Parallel Virtual Machine

- Co to jest?
- Konfigurowanie maszyny wirtualnej
- Tworzenie procesów
- Buforowanie danych
- Przesyłanie danych
- Operacje na grupach procesów
- Komunikacja kolektywna
- Przykład

Pakiet PVM

- PVM – *Parallel Virtual Machine* – środowisko oprogramowania równoległego:
  - na maszynach wieloprocesorowych SM,
  - w heterogenicznych sieciach maszyn sekwencyjnych i równoległych.
- Opracowane oryginalnie w Oak Ridge National Laboratory (1989):

  http://www.csm.ornl.gov/pvm/pvm_home.html

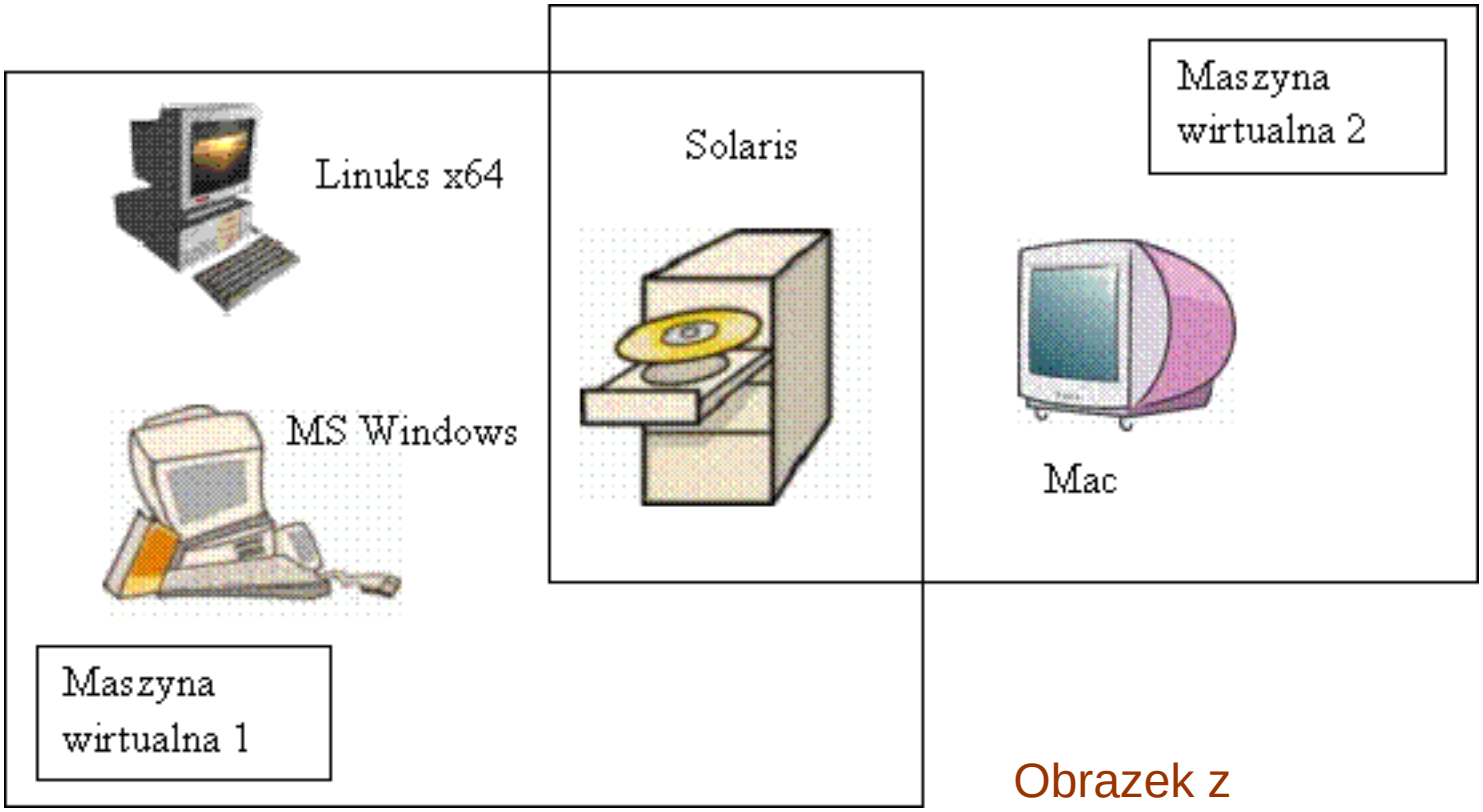
Założenia

- Przenośność – unikaj cech specyficznych dla okreslonego systemu, które ciężko będzie uzyskać gdzie indziej.
  „as simple as possible but can be optimized”
- Komunikacja przy pomocy gniazdek (*socket*)
- Protokoły TCP i UDP
- Maszyny wieloprocesorowe bez gniazdek wyposażyć we front-end, który je zastąpi

Równoległa maszyna wirtualna

- Abstrakcja izolująca programistę od heterogenicznej natury sprzętowej węzłów współuczestniczących w obliczeniach.
- Z punktu widzenia programisty dostępna za pomocą biblioteki procedur do przesyłania komunikatów:
  - interfejs do języków C i Fortran,
  - dostępne „wrappery” dla C++, F90, Perl, Python,
  - aktualna wersja 3.4.6.

Maszyna wirtualna



Obrazek z
http:://wazniak.mimuw.edu.pl

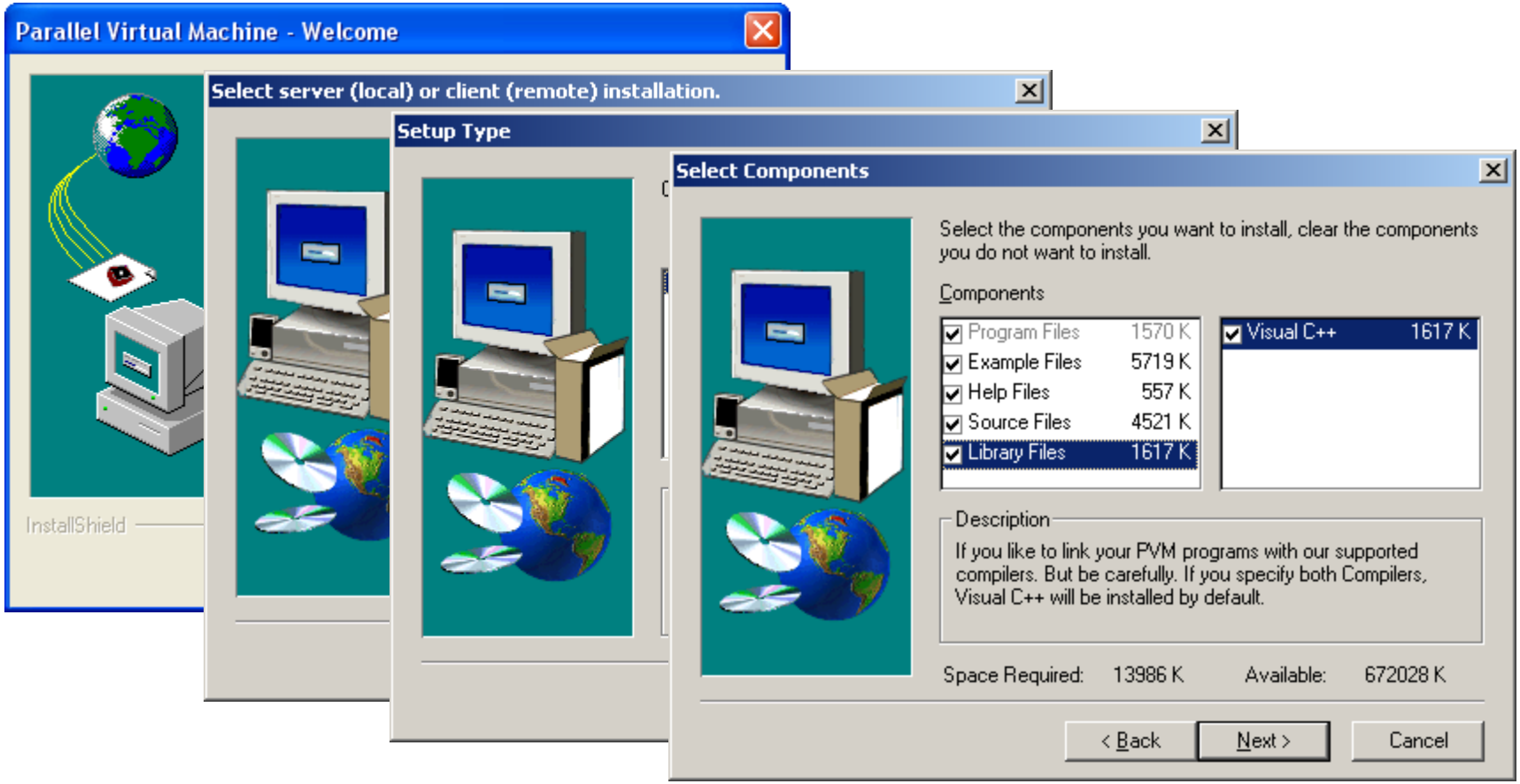
Elementy środowiska PVM

- Program konsoli zarządzania maszyną wirtualną – **`pvm`**
- Program demon, umożliwiający współdziałanie procesów (programów) aplikacji rozproszonej – **`pvmd3`**
- Serwer grup, zapewniający identyfikację grup procesów – **`pvmgs`**
- Biblioteka funkcji podstawowych – **`libpvm3.a`**
- Biblioteka funkcji operujących na grupach procesów – **`libgpvm3.a`**
- Biblioteka interfejsu Fortranu – **`libfpvm3.a`**

Uruchomienie PVM

- Instalacja pakietu PVM na wszystkich węzłach (architekturach) maszyny wirtualnej.
- Zdefiniowanie zmiennych środowiskowych:
  - PVM_ARCH – określającą architekturę sprzętową danego węzła np. LINUX,
  - PVM_ROOT – wskazującą katalog, gdzie pakiet został zainstalowany.
- W katalogu $PVM_ROOT/lib/$PVM_ARCH znajdą się biblioteki, a w $PVM_ROOT/bin/$PVM_ARCH programy.

Instalacja


Parallel Virtual Machine - Welcome
InstallShield
Select server (local) or client (remote) installation.
Setup Type
Select Components
Select the components you want to install, clear the components you do not want to install.
Components
Program Files 1570 K
Example Files 5719 K
Help Files 557 K
Source Files 4521 K
Library Files 1617 K
Visual C++ 1617 K
Description
If you like to link your PVM programs with our supported compilers. But be carefully. If you specify both Compilers, Visual C++ will be installed by default.
Space Required: 13986 K
Available: 672028 K
< Back
Next >
Cancel

```
jstar@edi:/$ sudo apt-get install pvm
Password:
Czytanie list pakietów... Gotowe
Budowanie drzewa zależności... Gotowe
Zostaną zainstalowane następujące dodatkowe pakiety:
  libpvm3
Zostaną zainstalowane następujące NOWE pakiety:
  libpvm3 pvm
0 aktualizowanych, 2 nowo instalowanych, 0 usuwanych i 65 nieaktualizowanych.
Konieczne pobranie 334kB archiwów.
Po rozpakowaniu zostanie dodatkowo użyte 831kB miejsca na dysku.
Czy chcesz kontynuować [T/n]? t
Pob: 1 ftp://ftp.icm.edu.pl etch/main libpvm3 3.4.5-7 [100kB]
Pob: 2 ftp://ftp.icm.edu.pl etch/main pvm 3.4.5-7 [234kB]
Pobrano 334kB w 0s (1079kB/s)
Zaznaczenie poprzednio niezaznaczonego pakietu libpvm3.
(Odczytywanie bazy danych ... 86034 plików i katalogów obecnie zainstalowanych.)
Rozpakowanie libpvm3 (z .../libpvm3_3.4.5-7_amd64.deb) ...
Zaznaczenie poprzednio niezaznaczonego pakietu pvm.
Rozpakowanie pvm (z .../archives/pvm_3.4.5-7_amd64.deb) ...
Konfigurowanie libpvm3 (3.4.5-7) ...

Konfigurowanie pvm (3.4.5-7) ...

jstar@edi:/$
```

Działanie maszyny wirtualnej

- Uruchomienie maszyny wirtualnej (osobnej) dla każdego użytkownika następuje po wystartowaniu procesu demona zarządzającego na pierwszym komputerze:
  - przydziela on identyfikatory procesów, zarządza grupami procesów, itp.
- Demon ten za pomocą polecenia **`rsh`** startuje demony na pozostałych komputerach konfiguracji, rozszerzając maszynę wirtualną.

Konsola PVM

- Program obsługi konsoli maszyny wirtualnej:

  `pvm [hostfile]`

  gdzie hostfile jest opcjonalną nazwą pliku konfiguracyjnego, definiującego początkowy skład maszyny wirtualnej.
- Po uruchomieniu brakujących demonów konsola zgłasza oczekiwanie na polecenia ze standardowego wejścia:

  `pvm>`

```
jstar@gaus:~$ pvm
pvm> help
help
help       Print helpful information about a command
Syntax:  help [ command ]
Commands are:
  add        Add hosts to virtual machine
  alias      Define/list command aliases
  conf       List virtual machine configuration
  delete     Delete hosts from virtual machine
  echo       Echo arguments
  export     Add environment variables to spawn export list
  getopt     Display PVM options for the console task
  halt       Stop pvmds
  help       Print helpful information about a command
  id         Print console task id
  jobs       Display list of running jobs
  kill       Terminate tasks
  mstat      Show status of hosts
  names      List message mailbox names
  ps         List tasks
  pstat      Show status of tasks
  put        Add entry to message mailbox
  quit       Exit console
  reset      Kill all tasks, delete leftover mboxes
  setenv     Display or set environment variables
  setopt     Set PVM options - for the console task *only*!
  sig        Send signal to task
  spawn      Spawn task
  trace      Set/display trace event mask
  unalias    Undefine command alias
  unexport   Remove environment variables from spawn export list
  version    Show libpvm version
pvm>
```

Komendy konsoli

- Najczęściej stosowane komendy konsoli:
  - add hostname [hostname] ... – dodaje węzły;
  - delete hostname [hostname] ... – usuwa węzły z maszyny, procesy są zabijane;
  - conf – wyświetla bieżącą konfigurację;
  - halt – przerywa wszystkie procesy, zamyka demony PVM (maszynę wirtualną) i kończy działanie konsoli;
  - quit – kończy działanie tylko konsoli;
  - spawn – uruchamia nowy proces użytkownika na wybranym węźle maszyny wirtualnej;
  - help [command] – wyświetla opis (danej komendy).

```
pvm> add edi
add edi
Password:
1 successful
            HOST     DTID
             edi    c0000
pvm>
```

Plik konfiguracyjny

- Każda linia opisuje jeden komputer:
  - zawiera nazwę węzła i dodatkowe opcje w postaci **`nazwa=wartość`** (oddzielone przecinkami) np.
  - lo=*username* – lokalna nazwa użytkownika,
  - so=*password* – lokalne hasło użytkownika,
  - ep=*dirs* – lista katalogów do przeszukiwania przy starcie programów użytkownika (domyślnie na danym węźle $HOME/pvm3/bin/$PVM_ARCH),
  - sp=*number* – określa względną szybkość węzła.

Tworzenie procesów

- Procesy użytkownika są uruchamiane z katalogu $HOME/pvm3/bin/$PVM_ARCH lub określonego przez opcję ‘ep’.
- Demon tworzy nowy proces na skutek komendy konsoli spawn lub funkcji pvm_spawn.
- Nazwy funkcji biblioteki PVM mają przedrostek pvm_ (w języku C) lub pvmf (Fortran).
- Aby korzystać z biblioteki PVM należy dołączyć plik nagłówkowy pvm3.h (C) lub fpvm.h (Fortran).
- Każdy proces w maszynie jest identyfikowany za pomocą unikalnego numeru tid.

Tworzenie procesów (2)

- Funkcja pvm_mytid:

```
C: int tid = pvm_mytid(void)
Fortran: call pvmfmytid(tid)
```

  - zwraca tid wywołującego ją procesu.

- Funkcja pvm_exit:

```
C: int info = pvm_exit(void)
Fortran: call pvmfexit(info)
```

  - kończy współpracę procesu z PVM.

- Funkcja pvm_spawn:
  - tworzy nowy proces: task – nazwa pliku z kodem, argv – wektor argumentów, flag – flagi określające opcje, where – określa nazwę węzła, ntask – liczba kopii procesu, tids – tablica zwracająca identyfikatory procesów lub kody błędów,
  - funkcja zwraca liczbę uruchomionych procesów.

pvm_spawn

```
C: int numt = pvm_spawn(char *task, char **argv,
                        int flag,   char *where,
                        int ntask,  int *tids)
Fortran: call pvmfspawn(task,  flag, where,
                        ntask, tids, numt)
```

- Flagi (flag):

Wartość:	Nazwa:	Znaczenie:
0	**PvmTaskDefault**	PVM wybiera gdzie uruchomić proces.
1	**PvmTaskHost**	Argument where określa gdzie uruchomić proces.
2	**PvmTaskArch**	Argument where określa architekturę PVM_ARCH.
4	**PvmTaskDebug**	Uruchamia proces pod debuggerem.
8	**PvmTaskTrace**	Generuje dane śledzenia (*trace*) procesu.
16	**PvmMppFront**	Uruchamia zadania na węźle *MPP front-end*.
32	**PvmHostCompl**	Argument where uzupełnia zbiór węzłów.

Kontrola procesów

- Funkcja pvm_parent: `int tid = pvm_parent(void)`
  - zwraca identyfikator procesu macierzystego lub gdy on nie istnieje wartość **PvmNoParent**.

- Funkcja pvm_kill: `int info = pvm_kill(int tid)`
  - pozwala zakończyć określony proces (nie bieżący),
  - funkcja wysyła sygnał SIGTERM do procesu,
  - zwraca kod błędu lub 0.

Przesyłanie komunikatów

- Przesłanie komunikatu wymaga:
  - utworzenia bufora nadawczego,
  - wpisania danych do tego bufora,
  - wysłania treści bufora (nadając mu identyfikator) do jednego lub wielu procesów,
  - zwolnienia (ewentualnego) bufora nadawczego,
  - utworzenia bufora odbiorczego,
  - odebrania komunikatu do tego bufora,
  - odczytania treści (rozpakowania) komunikatu,
  - zwolnienia (ewentualnego) bufora odbiorczego.

Przesyłanie komunikatów (2)

- W procesie może istnieć wiele buforów, ale w danej chwili tylko jeden jest aktywnym buforem nadawczym i jeden aktywnym buforem odbiorczym.
- Funkcje komunikacyjne operują na aktywnym buforze.

Tworzenie bufora

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

- Funkcja pvm_mkbuf:
  - zwraca identyfikator utworzonego bufora,
  - argument encoding określa sposób kodowania danych umieszczanych w buforze:

	Nazwa:	Wartość:	Znaczenie:
C:	**PvmDataDefault**	0	kodowanie XDR
	PvmDataRaw	1	brak kodowania
	PvmDataInPlace	2	wskaźniki do danych
Fortran:	**PVMDEFAULT**	0	kodowanie XDR
	PVMRAW	1	brak kodowania
	PVMINPLACE	2	wskaźniki do danych

Zwolnienie bufora

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

- Funkcja pvm_freebuf:
  - zwalnia bufor bufid.

Uaktywnienie bufora

- Utworzony bufor należy uaktywnić.

```
int oldbuf = pvm_setsbuf(int bufid)
```

- Funkcja pvm_setsbuf:
  - ustawia bufor bufid jako aktywny bufor nadawczy.

```
int oldbuf = pvm_setrbuf(int bufid)
```

- Funkcja pvm_setrbuf:
  - ustawia bufor bufid jako aktywny bufor odbiorczy.
- Obie funkcje zwracają identyfikator poprzedniego aktywnego bufora nadawczego/odbiorczego.

Uzyskanie identyfikatora bufora

- Identyfikator aktywnego bufora (lub 0 gdy nie ma bufora) zwracają funkcje odpowiednio: pvm_getsbuf i pvm_getrbuf.

```
int bufid = pvm_getsbuf(void)
int bufid = pvm_getrbuf(void)
```

- W chwili utworzenia procesu są tworzone po jednym aktywnym buforze nadawczym i odbiorczym (nie trzeba ich zwalniać).

Usuwanie zawartości bufora

- Przed umieszczeniem w buforze nadawczym nowego komunikatu należy usunąć jego poprzednią zawartość:
  - inaczej nowe dane zostaną dołączone do poprzednich.
- Funkcja pvm_initsend:
  - dodatkowo umożliwia zmianę sposobu kodowania danych.

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

- Dane z bufora odbiorczego są usuwane automatycznie.

Dodawanie danych do bufora

- Każdy bufor może przechowywać wiele porcji danych.
  - Każda porcja jest tablicą jednego typu.
- Do dodawania porcji do bufora służą funkcje:
  - pvm_packf, pvm_pkstr,
  - pvm_pkbyte, pvm_pkcplx, pvm_pkdcplx,
  - pvm_pkdouble, pvm_pkfloat, pvm_pkint, pvm_pkuint,
  - pvm_pkshort, pvm_pkushort, pvm_pklong, pvm_pkulong.

Funkcje pakujące (C)

```
int info = pvm_pkbyte(          char *data, int nitem, int stride)
int info = pvm_pkcplx(         float *data, int nitem, int stride)
int info = pvm_pkdcplx(       double *data, int nitem, int stride)
int info = pvm_pkdouble(      double *data, int nitem, int stride)
int info = pvm_pkfloat(        float *data, int nitem, int stride)
int info = pvm_pkint(            int *data, int nitem, int stride)
int info = pvm_pkuint(  unsigned int *data, int nitem, int stride)
int info = pvm_pkshort(         short *data, int nitem, int stride)
int info = pvm_pkushort(unsigned short *data, int nitem, int stride)
int info = pvm_pklong(           long *data, int nitem, int stride)
int info = pvm_pkulong( unsigned long *data, int nitem, int stride)
```

- Funkcje te umieszczają w aktualnym buforze nadawczym tablicę elementów:
  - data – wskazanie na element lub tablicę elementów danego typu,
  - nitem – liczba elementów danego typu umieszczana w buforze,
  - stride – krok, z jakim kolejne elementy są umieszczane w buforze.

Funkcje pakujące (C) (2)

```
int info = pvm_pkstr(char *sp)
```

- Funkcja pvm_pkstr umieszcza w buforze tekst wskazywany przez argument sp.

- Funkcja pvm_pa`int info = pvm_packf(const char *fmt, ... )`
  lub wiele sekwencji formatu fmt (notacja BNF):

format : null | init | format fmt

init : null | '%' '+'

fmt : '%' count stride modifiers fchar

fchar : 'c' | 'd' | 'f' | 'x' | 's'

count : null | [0-9]+ | '*'

stride : null | '.' ([0-9]+ | '*')

modifiers : null | modifiers mchar

mchar : 'h' | 'l' | 'u'

Formaty pakowania:

- **+ wymusza initsend - argument int (encoding)**
- **c pakowanie bajtów**
- **d pakowanie liczb całkowitych (int)**
- **f pakowanie liczb rzeczywistych (float)**
- **x pakowanie liczb zespolonych (complex float)**
- **s pakowanie tekstu (string)**

Modyfikatory typu:

- **h short (int)**
- **l long (int, float, complex float)**
- **u unsigned (int)**

Przykłady

```
C:
info = pvm_initsend(PvmDataDefault);
info = pvm_pkstr("initial data");
info = pvm_pkint(&size, 1, 1);
info = pvm_pkint(array, size, 1);
info = pvm_pkdouble(matrix, size*size, 1);

int    count, *iarry;
double darry[4];
pvm_packf("%+ %d %*d %41f", PvmDataRaw,
          count, count, iarry, darry);
```

Wysyłanie komunikatu

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

- Funkcja pvm_send:
  - wysyła zawartość aktualnego bufora nadawczego do procesu o identyfikatorze tid,
  - komunikat zostaje opatrzony etykietą msgtag,
  - funkcja ma charakter asynchroniczny.

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

- Funkcja pvm_mcast:
  - wysyła komunikat do wielu procesów,
  - tablica tids o rozmiarze ntasks zawiera identyfikatory procesów docelowych.

Pakowanie i wysyłanie

```
int info = pvm_psend(int tid, int msgtag,
                     char *data,
                     int count, int datatype)
```

- Funkcja pvm_psend:
  - pozwala spakować i przesłać jedną porcję danych,
  - pakuje count danych o adresie początkowym data do niezależnego bufora, a następnie wysyła komunikat,
  - argument datatype określa typ pakowanych danych:

PVM_STR	string	PVM_DOUBLE	double
PVM_BYTE	byte	PVM_DCPLX	double complex
PVM_SHORT	short	PVM_LONG	long integer
PVM_INT	int	PVM_USHORT	unsigned short int
PVM_FLOAT	real	PVM_UINT	unsigned int
PVM_CPLX	complex	PVM_ULONG	unsigned long int

Odbieranie komunikatu

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

- Funkcja pvm_recv:
  - czeka na komunikat o etykiecie msgtag, od procesu o identyfikatorze tid,
  - komunikat jest umieszczany w aktywnym buforze odbiorczym i zwracany jest identyfikator bufid tego bufora,
  - przed umieszczeniem danych bufor jest czyszczony,
  - msgtag = -1 – odczyt komunikatu o dowolnej etykiecie,
  - tid = -1 – odczyt komunikatu od dowolnego procesu.

Odbieranie komunikatu (2)

C:

```
#include <sys/time.h>
int bufid = pvm_trecv(int tid, int msgtag,
                      struct timeval *tmout)
```

Fortran:

```
call pvmftrecv(tid, msgtag,
               sec, usec, bufid)
```

- Funkcja pvm_trecv:
  - czeka na komunikat msgtag od procesu tid, aż upłynie wyspecyfikowany czas, potem wraca bez komunikatu,
  - **C**: pola tv_sec i tv_usec struktury tmout określają czas oczekiwania w [s] i [μs]

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

- Funkcja pvm_nrecv:
  - działa podobnie do pvm_trecv, ale czas oczekiwania wynosi zero.

Testowanie

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

- Funkcja pvm_probe:
  - funkcja nie odczytuje komunikatu lecz pobiera jego parametry (nadawcę i etykietę). Zwraca 0 jeśli go brak.

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

- Funkcja pvm_bufinfo:
  - zwraca informację o zawartości bufora o identyfikatorze bufid: bytes – długość komunikatu w bajtach, msgtag – etykieta komunikatu, tid – identyfikator procesu nadawcy.

Rozpakowywanie bufora

- Funkcje działające analogicznie, lecz przeciwnie do funkcji pakujących:

C:

```
int info = pvm_unpackf(      const char *fmt, ... )
int info = pvm_upkstr(             char *sp)
int info = pvm_upkbyte(            char *data, int nitem, int stride)
int info = pvm_upkcplx(           float *data, int nitem, int stride)
int info = pvm_upkdcplx(         double *data, int nitem, int stride)
int info = pvm_upkdouble(        double *data, int nitem, int stride)
int info = pvm_upkfloat(          float *data, int nitem, int stride)
int info = pvm_upkint(              int *data, int nitem, int stride)
int info = pvm_upkuint(    unsigned int *data, int nitem, int stride)
int info = pvm_upkshort(           short *data, int nitem, int stride)
int info = pvm_upkushort(unsigned short *data, int nitem, int stride)
int info = pvm_upkulong(  unsigned long *data, int nitem, int stride)
int info = pvm_upklong(            long *data, int nitem, int stride)
```

Przykłady

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

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

Przykłady

```
C:
      tid = pvm_parent();
      msgtag = 4 ;
      bufid = 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);
```

Przykłady

C:

```
struct timeval tmout;

tid = pvm_parent();
msgtag = 4 ;
tmout.tv_sec = 60;
tmout.tv_usec = 0;
if ((bufid = pvm_trecv( tid, msgtag, &tmout )) > 0) {
     pvm_upkint(tid_array, 10, 1);
     pvm_upkint(problem_size, 1, 1);
     pvm_upkfloat(input_array, 100, 1);
}
```

Grupy procesów

- Grupa - zbiór procesów o wspólnej nazwie tekstowej.
- Jeden proces może należeć do dowolnej liczby grup.
- Grupa nie może być pusta – jeśli opuści ją ostatni proces grupa przestaje istnieć.
- Procesy w danej grupie są ponumerowane (numery nie są związane z identyfikatorami tid).
- Gdy proces jest włączany do grupy jest mu przydzielany numer, gdy ją opuszcza numer jest zwalniany i może być ponownie przydzielony.
- Mogą wystąpić luki w numeracji procesów.

Dołączanie do / opuszczanie grupy

```
C: int inum = pvm_joingroup(char *group)
Fortran: call pvmfjoingroup(group, inum)
```

- Funkcja pvm_joingroup:
  - dołącza wywołujący ją proces do grupy o nazwie wskazanej przez argument group,
  - funkcja zwraca numer dołączonego procesu w grupie.

```
C: int info = pvm_lvgroup(char *group)
Fortran: call pvmflvgroup(group, info)
```

- Funkcja pvm_lvgroup:
  - usuwa wywołujący ją proces z grupy group.

Weryfikacja członków grupy

```
C: int size = pvm_gsize(char *group)
Fortran: call pvmfgsize(group, size)
```

- Funkcja pvm_gsize:
  - zwraca aktualną liczbę członków grupy group.

```
C: int tid = pvm_gettid(char *group, int inum)
Fortran: call pvmfgettid(group, inum, tid)
```

- Funkcja pvm_gettid:
  - zwraca identyfikator tid członka grupy group o numerze inum,
  - członkowie są numerowani od 0 do (size – 1), o ile nikt nie opuścił grupy (ciągła numeracja).

Bariera

```
C: int info = pvm_barrier(char *group, int count)
Fortran: call pvmfbarrier(group, count, info)
```

- Funkcja pvm_barrier:
  - oczekuje, aż count członków grupy group ją wywoła,
  - we wszystkich wywołaniach argument count musi być jednakowy (zaleca się aby był równy liczbie członków grupy),
  - inne funkcje nie gwarantują blokowania.

Rozgłaszanie

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

- Funkcja pvm_bcast:
  - wysyła komunikat o etykiecie msgtag zawierający dane z aktualnego bufora nadawczego do wszystkich członków grupy group,
  - nie wysyła do procesu ją wywołującego.

Redukcja

```
C: int info = pvm_reduce(void (*func)(),
                         void *data, int count, int datatype,
                         int msgtag, char *group, int rootginst)
```

- Funkcja pvm_reduce:
  - musi być wywołana przez wszystkich członków grupy,
  - argumenty wywołań powinny być jednakowe, za wyjątkiem data, wskazującego na dane o typie datatype (z wyjątkiem **`char`** i **`unsigned`**), na których ma być wykonana operacja func,
  - count – liczba danych dostarczanych przez każdy proces,
  - operacja jest wykonywana na każdej z count danych niezależnie, a wyniki są umieszczane pod adresem data w procesie o numerze rootginst w grupie group,
  - msgtag – unikalna etykieta komunikatu.

Operacje redukcji

- Argument func może przyjmować wartości:

PvmMin	– wyznaczenie minimum,
PvmMax	– wyznaczenie maksimum,
PvmSum	– wyznaczenie sumy,
PvmProduct	– wyznaczenie iloczynu.

- Można podać własną operację jako wskazanie na funkcję o prototypie (x – dane wejściowe, y – wyjściowe, num – ilość):

C:
```
void func(int *datatype, void *x, void *y,
          int *num, int *info)
```

```
C: int info = pvm_gather( void *result,  void *data,
                int count, int datatype, int msgtag,
                char *group, int rootginst)
```

Zbieranie

- Funkcja pvm_gather:
  - działa podobnie do pvm_reduce,
  - na danych nie jest wykonywana żadna operacja,
  - są one łączone w kolejności wyznaczonej przez rosnące numery procesów w grupie i kopiowane pod adres result tablicy typu datatype (o rozmiarze minimum *n*•count, gdzie *n* – liczba członków grupy) w procesie o numerze rootginst (w innych procesach argument result jest ignorowany).

Rozsyłanie

```
C: int info = pvm_scatter( void *result, void *data,
                int count, int datatype, int msgtag,
                char *group, int rootginst)
```

- Funkcja pvm_scatter:
  - dzieli wektor o adresie data (o rozmiarze minimum *n*•count, gdzie *n* – liczba członków grupy), przekazany przez proces o numerze rootginst (tylko w nim data ma znaczenie), na jednakowe części i kopiuje każdą z nich do obszarów o adresach określonych argumentami result we wszystkich procesach,
  - argument count określa liczbę danych w każdej z przesyłanych części.

Przykłady

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

```
C:
      inum = pvm_joingroup("worker");
                  ...
                  ...
      info = pvm_barrier("worker", 5);
```

```
C:
   info =  pvm_reduce(PvmMax, &myvals, 10, PVM_FLOAT,
                      msgtag, "worker", rootginst);
```

```
C:
   info =  pvm_gather(&getmatrix, &myrow, 10, PVM_INT,
                      msgtag, "workers", rootginst);
```

```
C:
   info =  pvm_scatter(&getmyrow, &matrix, 10, PVM_INT,
                       msgtag, "workers", rootginst);
```

Przykład użycia PVM

- Iteracyjne rozwiązywanie układu równań:

$$A \bullet x = b$$

- Zdominowana diagonalnie macierz A.
- Proces główny (PG) nie uczestniczy w obliczeniach:
  - czyta i sprawdza dane wejściowe,
  - uruchamia procesy obliczeniowe (PO),
  - rozsyła im dane
  - oraz przechowuje aktualne wartości wektora x i decyduje o zakończeniu obliczeń.

Wymieniane komunikaty

- UPDATE – przesłanie przez PO nowej wersji podwektora wektora x.
- NEWX – żądanie przesłania do PO aktualnego wektora x.
- STOP_TEST – przesłanie przez PO jego znacznika zakończenia obliczeń i odbiór znaczników pozostałych PO.

Pliki źródłowe

- defsPVM.h
- workPVM.c
- mainPWM.c

(na przykład
http://www.ia.pw.edu.pl/~karbowsk/orr/Bolek/)