CD-ROM: Oprogramowanie dla SCO OpenServer 5.0.5, Grafika i bazy danych w Linuksie

1/2000

magazyn

Linux & UNIX

Linux w NT

MPI – standard przesyłania komunikatów

JavaScript w praktyce

IDS – Intrusion Detection Systems

Testy AlphaServer 1000A x/500

Programowanie w GNOME

Cykl TCP/IP od podstaw

CD ROM

NR 1(17)/2000 STYCZEŃ CENA 17 ZŁ ISSN 1506-1299

magazyn Linux & UNIX

Adres kontaktowy: magazyn@tao.com.pl

Pakiety w dystrybucji LaTeX-a 5
Adam Dawidziuk, Piotr Bolek

Wykorzystanie JavaScript do wymiany informacji między ramkami 7
Fragment książki wydanej przez Wydawnictwo Translator

Który system jest szybszy. Komentarz do testów 10
Marcin Oczkowski

Y2K: pIERWSZE SPOTKANIE PO. XV Jubileuszowe Targi Komputer Expo 2000 11

Jak to zrobić w Linuksie 13
Bartosz Maruszewski

Produkty

Oprogramowanie HP OmniBack II 14
Stanisław Jagielski

Testowanie szybkości działania komputera AlphaServer 1000A x/500 17
Dariusz Knociński, Dariusz Pawlak

Drukarka HP DesignJet ColorPro CAD dla Linuxa 20

Rozwiązania dla sieci SAN 21
Piotr Michałkiewicz

INFORMIX MEDIA360 23

Programowanie

Szósty palec stopy 24
Marek Sawerwain

JavaScript w praktyce - część II 28
Wiktor Łukasik

Technologie Sieciowe

Intrusion Detection Systems 33
Rafał Wojtczuk

TCP/IP od podstaw - część IV 37
Wojciech Bojdoł

Linda kontra PVM? - część II 39
Piotr Koch

MPI - standard przesyłania komunikatów - część I 43
Piotr Koch

UNIX

iCOD. Nowe rozwiązanie HP 48
Piotr Michałkiewicz

CD-ROM SCO OpenServer Optional Services 49
Marek Partyka

SCO OpenServer Supplements 50
Ewa Baćmaga

CD-ROM SCO Vision2K 51
Ewa Baćmaga

Tips&Tricks

Proste wyświetlanie zawartości archiwów 53
Janusz Wirtz

MPI – standard przesyłania komunikatów (część I)

Piotr Koch

Przekazywanie komunikatów (ang. *message passing)* to jedno z najpopularniejszych rozwiązań w programowaniu równoległym. Istnieją liczne implementacje systemów umożliwiających programowanie tą metodą, zarówno na maszynach wieloprocesorowych z pamięcią dzieloną, jak i na sieci stacji roboczych. Pomimo podobnej funkcjonalności biblioteki te (dostarczane m. in. przez liczących się producentów sprzętu, jak choćby Cray, SGI, Convex czy Intel) istotnie różnią się między sobą, co utrudnia pisanie przenośnego oprogramowania wykorzystującego ich możliwości.

MPI (ang. *Message Passing Interface*) jest specyfikacją ogólnie przyjętego i coraz popularniejszego standardu biblioteki, który powstał w oparciu o najlepsze cechy istniejących produktów tego typu, przy współudziale ich autorów. W tworzeniu tego standardu brali udział zarówno dostawcy sprzętu tacy jak IBM, Intel, TMC, Meiko, Cray, Convex, Ncube, jak również twórcy oprogramowania specjalizujący się w pisaniu bibliotek do przetwarzania równoległego i rozproszonego, takich jak PVM, Linda, p4, Express czy Chameleon.

Głównymi zaletami biblioteki procedur MPI jest całkowita przenośność, łatwość użycia, różnorodność trybów przesyłania danych (włącznie z pełni asynchronicznym), rozbudowane operacje komunikacji globalnej (ang. *collective communication*), wsparcie dla tworzenia równoległych bibliotek (enkapsulacja komunikacji za pomocą kontekstów i grup procesów), mechanizm pozwalający tworzyć i operować na tzw. topologiach procesów, zwiększający czytelność algorytmów równoległych i mogący poprawić ich wydajność, wbudowane mechanizmy pomiaru wydajności (ang. *profiling*), buforowanie definiowane przez użytkownika, działanie w środowiskach wielowątkowych.

Do tej pory powstało wiele udanych implementacji MPI. Najpopularniejsze z nich, dostępne nieodpłatnie to: MPICH i LAM. Większość producentów superkomputerów dostarcza własne implementacje standardu. Powstaje coraz więcej oprogramowania wykorzystującego MPI. W artykule tym spróbuję zarysować możliwości tych pakietów i przedstawić podstawowe cechy i pojęcia związane ze standardem MPI.

Podstawowe właściwości pakietu MPI

- komunikacja punkt-punkt
- operacje kolektywne
- przetwarzanie grupowe
- konteksty komunikacji
- topologia procesów
- środowiskowe zarządzanie
- możliwość śledzenia przetwarzania
- interfejs wspomagający pomiar wydajności

Pojęcie procesu w MPI

- program składa się z niezależnych procesów posiadających własną przestrzeń adresową
- każdy proces wykonuje swój kod
- nie ma mechanizmów do przypisywania kodu procesorom
- grupy procesów mogą być tworzone i suwane
- zbiór procesów tworzących grupę jest statyczny, brak mechanizmu dynamicznego zarządzania procesami
- procesy mogą należeć do kilku grup jednocześnie
- komunikacja odbywa się dzięki procedurom MPI

Komunikacja w systemie MPI

- procesy są identyfikowane poprzez grupę, do której należą oraz relatywny numer w tej grupie (0, 1, 2,..., N-1)
- komunikaty są etykietowane przez kontekst komunikatu i relatywny do kontekstu znacznik komunikatu
- grupy są mechanizmem służącym do podziału przestrzeni procesów
- konteksty są używane do podziału przestrzeni komunikatów
- grupy i konteksty są łączone w obiekcie komunikatora, kontekst nie jest widoczny z poziomu aplikacji
- komunikator określa zakres operacji związanych z komunikacją

Obiekt komunikatora

- komunikator służy do tworzenia niezależnych przestrzeni komunikatów
- zapobiega niejednoznacznościom podczas wybierania komunikatów kiedy aplikacja wywołuje procedury biblioteczne w celu wykonania operacji przesłania komunikatu
- komunikator łączy razem grupy i konteksty, określa ramy operacji komunikacyjnych, jego działanie jest przeźroczyste dla użytkownika

MPI oferuje bardzo szeroki zakres funkcji , jest ich około 129 (*Tabela 1*). Tak duża ilość sugeruje dużą złożoność MPI. Jednak tak nie jest. Wynika to z faktu, ze proste podstawy MPI dały możliwość rozwinięcia funkcji wspierających rozwiązywanie określonych typów problemów (np. operacje w rożnych topologiach). Posługując się sześcioma funkcjami systemu MPI można napisać w pełni funkcjonalną aplikację.

MPI definiuje swoje typy danych. Pojęcie procesu w MPI ma wiele wspólnego z procesem w systemie Unix. Programy MPI składają się z autonomicznych procesów, wykonujących swój własny kod (w C lub Fortranie), w stylu MIMD (wiele instrukcji, wiele danych). Kod wykonywany przez każdy kod nie musi być identyczny. Procesy komunikują się przez wywoływanie funkcji systemu MPI, stanowiących pewnego rodzaju interfejs, tworzący wspólną płaszczyznę komunikacyjną pomiędzy rozproszonymi częściami programu. Naj-

Tabela 1. Klasyfikacja funkcji zaimplementowanych w systemie MPI.

A. Procedury zarządzające środowiskiem	
MPI_Abort	MPI_Errhandler_create
MPI_Errhandler_set	MPI_Error_class
MPI_Get_processor_name	MPI_Init
MPI_Errhandler_free	MPI_Errhandler_get
MPI_Initialized	MPI_Wtick
B. Procedury realizujące komunikacje punkt-punkt	
MPI_Bsend	MPI_Bsend_init
MPI_Cancel	MPI_Get_count
MPI_Iprobe	MPI_Irecv
MPI_Request_free	MPI_Rsend
MPI_Send_init	MPI_Sendrecv
MPI_Ssend_init	MPI_Start
MPI_Test_cancelled	MPI_Testall
MPI_Wait	MPI_Waitall
MPI_Buffer_attach	MPI_Buffer_detach
MPI_Get_elements	MPI_Ibsend
MPI_Recv	MPI_Recv_init
MPI_Rsend_init	MPI_Send
MPI_Sendrecv_replace	MPI_Ssend
MPI_Startall	MPI_Test
MPI_Testany	MPI_Testsome
MPI_Waitany	MPI_Waitsome
C. 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
D. 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
E. Procedury do komunikacji	
MPI_Comm _compare	MPI_Comm_create
MPI_Comm_group	MPI_Comm_rank
MPI_Comm_size	MPI_Comm_split
MPI_Comm_dup	MPI_Comm_free
MPI_Comm_remote_group	MPI_Comm_remote_size
MPI_Comm_test_inter	MPI_Intercomm_create
F. Procedury do definiowania typów	
MPI_Type_commit	MPI_Type_contiguous
MPI_Type_count	MPI_Type_extent
MPI_Type_free	MPI_Type_hindexed
MPI_Type_lb	MPI_Type_size
MPI_Type_hvector	MPI_Type_indexed
MPI_Type_struct	MPI_Type_ub
G. Procedury do wirtualnych topologii	
MPI_Cart_coords	MPI_Cart_create
MPI_Cart_rank	MPI_Cart_shift
MPI_Dims_create	MPI_Graph_create
MPI_Graph_neighbors	MPI_Graph_neighbors_count
MPI_Cart_get	MPI_Cart_map
MPI_Cart_sub	MPI_Cartdim_get
MPI_Graph_get	MPI_Graph_map
MPI_Graphdims_get	MPI_Topo_test

częściej każdy proces jest wykonywany we własnej przestrzeni adresowej, mimo że możliwa jest również implementacja MPI na maszyny z pamięcią dzieloną. Program wykonywany przez każdy proces może być sekwencyjny lub wielowątkowy.

MPI nie dostarcza mechanizmów do przydzielania systemowi MPI procesów w celu wspólnego przetwarzania i przypisywania procesów do procesorów. Opisywany standard nie pozwala dynamicznie tworzyć i usuwać procesów podczas wykonywania programu (całkowita ilość procesów jest stała). MPI identyfikuje procesy według relatywnej rangi w grupie, która jest liczbą całkowitą z zakresu 0...(rozmiar grupy – 1). Standard MPI nie specyfikuje w jaki sposób mają się rozpocząć równoległe obliczenia.

Pierwszą czynnością jaką powinien wykonać każdy proces chcący brać udział w przetwarzaniu rozproszonym jest wywołanie funkcji inicjalizującej proces w systemie MPI, wywołanie to powinno mieć miejsce przed jakimkolwiek wywołaniem funkcji systemu. Takie zadanie spełnia funkcja *MPI_Init()*. W wyniku jej wywołania zbiór atrybutów opisujących środowisko wykonawcze jest łączony z obiektem komunikatora MPI_COMM_WORLD.

```
int MPI_Init(int *argc, char ***argv)
```

Argumenty *argc* i *argv* są potrzebne gdy używamy biblioteki z poziomu języka C i służą do przekazania argumentów z jakimi został wywołany program.

Najprostszy program napisany z użyciem biblioteki MPI może wyglądać tak:

```
int main(int argc, char **argv)
{
MPI_Init(&argc, &argv);
/* program głowny */
MPI_Finalize(); /* zobacz poniżej */
}
```

Funkcja *MPI_Finalize()* kończy pracę całego systemu MPI, żadna procedura tego sytemu nie może być wywołana przez jakikolwiek proces (nawet *MPI_Init()*). Użytkownik musi się upewnić przed wywołaniem tej

funkcji, że wszystkie procedury komunikacyjne zostały zakończone.

Do podstawowych funkcji tego systemu, oprócz procedur do wysyłania i odbioru komunikatów, należą jeszcze dwie, które pozwalają stwierdzić jaka jest aktualna ilość procesów biorących udział w przetwarzaniu i poznać wywołującemu funkcje procesowi jego identyfikator w grupie.

```
int MPI_Comm_size( MPI_Comm comm, int *size )
```

Parametrem tej funkcji jest obiekt komunikatora przypisanego grupie, domyślnie po wykonaniu funkcji *MPI_Init()* procesy należą do grupy zarządzanej przez komunikator MPI_COMM_WORLD. Funkcja *MPI_Comm_size()* zwraca ilość procesów należących do grupy związanej z podanym komunikatorem. Podanie komunikatora MPI_COMM_WORLD powoduje zwrócenie przez funkcje ilości wszystkich procesów biorących udział w przetwarzaniu, czyli zarejestrowanych w systemie MPI poprzez wykonanie funkcji *MPI_Init()*. Można w ten sposób sprawdzić, czy udało się uruchomić odpowiednio dużo procesów zdalnych, aby wykonać dane zadanie.

```
int MPI_Comm_rank( MPI_Comm comm, int *rank )
```

Powyższa procedura pozwala uzyskać procesowi wywołującemu jego numer w grupie zwany rangą, który jest jego identyfikatorem podczas przesyłania komunikatów. Proces może dzięki tej funkcji zapytać system MPI "kim jestem ?", na co ten odpowiada mu numerem, który go jednoznacznie identyfikuje w zakresie działania podanego komunikatora, numer ten może być używany przez inne procesy do adresowania komunikatów do tego procesu.

W MPI nie ma zdefiniowanych funkcji związanych z obsługą procesów (tworzeniem, pobieraniem informacji o nich i o ich rodzicach), gdyż to nie mieści się w ramach tego standardu. To co jest potrzebne każdemu procesowi, to informacja jaki numer ma w danej grupie i ewentualnie ile procesów tworzy tę grupę. Realizujące to funkcje *MPI_Comm_size() i MPI_Comm_rank()* zostały opisane wcześniej.

Istnieje cały zbiór procedur konfigurujących środowisko pracy MPI, których jednak omawianie wykracza znacznie poza zakres artykułu, oto kilka przykładowych (patrz Tabela 1.A).

Istnieje funkcja pozwalająca dowiedzieć się na jakim procesorze został uruchomiony proces ją wywołujący. Postać tej procedury jest następująca:

```
int MPI_Get_processor_name( char *name, int
*resultlen)
```

W wyniku wywołania tej procedury zwracany jest ciąg znaków *name* opisujący procesor, na którym jest uruchomiony proces, może mieć on wartość:

```
""\processor 9 in rack 4 of mpp.piko.org" lub "231"
```

gdzie w pierwszym wypadku ciąg znaków wskazuję konkretny procesor w komputerze podłączonym do sieci, a w drugim jest numerem procesora uruchomionego w homogenicznym systemie. Wartość *resultlen* określa długość zwracanego, jako wynik operacji, ciągu znakowego.

Użycie tej procedury może być wykorzystane do optymalizacji pracy programu, procesy mogą być tak łączone w topologie (czyli w wirtualne, regularne kształty, dwu lub trój wymiarowe, w zakresie których komunikacja między procesami może odbywać się przez wskazanie kierunku lub miejsca zajmowanego w danej topologii, a nie rangi procesu). Połączenie procesów w odpowiednie topologie powoduje, że komunikaty mogą być wymieniane z sąsiednimi procesami szybciej (na przykład w obrębie jednej maszyny wieloprocesorowej), lub określone zadania mogą być zlecane określonym węzłom przetwarzającym, które tę operacje mogą zrealizować sprawniej.

W MPI programista ma do dyspozycji cały zbiór procedur umożliwiający przechwytywanie błędów, ustalanie jak mają być dane błędy obsługiwane i pozwalających na tworzenia własnych procedur obsługi błędów (funkcje *MPI_Errhandler_create(), MPI_Errhandler_set(), MPI_Error_class(), MPI_Errhandler_free()*). Przechwytywane kody błędów mogą być przetwarzane na odpowiednie komunikaty tekstowe przy użyciu funkcji *MPI_Error_string()*.

Komunikacja

Komunikacja wymaga aby istniały przynajmniej dwa procesy - wysyłający komunikat i odbierający. Gdy jakiś proces wysyła komunikat, to musimy zdecydować, czy odbiorca musi współpracować (to znaczy, czy odbiorca musi być gotów do otrzymania komunikatu, gdy jest on wysyłany), czy też proces może wysyłać jeden lub więcej komunikatów niezależnie od stanu odbiorcy.

Przy komunikacji synchronicznej wymiana komunikatów jest akcją atomową, wymagającą współudziału zarówno procesu wysyłającego - nadawcy, jak i procesu odbierającego - odbiorcy. Jeśli nadawca jest gotowy do wysyłania, ale odbiorca nie jest gotów do odbierania, to nadawca zostaje wstrzymany. Podobnie, gdy odbiorca jako pierwszy jest gotowy do komunikacji, to zostaje wstrzymany. Czynność komunikowania się synchronizuje ciągi wykonań obu procesów..

Inne rozwiązanie polega na tym, że nadawca może wysłać wiadomość i kontynuować pracę bez wstrzymywania wykonywania. Ten rodzaj komunikacji nazywamy asynchronicznym, ponieważ nie ma żadnego związku czasowego między ciągami wykonań obu procesów. W czasie gdy jest wysyłany komunikat, odbiorca może wykonywać dowolne instrukcje, a dopiero później sprawdzić, czy kanałem komunikacyjnym nadeszły komunikaty.

Istotną różnicą między tymi dwoma schematami jest potrzeba buforowania komunikatów. Przy komunikacji asynchronicznej nadawca może wysłać wiele komunikatów zanim odbiorca odbierze je z kanału komunikacyjnego. Zatem kanał musi być przygotowany do buforowania nieograniczonej liczby komunikatów. Jeśli liczba komunikatów w kanale jest ograniczona, to w końcu nadawca zostanie wstrzymany. Przy komunikacji synchronicznej w dowolnym momencie w kanale jest co najwyżej jeden komunikat, więc buforowanie jest zbędne.

Innym ważnym pojęciem związanym z przesyłaniem komunikatów jest blokujące lub nie blokujące wysyłanie komunikatów. Blokujące wysyłanie komunikatu powoduje, że proces wysyłający komunikat jest

wstrzymywany do czasu, kiedy proces odbierający nie odbierze całego komunikatu. W przeciwnym przypadku, kiedy nadajemy w sposób nie blokujący, proces nadający może przystąpić do kontynuowania obliczeń, nie czekając, aż komunikat zostanie odebrany. Drugi sposób komunikacji jest bardziej efektywny, pozwala lepiej wykorzystać czas procesora przeznaczony procesowi, ale zarazem wymaga więcej pracy ze strony programisty, który musi sprawdzić, czy odpowiednie komunikaty zostały odebrane.

Najprostszym mechanizmem komunikacji w MPI jest transmisja danych pomiędzy parą procesów, gdzie jedna strona odbiera, a druga nadaje. MPI dostarcza zbiór funkcji do wysyłania i odbioru komunikatów pozwalających wymieniać dane z określonym typem i przypisaną etykietą (ang. *tag*). Określenie typu przesyłanej wiadomości jest potrzebne do obsługi sieci heterogenicznych, typ jest potrzebny do przeprowadzenia odpowiedniej konwersji reprezentacji danych przy przesyłaniu z jednej architektury do drugiej. Etykieta pozwala wybierać komunikaty, które chcemy odbierać. Zakres oferowanych procedur jest bardzo szeroki, bez wątpienia jeśli chodzi o przesyłanie komunikatów, zarówno w komunikacji punkt-punkt jak i operacjach grupowych jest to jeden z najbardziej rozbudowanych standardów (patrz Tabela 1.B).

Nie sposób omówić wszystkich sposobów komunikacji, ale przyjrzymy się podstawowym funkcjom realizującym tę komunikację.

Podstawowe własności MPI w komunikacji punkt-punkt to:

- MPI zapewnia komunikacje punkt-punkt między parą procesów
- wybieranie komunikatu odbywa się według jego rangi (ang. *rank*, chodzi o numer nadany procesowi wewnątrz grupy) i etykiety (ang. *tag*)
- ranga i identyfikator są interpretowane w zależności od zasięgu komunikacji
- ranga i identyfikator mogą być dowolne

Komunikacja może odbywać się w następujących trybach:

- tryb standardowy, wysyłanie komunikatu może być rozpoczęte nawet jeśli odbiorca nie jest jeszcze przygotowany
- tryb gotowości, wysłanie komunikatu może mieć miejsce tylko wtedy gdy odbiorca jest przygotowany
- tryb synchroniczny, podobnie jak standardowy, z tym że operacja wysyłania nie będzie zakończona dopóki dostarczenie przez system komunikatu nie będzie zagwarantowane
- tryb buforowany, tak jak w trybie standardowym, ale zakończenie zawsze jest niezależne od przygotowania odbiorcy, komunikat może być zbuforowany aby zapewnić dostarczenie

Praca w trybie buforowanym

W trybie buforowanym użytkownik dostarcza bufor, który może być użyty do buforowania komunikatów, tak więc proces wysyłający komunikat może zawsze powrócić z operacji wysyłania zanim komunikat zostanie odebrany. Dostarczenie systemowi buforu użytkownika odbywa się przez wywołanie funkcji:

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

A zwrócenie zasobów systemu (zwolnienie bufora), zajmowanych przez bufor użytkownika odbywa się poprzez wywołanie konstrukcji:

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

Zwolnienie bufora może odbyć się dopiero wtedy, gdy wszystkie operacje komunikacyjne zakończyły pracę z buforem.

Tabela 2. Typy danych standardowo przesyłanych przez system MPI.

Typ danych w MPI	Typ danych w C
MPI_CHAR	Signed char
MPI_SHORT	Signed short int
MPI_INT	Signed int
MPI_LONG	Signed long int
MPI_UNSIGNED CHAR	unsigned char
MPI_UNSIGNED SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Blokujące nadawanie komunikatów

Procedury nadające komunikaty prócz wysyłania danych, zajmują się również ich konwersją na format zgodny z architekturą maszyny, która ma je odebrać. Stąd w większości funkcji znajduje się parametr *datatype* określający typ wysyłanych danych. Możliwe dostępne typy przedstawia Tabela 2.

Typ *MPI_PACKED* oznacza, że użytkownik sam zajął się pakowaniem danych do bufora. Pozwala to umieszczać kilka tablic różnych typów danych w jednym buforze (tak jak to miało miejsce na przykład w PVM). Istnieją następujące funkcje do pakowania, rozpakowywania danych:

```
int MPI_Pack ( void* inbuf, int incount,
MPI_Datatype datatype, void *outbuf,
    int outsize, int *position,
    MPI_Comm comm )
```

Powyższa funkcja realizuje pakowanie danych do bufora. Odebrany komunikat musi być rozpakowany, w ten sam sposób jak był pakowany, do realizacji tego przeznaczona jest funkcja *MPI_Unpack()*.

```
int MPI_Unpack( void* inbuf, int insize,
int *position,void *outbuf, int outcount,
MPI_Datatype datatype, MPI_Comm comm )
```