_ERR_TAG
MPI_ERR_COMM
MPI_ERR_RANK
MPI_ERR_REQUEST
MPI_ERR_ROOT
MPI_ERR_GROUP
MPI_ERR_OP
MPI_ERR_TOPOLOGY
MPI_ERR_DIMS
```

```
MPI_ERR_ARG
MPI_ERR_UNKNOWN

MPI_ERR_TRUNCATE
MPI_ERR_OTHER
MPI_ERR_INTERN

MPI_ERR_LASTCODE


/* 各種定数 (C 言語 Fortran 言語共通) */

MPI_BOTTOM
MPI_PROC_NULL

MPI_ANY_SOURCE
MPI_ANY_TAG
MPI_UNDEFINED

MPI_UB
MPI_LB


/* ステータスのサイズと予約された添字値 (Fortran 言語) */
MPI_STATUS_SIZE

MPI_SOURCE
MPI_TAG


/* エラー処理指示手段 (C 言語 Fortran 言語共通 */
MPI_ERRORS_ARE_FATAL
MPI_ERRORS_RETURN


/* 文字列の最大長 */
MPI_MAX_PROCESSOR_NAME

MPI_MAX_ERROR_STRING


/* 要素データ型 (C 言語) */
MPI_CHAR
MPI_SHORT

MPI_INT
MPI_LONG
MPI_UNSIGNED_CHAR

```

```
MPI_UNSIGNED_SHORT
MPI_UNSIGNED
MPI_UNSIGNED_LONG
MPI_FLOAT
MPI_DOUBLE
MPI_LONG_DOUBLE
MPI_BYTE
MPI_PACKED




/* 要素データ型 (Fortran 言語) */
MPI_INTEGER
MPI_REAL
MPI_DOUBLE_PRECISION
MPI_COMPLEX
MPI_DOUBLE_COMPLEX
MPI_LOGICAL
MPI_CHARACTER
MPI_BYTE
MPI_PACKED

/* リダクション関数のためのデータ型 (C 言語) */
MPI_FLOAT_INT
MPI_DOUBLE_INT
MPI_LONG_INT
MPI_2INT
MPI_SHORT_INT
MPI_LONG_DOUBLE_INT

/* リダクション関数のためのデータ型 (Fortran 言語) */
MPI_2REAL
MPI_2DOUBLE_PRECISION
MPI_2INTEGER
MPI_2COMPLEX
```

```
1
2   /* 付加的データ型 (Fortran 言語) */
3
4
5   MPI_INTEGER1
6   MPI_INTEGER2
7
8   MPI_INTEGER4
9   MPI_REAL2
10
11  MPI_REAL4
12  MPI_REAL8
13
14
15  /* 付加的データ型 (C 言語) */
16  MPI_LONG_LONG_INT
17
18
19  /* 予約されたコミュニケータ (C 言語 Fortran 言語共通) */
20  MPI_COMM_WORLD
21  MPI_COMM_SELF
22
23
24  /* コミュニケータおよびグループの比較結果 */
25
26
27  MPI_IDENT
28  MPI_CONGRUENT
29
30  MPI_SIMILAR
31  MPI_UNEQUAL
32
33
34  /* 環境に対する問い合わせのためのキー (C 言語 Fortran 言語共通) */
35  MPI_TAG_UB
36
37  MPI_IO
38  MPI_HOST
39
40
41  /* 集団演算 (C 言語 Fortran 言語共通) */
42  MPI_MAX
43  MPI_MIN
44
45  MPI_SUM
46  MPI_PROD
47  MPI_MAXLOC
48
```

```
MPI_MINLOC
MPI_BAND
MPI_BOR
MPI_BXOR
MPI_LAND
MPI_LOR
MPI_LXOR

/* ヌルハンドル */
MPI_GROUP_NULL
MPI_COMM_NULL
MPI_DATATYPE_NULL
MPI_REQUEST_NULL
MPI_OP_NULL
MPI_ERRHANDLER_NULL

/* 空グループ */
MPI_GROUP_EMPTY

/* トポロジーの種類（C 言語 Fortran 言語共通）*/
MPI_GRAPH
MPI_CART
```

以下に示すのは、C 言語の定義済みデータ型定義であり、やはりファイル mpi.h の中に置かれる。

```
/* 不透明型（C 言語）*/
MPI_Aint
MPI_Status

/* 各種構造体へのハンドル（C 言語）*/
MPI_Group
MPI_Comm
MPI_Datatype
MPI_Request
```

```
MPI_Op

/* ユーザー定義関数のプロトタイプ（C 言語）*/
typedef int  MPI_Copy_function(MPI_Comm *oldcomm, *newcomm, int *keyval,
                    void *extra_state)
typedef int  MPI_Delete_function(MPI_Comm *comm, int *keyval,
                                 void *extra_state)}
typedef void MPI_Handler_function(MPI_Comm *, int *, ...);
typedef void MPI_User_function( void *invec, void *inoutvec, int *len,
                      MPI_Datatype *datatype);
```

Fortran 言語については、ここに、ユーザー定義関数をどのように宣言するべきかの例を示す。

MPI_OP_CREATE のためのユーザー関数引数は以下のように宣言する:

```
FUNCTION USER_FUNCTION( INVEC(*), INOUTVEC(*), LEN, TYPE)
<type> INVEC(LEN), INOUTVEC(LEN)
 INTEGER LEN, TYPE
```

MPI_KEYVAL_CREATE のためのコピー関数引数は以下のように宣言する:

```
FUNCTION COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
                                                ATTRIBUTE_VAL_OUT, FLAG)
 INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT
 LOGICAL FLAG
```

MPI_KEYVAL_CREATE のための削除関数引数は以下のように宣言する:

```
FUNCTION DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE)
 INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE
```

A.3 一対一通信のための C 言語呼び出し形式

当該章での出現順にしたがって示す。

```
int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)

int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm, MPI_Status *status)
```

```
int MPI_Get_count(MPI_Status status, MPI_Datatype datatype, int *count)

int MPI_Bsend(void* buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm)

int MPI_Ssend(void* buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm)

int MPI_Rsend(void* buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm)

int MPI_Buffer_attach( void* buffer, int size)

int MPI_Buffer_detach( void** buffer, int* size)

int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm, MPI_Request *request)

int MPI_Ibsend(void* buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm, MPI_Request *request)

int MPI_Issend(void* buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm, MPI_Request *request)

int MPI_Irsend(void* buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm, MPI_Request *request)

int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source,
              int tag, MPI_Comm comm, MPI_Request *request)

int MPI_Wait(MPI_Request *request, MPI_Status *status)

int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)

int MPI_Request_free(MPI_Request *request)

int MPI_Waitany(int count, MPI_Request *array_of_requests, int *index,
              MPI_Status *status)

int MPI_Testany(int count, MPI_Request *array_of_requests, int *index,
              int *flag, MPI_Status *status)

int MPI_Waitall(int count, MPI_Request *array_of_requests,
              MPI_Status *array_of_statuses)
```

```
int MPI_Testall(int count, MPI_Request *array_of_requests, int *flag,
             MPI_Status *array_of_statuses)

int MPI_Waitsome(int incount, MPI_Request *array_of_requests, int *outcount,
             int *array_of_indices, MPI_Status *array_of_statuses)

int MPI_Testsome(int incount, MPI_Request *array_of_requests, int *outcount,
             int *array_of_indices, MPI_Status *array_of_statuses)

int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,
             MPI_Status *status)

int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)

int MPI_Cancel(MPI_Request *request)

int MPI_Test_cancelled(MPI_Status status, int *flag)

int MPI_Send_init(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm, MPI_Request *request)

int MPI_Bsend_init(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm, MPI_Request *request)

int MPI_Ssend_init(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm, MPI_Request *request)

int MPI_Rsend_init(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm, MPI_Request *request)

int MPI_Recv_init(void* buf, int count, MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm, MPI_Request *request)

int MPI_Start(MPI_Request *request)

int MPI_Startall(int count, MPI_Request *array_of_requests)

int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
             int dest, int sendtag, void *recvbuf, int recvcount,
             MPI_Datatype recvtype, int source, MPI_Datatype recvtag,
             MPI_Comm comm, MPI_Status *status)

int MPI_Sendrecv_replace(void* buf, int count, MPI_Datatype datatype,
             int dest, int sendtag, int source, int recvtag, MPI_Comm comm,
```

```
              MPI_Status *status)

int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
              MPI_Datatype *newtype)

int MPI_Type_vector(int count, int blocklength, int stride,
              MPI_Datatype oldtype, MPI_Datatype *newtype)

int MPI_Type_hvector(int count, int blocklength, MPI_Aint stride,
              MPI_Datatype oldtype, MPI_Datatype *newtype)

int MPI_Type_indexed(int count, int *array_of_blocklengths,
              int *array_of_displacements, MPI_Datatype oldtype,
              MPI_Datatype *newtype)

int MPI_Type_hindexed(int count, int *array_of_blocklengths,
              MPI_Aint *array_of_displacements, MPI_Datatype oldtype,
              MPI_Datatype *newtype)

int MPI_Type_struct(int count, int *array_of_blocklengths,
              MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types,
              MPI_Datatype *newtype)

int MPI_Address(void* location, MPI_Aint *address)

int MPI_Type_extent(MPI_Datatype datatype, int *extent)

int MPI_Type_size(MPI_Datatype datatype, int *size)

int MPI_Type_count(MPI_Datatype datatype, int *count)

int MPI_Type_lb(MPI_Datatype datatype, int* displacement)

int MPI_Type_ub(MPI_Datatype datatype, int* displacement)

int MPI_Type_commit(MPI_Datatype *datatype)

int MPI_Type_free(MPI_Datatype *datatype)

int MPI_Get_elements(MPI_Status status, MPI_Datatype datatype, int *count)

int MPI_Pack(void* inbuf, int incount, MPI_Datatype datatype, void *outbuf,
              int outsize, int *position, MPI_Comm comm)

int MPI_Unpack(void* inbuf, int insize, int *position, void *outbuf,
```

```
              int outcount, MPI_Datatype datatype, MPI_Comm comm)

int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm,
              int *size)
```

A.4 集団通信のための C 言語呼び出し形式

```
int MPI_Barrier(MPI_Comm comm )

int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root,
              MPI_Comm comm )

int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
              void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
              MPI_Comm comm)

int MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,
              void* recvbuf, int *recvcounts, int *displs,
              MPI_Datatype recvtype, int root, MPI_Comm comm)

int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype,
              void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
              MPI_Comm comm)

int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs,
              MPI_Datatype sendtype, void* recvbuf, int recvcount,
              MPI_Datatype recvtype, int root, MPI_Comm comm)

int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
              void* recvbuf, int recvcount, MPI_Datatype recvtype,
              MPI_Comm comm)

int MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,
              void* recvbuf, int *recvcounts, int *displs,
              MPI_Datatype recvtype, MPI_Comm comm)

int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype,
              void* recvbuf, int recvcount, MPI_Datatype recvtype,
              MPI_Comm comm)

int MPI_Alltoallv(void* sendbuf, int *sendcounts, int *sdispls,
              MPI_Datatype sendtype, void* recvbuf, int *recvcounts,
              int *rdispls, MPI_Datatype recvtype, MPI_Comm comm)

int MPI_Reduce(void* sendbuf, void* recvbuf, int count,
              MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

```
int MPI_Op_create(MPI_User_function *function, int commute, MPI_Op *op)

int MPI_Op_free( MPI_Op *op)

int MPI_Allreduce(void* sendbuf, void* recvbuf, int count,
              MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

int MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int *recvcounts,
              MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

int MPI_Scan(void* sendbuf, void* recvbuf, int count,
              MPI_Datatype datatype, MPI_Op op, MPI_Comm comm )
```

A.5 グループ、コンテクスト、コミュニケータのための C 言語呼び出し形式

```
int MPI_Group_size(MPI_Group group, int *size)

int MPI_Group_rank(MPI_Group group, int *rank)

int MPI_Group_translate_ranks (MPI_Group group1, int n, int *ranks1,
              MPI_Group group2, int *ranks2)

int MPI_Group_compare(MPI_Group group1,MPI_Group group2, int *result)

int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)

int MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)

int MPI_Group_intersection(MPI_Group group1, MPI_Group group2,
              MPI_Group *newgroup)

int MPI_Group_difference(MPI_Group group1, MPI_Group group2,
              MPI_Group *newgroup)

int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)

int MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)

int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3],
              MPI_Group *newgroup)

int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3],
```

```
              MPI_Group *newgroup)

int MPI_Group_free(MPI_Group *group)

int MPI_Comm_size(MPI_Comm comm, int *size)

int MPI_Comm_rank(MPI_Comm comm, int *rank)

int MPI_Comm_compare(MPI_Comm comm1, comm2, int *result)

int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)

int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)

int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)

int MPI_Comm_free(MPI_Comm *comm)

int MPI_Comm_test_inter(MPI_Comm comm, int *flag)

int MPI_Comm_remote_size(MPI_Comm comm, int *size)

int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)

int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,
              MPI_Comm peer_comm, int remote_leader, int tag,
              MPI_Comm *newintercomm)

int MPI_Intercomm_merge(MPI_Comm intercomm, int high,
              MPI_Comm *newintracomm)

int MPI_Keyval_create(MPI_Copy_function *copy_fn, MPI_Delete_function
              *delete_fn, int *keyval, void* extra_state)

int MPI_Keyval_free(int *keyval)

int MPI_Attr_put(MPI_Comm comm, int keyval, void* attribute_val)

int MPI_Attr_get(MPI_Comm comm, int keyval, void **attribute_val, int *flag)

int MPI_Attr_delete(MPI_Comm comm, int keyval)
```

A.6 プロセストポロジーのための C 言語呼び出し形式

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods,
              int reorder, MPI_Comm *comm_cart)
```

```
int MPI_Dims_create(int nnodes, int ndims, int *dims)

int MPI_Graph_create(MPI_Comm comm_old, int nnodes, int *index, int *edges,
             int reorder, MPI_Comm *comm_graph)

int MPI_Topo_test(MPI_Comm comm, int *status)

int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)

int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int *index,
             int *edges)

int MPI_Cartdim_get(MPI_Comm comm, int *ndims)

int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int *periods,
             int *coords)

int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)

int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)

int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)

int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors,
             int *neighbors)

int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source,
             int *rank_dest)

int MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm *newcomm)

int MPI_Cart_map(MPI_Comm comm, int ndims, int *dims, int *periods,
             int *newrank)

int MPI_Graph_map(MPI_Comm comm, int nnodes, int *index, int *edges,
             int *newrank)
```

A.7 環境問い合わせのための C 言語呼び出し形式

```
int MPI_Get_processor_name(char *name, int *resultlen)

int MPI_Errhandler_create(MPI_Handler_function *function,
             MPI_Errhandler *errhandler)
```

```
int MPI_Errhandler_set(MPI_Comm comm, MPI_Errhandler errhandler)

int MPI_Errhandler_get(MPI_Comm comm, MPI_Errhandler *errhandler)

int MPI_Errhandler_free(MPI_Errhandler *errhandler)

int MPI_Error_string(int errorcode, char *string, int *resultlen)

int MPI_Error_class(int errorcode, int *errorclass)

int double MPI_Wtime(void)

int double MPI_Wtick(void)

int MPI_Init(int *argc, char ***argv)

int MPI_Finalize(void)

int MPI_Initialized(int *flag)

int MPI_Abort(MPI_Comm comm, int errorcode)
```

A.8 プロファイリングのための C 言語呼び出し形式

```
int MPI_Pcontrol(const int level, ...)
```

A.9 一対一通信のための Fortran 言語呼び出し形式

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR

MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),
    IERROR

MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR

MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
```

```
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR

MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR

MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR

MPI_BUFFER_ATTACH( BUFFER, SIZE, IERROR)
    <type> BUFFER(*)
    INTEGER SIZE, IERROR

MPI_BUFFER_DETACH( BUFFER, SIZE, IERROR)
    <type> BUFFER(*)
    INTEGER SIZE, IERROR

MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI_IBSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR

MPI_WAIT(REQUEST, STATUS, IERROR)
    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
```

```
    LOGICAL FLAG
    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

MPI_REQUEST_FREE(REQUEST, IERROR)
    INTEGER REQUEST, IERROR

MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR)
    INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
    IERROR

MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS, IERROR)
    LOGICAL FLAG
    INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
    IERROR

MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR)
    INTEGER COUNT, ARRAY_OF_REQUESTS(*),
    ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR

MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, FLAG, ARRAY_OF_STATUSES, IERROR)
    LOGICAL FLAG
    INTEGER COUNT, ARRAY_OF_REQUESTS(*),
    ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR

MPI_WAITSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,
             ARRAY_OF_STATUSES, IERROR)
    INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
    ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR

MPI_TESTSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,
             ARRAY_OF_STATUSES, IERROR)
    INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
    ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR

MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)
    LOGICAL FLAG
    INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)
    INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
```

```
MPI_CANCEL(REQUEST, IERROR)
    INTEGER REQUEST, IERROR

MPI_TEST_CANCELLED(STATUS, FLAG, IERROR)
    LOGICAL FLAG
    INTEGER STATUS(MPI_STATUS_SIZE), IERROR

MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI_BSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI_SSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR

MPI_START(REQUEST, IERROR)
    INTEGER REQUEST, IERROR

MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERROR)
    INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERROR

MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF,
             RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE,
    SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,
             COMM, STATUS, IERROR)
```

```
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM,
    STATUS(MPI_STATUS_SIZE), IERROR

MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR

MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR

MPI_TYPE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR

MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
                 OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
    OLDTYPE, NEWTYPE, IERROR

MPI_TYPE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
                 OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
    OLDTYPE, NEWTYPE, IERROR

MPI_TYPE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
                 ARRAY_OF_TYPES, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
    ARRAY_OF_TYPES(*), NEWTYPE, IERROR

MPI_ADDRESS(LOCATION, ADDRESS, IERROR)
    <type> LOCATION(*)
    INTEGER ADDRESS, IERROR

MPI_TYPE_EXTENT(DATATYPE, EXTENT, IERROR)
    INTEGER DATATYPE, EXTENT, IERROR

MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)
    INTEGER DATATYPE, SIZE, IERROR

MPI_TYPE_COUNT(DATATYPE, COUNT, IERROR)
    INTEGER DATATYPE, COUNT, IERROR

MPI_TYPE_LB( DATATYPE, DISPLACEMENT, IERROR)
```

```
    INTEGER DATATYPE, DISPLACEMENT, IERROR

MPI_TYPE_UB( DATATYPE, DISPLACEMENT, IERROR)
    INTEGER DATATYPE, DISPLACEMENT, IERROR

MPI_TYPE_COMMIT(DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR

MPI_TYPE_FREE(DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR

MPI_GET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR

MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTCOUNT, POSITION, COMM,
             IERROR)
    <type> INBUF(*), OUTBUF(*)
    INTEGER INCOUNT, DATATYPE, OUTCOUNT, POSITION, COMM, IERROR

MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM,
             IERROR)
    <type> INBUF(*), OUTBUF(*)
    INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR

MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERROR)
    INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERROR
```

A.10 集団通信のための Fortran 言語呼び出し形式

```
MPI_BARRIER(COMM, IERROR)
    INTEGER COMM, IERROR

MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
    <type> BUFFER(*)
    INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR

MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
             ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
```

```
MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
            RECVTYPE, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
    COMM, IERROR

MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
            ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR

MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
            RECVTYPE, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
    COMM, IERROR

MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
            COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR

MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
            RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
    IERROR

MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
            COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR

MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,
            RDISPLS, RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
    RECVTYPE, COMM, IERROR
```

```
MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR

MPI_OP_CREATE( FUNCTION, COMMUTE, OP, IERROR)
    EXTERNAL FUNCTION
    LOGICAL COMMUTE
    INTEGER OP, IERROR

MPI_OP_FREE( OP, IERROR)
    INTEGER OP, IERROR

MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR

MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
              IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR

MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR
```

A.11 グループ、コンテクストその他のための Fortran 言語呼び出し形式

```
MPI_GROUP_SIZE(GROUP, SIZE, IERROR)
    INTEGER GROUP, SIZE, IERROR

MPI_GROUP_RANK(GROUP, RANK, IERROR)
    INTEGER GROUP, RANK, IERROR

MPI_GROUP_TRANSLATE_RANKS(GROUP1, N, RANKS1, GROUP2, RANKS2, IERROR)
    INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERROR

MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERROR)
    INTEGER GROUP1, GROUP2, RESULT, IERROR

MPI_COMM_GROUP(COMM, GROUP, IERROR)
```

```
    INTEGER COMM, GROUP, IERROR

MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR

MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR

MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR

MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR

MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR

MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR

MPI_GROUP_RANGE_EXCL(GROUP, N, RANGES, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR

MPI_GROUP_FREE(GROUP, IERROR)
    INTEGER GROUP, IERROR

MPI_COMM_SIZE(COMM, SIZE, IERROR)
    INTEGER COMM, SIZE, IERROR

MPI_COMM_RANK(COMM, RANK, IERROR)
    INTEGER COMM, RANK, IERROR

MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR)
    INTEGER COMM1, COMM2, RESULT, IERROR

MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
    INTEGER COMM, NEWCOMM, IERROR

MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
    INTEGER COMM, GROUP, NEWCOMM, IERROR

MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)
    INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR
```

```
MPI_COMM_FREE(COMM, IERROR)
    INTEGER COMM, IERROR

MPI_COMM_TEST_INTER(COMM, FLAG, IERROR)
    INTEGER COMM, IERROR
    LOGICAL FLAG

MPI_COMM_REMOTE_SIZE(COMM, SIZE, IERROR)
    INTEGER COMM, SIZE, IERROR

MPI_COMM_REMOTE_GROUP(COMM, GROUP, IERROR)
    INTEGER COMM, GROUP, IERROR

MPI_INTERCOMM_CREATE(LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,
              NEWINTERCOMM, IERROR)
    INTEGER LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,
    NEWINTERCOMM, IERROR

MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, INTRACOMM, IERROR)
    INTEGER INTERCOMM, INTRACOMM, IERROR
    LOGICAL HIGH

MPI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)
    EXTERNAL COPY_FN, DELETE_FN
    INTEGER KEYVAL, EXTRA_STATE, IERROR

MPI_KEYVAL_FREE(KEYVAL, IERROR)
    INTEGER KEYVAL, IERROR

MPI_ATTR_PUT(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR

MPI_ATTR_GET(COMM, KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
    LOGICAL FLAG

MPI_ATTR_DELETE(COMM, KEYVAL, IERROR)
    INTEGER COMM, KEYVAL, IERROR
```

A.12 プロセストポロジーのための Fortran 言語呼び出し形式

```
MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)
    INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
    LOGICAL PERIODS(*), REORDER

MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)
    INTEGER NNODES, NDIMS, DIMS(*), IERROR

MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES, REORDER, COMM_GRAPH,
            IERROR)
    INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERROR
    LOGICAL REORDER

MPI_TOPO_TEST(COMM, STATUS, IERROR)
    INTEGER COMM, STATUS, IERROR

MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, IERROR)
    INTEGER COMM, NNODES, NEDGES, IERROR

MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR)
    INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR

MPI_CARTDIM_GET(COMM, NDIMS, IERROR)
    INTEGER COMM, NDIMS, IERROR

MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
    INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
    LOGICAL PERIODS(*)

MPI_CART_RANK(COMM, COORDS, RANK, IERROR)
    INTEGER COMM, COORDS(*), RANK, IERROR

MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)
    INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR

MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERROR)
    INTEGER COMM, RANK, NNEIGHBORS, IERROR

MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR)
    INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR

MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR)
```

```
    INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR

MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, IERROR)
    INTEGER COMM, NEWCOMM, IERROR
    LOGICAL REMAIN_DIMS(*)

MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR)
    INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR
    LOGICAL PERIODS(*)

MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR)
    INTEGER COMM, NNODES, INDEX(*), EDGES(*), NEWRANK, IERROR
```

A.13 環境問い合わせのための Fortran 言語呼び出し形式

```
MPI_GET_PROCESSOR_NAME(NAME, RESULTLEN, IERROR)
    CHARACTER*(*) NAME
    INTEGER RESULTLEN, IERROR

MPI_ERRHANDLER_CREATE(FUNCTION, HANDLER, IERROR)
    EXTERNAL FUNCTION
    INTEGER ERRHANDLER, IERROR

MPI_ERRHANDLER_SET(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERRHANDLER, IERROR

MPI_ERRHANDLER_GET(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERRHANDLER, IERROR

MPI_ERRHANDLER_FREE(ERRHANDLER, IERROR)
    INTEGER ERRHANDLER, IERROR

MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR)
    INTEGER ERRORCODE, RESULTLEN, IERROR
    CHARACTER*(*) STRING

MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR)
    INTEGER ERRORCODE, ERRORCLASS, IERROR

DOUBLE PRECISION MPI_WTIME()
```

```
DOUBLE PRECISION MPI_WTICK()

MPI_INIT(IERROR)
    INTEGER IERROR

MPI_FINALIZE(IERROR)
    INTEGER IERROR

MPI_INITIALIZED(FLAG, IERROR)
    LOGICAL FLAG
    INTEGER IERROR

MPI_ABORT(COMM, ERRORCODE, IERROR)
    INTEGER COMM, ERRORCODE, IERROR
```

A.14 プロファイリングのための Fortran 言語呼び出し形式

```
MPI_PCONTROL(level)
    INTEGER LEVEL, ...
```

MPI Function Index

MPI_ABORT, 197
MPI_ADDRESS, 67
MPI_ALLGATHER, 107
MPI_ALLGATHERV, 108
MPI_ALLREDUCE, 122
MPI_ALLTOALL, 109
MPI_ALLTOALLV, 110
MPI_ATTR_DELETE, 169
MPI_ATTR_GET, 169
MPI_ATTR_PUT, 168

MPI_BARRIER, 93
MPI_BCAST, 93
MPI_BSEND, 27
MPI_BSEND_INIT, 53
MPI_BUFFER_ATTACH, 33
MPI_BUFFER_DETACH, 33

MPI_CANCEL, 51
MPI_CART_COORDS, 182
MPI_CART_CREATE, 177
MPI_CART_GET, 181
MPI_CART_MAP, 186
MPI_CART_RANK, 182
MPI_CART_SHIFT, 184
MPI_CART_SUB, 185
MPI_CARTDIM_GET, 181
MPI_COMM_COMPARE, 142
MPI_COMM_CREATE, 143
MPI_COMM_DUP, 143
MPI_COMM_FREE, 145
MPI_COMM_GROUP, 137
MPI_COMM_RANK, 142
MPI_COMM_REMOTE_GROUP, 155
MPI_COMM_REMOTE_SIZE, 155
MPI_COMM_SIZE, 141
MPI_COMM_SPLIT, 144
MPI_COMM_TEST_INTER, 154

MPI_DIMS_CREATE, 177

MPI_ERRHANDLER_CREATE, 192
MPI_ERRHANDLER_FREE, 193
MPI_ERRHANDLER_GET, 193
MPI_ERRHANDLER_SET, 192
MPI_ERROR_CLASS, 195
MPI_ERROR_STRING, 193

MPI_FINALIZE, 196

MPI_GATHER, 94
MPI_GATHERV, 95
MPI_GET_COUNT, 21
MPI_GET_ELEMENTS, 73
MPI_GET_PROCESSOR_NAME, 190
MPI_GRAPH_CREATE, 178
MPI_GRAPH_GET, 181
MPI_GRAPH_MAP, 187
MPI_GRAPH_NEIGHBORS, 183
MPI_GRAPH_NEIGHBORS_COUNT, 183
MPI_GRAPHDIMS_GET, 180
MPI_GROUP_COMPARE, 136
MPI_GROUP_DIFFERENCE, 138

MPI_GROUP_EXCL, 139
MPI_GROUP_FREE, 140
MPI_GROUP_INCL, 138
MPI_GROUP_INTERSECTION, 137
MPI_GROUP_RANGE_EXCL, 140
MPI_GROUP_RANGE_INCL, 139
MPI_GROUP_RANK, 135
MPI_GROUP_SIZE, 135
MPI_GROUP_TRANSLATE_RANKS, 136
MPI_GROUP_UNION, 137

MPI_IBSEND, 37
MPI_INIT, 196
MPI_INITIALIZED, 197
MPI_INTERCOMM_CREATE, 157
MPI_INTERCOMM_MERGE, 157
MPI_IPROBE, 48
MPI_IRECV, 38
MPI_IRSEND, 38
MPI_ISEND, 36
MPI_ISSEND, 37

MPI_KEYVAL_CREATE, 166
MPI_KEYVAL_FREE, 168

MPI_OP_CREATE, 118
MPI_OP_FREE, 120

MPI_PACK, 83
MPI_PACK_SIZE, 86
MPI_PCONTROL, 199
MPI_PROBE, 49

MPI_RECV, 19
MPI_RECV_INIT, 55
MPI_REDUCE, 111
MPI_REDUCE_SCATTER, 123
MPI_REQUEST_FREE, 41
MPI_RSEND, 28
MPI_RSEND_INIT, 54

MPI_SCAN, 124
MPI_SCATTER, 103
MPI_SCATTERV, 104
MPI_SEND, 16
MPI_SEND_INIT, 53
MPI_SENDRECV, 57
MPI_SENDRECV_REPLACE, 58
MPI_SSEND, 27
MPI_SSEND_INIT, 54
MPI_START, 55
MPI_STARTALL, 56

MPI_TEST, 40
MPI_TEST_CANCELLED, 52
MPI_TESTALL, 45
MPI_TESTANY, 44
MPI_TESTSOME, 46
MPI_TOPO_TEST, 180
MPI_TYPE_COMMIT, 70
MPI_TYPE_CONTIGUOUS, 60
MPI_TYPE_COUNT, 69
MPI_TYPE_EXTENT, 68
MPI_TYPE_FREE, 71
MPI_TYPE_HINDEXED, 65
MPI_TYPE_HVECTOR, 63
MPI_TYPE_INDEXED, 64
MPI_TYPE_LB, 70
MPI_TYPE_SIZE, 68
MPI_TYPE_STRUCT, 66
MPI_TYPE_UB, 70
MPI_TYPE_VECTOR, 61

MPI_UNPACK, 84

MPI_WAIT, 39
MPI_WAITALL, 44
MPI_WAITANY, 43
MPI_WAITSOME, 46
MPI_WTICK, 195
MPI_WTIME, 195