CD-ROM: Oprogramowanie narzędziowe dla Linuksa i Internetu

4/2000

magazyn

Linux & UNIX

Oprogramowanie FireMail

Implementacja RAID w Linuksie

Dokumentacja GNU/Linux

IPv6 w praktyce

LaTeX

Język Scheme

CD ROM

NR 4(20)/2000 KWIECIEŃ CENA 17 ZŁ ISSN 1506-1299

Adres kontaktowy: magazyn@tao.com.pl

Implementacja RAID w Linuxie 8
Dariusz Piotr Pawlak

Korzystanie z dokumentacji GNU/Linux 11
Grzegorz Jacek Nalepa

LaTeX dla matematyków 17
Adam Dawidziuk
Piotr Bolek

Produkty

Netscape Messaging Server 21
SCO Tarantella Enterprise II 23
Opracował Janusz Wirtz

Programowanie

Wprowadzenie do języka Scheme - część II 25
Marek Sawerwain

A kiedy już nam się znudzi LILO 30
Marek Sawerwain

Technologie sieciowe

Model krystaliczny (Crystalline Model) - część II 36
Piotr Koch

6bone w praktyce - część II 44
Wojciech Bojdoł

Procmail i FireMail. Nasz osobisty, inteligentny listonosz 46
Marcin Marszałek

Linux dla każdego

Slrn - oprogramowanie do obsługi newsów internetowych - część I 50
Maciej Wiśniewski

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

Model krystaliczny (*Crystalline Model*)

część II

Piotr Koch

W modelu krystalicznym procesy są luźno zsynchronizowane, pracują niezależnie między zdarzeniami komunikacyjnymi, aż do chwili, gdy ma miejsce zdarzenie komunikacyjne, w którym muszą wszystkie uczestniczyć. Jest to podstawowa zasada funkcjonowania tego modelu, wszystkie procesy są zorganizowane w regularną topologię i każdy może się komunikować wyłącznie ze swoim sąsiadem. W tym sensie model jest podobny do modelu CSP, tworzenie procesów ma miejsce tylko raz, w momencie uruchomienia programu, a procesy są rozmieszczone regularnie. W tym modelu nie ma pamięci dzielonej, każdy proces ma własną pamięć i nie ma bezpośredniego dostępu do pamięci innego procesu. Model krystaliczny zakłada, że program składa się z N identycznych procesów roboczych (ang.*worker processes*), ponumerowanych od 1...N, plus dodatkowy proces kontrolujący (ang. *controller process*). Procesy robocze pracują niezależnie i komunikują się z procesami tego samego typu, a proces kontrolera wykonywany niezależnie może się komunikować bezpośrednio z dowolnym procesem roboczym. Połączenia między procesami mogą mieć geometrie pierścienia, torusa, siatki czy superkostki. Procesy "workery" identyfikują swoich partnerów na podstawie kierunku, w którym ma miejsce wysyłanie lub odbieranie. Poważnym ograniczeniem jest to, że procesy nie mogą być tworzone podczas działania programu. Raz wybrany układ procesów "workerów" jest zachowany do końca życia programu (patrz Rysunek 1).

Wysyłanie i odbiór są w modelu krystalicznym symetryczne i zaimplementowane jako wbudowana funkcja o nazwie `cr_comm`. Ta funkcja pobiera jako argumenty jeden z ciągów "send" lub "recv" (aby stwierdzić czy dane będą wysyłane czy odbierane), identyfikator kierunku (będący liczbą całkowitą z zakresu 1...D, gdzie D jest liczbą możliwych kierunków przesłania komunikatów, dla struktury na rysunku. D = 3) i inne wyrażenia lub zmienne.

W ten sposób procesy połączone w pierścień mogą próbować przesuwać liczbę całkowitą zgodnie z kierunkiem ruchu wskazówek zegara:

```
integer, constant :: Anticlock = 1
integer, constant :: Clock = 2
.
.
.
integer :: a
.
.
.
call cr_comm("send", Clock, a)
call cr_comm("recv", Anticloc, a)
```

Jednak w momencie zamiany w powyższym przykładzie może dojść do blokady, bo jak widzimy każdy proces próbuje wysłać, gdy żaden nie chce odebrać. Rozwiązanie tego problemu jest następujące. Przesyłanie komunikatu odbywa się w dwóch etapach, najpierw połowa procesów wysyła zanim odbierze, a reszta odbiera zanim wyśle. Podział procesów na dwie równe części może się odbyć na zasadzie sprawdzenia parzystości identyfikatorów tych procesów

```
integer :: i, temp
.
.
.
if (even(id_relself()) then
call cr_comm("send", Clock, i)
call cr_comm("recv", Anticloc, i)
else
call cr_comm("recv", Clock, temp)
call cr_comm("send", Anticloc, i)
  i = temp
end if
```

Zmienna `temp` jest potrzebna po to, aby każdy nieparzysty proces nie stracił wartości, którą chce wysłać przed wysłaniem.

W celu uproszczenia pracy programisty istnieje specjalna procedura `cr_shift`, która pobiera dwa kody kierunku i wysyła wartości z listy zmiennych w kierunku wskazywanym przez pierwszy argument, a odbiera z kierunku wskazywanego przez drugi.

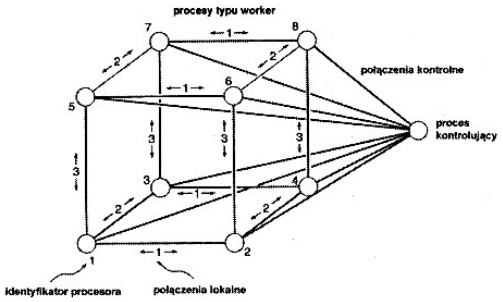
Wielokierunkowe przesunięcie przy użyciu tego polecenia może wyglądać następująco:

```
call cr_shift(Clock, Anticlock, i)
```

Co oznacza, przy założeniach co do topologii połączeń procesów jak we wcześniejszym przykładzie, że wartość `i` ma być odebrana od najbliższego procesu zgodnie z kierunkiem ruchu wskazówek zegara, a poprzednia wartość zmiennej `i` ma być przesłana do kolejnego procesu w przeciwną stronę.

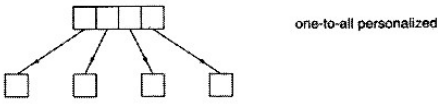
Proces kontrolujący

Do zadań procesu kontrolującego należy sterowanie procesami typu *worker*, wykonywanie operacji wejścia/wyjścia, zapewnienie interfejsu użytkownika, zbieranie wyników obliczeń od procesów roboczych. W najprostszym przypadku modelu krystalicznego, proces kontrolujący może się komunikować przy użyciu funkcji `cr_all`. Tak jak w przypadku `cr_comm`, pierwszy argument decyduje czy dane są wysyłane czy odbierane. Drugi argument jednak nie określa kierunku, ale jest tablicą, w której każdy element odpowia-

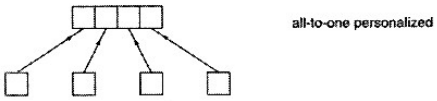


Rysunek 1. Struktura połączeń w modelu krystalicznym z przesyłaniem komunikatów.

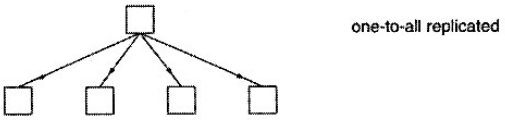
da jednemu procesowi roboczemu. Jeśli dane są wysyłane, proces *p* odbiera odpowiadający mu element *p* tej tablicy i na odwrót, gdy dane są odbierane pole *p* tej tablicy jest zapełniane przez wartość wysłaną przez proces *p*. Procesy krystaliczne współdziałają z procesem kontrolującym jak ze zwykłym procesem, przy



Rysunek 2. Indywidualizowana komunikacja jeden do wszystkich.



Rysunek 2a. Indywidualizowana komunikacja wszyscy do jednego.



Rysunek 3. Komunikacja jeden do wszystkich z powielaniem komunikatu.

użyciu `cr_comm`, z tym że kierunkiem nadawania i odbierania jest kierunek 0. Najczęstszym zastosowaniem `cr_all` jest użycie tej instrukcji do zbierania danych od procesów roboczych lub rozesłanie danych odczytanych z pliku wejściowego.

Instrukcję `cr_all` rozsyłającą dane od kontrolera do procesów "workerów", często nazywa się rozpraszaniem (ang. *scatter*) lub indywidualizowaną komunikacją jeden do wszystkich (ang. *one-to-all personalized communication*), ponieważ każdy proces odbiera komunikat od kontrolera, ale dane wysyłane do każdego odbiorcy są różne (patrz Rysunek 2).

Wariant `cr_all`, w którym dane z procesów "workerów" przesyłane są do procesu kontrolera zwykło się nazywać zbieraniem (ang. *gather*) lub indywidualizowaną komunikacją wszyscy do jednego (ang. *all-to-one personalized communication*). W tym wypadku każdy proces "worker" wysyła pojedynczy komunikat, a te są gromadzone w odpowiednim porządku w tablicy procesu kontrolnego.

Bardzo użytecznym wariantem instrukcji rozpraszającej (*scatter*) jest rozgłaszanie (ang. *broadcast*), służące do komunikacji jeden do wszystkich z powielaniem komunikatu (ang. *one-to-all replicated communication*), gdzie wszystkie procesy odbierają ten sam komunikat (patrz Rysunek 3).

System może implementować instrukcję *broadcast* przy użyciu polecenia `cr_all`, przez powielenie *p*-razy wiadomości przez proces kontrolera do wszystkich elementów tablicy i wywołanie funkcji `cr_all`, której ta tablica będzie drugi argumentem. W większości systemów z przesyłaniem komunikatów dostarczane są jednak oddzielne, specjalnie przeznaczone do realizacji tego typu komunikacji funkcje. W Fortanie-K (CR) służy do tego funkcja `cr_bcast`. Pobiera ona jeden argument i wysyła jego kopie do wszystkich procesów "workerów".

Broadcast jest często implementowany z użyciem drzew połączeń (ang. *spanning trees*), których rozgałęzione ścieżki łączą wszystkie węzły z jednym arbitralnym węzłem korzeniem (patrz Rysunek 4).

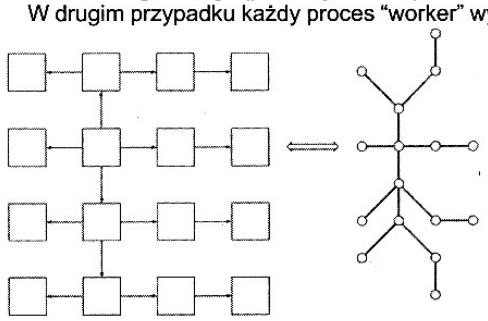
O ile odwrotnością *scatter* jest *gather*, to odwrotnością *broadcast* jest *reduction* (redukcja). W Fortanie-K (CR), redukcja wywoływana jest, kiedy kontroler i wszystkie procesy typu "worker" wywołają funkcję `cr_merge`. Jeśli mamy daną fukcję *f* i jeśli procesy "workery" i proces kontrolujący wykonają:

```
call cr_merge(f, a)
```

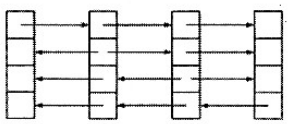
to kopia zmiennej `a` przechowywanej w procesie kontrolującym przybierze wartość będącą wynikiem stosowania funkcji *f* na stopniowych kombinacjach wartości procesów "workerów". `cr_merge` nie kopiuje automatycznie wyników do procesów "workerów", w tym celu musi być użyta jawnie funkcja `cr_bcast`. Nazwą funkcji może być dowolna funkcja, włącznie z funkcją zdefiniowaną przez użytkownika, a jest to możliwe dzięki temu, że każdy proces "worker" wykonuje wywołanie tej funkcji niezależnie.

Większość systemów krystalicznych dostarcza wielu funkcji pozwalających "workerom" na wymianę danych między sobą. Typowa jest obsługa komunikacji typu all-to-all replicated communication i all-to-all personalized communication. W pierwszym przypadku, każdy proces "worker" rozsyła jedną wiadomość do każdego innego (patrz Rysunek 5).

W drugim przypadku każdy proces "worker" wy-



Rysunek 4. Drzewo połączeń.



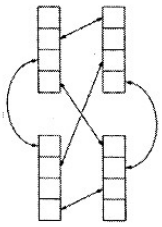
Rysunek 5. Komunikacja wszyscy do wszystkich z powielaniem komunikatu.

syła różny komunikat do każdego innego procesu typu "worker" (patrz Rysunek 6).

Podsumowując, niektóre systemy pozwalają na komunikację każdy z każdym, gdzie procesy robocze mogą komunikować się między sobą. Takie funkcje są podstawą systemów opierających się na przesyłaniu komunikatów.

Model proceduralnego przesyłania komunikatów

Większość programów równoległych składa się w rzeczywistości z kilku typów procesów, pełniących podobne funkcje. Owe grupy procesów (ang. *process groups*) tworzą zbiory usług wspólnie realizujących obliczenia na rzecz pozostałych procesów. Z tego powodu, systemy przesyłania komunikatów umożliwiają nadanie procesom etykiet oraz stworzenie ich grup. Zaletą takiego podejścia jest separacja kwestii tożsamości procesów i konfiguracji programu. Nazwa procesu nie jest absolutnym identyfikatorem całkowitym czy parą proces-procesor, lecz etykietą składającą się z nazwy grupy i indeksu procesu w tej grupie. Z drugiej strony konfiguracja pozwala na planowanie przydziału grupom dostępnych procesorów.



Rysunek 6. Indywidualizowana komunikacja wszyscy do wszystkich.

Istnieją dwa sposoby zarządzania grupami procesów. W pierwszym każdy proces jest tworzony jako członek istniejącej grupy procesów. Programista musi w pierwszej kolejności określić grupy, które będą stworzone oraz ilość procesów należących do każdej z nich. Parametry poszczególnych grup zawierają informacje, takie jak liczba i typ portów posiadanych przez jej członków oraz na przykład typ procesora, na którym grupa może zostać wykonana. Możliwe jest umieszczanie tych parametrów w bibliotekach oraz ich natychmiastowe uzyskanie w razie potrzeby. Dodatkową zaletą jest to, że konfiguracja jednocześnie wyszczególnia, jakie procesy zostaną stworzone i jaką mają rolę. Jakkolwiek opisanie złożonych konfiguracji wymaga bardzo złożonej notacji.

W drugim podejściu do zagadnienia zarządzania grupami procesów grupy nie istnieją przed uruchomieniem programu i, co oczywiste, procesy nie są wewnętrznymi członkami jakiejkolwiek z nich. Jakkolwiek zaraz po rozpoczęciu wykonywania programu procesy mogą wyrazić chęć wstąpienia do jednej lub kilku grup. Wówczas system sprawdza czy wskazana grupa (lub grupy) już istnieje. Jeżeli tak, procesowi nadawany jest unikalny identyfikator wewnętrzny grupy; w przeciwnym wypadku grupa zostaje stworzona, a proces staje się jej pierwszym członkiem. Główną zaletą tworzenia grup przez wykonujące się procesy jest to, że używane przez nie biblioteki mogą tworzyć grupy na własny użytek bez jawnej wiedzy programisty. Dlatego bardzo użyteczna jest możliwość przynależności jednego procesu do wielu grup. Inną korzyścią jest możliwość przynależności procesów różnych typów do jednej grupy. Umożliwia to znacznie prostsze programowanie systemów heterogenicznych, ponieważ procesy uruchomione na różnych typach procesorów nie wymagają odmiennych metod zarządzania grupami.

Największą korzyścią grupowania procesów jest możliwość realizacji rozgłaszania typu multicast, tj. szerokiego rozgłaszania komunikatów broadcast do wielu, lecz niekoniecznie wszystkich, procesów. Wiele aplikacji stosuje broadcast i multicast niemal tak często jak komunikację typu punkt-punkt. W heterogenicznych sieciach procesów broadcast musi zawierać pewne mechanizmy selekcji zbioru (lub zbiorów) procesów odbierających, aby nadawca mógł łatwo przesłać komunikat do grupy składającej się z procesów zdolnych do jego interpretacji.

Ostatnią kwestią, która musi zostać uwzględniona w modelu przesyłania komunikatów jest podanie procesom informacji o stanie systemu. W systemach bazujących na mechanizmie RTI (ang. *runtime initialization*) procesy wywołują funkcje umożliwiające określenie takich informacji jak własny identyfikator wewnętrzny grupy, liczba procesów w grupie itd. Funkcje te są zazwyczaj wywoływane raz podczas fazy inicjalizacji, a zwracane przez nie informacje zostają umieszczone w zmiennych lokalnych procesu. Dla odmiany, wykorzystując mechanizm SPB (ang. *static parameter binding*), system automatycznie dodaje pewne zmienne do każdego procesu i bezpośrednio w nich przechowuje informacje konfiguracyjne. Główną płynącą stąd korzyścią jest eliminacja konieczności zamieszczania przez programistę szeregu wywołań funkcji w kodzie źródłowym każdego procesu. Niestety, mechanizm ten wymaga wsparcia ze strony kompilatora. Ponieważ jednak większość systemów przesyłania komunikatów jest implementowanych jako biblioteki języków sekwencyjnych, RTI jest bardzo często wykorzystywany. RTI również lepiej realizuje kwestię dynamicznego tworzenia grup.

Inicjalizacja

Pierwszym krokiem w systemach tworzących najpierw procesy później grupy jest przypisanie procesów do grup. Ponieważ procesy wykonują się niezależnie i asynchronicznie realizacja tego kroku wymaga przeprowadzenia kilku etapów pośrednich. Przede wszystkim proces wstępuje do jednej lub kilku grup, po czym ustawia synchronizację barierową w poprzek pozostałych procesów by upewnić się, że system dotarł do stabilnego punktu, w którym wszelkie czynności

dołączania procesu do grupy zostały zakończone. Następnie zostają stworzone i przypisane każdemu fizycznemu procesorowi tablice opisujące, gdzie znajdują się procesy oraz do jakich grup każdy z nich należy. Od tego momentu procesy mogą kontynuować wykonanie, sprawdzać konfigurację systemu oraz nadawać i odbierać komunikaty. Takie podejście zapobiega powstaniu sytuacji wyścigu warunków, chociaż uniemożliwia naprawdę dynamiczne tworzenie procesów. Funkcja:

```
integer function mp_join(grp_name, desired_id)
type(string)        :: grp_name
integer             :: desired_id
```

czyni proces członkiem grupy `group_name` i zwraca jego wewnętrzny identyfikator. Jeżeli grupa nie istnieje wówczas wywołanie `mp_join()` tworzy nową grupę o nazwie `group_name`. Dodatnia wartość typu integer parametru `desired_id` wymusza nadanie procesowi określonego identyfikatora wewnętrznego natomiast przekazanie `MP_Any_Id` zezwala na nadanie mu dowolnej wartości. Ustawienie synchronizacji barierowej realizowane jest za pomocą funkcji `mp_synch()`, której działanie zostało opisane w poprzednim rozdziale. Określenie wielkości poszczególnych grup umożliwia funkcja:

```
integer function mp_grp_size(grp_name)
type(string)        :: grp_name
```

Każdy proces może ustalić czy jest członkiem grupy wywołaniem funkcji:

```
logical function mp_grp_mem(grp_name)
type(string)        :: grp_name
```

oraz uzyskać swój wewnętrzny identyfikator:

```
integer function mp_grp_id(grp_name)
type(string)        :: grp_name
```

Do sprawdzenia czy grupa zawiera proces o wskazanym identyfikatorze służy funkcja:

```
logical function mp_grp_mem_live(grp_name,
member_id)
type(string)        :: grp_name
integer             :: member_id
```

Komunikacja

Najprostszą funkcją komunikacji w Fortran-K (MP), która wysyła komunikat od jednego procesu do innego jest:

```
procedure mp_send(src_grp, dst_grp, dst_id,
msg_type, values)
type(string) :: src_grp   !identyfikuje nadawcę
                          ! komunikatu umożliwiając
                          ! procesowi odbierającemu
                          ! odesłanie odpowiedzi
type(string) :: dst_grp   !docelowa grupa
                          ! przesyłanego komunikatu
integer :: dst_id         !docelowy proces przesyłanego
                          ! komunikatu, w szczególności:
                          ! MP_All_Id oznacza broadcast do
                          ! wszystkich członków grupy
type(string) :: msg_type  !wymagany do rozróżniania
                          ! komunikatów różnych typów
```

W praktyce często przesyłane są komunikaty nie zawierające danych, a ich jedynym zadaniem jest sygnalizowanie, na przykład, stanu gotowości czy zakończenia wykonywania.

Odbiór komunikatów następuje w dwóch etapach. W pierwszym jedna lub więcej zmiennych zostaje przesłana systemowi w celu jej wypełnienia nadchodzącymi komunikatami. Funkcją, która to realizuje jest:

```
type(msginfo) function mp_pend(dst_grp,
msg_type, variables)
type(string) :: dst_grp   ! grupa do której należy
                          ! proces wywołujący
type(string) :: msg_type  ! określa jaki typ
                          ! komunikatów odbiorca jest
                          ! zdolny zaakceptować
```

Funkcja alokuje i zwraca nową strukturę predefiniowanego typu `msginfo` zawierającą następujące pola:

- dst_grp łańcuch znakowy identyfikujący docelową grupę komunikatu
- src_grp łańcuch znakowy identyfikujący źródłową grupę komunikatu
- src_id identyfikator nadawcy komunikatu
- msg_type łańcuch znakowy identyfikujący typ komunikatu

`mp_pend()` jest operacją nie blokującą. Aby określić czy i kiedy w buforze odbiorczym został umieszczony komunikat, proces może poddać strukturę `msginfo` zwracaną przez `mp_pend()` działaniu funkcji `mp_test()`:

```
type(msginfo) function mp_test(block, msg)

logical :: block !.true. - blokuje proces do chwili
                 ! wypełnienia
                 ! bufora wskazywanego przez msg
                 ! .false. - mp_test() zwraca msg w
                 ! przypadku zakończonej
                 ! transakcji; w przeciwnym
                 ! wypadku zwraca NULL
type(msginfo) :: msg !wskazuje bufor odbiorczy, w
                     ! szczególności:
                     ! MP_Any_Id wskazuje, że
                     ! wywołujący chce wiedzieć,
                     ! czy nadszedł jakikolwiek
                     ! zaległy komunikat
                     ! MP_All_Id wskazuje, że
                     ! wywołujący chce wiedzieć,
                     ! czy nadeszły wszystkie
                     ! zaległe komunikaty
```

Innym sposobem odbioru komunikatu jest użycie:

Tabela 1. Przykłady zgodności krotek.

integer :: i=3, j=4, X(3), Y(2, 2), Z(4)
logical :: b

Antykrotka	Krotka	Zgodność	Komentarz
{2}	{2}	tak	wartości zgodne
{2}	{3}	nie	wartości niezgodne
{i}	{3}	tak	typy i wartości zgodne
{i}	{2}	nie	typy zgodne, ale wartości niezgodne
{?i}	{2}	tak	typy zgodne (i <- 2)
{?i, ?b}	{2, .true.}	tak	typy zgodne
{?b, ?i}	{2, .true.}	nie	typy niezgodnym porządku
{X}	{(/1, 2, 3/)}	błąd	niewłaściwe porównanie
{?X}	{(/1, 2, 3/)}	tak	typ i rozmiar zgodne
{?X}	{(/1, 2, 3, 4/)}	nie	nieudane porównanie rozmiaru
{?Y}	{Z}	nie	nieudane porównanie wymiaru

```
procedure mp_recv(dst_grp, msg_type, variables)
type(string)        :: dst_grp
type(string)        :: msg_type
```

`mp_recv()` działa podobnie jak `mp_pend()` wewnątrz `mp_test()` blokując proces wywołujący do chwili otrzymania komunikatu. Nie istnieje tu jednakże możliwość określenia nadawcy komunikatu.

Konfiguracja

Konieczność ręcznej konfiguracji programu jest jednocześnie nużąca i stanowi potencjalne źródło błędów. Jednakże, w sytuacji braku wsparcia modelu przesyłania komunikatów przez kompilator nie pozostaje programiście jakakolwiek alternatywa.

Typowym sposobem konfiguracji programów jest pisanie przez użytkowników plików konfiguracyjnych, które tworzą i rozmieszczają opisane w nich procesy. Wadą tej metody jest to, że każdy proces musi być oddzielnym programem. Również dokonanie złożonej konfiguracji wymaga definiowania w plikach konfiguracyjnych wyrażeń warunkowych, pętli itp. Dlatego, alternatywnym sposobem jest dostarczenie bibliotek funkcji, które umożliwiają programistom opisanie struktury programu równoległego w sekwencyjnych językach programowania. W Fortran-K (MP) konfiguracje realizuje funkcja:

```
procedure mp_place(arc, pid, exec, args)

type(string) :: arch        ! typ procesora, na którym
                            ! proces zostanie
                            ! uruchomiony
type(string) :: pid         ! dokładnie który z proce-
                            ! sorów zostanie użyty
integer :: exec             ! nazwa procesu
                            ! wykonywalnego
type(string) :: args(:)     ! argumenty przekazywa-
                            ! ne do procesu
```

Komunikacja generowana (Generative Communication)

Programiści potrzebują modelu łączącego w sobie prostotę zmiennych dzielonych i skalowalność systemów z przesyłaniem komunikatów. Wymagania te spełnia model z komunikacją generowaną. Procesy w systemach z komunikacją generowaną nie dzielą zmiennych, natomiast wszystkie mają dostęp do wspólnej dzielonej przestrzeni. Nazwa komunikacja generowana bierze się od tego, że procesy nie przesyłają komunikatów, ale raczej generują pasywne struktury danych we współdzielonej przestrzeni, które inne procesy mogą czytać, modyfikować i likwidować. Jedną z najlepszych implementacji komunikacji generowanej jest Linda, stworzona przez Davida Gelerntera i Nicholasa Carriero w Yale University. Linda nie jest nowym językiem, ale małym zbiorem operacji, które mogą być dodane do dowolnego języka w celu zrównoleglenia operacji. Jest to kolejna mocna strona modelu z komunikacją generowaną, umożliwiającą w prosty sposób zrównoleglenie istniejących programów, stworzonych w konkretnych językach, przez programistów nie koniecznie znających systemy do programowania rozproszonego.

Podstawą modelu jest dzielona, asocjacyjna pamięć i operacje, którymi program może operować na niej. System z komunikacją generowaną może zawierać dowolną liczbę procesów, które nie maja bezpośrednich połączeń między sobą i nie mogą wzajemnie się modyfikować, mogą natomiast komunikować się przez pamięć dzieloną zwaną przestrzenią krotek (ang. *tuple space*).

Krotki i przestrzeń krotek

Komunikacja generowana wprowadza tylko jeden nowy rodzaj obiektów, zwany krotką (ang. *tuple*). Krotka jest nieuporządkowanym, skończonym zbiorem pól różnych typów. Każde pole może zawierać konkretne

wartości, mówimy wtedy, że jest rzeczywiste (ang. *actual*), lub może być nieokreślone, mówimy wtedy o polu formalnym. Pola formalne są używane do tworzenia wzorów dobierania.

Przykładowe krotki zawierające wyłącznie pola rzeczywiste mogą wyglądać następująco:

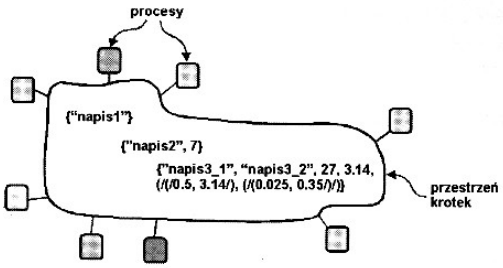
```
{"napis1"}
{"napis2", 7)
{"napis3_1", "napis3_2", 27, 3.14,
(/(/0.5,3.14/),(/(/0.025,0.35/)/)}
```

Pierwsza z krotek zawiera ciąg znaków "napis1", druga ciąg "napis2" i liczbę całkowitą 7, a trzecia zawiera dwa ciągi, liczbę całkowitą 27, rzeczywistą 3.14 i tablicę liczb rzeczywistych o rozmiarze 2 na 2 pola. Krotki istnieją jedynie w przestrzeni krotek, która nie przechowuje nic oprócz nich, gdyż same procesy znajdują się na zewnątrz, co pokazuje Rysunek 7.

Procesy dodają krotki do przestrzeni krotek używając operacji `out`. Operacja ta pobiera jako argumenty jedno lub więcej wyrażeń, `out` oblicza te wyrażenia w nieokreślonym porządku i tworzy krotkę zawierającą wyniki tych obliczeń. Przykładowy kod w języku Fortran-K (GC) tworzący, wcześniej przytoczone jako przykłady, krotki, może wyglądać tak:

```
type(string)        :: label_1="napis1"
integer, constant   :: i=7
type(string), auto  :: j
real                :: block(2, 2)
.
.
.
label_2="napis2"
j=27
block(1, 1)=0.5; block(1, 2)=3.14
block(2, 1)=0.025; block(2, 2)=0.35
.
.
.
out label_1
out label_2, i
out "napis3_1", "napis3_2", j, 3.14, block
```

Operacje zapisywania do pamięci nie wystarczą do stworzenia programu, dlatego model z komunikacją generowaną przewiduje dwie operacje do czytania i usuwania krotek. Pierwsza z nich nazywa się `rd`. Tak jak `out`, `rd` bierze jedno lub więcej wyrażeń jako argumenty, ale zamiast dodawać krotkę do przestrzeni krotek, `rd` blokuje proces wywołujący dopóki nie zostanie dopasowana krotka w przestrzeni krotek zgodnie z argumentami operacji. Wywołanie operacji `rd` może wyglądać nastepująco:



Rysunek 7. Przestrzeń krotek.

```
rd "napis1"
```

Kiedy przestrzeń krotek zawiera tylko trzy wyżej przedstawione w przykładach krotki, możliwe jest wywołanie:

```
rd "napis4"
```

Przestrzeń krotek nie zwiera krotki "napis4", więc krotka nie mogła być dopasowana. Proces jest zablokowywany dopóki inny proces nie umieści odpowiedniej krotki w przestrzeni.

Drugą operacją czytającą krotki jest operacja `in`, która działa jak `rd`, z tą wóżnicą, że gdy krotka zostanie dopasowana następuje usunięcie jej z przestrzeni krotek. Operacja `in` na konkretnej krotce może być wykonywana tylko raz, w odróżnieniu od operacji `rd`, która na jednej krotce może być wykonywana wielokrotnie. Procesy mogą się synchronizować używając krotek w podobny sposób jak używa się semaforów.

Problem z instrukcjami `in` i `rd` polega na tym, że proces musi dokładnie wiedzieć, co chce pobrać z przestrzeni krotek zanim instrukcja pobierania krotki zostanie wykonana. Jest to niewygodne założenie, gdyż program musi znać poszukiwaną wartość zanim odczyta ją z przestrzeni krotek, dlatego w komunikacji generowanej istnieje pojęcie antykrotki (ang. *anti-tuples*). Antykrotka jest specyficznym rodzajem krotki, w której część pól jest polami formalnymi, to znaczy mającymi określony typ, ale nie posiadającymi wartości. Pola formalne w antykrotce oznacza się znakiem zapytania postawionym przed zmienną lub typem, jak na przykład:

```
integer          :: i, b(4)
type(string)     :: s
.
.
.
in "napis1", ?i   ! drugie pole jest formalne
rd ?s, 3, ?b      ! pierwsze i trzecie pole są formalne
```

Jeśli proces używa antykrotki do operacji czytania, użyta zostaje dowolna krotka zgodna z antykrotką. Krotka i antykrotka są zgodne jedynie gdy:

- Mają tą samą ilość pól
- Typy odpowiednich pól są takie same
- Wartości odpowiednich pól rzeczywistych są takie same

Zasady porównywania krotek wyjaśnia na przykładach Tabela 1.

Jeśli operacja `rd` lub `in` nie może znaleźć pasującej krotki do antykrotki, proces wykonujący operację jest blokowany dopóki operacja dopasowania nie jest możliwa.

Krotki nie są przechowywane tak jak inne struktury danych, wewnątrz procesów. Są one natomiast przechowywane w odseparowanej przestrzeni krotek, która ma cztery ważne własności: jest dzielona, wszystkie procesy pracują z tą samą przestrzenią krotek i każdy proces ma do niej dostęp, nie jest zbiorem, ale "workiem" (ang. *bag*), w którym może się znajdować wiele identycznych krotek, jest asocjacyjna, krotki są usuwane z przestrzeni krotek przy wzorcach porównawczych, według zasad dopasowywania przedstawionych wcześniej, jest anonimowa, gdy raz umieścimy krotkę, system nie potrafi stwierdzić, kto i kiedy ją stworzył. Program może umieszczać informacje o pochodzeniu krotki w niej samej, ale za zarządzanie tymi informacjami jest odpowiedzialny programista.

Tworzenie procesów

Ostatnią z podstawowych operacji w modelu z komunikacją generowaną jest operacja `eval`. Ma ona ten sam końcowy rezultat co `out`, umieszcza krotkę w przestrzeni krotek, ale osiąga to w okrężną drogą. Zamiast obliczenia wartości każdego pola i stworzenia krotki w celu przechowania wyników, `eval` tworzy aktywną krotkę (ang. *active tuple*), która tworzy nowy proces w celu obliczenia każdego pola krotki. Każdy z tych procesów działa tak długo, dopóki nie obliczy wszystkich pól w aktywnej krotce. Gdy zostaną wypełnione wszystkie pola, krotka jest przekształcana w zwykłą krotkę, zwaną też pasywną, taką jaką tworzy się operacją `out`. Przykładowe utworzenie krotki aktywnej może wyglądać następująco:

```
eval "przykład", x, f(x), g(x, y, z)
```

gdzie `f(x)` i `g(x, y, z)` są wcześniej zdefiniowanymi funkcjami. Stworzony w ten sposób proces jest równy swojemu stwórcy, nie ma żadnych połączeń między nim a jego rodzicem i może wykonywać operacje na krotkach włącznie z operacją `eval`.

Reasumując

Wszystkie przedstawione wcześniej modele obliczeń posiadają zarówno wady jak i zalety, które rzutują na późniejszą użyteczność narzędzi do przetwarzania rozproszonego na nich się opierających.

Komunikacja bazująca na kanałach nabrała wiele rozgłosu w połowie lat osiemdziesiątych, kiedy pojawił się po raz pierwszy Occam i transputery. Doświadczenia wielu programistów wskazywały, że model ten nie nadaje się najlepiej do dużych aplikacji. Pojawiło się wówczas wiele krytyk tego modelu, które jednak bardziej odnosiły się do jego implementacji Occamu, niż do samej idei kanałów. Niektóre z nich były dosyć poważne. Najczęściej padającą krytyką kanałów jest ich koszt. Powiedzmy, że komunikat M wysłany przez proces A, przesyłany jest przez proces B do procesu C. Jeśli wszystkie procesy A, B i C działają na tym samym procesorze, komunikat M może być dwa lub więcej razy kopiowany, nawet jeżeli aktualnie nie musi być kopiowany w ogóle. Jeśli musi być użytych wiele buforów, przełączników i filtrów, to kosz wielokrotnego kopiowania staje się nie do zaakceptowania. Kuszące w tych okolicznościach wydaje się przekazywanie przez procesy wskaźników na komunikaty, zamiast samych komunikatów, ale nigdy nie wiadomo czy procesy dzielą tą samą przestrzeń adresową, czy są uruchomione na dwóch zupełnie odmiennych komputerach.

Inną kosztowną częścią modelu CSP jest konstrukcja `alt`. W Occamie, czas wykonania instrukcji `alt` jest proporcjonalny to ilości rozgałęzień w tej instrukcji. Zawsze tylko jedno z tych rozgałęzień zostaje obsłużone, tak więc, im bardziej rozbudowana konstrukcja `alt`, tym mniejsza jest jej efektywność Częste stosowanie rozbudowanych konstrukcji `alt`, nie tylko obniża efektywność programu, ale też wpływa niekorzystnie na czytelność programu.

Jedną z największych słabości modelu CSP jest ilość potrzebnych instalacji (połączeń kanałami między procesami). W większości wielkich systemów programista potrzebuje wysłać komunikaty do różnych odbiorców. Na pierwszy rzut oka problem nie wydaje się skomplikowany. Jeśli mamy do dyspozycji bezpośrednią komunikację przy użyciu kanałów typu punkt-punkt możemy połączyć każdy proces z każdym. W praktyce jednak taka ilość połączeń dramatycznie obciąża system i znacznie wpływa na prędkość komunikacji.

Podobnie jak model CSP, model krystaliczny też ma wiele zarówno mocnych jak i słabych stron. Komunikacja w modelu krystalicznym jest blokująca i synchroniczna. Pierwszy termin oznacza, że proces nie może kontynuować swojej pracy dopóki komunikacja nie zostanie zakończona. Drugi mówi, że komunikacja ma miejsce tylko wtedy, gdy obaj uczestnicy komunikacji są gotowi. Przeciwieństwem tego jest komunikacja asynchroniczna, gdzie proces może wysłać dane, bez upewniania się, że ktoś je odebrał. Przy komunikacji asynchronicznej istnieje możliwość zwiększenia wydajności programu przez nakładanie się obliczeń, proces nie musi czekać na odebranie wyników swojej pracy, może je wysłać i podjąć nową pracę. W tym sensie model krystaliczny opierający się na blokującej, synchronicznej komunikacji nie daje możliwości tworzenia programów opartych na tej zasadzie. Zezwolenie na komunikację asynchroniczną nie jest rozwiązaniem, wiążą się z tym pewne niebezpieczeństwa. Zobaczmy co się stanie jeśli komunikat przyjdzie zanim odbiorca jest gotów do odebrania. Komunikat może być tracony lub system musi go zbuforować. Nawet, gdy rozwiąże się ten problem, program staje się dużo bardziej skomplikowany, program musi sprawdzić czy nadeszło potwierdzenie odbioru takiego komunikatu i zająć się ponownym wysłaniem, jeśli komunikat zaginął. Kolejnym sprawą jest bezpieczeństwo. Rzeczywisty system dostarcza skończoną liczbę buforów, które raz przepełnione, stawiają programistę przed problemem ponownego rozesłania komunikatów. Asynchroniczna komunikacja wydaje się bardzo dobra na poziomie implementacji systemu, na poziomie użytkownika płynące z niej korzyści wydają się być dyskusyjne.

Inną kwestię stanowią operacje nie blokujące, które mogą być zaimplementowane w bezpieczny spo-

sób. Jeśli proces inicjuje nie blokujące wysłanie, możliwe jest jego przejście do dalszych obliczeń w czasie, gdy system czeka na bufor, który będzie dostarczony przez proces odbierający. Kiedy się to stanie dane komunikatu są kopiowane do bufora i flaga jest ustawiana tak aby wskazywała, że wysyłający proces może teraz już nadpisać zmienne, których wartości zostały wysłane. Nie blokujący odbiór jest podobny, proces daje systemowi jedną lub więcej zmiennych jako bufor i znacznik, którego wartość jest później sprawdzana przez proces w celu stwierdzenia czy dane zostały już zapisane do bufora. To pozwala odbiorcy kontynuować dalszą pracę, gdy czeka na komunikat.

W modelu CSP proces może użyć instrukcji `alt` do odebrania komunikatu z wyspecyfikowanego kanału wejściowego. W innych modelach wybór komunikatu może odbywać się przy użyciu jednego z trzech lub wszystkich poniżej opisanych mechanizmów:

- **Wybór według źródła** (ang. select by source). Jeśli każdy komunikat jest dostarczony z identyfikatorem źródła, wtedy odbiorca może wyspecyfikować, z jakiego źródła chciałby odebrać komunikat lub nawet może określić cały podzbiór z możliwych źródeł, z których akceptuje komunikaty. Wybieranie według źródła pomaga rozwiązać problem kontynuowania dialogu (ang. conversational continity). Przykładowo, gdy proces serwera p_s raz odbierze komunikat od procesu klienta p_c może chcieć odbierać tylko komunikaty od procesu p_c dopóki transakcja nie zostanie zakończona.
- **Wybór według nadejścia** (ang. select by arrival). Jeśli proces ma dostępnych kilka niezależnych połączeń do przesyłania komunikatów, tak jak przy wielu kanałach wejściowych w modelu CSP, odbiorca może wyspecyfikować połączenie lub połączenia, przy użyciu których chce odbierać komunikaty. Takie połączenia są często nazywane potami komunikatów (ang. message ports).
- **Wybór według typu** (ang. select by type). Jeśli komunikaty są różnego typu, wtedy procesy mogą wybierać do odebrania komunikaty określonego typu lub typów.

Każdy z tych schematów potrzebuje pewnego mechanizmu nadawania etykiet, dla procesów, portów lub oznaczania komunikatów. Niektóre systemy z przesyłaniem komunikatów etykietują procesy tworzenia par z ID procesora i ID lokalnego procesu. Każdy port procesu jest numerowany od jedynki. Podobnie, identyfikatory komunikatów są wartościami arbitralnie przydzielanymi przez użytkownika podczas wysyłania i odbierania komunikatu.

Problem z takim prostym rodzajem nazewnictwa polega na tym, że utrudnia on tworzenie dużych programów, składających się kilku mniejszych korzystających z bibliotek. Chodzi o to, że jeśli wybrany system bazuje na ID procesów, każdy proces, w każdej konfiguracji programu musi znać ID innych procesów. Jednakże, nie ma tam wbudowanych połączeń między logicznie tożsamymi procesami a fizycznymi ID nadanymi przez program. W rezultacie, użytkownik piszący program musi rozwiązywać problem związany ze zmianą konfiguracji poszczególnych procesów. Jeśli programista popełni błąd na tym poziomie, może być on bardzo trudny do znalezienia.

Podobnie, wybór komunikatu przez jego typ lub port, na którym się pojawił, jest niebezpieczny jeśli program zbudowany jest z połączenia pisanych niezależnie bibliotek. Przypuśćmy, że autorzy dwóch bibliotek używają wartości 1...T jako znaczników komunikatów. Jeśli proces używa obydwu bibliotek i odbiera komunikaty, które są z zakresu 1...T, nie ma żadnej możliwości odróżnienia, która biblioteka nadała komunikat. Niezależnie tworzone biblioteki mogą również kolidować próbując używać tych samych portów dla komunikatów. Na przykład, jedna biblioteka może spodziewać się wiadomości od innego procesu na porcie 3, gdy druga może używać portu 3 do odbierania kopii zapotrzebowań systemu plików.

Problem konfiguracji programu jest dobrze rozwiązany w modelu z proceduralnym przesyłaniem komunikatów, gdzie procesy nie są anonimowe, mają nadane etykiety utworzone z grupy do której przynależy proces i indeksy procesu w grupie. Podział procesów na grupy w tym modelu pozwala na łatwe rekonfigurowanie i rozbudowę systemu, ponadto umożliwia przypisywanie grup procesów świadczących jakieś konkretne usługi najbardziej odpowiednim architekturom procesorów. Istnienie podziału na grupy czyni możliwym przesyłanie komunikatów nie tylko typu broadcast, ale i multicast. Te cechy w połączeniu z kolejną cenną zaletą tego modelu, komunikacją nie blokującą, czynią narzędzia oparte na tym modelu niezwykle atrakcyjnymi pod względem wydajności obliczeń. Należy też pamiętać, że systemy oparte na tym modelu, takie jak PVM czy MPI oferują dużo większy zakres funkcji i możliwości niż zostało to zaprezentowane na podstawie akademickiego języka Fortran-K (MP).

Jedną z wad narzędzi opartych na komunikacji z proceduralnym przesyłaniem komunikatów jest ich złożoność. Tej wady są pozbawione systemy zbudowane w oparciu o komunikację generowaną. Prostota i skalowalność, czyli zdolność adaptowania się systemu do wzrastających zapotrzebowań, to ich podstawowe zalety. Realizacja sytemu rozproszonego odbywa się przez dodanie kilku prostych funkcji operujących na przestrzeni krotek, które mogą być dodane do dowolnego języka programowania. Przy użyciu tych prostych operacji można zaimplementować inne mechanizmy, poczynając od funkcji synchronizujących, aż do komunikacji z przesyłaniem komunikatów. Przestrzeń krotek jest traktowana jak worek, w którym przechowywane są krotki. Nie mamy wpływu na to gdzie przechowywane są te dane, ani przez które procesory wykonywane są poszczególne obliczenia.

Jak widzimy wszystkie modele mają swoje wady i zalety, ale najbardziej obiecującymi wydają się być model z proceduralnym przesyłaniem komunikatów, ze względu na ogrom potencjalnych możliwości oraz model z komunikacją generowaną, z powodu swojej prostoty i dużych możliwości.

W artykule w dużej mierze zostały wykorzystane materiały z książki Gregory V. Wilson "Practical parallel programming" The MIT Press Cambridge, Massachusetts London, England 1995. ■