MPI-PreDebugger：通信依存解析に基づくメッセージ通信並列プログラム向けデバッグ支援ツール

置田真生[†] 伊野文彦[††]
藤本典幸[††] 萩原兼一[††]

一般に，メッセージ通信ライブラリを用いた並列プログラムのデバッグは繁雑である．その原因として通信を介したプロセス間の依存関係（通信依存）があげられる．バグを含むプロセス（異常プロセス）に異常が生じると，通信依存を持つプロセスへ異常が伝播し本来バグを含まないプロセスにも異常が生じる．既存のデバッガを用いると各プロセスごとの異常箇所は分かるが，バグを含む箇所の特定はユーザが行う必要がある．そこで，我々は通信依存を解析し異常プロセスを自動的に検索する手法を提案する．また，その手法に基づいて開発したデバッグ支援ツール MPI-PreDebugger（MPI-PD）について述べる．MPI-PD は，プログラム実行時に通信ごとに異常の発生を確認し，異常発生時に自動的に検索した異常プロセスを指摘する．MPI-PD が異常プロセスを検索することでバグを含むと考えられる範囲を絞り込み，バグ特定におけるプログラマの負担を軽減する．本稿では，異常プロセスを検索する手法と MPI-PD のデバッグ支援機能について述べ，評価実験の結果を示す．この実験では，MPI-PD を用いることで並列プログラムのデバッグ時間を最大で約 6 分の 1 に短縮でき，デバッグの繁雑さを軽減できた．

MPI-PreDebugger: A Debugging Support Tool for Message-passing Parallel Programs via Communication Dependency Analysis

Masao Okita,[†] Fumihiko Ino,[††] Noriyuki Fujimoto[††] and Kenichi Hagihara[††]

In general, debugging for message-passing parallel programs is complicated. One reason for this is that errors may spread along communication dependency between processes. Because of the spread of errors, errors are observed not only in a process which contains the original error but also in other processes. So it is time-consuming to find the original error. Existing debuggers support for programmers to find errors but cannot identify the original error from found errors. In this paper, we propose a method to automatically find the original error according to communication dependencies between processes. We also introduce MPI-PD (MPI-PreDebugger), which implements the proposed method. MPI-PD checks communication errors during program execution. If MPI-PD observes errors, then it tracks communication dependencies and points out the original error processes automatically. Finally, we give some experimental results. In the experiments, MPI-PD reduced debugging time to about 1/6 and freed programmers from some complicated work.

1. はじめに

一般に，メッセージ通信ライブラリ MPI[1)]を用いた並列プログラム（**MPI プログラム**）のデバッグは繁雑である．その原因として，逐次プログラムと比較してデバッグ情報が膨大であること[2)]，また通信を介したプロセス間の依存関係（**通信依存**）が存在することがあげられる．あるプロセスでバグによる異常が生じると，通信依存を持つプロセスに異常が伝播し本来バグを含まないプロセスでも異常が生じる．このような異常の伝播がバグを含む箇所の特定を繁雑にしている．

並列プログラムのデバッグを支援するために，デバッガに関する様々な研究が行われている[3)~10)]．並列プログラムの実行状況を視覚的に理解できるものと

† 大阪大学大学院基礎工学研究科情報数理系専攻
Department of Informatics and Mathematical Science, Graduate School of Engineering Science, Osaka University
†† 大阪大学大学院情報科学研究科コンピュータサイエンス専攻
Department of Computer Science, Graduate School of Information Science and Technology, Osaka University

して，Vampir[3]，XMPI[4]および ATEMPT[5]があげられる．これらは並列プログラムの実行時にログファイルを生成し，ログファイルを基に各プロセスにおける計算および通信の様子を時系列状に視覚化することで，プログラマがプロセスの挙動を把握できる．

また，並列プログラムに対してステップ単位の詳細なデバッグ作業を支援するものとして，TotalView[6]，MPIGDB[7]および dbxR-II[8]があげられる．TotalView はデータの視覚化機能などを備えており，並列プログラムをステップ実行しながら組合せ判定が未解決のメッセージに関して通信依存を Message Queue Graph（MQG）として視覚化できる[11]．MPIGDB は逐次デバッガ GDB を基にしていて，Multi-Purpose Daemon（MPD）[12]を通じてプロセスごとにアタッチした GDB を制御できる．dbxR-II[8]は非決定的な並列プログラムに対して，同じ実行動作を繰り返し得る再演法を実現しサイクリックなデバッグを可能にする．

さらに，バグを含む箇所の特定を支援するものとして，GUARD[9]および Netzer らの方法[10]があげられる．GUARD は並列プログラムを移植するときに有用であり，デバッグ対象とする移植先プログラムに加え，バグを含まない移植元プログラムを必要とする．両プログラムの主要なデータ構造の値を実行時に比較することでバグを含む箇所の特定を支援する．一方，Netzer らはメッセージの到着順序が非決定的であるときバグが生じやすいと考え，メッセージが想定外の順序で到着したために，プログラムの動作がプログラマの意図に反した箇所を実行時に指摘するアルゴリズムを示した．

このように，既存のデバッガは主に視覚化や再演制御に重点を置いており，バグを含む箇所を自動的に特定できるものは，我々の知る限り GUARD や Netzer らの方法のみである．したがって，プログラマはデバッガの視覚化機能などを用いて各プロセスの挙動を把握した後，さらにプログラマ自身が詳細な解析を行ってバグを含む箇所を特定する必要がある．

そこで我々は，通信依存解析に基づいて，バグを含むプロセスを自動的に検索できるツール MPI-PreDebugger（**MPI-PD**）を作成した．MPI-PD が対象とする MPI プログラムは，バグによる異常がプログラムの実行を停止（異常停止）させるものである．MPI-PD がバグを含むプロセス（**異常プロセス**）を検索することでバグを含むと考えられる範囲を絞り込み，バグ特定におけるプログラマの負担を軽減する．

MPI-PD は，MPI プログラム実行時に通信ごとに異常の発生を確認し，異常停止時にログファイルを生成する．その後，ログファイルを基に通信依存を解析し，その解析結果をイベントグラフ[5]として視覚化する．プログラマはイベントグラフを見るだけで異常プロセスおよび異常停止時の状況を確認できる．

本稿では MPI プログラムのデバッグ作業を，異常停止時の状況を把握し異常プロセスを特定する特定フェーズ P1，絞り込んだ異常プロセスを詳細にデバッグしバグを修正する修正フェーズ P2 の 2 つに分ける．プログラマはまず MPI-PD を利用して P1 に取り組む．次に既存のデバッガを利用して P2 に取り組む．

以下，まず MPI プログラムにおけるデバッグおよび通信依存について整理する．次に提案する異常プロセスの検索手法を示し，MPI-PD の概要を述べる．その後，MPI-PD および TotalView を用いて MPI プログラムをデバッグした評価実験を示す．

2. 諸定義：MPI プログラムにおけるデバッグ

本章では MPI-PD が対象とする問題およびその解法を示すための諸定義を示す．

2.1 デバッグの定義

MPI プログラム S のソースコードを文のならび $s_1, s_2, ..., s_l$ と表記する．ここで，文 s_k $(1 \leq k \leq l)$ は，基となる逐次言語（C 言語など）の文に加えて MPI 通信命令（通信命令）で定義される[1]．

S を N 個のプロセスで並列実行したとき，プロセス p で得られる実行系列を**イベント**のならび $e_1^p, e_2^p, ..., e_{M_p}^p$ と表記する．ここで，イベントは文の実行を表し，実行時の参照値などを情報に持つ．e_i^p および M_p は，各々 p における i 番目のイベント，文の総実行回数を表す．

さらに，S の実行が成功するとは，S が異常停止をともなうバグを持たず，かつ意味上のバグを持たないことであると考え，その成否 $R(S)$ を以下のように定義する．

$$R(S) \equiv \bigwedge_{p=1}^{N} \bigwedge_{i=1}^{M_p} (exit(e_i^p) \wedge valid(e_i^p)) \qquad (1)$$

論理関数 $exit(e_i^p)$ は e_i^p が正常完了したかどうかを表す．$exit(e_i^p) = T$（真）のとき p は次のイベント e_{i+1}^p を処理でき，$exit(e_i^p) = F$（偽）のとき p は異常停止する．たとえば，e_i^p においてセグメンテーションフォールトや，2.2 節で述べる通信フォールトが発生するとき，$exit(e_i^p) = F$ であり p は異常停止する．

次に，$valid(e_i^p)$ は e_i^p における処理内容が妥当かどうかを表す．$valid(e_i^p) = T$ のとき e_i^p の処理内容は意

表 1 ブロッキング型の受信イベント（MPI_Recv）に対する正常完了の定義
Table 1 Definition of normal exit on blocking receive event (MPI_Recv).

定義式			表記
$reach(e_i^s)$	$\equiv \bigwedge_{l=1}^{i-1} exit(e_l^s)$	(2)	$bufsz(e)$：e の通信バッファサイズ.
$pair(e_i^s, e_j^r)$	$\equiv ((e_i^s$ は送信イベント$) \wedge (comm(e_i^s) = comm(e_j^r))$		$comm(e)$：e のコミュニケータ.
	$\wedge\ (ptnr(e_i^s) = r) \wedge (ptnr(e_j^r) = s) \wedge (tag(e_i^s) = tag(e_j^r))$	(3)	$ptnr(e)$：e の通信相手のプロセス番号.
$oflw(e_i^s, e_j^r)$	$\equiv (bufsz(e_i^s) > bufsz(e_j^r))$	(4)	$tag(e)$：e の通信タグ.
$exit(e_j^r)$	$\equiv \exists i\ ((i \geq k) \wedge reach(e_i^s) \wedge pair(e_i^s, e_j^r) \wedge \neg oflw(e_i^s, e_j^r))$	(5)	

味的に正しく，$valid(e_i^p) = F$ のとき e_i^p は意味上の誤りを持つ．たとえば，e_i^p において演算子の指定を誤った代入文を実行する場合，あるいは送信バッファの指定を誤った通信命令を実行する場合，$valid(e_i^p) = F$ であり正しい計算結果が得られない．

本稿において MPI プログラム S をデバッグするとは，$R(S) = F$ を生じさせる文を修正し，$R(S') = T$ を満たす修正済 MPI プログラム S' を得ることを指す．

以降では，通信命令の実行に対応するイベントを**通信イベント**と呼び，それ以外を**計算イベント**と呼ぶ．さらに，通信命令の種類に基づいて通信イベントを**送信イベント**，**受信イベント**，**通信完了イベント**（MPI_Wait のとき）および**集合通信イベント**に分類する．

2.2 MPI 通信命令におけるフォールトの定義

本節では通信イベント e_i^p におけるフォールト（**通信フォールト**）を定義する．

以下に示す 2 つの条件のうち，どちらか一方が適合すれば e_i^p において通信フォールトが発生する．

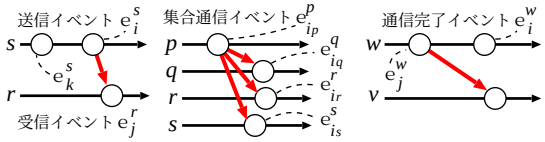
- 通信の不成立
- 受信バッファにおけるオーバフローの発生

前者では，e_i^p との通信に参加するプロセス q において処理すべき通信イベント e_j^q が欠落し，通信が成立しない．この条件は，(1) e_j^q の処理到達可能性 $reach(e_j^q)$，および，(2) e_i^p および e_j^q の組合せ $pair(e_i^p, e_j^q)$ を調べることで判定できる．

後者では，通信の成立後，受信バッファがオーバフローすることで通信に失敗する．この条件は，(3) オーバフローの発生 $oflw(e_i^p, e_j^q)$ を調べることで判定できる．

以下，ブロッキング型の受信イベント（MPI_Recv）e_j^r に対して上の条件（1）〜（3）を定め $exit(e_j^r)$ を定義する（**表 1**）．紙面の都合上，残りの通信イベントは省く．以降では，表 1 の表記欄に示す記号を用いる．

プロセス r が e_j^r を処理するときプロセス s が e_k^s を処理中であるとする（**図 1** (a)）．e_j^r と通信する送信イベントを e_i^s として，$reach(e_i^s)$ は s が $i-1$ 番目までのイベントの処理を正常完了できればよい（式 (2)）．次に，$pair(e_i^s, e_j^r)$ は通信命令，コミュニケータ，通信相手および通信タグが一致すればよい（式 (3)）．最後に，$oflw(e_i^s, e_j^r)$ は e_i^s の送信バッファサイズが e_j^r の受信バッファサイズを超えればよい（式 (4)）．式 (2)〜(4) を用いて $exit(e_j^r)$ は定義できる（式 (5)）．



(a) 1 対 1 通信 (b) 集合通信 (c) ノンブロッキング通信
図 1 通信イベント
Fig. 1 Communication events.

他の通信イベント e に関しても，同様に $exit(e)$ を定義できる．集合通信イベント e_i^p の場合，集合通信に参加するすべてのプロセス p〜s について条件（1）〜（3）を確認すればよい（図 1 (b)）．通信完了イベント e_i^w の場合，組となるノンブロッキング型の通信イベント（MPI_Isend もしくは MPI_Irecv）e_j^w について条件（1）〜（3）を確認すればよい（図 1 (c)）．

2.3 通信依存の定義

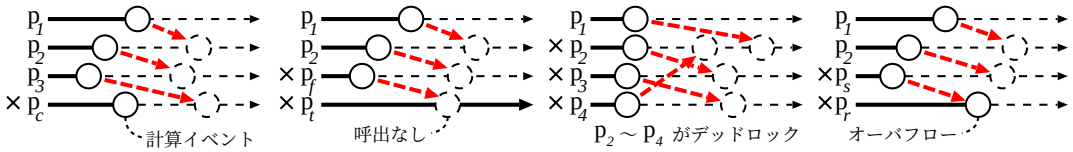
2.2 節の条件（1）および条件（3）で述べたとおり，p における通信イベント e_i^p の正常完了 $exit(e_i^p)$ は，その通信相手となるプロセス q における通信イベント e_j^q の組で評価できる．このように e_i^p の正常完了 $exit(e_i^p)$ が e_j^q に依存するとき，p が q に**通信依存**を持つと呼ぶ．

1 章で述べたとおり，MPI プログラムの実行においてプロセスが異常停止すると，そのプロセスに通信依存を持つすべてのプロセスが連鎖的に異常停止する．通信依存を持つプロセスは，それ自身がバグを含むか否かにかかわらず異常停止するため，このような連鎖的な異常停止の発生は異常停止時の状況を複雑にする．

以降では，連鎖的に異常停止したプロセスのうち，最初に異常停止したプロセスを**異常プロセス**と呼ぶ．

3. 提案するデバッグ支援手法

本章では，MPI-PD の核となるデバッグ支援手法



(a) 計算フォールト (b) 正常終了プロセスへの通信依存 (c) デッドロック (d) オーバフロー

図 2 異常プロセス検索の例
Fig. 2 Examples of original error processes.

（提案手法）について述べる．

我々はMPIプログラム S に対する実行の成否 $R(S)$ をプロセス軸 p および時間軸 i の二次元空間で定義した（式 (1)）．また，通信依存が異常停止時の状況を複雑にすることを指摘した（2.3 節）．

以上をふまえ，提案手法は通信依存を解析することで，異常プロセス p，および各プロセスにおいて最初にフォールトしたイベントの番号 i を検索する．プログラマは MPI-PD が検索した各イベント e_i^p に対して既存のデバッガ[6)~8)]を用いて $valid(e_i^p)$ の値を確認するなどの詳細なデバッグに専念できる．

3.1 異常プロセス検索アルゴリズム

提案する異常プロセス検索アルゴリズムを**図 3** に示す．紙面の都合上，図 3 のアルゴリズムは MPI-PD のものを簡略化し，集合通信イベントの扱いを省略した．

このアルゴリズムは，以下に示す 2 段階の検索処理から成る．

S1 ：各プロセス p において最初にフォールトしたイベント fe^p を検索（7〜12 行目）．

S2 ：異常プロセスを検索（13〜40 行目）．

S1 では，p ごとに $exit(e_i^p) = F$ を満たす最小の i を持つイベントを返せばよい．そのような i が存在しない場合は空の要素 e_{null} を返す．なお，イベントの集合 E が与えられて，各イベント e_i^p ごとに $exit(e_i^p)$ の値を実行時に定める方法は 4.2 節で述べる．

S2 では，通信依存を再帰的に追跡することで，異常プロセスを検索する．異常プロセスを指摘するとき，異常停止時の状況は以下の 4 通りである（**図 2**）．

（a） 計算イベントにおいてフォールト（22 行目）．
（b） 正常終了プロセスへの通信依存が存在（24 行目）．
（c） デッドロックが発生（26 行目）．
（d） 通信バッファがオーバフロー（26 行目）．

まず状況（a）では，計算イベントにおいて異常停止するプロセス p_c は異常停止時に他のプロセスへの通信依存を持たないため，p_c を異常プロセスと判断する（図 2 (a)）．次に状況（b）では，正常終了したプロセス p_t が通信命令を呼び忘れたのか，あるいは異常終了したプロセス p_f が通信相手の指定を誤っているのかを判断することが難しいため，p_t および p_f を異常プロセスと判断する（図 2 (b)）．また状況（c）では，デッドロックに参加するプロセスの集合（P_{dep} の一部）を異常プロセスと判断する（図 2 (c)）．最後に状況（d）では，送信プロセス p_s および受信プロ

```
1:  // 入力
2:  // P = {1, 2, .., N}：プロセス番号の集合
3:  // E = {e_i^p | p ∈ P, 1 ≤ i ≤ E_p}：イベントの集合
4:  // 出力
5:  // P_e：着目すべきプロセス番号の集合変数
6:  // E_e：着目すべきイベントの集合変数
7:  E_e := ∅;                          // E_e を空集合に初期化
8:  foreach p ∈ P begin
9:      // p において最初にフォールトしたイベントを fe^p へ
10:     fe^p := FirstFaultEvent(p);   // 省略.
11:     E_e := E_e ∪ {fe^p};          // E_e に fe^p を追加
12: end
13: P_e := ∅;                          // P_e を空集合に初期化
14: foreach p ∈ P begin
15:     if (SearchOrgErrProcess(p, ∅) ≠ 0)
16:         P_e := P_e ∪ {p};          // P_e に p を追加
17: end
18: // p の通信依存を調べ P_e を更新する再帰関数
19: function SearchOrgErrProcess(p, P_dep) begin
20:     if ((p ∈ P_e) || ((fe^p = e_null) && (P_dep = ∅)))
21:         return 0;                  // p は判定済か正常
22:     else if (fe^p は計算イベント) // (a) p が異常
23:         return -1;
24:     else if (fe^p = e_null)        // (b) p および呼出元
25:         return -2;                 //     が異常
26:     else if (p ∈ P_dep)            // (c), (d) P_dep の
27:         return p;                  //     一部が異常
28:     endif
29:     q := ptnr(fe^p);               // fe^p の通信相手
30:     Q_dep := P_dep ∪ {p};          // 呼出履歴の更新
31:     retval := SearchOrgErrProcess(q, Q_dep);
32:     if (retval ≠ 0)
33:         P_e := P_e ∪ {q};          // P_e に q を追加
34:     if (retval = p)
35:         retval := 0;
36:     else if (retval < 0)
37:         retval++;
38:     endif
39:     return retval;
40: end
```

図 3 異常プロセス検索アルゴリズム
Fig. 3 Algorithm to search original error processes.

セス p_r のどちらに誤りがあるのかを判断することが難しいため，p_s および p_r を異常プロセスと判断する（図 2 (d)）．

なお，図 3 のアルゴリズムは，通信相手の指定が正しいものとして通信依存を追跡する．したがって，指定を誤った異常プロセスが状況（a)〜(d）で指摘する結果から洩れる可能性はある．詳細は 5.2 節で考察する．

4. MPI-PreDebugger の概要

MPI-PD は C 言語と GUI ツールキット Ruby/GTK が動作する環境で利用できる．対象とする並列プログラムは C 言語で記述した MPI プログラムである．

MPI-PD のデバッグ支援機能はログ生成部と視覚化部に分類できる（**表 2**）．以降，MPI-PD を用いてデバッグする際の手順，およびログ生成部について述べる．なお，異常プロセス検索機能は図 3 に示したアルゴリズムに基づく．紙面の都合上，その詳細を省く．

4.1 利 用 手 順

MPI-PD を用いてデバッグする手順を**図 4** に示す．

まず，プログラマは自動変換ツール mpi2pd を用いてデバッグ対象の MPI プログラムの通信命令を，パターンマッチに基づいて通信フォールト確認処理つきの PD 通信命令（変換前の通信命令を内包）に変換する．その後，変換済 MPI プログラムをコンパイル，ログ生成ライブラリ libpdmpi.a とリンクすることで実行ファイルを得る．その実行ファイルを並列実行すると，MPI-PD は通信ごとに異常の発生を確認し，異常が発生するときログファイルを生成する（4.2 節）．

次に，ログファイルを視覚化ツール pdview に入力しイベントグラフを表示する．イベントグラフは縦軸にプロセスを並べ，横軸にイベントを処理順に配置した図である．pdview が描画するイベントは，各々のプロセスでフォールトした通信イベント，およびその直前に正常終了できた通信イベントである．また，pdview は検索できた異常プロセスを強調表示し，異常停止時の状況（a)〜(d）を示す．プログラマはイベントグラフ内の画像オブジェクトをマウスでクリックすることで，ログファイルに記録した以下の情報を通信イベントごとに確認できる．

- イベント番号，プロセス番号，ソースコード行，通信命令の種類およびその引数の値
- フォールトの原因（$exit(e_i^p) = F$ を生じさせた式：MPI_Recv であれば式 (2)〜(4)）

これらの情報をもとに，プログラマはデバッグを行う．

表 2 MPI-PD のデバッグ支援機能
Table 2 Debugging support functions of MPI-PD.

部名	ツール名	デバッグ支援機能
ログ生成部	mpi2pd	MPI プログラム変換機能
	libpdmpi.a	通信フォールト確認機能 ログファイル生成機能
視覚化部	pdview	異常プロセス検索機能 イベントグラフ表示機能

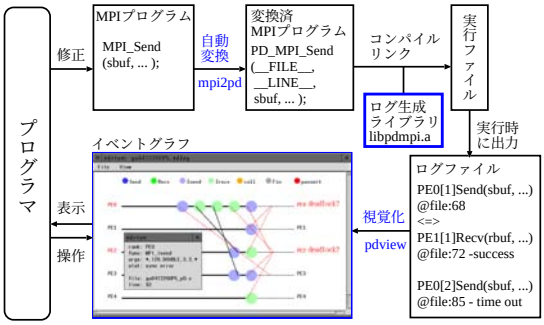


図 4 MPI-PD を用いたデバッグの手順
Fig. 4 Debugging process with MPI-PD.

4.2 通信フォールトの確認

MPI-PD は，通信イベント e_i^p に対する $exit(e_i^p)$ の値を MPI プログラムの実行中に決定する．

具体的には，プロセス p ごとに**管理プロセス** m_p を用意し，p が e_i^p を処理する直前に，m_p が通信イベントごとに定められた定義式（MPI_Recv であれば式 (2)〜(5)）に基づいて $exit(e_i^p)$ の値を決定する．$exit(e_i^p) = T$ のとき，m_p は p が e_i^p を処理することを許可し，e_i^p の情報を通信履歴を蓄えるバッファに記録する．$exit(e_i^p) = F$ のとき，m_p は e_i^p において通信フォールトが発生すると判断し，p を停止させ，通信履歴を基に e_i^p およびそれまでに正常完了できた全 $e_j^p (j < i)$ の情報をログファイルに記録する．

なお，MPI ではノンブロッキング通信が利用できるため，通信命令の呼出順序と通信フォールトの決定順序は必ずしも一致しない．たとえば，通信完了命令（MPI_Wait）のフォールトを判定するためには，過去に呼び出したノンブロッキング通信命令を参照する必要がある．そこで，過去に処理した通信イベントを記憶するために，m_p は通信キュー Q_p を持つ．

また，$exit(e_i^p)$ の値を決定する際，その評価を以下の 2 段階に分ける必要がある．

- $E1$：通信の成立
- $E2$：オーバフローの無発生

この理由を，受信イベント e_j^r を例に説明する．式 (5) から，e_j^r に対して $E1$ は $reach(e_i^s) \wedge pair(e_i^s, e_j^r)$ であ

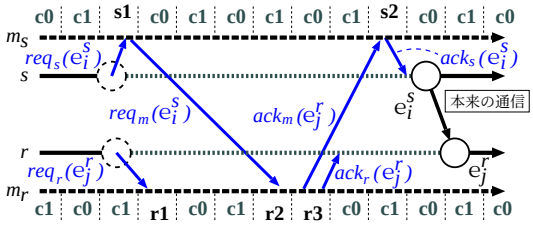


図 5 通信フォールト確認の流れ（MPI_Ssend）
Fig. 5 Checking communication fault (MPI_Ssend).

り，$E2$ は $\neg oflw(e_i^s, e_j^r)$ である．ここで，先に述べたとおり，通信命令の呼出順序と通信フォールトの決定順序は必ずしも一致しない．したがって，$exit(e_j^r) = F$ となる e_i^s が Q_s に存在する場合，$E1 = T$ かつ $E2 = F$ となる e_i^s に対しては通信フォールトであると断定できるが，$E1 = F$ であればさらに Q_s を検索する必要がある．ゆえに，式 (5) を $E1$ と $E2$ に分けて評価する．

さらに，通信命令の呼び忘れおよび無限ループなど，$E1 = T$ となりえない e_i^p に対して $E1 = F$ と決定し $exit(e_i^p) = F$ とする必要がある．しかし，無限ループおよび正常なループを実行時に自動的に区別することは容易ではない．そこで，e_i^p ごとにタイムアウト時間 $t(e_i^p)$ を設定し，$t(e_i^p)$ までに $E1 = T$ を満たす通信イベントが処理されなければ $exit(e_i^p) = F$ とする．

なお，$t(e_i^p)$ の値として不適切な値を設定することで，本来 $E1 = T$ となる e_i^p を誤って $E1 = F$ と決定する可能性はある．このような状況を回避するために，MPI-PD は $t(e_i^p)$ の値をプロセスごとに任意の値に設定できる．プログラムの種類や問題サイズごとに適切な $t(e_i^p)$ の値を設定することは今後の課題である．

以下，**図 5** を用いて管理プロセスの動作を示す．なお，簡単のためプロセス s からプロセス r への同期モードのブロッキング送信（MPI_Ssend）の例とした．

送信側 m_s および受信側 m_r で共通：

（c0）待機状態．管理を担当するプロセスおよび他の管理プロセスからの問合せを待つ．

（c1）タイムアウト確認．各自の通信キューを参照し，各通信イベント e^p に対して $t(e^p)$ を確認．タイムアウトした e^p に対して $exit(e^p) = F$ と決定し，p に停止要求 $abort_p(e^p)$ を送信．

送信側 m_s：

（s1）送信要求受理．s からの送信要求 $req_s(e_i^s)$ を受信し，e_i^s を $t(e_i^s)$ とともに Q_s に登録．受信した情報を基に $r = ptnr(e_i^s)$ を求め e_i^s の情報を通信要求 $req_m(e_i^s)$ として m_r に送信．

（s2）送信実行．他の管理プロセスからの通信許可 $req_m(e_j^r)$ を受信するたびに Q_s を検索し $exit(e_j^r) = T$ となる e_i^s を選ぶ．s へ送信許可 $ack_s(e_i^s)$ を送信し，e_i^s を Q_s から除去，e_i^s と e_j^r の情報を通信履歴に記録．

受信側 m_r：

（r1）受信要求受理．r からの受信要求 $req_r(e_j^r)$ を受信し，Q_r より $E1 = T$ となる e_i^s を検索．

- $E1 = T$ となる e_i^s が Q_r に存在．
  ⇒ e_i^s を通信相手と決定し（r3）に遷移．
- $E1 = T$ となる e_i^s を Q_r に存在しない．
  ⇒ $exit(e_j^r)$ の評価を保留し，e_j^r を $t(e_j^r)$ とともに Q_r に登録．

（r2）通信要求受理．他の管理プロセスからの通信要求 $req_m(e_i^s)$ を受信するたびに Q_r より $E1 = T$ となる e_j^r を検索．

- $E1 = T$ となる e_j^r が Q_r に存在．
  ⇒ e_j^r を通信相手と決定し（r3）に遷移．
- $E1 = T$ となる e_j^r が Q_r に存在しない．
  ⇒ $exit(e_i^s)$ の評価を保留し，e_i^s を $t(e_i^s)$ とともに Q_r に登録．

（r3）受信実行．e_i^s および e_j^r について $E2$ を評価．

- $E2 = T$.
  ⇒ $exit(e_j^r) = T$ と決定．m_s に通信許可 $ack_m(e_j^r)$ を送信し，e_j^r を Q_r から除去，r へ受信許可 $ack_r(e_j^r)$ を送信．e_i^s と e_j^r の情報を通信履歴に記録．
- $E2 = F$.
  ⇒ $exit(e_j^r) = F$ と決定し，r へ停止要求 $abort_r(e_j^r)$ を送信．

なお，送信命令がバッファモード（MPI_Bsend）の場合，通信フォールトの確認手順を変更する．この場合，s は m_s に $req_s(e_i^s)$ を送信した後，$ack_s(e_i^s)$ の受信を待たずに本来の通信を行う．以降，通信イベント e_j^s $(j > i)$ を処理するたびに m_s からの $abort_s(e_i^s)$ を受信したか否かを確認する．$abort_s(e_i^s)$ を受信した場合，プログラムを停止させ，そうでない場合，e_j^s の通信フォールトを確認する．

また，集合通信の場合，同期モードと同様の処理を通信に参加するすべての管理プロセスで実行する．ノンブロッキング通信の場合，(s1) および (r1) を各々，送信イベント処理時および受信イベント処理時に実行し，ack_s および ack_r の送信を通信完了イベント処理時に実行する．

このように，管理プロセスどうしで情報を交換することで，MPI プログラムを実行するプロセスが異常停止する前に通信フォールトを確認しログを記録できる．

表 3 実験の概要
Table 3 Overview of experiments.

実験項目	実験内容	対象プログラム 個数	行数	作成者	デバッグ作業者
（1）利用の可能性	バグ事例の分析（1-a）	28 個	50〜300 行	初習者	作成者
	大規模プログラムへの適用（1-b）	2 個	1 万行以上	自動生成	自動生成機構の開発者
（2）利用の効果	MPI-PD 使用時と未使用時を比較	12 個	実験（1-a）と同様		作成者および第三者
（3）既存デバッガとの比較	機能の比較	7 個	実験（1-b）と同様		第三者

表 4 サンプルプログラムの個数とその作成者
Table 4 Number of sample programs for experiments and their author.

作成者	A	B	C	D	E	F
プログラム個数	10	8	6	2	1	1

表 5 MPI プログラムへの適用結果
Table 5 Experimental results on applying MPI programs.

試行項目	プログラムの数（個）	試行結果（個） 成功	失敗
イベントグラフの表示	30 (2)	17 (2)	13 (0)
異常プロセスの指摘	17 (2)	14 (2)	3 (0)

括弧内は SPPC に基づき自動生成したプログラムの内訳を表す.

5. MPI-PD の評価実験

本章では，MPI-PD の有用性を示すために，以下の 3 項目を確認することを目的とした実験の結果を示す.

実験（1） 利用の可能性．バグの事例を分析することで MPI-PD が対処できる事例がどの程度存在するのかを示す．また，1 万行を超える MPI プログラムに対して MPI-PD を用いてデバッグした事例を示す（5.2 節）.

実験（2） 利用の効果．MPI-PD を用いて異常プロセスをあらかじめ絞り込むことでデバッグのための作業時間をどの程度短縮できるのかを示す（5.3 節）.

実験（3） 既存のデバッガとの比較．TotalView および MPI-PD を比較した結果を示す（5.4 節）.

5.1 実験方法

表 3 に各項目実験（1）〜（3）における概要を示す．実験で用いた 2 種類の MPI プログラムについて述べる.

まず，MPI プログラム初習者を対象とするプログラミング演習を通じて得られた MPI プログラムを用いた．演習の参加者は 6 名の大学院生 A〜F であり，連立方程式をガウスの消去法を用いて解くプログラムを作成した．演習の過程で得られたバグを含む MPI プログラムは 28 個であり，50〜300 行の規模である（表 3 の実験（1-a））．**表 4** に各被験者が提出したサンプルプログラムの個数を示す.

さらに，タスクスケジューリングアルゴリズム SPPC [13] に基づいて自動生成した MPI プログラムを用いた．用いた 2 個の MPI プログラムは各々1 万行以上の規模であり（表 3 の実験（1-b））, 人間の作成した MPI プログラムと比較して通信の組合せが不規則かつ複雑である点が特徴である.

MPI プログラムの実行には，16 台の PC を Myrinet [14] ネットワークで接続した PC クラスタを用いた．また，MPI の実装として MPICH [7] を用いた.

5.2 MPI-PD が対処できるバグの事例分析

バグを含む 30 個の MPI プログラムに対して MPI-PD を用いてイベントグラフを表示することを試みた．この結果，過半数を超える 17 個の MPI プログラムに対してイベントグラフを表示できた（**表 5**）.

イベントグラフを表示できなかった 13 個の MPI プログラムは，異常停止をともなわないバグを含んでいた．現在の MPI-PD は異常停止をともなうバグを対象とするため，これらの MPI プログラムを実行したときにログファイルが生成されず，イベントグラフを表示できなかった．このようなバグの例として，演算対象を誤ることで計算結果が異なるバグやノンブロッキング通信時に通信バッファの内容を変更してしまうことで通信内容が異なるバグがあげられ，これらのバグへの対処は今後の課題である.

イベントグラフを表示できた 17 個の MPI プログラムを対象として，MPI-PD が異常プロセスを正しく指摘できているか否かを確認した．結果，14 個の MPI プログラムに対して確認できた（表 5）.

正しい異常プロセスを確認できなかった 3 個の MPI プログラムはすべてのプロセスが計算イベントにおいて異常停止しており，計算イベントを記録しない MPI-PD はそれらを記録できなかった．ゆえに，フォールトしたイベントの情報がログファイルになく，MPI-PD は異常プロセスを正しく指摘できなかった.

しかしながら，すべてのプロセスが計算イベントにおいて異常停止するとき，連鎖的な異常停止は発生しない．したがって，この場合プログラマはすべてのプロセスに着目すべきであるといえる．すなわち，プロ

表 6 デバッグ作業時間の計測結果
Table 6 Experimental results on debugging time.

S	E	作業時間（秒） P1：異常プロセス特定 使用 T'_{P1} G	作業時間（秒） P1：異常プロセス特定 使用 T'_{P1} H	作業時間（秒） P1：異常プロセス特定 未使用 T_{P1}	作業時間（秒） P2：修正 未使用 T_{P2}	短縮倍率（倍） 部分 R_1 G	短縮倍率（倍） 部分 R_1 H	短縮倍率（倍） 全体 R_2 G	短縮倍率（倍） 全体 R_2 H	異常停止時の状況およびその原因（3.1 節参照）	備考
1	2	96	118	85	25	0.9	0.7	0.9	0.8	(c) 通信相手の誤り（Recv）	A
2	8	120	71	119	886	1.0	1.7	1.0	1.0	(d) 通信サイズの誤り（Bcast）	C
3	9	168	124	222	639	1.3	1.8	1.1	1.1	(c) 通信相手の誤り（Irecv）	C
4	520	108	174	253	45	2.3	1.5	2.0	1.4	(d) 通信サイズの誤り（Bcast）	A
5	24	80	62	260	53	3.3	4.2	2.4	2.7	(c) 通信相手の誤り（Irecv）	A
6	536	74	115	242	184	3.3	2.1	1.7	1.4	(c) 通信相手の誤り（Bcast）	C*
7	4	75	54	254	379	3.4	4.7	1.4	1.5	(c) 通信相手の誤り（Bcast）	B*
8	4	82	89	325	930	4.0	3.7	1.2	1.2	(a) 計算フォールト	A*
9	4	64	96	280	155	4.4	2.9	2.0	1.7	(c) 通信量多でデッドロック（Send）	C*
10	27	96	73	630	85	6.6	8.6	4.0	4.5	(c) 通信相手の誤り（Irecv）	A
11	15	129	135	1,022	1,586	7.9	7.6	1.5	1.5	(a) 計算フォールト	B*
12	40	150	64	1,387	202	9.3	21.7	4.5	6.0	(d) 通信サイズの誤り（Bcast）	B*

S: プログラム番号，E：イベント総数，$R_1 = T_{P1}/T'_{P1}$，$R_2 = (T_{P1} + T_{P2})/(T'_{P1} + T_{P2})$
備考欄の A〜C はプログラム作成者を表し，*は MPI-PD 未使用時にエラー出力がなかったことを表す．

グラマは各プロセスが最後に呼び出した通信命令から次に呼び出す予定であった通信命令までのソースコードに着目すればよく，特にすべてのプロセスが共通して実行する部分を見直せばよい．

なお，異常プロセスを指摘できた 14 個のうち 2 個は SPPC に基づいて自動生成した MPI プログラムである．これらの MPI プログラムは自動生成機構に依存するバグを含んでいて，誤った通信相手をノンブロッキング送信命令（MPI_Isend）で指定したため，異常停止していた．大規模かつ通信の組合せが不規則で複雑なプログラムに対しても，MPI-PD を利用することで約 5 分程度で異常プロセスを容易に判断できることが分かった．

3.1 節で述べた指摘洩れは，今回の実験では確認できなかった．通信相手の指定を誤った事例に対して，MPI-PD は本来の異常プロセスを含めて，デッドロックに参加するプロセスを異常プロセスであると指摘した．MPI-PD が指摘したプロセスは正常なプロセスを含む可能性があるため，それらを除いて異常プロセスを指摘できるようにすることは，課題の 1 つである．

5.3 デバッグ作業時間の短縮効果

5.2 節で異常プロセスを正しく指摘できた 12 個の MPI プログラム（ガウスの消去法）に対し，各々のプログラム作成者（被験者 A〜C）が MPI-PD を使用せずに，また第三者（被験者 G および H）が MPI-PD を使用してデバッグを試みた．

デバッグの作業時間は，デバッグ作業の開始からバグを含む異常プロセスを特定するまでに要した P1 時間（P1：特定フェーズ），さらに特定した異常プロセスに対して詳細なデバッグを行いバグを修正するまでに要した P2 時間（P2：修正フェーズ）に分けて計測した．なお，MPI-PD 使用者（第三者）は対象プログラムに対する知識が少なく，バグの修正が容易でないため P2 時間は計測しない．

また，演習で得られた MPI プログラムが小規模であるため，異常停止までのプログラムの実行時間は 1 秒未満であり，PC クラスタのハードウェア性能が P1 時間および P2 時間に大きく影響することはなかった．

表 6 に，各々の作業時間の計測結果を示す．ここで，T'_{P1} はそれぞれ MPI-PD 使用者の P1 時間を表し，T_{P1} は MPI-PD 未使用者の P1 時間を表す．T_{P2} は MPI-PD 未使用者の P2 時間を表す．また，短縮倍率 R_1 は $R_1 = T_{P1}/T'_{P1}$ から求めた．R_1 の値が大きいほど P1 時間に対する MPI-PD の短縮効果が大きいことを表す．同様に，短縮倍率 R_2 は $R_2 = (T_{P1} + T_{P2})/(T'_{P1} + T_{P2})$ から求め，R_2 の値が大きいほど全体の作業時間に対する MPI-PD の短縮効果が大きいことを表す．

表 6 から以下の 3 つの特徴を確認できる．

（1）$S = 1$ および $S = 2$ を除いて $R_1 > 1$.
（2）T'_{P1} のばらつき（標準偏差 33 および 36）に対して T_{P1} のばらつき（標準偏差 396）が大きい．
（3）$S = 5$, $S = 10$ および $S = 12$ において $R_2 > 2$.

まず，特徴（1）は MPI-PD を利用することで，多くの場合 P1 時間を短縮できたことを意味する．この理由として，MPI-PD の通信フォールト確認機能および異常プロセス検索機能の支援があげられる．通信フォールト確認機能が異常停止の原因を指摘し異常プ

表 7 MPI-PD および TotalView に関する感想
Table 7 Remarks on MPI-PD and TotalView.

項目	MPI-PD	TotalView
長所	G_{M1}: 異常停止の原因となった通信命令を特定可能 G_{M2}: 異常停止時の直前の通信状況が分かりやすい	G_{T1}: ソースコードビューワと連動してデバッグ可能 G_{T2}: ブレイクポイントを設定してステップ実行が可能
短所	B_{M1}: 計算部分の異常停止箇所が分かりづらい B_{M2}: 制御フローが分かりにくい	B_{T1}: MQG が複雑になりやすい B_{T2}: MQG で個々のメッセージを調べる必要がある

ロセス検索機能があらかじめ異常プロセスを絞り込むため，MPI-PD 使用者はイベントグラフを見るだけで異常プロセスやプロセス間の通信依存を知ることができる．これに対して，MPI-PD 未使用者はプロセスごとの独立したエラー出力（MPI 実装が出力）を基に異常停止時の全体状況を把握する必要がある．すなわち，どのプロセスが異常停止の連鎖を引き起こしたのか，もしくは巻き込まれたのかを各エラー出力を基に確認し，異常プロセスを特定する必要がある．これらの作業は MPI-PD が自動処理するため，$R_1 > 1$ となったと考えられる．なお，$S = 1$ および $S = 2$ はプログラムが 50 行程度と短いため，処理内容が単純であり，対象プログラムに詳しい作成者の方が P1 時間が短かった．

次に，特徴（2）は MPI-PD を利用することで P1 時間の変動を抑制できたことを意味する．以下，$T_{P1} > 600$（10 分）となった $S = 10$〜12 に着目する．

$S = 11$ および $S = 12$ において T_{P1} の値が大きい要因として，MPI プログラムが異常停止したときのエラー出力がなかった点があげられる．このため，MPI-PD 未使用者はプログラムが異常停止した箇所の手がかりを得ることができず，その箇所を特定する作業（printf などの出力関数の埋め込み）が必要となり，T_{P1} の値が相対的に大きくなった．

$S = 10$ はノンブロッキング通信命令を多用するプログラムである．あるノンブロッキング送信命令（MPI_Isend）において通信相手を誤って指定するバグを含んでおり，通信完了命令（MPI_Waitall）を呼び出したときにプログラムが異常停止する．MPI-PD 未使用者は，MPI_Waitall において通信の完了を指示した 24 個の MPI_Isend を個々に確認し，通信相手の指定を誤った MPI_Isend を特定する必要がある．ゆえに，T_{P1} の値が大きくなった．一方，MPI-PD 使用者は，MPI-PD が通信イベントごとに通信フォールトを確認するため，T'_{P1} の変動を抑えることができた．

最後に，特徴（3）は $S = 5$，$S = 10$ および $S = 12$ において，MPI-PD を利用することでバグ修正までの全体の作業時間を半分以下に短縮できたことを意味する．ゆえに，前述の $S = 10$ のように，異常プロセスを特定する作業に時間を要するが容易に修正できるプログラムに対しては，MPI-PD は P1 時間の短縮だけでなく全体の作業時間を短縮する観点からも有用であった．なお，P2 に既存のデバッガを利用することでさらに全体の作業時間を短縮できると考えられる．

表 8 MPI-PD のイベントグラフおよび TotalView の Message Queue Graph との比較
Table 8 Comparisons between MPI-PD event graph and TotalView Message Queue Graph (MQG).

比較項目	イベントグラフ	MQG
C1：異常プロセスの検索	あり	なし
C2：異常停止時の状況表示	あり	なし
C3：時間軸に沿った表示	あり	なし
C4：通信の完了状況の確認	なし	あり

5.4 既存のデバッガとの比較

$S = 1$〜12 に対し，MPI-PD 使用者（被験者 H）が TotalView を用いて再びデバッグを試みた．このときの被験者の感想を基に，MPI-PD および TotalView を比較した（**表 7**）．また，MPI-PD のイベントグラフおよび TotalView の MQG との比較を**表 8** に示す．

TotalView の MQG は，MPI-PD のイベントグラフと同様に通信依存を視覚化できるが，プログラマをバグ箇所へ誘導する機能（表 8 中の C1 および C2）を持たない．しかし，個々のメッセージの通信状況を確認することができ（C4），ステップ単位でその状況を更新できる．ただし，$S = 10$ のように，MPI_Waitall で多数のノンブロッキング通信命令を完了させるプログラムに対しては，大半のノンブロッキング通信命令を未解決状態（Pending）と判定してしまい，被験者 H は与えられた MQG のどの部分から通信依存を確認すればよいのか分からないような状況であった．

したがって，表 7 のように，MPI-PD は特定フェーズ P1 に関して肯定的な意見（G_{M1} と G_{M2}）を得たが，修正フェーズ P2 に関して否定的な意見（B_{M1} と B_{M2}）を得た．逆に，TotalView は P2 に関して肯定的な意見（G_{T1} と G_{T2}）を得たが，P1 に関して否定的な意見（B_{T1} と B_{T2}）を得た．

ゆえに，MPI-PD と TotalView を併用すればデバッグの作業効率を向上できると考えられる．

6. ま と め

本稿では，メッセージ通信ライブラリ MPI を用いた並列プログラムのデバッグを支援するツール MPI-PD について述べた．MPI-PD は，異常停止をともなうバグを含む MPI プログラムに対して，その通信依存を解析し異常プロセスを検索できる．また検索結果をイベントグラフとして視覚化できる．評価実験において，MPI-PD を用いることでバグを修正するまでの作業時間を最大で約 6 分の 1 に短縮できた．また，1 万行を超える複雑な MPI プログラムに対しても，修正すべきソースコード行をプログラマが容易に特定できた．

今後の課題としては，既存のデバッガとの連携や異常停止をともなわないバグへの対応があげられる．

謝辞 本研究の一部は，日本学術振興会未来開拓学術研究推進事業（JSPS-RFTF99I00903），栢森情報科学振興財団，日本電気（株）および科学研究費補助金基盤研究（C）(2)（14580374）の補助による．有益なご意見をいただいた査読者の方々に感謝いたします．

参 考 文 献

1) Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, *Int'l J. of Supercomputing Applications*, Vol.8, No.3/4 (1994).
2) 山田 剛：並列処理システムにおけるプログラムデバッギング，情報処理学会誌，Vol.34, No.9, pp.1170–1178 (1993).
3) Pallas GmbH: Vampir 2.0 User's Manual, Document Number VA20-UG-12 (1999). http://www.pallas.com/e/products/vampir/
4) LAM / MPI: XMPI—A Run/Debug GUI for MPI (2002). http://www.lam-mpi.org/
5) Kranzmuller, D., Grabner, S. and Volkert, J.: Event Graph Visualization for Debugging Large Applications, *Proc.SIGMETRICS Symp. on Parallel and Distributed Tools* (*SPDT'96*), pp.108–117 (1996).
6) Etnus, LLC.: TotalView Users Guide (2001). http://www.etnus.com/Download/TV.html
7) Gropp, W., Lusk, E., Doss, N. and Skjellum, A.: A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard, *Parallel Computing*, Vol.22, No.6, pp.789–828 (1996). http://www.mcs.anl.gov/mpi/mpich/
8) 三栄 武，高橋直久：PVM プログラムのための再演型デバッガの実現と評価，情報処理学会論文誌，Vol.37, No.7, pp.1308–1319 (1996).
9) Abramson, D., Foster, I., Michalakes, J. and Socič, R.: Relative Debugging: A New Methodology for Debugging Scientific Applications, *Commun. ACM*, Vol.39, No.11, pp.69–77 (1996).
10) Netzer, R.H., Brennan, T.W. and Damodaran-Kamal, S.K.: Debugging Race Conditions in Message-Passing Programs, *Proc. SIGMETRICS Symp. on Parallel and Distributed Tools* (*SPDT'96*), pp.31–40 (1996).
11) Cownie, J. and Gropp, W.: A Standard Interface for Debugger Access to Message Queue Information in MPI, *Proc. 6th European PVM/MPI Users' Group Meeting*, pp.51–58 (1999).
12) Butler, R., Gropp, W. and Lusk, E.: A Scalable Process-Management Environment for Parallel Programs, *Proc. 7th European PVM/MPI Users' Group Meeting*, pp.168–175 (2000).
13) 山本裕己，譚 林，藤本典幸，萩原兼一：一対一プロセッサ間通信の一括化を考慮したタスクスケジューリングアルゴリズム，情報処理学会研究報告，2002-MPS-38, pp.24–28 (2002).
14) Boden, N.J., Cohen, D., Felderman, R.E., Kulawik, A.E., Seitz, C.L., Seizovic, J.N. and Su, W.-K.: Myrinet—A Gigabit-per-Second Local-Area Network, *IEEE Micro*, Vol.15, No.1, pp.29–36 (1995).

(平成 14 年 1 月 29 日受付)
(平成 14 年 5 月 8 日採録)

置田 真生

平成 13 年大阪大学基礎工学部情報科学科卒業．現在，同大学大学院基礎工学研究科修士課程在学中．並列ソフトウェア全般に興味を持つ．

伊野 文彦（正会員）

平成 12 年大阪大学大学院基礎工学研究科修士課程修了．平成 14 年同大学院同研究科博士課程退学．同年，同大学助手．並列計算機のソフトウェア開発環境に関する研究に従事．

藤本　典幸（正会員）

平成 4 年大阪大学基礎工学部情報工学科卒業．平成 6 年同大学大学院基礎工学研究科修士課程修了．平成 9 年同大学院基礎工学研究科博士課程単位取得退学．平成 9 年同大学助手．平成 14 年より同大学助教授．工学博士．並列アルゴリズム，並列言語の処理系・開発環境等に興味を持つ．

萩原　兼一（正会員）

昭和 49 年大阪大学基礎工学部情報工学科卒業．昭和 54 年同大学大学院基礎工学研究科博士課程修了．工学博士．同大学基礎工学部助手，講師，助教授を経て，平成 5 年奈良先端科学技術大学院大学教授．平成 6 年より大阪大学教授．平成 4〜5 年文部省在外研究員（米国メリーランド大）．現在，並列処理の基礎および応用に興味を持っている．