CD-ROM: HITY: VisiBroker 3.4 for Java, InterBase 5, Informix Cloudscape 3.01

2/2000

magazyn

Linux & UNIX

Czym jest Postfix

Rozwiązania pracy grupowej Lotusa

Serwis LinuxNews.pl

LaTeX

E-commerce w Uniksie

Porównanie bibliotek Qt i Gtk+



CD ROM

Adres kontaktowy: magazyn@tao.com.pl

E-commerce w świecie Unixa na przykładzie InterMarket 3D część V..... 8

Serwis LinuxNews.pl..... 12
Ryszard Raszplewicz

Rozwiązania firmy Lotus w erze e-businessu - wprowadzenie..... 14
Opracował Janusz Wirtz

Uruchamianie systemu (Init-V)..... 18
Fragment książki wydanej przez Wydawnictwo Translator

Przepływ informacji w LaTeX-u..... 23
Piotr Bolek
Adam Dawidziuk

Produkty

Analizator sieci OBSERVER..... 28
Tomasz Barbaszewski

Programowanie

Gtk i Qt - próba porównania..... 32
Grzegorz Jacek Nalepa

Wprowadzenie do języka Scheme - część I..... 37
Marek Sawerwain

Technologie Sieciowe

ISS System Scanner. Czy Twój system jest bezpieczny?..... 42
Tomasz Ryś

MPI - standard przesyłania komunikatów - część II..... 46
Piotr Koch

Co to jest postfix?..... 52
Wojciech Bojdoł

Firma TAO Systems Polska, wydawca miesięcznika Linux & UNIX poszukuje osób do testowania i opisywania programów działających w systemie Linux. Zainteresowane osoby prosimy o kontakt z Redaktorem Naczelnym - Marcinem Koprowskim.
tel. 0501-134656, e-mail: marcin.koprowski@tao.com.pl

MPI – standard przesyłania komunikatów (część II)

Piotr Koch

Przyjrzyjmy się funkcjom, które realizują blokujące przesyłanie komunikatów, ich parametry to:

buf	wskazujący początek bufora komunikatów
count	ilość elementów do przesłania
datatype	wskazujący typ danych
dest	identyfikator (ranga)
tag	etykieta komunikatu
comm	obiekt komunikatora

Trzy przedstawione poniżej funkcje, różnią się między sobą trybem w jakim działają:

standardowy

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

buforowany

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

synchroniczny

```
int MPI_Ssend( void* buf, int count, MPI Datatype_datatype,int dest, int tag, MPI_Comm comm )
```

gotowości (ang. *read mode*)

```
int MPI_Rsend( void* buf, int count, MPI Datatype_datatype,int dest, int tag, MPI_Comm comm )
```

Nie blokujące nadawanie komunikatów

Istnieją nie blokujące odpowiedniki powyższych funkcji, różniące się oprócz sposobu działania dodatkowym parametrem *request*, który jest obiektem zwracanym przez system, pozwalającym sprawdzić funkcją *MPI_Test()* czy nie blokująca operacja już się zakończyła (ewentualnie z jakim błędem). Obiekt ten może być też parametrem funkcji *MPI_Wait()*, powodującej oczekiwanie wywołującego ją procesu na zakończenie operacji przesyłania danych. Jak w poprzednim wypadku istnieją cztery wersje nadające komunikat, w czterech różnych trybach:

nie blokujący standardowy

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

nie blokujący buforowany

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

nie blokujący synchroniczny

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

nie blokujący w trybie gotowości

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

Przy posługiwaniu się komunikacją nie blokującą może istnieć potrzeba wstrzymania pracy aż komunikat zostanie odebrany lub sprawdzenia czy odbiorca już go odebrał, te funkcje realizują dwie poniższe procedury:

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

Funkcja *MPI_Wait()* służy do wstrzymania działania procesu do momentu, gdy operacja nie blokującego odbioru lub wysyłania komunikatu identyfikowana przez parametr *request* zostanie zakończona. Obiekt *request* jest najczęściej tworzony przez nie blokującą operację odbioru/nadawania komunikatu. W takim wypadku dealokcja tego obiektu następuje w momencie wywołania operacji *Wait()*. Do sprawdzenia czy komunikat wysłany w sposób nie blokujący został odebrany bez wstrzymywania pracy programu służy funkcja *MPI_Test()*.

```
int MPI_Test( MPI_Request *request, int *flag,
MPI Status *status )
```

W wyniku wywołania tej funkcji zostaje ustawiona zmienna flag. Gdy operacja związana z obiektem *request* została zakończona, obiekt ten zostaje zwolniony, a zmienna *request* przyjmuje wartość MPI_REQUEST_NULL. W przeciwnym wypadku obiekt dalej istnieje, a zmienna *flag* przyjmuje wartość *false*.

Odbieranie komunikatów

Istnieją dwie funkcje realizujące odbieranie komunikatów, blokująca *MPI_Recv()* i nie blokująca *MPI_Irecv()*. Funkcjie te mają następujące parametry:

buf	wskazuje początek bufora komunikatu
count	maksymalna ilość elementów, które mogą być odebrane z bufora
datatype	wskazuje typ danych, który musi być taki sam jak przy nadawaniu tegoż komunikatu
source	identyfikator (ranga) procesu, od którego oczekujemy komunikatu
tag	oczekiwana etykieta komunikatu
comm	obiekt komunikatora.

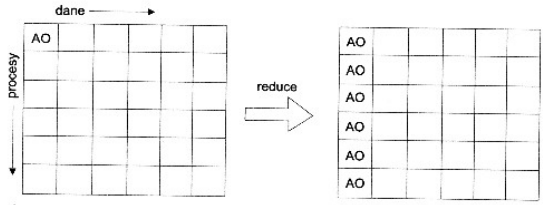
```
int MPI_Recv( void* buf, int count, MPI_Data-
type datatype,
int source, int tag, MPI_Comm comm,
MPI_Status *status)
```

Funkcja ta realizuje blokujący odbiór komunikatu. Powrót z procedury następuje dopiero gdy przychodzi komunikat o odpowiednich polach *tag*, *source*, *comm*. To znaczy, odbierając komunikat możemy określić z jakiego źródła, z jaką etykietą i z zasięgu jakiego komunikatora chcemy odebrać wiadomość. Musimy zapewnić aby wielkość bufora była taka, aby można było w nim pomieścić odebrane dane.

```
int MPI_Irecv( void* buf, int count, MPI_Dataty-
pe datatype,int source, int tag, MPI_Comm comm,
MPI_Request *request)
```

Ta procedura odbiera komunikat w sposób nie blokujący. Posiada w związku z tym dodatkowy parametr, przez który zwracany jest obiekt, z użyciem którego można stwierdzić czy cały komunikat znalazł się już w buforze (użycie tego obiektu zostało opisane przy okazji omawiania funkcji *MPI_Wait()* i *MPI_Test()*).

Istnieją również złożenia funkcji nadających z odbierającymi, działające w ten sposób, że korzystają z dwóch buforów lub wysyłają i odbierają korzystając z jednego bufora.



(Rysunek 1)

```
int MPI_Sendrecv( void *sendbuf, int sendcount,
MPI_Datatype sendtype, int dest, int sendtag,
void *recvbuf, int recvcount,
MPI_Datatype recvtype, int source,
MPI_Datatype recvtag, MPI_Comm comm,
MPI Status *status )
```

```
int MPI Sendrecv_replace( void* buf, int count,
MPI_Datatype datatype, int dest, int sendtag,
int source, int recvtag, MPI Comm comm,
MPI Status *status )
```

Funkcje pomocnicze

Istnieje zbiór funkcji sprawdzających czy nie nadszedł oczekiwany komunikat, bez odbierania tegoż komunikatu. Informacje są zwracane przez zmienną status. Poniższe funkcje mogą być pomocne w wypadku, kiedy chcemy zaalokować bufor do odbioru komunikatu, a nie znamy jego wielkości. Sprawdzenie może odbywać się w sposób blokujący, przy użyciu funkcji *MPI_Probe()*,

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

Tabela 1D. Zbiór procedur do obsługi grup procesów.

Procedury do obsługi grup procesów	
MPI_Group_compare	MPI_Group_difference
MPI_Group_incl	MPI_Group_intersection
MPI_Group_rank	MPI_Group_size
MPI_Group_excl	MPI_Group_free
MPI_Group_range_excl	MPI_Group_range_incl
MPI_Group_translate_ranks	MPI_Group_union

lub nie blokujący przy użyciu *MPI_Iprobe()*.

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

Przetwarzanie w grupach procesów

Podobnie jak i w przypadku komunikacji punkt – punkt, MPI okazuje się i tutaj systemem bardzo rozbudowanym, oferującym w związku z tym największe możliwości. Oferuje cały zbiór funkcji operujących na grupach (tworzenie grupy, dodawanie procesów do grupy, przeliczanie rang procesów miedzy poszczególnymi grupami). Na całość komunikacji w grupach składają oprócz operacji manipulujących na procesach i samych grupach, same funkcje realizujące samą komunikację. Przede wszystkim stosowane do rozsyłania danych i wykonywania różnych operacji redukcji (tabela 1.D).

Grupy procesów

Grupy procesów w MPI

- grupę tworzy zbiór N procesów
- procesy mają unikatowe identyfikatory będące liczbami całkowitymi od 0 do N-1
- nadane identyfikatory wewnątrz każdej grupy nazywa się rangą (ang. *rank*)
- każda grupa jest reprezentowana przez niewidoczny dla użytkownika obiekt
- początkowo wszystkie tworzone procesy są członkami grupy stworzonej przez predefiniowany komunikator MPI_COMM_WORLD

Do tworzenia i operowania na grupach procesów MPI przewiduje bardzo szeroką gamę procedur (patrz Tabela 1.D).

Oto omówienie kilku bardziej przydatnych funkcji dotyczących procesów.

Konwersji rangi procesu z jednej grupy do drugiej można dokonać przy użyciu funkcji:

```
MPI_Group_translate_ranks ( MPI_Group group1,
int n, int *ranks1, MPI_Group group2,
int *ranks2 )
```

Istnieje możliwość sprawdzenia czy grupy są identyczne, a przynajmniej podobne:

```
MPI_Group_compare ( MPI_Group group1,
MPI_Group group2, int *result )
```

Tabela 1C. Procedury do kolektywnego przetwarzania..

Procedury do kolektywnego przetwarzania	
MPI_Allgather	MPI_Allgatherv
MPI_Alltoallv	MPI_Barrier
MPI_Gatherv	MPI_Op_create
MPI_Reduce_scatter	MPI_Scan
MPI_Allreduce	MPI_Alltoall
MPI_Bcast	MPI_Gather
MPI_Op_free	MPI_Reduce
MPI_Scatter	MPI_Scatterv

Grupę tworzy się przez dodawanie do niej procesów. Sposoby w jaki proces lub cała grupa procesów może dołączyć do grupy mogą być bardzo różne.

```
MPI_Group_incl ( MPI_Group group, int n,
int *ranks,
MPI_Group newgroup )
```

Reorganizuje procesy z już istniejącej grupy *group* w nową grupę *newgruop* poprzez podanie listy procesów mających składać się na nową grupę *ranks*.

```
MPI_Group_excl ( MPI_Group group, int n,
int *ranks, MPI_Group *newgroup )
```

Działa podobnie jak poprzednia funkcja, z tym że wymienione w liście *ranks* procesy są jedynymi z grupy *group*, które nie wchodzą do nowej grupy *newgroup*.

```
MPI_Group_range_incl ( MPI_Group group, int n,
int ranges[][3], MPI_Group *newgroup )
```

MPI_Group_range_incl() dodaje procesy do grupy *newgroup* z grupy *group*, w zakresie sprecyzowanym przez parametr *ranges*. Parametr *ranges* jest trzy elementowym wektorem, który określa pierwszą rangę procesu, która ma być dodana do nowej, ostatnią rangę procesu, który ma wejść do nowej grupy oraz krok z jakim mają być pobierane elementy (1-oznacza, że wszystkie elementy z zakresu mają być wzięte pod uwagę, 2 – oznacza, że będzie brany co drugi element itd.)

```
MPI_Group_range_excl ( MPI_Group group, int n,
int ranges[][3], MPI_Group *newgroup )
```

MPI_Group_range_excll() działa podobnie jak *MPI_Group_range_incl()*, z tą różnicą, że zakres określony przez parametr *ranges* służy do określenia zakresu procesów z starej grupy, które nie będą wchodzić w skład nowej.

Komunikacja kolektywna

Kolektywna komunikacja w MPI

- kolektywna komunikacja wymaga aby wszystkie procesy były w tym samym zasięgu komunikacyjnym
- zasięg komunikacyjny określa komunikator
- procedury do kolektywnej komunikacji nie operują identyfikatorami komunikatów

Kolektywna komunikacja jest organizowana poprzez zbiór funkcji pokazany w Tabeli 1.C.

Jak widzimy w powyższej tabeli, MPI oferuje różnorodne procedury realizujące komunikacje typu reduce, broadcast, scater, gather oraz "all gather" i "all-to-all". Spróbujemy wyjaśnić przy pomocy rysunków na czym polega każdy z tych sposobów komunikowania.

Operacja rozpraszania (ang. *scatter*) polega na rozesłaniu danej z jednego procesu do wszystkich innych należących do danej grupy.

```
int MPI_Scatter ( void *sendbuf, int sendcnt,
MPI_Datatype sendtype, void *recvbuf, int
recvcnt, MPI_Datatype recvtype, int root,
MPI_Comm comm)
```

(patrz Rysunek 1)
Przeciwieństwem do operacji rozpraszania jest operacja zbierania (ang. *gather*). Operacja ta powoduje, że wartości z wszystkich procesów w grupie są gromadzone w jednym procesie.

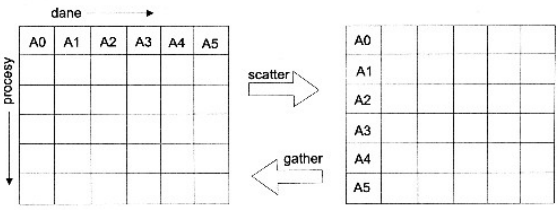
```
int MPI_Gather ( void *sendbuf, int sendcnt,
MPI_Datatype sendtype, void *recvbuf, int
recvcount, MPI_Datatype recvtype, int root,
MPI_Comm comm)
```

(Rysunek 2)
Operacja "allgather" polega na tym, że dane z każdego procesu w grupie są rozsyłane do wszystkich innych.

```
int MPI_Allgather ( void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int recvco-
unt, MPI_Datatype recvtype, MPI_Comm comm )
```
(Rysunek 3)

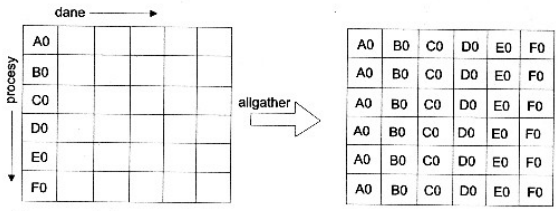
Komunikacja "all to all" polega na tym, że dane są wymieniane pomiędzy wszystkimi procesami w grupie.

```
int MPI_Alltoall( void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int
recvcnt, MPI_Datatype recvtype, MPI_Comm comm )
```
(Rysunek 4)



Rysunek 2

W rozbudowanych aplikacjach bardzo użyteczne są operacje redukcji. Procedury realizujące coś takiego, są zaimplementowane we wszystkich porównywanych narzędziach opierających się na przesyłaniu komunikatów (MPI, PVM, P4). Jeśli chodzi o ten rodzaj funkcji, twórcy standardu MPI wykazali się wyjątkowym rozmachem i stworzyli bardzo rozbudowany mechanizm. Wszystkie możliwe rodzaje operacji redukcji przedstawione są poniżej.



Rysunek 3

```
int MPI_Reduce ( void *sendbuf, void *recvbuf,
int count, MPI_Datatype datatype, MPI_Op op,
int root, MPI_Comm comm )
```

Działanie *MPI_Reduce()* polega na redukcji wartości z wszystkich procesów do jednej wartości (Rysunek 5).

```
int MPI_Allreduce ( void *sendbuf, void *re-
cvbuf, int count, MPI_Datatype datatype,
MPI_Op op, MPI_Comm comm.)
```

Łączy wartości z wszystkich procesów i dystrybuuje wynik z powrotem do wszystkich procesów (Rysunek 6).

```
int MPI_Reduce_scatter ( void *sendbuf, void
*recvbuf, int *recvcnts, MPI_Datatype dataty-
pe, MPI_Op op, MPI_Comm comm)
```

Oblicza wszystkie wartości, tak jak w wypadku redukcji, a następnie rozsyła, tak jak przy operacji scatter (Rysunek 7).

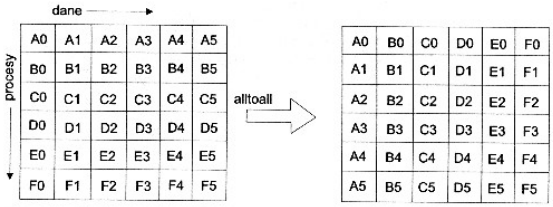
```
int MPI_Scan ( void *sendbuf, void *recvbuf,
int count, MPI_Datatype datatype, MPI_Op op,
MPI_Comm comm )
```

Oblicza skanując (dokonując cząstkowej redukcji) dane ze zbioru procesów (Rysunek 8).

Synchronizacja barierowa

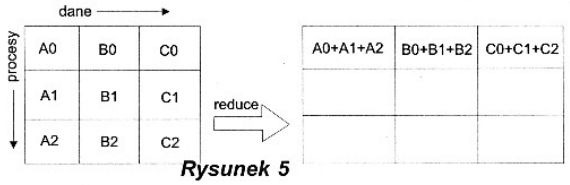
Bariera jest takim punktem w programie, do którego muszą dotrzeć wszystkie procesy, aby którykolwiek z nich mógł ją przekroczyć. Synchronizacja barierowa jest często używana w programach obliczeniowych do upewnienia się, że został zakończony jakiś etap obliczeń zanim zastanie rozpoczęty następny. Ten sam efekt może być osiągnięty przez wielokrotne tworzenie procesów, jednak synchronizowanie raz utworzonej grupy procesów jest bardziej efektywne (Rysunek 9).

```
int MPI_Barrier( MPI_Comm comm )
```

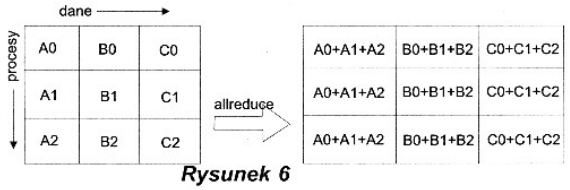


Rysunek 4

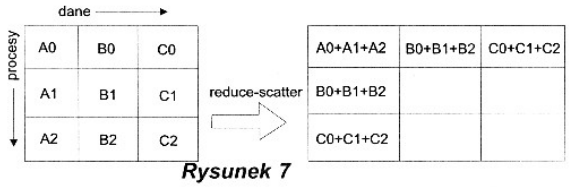




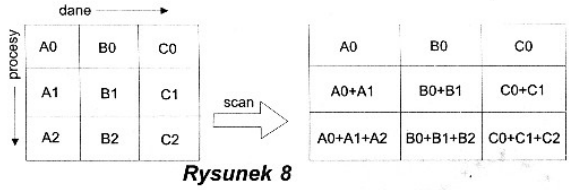
Rysunek 5



Rysunek 6



Rysunek 7



Rysunek 8

Funkcja ta realizuje synchronizację barierową w MPI. Blokuje ona proces wywołujący ją na czas, do kiedy wszystkie procesy należące do grupy skojarzonej z komunikatorem podanym jako parametr funkcji jej nie wywołają.

Modularność

MPI daje możliwość modularnego programowania dzięki mechanizmom komunikatora, umożliwiającym ukrywanie pewnych informacji, gdy budujemy modularne programy, co umożliwia wyspecyfikowanie pewnych komponentów programów, wewnątrz których dokonujemy enkapsulacji operacji komunikacyjnych i dostarczamy lokalnej przestrzeni nazw dla tych procesów. Oznacza to, że mamy możliwość takiego rozdzielenia grup procesów poprzez przydzielenie im oddzielnych komunikatorów, że przesyłanie komunikatów w jednej i drugiej grupie jest od siebie niezależne. Nie ma możliwości, że przez pomyłkę, komunikat dotrze do złego odbiorcy, lub pokryją się etykiety komunikatów wysyłanych przez procesy dokonujące różnych obliczeń, prowadząc do niedeterministycznego działania programu.

W operacjach komunikacyjnych w MPI zawsze wyspecyfikowany jest obiekt komunikatora. Identyfikuje on grupę procesów, operacje komunikacyjne jakie mogą mieć miejsce oraz kontekst, w jakim te operacje mogą mieć miejsce. Widzimy więc, że grupa procesów pozwala na komunikowanie się podzbioru procesów do niej należących pomiędzy sobą z użyciem identyfikatorów i na wykonywanie operacji kolektywnych bez angażowania innych procesów. Konteksty wiążą się z komunikatami. Operacje odbioru mogą odbierać komunikaty wyłącznie jeśli były wysłane w tym samym kontekście. Stąd, dwóm procedurom używającym różnych kontekstów dla swojej wewnętrznej komunikacji, nie grozi niebezpieczeństwo powstania niejednoznaczności.

Często do operacji komunikacyjnych używa się domyślnego komunikatora MPI_COM_WORLD, który zawiera wszystkie procesy uczestniczące w przetwarzaniu MPI i definiuje domyślny kontekst. Teraz poznamy pięć funkcji, które pozwalają używać obiektu komunikatora w bardziej uniwersalny sposób.

```
int MPI_Comm_dup ( MPI_Comm comm,
MPI_Comm *comm_out )
```

Pozwala tworzyć nowy obiekt komunikatora, składający się z tej samej grupy procesów, ale posiadający nowy kontekst.

```
int MPI_Comm_split ( MPI_Comm comm, int
color, int key, MPI_Comm *comm_out )
```

Tworzy jeden lub więcej nowych komunikatorów zawierających podzbiór danej grupy procesów. Procesy podzielone w ten sposób mogą komunikować się między sobą bez obawy o konflikt z innymi, konkurencyjnie wykonującymi się obliczeniami. Funkcja ta jest kolektywną, więc wymaga wywołania przez wszystkich członków należących do grupy. Nowy komunikator jest tworzony dla każdej wartości *color*, różnej od MPI_UNDEFINED. Każdy nowy komunikator zawiera te procesy, które wyspecyfikowały odpowiadającą mu wartość *color* w wywołaniu *MPI_Comm_split ()*.

```
int MPI_Intercomm_create ( MPI_Comm lo-
cal_comm, int local_leader, MPI_Comm pe-
er_comm, int remote_leader, int tag, MPI_Comm
*comm_out )
```

Program może stworzyć interkomunikator, który umożliwi komunikację pomiędzy dwoma grupami posiadającymi różne komunikatory i konteksty (domyślny komunikator MPI_COMM_WORLD jest intrakomunikatorem). Iterkomunikator może być stworzony poprzez kolektywne wywołanie funkcji *MPI_Intercomm_create()* w obu grupach, które mają być połączone. Wykonanie tej funkcji wymaga wytypowania "lidera". "Lider" jest procesem, poprzez którego odbywa się komunikacja pomiędzy obiema połączonymi grupami, może nim być dowolny proces w grupie. W wywołaniu funkcji podaje się range "lidera" lokalnego i zdalnego, czyli wytypowanego w grupie, z którą chcemy się komunikować. Etykieta *tag* jest wartością jaką będą znaczone przesyłane komunikaty pomiędzy dwoma "liderami"

```
int MPI_Comm_free ( MPI_Comm *commp )
```

Funkcja ta służy do zwolnienia obiektu komunikatora (komunikator przestaje istnieć).

dokończenie artykułu na stronie 53