PVM 3 ユーザーズガイド
& リファレンスマニュアル
日本語版

Al Geist
Adam Beguelin
Jack Dongarra
Weicheng Jiang
Robert Manchek
Vaidy Sunderam
pvm@msr.epm.ornl.gov
(訳 村田 英明*)

1995 年 2 月 5 日

要 旨

本書は,PVM バージョン 3.1 のユーザーズガイド及びリファレンスマニュアルである．PVM の概要について述べるとともに, バージョン 3 の入手方法, インストール方法及び使用方法について述べる.

PVM(**P**arallel **V**irtual **M**achine) は, ネットワークで接続された異機種の並列コンピュータ及び逐次コンピュータを, 単一の大きな並列計算資源に統合するパッケージソフトウェアである.

PVM ソフトウェアシステムは, デーモンとユーザライブラリの 2 つから成る．デーモンは誰もがインストールすることができる. ユーザライブラリは, 他のマシンでのプロセス初期化, プロセス間通信, 及びマシン設定の変更のためのルーチン群を提供する.

本書では, 今回のリリースでの新しい特徴について詳しく述べるとともに, バージョン 3 の内部処理及びユーザインターフェース仕様についても述べる．現在広く普及しているプログラミングパラダイムの中から,PVM がサポートしているものについて C 及び FORTRAN の例を示す． また, 負荷分散, 性能, フォールトトレランスの各問題について議論し, PVM プログラムの基本的なデバッグ方法について示す.

付録には, PVM 2.4 ルーチンと PVM 3 ルーチンとの対応表及び PVM 3.1 の man ページを示した.

[この ORNL 報告は, ドラフト版である．記述例, 動的グループに関する議論, 及び PVM 3 内部の詳細の追加を予定している.]

*murata@sgb.kobe.mhi.co.jp

PVM 3 ユーザーズガイド & リファレンスマニュアル 日本語版について

1994 年 2 月 5 日
村田 英明

本書は, PVM 3 User's guide and reference manual として配布されているものの日本語訳です. 日本での本書の配布に関して, 原著者の一人, Al Geist 氏より了解を得ました. 配布条件については,

```
The authors names, affiliations, and the notice of who funded
the PVM research should remain in all distributions.
```

とのことでしたので, 本ページを含む限り再配布に問題はないと思います.

日本語訳は, 村田 (三菱重工業 (株) エレクトロニクス事業部, murata@sgb.kobe.mhi.co.jp) が行いました. できる限り原文に忠実に翻訳するよう努力しましたが, 不備な点については是非とも御示唆願います. また, 日本語版の著作権は, 村田が有するものと致します.

ここで, 高橋栄一氏 (電子技術総合研究所, etakahas@etl.go.jp) に, 本書を非常に注意深く読んで頂き多くの不備な点を修正して下さったことを感謝いたします.

原著名

PVM 3 USER'S GUIDE AND REFERENCE MANUAL

原著者

Al Geist	Oak Ridge National Laboratory, Oak Ridge, TN 37831-6367
Adam Beguelin	Carnegie Mellon University and
	Pittsburgh Supercomputing Center, Pittsburgh, PA 15213-3890
Jack Dongarra	Oak Ridge National Laboratory, Oak Ridge, TN 37831-6367
Weicheng Jiang	University of Tennessee, Knoxville, TN 37996-1301
Robert Manchek	University of Tennessee, Knoxville, TN 37996-1301
Vaidy Sunderam	Emory University, Atlanta, GA 30322

原著発行日

1993 年 5 月

NOTICE (PVM 3 ソースコードより抜粋)

Neither the Institutions (Emory University, Oak Ridge National Laboratory, and University of Tennessee) nor the Authors make any representations about the suitability of this software for any purpose. This software is provided "as is" without express or implied warranty.

PVM 3 was funded in part by the U.S. Department of Energy, the National Science Foundation and the State of Tennessee.

国内での PVM に関する情報について

国内での PVM に関する情報として, 現在筆者が利用させて頂いているものを簡単にご紹介します.

1. メイリングリスト

PVM や並列プログラミングに関する話題のメイリングリストが運営されています. 本文に guide とタイプした電子メイルを pvm-request@etl.go.jp へ送ると, PVM メイリングリストの案内が折り返し送付されます.

2. Anonymous FTP サイト

PVM の最新のソースコード並びにドキュメントは, 国内の多くの anonymous ftp サイトにあります. ここでは電総研の ftp サイトをご紹介します.

```
ftp://etlport.etl.go.jp/pub/pvm
```

3. WWW サーバ

東京大学の安東氏の作成された WWW サーバをご紹介します.

```
http://www.race.u-tokyo.ac.jp/PVM
```

改訂履歴

番号	日付	改訂内容
第 0 版	1994 年 8 月 24 日	PVM 3 User's guide and reference manual の日本語訳として作成
第 1 版	1995 年 2 月 5 日	第 0 版に, 加筆修正

もくじ

1　イントロダクション

本書は,PVM(**P**arallel **V**irtual **M**achine) バージョン 3 のユーザーズガイドであり,PVM を利用するための基本的な情報とサンプル例について述べている．付録には,PVM3.1 の全機能, エラー状態, クイックリファレンス並びにインストール手続きをまとめた.

PVM 3 は, ネットワークに接続された異機種 UNIX コンピュータ群を, 単一の並列コンピュータとして利用することを可能にするソフトウェアシステムである．これによって, 多数のコンピュータの持つ計算パワーを, 一つの大規模計算問題に結集して処理を行うことが出来る.

PVM の開発は,Oak Ridge 国立研究所 (ORNL) にて,1989 年の夏に始まった．現在継続中の研究プロジェクトには,Emory 大学の Vaidy Sunderam, ORNL の Al Geist,Tennessee 大学の Robert Manchek, Carnegie Mellon 大学及び Pittsburgh Supercomputing Center の Adam Beguelin, Tennessee 大学の Weicheng Jiang, ORNL 及び Tennessee 大学の Jack Dongarra らが参加している．この研究プロジェクトは, 米国エネルギー省, 国立科学基金及び Tennessee 大学から資金援助を受けている.

PVM プロジェクトは, その実験的性質ゆえの副産物として, 科学者のコミュニティあるいはその他の分野の研究者に役立つようなソフトウェアを, これまで作り出してきた. PVM 3 は無料で配布されており, 先端科学分野における大規模計算のためのソフトウェアとして世界中で利用されている.

PVM の下では, 逐次コンピュータ, 並列コンピュータ及びベクトルコンピュータの集まりをユーザが定義し, 一つの大きな分散メモリ型コンピュータとして表す．本書では, この論理的な分散メモリ型コンピュータを**バーチャルマシン** *(virtual machine)* と呼ぶこととする．また, コンピュータの集まりの一つ一つを**ホスト** *(host)* と呼ぶこととする．PVM は, バーチャルマシンにおいて自動的にタスクを起動する機能を備えており, タスク間の通信及び同期を実現する．タスクは PVM における計算の単位であり,UNIX におけるプロセスに類似したものである．タスクは, 通常 UNIX のプロセスにより実装されることが多いが, 必ずしもそうである必要はない．ユーザは C または FORTRAN を用いてアプリケーションを記述する．その際, ほぼ全ての分散メモリ型コンピュータで共通なメッセージパッシングライブラリを利用して並列化する．アプリケーションを構成する複数のタスクは協調して動作し並列に計算をおこなう.

PVM は, アプリケーション, マシン及びネットワークレベルでの異機種間利用をサポートする．言い替えると,PVM の下では, アプリケーションを構成するタスクは, 問題に最も適したアーキテクチャを利用することができる．PVM は, 異なるコンピュータ間の整数あるいは浮動小数点数の表現の違いを吸収するための, データ変換を扱うことができる．そして,PVM は多様なネットワークで接続されたバーチャルマシンを実現する.

PVM ソフトウェアシステムの構成は, 大きく 2 つに分けられる．一つはデーモンであり,*pvmd3* と呼ばれる．これは *pvmd* と略されることもある．デーモンは, バーチャルマシンを構成する全てのコンピュータ上に常駐する (デーモンの例として *sendmail* が挙げられる．sendmail は,UNIX システムに常駐し, メイルの送信及び受信を扱う)．ユーザは, ログイン可能でさえあればどんなコンピュータにも,pvmd3 をインストールすることができる. PVM アプリケーションを実行する場合, まず最初にユーザはどれか一つのコンピュータで

pvmd3 を起動する．次にこの pvmd3 は, ユーザが定義したバーチャルマシンを構成するコンピュータそれぞれにおいて順次 pvmd3 を起動する．最後に, どれか一つのコンピュータに表示された UNIX プロンプトに対してコマンドを入力することにより, PVM アプリケーションを実行する．複数のユーザは, 互いにコンピュータをオーバーラップさせてバーチャルマシンを構成でき, また, 各ユーザは一人で複数の PVM アプリケーションを同時に実行することも可能である．

PVM ソフトウェアシステムを構成するもう一つは,PVM インターフェースルーチンのライブラリである．これは `libpvm3.a` と呼ばれる．ライブラリは, メッセージパッシング, プロセスの生成, タスクの協調, 及びバーチャルマシンの再構成のためのルーチンを提供する．PVM を利用するためには, アプリケーションプログラムとこのライブラリを必ずリンクする必要がある．

2 PVM 3 の特徴

PVM バージョン 3 は, バージョン 2 に比べて多くの改良がなされている．以下に,PVM 3 で新たに可能となった特徴について述べる．

2.1 ユーザインターフェースの改良

PVM 開発プロジェクトチームがユーザから得たフィードバックの一つとして, 幾つかのマルチプロセッサコンピュータではベンダーから提供されるライブラリと PVM 2.x ルーチンとの間に名前の衝突があることが分かった. 例えば,PVM 2.4 ルーチンである `barrier()` は, 幾つかのマルチプロセッサコンピュータでは (機能的には僅かな違いがあるものの) 既に存在していた． この名前の衝突を回避するために,pvm 3 のユーザルーチンは全て `pvm_` (C の場合) 及び `pvmf`(FORTRAN の場合) で始まる名前が付けられている． その他にも, インターフェースには新たな引数と特色が組み込まれており, アプリケーション開発者にとって, 更にフレキシブルなものとなっている．

ユーザインターフェースは完全に変更されたものの,PVM 2.4 から PVM 3.x への変更は直接的かつ容易である．付録 B には,PVM 2.4 と PVM 3.x の対応表を示した．アプリケーションの変更を望まないユーザのために,PVM 2.4.2 は `netlib@ornl.gov` に用意しておく予定である．

2.2 整数タスク識別子

PVM 3 において登録された全てのプロセスは, 整数タスク識別子によって表される．以前のバージョンでは, コンポーネント名とインスタンス番号の対によってプロセスを表しており, この点が変更された．これ以降本書では, このタスク識別子のことを `tid` と表す．tid は, PVM におけるプロセスを識別するための最も基本的かつ効率的な手段である．なぜならば,pvmd がバーチャルマシン全体で一意な tid を定めるため, ユーザがこれを選ぶことはできない． PVM 3 の幾つかのルーチンは tid を返り値としており, ユーザアプリケーションは同一システム内の他のプロセスを識別することができる． tid を返り値とするルーチンには,pvm_mytid(), pvm_spawn(), pvm_parent(), pvm_bufinfo() 及び pvm_gettid() がある．グループ化すると効率は落ちるが, 名前とインスタンス番号によるプロセスの識別が

可能となる．ユーザがプロセスに対して所属するグループの名前を定義すると,PVM はグループ内で一意なインスタンス番号を返す.

2.3 プロセス制御

PVM には, ユーザプロセスを PVM タスクにするルーチン及び PVM タスクを再びユーザプロセスにするルーチンがある．また, バーチャルマシンにホストを追加・削除するルーチン,PVM タスクを起動・終了するルーチン, 他の PVM タスクにシグナルを送信するルーチン, 及びバーチャルマシンの設定とアクティブな PVM タスクに関する情報を得るルーチンがある.

2.4 フォールトトレランス

もしあるホストに障害が発生すると,PVM はそのホストを自動的に検出し, バーチャルマシンから削除する．アプリケーションは, 各ホストの状態を PVM に要求することができ, 代替のホストを追加することが可能である．ホストの障害に対するアプリケーションの復旧は, 全てアプリケーション開発者に任されている．ホストの障害によって強制終了させられたタスクに対して, PVM は自動的な復旧を試みることはない.

2.5 ダイナミックプロセスグループ

PVM 3 では, ダイナミックプロセスグループが実装されている．本実装では, プロセスは複数のグループに所属することができ, 実行中にいつでもグループを変更できる.

ブロードキャストやバリア同期といった論理的なタスクのグループを扱うための関数は, ユーザが明示的に定義したグループ名を引数としてとることができる．グループへの所属と離脱のためのルーチンが提供される．タスクは他のグループについての情報を問い合わせることもできる.

2.6 シグナル

PVM は, 他の PVM タスクにシグナルを送るための 2 つの方法を提供する．一つは,UNIX のシグナルを他のタスクへ送る方法である．もう一つは, あるイベントに対してユーザが定義したタグを持つメッセージを, タスクの集合に対して通知する方法である．アプリケーションは, このメッセージをチェックすることができる．この通知イベントには, タスクの終了, ホストの削除 (あるいは障害), 及びホストの追加がある.

2.7 通信

PVM は, タスク間でのメッセージのパックと送信を行うルーチンを提供する．PVM モデルでは, 任意のタスク間でメッセージを送ることができること, 並びにメッセージのサイズ及び数に制限はないことを仮定している．全てのホストの物理メモリは有限であり, 潜在的なバッファの大きさを制限する．一方, 通信のモデルでは, マシン固有のメモリ制限には束縛されず, 十分なメモリを利用可能であると仮定している．PVM の通信モデルは, 非同期ブロック送信, 非同期ブロック受信及び非ブロック受信の 3 つの関数を提供する．ここで我々の定義では, ブロック送信は, 送信バッファが再使用のために開放された時点でリター

ンし, 受信側の状態には依らないものとする. 非ブロック受信は, 到着したデータとともにリターンするか, あるいはデータ未着の場合それを示すフラグとともに直ちにリターンするものとし, 一方ブロック受信は, データが受信バッファに存在する時のみリターンするものとする. これらの 1 対 1 通信関数に加えて, モデルはタスク集合に対するマルチキャスト, 並びにユーザが定義するタスクのグループへのブロードキャストをサポートする. 送信元を指定する, あるいは無視する際には, ワイルドカードを用いることができる.

PVM モデルは, メッセージ順序の保存を保証している. まず, タスク 1 がタスク 2 へメッセージ A を送信し, その後タスク 1 がタスク 2 へメッセージ B を送信すると, メッセージ A はメッセージ B よりも早く到着する. 更に, タスク 2 が受信を行う前に両方のメッセージが到着した場合, ワイルドカードを指定した受信は常にメッセージ A を返す.

メッセージバッファは動的に割り当てられる. 従って, 送信あるいは受信可能なメッセージの大きさは, ホストで利用可能なメモリ量によってのみ制限される.

2.8 マルチプロセッサの統合

当初,PVM は, ネットワークで接続された複数のコンピュータを単一の論理的なコンピュータとして統合することを目的として開発された. ここで,iPSC860 のような分散メモリ型マルチコンピュータでは, ホストプロセッサあるいは特殊なノードプロセッサのみがネットワークに接続され, その他のプロセッサは内部通信路を用いて互いに通信する. PVM 2.4 において, このようなマシンを利用する場合には, ホストプロセッサで動作する特別な PVM プログラムを記述する必要があった. このプログラムは,PVM でのメッセージのフォーマットとマシン固有のメッセージのフォーマットとの相互変換を行い, 必要に応じて PVM ルーチンとマシン固有の通信ルーチンとのメッセージの相互転送を行う. また, ホストプロセッサ自身は計算を行わないのが普通であった.

PVM 3 では,UNIX ソケット及び TCP/IP ソフトウェアへの依存性が緩和された. 例えば,PVM 3 で記述されたプログラムは,SUN のネットワーク,Intel の Paragon のノードのグループ, ネットワークで接続された複数の Paragon, あるいは世界中に分散した異機種マルチコンピュータの組合せのいずれであってもで実行可能である. PVM 3 では, マルチプロセッサ内の通信は, マシン固有の通信ルーチンを使用するように設計されている. 同一マルチプロセッサ内のノード間で交換するメッセージは直接転送され, 一方ネットワーク上の他のマシンへのメッセージは, マルチプロセッサ上でユーザが起動した PVM デーモンに対して送られ, 更に他のマシンへと転送される.

現在,Intel の iPSC860 及び Paragon が PVM 3.1 に統合されており, ノード間通信に IntelNX メッセージパッシングルーチンを利用している. Cray,Convex,SGI,DEC,KSR 及び IBM の各社は, それぞれ自社のマルチプロセッサマシンの発売時には,PVM 3 互換性を提供する予定である. 今後の PVM 3.x のリリースでは, より多くのマルチプロセッサが加えられるであろう.

3 PVM の入手とインストール

PVM の入手とインストールには, 一人の人間で十分である. そして一旦インストールすれば, 同じ組織のだれでもが PVM を利用することができる. PVM では, 利用可能なコン

ピュータのアーキテクチャ名を *ARCH* で表す．表 1では,PVM 3.1 でサポートされるアーキテクチャを ARCH 名とともに示している．

3.1 PVM の入手

サイトに PVM がインストールされていない場合,PVM ソフトウェアを入手し, インストールする必要がある．PVM のインストールに, 特別な権限は不要である．ログイン可能でさえあれば, 誰でもがインストールできる．ただし, 礼儀としてシステム管理者に知らせておくべきであろう．

次のステップは, ソフトウェア及びドキュメントの入手である．本ユーザーズガイド,PVM 3 のソースコード, ユーザ提供によるサンプル例及び PVM 関連ソフトウェアは, 全て *netlib* から入手可能である． netlib とはインターネットが提供するソフトウェア配布サービスであり, 入手のための 2 つの方法がある．第一の方法は,*xnetlib* と呼ばれるツールを利用する．xnetlib は X ウィンドウシステムのユーザインターフェースを備えており,netlib にあるソフトウェアについて問い合わせやブラウジング, あるいは選択したソフトウェアを自動的にユーザの手元のコンピュータに転送する, といったことが可能である． xnetlib を入手するには,`send xnetlib.shar from xnetlib` と書かれた電子メイルを, アドレス `netlib@ornl.gov` へ送るか, または `cs.utk.edu` の `pub/xnetlib` から anonymous ftp を行えば良い．

第二の方法は, 電子メイルを用いる． PVM ソフトウェアを入手するには, `send index from pvm3` と書かれた電子メイルを, アドレス `netlib@ornl.gov` へ送れば良い． 自動電子メイル応答により, ファイルのリストと入手の手順が返送される．この方法の有利な点は, インターネットに電子メイルでアクセスできる者なら誰でも入手できることにある．

3.2 ソースコードの展開

ソースファイルの大きさは展開後で約 1Mbyte であり,shar または uuencode/compress された tar 形式で得られる. shar 形式であれば,$HOME ディレクトリでまず以下のようにタイプする.

```
% sh pvm3_shar
```

また,tar 形式であれば以下のようにタイプする.

```
% uudecode pvm3.1.tar.z.uu
% uncompress pvm3.1.tar.Z
% tar xvf pvm3.1.tar
```

以上の操作により,pvm3 というディレクトリが作成される.

3.3 構築

$HOME ディレクトリに pvm3 ディレクトリを展開したら,PVM 構築の準備ができたことになる. PVM のソースは, 各マシンに対応する幾つかのディレクトリと makefile からなる. 基本となる幾つかの makefile は, 新しいマシンでの PVM の構築の手助けとなる.

PVM を実行するマシンの .cshrc の中に "source pvm3/lib/cshrc.stub" の 1 行を追加するこのスタブは, 必ずパスの設定よりも後ろに置く必要がある. このスタブは, 自動的にマシンのタイプを決定し,PVM の存在する位置をパスに加える.

pvm3 ディレクトリで make とタイプすれば, アーキテクチャ毎に異なるモジュールの構築を自動的に開始する. makefile はマシンのアーキテクチャを自動的に識別し, `pvmd3`, `libpvm3.a`, 及び `libfpvm3.a` 構築を行う. ダイナミックグループライブラリ `libgpvm3.a` を構築する場合には,makefile の中のデフォルトの構築ターゲットリストに g を加えるか,make g とタイプする. これらのファイルは全て,`pvm3/lib/ARCH` に置かれる.

Intel Paragon または iPSC/860 上で PVM を構築する場合は, 更に以下の手順が必要である. 上記ライブラリのノード限定バージョンは,`libpvm3e.a`, `libfpvm3pe.a` として作られる. 従って,SUN または IRIS 上のクロスコンパイラを使用するなら,`make PGON_x` または `make I860_x` とタイプしなければならない.

3.4 インストール

PVM は, デフォルトの実行ファイル格納場所として,`$HOME/pvm3/bin/$ARCH` を参照する. 例えば, あるユーザ PVM アプリケーションが,sunny という SPARCstation 上で *foo* タスクを実行する場合,sunny 上に `$HOME/pvm3/bin/SUN4/foo` が存在しなくてはならない. hostfile に異なる検索パスを指定することで, デフォルトの検索パスを変更することができる.

PVM が `/usr/local` といった一つの場所にインストールされている場合, 各ユーザは実行ファイルを置くための `$HOME/pvm3/bin/$ARCH` を作成し, シンボリックリンクを `pvm3/lib` に張る必要がある.

```
% ln -s path_to_pvm3_lib ~/pvm3/lib
```

4 PVM の設定と起動

本章では, バーチャルマシンを構成するための初期設定を定義する際に必要となる情報について述べる. また,PVM の起動手順の対話型コマンドモニタについても併せて述べる. 対話型コマンドモニタは PVM コンソールまたは pvm とも略される.

4.1 PVM の起動

PVM の起動とは,pvm と pvmd3 の双方を実行することを意味する. 引数を何も指定しない場合は, これらのコマンドはローカルホストの上で pvmd3 を起動する. コンソール (pvm) は,PVM の稼働するマシンであればどこででも, 起動と停止を繰り返すことができる. しかしながら, 普通はユーザはただ一つのコンソールを起動する. コンソールは, バーチャルマシンに対して対話的にホストを追加・削除することができ, また対話的に PVM プロセスの起動・停止を行うことができる.

PVM コンソールはプロンプトとして

```
pvm >
```

を出力し, コマンドの入力待ちとなる. 以下に利用できるコマンドを示す.

help または ? 全ての対話的コマンドに関する情報を得る. help の後に続けてコマンド名を指定すると, そのコマンドに関する全てのオプションとフラグの一覧を出力する.

version libpvm のバージョンを表示する.

conf バーチャルマシンの設定を表示する. 表示内容は, ホスト名,pvmd タスク ID, アーキテクチャタイプ, 最大フラグメントサイズ, 及び相対速度比である.

add add の後に続いて指定された一つまたはそれ以上のホストを, バーチャルマシンに追加する.

delete delete の後に続いて指定された一つまたはそれ以上のホストを, バーチャルマシンから削除する. ホストで実行されている PVM プロセスは全て失われる.

mstat 指定するホストの状態を示す.

ps -a バーチャルマシンで現在実行中のプロセスの一覧を表示する. 表示内容には, 各プロセスの位置, タスク ID, 及び親のタスク ID がある.

pstat 一つの PVM プロセスに関する状態を表示する.

spawn PVM アプリケーションを起動する.

kill PVM プロセスを終了させる.

reset コンソール以外の全ての PVM プロセスを終了させるとともに,PVM の内部テーブル及びメッセージ待ち行列をリセットする. デーモンはアイドル状態で残る.

quit デーモン及び実行中の PVM ジョブより離れ, コンソールを終了する.

```
# configuration used for my run
sparky
azuru.epm.ornl.gov
thud.cs.utk.edu
sun4
```

図 1: バーチャルマシンを設定するホストファイル

halt コンソールを含む全ての PVM プロセスを終了させ,PVM をシャットダウンする．デーモンも全て終了する．

PVM を実行する最もポピュラーな方法は 2 つある．一つは, まず pvm を起動した後手動でホストを追加していく方法である．もう一つは,pvmd3 をホストファイルとともに起動した後必要ならば pvm を起動する方法である．

ただ一つの端末またはウィンドウの環境で作業している場合のために, 幾つかのオプション機能が用意されている． ユーザは, まず pvm を起動し必要な設定を行った後 pvm を終了する．次に pvmd3 をバックグラウンドで実行する．PVM をバックグラウンドで実行するには以下のようにタイプする．

```
% pvmd3 hostfile &
```

この方法では, ホストファイル[1] に pw が設定してある場合に上手く動作しない． なぜならば,UNIX はパスワードの入力を要求するにもかかわらず,UNIX 自身がバックグラウンドジョブへのユーザ入力を扱うことができないためである． この問題には以下のように対処する．

```
% pvmd3 hostfile
```

設定後,PVM は [t80040000] ready と表示する．このとき,control-Z と bg を続けてタイプすれば, PVM をバックグラウンドにできる．

シャットダウンするには,PVM コンソールのプロンプトで halt をタイプする．

4.2 バーチャルマシンの設定

PVM をインストールする者は, 一つのサイトに一人で十分である．けれども各 PVM ユーザはそれぞれ, 個別にホストファイルを持たなくてはならない．このホストファイルに, ユーザは自分独自のバーチャルマシンを記述する．

ホストファイルの形式は非常に簡単で, 一行に一つホスト名を記述した一覧表である．先頭行のホストは, 最初に PVM を起動するホストでなくてはならない． 空行及び # で始まる行は無視される．これによって, ホストファイルにドキュメント性を持たせることができる．また, 種々のホストをコメントアウトすることで, 初期設定を簡単に変更できる (図 1)．

[1] 訳注:PVM ホストを設定するためのファイル． UNIX の /etc/hosts あるいは.rhosts とは異なる． “pw” については次のページを参照.

各行のホスト名の後には, 幾つかのオプションを指定することができる. オプションは空白で区切る.

lo = userid ユーザがログイン名を指定する. これが無い場合は, 起動をかけたマシンでのログイン名を用いる.

pw ユーザにパスワードの入力を促す. リモートシステムに対して異なるユーザ ID と異なるパスワードを持っていた場合に有用である. デフォルトでは,PVM は rsh を利用してリモートの pvmd を起動するが,pw が指定されていた場合には rexec() を利用する.

dx = location_of_pvmd ユーザがデフォルト以外の場所にある pvmd を指定する. 個人的な pvmd のコピーを利用する場合に有用である.

ep = paths_to_user_executables ユーザが要求実行ファイルの検索パスを指定する. 複数のパスを指定する場合はコロン (:) で区切る. `ep=` の指定が無い場合は, `$HOME/pvm3/bin/ARCH` の中にアプリケーションタスクを探す.

一連のホストについて上記オプションのデフォルトを変更したい場合, ホスト名のフィールドに * を指定した一行を追加する. その行以降の全てのホストのデフォルトを上書きする.

初期設定には含めたくないが, 後から追加したいホストがある場合, 行頭を & で始める. 典型的なホストファイルとオプションの例を図 2に示す.

4.3 起動に関するトラブルシューティング

PVM は, 起動時に問題が発生した場合, スクリーン上か, またはログファイル `/tmp/pvml.<uid>` にエラーメッセージを出力する. このエラーメッセージを解釈し, 問題の解決に役立てる方法を説明する.

```
[t80040000] Can't start pvmd
```

というメッセージが出力された場合, たいていは `tmp/pvml.<uid>` が原因である. このファイルは認証のために存在し,PVM の実行中は必ず存在する. pvmd3 が手動で終了させられた場合, 時としてこのファイルが残り, PVM の起動を邪魔することがある. このファイルを消せば良い.

その他に, このメッセージが出力される理由としては,PVM が未だインストールされていない, あるいは `.rhosts` に問題があるといった点が挙げられる. 特定のホストに対して `pw` オプションを設定していない場合には, リモートホストの `.rhosts` には,PVM の起動をかけたホストの名前が含まれていなくてはならない.

```
[t80040000] Login incorect
```

というメッセージが出力された場合, おそらくはローカルマシンと同じユーザ名のアカウントがリモートマシンに存在しないことを意味している. `lo=` オプションを付けることで対処できる.

```
# 注釈は#で始まる（空行は無視される）
getsw
ipsc dx=/usr/geist/pvm3/lib/I860/pvmd3
ibm1.scri.fsu.edu lo=gst pw

# オプションのデフォルト値の変更は*で始める
* ep=$sun/problem1:~/nla/mathlib
sparky
# azuru.epm.ornl.gov
midnight.epm.ornl.gov

# オプションのデフォルト値を更に変更
* lo=gagest pw ep=problem1
thud.cs.utk.edu
speedy.cs.utk.edu

# 後から追加したいマシンには&を付ける
&sun4 ep=problem1
&castor dx=/usr/local/bin/pvmd3
&dasher.cs.utk.edu lo=gageist
&elvis dx=~/pvm3/lib/SUN4/pvmd3
```

図 2: PVM ホストファイルのオプション例

これら以外の変なメッセージが出力された場合, 次にチェックするのは .cshrc である. .cshrc は PVM を起動する際のインターフェースとなるので, .cshrc の中ではユーザとの間にいかなる I/O も行わないことが重要である. 例えば, ログイン時に他のログイン中のスタッフを表示させたかったら, .login スクリプト内で行うか, あるいは,“if” ステートメントを用いて対話的にログインした時のみ表示を行うようにするべきである. 決して csh のコマンドスクリプトで実行させてはならない. 以下に, 例を示す.

```
if ( { tty -s } && $?prompt ) then
        echo terminal type is $TERM
        stty erase '^?'  kill '^u' intr '^c' echo
endif
```

4.4 PVM アプリケーションのコンパイル

PVM を呼び出す C プログラムは libpvm3.a とリンクする必要がある. ダイナミックグループ機能を利用する場合には, 更に libgpvm3.a を libpvm3.a よりも前にリンクしなく

てはならない.

PVM を呼び出す FORTRAN プログラムには libfpvm3.a と libpvm3.a の両方をリンクする必要がある. ダイナミックグループ機能を利用する場合には,libfpvm3.a,libgpvm3.a 及び libpvm3.a をこの順序でリンクしなくてはならない.

サンプルの makefile は,PVM のソースコードのディレクトリ pvm3/examples にある. この makefile は C 及び FORTRAN のアプリケーションと PVM ライブラリのリンク方法について示している. またファイルの先頭部分には, 幾つかのアーキテクチャで追加してリンクする必要のあるライブラリについても示している.

Intel Paragon のノードで実行する PVM プログラムは,libpvm3.a あるいは libfpvm3.a の代わりに libpvm3pe.a あるいは libfpvm3pe.a とリンクしなくてはならない.

4.5 PVM アプリケーションの実行

一度 PVM を起動してしまえば, 設定に含まれるどんなマシンからでも,UNIX コマンドプロンプトより PVM ルーチンを利用するアプリケーションを起動することが可能となる. PVM を起動したマシンとアプリケーションを起動するマシンが同じである必要はない.

タスクの標準出力及び標準エラー出力は,PVM を起動したホスト上のログファイル /tmp/pvml.<uid> に書き込まれる. 書き込まれた内容をスクリーン上に表示させたい場合は,tail -f /tmp/pvml.<uid> を実行すればよい. 手動で起動した PVM タスクの標準出力及び標準エラー出力は, 全てスクリーンに表示される.

PVM が存在する限り一連のアプリケーションを次々と実行することができるので, 各アプリケーション毎に PVM を起動し直す必要はない. けれども, アプリケーションがクラッシュした場合には,PVM をリセットすることは可能である. 次章以降では,PVM アプリケーションの書き方について述べる.

5 ユーザインターフェース

全てのルーチンのアルファベット順の一覧表を付録 A に示す. 付録 A では, 各ルーチンの引数, エラーコード及び考えられるエラーの原因について詳細に述べている. 一覧表の各項目には,C 及び FORTRAN からの使用例を併せて示している.

PVM 3.1 ルーチンの簡単な要約は, クイックリファレンスとして付録 D に示す. このガイドはハンディリファレンスカードとして折り畳めるように作られている.

本章では,PVM 3.1 ユーザライブラリにおける各ルーチンの要約を述べる. 各ルーチンを機能毎にまとめた構成となっている. 例えば実行時設定の節では, まず, 実行時設定の目的について議論した後, その機能の効果的な利用法について述べ, 最後にその機能を実現する C 及び FORTRAN の PVM ルーチンを示す.

PVM 3 では, 全ての PVM タスクはローカルな pvmd から整数を与えられ, これによって識別される. 以降では, この整数を tid と呼ぶこととする. tid は UNIX システムにおけるプロセス ID に似ているが, バーチャルマシンにおけるプロセスの位置が tid の中にエンコードされている点が異なる. エンコードすることにより, 効率の良い通信ルーティング及びマルチプロセッサの統合を実現する.

PVM ルーチンは全て C で記述されている． C++ アプリケーションは PVM ライブラリとリンク可能である．FORTRAN アプリケーションは,PVM 3 ソースに含まれる FORTRAN 77 インターフェースを通じて,PVM ライブラリを呼び出すことができる．このインターフェースは, 引数を FORTRAN 77 で参照できるように変換し, 必要ならばその値を下位の C 関数に渡す． また,FORTRAN からの C 関数の呼び出し形式については, 従来からある種々の形式を考慮した.

5.1 プロセス制御

```
int tid = pvm_mytid( void )
```

```
call pvmfmytid( tid )
```

最初の呼び出しでは, 呼び出したプロセスを PVM に登録する．その際, pvm_spawn() で起動されたプロセスでなければ, 一意な tid を生成する． 何回でも呼び出すことができ, 呼び出しプロセスの tid を返す． pvm_mytid() は, 他の全ての PVM 呼び出しに先立って呼び出されなくてはならない．PVM を起動しないままアプリケーションが pvm_mytid() を呼び出すと, 呼び出されたルーチンは, エラーコードを返す.

```
int info = pvm_exit( void )
```

```
call pvmfexit( info )
```

pvm_exit() は, 呼び出しプロセスが PVM を離れることを, ローカルな pvmd に対して知らせる．本ルーチンは, 呼び出しプロセスを終了させるわけではない．プロセスは, 呼び出し後も他の UNIX プロセスと同様に処理を続けることができる.

```
int numt = pvm_spawn( char *task, char **argv, int flag,char *where,
ntask, int *tids )
```

```
call pvmfspawn( task, flag, where, ntask, tids, numt )
```

pvm_spawn() は, 実行可能ファイル task のコピーを ntask 個生成し, バーチャルマシンで起動する．argv は task に与える引数の配列へのポインタであり, その終りは NULL で示される．引数を何も与えない場合 argv は NULL となる．flag は以下のオプションを指定する.

0 PvmTaskDefault PVM がプロセスの生成場所を決定する.

1 PvmTaskHost where 引数で生成するホストを指定する.

2 PvmTaskArch where 引数で, ARCH の中から生成するアーキテクチャを指定する.

4 PvmTaskDebug デバッガーの制御下でプロセスを起動する.

8 PvmTaskTrace プロセスからの PVM 呼び出しのトレースデータを出力する.

FORTRAN では, これらのシンボル名をそれぞれ, **PVMDEFAULT**, **PVMHOST**, **PVMARCH**, **PVMDEBUG**, **PVMTRACE** と短縮して用いる． これらは, インクルードファイル **pvm3/include/fpvm3.h** の中で, parameter 文によって予め定義している.

PvmTaskTrace は,PVM 3.1 では実装されていなが, 今後の PVM 3.x ではサポートされる予定である.

numt は, 生成に成功したタスクの数を返す. もしタスクを一つも生成できなかった場合には, エラーコードを返す. タスクを起動した後,pvm_spawn() はタスク tid のベクトルを返す. 起動できなかった場合は, ベクトルの $ntask - numt$ の位置にエラーコードをセットする.

pvm_spawn() は, マルチプロセッサにおいてもタスクを起動することができる. Intel iPSC/860 の場合は, 以下に示す制限がある. 各タスク生成呼び出しは ntask 個のサブキューブを得ることができ, プログラム task を全てのノードにロードすることができる. iPSC/860 の OS は, 全ユーザに対するサブキューブの割り当てを総計 10 個までに制限する. 従って, iPSC/860 で一つのタスクのブロックを起動するには, 幾つかの pvm_spawn() を呼び出すよりも単一の pvm_spawn() を呼び出したほうが良い. iPSC/860 で別々に起動した 2 種類のタスクのブロックは, 互いに別々のサブキューブにあっても, 他の PVM タスクと同様に通信することができる. またiPSC/860 の OS では, ノード間及びノードと外部との通信メッセージのサイズが 256K バイト未満に制限される.

```
info = pvm_kill( int tid )
call pvmfkill( tid, info )
```

pvm_kill() は tid なる識別子を持つタスクを終了させる. 本ルーチンは, 自分自身を終了させるようには設計されていない. 自分自身を終了させるには pvm_exit() と exit() を続けて呼び出す.

5.2 情報

```
int tid = pvm_parent( void )
call pvmfparent( tid )
```

ルーチン pvm_parent() は, 呼び出しタスクを生成したプロセスの tid を返す. pvm_spawn() により生成されたものでない場合は, 値 PvmNoParent を返す.

```
int pstat = pvm_pstat( int tid )
call pvmfpstat( tid, pstat )
```

ルーチン pvm_pstat() は,tid で指定された PVM タスクの状態を返す. タスクが実行中であれば,PvmOk を返す. そうでなければ PvmNoTask を返し, tid が無効であれば,PvmBadParam を返す.

```
int pstat = pvm_mstat( char *host )
call pvmfmstat( host, mstat )
```

ルーチン pvm_mstat() は,host が稼働中であれば,PvmOk を返す. 指定した host がバーチャルマシンに無い場合は, PvmNoHost を返す. この情報は, アプリケーションレベルでのフォールトトレランスを実現する際に有用である.

```
int info = pvm_config( int *nhost, int *narch, struct hostinfo **hostp )
call pvmfconfig( nhost, narch, info )
```

ルーチン pvm_config() は, バーチャルマシンに関する情報を返す. その情報には, ホストの数を示す nhost, データフォーマットの種類数を示す narch が含まれる. hostp は,hostinfo 構造体の配列へのポインタである. 配列のサイズは nhost に等しい. hostinfo 構造体には, pvmd tid, ホスト名, アーキテクチャ名, 最大パケット長, ホストの相対 CPU 速度が含

まれる.

```
int info = pvm_tasks( int which, int *ntask, struct taskinfo **taskp )
call pvmftasks( which, ntask, info )
```

ルーチン pvm_tasks() は, バーチャルマシンで実行中の PVM タスクに関する情報を返す. 整数 which には, 情報を得たいタスクを指定する. 0 を与えた場合には, 全てのタスクを指定したことになる. pvmd の tid ならば, そのホストで実行中のタスクを指定したことになる. tid ならば, そのタスクを指定を意味する.

ntask により, タスクの数を返す. taskp は taskinfo 構造体の配列へのポインタである. 配列のサイズは ntask に等しい. taskinfo 構造体には, そのタスクの tid, pvmd の tid, 親プロセスの tid, 状態フラグ, 実行ファイルの名前が含まれる.

5.3 実行時設定

```
int info = pvm_addhosts( char **hosts, int nhost, int *infos )
int info = pvm_delhosts( char **hosts, int nhost, int *infos )
call pvmfaddhost( host, info )
call pvmfdelhost( host, info )
```

C ルーチンは, バーチャルマシンに複数のホスト hosts を一括して追加あるいは削除する. FORTRAN ルーチンは, ただ一つのホスト host を追加あるいは削除する. infos は,nhost の大きさの配列であり, 追加あるいは削除されたホストそれぞれについてのステータスコードが格納される.

info の返り値が負であった場合, 再度全ホストを追加あるいは削除する必要は無い. infos をチェックすることで, どのホストが問題を引き起こしたかをチェックすることができる.

5.4 シグナル

```
int info = pvm_sendsig( int tid, int signum )
call pvmfsendsig( tid, signum, info )
```

pvm_sendsig() は, tid で指定された他の PVM タスクに, シグナル signum を送る.

```
int info = pvm_notify( int about, int msgtag, int ntask, int tids )
call pvmfnotify( about, msgtag, ntask, tids, info )
```

pvm_notify() ルーチンは, バーチャルマシンで何かイベントが発生した時に,ntask と tids で指定されたタスク集合へ msgtag を付したメッセージを送信する. about はイベントのタイプを指定する. 現在のオプションを以下に示す.

PvmTaskExit - タスクの終了.

PvmHostDelete - ホストが削除された (あるいは失敗した).

PvmHostAdd - ホストが追加された.

5.5 エラーメッセージ

```
info = pvm_perror( char *msg )
```

```
call pvmfperror( msg, info )
```

このルーチンは,UNIX の perror() 関数に似ており, 最も最近の PVM 呼び出しのエラー状態を表示する.

```
int oldset = pvm_serror( int set )
```

```
call pvmfserror( set, oldset )
```

`set = 1` で pvm_serror() を呼び出した場合には, 自動エラーメッセージ出力が有効になる. 以後の PVM 呼び出しにおいてエラー状態で返った場合は, 自動的にエラーメッセージを出力する. `set = 0` で pvm_serror() を呼び出した場合には, 自動エラーメッセージ出力が無効になる. 呼び出し前の値は `oldset` に設定される. 今後のオプションとして,`set = 2` の場合は, プロセスはエラーメッセージ出力後に終了することを予定している.

5.6 メッセージパッシング

PVM におけるメッセージの送信は,3 つのステップで構成される. まず第 1 に, pvm_initsend() あるいは pvm_mkbuf() を呼び送信バッファを初期化する. 第 2 に,pvm_pk*() ルーチンを組み合わせてバッファにメッセージを “パック” する. (FORTRAN では,pvmfpack() サブルーチンのみを用いて全てのメッセージを “パック” できる.) 第 3 に,pvm_send() ルーチンあるいは pvm_mcast() ルーチンを用いてメッセージ全体を送信する.

メッセージは, ブロックまたは非ブロック受信ルーチンの呼び出しにより受信する. 次に “アンパック” により, パックされた各データを受信バッファより取り出す. 受信ルーチンにおいては, いかなるメッセージも受信, 特定の送信元からのみ受信, 引数で与えたタグの付されたメッセージのみ受信, 送信元とタグの両方を指定し受信, のいずれの設定も可能である.

必要ならば,PVM 3 ではより一般的な受信コンテキストを扱うことができる. pvm_recvf() により, ユーザは以後の全ての PVM 受信関数で用いるコンテキストを定義することができる.

3.1 で新たに登場した関数に pvm_advise() がある. “アドバイス” ルーチンは, 後続するタスクにおいて直結タスク間通信を実現できるかどうか調べる. 選択的に直結通信リンクをセットアップしたい場合には, アプリケーションから何度もこのルーチンを呼び出して構わないが, 典型的な利用法は,pvm_mytid() の後に一度だけ呼ぶことである. 直結通信リンクの利点は, タスク間での通信性能を押し上げることにある. 反対に欠点としては, ある UNIX システムでは利用できる直接通信リンク数が少なく, スケーラブルに利用できない点が挙げられる.

5.6.1 メッセージバッファ

以下のメッセージバッファルーチンは, アプリケーション内で, 複数のメッセージバッファを扱いたい場合に用いる. たいていのプロセス間メッセージパッシングでは, 複数メッセージバッファは必要ではない. PVM 2.4 では, 単一送信バッファ及び単一受信バッファのみ実装されていた. 一方 PVM 3 では, 任意の時点においてただ一つのアクティブな送信バッファ及び受信バッファが, 各プロセスに存在する. 開発者は, いくらでもメッセージバッファを作成できる. これらを切替えながらデータをパックし送信することが可能である. パック, 送信, 受信及びアンパックの各操作は, アクティブなバッファに対してのみ有効である.

```
int bufid = pvm_mkbuf( int encoding )
```

```
call pvmfmkbuf( encoding, bufid )
```

pvm_mkbuf ルーチンは, 新たに空の送信バッファを生成する. その際, メッセージをパックするためのエンコーディング方法を指定する.

バッファ識別子 bufid を返す. encoding には以下のオプションを指定できる.

PvmDataDefault

: PVM によってバーチャルマシンの設定が異機種結合となった場合は,XDR エンコーディングが用いられる. そうでない場合は何のエンコーディングもしない (PVM 2.4 でのデフォルトと同じ). PVM 3.1 リリースでは,XDR エンコーディングを常に行う. なぜならば, タスクはいつでも計算機を追加し異機種結合にする可能性があるためである.

PvmDataRaw

: エンコーディングをしない. メッセージは元のフォーマットのまま送られる. 受信プロセスがこのフォーマットを読むことができない場合, アンパックの途中でエラーが返される.

PvmDataInPlace

: データは送信後も残る. (初期のリリースでは実装されていない.) バッファはサイズとポインタのみを保持する. pvm_send() が呼ばれると, データの各要素は直接ユーザメモリからコピーされる. このオプションは, メッセージのコピー回数を減らす働きがあるが, パックから送信までの間はデータの各要素を変更することはできない. このオプションの別の使い方としては, パックを一度だけ呼んだ後, データの各要素の変更と送信を何回か繰り返す. 適用例として, 離散 PDE の実装における境界領域の受渡しがある.

FORTRAN では, これらのエンコーディング指定をそれぞれ PVMDEFAULT, PVMRAW, PVMINPLACE と短縮して用いる. これらは, インクルードファイル pvm3/include/fpvm3.h の中で,parameter 文によって予め定義されている.

```
int bufid = pvm_initsend( int encoding )
```

```
call pvmfinitsend( encoding, bufid )
```

pvm_initsend ルーチンは, 送信バッファをクリアし, 新しいメッセージのパックのための送信バッファを新たに生成する. パックに用いるエンコーディング方法を encoding で指定する. 新しいバッファの識別子は,bufid で返される. ユーザは単一のバッファを使い続けるならば, 新しいメッセージをパックする前に pvm_initsend() を必ず呼ぶ必要がある. そうしないと, 既に存在するメッセージに追加されることになる.

```
int info = pvm_freebuf( int bufid )
```

```
call pvmffreebuf( bufid, info )
```

pvm_freebuf() ルーチンは,bufid で指定されるバッファを廃棄する. メッセージを送信後, 一度だけ呼ぶ必要がある.

必要ならば,pvm_mkbuf() を呼び新たなメッセージのためのバッファを生成する. pvm_initsend() を用いる場合には, どちらの関数も呼び出す必要はない. pvm_initsend() は, ユーザに代わってそれらの機能を果たしている.

```
int bufid = pvm_getsbuf( void )
call pvmfgetsbuf( bufid )
```

pvm_getsbuf() は, アクティブ送信バッファの識別子を返す.

```
int bufid = pvm_getrbuf( void )
call pvmfgetrbuf( bufid )
```

pvm_getrbuf() は, アクティブ受信バッファの識別子を返す.

```
int bufid = pvm_setsbuf( void )
call pvmfsetsbuf( bufid )
```

pvm_setsbuf() は,bufid で指定するバッファをアクティブ送信バッファにする. 元のアクティブバッファの状態を保存するとともに, その識別子を oldbuf 返す.

```
int bufid = pvm_setrbuf( void )
call pvmfsetrbuf( bufid )
```

pvm_setrbuf() は,bufid で指定するバッファをアクティブ受信バッファにする. 元のアクティブバッファの状態を保存するとともに, その識別子を oldbuf 返す. pvm_setsbuf() 及び pvm_setrbuf() の bufid に 0 を指定した場合, 現在のアクティブバッファの状態を保存し, アクティブバッファが存在しなくなる. この手法は, アプリケーションのメッセージバッファの状態を保存するのに用いられ, 同じく PVM メッセージを使用する数学ライブラリとグラフィカルインターフェースがアプリケーションのバッファの状態を変更してしまうことを防ぐ.

メッセージバッファルーチンによって, 再パックすることなくメッセージを中継することができる.

```
bufid = pvm_recv( src, tag );
oldid = pvm_setsbuf( bufid );
info  = pvm_send( dst, tag );
info  = pvm_freebuf( oldid );
```

5.6.2 データのパック

以下の C ルーチンは, 与えられたデータ型の配列をアクティブ送信バッファにパックする. これらは何回でも呼び出すことができ,1 つのメッセージにパックする. 即ち, 異なるデータ型を持つ複数の配列を 1 つのメッセージに含めることができる. パックされたメッセージの複雑さに制限はないが, アプリケーションはパックされたときと同じように正確にアンパックしなくてはならない. C の構造体は, 各要素毎にパックしなくてはならない.

各ルーチンの最初の引数はパックされるデータへのポインタである. nitem はパックされる配列の要素数である. stride はパックされるときに用いる 1 データ要素あたりの幅である. ここで例外として pvm_pkstr() がある. これは,NULL で終了する文字列をパックし, nitem や stride を指定する必要はない.

```
int info = pvm_pkbyte( char *cp, int nitem, int stride )
```

```
int info = pvm_pkcplx( float *xp, int nitem, int stride )
int info = pvm_pkdcplx( double *zp, int nitem, int stride )
int info = pvm_pkdouble( double *dp, int nitem, int stride )
int info = pvm_pkfloat( float *fp, int nitem, int stride )
int info = pvm_pkint( int *np, int nitem, int stride )
int info = pvm_pklong( long *np, int nitem, int stride )
int info = pvm_pkshort( short *np, int nitem, int stride )
int info = pvm_pkstr( char *cp )
```

FORTRAN では, 上記の C ルーチンのパック機能の全てをただ一つのサブルーチンで実現する.

```
call pvmfpack( what, xp, nitem, stride, info )
```

引数 xp はパックされる配列の第一要素である. ここで,FORTRAN では文字列をパックする際に,nitem で文字数を指定する必要があることに注意する. 整数 what は, パックされるデータ型を指定する. サポートされるデータ型を以下に示す.

STRING	0	REAL4	4
BYTE1	1	COMPLEX8	5
INTEGER2	2	REAL8	6
INTEGER4	3	COMPLEX16	7

これらは, インクルードファイル pvm3/include/fpvm3.h の中で,parameter 文によって予め定義されている. いくつかのベンダ向けの PVM の実装では, 上記リストを 64bit アーキテクチャに含むよう拡張することも可能である. INTEGER8,REAL16 等の追加については, XDR がこれらのデータ型をサポート後, ただちに行う予定である.

5.6.3 データの送信及び受信

```
int info = pvm_send( int tid, int msgtag )
call pvmfsend( tid, msgtag, info )
```

pvm_send() ルーチンは, メッセージに整数識別子 msgtag をラベル付けし, tid で指定されるプロセスへ直ちに送信する.

```
int info = pvm_mcast( int *tids, int ntask, int msgtag )
call pvmfmcast( ntask, tids, msgtag, info )
```

pvm_mcast() ルーチンは, メッセージに整数識別子 msgtag をラベル付けし, 長さ ntask の配列 tids で指定される全てのタスクへブロードキャスト送信する.

```
int bufid = pvm_nrecv( int tid, int msgtag )
call pvmfnrecv( tid, msgtag, bufid )
```

要求したメッセージが到着していない場合, 非ブロック受信 pvm_nrecv() は bufid = 0 で返る. このルーチンを繰り返し呼び出してメッセージが到着しているかどうかをチェックすることで, 各呼び出し間を他の有用な処理に充てることができる. 他にすべき有用な処理がない場合は, ブロック受信 pvm_recv() を用いることができる. tid から msgtag でラベル付けされたメッセージが到着すると,pvm_nrecv() はアクティブ受信バッファを新しく生成し, そこにメッセージを置いた後, バッファの識別子を返す. 以前のアクティブ受信

バッファはクリアされるので, 必要ならば pvm_setrbuf() 呼び出しによって保存しておく. msgtag あるいは tid で -1 を指定するとワイルドカードとなり, いかなる対象ともマッチする.

```
int bufid = pvm_recv( int tid, int msgtag )
call pvmfrecv( tid, msgtag, bufid )
```

ブロック受信ルーチンは,tid から msgtag でラベル付けされたメッセージが到着するまで待つ. msgtag あるいは tid が -1 ならば, ワイルドカードとしていかなる対象ともマッチする. メッセージが到着すると, アクティブ受信バッファを新しく生成しメッセージを置く. 以前のアクティブ受信バッファはクリアされるので, 必要ならば pvm_setrbuf() 呼び出しによって保存しておく.

```
int bufid = pvm_probe( int tid, int msgtag )
call pvmfprobe( tid, msgtag, bufid )
```

要求するメッセージが到着していない場合,pvm_probe() は bufid に 0 を返す. 到着していればメッセージの bufid を返すが, まだ “受信” は行わない. このルーチンを繰り返し呼び出してメッセージが到着しているかどうかをチェックすることで, 各呼び出し間を他の有用な処理に充てることができる. 更に返された bufid を引数として pvm_bufinfo() を呼び出せば, 受信前にメッセージに関する情報を得ることができる.

```
int info = pvm_bufinfo(int bufid, int *bytes, int *msgtag, int *tid )
call pvmfbufinfo( bufid, bytes, msgtag, tid, info )
```

pvm_bufinfo() ルーチンは bufid で識別されるバッファの中のメッセージに関する情報を返す. 情報には, 実際の msgtag, 実際の送信元 tid, 及びバイト長がある. ワイルドカード指定で受信することにより, メッセージのラベル及び送信元を決定することができる.

```
int (*old)() = pvm_recvf(int (*new)(int buf, int tid, int tag ))
```

pvm_recvf() ルーチンは, 受信関数で用いられる受信コンテキストを変更することで,PVM の拡張を可能にする. デフォルトの受信コンテキストは, 送信元とメッセージのタグのマッチであるが, これをユーザが任意の比較関数に変更することができる. (付録 A に,pvm_recvf() を使ったプローブ関数の構成例を示す.) pvm_recvf() の FORTRAN インターフェースルーチンは提供されない.

```
int info = pvm_advise( route )
call pvmfadvise( route, info )
```

pvm_advise() ルーチンは, タスク間の直接リンクの設定に関して PVM に指示を行う関数である.

PvmDontRoute - タスク間の直接リンクを許可しない.

PvmAllowDirect - デフォルトの直接リンクを許可する. しかし要求はしない.

PvmRouteDirect - 直接リンクを要求する.

一旦リンクが確立すると, 双方のタスクがこれを利用し, アプリケーションが終了するまで残る. 双方のタスクが PvmDontRoute を指定している, あるいは使えるリソースがないために, リンクが確立できない場合は,PVM デーモンを介したデフォルトのルートが用いれら

る．pvm_advise() を何度も呼び出し, 選択的に直接リンクを確立することもできるが, 普通は各タスク処理の最初の方で一度だけ設定する．

5.6.4 データのアンパック

以下の C ルーチンは,(複数の) データ型をアクティブ受信バッファよりアンパックする．アプリケーションは, パックされたときのデータ型, 要素数及びデータ幅に併せてアンパックするしなくてはならない． nitem はアンパックされる要素数であり,stride はパックされる時に用いるストライド[2]である．

```
int info pvm_upkbyte( char *cp, int nitem, int stride )
int info pvm_upkcplx( float *xp, int nitem, int stride )
int info pvm_upkdcplx( double *zp, int nitem, int stride )
int info pvm_upkdouble( double *dp, int nitem, int stride )
int info pvm_upkfloat( float *fp, int nitem, int stride )
int info pvm_upkint( int *np, int nitem, int stride )
int info pvm_upklong( long *np, int nitem, int stride )
int info pvm_upkshort( short *np, int nitem, int stride )
int info pvm_upkstr( char *cp )
```

FORTRAN では, 上記の C ルーチンのアンパック機能の全てをただ一つのサブルーチンで実現する．

```
call pvmfunpack( what, xp, nitem, stride, info )
```

引数 xp はアンパックされる配列の第一要素である．整数 what は, アンパックされるデータ型を指定する (指定方法は pvmfpack() に同じ)．

6 実行時プロセスグループ

動的プロセスグループ機能は, コア PVM ルーチンの上位に構築されている．これらは別のライブラリ libgpvm3.a となっており, いずれの機能を用いる場合でもユーザプログラムとリンクする必要がある． pvmd がグループ機能を実現するのではなく, グループ機能を最初に実行するときに自動的に起動されるグループサーバが, これらの機能を実現する．メッセージパッシングインターフェースにおいては, どのようにグループ機能を実現するかは議論の分かれるところである． そこには, 効率と信頼性の問題や, 静的.vs. 動的のトレードオフが存在する．また, グループに所属するタスクのみがグループ機能を呼び出せれば良いという意見もある．明示的なグループ機能は,PVM 3 の新しい特徴である． 提供されるルーチンの機能性と柔軟性は, アプリケーションのグループ機能の要求に応えることができる． PVM の哲学を踏襲し, 効率を幾らか犠牲にはしているものの, グループ機能はユーザに対して非常に一般的かつ透過的に設計されている．あらゆる PVM タスクは, いつでもグループに所属あるいは離脱することができ, 対象グループの他のタスクに知らせる必要はない．タスクは, 自らがメンバーではない他のグループに対してブロードキャストすることができる．一般的には, どの PVM タスクも以下のグループ機能を何時でも呼び出すことがで

[2]訳注: データをとびとびにアクセスする場合の間隔のことであり, この場合は (stride-1) 個おきにデータがパックされる．

きる． 例外は pvm_lvgroup() と pvm_barrier() であり, これらは本質的に引数で指定するグループに呼び出しタスクが所属している必要がある.

```
int inum = pvm_joingroup( char *group )
```

```
int inum = pvm_lvgroup( char *group )
```

```
call pvmfjoingroup( group, inum )
```

```
call pvmflvgroup( group, inum )
```

これらのルーチンは, ユーザが名前で指定するグループへのタスクの所属あるいは離脱を実現する． 最初の pvm_joingroup() は名前 group のグループを生成し, 呼び出しタスクをそのグループへ入れる． pvm_joingroup() はグループ内でのプロセスのインスタンス番号を返す. インスタンス番号は 0 から始まり, グループのメンバーの数から 1 を引いたもので終る. PVM 3 では, タスクは複数のグループに所属することができる.

プロセスがグループから離脱し再度所属する場合は, 同じインスタンス番号になるとは限らない. インスタンス番号は再利用され, 所属する際には可能な最も小さい番号を割り当てられる. しかし, タスクが何度もグループに所属を繰り返す場合には, 以前と同じ番号を割り当てられる保証はない.

所属と離脱を繰り返しても連続なインスタンス番号をユーザが維持できるように, これを助ける関数 pvm_lvgroup() が用意されている． この関数は, タスクが離脱したことを確認できるまで制御を返さない. このすぐ後に pvm_joingroup() を呼び出せば, 空いたインスタンス番号を新しいタスクに割り当てることができる． アルゴリズムが連続なインスタンス番号を必要とする場合に, それを維持するのはユーザの責任である. いくつかのタスクがグループから離脱し, 他のタスクが全く所属しなかった場合, インスタンス番号にはギャップが発生する.

```
int tid = pvm_gettid( char *group, int inum )
```

```
call pvmfgettid( group, inum, tid )
```

pvm_gettid() は, 与えられたグループ名とインスタンス番号から tid を返す.

```
int inum = pvm_getinst( char *group, int tid )
```

```
call pvmfgetinst( group, tid, inum )
```

pvm_getinst() ルーチンは, 指定されたグループの tid を持つプロセスのインスタンス番号を返す.

```
int size = pvm_gsize( char *group )
```

```
call pvmfgsize( group, size )
```

pvm_gsize() ルーチンは, 指定されたグループのメンバー数を返す.

```
int info = pvm_barrier( char *group, int count )
```

```
call pvmfbarrier( group, count, info )
```

pvmbarrier() の呼び出し時には, グループ内で count で指定された数のメンバーが pvm_barrier を呼び出すまで, プロセスはブロックされる． 一般的には,count はグループのメンバーの総数である． 動的プロセスグループ環境下では, 与えられたインスタンスだけではそのグループのメンバー数がいくつか知ることができないので,count は必須である. 所属していないグループに対する pvm_barrier の呼び出しは, エラーとなる. 引数 count が barrier の各呼び出しで一致していない場合もまた, エラーとなる. 例えば, あるグループで複数のメ

ンバーが count を 5 として pvm_barrier() を呼び出しているところに, あるメンバーが count を 4 にして pvm_barrier() を呼び出すとエラーとなる.

```
int info = pvm_bcast( char *group, int msgtag )
```

```
call pvmfbcast( group, msgtag, info )
```

pvm_bcast() は, 整数識別子 msgtag でラベル付けしたメッセージを, 指定されたグループに所属する全てのタスクにブロードキャストする. ユーザからのフィードバックとして,pvm_bcast() は送信者自身に対してはメッセージを送らないで欲しいという要求があった. 現在のところ, 送信者がグループのメンバーであった場合には送ってしまう仕様となっているが,PVM の次期リリースでは送信者自身にはメッセージを送信しない予定である.

pvm_bcast() にとって “全てのタスク” とは, そのルーチンが呼び出された時点でグループサーバがグループに所属するとみなしたタスクをいう. ブロードキャストを実行中にグループに所属することとなったタスクにはメッセージは送られず, ブロードキャストを実行中にグループサーバ離脱することとなったタスクにはメッセージのコピーが送信される.

7 C と FORTRAN による記述例

本章では,PVM 3 におけるアプリケーションの構成法について, 異なる二つの形式による例を示す. 例は, 理解し易くかつ説明を容易にするために, わざと単純にしている. 各プログラムは C 及び FORTRAN で記述されており, 計 4 つのリストを掲載した. これらの例とその他 2,3 の例が,PVM ソースコードの pvm3/examples にも提供されている.

最初の例は, マスター / スレーブモデルとスレーブ間通信である. 第二の例は, 単一プログラム多重データ (SPMD : Single Program Multiple Data) モデルである.

マスター / スレーブモデルでは, マスタープログラムは複数のスレーブプログラムを生成・制御し, 各スレーブプログラムは計算を行う. PVM はこのモデルに限定されるものではない. 例えば, どの PVM タスクも他のマシンでプロセスを生成することができる. しかし, マスター / スレーブモデルは大変有用なプログラミングパラダイムであり, 説明も簡単である. マスターは pvm_mytid() を呼び出し, PVM システムの利用とプロセス間通信を実行可能にする. そして pvm_spawn() を呼び出し,PVM を構成する各マシンでスレーブプログラムの実行を開始する. 各スレーブプログラムは pvm_mytid() を呼び出し, プロセス間通信を実行可能にする. 続いて,pvm_send() あるいは pvm_recv() を呼び出し, プロセス間でメッセージを交換する.

処理が終了したら, 全ての PVM プログラムは pvm_exit() を呼び出して, PVM がプロセスとのソケットの接続を断つことを許可すると同時に, どのプロセスが実行中か追跡できるようにする.

SPMD モデルではプログラムはただ一つであり, 計算を制御するマスタープログラムは存在しない. そのようなプログラムは, 時としてホストレスプログラムと呼ばれる. この場合, 全てのプロセスをどう初期化するかという問題が発生する. 例 2 では, ユーザはプログラムのコピーの 1 番目を起動している. このコピーは pvm_parent() をチェックし, 自分が PVM によって生成されたのではなく 1 番目のコピーであると判断する. そして自分自身のコピーを複数生成し, tid の配列を渡す. この点において各プロセスは対等であり, 他のプロセスと協調しながらデータを分割して処理する. pvm_parent は PVM コンソールの tid を

返すため, pvm_parent を使うことで, コンソールからの SPMD プログラムの起動を排除する. こうしたタイプの SPMD プログラムは UNIX プロンプトから起動されなくてはならない.

表 1: PVM 3 で使用する ARCH 名

ARCH	Machine	Notes
AFX8	Alliant FX/8	
ALPHA	DEC Alpha	DEC OSF 1
BAL	Sequent Balance	DYNIX
BFLY	BBN Butterfly TC2000	
BSD386	80386/486 Unix Box	BSDI
CM2	Thinking Machines CM2	Sun front-end
CM5	Thinking Machines CM5	
CNVX	Convex C-series	
CNVXN	Convex C-series	native mode
CRAY	C-90,YMP,Cray-2	UNICOS
CRAYSMP	Cray S-MP	
DGAV	Data General Aviion	
HP300	HP-9000 model 300	HPUX
HPPA	HP-9000 PA-RISC	
I860	Intel iPSC/860	link -lrpc
IPSC2	Intel iPSC/2 386 host	SysV
KSR1	Kendall Square KSR-1	OSF-1
NEXT	NeXT	
PGON	Intel Paragon	link -lrpc
PMAX	DECstation 3100, 5100	Ultrix
RS6K	IBM/RS6000	AIX
RT	IBM/RT	
SGI	Silicon Graphics IRIS	link -lsun
SUN3	Sun 3	SunOS
SUN4	Sun 4, SPARCstation	
SYMM	Sequent Symmetry	
TITN	Stardent Titan	
UVAX	DEC MicroVAX	

```
#include "pvm3.h"
#define SLAVENAME "slave1"

main()
{
    int mytid;                  /* my task id */
    int tids[32]; /* slave task ids */
    int n, nproc, i, who, msgtype;
    float data[100], result[32];

    /* enroll in pvm */
    mytid = pvm_mytid();

    /* start up slave tasks */
    puts("How many slave programs (1-32)?");
    scanf("%d", &nproc);

    pvm_spawn(SLAVENAME, (char**)0, 0, "", nproc, tids);

    /* Begin User Program */
    n = 100;
    /* initialize_data( data, n ); */
    for( i=0 ; i<n ; i++ ){
       data[i] = 1;
    }

    /* Broadcast initial data to slave tasks */
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&nproc, 1, 1);
    pvm_pkint(tids, nproc, 1);
    pvm_pkint(&n, 1, 1);
    pvm_pkfloat(data, n, 1);
    pvm_mcast(tids, nproc, 0);

    /* Wait for results from slaves */
    msgtype = 5;
    for( i=0 ; i<nproc ; i++ ){
       pvm_recv( -1, msgtype );
       pvm_upkint( &who, 1, 1 );
       pvm_upkfloat( &result[who], 1, 1 );
       printf("I got %f from %d\n",result[who],who);
    }
    /* Program Finished exit PVM before stopping */
    pvm_exit();
}
```

図 3: C 版マスターの例

```
#include <stdio.h>
#include "pvm3.h"

main()
{
    int mytid;       /* my task id */
    int tids[32];    /* task ids   */
    int n, me, i, nproc, master, msgtype;
    float data[100], result;
    float work();

    /* enroll in pvm */
    mytid = pvm_mytid();

    /* Receive data from master */
    msgtype = 0;
    pvm_recv( -1, msgtype );
    pvm_upkint(&nproc, 1, 1);
    pvm_upkint(tids, nproc, 1);
    pvm_upkint(&n, 1, 1);
    pvm_upkfloat(data, n, 1);

    /* Determine which slave I am (0 -- nproc-1) */
    for( i=0; i<nproc ; i++ )
       if( mytid == tids[i] ){ me = i; break; }

    /* Do calculations with data */
    result = work( me, n, data, tids, nproc );

    /* Send result to master */
    pvm_initsend( PvmDataDefault );
    pvm_pkint( &me, 1, 1 );
    pvm_pkfloat( &result, 1, 1 );
    msgtype = 5;
    master = pvm_parent();
    pvm_send( master, msgtype );

    /* Program finished. Exit PVM before stopping */
    pvm_exit();

float
work(me, n, data, tids, nproc )
    int me, n, *tids, nproc;
    float *data;
{
    int i, dest;
    float psum = 0.0;
    float sum = 0.0;
    for( i=0 ; i<n ; i++ ){
       sum += me * data[i];
    }
    /* illustrate node-to-node communication */
    pvm_initsend( PvmDataDefault );
    pvm_pkfloat( &sum, 1, 1 );
    dest = me+1;
    if( dest == nproc ) dest = 0;
    pvm_send( tids[dest], 22 );
    pvm_recv( -1, 22 );
    pvm_upkfloat( &psum, 1, 1 );

    return( sum+psum );
}
```

図 4: C 版スレーブの例

```
      program master1
      include '../include/fpvm3.h'
c -------------------------------------------------------
c Example fortran program illustrating the use of PVM 3.0
c -------------------------------------------------------
      integer i, info, nproc, msgtype
      integer mytid, tids(0:32)
      integer who
      double precision result(32), data(100)
      character*12 nodename
      character*8 arch

c ------------ Starting up all the tasks ---------------------------

c     Enroll this program in PVM
      call pvmfmytid( mytid )

c     Initiate nproc instances of slave1 program
      print *,'How many slave programs (1-32)?'
      read *, nproc
c
c     If arch is set to '*' then ANY configured machine is acceptable
c     otherwise arch should be set to architecture type you wish to use.
      nodename = 'slave1'
      arch = '*'

      call pvmfspawn( nodename, PVMDEFAULT, arch, nproc, tids, info )
      do 100 i=0, nproc-1
         print *,'tid',i,tids(i)
100   continue

c ------- Begin user program --------

      n = 10
c     Initiate data array
      do 20 i=1,n
         data(i) = 1
 20   continue

c     broadcast data to all node programs
      call pvmfinitsend( PVMDEFAULT, info )
      call pvmfpack( INTEGER4, nproc, 1, 1, info )
      call pvmfpack( INTEGER4, tids, nproc, 1, info )
      call pvmfpack( INTEGER4, n, 1, 1, info )
      call pvmfpack( REAL8,    data, n, 1, info )
      msgtype  = 1
      call pvmfmcast( nproc, tids, msgtype, info )

c     wait for results from nodes
      msgtype  = 2
      do 30 i=1,nproc
         call pvmfrecv( -1, msgtype, info )
         call pvmfunpack( INTEGER4, who, 1, 1, info )
         call pvmfunpack( REAL8, result(who+1), 1, 1, info )
         print *, 'I got',result(who+1), '  from', who
 30   continue

c --------- End user program --------

c     program finished leave PVM before exiting
      call pvmfexit(info)
      stop
      end
```

図 5: FORTRAN 版マスターの例

```
      program slave1
      include '../include/fpvm3.h'
c -------------------------------------------------------
c Example fortran program illustrating use of PVM 3.0
c -------------------------------------------------------
      integer  info, mytid, mtid, msgtype, me
      integer  tids(0:32)
      double precision result, data(100)
      double precision work

c  Enroll this program in PVM
      call pvmfmytid( mytid )
c  Get the master's task id
      call pvmfparent( mtid )

c ------- Begin user program --------

c     Receive data from host
      msgtype  = 1
      call pvmfrecv( mtid, msgtype, info )
      call pvmfunpack( INTEGER4, nproc, 1, 1, info )
      call pvmfunpack( INTEGER4, tids, nproc, 1, info )
      call pvmfunpack( INTEGER4, n, 1, 1, info )
      call pvmfunpack( REAL8,    data, n, 1, info )

c     Determine which slave I am (0 -- nproc-1)
      do 5 i=0, nproc
         if( tids(i) .eq. mytid ) me = i
  5   continue

c     Do calculations with data
      result = work( me, n, data, tids, nproc )

c     Send result to host
      call pvmfinitsend( PVMDEFAULT, info )
      call pvmfpack( INTEGER4, me, 1, 1, info )
      call pvmfpack( REAL8,    result, 1, 1, info )
      msgtype  = 2
      call pvmfsend( mtid, msgtype, info )

c --------- End user program --------

c     Program finished. Leave PVM before exiting
      call pvmfexit(info)
      stop
      end
```

図 6: FORTRAN 版スレーブの例

```
/*
*    SPMD example using PVM 3.0
*/

#define NPROC 4

#include <sys/types.h>
#include "pvm3.h"

main()
{
    int mytid;                  /* my task id */
    int tids[NPROC];            /* array of task id */
    int me;                     /* my process number */
    int i;

    /* enroll in pvm */
    mytid = pvm_mytid();

    /* find out if I am parent or child */
    tids[0] = pvm_parent();
    if( tids[0] < 0 )       /* then I am the parent */
    {
       tids[0] = mytid;
       me = 0;
       /* start up copies of myself */
       pvm_spawn("spmd", (char**)0, 0, "", NPROC-1, &tids[1]);

       /* multicast tids array to children */
       pvm_initsend( PvmDataDefault );
       pvm_pkint(tids, NPROC, 1);
       pvm_mcast(&tids[1], NPROC-1, 0);
    }
    else    /* I am a child */
    {
       /* receive tids array */
       pvm_recv(tids[0], 0);
       pvm_upkint(tids, NPROC, 1);
       for( i=1; i<NPROC ; i++ )
          if( mytid == tids[i] ){ me = i; break; }
     }
/*-------------------------------------------------------------
 * all NPROC tasks are equal now
 * and can address each other by tids[0] thru tids[NPROC-1]
 * for each process me => process number [0-(NPROC-1)]
 *-------------------------------------------------------------
 */
     printf("me = %d mytid = %d\n",me,mytid);
     dowork( me, tids, NPROC );

     /* program finished exit pvm */
     pvm_exit();
     exit(1);
}
```

図 7: C 版SPMD の例 (その 1)

```
/* Simple example passes a token around a ring */

dowork( me, tids, nproc )
     int me;
     int *tids;
     int nproc;
{
     int token;
     int dest;
     int count  = 1;
     int stride = 1;
     int msgtag = 4;

     if( me == 0 )
     {
        token = tids[0];
        pvm_initsend( PvmDataDefault );
        pvm_pkint( &token, count, stride );
        pvm_send( tids[me+1], msgtag );
        pvm_recv( tids[nproc-1], msgtag );
        printf("token ring done\n");
     }
     else
     {
        pvm_recv( tids[me-1], msgtag );
        pvm_upkint( &token, count, stride );
        pvm_initsend( PvmDataDefault );
        pvm_pkint( &token, count, stride );
        dest = (me == nproc-1)? tids[0] : tids[me+1] ;
        pvm_send( dest, msgtag );
     }
}
```

図 8: C 版 SPMD の例 (その 2)

```
c----------------------------------------
c   SPMD Fortran example using PVM 3.0
c----------------------------------------
       program spmd
       include '../include/fpvm3.h'
       PARAMETER( NPROC=4 )

       integer mytid, me, i
       integer tids(0:NPROC)

c      -------------
c      Enroll in pvm
c      -------------
       call pvmfmytid( mytid )

c      ------------------------------------------------------------------
c      Find out if I am parent or child - spawned processes have parents
c      ------------------------------------------------------------------
       call pvmfparent(tids(0))
       if( tids(0) .lt. 0 )  then
         tids(0) = mytid
         me = 0
c        -------------------------
c        start up copies of myself
c        -------------------------
         call pvmfspawn('spmd',PVMDEFAULT,'*',NPROC-1,tids(1),info)
c        --------------------------
c        multicast tids array to children
c        --------------------------
         call pvmfinitsend( PVMDEFAULT, info )
         call pvmfpack( INTEGER4, tids, NPROC, 1, info )
         call pvmfmcast( NPROC-1, tids(1), 0, info )
       else
c        --------------------------------
c        receive the tids array and set me
c        --------------------------------
         call pvmfrecv( tids(0), 0, info )
         call pvmfunpack( INTEGER4, tids, NPROC, 1, info )
         do 30 i=1, NPROC-1
           if( mytid .eq. tids(i) ) me = i
   30    continue
       endif
c-------------------------------------------------------------
c  all NPROC tasks are equal now
c  and can address each other by tids(0) thru tids(NPROC-1)
c  for each process me => process number [0-(NPROC-1)]
c-------------------------------------------------------------
       print*,'me =',me, '  mytid =',mytid
       call dowork( me, tids, NPROC )

c      -------------------------
c      program finished exit pvm
c      -------------------------
       call pvmfexit(info)
       stop
       end
```

図 9: FORTRAN 版 SPMD の例 (その 1)

```
      subroutine dowork( me, tids, nproc )
      include '../include/fpvm3.h'
c-----------------------------------------------
c Simple subroutine to pass a token around a ring
c-----------------------------------------------
      integer me, nproc
      integer tids( 0:nproc)

      integer token, dest, count, stride, msgtag

      count  = 1
      stride = 1
      msgtag = 4

      if( me .eq. 0 ) then
         token = tids(0)
         call pvmfinitsend( PVMDEFAULT, info )
         call pvmfpack( INTEGER4, token, count, stride, info )
         call pvmfsend( tids(me+1), msgtag, info )
         call pvmfrecv( tids(nproc-1), msgtag, info )
         print*, 'token ring done'
      else
         call pvmfrecv( tids(me-1), msgtag, info )
         call pvmfunpack( INTEGER4, token, count, stride, info )
         call pvmfinitsend( PVMDEFAULT, info )
         call pvmfpack( INTEGER4, token, count, stride, info )
         dest = tids(me+1)
         if( me .eq. nproc-1 ) dest = tids(0)
         call pvmfsend( dest, msgtag, info )
      endif

      return
      end
```

図 10: FORTRAN 版 SPMD の例 (その 2)

8 アプリケーションの記述

アプリケーションの立場から見たPVMは, 計算モデルとしてメッセージパッシングをサポートする包括的で柔軟な並列計算資源である. この資源に対しては, 3つのレベルでアクセスすることができる. **透過**モードでは, タスクは自動的に適切なホスト上 (普通は最も負荷の低いコンピュータ) で実行される. **アーキテクチャ依存**モードでは, ユーザが特定のタスクを実行するためのアーキテクチャを指定することができる. **低レベル**モードでは, ホストを指定することができる. このような階層構成により, ネットワーク上の個々のマシンの特性を引き出しうる能力を保ちながら, 同時に柔軟さを得ている.

PVMにおけるアプリケーションは, 任意の制御依存構造を持つことができる. 言い換えるならば, 並列アプリケーション実行中は何時でも, 任意のプロセスが互いに関連し, 通信あるいは同期を行うことができる. これによって,MIMD並列計算の最も一般的な形態を実現できるが, 実際の並列アプリケーションはもっと構造化されている場合が多い. 典型的な2つのプログラム構造として,SPMDモデルとマスター / スレーブモデルがある. SPMDモデルでは, 全てのプロセスは独立している. マスター / スレーブモデルでは, 一つまたは複数のマスタープロセスの下で, 計算スレーブプロセスの集合が処理を行う.

8.1 一般的な性能に関する考察

PVMのプログラミングパラダイムには, ユーザが選択を迫られるような制限はない. 適当なPVM構成部品を利用することで, あらゆる特殊な制御依存構造をPVMシステムに実装することができる. 一方, アプリケーション開発者がメッセージパッシングシステムをプログラムする際には, 考察すべき点がいくつか存在する.

第一の考察は, タスクの粒度である. これは一般的には, プロセスが受信するバイト数と, プロセスが実行する浮動小数点演算の数との比で図られる. PVM設定でのマシンの計算速度及びマシン間ネットワークの帯域幅から簡単な計算をすれば, ユーザは所望のタスク粒度の下界を得ることができる.

第二の考察は, 送信するメッセージ数である. 多数のバイトを受信する場合, 小さなサイズのメッセージが多数送られてくるケースと, 大きなメッセージが少数送られてくるケースの二通りがある. 大きなサイズのメッセージを少数送れば, メッセージをスタートアップするための時間は削減できるものの, 全体の処理時間の短縮には至らない. 一方, 小さなサイズのメッセージは他の計算とオーバーラップさせることもできるので, メッセージのオーバヘッドを覆い隠す場合がある. 通信と計算のオーバーラップの可能性と, 送信するメッセージ数の最適値は, アプリケーションに依存する.

第三の考察は, 機能並列とデータ並列のどちらにアプリケーションは適しているかという点である. 機能並列は, 異機種構成のPVM設定で, 種々の異なるタスクを実行する場合に適する. 例えば, ベクトル型スーパーコンピュータはベクトル化に適した部分を処理し, マルチプロセッサは並列化に適した部分を処理し, グラフィックスワークステーションは生成されたデータをリアルタイム表示する部分を処理する, といった構成が考えられる. この場合, 各マシンは (たぶん同じデータに対して) 異なる処理を行う.

データ並列モデルでは, データは分割されてPVM設定のマシン全てに分散される. 分散された各データに対して操作 (似ていることもある) を行い, 問題の処理が終了するまでプ

ロセス間で情報が受渡しされる．データ並列は, 分散メモリ型マルチプロセッサでは一般的に用いられている．その理由としては, 全マシンで実行する並列プログラムをただ一つだけ記述すればよい, また数百のプロセッサにスケールアップ可能である, 等が挙げられる． 多くの線形代数, 偏微分方程式, 行列アルゴリズムがデータ並列モデルを使って開発されている．

もちろん PVM では, 両方のモデルをハイブリッドに複合し, 各マシンの特性を引き出すことも可能である． 例えば, 上記のマルチプロセッサで動作する機能並列の各プログラムを,PVM のデータ並列モデルで記述することもできる．

8.2 ネットワークに関する考察

アプリケーション開発者は, ネットワーク上のマシンを使って並列アプリケーションを実行したいのかどうかを, 更に考察する必要がある．並列プログラムは, 他のユーザとネットワークを共有する．マルチユーザ, マルチタスク環境は, 通信と計算速度の両面から並列プログラムに複雑な影響を与える．

第一の考察は, 設定した各マシンの計算能力が異なることによる影響である．これは, バーチャルマシンを構成する各マシンの機種が異なり, しかも計算速度比の異なっている場合に起こる．例えば機種の異なるワークステーションであれば, その計算速度に 2 桁の開きが生じることもありうる． スーパーコンピュータならなおさらである． しかしながら, たとえユーザが同一機種のマシンを指定したとしても, 依然として各マシンで得られる性能に大きな隔たりを見つけるかもしれない．これは, ユーザー自身のマルチタスク処理によって引き起こされるか, あるいは設定したマシンの幾つかで他ユーザのタスクが実行されることによって引き起こされる．ユーザが問題を均一な処理単位に分割し (並列化に共通のアプローチである) 各マシンで実行する場合, 上記の考察は性能に悪影響を及ぼす．最も遅いマシンで実行されるタスクに合わせてアプリケーションは遅くなる． タスクが互いに協調して処理を行う場合, 最も速いマシンのタスクは, 遅いタスクのデータを待つために速度を落してしまう．

第二の考察は, ネットワークでの長いメッセージ遅延である． これは, 広域ネットワークを用いた場合にマシン間の距離によって生じる可能性がある． あるいは, ローカルエリアネットワークにおいても, 自分自身または他ユーザのプログラムと衝突して遅延が生じることもある．イーサネットはバスとみなさなくてはならない．バス上に同時に存在できるメッセージはただ一つである． 各タスクは隣のタスクにのみ送信するようにアプリケーションが設計されていれば, まず衝突は発生しないと考えるかもしれない．たしかに, 分散メモリ型マルチプロセッサでは衝突は発生せず, 全ての送信は並列に処理されるであろう．しかしイーサネットでの送信は直列に処理され, 可変長の遅延で隣のタスクに到着する．トークンリング,FDDI 及び HiPPI といった他のネットワークでも可変長の遅延は発生する．ユーザは遅延に対する耐性をアルゴリズムに組み込むべきか判断する必要がある．

ネットワークに関するこれらの考察は, 並列アプリケーションにある種の負荷分散を組み込む際にも必要となってくる．次節では, 負荷分散の代表的手法について述べる．

8.3 負荷分散

マルチユーザネットワーク環境において性能向上を図るには, 負荷分散こそ最も重要かつ唯一の方策である. 並列プログラムには, 多くの負荷分散手法がある. 本節では, ネットワークコンピューティングで用いられる代表的な手法を三つ述べる.

最も簡単な方法は, **静的**負荷分散である. この方法では, 問題は分割されてプロセッサにただ一度だけ割り当てられる. データ分割は, ジョブの開始前にオフラインで行われるか, あるいはアプリケーションの初期の段階で行われる. タスクの大きさや各マシンに割り当てられるタスクの数は, マシンの計算能力に応じて計算して求めるため, 各マシン毎にバラバラとなる. 開始時より全てのタスクはアクティブになることもでき, 互いに通信し協調して動作する. ネットワークの負荷が軽い場合は, **静的**負荷分散は非常に効果的である.

計算負荷にバラツキがある場合は, **動的**負荷分散が必要となる. 最も有名な手法は, **タスクプール**のパラダイムである. 本手法はマスター / スレーブプログラムの実装で用いられることが多い. マスターは, プールを生成・保持し, スレーブプログラムがアイドル状態になると, 新たなタスクを供給する. プールは通常待ち行列として実装され, 大きさが変わるタスクや大きなタスクは待ち行列の先頭の方に置かれる. 本手法では, プールにタスクがある限り, 各スレーブプログラムは常にビジー状態を維持する. **タスクプール**のパラダイムの実現例として,`xep` プログラムを PVM ソースコードの `pvm3/mandelbrot` に用意した. タスクは任意の時点で起動と終了を繰り返すので, スレーブプログラム間では通信を行わずマスターやファイルとのみ情報交換を行う. このようなアプリケーションに, 本手法は適する.

第三の負荷分散手法は, マスタープロセスを使わずに, 事前に決めておいた時刻に全てのプロセスが仕事の再検査と再分配を行う手法である. 例として, 非線形偏微分方程式の解法が挙げられる. 各線形ステップは静的に負荷分散され, 各ステップ間でプロセスは, どのように問題が変化したか, どのようにメッシュ点を再分配するかを検査する. この基本的な手法には幾つかのバリエーションが存在する. ある実装では, 全プロセッサで同期をとらないかわりに, 隣り合うプロセッサの負荷を追加して割り当てる. またある実装では, 固定時間間隔で再分配を行うかわりに, いずれかプロセスの負荷が一定の限度を越えた時にシグナルを発っして再分配が始まる.

9 デバッグ手法

一般的には, 並列プログラムのデバッグは逐次プログラムのそれと比べて非常に難しい. 同時に多数のプロセスが走ること以外にも, それらの相互作用もまたエラーを引き起こす可能性がある. 例えば, あるプロセスが誤ったデータを受け取り, しばらくしてからゼロ割りを引き起こす. 別の例としては, プログラムの誤りによって全てのプロセスがメッセージ待ちの状態に陥る**デッドロック**がある. PVM ルーチンは全て, 実行中に何らかのエラーを検出すればエラー状態値とともに制御を返す. 表 2にエラーコードの一覧を示す.

PVM は二つのルーチン pvm_serror() と pvm_perror() を提供し,PVM ルーチンのエラーの自動検出と出力を実現している. また,dbx のような標準的な逐次プログラムデバッガーの下に,PVM タスクを手動で起動できる点にも留意されたい.

PVM タスクは, デバッガーの下で生成することができる. pvm_spawn() 呼び出しにおいて `flag` オプションに PvmTaskDebug を指定すると,PVM は `pvm3/lib/debugger` を実行

する．提供されているスクリプトは,PVM を起動したホストで xterm ウィンドウを立ち上げ, デバッガーの下でタスクを生成する． pvm_spawn() に `flag` と `where` を指定することにより, バーチャルマシン上の任意のホストで, デバッグ対象となるタスクを実行することができる．シェルスクリプトを書き換えて, 好みのデバッガーあるいはもし可能ならば並列デバッガーを利用することも可能である．

生成されたタスクから標準出力あるいは標準エラー出力に送られる診断出力メッセージは, ユーザの画面上には表示されない．これらは,PVM を起動したホストの `/tmp/pvml.<uid>` なる単一のログファイルに全て記録される．

PVM プログラムのデバッグは, 経験的に以下の 3 つのステップに分けることができる．まず第一に, 可能ならば, 単一のプロセスでプログラムを実行し, 他の逐次プログラムと同様にデバッグを行う． このステップの目的は, 並列処理とは無関係な, インデックスのバグや論理的なエラーを取り去ることにある．

第二に, 一つのマシンのみを用いて 2 〜 4 プロセスでプログラムを実行する．PVM はマルチタスクでこれらのプロセスを一つのマシンで実行する．このステップの目的は, 通信に関する構文や論理の誤りをチェックすることにある．例えば, 送信側ではメッセージのタグに 5 を用いているのに, 受信側ではタグが 4 のメッセージを待っている, ということがある．もっとよくある間違いは, 一意でないメッセージタグを使っている場合であり, 必ず発見すべきである． 例えば, 同じメッセージタグを常に使って, 3 つに分割されたメッセージを初期データとして送信する場合を考える． このとき, 受信側プロセスは, どのメッセージにどのデータが入っているかを決定することができない．PVM はどんなメッセージであっても送信元とタグさえ合っていれば, 制御を返す．従って, 送信元とタグから一意にメッセージの内容を特定するのは, 全てユーザの責任である． 一意でないタグのエラーのデバッグは, 時として非常な困難を伴う．なぜならば, こうしたエラーは微妙な同期のタイミングに左右され, 実行を繰り返しても再現性がないからである．PVM エラーコードや簡単な出力命令を用いても, こうしたエラーを検出できない場合は, 一つあるいは全タスクをデバッガーの下で起動し, プログラムを完全にデバッガーの制御下で実行することができる．デバッガーは他のタスクとのメッセージのやりとりを妨げずに,dbx の下でブレークポイント, 変数トレース, シングルステップ実行, トレースバック, 実行及び停止を各プロセス毎に可能にする．

第三のステップでは,2 〜 4 プロセスのプログラムを複数のマシンを使って実行する． このステップの目的は, ネットワーク遅延による同期のエラーをチェックすることにある．このステップで検出されるエラーとしては, メッセージの到着順序に依存したアルゴリズム, ネットワークの遅延に依存したロジックの誤りによるデッドロックがある． このステップでも再びデバッガーを使うことになるが, 今度はそれほど有効ではない． なぜなら, デバッガーはそうしたタイミングのエラーをシフトしたり隠してしまうからである．

10　インプリメントの詳細

PVM は, バーチャルマシンの各ホストにつき一つのデーモン (pvmd3) の集合である．以後 pvmd3 は pvmd と呼ぶこととする．各ユーザは, 互いに協調して動作するデーモンの集合を設定し実行する． そのために, マシンの集合を定義する．あるユーザの pvmd は, 他の

ユーザのそれと干渉することはない．PVM における pvmd は非特権ユーザ ID で動作し，一人のユーザに対してのみサービスを提供するよう設計されている． その結果, セキュリティの危険を抑制し, 他の PVM ユーザに与える影響を最小にしている．バーチャルマシンは,libpvm3.a とリンクした実行モジュールを, タスクとして実行する．pvmd がタスクに提供しているシステムコールライクなインターフェースには, ホストでの実行, プロセスの制御, 通信, 障害検知等がある．

PVM バージョン 3 の概要を図 10 に示す. **テキストとは独立したファイル pvminternal.ps に入っている** 図 10 では, 4 個のタスクを実行する 3 台のホストを示している．その中の 1 台は “コンソール” であり, ここでユーザはタスクの一覧を表示させたり, マシンの再設定を行ったりする．幾つかのファイルや通信ソケットも併せて示しており, その詳細を以降で説明する．

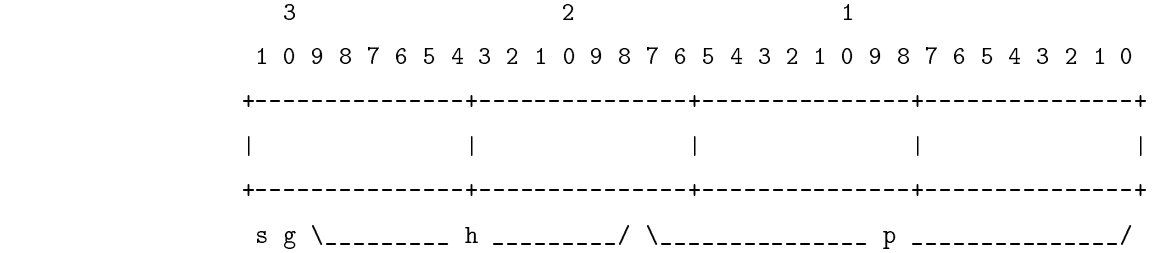
pvmd 間の通信は,UDP ソケットを通じて行われる．信頼性のあるパケット配送システムが UDP の上位に実装されている． TCP ソケットが用いられるのは,pvmd とローカルの pvm タスクとの間の通信, 同一ホスト内にあるタスク間の直接通信, あるいは異なるホスト間で PVM が直接通信の “指示” を受けた場合である．マシン固有の通信ルーチン (メッセージハードウェアを利用) は,Intel i860 のようなマルチプロセッサの内部で利用されている．マシン固有の共有メモリを介したメッセージの交換は,KSR や IBM Poewer/4 のようなマシンにおいてまもなく利用可能となる予定である．PVM では, バーチャルマシンの各ホストは,IP を経由して互いに直接接続できることを仮定している．すなわち, 任意のホスト間でメッセージは直接送られ,pvmd が中継をすることはない． 必要ならば,pvmd に第 3 者経由でメッセージを送らせることもで容易であるが, 今のところ実装されていない．

10.1 TID の詳細

タスク ID と呼ばれる 32 ビット整数は, バーチャルマシンにおける pvmd, タスク及びタスクのグループの特定に用いられる．バーチャルマシン全体で tid は一意であり, ホストの UNIX プロセス ID がプロセスを特定するのと同様である．tid 自身は再利用されるが, 同時に 2 つのタスクが同じ tid を共有することはない．

システムの外側から見る限り tid は隠蔽されており, ユーザはライブラリに提供された関数を除いては, それらを解釈あるいは変更すべきではない．tid は, 名前と分散したタスクとの結び付けを容易にするとともに, メッセージのルーティングの助けとなるように設計されている．

tid は, 以下の図に示すように 4 つのフィールドに分かれている．以下の図は, 議論を進めやすくするためにフィールド分けしたものであり, 早晩変更されるか (動的に変化する可能性もある), 順序が異なることもありうる．



s,g, 及び h の 3 つのフィールドは, グローバルな意味を持つ．即ちバーチャルマシンの各 pvmd は全く同じ解釈を行う．h フィールドはバーチャルマシンと関連したホスト番号を格納する．グループに所属したならば, 各 pvmd は一意なホスト番号を割り当てられる．しかるに, 各 pvmd はバーチャルマシンのアドレス空間の一部を割り当てられたことになる．ホスト番号 0 は, ローカルホストあるいは “シャドウ”pvmd の参照に用いられるとともに, コンテキストに依存した幾つかの操作にも利用される． バーチャルマシンにおける最大のホスト番号は 2^{h-1} に制限 (現在のところ 4096) されるが, 十分であろう．

各 pvmd は,p フィールドに独自の意味を与えることができる．例外は全て 0 の場合で, これは pvmd 自身を表す． 例えば UNIX ワークステーションにおいては, ビット幅さえ十分であれば UNIX プロセス ID を直接 p フィールドに格納することも可能である． マルチプロセッサにおいては,p フィールドは更にプロセッサフィールドとそうでない部分に分けることができる．ホスト 1 台あたりの番号タスク数は, tid フィールドによって制限される．現在のところ,p フィールドは 18 ビット割り当てられているので,1 ホストでは同時に 262143 タスクが存在することができる．

こうした tid の設計は, 実装上以下の 3 つ点で重要となる．まず最初に, 各 pvmd はプライベートなアドレス空間を持つので, 他のホストとの通信を行うことなく, ローカルに tid をタスクに割り当てることができる． 次に, メッセージはどのバーチャルマシンからでも, アドレス参照を必要とせずに直接宛先ホストに配送される．これはホスト番号が tid にコード化されていることによるものである． その結果タスクのデータベースを集中して持つ必要がなくなり, バーチャルマシンのサイズの増大に伴って発生する通信ボトルネックを抑制することができる．最後に, マルチプロセッサで稼働する pvmd は割り当てられたアドレス空間を自由にカスタマイズして利用することができる．それでも依然として, タスクはシステムのどこからでも直接アドレス可能である．

tid が保持するデータは, 各フィールドの値の範囲により分別することができる．

	_s_	_g_	_h_	_p_
タスク識別子	0	0	1.. ホスト最大値	1.. ローカル最大値
pvmd 識別子	1	0	1.. ホスト最大値	0
タスクからローカルの pvmd を参照	1	0	0	0
pvmd から自分自身を参照	1	0	0	0
マルチキャストのアドレス	0	1	1..maxhost	x
エラーコード	1	1	小さな負の数	

例えば, ホスト 12 上のプロセス 3153 の tid は 0-0-12-3153 となる．

10.2 メッセージ

PVM では, メッセージは高レベル関数によってブロック化される．ソースコードのおよそ半分が, メッセージのパック並びにアンパック, フラグメント化, フラグメントの配送に費されている． pvmd 及びタスクが用いるメッセージのフォーマットは類似しているので, タスクは,pvmd と同じメッセージパック関数を用いて, 他のタスクや pvmd のためのメッセージを構成することができる．タスク及び pvmd は, どの tid に対してもメッセージを送ることができるが, pvmd とその管理下のタスクとの通信は現在のところできない．以降で

理由を述べる.

10.3 Libpvm の内部

10.3.1 pvmd との接続

本節では, ジェネリックな libpvm を用いた pvmd とその管理下のタスクとの通信の方法について述べる. 標準的な UNIX のホストを利用しているものとする.

まず最初に, libpvm は,pvmd が起動時に書き込みを行っている /tmp/pvmd.<uid> を読み込む. この中に,pvmd がタスクから接続要求を受けるためのソケットアドレスが,“IP アドレス: ポート” の形式で書かれている. IP アドレスフィールドは, ローカルホストにおいては通常 “7f000001” である. libpvm は, このアドレスに対してソケットを接続し,pvmd に利用開始のメッセージを送る.

接続する際に,pvmd と libpvm は互いに相手が同一ユーザであることを確認しなくてはならない. この認証は, 他のユーザのタスクが接続しないように, また他の pvmd に対してサービスを要求しないようにするために必要である. まず pvmd とタスクは, 各々のユーザ ID で所有権限と書き込み権限を持つファイルを生成する. 次に互いのファイル名を交換し, 互いに相手のファイルに書き込み合いをして各々ファイルをチェックする. もちろん, こうした認証はファイル権限に基づくものであるため, 容易にクラックされる可能性がある. 注意すべき点は, この認証ではユーザの libpvm の変更を許しており, 従って libpvm が pvmd をだますこと (例えば, 嘘のプロセス ID を知らせる) ができるということである.

タスクが pvmd と再接続するときは,pvmd に pid を知らせる必要がある. この pid は,spawn 操作における fork() からの返り値である. spawn 操作では,pvmd はタスクのコンテキストを生成し,fork() からの返り値の pid を格納する. このコンテキストはタスクの起動しバッファに再接続する前に, そのタスク宛のメッセージの到着を可能にする. デバッガーの下でのタスク実行中は,fork() によって返される. pid は常にデバッガーのものであり, その一方でタスクはデバッガーの子プロセスなので pid は不確定である. タスクの exec() 前に,pvmd はタスクへの要求 pid を環境変数 PVMPID に設定する. 環境変数 PVMPID が設定されていれば, libpvm はその pid を pvmd との認証に用いるとともに, 実際の pid もまた送る.

10.4 pvmd の内部

10.4.1 タスクの管理

各 pvmd は, 自身が管理するホストにおいて実行中の全タスクのリストを保持している. 各タスクのコンテキストを保持する構造体を以下に示す.

```
struct task {
    struct task *t_link;      /* task の連鎖 */
    struct task *t_rlink;
    int t_tid;                /* タスクの ID */
    int t_ptid;               /* 親タスクの ID */
    int t_flag;               /* 状態 */
```

```
    int t_pid;                /* UNIX の PID */
    int t_sock;               /* タスク - デーモン間の TCP ソケット */
    struct sockaddr_in t_sad; /* ソケットのアドレス */
    int t_salen;              /* ソケットアドレスの長さ */
    struct waitc *t_wait;     /* 待ちチャネル */
    struct pkt *t_txq;        /* タスクへの送信待ち行列 */
    struct pkt *t_rxp;        /* タスクからのパケットの再構成 */
    struct mesg *t_rxm;       /* タスクから我々へのメッセージの再構成 */
    int t_out;                /* 標準出力 / 標準エラー出力へのパイプ */
    char *t_authnam;          /* デーモン認証ファイル名 */
    int t_authfd;             /* デーモン認証ファイル記述子 */
    char *t_a_out;            /* a.out の名前 */
    struct mca *t_mca;        /* 現在のところタスクが使用中 */
}
```

10.4.2 pvmd 待ちコンテキスト

pvmd に要求する操作には, 更に他のデーモンのサービスを必要とするものがある. 例としては, リモートのホストの状態のチェックがある. pvmd は, 操作完了を待つ間もブロックしないので, 重要なデータを全て “待ちコンテキスト” (waitc) に記録した上で, パケットの配送や他の要求の処理を続行する. 応答が返って来ると,waitc の中にしまっておいたデータを取り出し, 元の操作を再生する.

waitc の定義を以下に示す.

```
struct waitc {
    struct waitc *wa_link;  /* 待ちコンテキストの連鎖 */
    struct waitc *wa_rlink;
    int wa_wid;             /* 一意な識別子 */
    int wa_kind;            /* 待ちの理由 */
    int wa_on;              /* 待たせる側 */
    int wa_tid;             /* 待つ側 */
    int wa_dep;             /* 我々が依存するリモートの待ち ID または 0 */
    struct waitc *wa_peer;  /* dll of peers */
    struct waitc *wa_rpeer;
                            /* 以下は, 待ちの種類に関連するもの */
    struct mesg *wa_mesg;   /* 作られているメッセージ */
    void *wa_spec;          /* 待ちの種類に関連するデータ */
}
```

pvmd の中では, 各待ちコンテキストは, 待ち ID(wid) なる一意な識別番号を持つ. wid はサービス要求とともに渡され, 応答とともに返される.

waitc には, 一部の操作のみが必要とするデータを保持するためのフィールドがいくつか

ある． 共通に用いられるフィールドには, 操作に対して順次付される番号 wid, 操作の種類を保持する kind, 待つ側と待たせる側を示す tid と on がある.

待ちコンテキストは,1 つ以上のリモートホストを含むような追跡操作でピアされる． 同一ピアグループ内の各 waitc は, プライベートにリモートホストの情報を持つとともに, ローカルオペレーションの情報を共有する． 各リモート pvmd が応答すると waitc はフリーされ, ピアグループは 1 つの waitc が残るまで全て壊される．ローカルの pvmd が応答すると操作が再開される.

10.4.3 マシン再設定

バーチャルマシンへのホストの追加及び削除は, いつでも行うことができる． 例外は, マスター pvmd(最初の pvmd) で, マシンが存在する限り常にメンバーでなくてはならない.

ホストテーブルは全ての pvmd にわたって同期している必要があり，マスター pvmd がこれを管理する.

どのタスクも pvm_addhost( ) 関数によって新しいホストの追加を要求することができる. 要求がマスター pvmd に中継されると, マスター pvmd は新しいホストでの pvmd の起動と設定を行った後, 新しいホストテーブルを全スレーブ pvmd にブロードキャストする.

マスター pvmd は, 新しい pvmd を起動するのに rsh( ) と rexec( ) の両方を使う． この操作はハングアップしやすく遅延も大きくなりがちなので, ダーティワークをこなすためのシャドウ pvmd を生成する. シャドウ pvmd は,pvmd そのものであるが, マスター pvmd 以外の pvmd 及びタスクとは一切通信しない. ホストテーブルのスロット 0 に置かれる．シャドウ pvmd が作業を始めると, 結果は全て通常の pvmd-pvmd メッセージチャネルを通じて返される． これは, シャドウ pvmd との通信のために特別な機構 (ファイルに書き込む) の追加を避けるためである.

新しい pvmd は, 起動の際にマスター pvmd と設定に関する情報を少しだけ, 交換する．rsh により張られた接続を介した通信を以下に示す.

```
マスター -> スレーブ                        スレーブ -> マスター
------------------                          ------------------
(rexec) pvmd -s [-dxx]
                                            ddpro<2309>
name<tonka> n<2> ip<80a9c93c:0000> n<1> ip<80a9c9e4:1388> mtu<4096>
                                            arch<SYMM> ip<80a9c93c:1388> mtu<4096>
ep<pvm3/bin/%>
                                            ok
```

スレーブが, 自分自身とマスターのみの一時的なホストテーブルを持っているときは，ソケットをクローズし, 残りを通常のメッセージを通じて行う．古いスレーブデーモンも含むよう更新されたホストテーブルを受信する.

10.4.4 pvmd の障害検出

障害検出における pvmd の役割は, アプリケーションのハングアップ等のホストの障害を, メッセージのような検出可能なイベントに変換することにある． pvmd 自身はホストの障

害とは関係ない. なぜなら,pvmd はメッセージの配送とデータベースの機能のみに特化しているからである. マスター pvmd だけは例外で, 常に全システムからアクセスできなくてはならない. なぜならば, 調停といったマスター pvmd を必要とする操作が存在するにもかかわらず, マスター pvmd の機能を肩代りするような方法は実装されていないためである. 従って, マスターとの接続を失った pvmd は後始末をして終了し, それを知ったマスターは終了したことをマークする.

ホストの障害は, メッセージを送ろうとした時に検出される. ACK として送ったパケットが返って来ず, ついには pvmd があきらめると, ホストテーブルから障害を起こしたホストを削除する. また, そのホストで中断していた全ての操作を,waitc に保存していた情報を用いて完全に終了させる. 以降の障害ホストへの要求は全て負の状態で返す.

マシンがアイドル状態になる, 例えば全てのタスクがメッセージ受信待ちに入った時には, 障害のメッセージが生成されることはない. この理由としては, 各 pvmd はアイドルタイマーを持っており, 他の pvmd に対して周期的にパケットを送出しているため, トラフィックがゼロには決してらない.

一旦ホストが障害を起こすと,pvmd は永久に死んだと見なされる. しばらくしてから, 同じホストで別の pvmd が起動するが, それは全くの別物である. どちらにしても, 障害ホストで実行中であったタスクを回復する方法はない.

10.5 マルチプロセッサでのインターフェース

PVM バージョン 3 は, マルチプロセッサ固有の関数を呼び出すようにコンパイルすることも可能である. システムに特化したメッセージパッシングを PVM アプリケーション上で実現できる. マルチプロセッサ内の 2 つのノード間のメッセージは直接配送される. 一方インターネット上のホストを宛先とするメッセージは, マルチプロセッサ上の PVM デーモンに送られ更に配送される. 共有メモリシステムでのデータの移動は, 共有バッファプールとロックのためのプリミティブを用いて実装されている.

2 つの例で,PVM バージョン 3 におけるマルチプロセッサの利用法を示す. 最初の例では,Intel の Paragon 上のノードでの PVM タスクの実行と,PVM 送信ルーチン呼び出しによるノード間のデータ転送とを仮定する.PVM は宛先 tid と送信者 tid の h フィールドを比較し, メッセージが同一ホスト内に留まることを知る. 宛先タスクへ直接メッセージを送るために, PVM から更に Paragon 固有の NX メッセージパッシングルーチンを呼び出し, 引数には宛先 tid の p フィールドを与える. 次の例は,PVM タスクがインターネット上のホストで実行中のタスクにメッセージを送る場合である. libpvm は,h フィールドからメッセージの宛先が同一ホストの範囲外にあることを知ると, NX メッセージパッシングルーチンによりサービスノード上の pvmd プロセスへメッセージを転送する. Paragon のサービスノードはネットワークへアクセスできるため,pvmd は, 宛先ホストで実行中の PVM デーモンへメッセージを配送する. リモートのデーモンは, 宛先 tid の p フィールドから宛先タスクを求め, メッセージを配送する. このようにして PVM アプリケーションは, 複数の異機種並列コンピュータにタスクを分散し, タスク間でのメッセージ交換を実現することで, 全体を一つの大きな並列コンピュータで扱うことができる.

マルチプロセッサで稼働する PVM デーモンは, 外部ネットワークとマルチプロセッサの

ノードの両方からのメッセージを監視する. その実装効率は, オペレーティングシステムとメッセージパッシングソフトウェアに大きく依存する. マルチプロセッサへ PVM へ移植する際に最も大きな障害となるのは, ノード上のタスクと pvmd との通信の実現である. 例えば, ベンダーが提供する通信ルーチン群が select() を用いた UNIX のファイル I/O と互換性を持つことはまずない. 自明な解決法としては, 双方のインターフェースを常にポーリングするループを pvmd に構成することが考えられる. この方法は, 大量の通信が続く場合は有効であるが, タスクがアイドル状態にある場合には CPU 時間を浪費してしまう.

また別の解としては,pvmd はノード上のタスクに直接送信することはできるが, 受信においては常にバッファプロセスを介するようにする. このとき, バッファプロセスは, ノードからメッセージが到着するまでブロックし, 普通の select() ループで待つ pvmd に到着したメッセージを pvmd にフォワードする. しかしながら, この方法では余分なコピーというペナルティが発生してしまう.

ハイブリッドな解決法としては, 忙しい間は pvmd にポーリングを許し, 余分なコピーを発生させずにメッセージを直接転送可能にするが, タスクが通信しないときは “パワーダウン” させる. pvmd は普通の select() ループで待ち, 300msec 毎にタイムアウトして各ノードをポーリングし, メッセージの有無をチェックする. メッセージが到着すると,pvmd はポーリングに切替え, 出来る限り速くパケットを受信する. この方式は, 外部から MPP ノードへの通信に伴う遅延をかなり大きくしてしまうが, 帯域を下げられ,CPU 時間のコストも小さい. 直接データパスでなければ, ノードからデータ準備のシグナルを pvmd に送ることで,2 番目のプロセスからポーリングを除くことができる.

マルチプロセッサの統合のための要点を以下に示す.

1. p フィールドにローカルな意味を持たせる. pvmd だけがローカル tid を示す部分を, 分割情報あるいはプロセッサ番号等に解釈することができる. あるシステムでは識別子を直接 p フィールドにマッピングできると思われるが, そうでない場合には, 何らかの手法が必要となる.

2. pvmd と libpvm を変更し, マシン固有のメッセージパッシングルーチンを用いて, メッセージを送信できるようにする.

3. プロセスを生成できるように,pvmd に変更を加える. これには, アドレスの分割やノードプロセッサへのロード等が含まれる. MPP における pvm_spawn() を定義することになる.

4. libpvm に変更を加え, 同一 MPP 内では pvmd を介すことなくメッセージを配送できるようにする.

上記のメッセージ転送システムを構築しておけば, 高レベルな操作の移植が容易になる. システムの変更は段階的に, 即ち最初は確実な動作を目指し, 次にパフォーマンスの最適化を行うべきであろう.

10.6 デバッグのために

非サポートではあるものの, デバッグのための手法がいくつか pvmd と libpvmd にビルトインされている. 簡単なドキュメントを用意しており, 変更はいつでも行われる. しかし

ながら, 以下に手短に述べることも時には役に立つであろう.

malloc のラッパー

開発を容易にするために pvmd と libpvm は, 健全性をチェックできる malloc のライブラリとして,`imalloc` を使用している. imalloc 関数は, 標準 libc の関数である malloc,realloc 及び free のラッパーである. 数個のエラーを検出すると imalloc はプログラムを終了するので, デバッガーでエラーを検出することができる.

imalloc でチェックできるエラーとそのための関数を以下に示す.

1. malloc に要求する大きさが不正な値かどうかチェックする. 大きさ 0 は 1 に変更されるので, 成功する.
2. 各割り当てブロックは, 全てハッシュ表で追跡するので, 同一空間に対する不正な free() を検出する.
3. I_malloc() 及び i_malloc() は, ブロックの先頭と末尾に疑似乱数のパターンを書き込むので,i_free() の際には, これまでにブロックの終りに不正な書き込みがあったかどうかをチェックできる.
4. I_free() は, ゼロクリアしてからフリーするので, フリー後の参照は失敗し, それを明らかにする.
5. 各ブロックには連続番号と使用状態を示す文字列が付されている.
6. ヒープ空間はいつでも i_dump() でダンプすることができ, オプションとして健全性をチェックできる. これはメモリーリークの追跡に役立つ.

こうしたチェックに要するオーバヘッドは非常に大きいため, デフォルトでは無効となっている. PVM の Makefile で USE_PVM_ALLOC を有効にすれば良い.

デバッグのマスク

pvmd と libpvm は, デバッグのログを残す機能を持っている. 種々の出力命令を有効にするためのマスクがある. マスクの各 bit は, イベントの各クラスを有効にする. pvmd の -d オプションを付けると, デバッグマスクがセットされる. マスター pvmd のデバッグのマスク値は, スレーブ pvmd にも継承される. libpvm のデバッグマスクは pvm_setdebug() でセットされる.

pvmd のデータ構造

コンソールの隠しコマンドに,“tickle” がある. これは pvmd のデータ構造をダンプさせることができる. コンソールのプロンプトに `help tk` と入力すれば, 調べることのできるデータ構造のリストが表示される.

11 サポート

PVM 利用の手助けとなる道筋を幾つか挙げる．PVM ユーザがアイデア, 技法, 成功例及び問題点を自由に交換するための掲示板がある．ニュースグループの名前は `comp.parallel.pvm` である．Cray Research, Convex, SGI, IBM, Intel, DEC, KSR 及び Thinking Machines の各社は, 自社システムでの PVM サポートを表明している．PVM 開発者は, 時間の許す限り PVM のサポートも行っている． PVM の問題点, 疑問点は,`pvm@msr.epm.ornl.gov` に送れば, 迅速で親切な返答を得ることが出来る． 最初の PVM ユーザーズグループ会議は 1993 年に Knoxville で開催された．会議で使われたスライドのポストスクリプト版は, netlib@ornl.gov の pvm3 ディレクトリから入手できる．

表 2: PVM 3.1 ルーチンのエラーコード

エラーコード		意味
PvmOk	0	成功
PvmBadParam	-2	パラメータ値不良 (負のメッセージ ID など)
PvmMismatch	-3	バリアの待ち合わせタスク数の不一致
PvmNoData	-5	バッファの終りを超えて読もうとした
PvmNoHost	-6	指定されたホストは存在しない
PvmNoFile	-7	指定された実行ファイルは存在しない
PvmNoMem	-10	メモリが確保できない
PvmBadMsg	-12	受けとったメッセージがデコードできない
PvmSysErr	-14	pvmd と通信できない / システム・エラーが発生した
PvmNoBuf	-15	カレント・バッファが割り当てられていない
PvmNoSuchBuf	-16	メッセージ ID 不良
PvmNullGroup	-17	グループ名に空文字列は使えない
PvmDupGroup	-18	すでに指定されたグループに所属している
PvmNoGroup	-19	指定された名前のグループは存在しない
PvmNotInGroup	-20	指定されたタスクはグループに所属していない
PvmNoInst	-21	指定されたインスタンスはグループに所属していない
PvmHostFail	-22	指定されたホストにアクセスできなかった
PvmNoParent	-23	親タスクは存在しない
PvmNotImpl	-24	その機能は実装されていない
PvmDSysErr	-25	pvmd のシステム・エラー
PvmBadVersion	-26	pvmd 同士のプロトコルのバージョンが合っていない
PvmOutOfRes	-27	リソースを使いつくした
PvmDupHost	-28	ホストはすでにコンフィギュレーション済み
PvmCantStart	-29	新たにスレーブの pvmd を起動できなかった
PvmAlready	-30	すでに実行中である
PvmNoTask	-31	指定されたタスクは存在しない
PvmNoEntry	-32	指定された名前とインデックスの組は存在しない
PvmDupEntry	-33	指定された名前とインデックスの組がすでに存在する

A　PVM3.0 ルーチン・リファレンス

付録として,PVM 3.0 ルーチンをアルファベット順で示した. 各ルーチンについて,C 及び FORTRAN 版の利用法, 例及び特徴について説明する.

バーチャルマシンに 1 つ以上のホストを追加する

形式

C `int info = pvm_addhosts( char **hosts, int nhost, int *infos )`

Fortran `call pvmfaddhost( host, info )`

パラメータ

- `hosts` – 追加するマシン名の文字列へのポインタの配列. 指定したマシンには,pvmd を既にインストールしてあることと, ユーザが有効なアカウントを所持していることが必要である.
- `nhost` – 追加するホストの数.
- `infos` – nhost の大きさの整数配列. 個々のホストを追加する毎に返される状態コードを格納する. 負値はエラーを示す.
- `host` – 追加するマシン名の文字列.
- `info` – ホストを追加して返される状態コードを格納する. 負値はエラーを示す.

説明

`pvm_addhosts` ルーチンは,`hosts` で示された一連のコンピュータを, 現在バーチャルマシンを構成している設定に追加する. pvm_addhosts が成功した場合は,`info` は 0 である. 何らかのエラーが起こった場合は `info` < 0 となる. 配列 `infos` をチェックすれば, エラーを起こしたホストを特定できる.

FORTRAN ルーチン `pvmfaddhost` は,1 回の呼び出しで 1 つのホストを追加する.

ホストに障害が発生しても,PVM システムは機能を保つ. ユーザは, 本ルーチンを使うことで,PVM アプリケーションの耐障害性を高めることができる. pvm_mstat や pvm_config を用いれば, アプリケーションからホストの状態を要求することが出来る.

ホストは障害を起こすと, 自動的に設定から除去される. pvm_addhosts() を使うことで, アプリケーションがホストを追加できる. ホストの障害に対する耐性をアプリケーションに持たせるのは, 開発者の役目である. この機能の別な応用としては, 利用可能になったホストの追加が考えられる. 例えば, 週末にはホストを追加する, アプリケーションが自ら計算パワーを必要と判断しホストを追加する, 等が挙げられる.

pvm_addhosts 及び pvm_delhosts は, バーチャルマシン全体の同期を必要とするため, その操作は高くつく. 全てのコンピュータに設定の変更が反映されなくてはならないだけでなく, 削除に関してコンピュータが行う操作も慎重に扱われる必要がある.

例

```
C:       static char *hosts[] = {
                    "sparky ",
                    "thud.cs.utk.edu",
         };
         info = pvm_addhosts( hosts, 2, infos );
Fortran: CALL PVMFADDHOST( 'azure', INFO )
```

エラー

名前	原因
PvmBadParam	引数の値が無効である.
PvmSysErr	ローカル pvmd の応答がない.
PvmOutOfRes	PVM のシステム資源が不足.

PVM に直接タスク間通信を指示する

形式

C `int info = pvm_advise( int route )`

Fortran `call pvmfadvise( route, info )`

パラメータ

`route` – ダイレクトタスク間リンクの設定を PVM に指示するための整数を以下に示す.

	route オプション		
	PvmDontRoute	1	呼び出しタスクのダイレクトリンクを不可にする.
ン	PvmAllowDirect	2	ダイレクトリンクを可能にするが, 要求はしない.
	PvmRouteDirect	3	ダイレクトリンクを要求する.

`info` – エラー状態を返す.

説明

pvm_advise ルーチンは, 以降の通信においてタスク間ダイレクトリンク (TCP を利用) を設定するかどうか指示を与える. 一旦リンクを確立すると, アプリケーション終了まで継続する. 双方のタスクが PvmDontRoute を要求しているか, あるいはリソースが不足しているためにダイレクトリンクが確立できない場合は,PVM デーモンのデフォルトの配送経路となる. pvm_advise() は多重に呼び出すことで, ダイレクトリンクを選択的に用いることもできるが, 普通はタスク処理の最初の方で一度だけ設定する. PvmAllowDirect はデフォルトの指示値である. タスク A でこの設定を行うと, タスク A と他のタスクとの間にダイレクトリンクが可能になる. タスク間でダイレクトリンクが確立されれば, タスクの双方がそれを利用してメッセージを送信できる. pvm_advise はエラー状態を info に格納して返る.

ダイレクトリンクは, デフォルトの配送に比べパフォーマンスが向上する. ダイレクトリンクはスケーラビリティに欠けるところが欠点として挙げられる. UNIX の中には, リンクの数を 30 に制限しているものがある.

例

C: `info = pvm_advise( PvmRouteDirect );`

Fortran: `CALL PVMFADVISE( PVMROUTEDIRECT, INFO )`

エラー

pvm_advise より返されるエラー状態を示す.

名前	原因
PvmBadParam	引数の値が無効である.

グループの全プロセスがこれを呼び出すまで, 呼び出しプロセスをブロックする.

形式

C `int info = pvm_barrier( char *group, int count )`

Fortran `call pvmfbarrier( group, count, info )`

パラメータ

`group` – グループ名の文字列. グループは必ず存在しなくてはならない. 呼び出しプロセスはグループに所属していなくてはならない.

`count` – ブロックを通り抜けるまでに,pvm_barrier を呼び出さなくてはならないプロセスの数. 特に要求の無い限り, 指定したグループの所属プロセス数であると思われる.

`info` – ルーチンの状態コードを返す. 負値はエラーを表す.

説明

pvm_barrier ルーチンは, 同一グループの count で示された数のプロセスがこれを呼び出すまで, 呼び出しプロセスをブロックする. count 引数が必要な理由は, グループ内のプロセスが pvm_barrier を呼び出した後に, 他のプロセスが所属を要求する場合があることによる. PVM は, 任意の瞬間にグループ内のどれだけの数のプロセスが待っているかを知ることができない. count を実際より少なく設定することは可能だが, 通常はグループ内のメンバーの総数である. 即ち, pvm_barrier 呼び出しが提供する論理的な機能は, グループの同期である. バリア同期の実行中は, グループ内でそれを実行する全てのプロセスが, 同じ count の値である必要がある. バリア同期が終了すると, グループは再び同じグループ名で pvm_barrier を呼び出すことが出来る.

count に -1 を指定するのは特殊なケースであり,PVM は pvm_gsize() の値, 即ちグループの全メンバー数を用いる. これは, グループを構成した後はメンバーを変更しないようなアプリケーションに有用である.

pvm_barrier に成功すれば info は 0 となり, エラーが起きた場合は, 負となる.

例

```
C:       inum = pvm_joingroup( "worker" );
                .
                .
         inum = pvm_barrier( "worker", S );
Fortran: CALL PVMFJOINGROUP( "shakeers", INUM )
         COUNT = 20
         CALL PVMFBARRIER( "shakeers", COUNT, INFO )
```

エラー

pvm_barrier より返されるエラー状態を示す.

名前	原因
PvmSysErr	pvmd が起動されていない, またはクラッシュした.
PvmBadParam	引数の値が無効である.
PvmNoGroup	存在しないグループ名である.
PvmNotInGroup	呼び出しプロセスは, 指定したグループに所属していない.

アクティブメッセージバッファのデータを直ちにブロードキャストする.

形式

C `int info = pvm_bcast( char *group, int msgtag )`

Fortran `call pvmfbcast( group, msgtag, info )`

パラメータ

`group` – 存在するグループ名の文字列.

`msgtag` – ユーザが整数で定義するメッセージ. msgtag >= 0 でなくてはならない. ユーザがプログラムにおいて, メッセージの種類の識別に用いる.

`info` – ルーチンの状態コードを返す. 負値はエラーを表す.

説明

pvm_bcast ルーチンは, アクティブメッセージバッファに保持されたメッセージを group のメンバー全てにブロードキャストする. 呼び出しタスクが対象グループに所属している場合, バージョン 3.0 および 3.1 では自分自身にもメッセージが送られる. 次の PVM リリースでは, 送信者自身にはブロードキャストメッセージが送られないようになる. どの PVM タスクも pvm_bcast() を呼び出すことができ, グループに所属している必要はない. メッセージの内容は msgtag で識別される. pvm_bcast が成功すれば,info は 0 になる. 何かエラーが発生すると,info < 0 となる.

pvm_bcast は非同期である. メッセージが確実に受信側プロセッサへの通信経路に乗れば, 送信側プロセッサは直ちに計算に復帰する. これは同期通信と対照的である. 同期通信では, 受信側プロセッサが全て受信するまで, 送信側プロセッサは停止する.

pvm_bcast() は, まず最初にグループデータベースをチェックしグループの tid を決定する. これらの tid が, マルチキャストの対象となる. グループがブロードキャスト中に変更されてもブロードキャストには反映されない. 大抵のマルチプロセッサでは, マルチキャストはサポートされない. 普通は, 機種固有の関数として全プロセッサへのブロードキャストのみがサポートされる. こうした機能欠落があるマルチプロセッサでは,pvm_bcast は必ずしも効率の良い通信とはならない.

例

C:

```
info = pvm_initsend(PvmDataRaw);
info = pvm_pkint( array, 10, 1 );
msgtag = 5 ;
info = pvm_bcast( "worker", msgtag );
```

Fortran:
```
CALL PVMFINITSEND( PVMDEFAULT )
CALL PVMFPKFLOAT( DATA, 100, 1, INFO )
CALL PVMFBCAST( 'worker', 5, INFO )
```

エラー

pvm_bcast より返されるエラー状態を示す.

名前	原因
PvmSysErr	pvmd が起動されていない, またはクラッシュした.
PvmBadParam	引数の値が無効である.
PvmNoGroup	存在しないグループ名である.

要求されたメッセージバッファに関する情報を返す.

形式

C `int info = pvm_bufinfo( int bufid, int *bytes, int msgtag, int *tid )`

Fortran `call pvmfbufinfo( bufid, bytes, msgtag, tid , info)`

パラメータ

`bufid` – 特定のバッファを指定するための整数.

`bytes` – メッセージ全体のバイト長を整数で返す.

`msgtag` – 実際のメッセージラベルを返す. msgtag にワイルドカードを指定してメッセージを受信した時に有用である.

`tid` – メッセージの送信元を表す整数. tid ワイルドカードを指定してメッセージを受信した時に有用である.

`info` – ルーチンの状態コードを返す. 負値はエラーを表す.

説明

pvm_bufinfo ルーチンは, 要求されたメッセージバッファに関する情報を返す. 普通は最も最近のメッセージに関して大きさ, あるいは送信元を決定するために用いられる. 任意のメッセージを受信使用とした時に, とりわけ pvm_bufinfo は有用となる. 最初に到着したメッセージの送信元 `tid` と `msgtag` によって動作が決まる. pvm_bufinfo は成功すれば,`info` は 0 になる. 何かエラーが発生すると, `info` < 0 となる.

例

```
C:       info = pvm_recv( -1, -1 );
         info = pvm_bufinfo( bufid, &bytes, &type, &source );
Fortran: CALL PVMFRECV( -1, -1, BUFID )
         CALL PVMFBUFINFO( BUFID, BYTES, TYPE, SOURCE, INFO )
```

エラー

pvm_bufinfo より返されるエラー状態を示す.

名前	原因
PvmNoBuf	受信バッファが一つもない.

現在の設定のバーチャルマシンに関する情報を返す.

形式

C
```
int info = pvm_config( int *nhost, int *narch, int *infos,
                       struct hostinfo **hostp )
struct hostinfo {
    int hi_tid;
    char *hi_name;
    char *hi_arch;
    int  hi_mtu;
    int  hi_spee;
} hostp;
```

Fortran
```
call pvmfconfig( nhost, narch, info )
```

パラメータ

- `nhost` – バーチャルマシンにおけるホストの数を返す.
- `narch` – 現在使用中のデータフォーマットの種類数を返す.
- `hostp` – ホスト情報構造体の配列へのポインタ. 各ホストについて, pvmd のタスク ID, 名前, アーキテクチャ, 最大パケット長, 及び相対速度を含む.
- `info` – ルーチンの状態コードを返す. 負値はエラーを表す.

説明

pvm_config ルーチンは, 現在の設定のバーチャルマシンに関する情報を返す. 得られる情報は, コンソール上でコマンド conf のそれと似ている. pvm_config は成功すれば,info は 0 になる. 何かエラーが発生すると, info < 0 となる.

例

C:
```
info = pvm_config( &nhost, &narch, &hostp );
```

Fortran:
```
CALL PVMFCONFIG( NHOST, NARCH, INFO )
```

エラー

名前	原因
PvmSysErr	ローカル pvmd の応答がない.

バーチャルマシンから 1 つ以上のホストを追加する

形式

C `int info = pvm_delhosts( char **hosts, int nhost, int *infos )`

Fortran `call pvmfdelhost( host, info )`

パラメータ

`hosts` – 削除するマシン名の文字列へのポインタの配列.

`nhost` – 削除するホストの数.

`infos` – nhost の大きさの整数配列. 個々のホストを削除する毎に返される状態コードを格納する. 負値はエラーを示す.

`host` – 削除するマシン名の文字列.

`info` – ホストを削除して返される状態コードを格納する. 負値はエラーを示す.

説明

pvm_delhosts ルーチンは,hosts で示された一連のコンピュータを, 現在バーチャルマシンを構成している設定に削除する. 対象となるホスト上で実行中の全 PVM プロセスを強制終了した後, そのホストを削除する. pvm_delhosts が成功した場合は,info は 0 である. 何らかのエラーが起こった場合は info $<$ 0 となる. 配列 infos をチェックすれば, エラーを起こしたホストを特定できる.

FORTRAN ルーチン pvmfdelhost は,1 回の呼び出しで 1 つのホストを削除する.

ホストは障害を起こすと, 自動的にバーチャルマシンの設定から除去され, PVM システムは機能を保つ. pvm_mstat や pvm_config を用いれば, アプリケーションからホストの状態を要求することが出来る. ホストの障害に対する耐性をアプリケーションに持たせるのは, 開発者の役目である.

pvm_addhosts 及び pvm_delhosts は, バーチャルマシン全体の同期を必要とするため, その操作は高くつく. 全てのコンピュータに設定の変更が反映されなくてはならないだけでなく, 削除に関してコンピュータが行う操作も慎重に扱われる必要がある.

例

C:
```
static char *hosts[] = {
            "sparky ",
            "thud.cs.utk.edu",
};
info = pvm_delhosts( hosts, 2 );
```

Fortran:
```
CALL PVMFDELHOST( 'azure', INFO )
```

エラー

pvm_delhosts で返される可能性のあるエラー状態を以下に示す.

名前	原因
PvmBadParam	引数の値が無効である.
PvmSysErr	ローカル pvmd の応答がない.
PvmOutOfRes	PVM のシステム資源が不足.

ローカル pvmd に PVM から離脱することを知らせる.

形式

C `int info = pvm_exit( void )`

Fortran `call pvmfexit( info )`

パラメータ

`info` – ルーチンの状態コードを返す. 負値はエラーを表す.

説明

pvm_exit ルーチンは, ローカル pvmd に PVM から離脱することを知らせる. これは, プロセスを強制終了させるものではなく, 他の逐次プロセスと同様に処理を続行することができる.

pvm プロセスは, 処理を中止あるいは終了する前には pvm_exit() を呼び出した方が良い. 特に,pvm_spawn で起動されたのでなければ, 必ず呼び出さなくてはならない. そうしなければ, プロセスはクラッシュしたのか正常終了したのかを PVM が知ることができない.

例

C:
```
pvm_exit();
exit();
```

Fortran:
```
CALL PVMFEXIT(INFO)
STOP
```

エラー

名前	原因
PvmSysErr	ローカル pvmd の応答がない.

要求されたメッセージバッファに関する情報を返す.

形式

C `int info = pvm_freebuf( int bufid )`

Fortran `call pvmffreebuf( bufid, info )`

パラメータ

`bufid` – メッセージバッファ識別子を表す整数.

`info` – ルーチンの状態コードを返す. 負値はエラーを表す.

説明

pvm_freebuf ルーチンは, bufid で特定されるメッセージバッファのメモリ及びテーブルを解放する. メッセージバッファは pvm_mkbuf,pvm_initsend, 及び pvm_recv で生成される. pvm_freebuf は成功すれば,info は 0 になる. 何かエラーが発生すると, info < 0 となる.

pvm_mkbuf で生成した送信バッファは, 送信してもはや必要でなくなったら, pvm_freebuf を呼び出して解放した方が良い.

多重バッファの方針の下では, 受信バッファは, 保存していないのであれば普通解放する必要はない. しかし,pvm_freebuf は受信バッファを破壊することもできる. メッセージが到着した後で何らかのイベントが発生して不要になったメッセージは, 破壊すればメモリの節約になる.

普通は, 多重送信バッファや受信バッファは必要ではなく, ユーザは pvm_initsend ルーチンを使ってデフォルトの送信バッファをリセットするのが簡単である.

多重バッファが有効な例としては,PVM を使ったライブラリやグラフィカルユーザインターフェースを利用しながらアプリケーション自身の通信には影響を与えたくない場合が考えられる.

多重バッファを利用する時, 一般的には各メッセージのパック毎にバッファの生成と解放を繰り返す. 実際には,pvm_initsetnd は単に pvm_freebuf の後に続けてデフォルトバッファの pvm_mkbuf を呼び出している.

例

```
C:       bufid = pvm_mkbuf( PvmDataDefault );
         info = pvm_freebuf( bufid );
Fortran: CALL PVMFMKBUF( PVMDEFAULT, BUFID )
         CALL PVMFFREEBUF( BUFID, INFO )
```

エラー

pvm_freebuf より返されるエラー状態を示す.

名前	原因
PvmBadParam	引数の値が無効である.
PvmNoSucbBuf	与えられた bufid が無効である.

グループでの PVM プロセスのインスタンス番号を返す.

形式

C `int inum = pvm_getinst( char *group, int tid )`

Fortran `call pvmfgetinst( group, count, inum )`

パラメータ

`group` – 存在するグループ名の文字列.

`tid` – PVM プロセスのタスク識別子.

`inum` – ルーチンが返すインスタンス番号. インスタンス番号は 0 から順に数え上げられる. 負値はエラーを表す.

説明

`pvm_getinst` ルーチンは, グループ名 `group` と PVM タスク識別子 `tid` を引数にとり, 入力によって決まる一意なインスタンス番号を返す. pvm_getinst は, 成功すれば `inum` は >= 0 となり, エラーが起きた場合は, 負となる.

例

```
C:       inum = pvm_getinst( "worker", pvm_mytid( ) );
         --------
         inum = pvm_getinst( "worker", tid[i] );
Fortran: CALL PVMFGETINST( 'GROUP3', TID, INUM )
```

エラー

`pvm_getinst` より返されるエラー状態を示す.

名前	原因
PvmSysErr	pvmd が起動されていない, またはクラッシュした.
PvmBadParam	引数の値が無効である.
PvmNoGroup	存在しないグループ名である.
PvmNotInGroup	tid は指定したグループに所属していない.

アクティブな受信バッファのメッセージバッファ ID を返す.

形式

C `int bufid = pvm_getrbuf( void )`

Fortran `call pvmfgetrbuf( bufid )`

パラメータ

`bufid` – アクティブな受信バッファのメッセージバッファ ID として返される整数.

説明

`pvm_getrbuf` ルーチンは, アクティブな受信バッファのメッセージバッファ識別子を `bufid` に返す. 呼び出し時にアクティブな受信バッファが存在しないときは 0 を返す

例

C: `bufid = pvm_getrbuf();`

Fortran: `CALL PVMFGETRBUF( BUFID )`

エラー

`pvm_getrbuf` は, エラー状態を返さない.

アクティブな送信バッファのメッセージバッファ ID を返す.

形式

C `int bufid = pvm_getsbuf( void )`

Fortran `call pvmfgetsbuf( bufid )`

パラメータ

`bufid` – アクティブな送信バッファのメッセージバッファ ID として返される整数.

説明

`pvm_getsbuf` ルーチンは, アクティブな送信バッファのメッセージバッファ識別子を `bufid` に返す. 呼び出し時にアクティブな送信バッファが存在しないときは 0 を返す

例

C: `bufid = pvm_getsbuf();`

Fortran: `CALL PVMFGETSBUF( BUFID )`

エラー

`pvm_getsbuf` は, エラー状態を返さない.

グループ名とインスタンス番号で識別されるプロセスの tid を返す.

形式

C `int tid = pvm_gettid( char *group, int inum )`

Fortran `call pvmfgettid( group, inum, tid )`

パラメータ

`group` – 存在するグループ名の文字列.

`inum` – グループにおけるインスタンス番号を表す整数.

`tid` – PVM プロセスのタスク識別子が返される.

説明

`pvm_gettid` ルーチンは, グループ名 `group` とインスタンス番号 `inum` を引数にとり,PVM プロセスのタスク識別子を返す. pvm_gettid は, 成功すれば `inum` は > 0 となり, エラーが起きた場合は, < 0 となる.

例

C: `inum = pvm_gettid( "worker", 0 );`

Fortran: `CALL PVMFGETTID( 'worker', 5, TID )`

エラー

`pvm_gettid` より返されるエラー状態を示す.

名前	原因
PvmSysErr	pvmd と接続できない (おそらくは pvmd が起動されていない).
PvmBadParam	引数の値が無効である (おそらくは文字列が NULL である).
PvmNoGroup	存在しないグループ名である.
PvmNotInst	インスタンスはグループに所属していない.

グループに所属するメンバー数を返す.

形式

C `int size = pvm_gsize( char *group )`

Fortran `call pvmfgsize( group, size )`

パラメータ

`group` – 存在するグループ名の文字列.

`size` – 現在グループに所属するメンバー数を整数で返す. 負値はエラーを表す.

説明

`pvm_gsize` ルーチンは, `group` で指定するグループの大きさを返す. エラーの場合は `size` が負となる.

PVM 3.0 では, グループは動的に変化するので, ルーチンは与えられたグループの瞬間の大きさを返すことのみを保証する.

例

C: `size = pvm_gsize( "worker", 0 );`

Fortran: `CALL PVMFGSIZE( 'group2', SIZE )`

エラー

`pvm_gsize` より返されるエラー状態を示す.

名前	原因
PvmSysErr	pvmd は起動していなか, またはクラッシュしている.
PvmBadParam	グループ名が無効である.

デフォルトの送信バッファをクリアし, メッセージのエンコードを指定する.

形式

C `int bufid = pvm_initsend( int encoding )`

Fortran `call pvmfinitsend( encoding, bufid )`

パラメータ

`encoding`– 次のメッセージのエンコード方法を指定するための整数を以下に示す.

C のオプション

`PvmDataDefault`	0	異機種結合なら XDR を使う
`PvmDataRaw`	1	エンコードしない
`PvmDataInPlace`	2	データは定位置にある

FORTRAN のオプション

`PVMDEFAULT`	0	異機種結合なら XDR を使う
`PVMRAW`	1	エンコードしない
`PVMINPLACE`	2	データ定位置にある

`bufid` – メッセージバッファ識別子を整数で返す. 負値はエラーを示す.

説明

`pvm_initsend` ルーチンは, 送信バッファをクリアし新しいメッセージをパックする準備をする. パックに用いるエンコードの方法は,`encoding` で指定する. デフォルトの設定では, バーチャルマシンの全てのコンピュータが同じデータフォーマットを使う時はエンコード無し, そうでなければ XDR エンコードが使われる. その他のオプションは, ユーザがバーチャルマシンに関して持っている知識を活用するエンコードである.

例えば, 次のメッセージをある固有の形式だけを解釈するマシンに送る場合には, ユーザは PvmDataRaw を呼び出すことで. エンコードのコストを節約できる.

PvmDataInPlace エンコードは, パック間データが定位置にあることを指定する. メッセージバッファは, 大きさと送信要素へのポインタのみを含む. pvm_send が呼び出されると, 各要素はユーザメモリから直接コピーされて送信される. ユーザがパックから送信までの間に要素の更新を行わないことが必要となるものの, 数多くのメッセージのコピーを低減する.

PVM 3.0 の初期の版では, デフォルトのオプションは XDR 変換を行うようになっていた. これは, メッセージを送信する前に異機種マシンを追加するかどうか, PVM は

知りようがなかったためである． PvmDataInPlace もまた初期の版では実装されていなかった.

将来の拡張 PVM では, 更なるオプションが使用可能になる予定である．ユーザやベンダーが最適なエンコードを持っており, ルーチンでの使用を希望するなら, それらのエンコードルーチンを新しいエンコードオプションとして PVM ソースコードに追加できる. 例えば,Cray Research には Cray 64 ビットフォーマットと IEEE 32 ビットフォーマットとの間での高速ベクトル化ルーチンがあり, オプションとして追加することができる.

pvm_initsend が成功すれば,bufid にはメッセージバッファ識別子が格納される． 何かエラーが発生した場合は,bufid は < 0 となる.

例

C:
```
bufid = pvm_initsend( PvmDataDefault );
info = pvm_pkint( array, 10, 1 );
msgtag = 3;
info = pvm_send( tid, msgtag );
```

Fortran:
```
CALL PVMFINITSEND( PVMDEFAULT, BUFID )
CALL PVMFPACK( REAL4, DATA, 100, 1, INFO )
CALL PVMFSEND( TID, 3, INFO )
```

エラー

pvm_initsend より返されるエラー状態を示す.

名前	原因
PvmBadParam	引数の値が無効である.
PvmNoMem	malloc に失敗した. バッファを生成するのに十分なメモリが無い.

呼び出しタスクを名前で与えられたグループに所属させる.

形式

C `int size = pvm_joingroup( char *group )`

Fortran `call pvmfjoingroup( group, info )`

パラメータ

`group` – 存在するグループ名の文字列.

`inum` – ルーチンが返すインスタンス番号. インスタンス番号は 0 から順に数え上げられる. 負値はエラーを表す.

説明

`pvm_joingroup` ルーチンは, 呼出タスクを `group` で指定する名前のグループに所属させ, グループでのインスタンス番号を inum で返す. エラーの場合は `inum` が負となる.

インスタンス番号は 0 から順に数え上げられる. グループを利用している時は, (group, inum) の組で, PVM プロセスを一意に識別できる. これは, 前述の PVM 名前付け手法に基づき一貫性が保証される. タスクが pvm_lvgroup を呼び出してグループを離脱した後で, 再び同じグループに所属した場合には, 同じインスタンス番号を得られる保証はない. PVM はインスタンス番号の再利用に努めるため, タスクがグループに所属する時はできる限り小さな番号を割り当てようとする. PVM 3.0 では, タスクは複数のグループに同時に所属することができる.

例

C: `inum = pvm_joingroup( "worker" );`

Fortran: `CALL PVMFJOINGROUP( 'group2', INUM )`

エラー

`pvm_joingroup` より返されるエラー状態を示す.

名前	原因
PvmSysErr	pvmd は起動していなか, またはクラッシュしている.
PvmBadParam	NULL のグループ名である.
PvmDupParam	既にグループに所属している.

指定する PVM プロセスを終了させる.

形式

C `int info = pvm_kill( int tid )`

Fortran `call pvmfkill( tid, info )`

パラメータ

`tid` – PVM プロセスのタスク識別子.

`info` – ルーチンの状態コードを返す. 負値はエラーを表す.

説明

`pvm_kill` ルーチンは, `tid` で識別される PVM プロセスにシグナル (SIGLKILL) を送る. マルチプロセッサの場合は, 終了シグナルはホストに依存したプロセス終了方法に置き換えられる. pvm_kill が成功すれば,`info` は 0 となる. 何かエラーが発生した場合は,`info` は < 0 となる.

`pvm_kill` ルーチンは, 呼び出し側を終了させるように設計されていない. 自分自身を終了させる場合,C では pvm_exit() と exit() を続けて呼び出す. FORTRAN では,pvmfexit と stop を実行する.

例

C: `inum = pvm_kill( tid );`

Fortran: `CALL PVMFKILL( TID, INFO )`

エラー

`pvm_kill` より返されるエラー状態を示す.

名前	原因
PvmBadParam	tid の値が無効である.
PvmSysErr	pvmd が応答しない.

呼び出しタスクを名前で与えられたグループから離脱させる.

形式

C `int size = pvm_lvgroup( char *group )`

Fortran `call pvmflvgroup( group, info )`

パラメータ

`group` – 存在するグループ名の文字列.

`inum` – ルーチンが返すインスタンス番号. インスタンス番号は 0 から順に数え上げられる. 負値はエラーを表す.

説明

`pvm_lvgroup` ルーチンは, 呼出タスクを `group` で指定する名前のグループから離脱させる. エラーの場合は `inum` が負となる.

タスクが pvm_lvgroup または pvm_exit を呼び出してグループを離脱した後で, 再び同じグループに所属した場合には, 新しいインスタンス番号を割り当てられることもあり得る. 元のインスタンス番号は pvm_joingroup の呼び出しによって任意のプロセスに再割り当てされる.

例

C: `inum = pvm_lvgroup( "worker" );`

Fortran: `CALL PVMFLVGROUP( 'group2', INFO )`

エラー

`pvm_lvgroup` より返されるエラー状態を示す.

名前	原因
PvmSysErr	pvmd が応答しない.
PvmBadParam	NULL のグループを与えた.
PvmNoGroup	存在しないグループ名である.
PvmNotInGroup	所属していないグループに対し離脱を要求した.

アクティブメッセージバッファのデータをタスク集合にマルチキャストする.

形式

C `int info = pvm_mcast( int *tids, int ntask, int msgtag )`

Fortran `call pvmfmcast( ntask, tids, msgtag, info )`

パラメータ

`ntask` – 送信相手のタスク数を指定する整数.

`tids` – 送信相手のタスクのタスク ID を内容とする, 少なくとも `ntask` の整数の配列.

`msgtag` – ユーザが整数で定義するメッセージ. msgtag >= 0 でなくてはならない. ユーザがプログラムにおいて, メッセージの種類の識別に用いる.

`info` – ルーチンの状態コードを返す. 負値はエラーを表す.

説明

pvm_mcast ルーチンは, アクティブメッセージバッファに保持されたメッセージを配列 `tids` で指定された `ntask` 個のタスクにマルチキャストする. 呼び出しタスクが tid の配列に含まれる場合, 自分自身にもメッセージが送られる. メッセージの内容は `msgtag` で識別される. pvm_mcast が成功すれば,`info` は 0 になる. 何かエラーが発生すると,`info` < 0 となる.

受信側プロセスは,pvm_recv または pvm_nrecv を呼び出すことによってマルチキャストのコピーを受信することができる. pvm_mcast は非同期かつ pvmd 間で最小被覆木アルゴリズムを構成するように送信される. メッセージが確実に受信側プロセッサへの通信経路に乗れば, 送信側プロセッサは直ちに計算に復帰する. これは同期通信と対照的である. 同期通信では, 受信側プロセッサが全て受信するまで, 送信側プロセッサは停止する.

pvm_mcast() は, まず最初に指定されたタスクを持つ pvmd を決定する. その後, これら pvmd に対し被覆木ブロードキャストを行い,pvmd は余分なトラフィック無しに順番にローカルのタスクへメッセージを配送する.

大抵のマルチプロセッサでは, マルチキャストはサポートされない. 普通は, 機種固有の関数として全プロセッサへのマルチキャストのみがサポートされる. こうした機能欠落があるマルチプロセッサでは,pvm_mcast は必ずしも効率の良い通信とはならない.

例

C:
```
info = pvm_initsend(PvmDataRaw);
info = pvm_pkint( array, 10, 1 );
msgtag = 5 ;
info = pvm_mcast( tids, ntask, msgtag );
```

Fortran:
```
CALL PVMFINITSEND( PVMDEFAULT )
CALL PVMFPKFLOAT( DATA, 100, 1, INFO )
CALL PVMFMCAST( NPROC, TIDS, 5, INFO )
```

エラー

pvm_mcast より返されるエラー状態を示す.

名前	原因
PvmBadParam	msgtag が負である.
PvmSysErr	pvmd が応答しない.
PvmNoBuf	送信バッファがない.

新規のメッセージバッファを生成する.

形式

C `int bufid = pvm_mkbuf( int encoding )`

Fortran `call pvmfmkbuf( encoding, bufid )`

パラメータ

`encoding`– バッファのエンコード方法を指定するための整数を以下に示す.

C のオプション		
`PvmDataDefault`	0	異機種結合なら XDR を使う
`PvmDataRaw`	1	エンコードしない
`PvmDataInPlace`	2	データは定位置にある

FORTRAN のオプションは短縮されている.		
`PVMDEFAULT`	0	異機種結合なら XDR を使う
`PVMRAW`	1	エンコードしない
`PVMINPLACE`	2	データ定位置にある

`bufid` – メッセージバッファ識別子を整数で返す. 負値はエラーを示す.

説明

`pvm_mkbuf` ルーチンは, 新規のメッセージバッファを生成しエンコード方法を `encoding` に設定する. pvm_mkbuf が成功すれば,bufid には新規のメッセージバッファ識別子が格納され, 送信バッファとして利用できる. 何かエラーが発生した場合は,`bufid` は < 0 となる.

デフォルトの設定では, バーチャルマシンの全てのコンピュータが同じデータフォーマットを使う時はエンコード無し, そうでなければ XDR エンコードが使われる. その他のオプションは, ユーザがバーチャルマシンに関して持っている知識を活用するエンコードである.

例えば, 次のメッセージをある固有の形式だけを解釈するマシンに送る場合には, ユーザは PvmDataRaw を呼び出すことで. エンコードのコストを節約できる.

PvmDataInPlace エンコードは, パック間データが定位置にあることを指定する. メッセージバッファは, 大きさと送信要素へのポインタのみを含む. pvm_send が呼び出されると, 各要素はユーザメモリから直接コピーされて送信される. ユーザがパックから送信までの間に要素の更新を行わないことが必要となるものの, 数多くのメッセージのコピーを低減する.

PVM 3.0 の初期の版では, デフォルトのオプションは XDR 変換を行うようになっていた. これは, メッセージを送信する前に異機種マシンを追加するかどうか, PVM は

知りようがなかったためである. PvmDataInPlace もまた初期の版では実装されていなかった.

将来の拡張 PVM では, 更なるオプションが使用可能になる予定である. ユーザやベンダーが最適なエンコードを持っており, ルーチンでの使用を希望するなら, それらのエンコードルーチンを新しいエンコードオプションとして PVM ソースコードに追加できる. 例えば,Cray Research には Cray 64 ビットフォーマットと IEEE 32 ビットフォーマットとの間での高速ベクトル化ルーチンがあり, オプションとして追加することができる.

多重バッファを扱いたい場合は,pvm_mkbuf と pvm_freebuf を組み合わせて使はなくてはならない. 送信後不要になったメッセージは,pvm_freebuf で解放すべきである.

受信バッファは pvm_recv 及び pvm_nrecv で自動的に生成され, 明示的に pvm_setrbuf で保存していないのであれば, 必ずしも解放する必要はない.

普通は, 多重送信バッファや受信バッファは必要ではなく, ユーザは pvm_initsend ルーチンを使ってデフォルトの送信バッファをリセットするのが簡単である.

多重バッファが有効な例としては,PVM を使ったライブラリやグラフィカルユーザインターフェースを利用しながらアプリケーション自身の通信には影響を与えたくない場合が考えられる.

多重バッファを利用する時, 一般的には各メッセージのパック毎にバッファの生成と解放を繰り返す.

例

C:
```
bufid = pvm_mkbuf( PvmDataDefault );
  /* ここでメッセージを送信する */
info = pvm_freebuf( bufid );
```

Fortran:
```
CALL PVMFMKBUF( PVMDEFAULT, BUFID )
  /* ここでメッセージを送信する */
CALL PVMFFREEBUF( BUFID, INFO )
```

エラー

pvm_mkbuf より返されるエラー状態を示す.

名前	原因
PvmBadParam	引数の値が無効である.
PvmNoMem	malloc に失敗した. バッファを生成するのに十分なメモリが無い.

バーチャルマシンのホストの状態を返す.

形式

C `int mstat = pvm_mstat( char *hosts )`

Fortran `call pvmfmstat( host, mstat )`

パラメータ

`host` – ホスト名の文字列.

`mstat` – 以下に示すマシンの状態を返す.

値	意味
PvmOk	ホストはOK
PvmNoHost	ホストはバーチャルマシンに無い
PvmHostFail	ホストに到達できない (たぶん失敗した)

説明

pvm_mstat ルーチンは, host で指定された名前のコンピュータの状態を mstat に返す. 特定のホストが障害を起こしているか, 及び再設定が必要か調べることができる.

例

C:
```
mstat = pvm_mstat( "msr.ornl.gov" );
```

Fortran:
```
CALL PVMFMSTAT( 'msr.ornl.gov', MSTAT );
```

エラー

pvm_mstat より返されるエラー状態を示す.

名前	原因
PvmSysErr	ローカル pvmd の応答がない.
PvmNoHost	ホストはバーチャルマシンに無い.
PvmHostFail	ホストに到達できない (たぶん失敗した).

グループ名とインスタンス番号で識別されるプロセスの tid を返す.

形式

C `int tid = pvm_mytid( void )`

Fortran `call pvmfmytid( tid )`

パラメータ

`tid` – 呼び出し PVM プロセスのタスク識別子が返される. 負値はエラーを示す.

説明

PVM 3.0 を利用するプログラムの最初の PVM 呼び出しは, 必ず `pvm_mytid` ルーチンでなくてはならない. このルーチンの最初の呼び出しによって, 呼び出しプロセスは PVM を利用可能となる. プロセスが pvm_spawn で生成されたのでなければ, このルーチンは一意な `tid` を生成する. pvm_mytid は, 呼び出しプロセスのタスク識別子を返し, アプリケーションから何度でも呼び出すことができる.

`tid` は, ローカル pvmd によって生成される 32 ビットの正数である. 32 ビットは, 更に幾つかのフィールドに分けられ, バーチャルマシンにおけるプロセスの位置 (ローカル pvmd のアドレス), マルチプロセッサの CPU 番号, 及びプロセス ID 等をエンコードする. これらの情報は PVM で利用さるものであり, アプリケーションでの利用は考慮されていない.

PVM が起動する前に, アプリケーションから pvm_mytid を呼び出した場合は, `tid` は < 0 となる.

例

C: `inum = pvm_mytid( );`

Fortran: `CALL PVMFMYTID( TID )`

エラー

`pvm_mytid` より返されるエラー状態を示す.

名前	原因
PvmSysErr	pvmd の応答がない.

イベントをタスクの集合に通知する.

形式

C `int info = pvm_notify( int what, int msgtag, int ntask, int *tids )`

Fortran `call pvmfnotify( what, msgtag, ntask, tids, info )`

パラメータ

`what` – 通知のトリガーとなったイベントを識別子する整数.

値	意味
PvmTaskExit	ローカル pvmd の応答がない.
PvmNoHost	ホストはバーチャルマシンに無い.
PvmHostFail	ホストに到達できない (たぶん失敗した).

`msgtag` – 通知に用いる整数メッセージタグ.

`ntask` – 配列 tids の大きさを指定する整数.

`tids` – 通知の対象となる pvmd またはタスクのリストを含む,ntask 長の整数配列.

`info` – ルーチンの状態コードを返す. 負値はエラーを表す.

説明

`pvm_notify` ルーチンは, 包括的な通知のための関数である. pvm_notify の実行後, 指定したイベントが発生したら, メッセージが生成され, tids 配列で指定した全てのタスクにマルチキャストされる. タスクは msgtag で指定されたメッセージを受信し適切な処理を行う責任がある. PVM の将来のバージョンでは, 通知可能なイベントが拡張され, ユーザ定義イベントも可能になる予定である.

例

C: `info = pvm_notify( PvmHostAdd, 9999, ntask, tids );`

Fortran: `CALL PVMFNOTIFY( PVMHOSTDELETE, 1111, NPROC, TIDS, INFO );`

エラー

`pvm_notify` より返されるエラー状態を示す.

名前	原因
PvmSysErr	pvmd が応答しない.
PvmBadParam	引数の値が無効である.

非ブロック受信.

形式

C `int bufid = pvm_nrecv( int tid, int msgtag )`

Fortran `call pvmfnrecv( tid, msgtag, bufid )`

パラメータ

`tid` – 送信側プロセスのタスク識別子として, ユーザが与える整数.

`msgtag` – ユーザが整数で定義するメッセージ. msgtag >= 0 でなくてはならない. ユーザがプログラムにおいて, メッセージの種類の識別に用いる.

`bufid` – 新規のアクティブな受信バッファの識別子を整数で返す. 負値はエラーを表す.

説明

pvm_nrecv ルーチンは,msgtag でラベル付けされたメッセージが tid から到着しているかどうかをチェックする. マッチするメッセージが到着すると, 直ちに pvm_nrecv は新規のアクティブ受信バッファにメッセージを置く. その時の受信バッファはクリアされ,bufid を返す.

要求メッセージが到着していない場合は,pvm_nrecv は bufid を 0 にして直ちに返る. 何かエラーが発生すると,bufid < 0 となる.

msgtag や tid の値が -1 であった場合は, 全てにマッチする. このことは, ユーザに対し以下のオプションを提供する. tid = -1 でありかつ msgtag をユーザが定義している場合は, pvm_nrecv は msgtag にマッチするメッセージを任意のプロセスから受理する. msgtag = -1 でありかつ tid をユーザが定義している場合は, pvm_nrecv は任意のメッセージを tid のプロセスから受理する.

pvm_nrecv は非同期である. ここで非ブロックとは, メッセージ自身か, あるいはローカル pvmd にメッセージが到着していないという情報とともに直ちに制御を返すことを意味する.

メッセージが到着したかどうかをチェックするために, 何度も pvm_nrecv を呼び出すことができる. 更に, アプリケーションがメッセージを待つ以外にすることがなくなった場合は, 同一メッセージに対して pvm_recv を実行してよい.

メッセージとともに pvm_nrecv が返ったら, ユーザはデータをユーザメモリ領域でアンパックできる.

例

C:

```
tid = pvm_parent();
msgtag = 4;
arrived = pvm_nrecv( tid, msgtag );
if( arrived )
  info = pvm_upkint( tid_array, 10, 1 );
else
  /* go do other computing */
```

Fortran:

```
CALL PVMFNRECV( -1, 4, ARRIVED )
IF ( ARRIVED .NE. 0 ) THEN
  CALL PVMFUNPACK( INTEGER4, TIDS, 25, 1, INFO )
  CALL PVMFUNPACK( REAL8, MATRIX, 100, 100, INFO )
ELSE
  GO DO USEFUL WORK
ENDIF
```

エラー

pvm_nrecv より返されるエラー状態を示す.

名前	原因
PvmBadParam	tid または msgtag の値が無効である.
PvmSysErr	pvmd が応答しない.

規定したデータ型の配列をアクティブなメッセージバッファにパックする.

形式

C

```
int info = pvm_pkbyte( char *xp, int nitem, int stride )
int info = pvm_pkcplx( float *cp, int nitem, int stride )
int info = pvm_pkdcplx( double *zp, int nitem, int stride )
int info = pvm_pkdouble(double *dp, int nitem, int stride )
int info = pvm_pkfloat( float *fp, int nitem, int stride )
int info = pvm_pkint( int *ip, int nitem, int stride )
int info = pvm_pklong( long *ip, int nitem, int stride )
int info = pvm_pkshort( short *jp, int nitem, int stride )
int info = pvm_pkstr( char *sp )
```

Fortran

```
call pvmfpack( what, xp, nitem, stride, info )
```

パラメータ

- `nitem` – パックするアイテムの総数 (バイト数ではない).
- `stride` –
- `xp` – バイトブロックの開始位置へのポインタ. いかなるデータ型のアンパックに対してもマッチする.
- `cp` – 少なくとも nitem*stride のアイテム数の複素数の配列.
- `zp` – 少なくとも nitem*stride のアイテム数の倍精度複素数の配列.
- `dp` – 少なくとも nitem*stride のアイテム数の倍精度実数の配列.
- `fp` – 少なくとも nitem*stride のアイテム数の単精度実数の配列.
- `ip` – 少なくとも nitem*stride のアイテム数の整数の配列.
- `jp` – 少なくとも nitem*stride のアイテム数の 2 バイト整数の配列.
- `sp` – NULL で終る文字配列へのポインタ.
- `what` – パックされるデータ型を指定する整数.

what オプション

STRING	0	REAL4	4
BYTE1	1	COMPLEX8	5
INTEGER2	2	REAL8	6
INTEGER4	3	COMPLEX16	7

- `info` – ルーチンの状態コードを返す. 負値はエラーを表す.

説明

各 pvm_pk* ルーチンは, 与えられたデータ型の配列をアクティブな送信バッファへパックする. 各ルーチンの引数は, パックされる一番目のアイテムへのポインタ, 配列からパックされるアイテムの総数 nitem, 及びパックするときの幅 stride からなる.

例外は pvm_pkstr() で,NULL で終る文字配列をパックするように定義されており, nitem 及び stride は不要である. FORTRAN ルーチン pvmfpack(STRING, ...) は,nitem が文字数を,stride が 1 であることを求めている.

パックが成功すれば,info は 0 となる. 何かエラーが発生した場合は, info は < 0 となる.

単一の変数 (配列でない) は,nitem $= 1$ 及び stride $= 1$ でパックできる. C 構造体は一度に一つのデータ型をパックしなくてはならない.

what オプションに対する将来の拡張としては, XDR エンコードが 64 ビット型に対して可能になり次第, サポートする予定である.

一方, ユーザは Cray のような 64 ビットマシンから SPARCstation のような 32 ビットマシンへデータを転送した場合には精度が失われることに注意すべきである. 覚え方としては,what 引数の名前には, 所望の精度のバイト数が含まれる. PVMRAW のエンコードを行えば, 異機種設定であっても 64 ビットマシンの間では, 64 ビット精度でデータが交換される.

データを正確に取り出すために, メッセージはパックした時と全く同様にアンパックしなくてはならない.

例

C:

```
info = pvm_initsend();
info = pvm_pkstr( "initial data" );
info = pvm_pkint( \&size, 1, 1 );
info = pvm_pkint( array, size, 1 );
info = pvm_pkdouble( matrix, size*size, 1 );
msgtag = 3;
info = pvm_send( tid, msgtag );
```

Fortran:

```
CALL PVMINITSEND( PVMRAW )
CALL PVMFPACK( INTEGER4, NSIZE, 1, 1, INFO )
CALL PVMFPACK( STRING, 'row 5 of NXN matrix' 19, 1, INFO )
CALL PVMFPACK( REAL8, MSGTAG, INFO )
CALL PVMFSEND( TID, MSGTAG, INFO )
```

エラー

名前	原因
PvmNoMem	malloc に失敗した. メッセージバッファの大きさがホストで利用可能なメモリ量を越えた.
PvmNoBuf	パックするアクティブな送信バッファがない. pvm_initsend を呼び出すこと.

呼び出しプロセスを生成したプロセスの tid を返す.

形式

C `int tid = pvm_parent( void )`

Fortran `call pvmfparent( tid )`

パラメータ

`tid` – 呼び出し PVM プロセスを生成したプロセスの tid を返す. 呼び出しプロセスが pvm_spawn で生成されたプロセスでなければ, tid = PvmNoParent となる.

説明

`pvm_parent` ルーチンは呼び出し PVM プロセスを生成したプロセスの tid を返す. 呼び出しプロセスが pvm_spawn で生成されたプロセスでなければ, tid = PvmNoParent となる.

SPMD(ホストのない) プログラムでは, 与えられたプログラムのインスタンスが自分自身をコピーするかどうかの決定に,pvm_parent() を利用することができる.

マスター / スレーブ方式では, スレーブがマスターの tid を決定するのに pvm_parent が利用される. そのため, スレーブは計算結果をマスターに返すことができる.

例

C: `tid = pvm_parent( );`

Fortran: `CALL PVMFPARENT( TID )`

エラー

`pvm_parent` より返されるエラー状態を示す.

名前	原因
PvmNoParent	呼び出しプロセスは,pvm_spawn で生成されたものではない.

最も最近の PVM 呼び出しのエラー状態を出力する.

形式

C `int info = pvm_perror( char *msg )`

Fortran `call pvmfperror( tid, info )`

パラメータ

`msg` – 最も最近の PVM 呼び出しのエラー状態のメッセージに追加される文字列として, ユーザが与える.

`info` – ルーチンの状態コードを返す. 負値はエラーを表す.

説明

pvm_perror ルーチンは, 最も最近の PVM 呼び出しのエラー状態を出力する. ユーザはエラーメッセージに追加する情報, 例えば位置等を,msg で与えることができる. 標準出力及び標準エラー出力は全て, マスター pvmd のあるホスト上のファイル /tmp/pvml.<*uid*> に置かれる.

例

```
C:       if( pvm_send( tid, msgtag ))
           pvm_perror();
Fortran: CALL PVMFSEND( TID, MSGTAG, INFO )
         IF(INFO .LT. 0) CALL CALL PVMFPERROR( 'STEP 6' INFO )
```

エラー

pvm_perror はいかなるエラーも返さない.

メッセージの到着をチェックする.

形式

C `int bufid = pvm_probe( int tid, int msgtag )`

Fortran `call pvmfprobe( tid, msgtag, bufid )`

パラメータ

- `tid` – 送信側プロセスのタスク識別子として, ユーザが与える整数. (-1 はワイルドカードとして, あらゆる tid とマッチする.)
- `msgtag` – ユーザが整数で定義するメッセージ. msgtag >= 0 でなくてはならない. ユーザがプログラムにおいて, メッセージの種類の識別に用いる. (-1 はワイルドカードとして, あらゆる msgtag とマッチする.)
- `bufid` – 新規のアクティブな受信バッファの識別子を整数で返す. 負値はエラーを表す.

説明

pvm_probe ルーチンは,msgtag でラベル付けされたメッセージが tid から到着しているかどうかをチェックする. マッチするメッセージが到着すると,pvm_probe は bufid にバッファ識別子を返す. bufid を pvm_bufinfo に与えて呼び出すことによって, メッセージの送信元や大きさ等の情報を得ることができる.

要求メッセージが到着していない場合は,pvm_probe は bufid に 0 を返す. 何かエラーが発生すると,bufid < 0 となる.

msgtag や tid の値が -1 であった場合は, 全てにマッチする. このことは, ユーザに対し以下のオプションを提供する. tid = -1 でありかつ msgtag をユーザが定義している場合は, pvm_probe は msgtag にマッチするメッセージを任意のプロセスから受理する. msgtag = -1 でありかつ tid をユーザが定義している場合は, pvm_probe は任意のメッセージを tid のプロセスから受理する.

メッセージが到着したかどうかをチェックするために, 何度も pvm_probe を呼び出すことができる. メッセージが到着したら, アンパックルーチンを使ってユーザメモリへデータをアンパックする前に, 必ず pvm_nrecv を呼び出さなくてはならない.

例

C:

```
tid = pvm_parent();
msgtag = 4;
arrived = pvm_probe( tid, msgtag );
if( arrived )
  info = pvm_bufinfo( arrived, &len, &tag, &tid );
else
  /* go do other computing */
```

Fortran:
```
CALL PVMFPROBE( -1, 4, ARRIVED )
IF ( ARRIVED .GT. 0 ) THEN
  CALL PVMBUFINFO( ARRIVED, LEN, TAG, TID, INFO )
ELSE
  GO DO USEFUL WORK
ENDIF
```

エラー

pvm_probe より返されるエラー状態を示す.

名前	原因
PvmBadParam	tid または msgtag の値が無効である.
PvmSysErr	pvmd が応答しない.

指定した PVM プロセスの状態を返す.

形式

C `int bufid = pvm_pstat( int tid, int msgtag )`

Fortran `call pvmfpstat( tid, msgtag, bufid )`

パラメータ

`tid` – 問い合わせる PVM プロセスのタスク識別子を表す整数.

`status` – tid によって指定した PVM プロセスの状態として返る整数. タスクの実行中は, 状態は PvmOk となり, そうでない場合は PvmNoTask となる. そして tid が間違っている場合は PvmBadParam となる.

説明

`pvm_pstat` ルーチンは,`tid` で指定したプロセスの状態を返す.

例

```
C:       tid = pvm_parent();
         status = pvm_pstat( tid );
Fortran: CALL PVMFPARENT( TID )
         CALL PVMFPSTAT( TID, STATUS )
```

エラー

`pvm_pstat` より返されるエラー状態を示す.

名前	原因
PvmBadParam	引数の値が無効である.
PvmSysErr	pvmd が応答しない.
PvmNoTask	タスクは実行中でない.

指定したメッセージタグのメッセージが指定した送信元から到着するまでブロックし, 到着したメッセージを新規の受信バッファに置く.

形式

C `int bufid = pvm_recv( int tid, int msgtag )`

Fortran `call pvmfrecv( tid, msgtag, bufid )`

パラメータ

- `tid` – 送信側プロセスのタスク識別子として, ユーザが与える整数. (-1 はワイルドカードとして, あらゆる tid とマッチする.)
- `msgtag` – ユーザが整数で定義するメッセージ. msgtag >= 0 でなくてはならない. ユーザがプログラムにおいて, メッセージの種類の識別に用いる. (-1 はワイルドカードとして, あらゆる msgtag とマッチする.)
- `bufid` – 新規のアクティブな受信バッファの識別子を整数で返す. 負値はエラーを表す.

説明

pvm_recv ルーチンは, `msgtag` でラベル付けされたメッセージが `tid` から到着するまでブロックする. マッチするメッセージが到着すると, 直ちに pvm_recv は新規のアクティブ受信バッファにメッセージを置く. その時の受信バッファはクリアされる.

`msgtag` や `tid` の値が -1 であった場合は, 全てにマッチする. このことは, ユーザに対し以下のオプションを提供する. tid = -1 でありかつ msgtag をユーザが定義している場合は, pvm_recv は msgtag にマッチするメッセージを任意のプロセスから受理する. msgtag = -1 でありかつ tid をユーザが定義している場合は, pvm_recv は任意のメッセージを tid のプロセスから受理する.

pvm_recv はブロックである. ここでブロックとは, 要求メッセージが到着していない場合は,pvm_recv は `bufid` を 0 にして直ちに返る. 何かエラーが発生すると,`bufid` < 0 となる.

pvm_recv が成功すれば,`bufid` に新規のアクティブな受信バッファの識別子を返す. 何かエラーが発生した場合は,`bufid` は < 0 となる.

pvm_recv はブロックである. ここでブロックとは, 指定した tid と msgtag にマッチするメッセージがローカル pvmd に到着するまで待つことを意味する.

メッセージとともに pvm_recv が返ったら, ユーザはデータをユーザメモリ領域にてアンパックできる.

例

C:
```
tid = pvm_parent();
msgtag = 4;
arrived = pvm_recv( tid, msgtag );
info = pvm_upkint( tid_array, 10, 1 );
info = pvm_upkint( problem_size, 1, 1 );
info = pvm_upkfloat( input_array, 100, 1 );
```

Fortran:
```
CALL PVMFRECV( -1, 4, ARRIVED )
CALL PVMFUNPACK( INTEGER4, TIDS, 25, 1, INFO )
CALL PVMFUNPACK( REAL8, MATRIX, 100, 100, INFO )
```

エラー

pvm_recv より返されるエラー状態を示す.

名前	原因
PvmBadParam	tid が無効である, または msgtag < -1 である.
PvmSysErr	pvmd が応答しない.

メッセージ受理時の比較関数を再定義する.

形式

C `int (*old) = pvm_recvf( int (*new)( int bufid, int tid, int tag ) )`

Fortran 利用できない

パラメータ

`tid` – 送信側プロセスのタスク識別子として, ユーザが与える整数.

`tag` – ユーザが整数で定義するメッセージタグ.

`bufid` – メッセージバッファの識別子.

説明

`pvm_recvf` は,pvm_recv 及び pvm_nrecv で用いられる比較関数を定義する. PVM メッセージパッシングをカスタマイズする手段を提供する. pvm_recvf は, 受信の際メッセージを評価するためにユーザが提供する比較関数を設定する. デフォルトの比較関数は, 入力される全てのメッセージに対して, 送信元とメッセージタグを比較する.

pvm_recvf の利用を強く奨める対象は, signal のような関数を良く理解しており, かつデフォルトよりも複雑なメッセージコンテキストを受信ルーチンに導入したいと考えている洗練された C プログラマーである.

デフォルトのマッチング関数の場合,pvm_recvf は 0 を返す. そうでない場合は, マッチング関数を返す. マッチング関数は以下の値を返さなくてはならない.

値	動作
< 0	このエラーコードを直ちに返す.
0	メッセージをピックしない.
1	このメッセージをピックし, 残りをスキャンしない.
> 1	全てのメッセージをスキャンし, 最も高位のランクのメッセージをピックする.

例 : recvf によるプローブの実装

C:

```
#include "pvm3.h"
static int foundit = 0;

static int
foo_match(mid, tid, code)
        int mid;
        int tid;
        int code;
{
        int t, c, cc;

        if ((cc = pvm\_bufinfo(mid, (int *)0, &c, &t)) < 0)
                return cc;
        if ((tid == -1 || tid == t)
        &&(code == -1 || code == c))
                foundit = 1;
        return 0;
}
int
probe(src, code)
{
        int (*omatch)();
        int cc;

        omatch = pvm_recvf(foo_match);
        foundit = 0;
        if ((cc = pvm_nrecv(src, code)) < 0)
                return cc;
        pvm_recvf(omatch);
        return foundit;
}
```

エラー

pvm_recvf が返すエラー状態はない.

アクティブメッセージバッファのデータを直ちに送信する.

形式

C `int info = pvm_send( int tid, int msgtag )`

Fortran `call pvmfsend( tid, msgtag, info )`

パラメータ

`tid` – 送信側プロセスのタスク識別子として, ユーザが与える整数.

`msgtag` – ユーザが整数で定義するメッセージ. msgtag >= 0 でなくてはならない. ユーザがプログラムにおいて, メッセージの種類の識別に用いる.

`info` – ルーチンの状態コードを返す. 負値はエラーを表す.

説明

pvm_send ルーチンは, アクティブメッセージバッファに保持されたメッセージを `tid` で指定するプロセスに送信する. メッセージの内容は `msgtag` で識別される. pvm_send が成功すれば,`info` は 0 になる. 何かエラーが発生すると,`info` < 0 となる.

pvm_sendsig は非同期である. メッセージが確実に受信側プロセッサへの通信経路に乗れば, 送信側プロセッサは直ちに計算に復帰する. これは同期通信と対照的である. 同期通信では, 受信側プロセッサが全て受信するまで, 送信側プロセッサは停止する.

例

```
C:       info = pvm_initsend(PvmDataDefault);
         info = pvm_pkint( array, 10, 1 );
         msgtag = 3 ;
         info = pvm_send( tid, msgtag );
Fortran: CALL PVMFINITSEND( PVMRAW )
         CALL PVMFPACK( REAL8, DATA, 100, 1, INFO )
         CALL PVMFSEND( TID, 3, INFO )
```

エラー

pvm_send より返されるエラー状態を示す.

名前	原因
PvmBadParam	tid または msgtag が無効である.
PvmSysErr	pvmd が応答しない.
PvmNoBuf	アクティブな送信バッファがない. 送信前に pvm_initsend を呼び出すこと.

pvmfsendsig() pvm_sendsig()

他の PVM プロセスにシグナルを送信する.

形式

C `int info = pvm_sendsig( int tid, int signum )`

Fortran `call pvmfsendsig( tid, signum, info )`

パラメータ

`tid` – 送信側プロセスのタスク識別子として, ユーザが与える整数.

`signum` – シグナル番号を表す整数.

`info` – ルーチンの状態コードを返す. 負値はエラーを表す.

説明

`pvm_sendsig` ルーチンは,`signum` 番号のシグナルを `tid` で指定するプロセスに送信する. pvm_sendsig が成功すれば,`info` は 0 になる. 何かエラーが発生すると, `info` < 0 となる.

pvm_sendsig がプログラマに提供するシグナル操作機能は, 実験的なものである. 割り込みを許す並列環境では, 非決定的動作, デッドロック, 及びシステムクラッシュは容易に引き起こされる. 例えば,UNIX カーネルを呼び出し中のプロセスに割り込みが発生すると, 正常な復帰は難しい.

例

```
C:       tid = pvm_parent();
         info = pvm_sendsig( tid, SIGKILL );
Fortran: CALL PVMFBUFINFO( BUFID, BYTES, TYPE, TID, INFO )
         CALL PVMFSENDSIG( TID, SIGNUM, INFO )
```

エラー

`pvm_sendsig` より返されるエラー状態を示す.

名前	原因
PvmSysErr	pvmd が応答しない.
PvmBadParam	tid または msgtag が無効である.

以降の PVM 呼び出しでの自動エラーメッセージ出力の有効 / 無効を設定する.

形式

C `int oldset = pvm_serror( int set )`

Fortran `call pvmfserror( set, oldset )`

パラメータ

`set` – 有効 (1) または無効 (0) の設定を定義する.

`oldset` – 変更前の pvm_serror の設定が定義される.

説明

`pvm_serror` ルーチンは, 呼び出しプロセスにおいて, これ以降の PVM 呼び出しでの自動エラーメッセージ出力の有効 / 無効を設定する. 全ての PVM ルーチンの, エラー状態を関連するエラーメッセージとともに自動的に出力する. `set` 引数に対し, メッセージ出力を有効にする場合は 1, 無効にする場合は 2 に設定して呼び出す. pvm_serror() は,`oldset` に元の設定値を返す.

標準出力及び標準エラー出力は全て, マスター pvmd のあるホスト上のファイル /tmp/pvml.<*uid*> に置かれる.

例

C: `info = pvm_serror( 1 );`

Fortran: `CALL PVMFSERROR( 0, INFO )`

エラー

`pvm_serror` より返されるエラー状態を示す.

名前	原因
PvmSysErr	pvmd が応答しない.
PvmBadParam	tid または msgtag が無効である.

アクティブなバッファを切替え, 以前のアクティブな受信バッファを保存する.

形式

C `int bufid = pvm_setrbuf( int bufid )`

Fortran `call pvmfsetrbuf( bufid, oldbuf )`

パラメータ

`bufid` – アクティブな受信バッファのメッセージバッファ ID として返される整数.

説明

`pvm_setrbuf` ルーチンは, アクティブなバッファの ID を `bufid` 切替え, 以前のアクティブな受信バッファを `oldbuf` に返す. `bufid` を 0 に設定すると, 現在のアクティブなバッファは保存された後, アクティブな受信バッファが存在しなくなる.

受信が成功すると, 自動的に新規のアクティブな受信バッファが生成される. それ以前の受信がアンパックされていなくて, 今後の処理のために保存が必要な場合は, そのバッファを保存した後リセットし, アクティブバッファのアンパックに備えることができる.

本ルーチンは, 多重バッファを扱う際に必要となる. 例としては,2 つのバッファを切替える方法がある. 一方のバッファをグラフィカルユーザインターフェース情報の送信に利用し, もう一方のバッファをアプリケーションでのデータの送信に利用する

例

C: `rbuf1 = pvm_setrbuf( rbuf2 );`

Fortran: `CALL PVMFSETRBUF( NEWBUF, OLDBUF )`

エラー

`pvm_setrbuf` より返されるエラー状態を示す.

名前	原因
PvmBadParam	引数の値が無効である.
PvmNoSucbBuf	与えられた bufid が無効である.

アクティブな送信バッファを切替える.

形式

C `int bufid = pvm_setsbuf( int bufid )`

Fortran `call pvmfsetsbuf( bufid, oldbuf )`

パラメータ

`bufid` – アクティブな送信バッファのメッセージバッファ ID として返される整数.

説明

`pvm_setsbuf` ルーチンは, アクティブな送信バッファの ID を `bufid` 切替え, 以前のアクティブな受信バッファを `oldbuf` に返す. `bufid` を 0 に設定すると, 現在のアクティブなバッファは保存された後, アクティブな受信バッファが存在しなくなる.

本ルーチンは, 多重バッファを扱う際に必要となる. 例としては,2 つのバッファを切替える方法がある. 一方のバッファをグラフィカルユーザインターフェース情報の送信に利用し, もう一方のバッファをアプリケーションでのデータの送信に利用する

例

C: `rbuf1 = pvm_setsbuf( rbuf2 );`

Fortran: `CALL PVMFSETSBUF( NEWBUF, OLDBUF )`

エラー

`pvm_setsbuf` より返されるエラー状態を示す.

名前	原因
PvmBadParam	引数の値が無効である.
PvmNoSucbBuf	与えられた bufid が無効である.

新規の PVM プロセスを起動する.

形式

C `int bufid = pvm_spawn( char *task, char **argv`
`int flag, char *where,`
`int ntask, char *tids )`

Fortran `call pvmfspawn( task, flag, where,`
`ntask, tids, numt )`

パラメータ

`task` – 起動する PVM プロセスの実行形式ファイル名の文字列. 起動時には, 各ホストに実行形式が存在していなくてはならない. PVM が参照するデフォルトの位置は, `HOME/pvm3/bin/architecture_name/filename` である.

`argv` – 実行形式への引数の配列であり,NULL で終りを指定する. 実行形式が引数を取らない場合は,pvm_spawn の第 2 引数は NULL である.

`flag` – spawn のオプションを指定する整数を以下に示す.

C での flag は以下の通り.

オプション		意味
PvmTaskDefault	0	PVM がプロセスの生成場所を決定する.
PvmTaskHost	1	`where` でホストを指定する.
PvmTaskArch	2	`where` でアーキテクチャを指定する.
PvmTaskDebug	4	デバッガーの制御下でプロセスを起動する.
PvmTaskTrace	8	プロセスからの PVM 呼び出しのトレースデータを出力する.

FORTRAN での flag は以下の通り.

オプション		意味
PVMDEFAULT	0	PVM がプロセスの生成場所を決定する.
PVMHOST	1	`where` でホストを指定する.
PVMARCH	2	`where` でアーキテクチャを指定する.
PVMDEBUG	4	デバッガーの制御下でプロセスを起動する.
PVMTRACE	8	プロセスからの PVM 呼び出しのトレースデータを出力する.

`where` – どこで PVM プロセスを起動するか指定する. `where` が, “ibm1.epm.ornl.gov” のようなホスト名であるか,“SUN4” のようなアーキテクチャであるかは,flag に依存する. `flag` が 0 ならば,`where` は無視され, PVM は適切なホストを選択する.

`ntask` – 起動する実行形式のコピーの数を指定する整数.

tids – 少なくとも ntask の大きさの整数配列．pvm_spawn によって起動されたプロセスの tid を配列に格納して返る．各タスクにおいて起動時にエラーが発生した場合は, 配列の対応する位置にエラーコードが格納される．

numt – 実際に起動するタスク数を表す整数．負値はシステムのエラーを示す．ntask 未満の正数は, 一部にエラーが起こったことを示す．この場合, ユーザはエラーコードのために tids 配列をチェックする必要がある．

説明

pvm_spawn は,task で名前を指定された実行形式のコピーを ntask 個起動する．

PVM プロセスを起動するホストは,flag と where で指定される．起動した各プロセスのタスク識別子は tids に格納される．

pvm_spawn が一つ以上のタスクを起動する場合, numt は実際に起動したタスク数となる． システムにエラーが起こった場合は,numt < 0 となる． numt が ntask 未満の場合は, 幾つかの起動に失敗したことを示しており, ユーザはエラーコードのために tids 配列の ntask - numt 個の位置をチェックする必要がある．

flag が 0 で where が NULL(FORTRAN では *) の場合は，バーチャルマシン全体に ntask 個のプロセスをヒューリスティックに分散する． 最初にヒューリスティックは, マシン負荷と性能の測定量を用いて, 最も適切なホストを決定する．

特殊な場合として, マルチプロセッサで where を指定した場合は，pvm_spawn はベンダー提供のルーチンを利用して ntask 個のコピーを一つのマシンで起動する．

PvmTaskDebug が設定されている場合は,pvmd はデバッガーの制御下でタスクを起動する． この場合,pvm3/bin/ARCH/task args の代わりに，pvm3/lib/debugger pvm3/bin/ARCH/task args を実行する． デバッガーはシェルスクリプトから起動するので, ユーザは好みのデバッガーに変更することができる． 現在のスクリプトは,xterm を dbx(または互換デバッガー) とともに起動する．

例

C:
```
numt = pvm_spawn( "host", 0, PvmTaskHost, "sparky", 1, &tid[0]);
numt = pvm_spawn( "host", 0, PvmTaskHost+PvmTaskDebug,
                  "sparky", 1, &tid[0]);
                  "host", 0, PvmTaskHost, "sparky", 1, &tid[0]);
numt = pvm_spawn( "node", 0, PvmTaskArch, "RIOS", 1, &tid[i]);
numt = pvm_spawn( "FEM1", args, 0, 0, 16, tids);
numt = pvm_spawn( "pde", 0, PvmTaskHost, "paragon.ornl", 512, &tid[0]);
```

Fortran:
```
FLAG = PVMARCH + PVMDEBUG
CALL PVMFSPAWN( 'node', FLAG, 'SUN4', 1, TID(3), NUMT )
CALL PVMFSPAWN( 'FEM1', PVMDEFAULT, '*', 16, TIDS, NUMT )
CALL PVMFSPAWN( 'TBMD', PVMHOST, 'cm5.utk.edu', 32, TIDS, NUMT )
```

エラー

pvm_spawn より返されるエラー状態を示す.

名前	原因
PvmBadParam	引数の値が無効である.
PvmNoHost	ホストはバーチャルマシンに無い.
PvmNoFile	指定した実行形式のファイルが無い. デフォルトの参照位置は, /pvm3/bin/ARCH. ここで ARCH は PVM のアーキテクチャ名である.
PvmNoMem	malloc に失敗した. ホストに十分なメモリが無い.
PvmSysErr	pvmd が応答しない.
PvmOutOfRes	PVM のシステム資源が不足.

バーチャルマシンで実行中のタスクの情報を返す.

形式

C
```
int info = pvm_tasks( int where, int *ntask,
                      struct taskinfo **taskp)
```

```
struct taskinfo{
    int ti_tid;
    int ti_ptid;
    int ti_host;
    int ti_flag;
    char *ti_a.out;
} taskp;
```

Fortran
```
call pvmftasks( where, ntask, info )
```

パラメータ

`where` – 情報を要求するタスクを整数で指定する.

0	バーチャルマシン上で全てのタスク.
pvmd tid	与えられたホストの全てのタスク.
tid	特定のタスク.

`ntask` – 起動する実行形式のコピーの数を指定する整数.

`taskp` – 各タスクの情報を含む構造体の配列へのポインタ. この中には, タスク ID, 親タスクの ID,pvmd のタスク ID, 状態フラグ及びタスクの実行形式ファイル名がある. 状態フラグには, メッセージ待ち,pvmd 待ち, 及び実行中である.

`info` – ルーチンの状態コードを返す. 負値はエラーを表す.

説明

ルーチン pvm_tasks は, 現在バーチャルマシンで実行中のタスクの情報を返す. 返される情報は, コンソール上で ps コマンドを実行したときと同じである.

pvm_tasks が成功すれば,`info` は 0 になる. 何かエラーが発生すると, `info` < 0 となる.

例

C:
```
numt = pvm_tasks( 0, &ntask, &taskp);
```

Fortran:
```
CALL PVMFTASKS( DTID, NTASK, INFO )
```

エラー

pvm_tasks より返されるエラー状態を以下に示す.

名前	原因
PvmBadParam	引数の値が無効である.
PvmSysErr	pvmd が応答しない.
PvmNoHost	指定したホストがバーチャルマシンに無い.

アクティブなメッセージバッファを規定したデータ型の配列にアンパックする.

形式

C

```
int info = pvm_upkbyte( char *xp, int nitem, int stride )
int info = pvm_upkcplx( float *cp, int nitem, int stride )
int info = pvm_upkdcplx( double *zp, int nitem, int stride )
int info = pvm_upkdouble(double *dp, int nitem, int stride )
int info = pvm_upkfloat( float *fp, int nitem, int stride )
int info = pvm_upkint( int *ip, int nitem, int stride )
int info = pvm_upklong( long *ip, int nitem, int stride )
int info = pvm_upkshort( short *jp, int nitem, int stride )
int info = pvm_upkstr( char *sp )
```

Fortran

```
call pvmfunpack( what, xp, nitem, stride, info )
```

パラメータ

`nitem` – アンパックするアイテムの総数 (バイト数ではない).

`stride` –

`xp` – バイトブロックの開始位置へのポインタ. いかなるデータ型のアンアンパックに対してもマッチする.

`cp` – 少なくとも nitem*stride のアイテム数の複素数の配列.

`zp` – 少なくとも nitem*stride のアイテム数の倍精度複素数の配列.

`dp` – 少なくとも nitem*stride のアイテム数の倍精度実数の配列.

`fp` – 少なくとも nitem*stride のアイテム数の単精度実数の配列.

`ip` – 少なくとも nitem*stride のアイテム数の整数の配列.

`jp` – 少なくとも nitem*stride のアイテム数の 2 バイト整数の配列.

`sp` – NULL で終る文字配列へのポインタ

`what` – アンパックされるデータ型を指定する整数.

what オプション

STRING	0	REAL4	4
BYTE1	1	COMPLEX8	5
INTEGER2	2	REAL8	6
INTEGER4	3	COMPLEX16	7

`info` – ルーチンの状態コードを返す. 負値はエラーを表す.

説明

各 pvm_upk* ルーチンは, アクティブな受信バッファより与えられたデータ型の配列をアンパックする. 各ルーチンの引数は, アンパックされる一番目のアイテムへのポインタ, 配列からアンパックされるアイテムの総数 nitem, 及びアンパックするときの幅 stride からなる.

例外は pvm_upkstr() で,NULL で終る文字配列をアンパックするように定義されており, nitem 及び stride は不要である. FORTRAN ルーチン pvmfunpack(STRING, ...) は,nitem が文字数を,stride が 1 であることを求めている.

アンパックが成功すれば,info は 0 となる. 何かエラーが発生した場合は, info は < 0 となる.

単一の変数 (配列でない) は,nitem $= 1$ 及び stride $= 1$ でアンパックできる. C 構造体は一度に一つのデータ型をアンパックしなくてはならない.

what オプションに対する将来の拡張としては, XDR エンコードが 64 ビット型に対して可能になり次第, サポートする予定である.

一方, ユーザは Cray のような 64 ビットマシンから SPARCstation のような 32 ビットマシンへデータを転送した場合には精度が失われることに注意すべきである. 覚え方としては,what 引数の名前には, 所望の精度のバイト数が含まれる. PVMRAW のエンコードを行えば, 異機種設定であっても 64 ビットマシンの間では, 64 ビット精度でデータが交換される.

データを正確に取り出すために, メッセージはパックした時と全く同様にアンアンパックしなくてはならない.

例

C:
```
info = pvm_upkstr( string );
info = pvm_upkint( &size, 1, 1 );
info = pvm_upkint( array, size, 1 );
info = pvm_upkdouble( matrix, size*size, 1 );
```

Fortran:
```
CALL PVMFRECV( TID, MSGTAG )
CALL PVMFUNPACK( INTEGER4, NSIZE, 1, 1, INFO )
CALL PVMFUNPACK( STRING, STEPNAME, 8, 1, INFO )
CALL PVMFUNPACK( REAL4, A(5,1), NSIZE, NSIZE, INFO )
```

エラー

名前	原因
PvmNoMem	malloc に失敗した. メッセージバッファの大きさがホストで利用可能なメモリ量を越えた.
PvmNoBuf	アンパックするアクティブな受信バッファがない. pvm_initsend を呼び出すこと.