PVM — Parallel Virtual Machine

Sławomir 'civic' Białek
<slawek@bialek.org>

22 września 2002 roku

Czym jest PVM?

PVM – Parallel Virtual Machine jest rozwiązaniem pozwalającym wykorzystać heterogeniczny kluster komputerów w funkcji pojedyńczego komputera równoległego.

Projekt PVM rozpoczął się w 1989 roku w Oak Ridge National Laboratory. Stworzona tam wersja PVM 1.0 była jednak wykorzystywana tylko na ich wyłączne potrzeby. Wersja PVM 2 została napisana na University of Tennessee i udostępniona w Marcu 1991 roku. PVM zaczęto wykorzystywać do rozwiązywania wielu naukowych problemów wymagających większej mocy obliczeniowej. W lutym 1993 roku ujrzała światło dzienne po przepisaniu większości kodu wersja PVM 3, rozwijana i używana do dziś (aktualne wersje to 3.4.x).

Programowanie współbieżne jest jedną z kluczowych technologii dla rozwiązywania dzisiejszych zadań wymgających dużych mocy obliczeniowych. Jednakże koszty potężnych maszyn MPP (Massively Parallel Processors) często stwarzają równie potężną barierę. I tutaj z pomocą przychodzą rozwiązania takiej jak PVM.

PVM pozwala zamienić heterogeniczny kluster komputerów, od najprostszych klasy PC pracujących pod systemami operacyjnymi Windows, Linux, *BSD poprzez mniejsze i większe serwery Sun, IBM, HP, DEC i SGI do dużych komputerów równoległych w wielki wirtualny komputer równoległy. Skomplikowane problemy obliczeniowe mogą być dzięki temu rozwiązywane przy użyciu połączonej mocy obliczeniowej wielu komputerów.

System PVM udostępnia użytkownikowi możliwość automatycznego uruchamiania procesów na maszynie wirtualnej, funkcji do ich komunikacji i synchornizacji. PVM w sposób przezroczysty dla programisty zajmuje się przekazywaniem komunikatów, konwersją formatów danych oraz zarządzaniem zadaniami.

Na podkreślenie zasługuje fakt iż PVM jest oprogramowaniem dostępnym za darmo. Tak więc nie tylko pozwala obniżyć koszty sprzętu, ale też sam jako oprogramowanie nic nie kosztuje. Choć oczywiście istnieją firmy, które świadczą komercyjny support lub zajmują się tworzeniem komercyjnego oprogramowania wykorzystującego PVM.

Architektura PVM

PVM jest pakietem oprogramowania, który składa się głównie z dwóch elementów:

- demona – uruchamianego na komputerach, które stają się węzłami sieci tworzącej wirtualny komputer równoległy

- biblioteki funkcji zawierającej interfejs dla programów mających działać z PVM

Biblioteka funkcji wraz z demonem udostępniają programowi mechanizmy przekazywania komunikatów, uruchamiania i synchronizacji zadań oraz możliwość modyfikacji maszyny wirtualnej PVM. Wykorzystanie możliwości PVM dostępnie jest głównie z języków C, C++ i Fortran bowiem to dla nich zostały przygotowane biblioteki. Aczkolwiek istnieją również pakiety pozwalające wykorzystać PVM również w Javie, Perlu oraz Tcl/Tk, a także innych językach i technologiach.

Model programowania PVM zakłada, że aplikacja składa się z szeregu zadań. Umożliwia to zastosowanie PVM zarówno poprzez dzielenie zadania na funkcjonalne podzadania, a także poprzez podział zadania na wiele równoległych zadań, które zajmują się fragmentem danych wejściowych lub obie te metody równocześnie.

Z wyjątkiem wywoływania specyficznych dla PVM funkcji dotyczących np. uruchamiania zadań czy przekazywania komunikatów i synchronizacji programy pisze się w sposób nie wiele odbiegający od programowania bez PVM. Również programista nie musi się przejmować tym na jakich i ilu komputerach (ale może też mieć na to wpływ jeśli chce) zostanie cały proces uruchomiony, aplikacje pisze tak jak dla jednego równoległego komputera. Resztą zajmuje się w/w biblioteka wraz z demonem. Oczywiście liczba dostępnych komputerów nie musi i zwykle nie jest równa liczbie zadań, które uruchomi program. W skrajnym przypadku cały program składający się z wielu równoległych zadań może zostać uruchomiony na maszynie wirtualnej PVM stworzonej na pojedyńczym komputerze.

Demon niezbędny w każdym węźle tworzącym część wirtualnego komputera może zostać uruchomiony przez dowolnego użytkownika. Nie są wymagane żadne specjalne uprawnienia. Komunikacja pomiędzy węzłami realizowana jest poprzez datagramy UDP tak więc w warstwie sieciowej koniecznie jest zapewnienie możliwości przekazywania pakietów UDP pomiędzy węzłami sieci PVM (uwaga na firewalle).

Zadania zarządzane przez system PVM są reprezentowane w PVM przez unikalny TID (ang. task identifier). PVM zawiera funkcje, które powodują iż proces użytkownika staje się procesem maszyny wirtualnej PVM. PVM dostarcza funkcje do rozszerzania konfiguracji o nowe komputery jak i usuwania komputerów z PVM, do uruchamiania i usuwania zadań, do wysyłania sygnałów do innych procesów PVM oraz do testowania stanu konfiguracji i aktywnych procesów PVM.

PVM pozwala także na organizację zadań w grupy, które mogą się zmieniać dynamicznie podczas wykonania programu. Zadanie PVM może zostać dołączone lub może opuścić dowolną grupę w dowolnym momencie. Zadania należące do grup mogą wysyłać komunikaty do zadań innych grup jak i mogą otrzymać informację o innych zadaniach tej samej grupy. System PVM dostarcza specjal-

nej funkcji do wykonania zdefiniowanej przez użytkownika globalnej operacji na danych obliczonych przez wszystkie zadania grupy. W celu synchronizacji zadań tej samej grupy wprowadzono mechanizm blokowania wykonywania się zadań poprzez tzw. bariery. Mechanizm ten polega na wstrzymaniu wykonania zadań danej grupy (poprzez ustawienie bariery), aż do chwili ustawienia bariery przez ostatnie z zadań tej grupy. Wtedy wszystkie zadania rozpoczynają dalszą pracę.

Zadania PVM mogą wysyłać również sygnały do innych zadań. Zadania mogą otrzymywać informację o zdarzeniu poprzez przesłany komunikat wraz ze zdefiniowaną przez użytkownika etykietą, która będzie rozpoznawana przez aplikację. Zdarzenia mogą również dotyczyć istnienia zadań, usunięcia (uszkodzenia) komputera z sieci oraz dodania nowego komputera.

PVM dostarcza stosownych funkcji do pakowania, rozpakowywania i wysyłania komunikatów. Każde zadanie może wysyłać komunikaty do dowolnych innych zadań bez (teoretycznie) limitu na wielkość i ilość przesyłanych komunikatów. Jednakże, ponieważ bufory komunikatów są alokowane dynamicznie to wielkość komunikatu jaki może zostać wysłany lub odebrany zależy od wielkości dostępnej wolnej pamięci komputera odpowiednio wysyłającego lub odbierającego.

Komunikat może zawierać wiele tablic, z których każda może zawierać dane innego typu oraz nie ma ograniczeń na złożoność struktury pakowanych danych. Natomiast aplikacja powinna rozpakowywać komunikat dokładnie w ten sam sposób w jaki był on pakowany aby utrzymać integralność danych. Funkcje PVM do pakowania i rozpakowywania komunikatów obsługują wszystkie konieczne konwersje danych. Jednakże programista musi uważać na możliwość utraty dokładności w przypadku przesyłania danych np. z komputera 64-bitowego (CRAY) na komputer 32-bitowy (SPARC station). Model komunikacji w systemie PVM pozwala na asynchroniczne wysyłanie komunikatów, asynchroniczne blokowane i nie blokowane odbieranie komunikatów oraz zawiera funkcję testującą przybycie komunikatu bez jego odebrania. Dodatkowo PVM zawiera funkcję do wysyłania komunikatów do wielu zadań jednocześnie, na przykład do zdefiniowanej przez użytkownika grupy zadań.

Model PVM pozwala na naturalną wewnętrzną komunikację pomiędzy procesorami komputera wieloprocesorowego. Komunikaty pomiędzy procesorami są przesyłane bezpośrednio, natomiast komunikaty od lub do komputera wieloprocesorowego są przesyłane poprzez sieć odpowiednio od lub do demona PVM-a.

Użytkownik dostaje do dyspozycji również konsolę systemu PVM (o nazwie pvm), który umożliwia użytkownikowi interakcyjny sposób współpracy z maszyną wirtualną. Konsola może być uruchamiana wielokrotnie po skonfigurowaniu PVM, na każdym z komputerów aktualnej konfiguracji. Można również, uruchomić konsolę bez wcześniejszego skonfigurowania PVM i wtedy proces konsoli automatycznie uruchomi proces demona pvmd3 na tym samym komputerze.

Zastosowanie PVM

Ze względu na architekturę PVM, a także dostępność i koszty PVM można wykorzystać m.in. jako:

- możliwość stworzenia taniego super komputera wykorzystującego wolne cykle (po ustawieniu odpowiedniego priorytetu) istniejących i wykonujących inne funkcje komputerów
- możliwość taniego zwiększenia mocy obliczeniowej udostępnianiej pojedyńczemu programowi poprzez połączenie mocy więcej niż jednego komputera
- narzędzie edukacyjne do nauki programowania współbieżnego oraz do badań naukowych

Funkcje PVM

Większość możliwości PVM została już wspomniana we wcześniejszych punktach, jednak podsujmy jeszcze raz w uporządkowany sposób co oferuje PVM dla użytkownika:

- Zarządzanie zasobami – możliwość dodawania i usuwania komputerów z maszyny wirtualnej PVM. Należy podreślić, że z możliwości tej można korzystać również w czasie pracy aplikacji i maszyny wirtualnej, oczywiście bez szkody dla nich. Nie dość, że PVM umożliwia dość duże skalowanie systemu poprzez dodawanie kolejnych maszyn to jeszcze można to robić już w czasie działania aplikacji, zarówno w górę jak i w dół.
- Zarządzanie zadaniami – możliwość tworzenia, kończenia, a także zabijania zadań wchodzących w skład danego procesu/aplikacji. Przy tworzeniu zadania można pozostawić PVM decyzję co do tego, na którym węźle zostanie uruchomione zadanie lub określić architekturę sprzętową na której powinno zostać wykonane lub nawet wskazać konkretny węzeł sieci tworzącej maszynę wirtualną.
- Dynamiczne grupy zadań – umożliwiają grupowanie zadań i dostarczają dodatkowych możliwości dla grupy: komunikatów rozsyłanych po członkach grupy, mechanizmu synchronizacji grupy opartego o bariery (ang. barrier).
- Komunikacja – PVM umożliwia przekazywanie sygnałów oraz komunikatów do innych zadań. W przypadku komunikatów, które mogą być dowolnymi danymi PVM zapewnia funkcje do pakowania i rozpakowywania danych,

które to zapewniają odpowiednie konwersje danych pomiędzy różnymi architekturami sprzętowymi.

- Tolerancja uszkodzeń – PVM radzi sobie również z niezamierzoną rekonfiguracją wirtualnej maszyny, tzn. gdy któraś z maszyn przestanie prawidłowo pracować. Zadania pracujące na komputerze, który ulega awarii zostają oczywiście stracone, ale nie wpływa to na zadania pracujące w pozostałych węzłach, a także mogą być one o tym fakcie poinformowane co umożliwia aplikacji podjęcie odpowiednich działań.

Programowanie z PVM

Ponieważ PVM nie jest zbyt skomplikowanym dla użytkownika - programisty systemem najprościej go poznać programując z nim. Napiszemy więc prosty program w C korzystający z podstawowych możliwości PVM wprowadzając i omawiając je kolejno.

Oczywiście każdy program korzystający z biblioteki *libpvm* musi zawierać włączenie pliku nagłówkowego zawierające deklaracje API biblioteki, czyli rozpoczynamy od: `#include "pvm3.h"`.

Aby proces stał się procesem wirtualnej maszyny PVM powinien wywołać funkcję `pvm_mytid()` (choć podobny efekt spowoduje wywołanie każdej funkcji libpvm to jednak dla porządku warto wywołać ją jawnie). Wszystkie funkcje PVM sygnalizują błędy zwracając ujemną wartość tak więc program powinien sprawdzić czy operacja się powiodła. W przypadku powodzenia funkcja ta zwraca unikalny TID zadania. Proces może zakończyć swoją współpracę z PVM wywołując funkcję `pvm_exit()`.

Do uruchomienia równoległych zadań programista ma do dyspozycji funkcję `pvm_spawn()`, która jako argumenty przyjmuję nazwę procesu do wykonania, parametry do przekazania, liczbę egzemplarzy zadania do uruchomienia oraz opcjonalnie określenie na którym węźle bądź na jakiej architekturze uruchomić zadanie. Funkcja zwraca ilość uruchomionych zadań oraz tablicę ich TIDów. Do dyspozycji jest również funkcja `pvm_kill()` pozwalająca wymusić zakończenie zadania o konkretnym TID.

PVM udostępnia również szereg funkcji zwracających wiele informacji na temat stanu i konfiguracji maszyny wirtualnej oraz konkretnych zadań. Są to funkcje takie jak `pvm_parent()`, `pvm_config()`, `pvm_tasks()` itp.

Przyjżyjmy się teraz sposobom na przekazywanie komunikatów. Jeżeli chcemy wysłać komunikat najpierw musimy wywołać funkcję inicjalizującą bufor oraz sposób przesyłania. Służy do tego funkcja `pvm_initsend()`. Następnie należy „zapakować" dane do przesłania co zapewnia przy właściwym stosowaniu bez-

problemowe przesyłanie danych pomiędzy różnymi architekturami. Służą do tego funkcje z rodziny `pvm_pack()`, na przykład `pvm_packf()` działająca w sposób właściwy dla funkcji typu *printf* pozwalająca przesyłać jako ciąg różne typy danych. „Zapakowane" dane zostają umieszczone w buforze.

Wysłania danych oczekujących w buforze możemy dokonać na kilka sposobów. Jednym z nich jest wywołanie funkcji `pvm_send()`, która pozwala przekazać dane do procesu o konkretnym TID. Oprócz samej zawartości przy wysyłaniu komunikatu musimy określić tzw. etykietę komunikatu w postaci liczby całkowitej. Etykiety powinno się używać do oznaczenia rodzaju komunikatu. Zadanie odbierające komunikaty może następnie wybierać jakie komunikaty chce w danej chwili odebrać określając to właśnie poprzez tą etykietę. Może również dzięki temu zastosować odpowiedni sposób obróbki odebranych danych, w tym - co ważne - odpowiednią funkcję „rozpakowującą", która oczywiście powinna być konkretnym odpowiednikiem dla funkcji, którą użyto do „zapakowania" danych. W przypadku potrzeby wysłania komunikatu do większej liczby zadań można użyć funkcji `pvm_mcast()`, która pozwala podać tablicę TIDów jako adresatów komunikatu lub `pvm_bcast()`, która pozwala skierować komunikat do dowolnej grupy procesów.

Zadanie gotowe do odbierania komunikatów mogą to zrobić wywołując funkcję `pvm_recv()`, która odbiera komunikat od konkretnego lub dowolnego zadania (wg TID) o konkretnej lub dowolnej etykiecie, ale zawsze blokując wykonanie programu do czasu aż nadajdzie określony komunikat. Dlatego też istnieje podobna funkcja `pvm_trecv()`, która jednakże pozwala na określenie czasu po którym oczekiwanie na komunikat się zakończy. Do dyspozycji jest również całkiem nie blokująca `pvm_nrecv()`, która wraca zawsze od razu po wywołaniu bez względu na to czy jest dostępny określony komunikat czy nie. W niektórych przypadkach przydatna może być tez funkcja `pvm_probe()`, która tylko sprawdza czy określony (tak jak we wszystkich funkcjach odbierających wg TID lub etykiety komunikatu) komunikat jest dostępny.

Po odebraniu komunikatu jak już to było wspomniane należy wykonać odpowiednią funkcję „rozpakowującą" dane. Służą do tego funkcje z rodziny `pvm_unpack()`.

Wyposażeni w tą podstawową wiedzę możemy stworzyć już pierwszy program równoległy składający się z dwóch procesów. Pierwszy najpierw wywoła drugie zadanie potomne uruchamiając drugi proces, a następnie będzie oczekiwał na dowolny komunikat, który następnie wypisze na ekranie (zakładamy, że będzie to komunikat tekstowy). Drugie zadanie wyśle do swojego procesu - rodzica komunikat tekstowy.

```c
/* ex1.c */
#include <stdio.h>
#include "pvm3.h"

main()
{
        int cc, tid;
        char buf[255];

        printf("mój TID: %x\n", pvm_mytid());

        cc = pvm_spawn("ex1_sub", (char**)0, 0, "", 1, &tid);

        if (cc > 0) {
                cc = pvm_recv(-1, -1);
                pvm_bufinfo(cc, (int*)0, (int*)0, &tid);
                pvm_upkstr(buf);
                printf("komunikat od %x: %s\n", tid, buf);

        } else
                printf("wystąpił błąd przy próbie uruchomienia ex1_sub\n");

        pvm_exit();
        exit(0);
}

/* ex1_sub.c */
#include "pvm3.h"

main()
{
        int ptid;
        char buf[255];

        ptid = pvm_parent();
        strcpy(buf, "komunikat tekstowy od ex1_sub");
        pvm_initsend(PvmDataDefault);
        pvm_pkstr(buf);
        pvm_send(ptid, 1);

        pvm_exit();
        exit(0);
}
```

Przykład jest dość prosty ale pokazuje jak działają dwie z ważniejszych funkcji PVM - uruchamianie zadań oraz przekazywanie komunikatów. Przyjżyjmy się jeszcze jednemu przykładowi, który co prawda używa prawie tych samych funkcji PVM jednak prezentuje jeden z paradygmatów programów równoległych zwany często „master-slave". Polega on na tym, że jeden z procesów (zazwyczaj ten pierwszy uruchamiany na początku) zajmuje się kontrolowaniem całego przebiegu aplikacji. Uruchamia równoległe zadania, inicjalizuje je, przekazuje im dane i odbiera wyniki, podejmuje odpowiednie akcje w sytuacjach szczególnych. Poniższy program, również podzielony na dwa zadania - master i slave, w części master, która powinna być uruchomiona przez użytkownika stara się uruchomić trzy razy więcej procesów slave niż jest węzłów tworzących wirtualną maszynę PVM (jednak nie więcej niż 32). Jeżeli operacja się powiedzie wysyła przygotowaną procję danych do wszystkich utworzonych procesów, a następnie oczekuje od każdego z nich na odpowiedź. Procesy slave odbierają komunikat z porcją danych, następnie wykonują na nich pewną operację podczas której komunikują się z innymi procesami slave i wreszcie wysyłają wyniki do procesu master.

```
/* master.c */
#include <stdio.h>
#include "pvm3.h"

main()
{
    int mytid;                  /* TID */
    int tids[32];               /* slave task ids */
    int n, nproc, numt, i, who, msgtype, nhost, narch;
    float data[100], result[32];
    struct pvmhostinfo *hostp;

    mytid = pvm_mytid();

    pvm_config( &nhost, &narch, &hostp );
    nproc = nhost * 3;
    if( nproc > 32 ) nproc = 32 ;
        printf("Uruchamiam %d zadań." , nproc);

    numt=pvm_spawn("slave", (char**)0, 0, "", nproc, tids);
    if( numt < nproc ){
       printf("\n Błąd przy uruchamianiu slave:\n");
       for( i=numt ; i<nproc ; i++ ) {
          printf("TID %d %d\n",i,tids[i]);
       }
```

```
        for( i=0 ; i<numt ; i++ ){
            pvm_kill( tids[i] );
        }
        pvm_exit();
        exit(1);
    }
    printf("SUCCESSFUL\n");

    /* Begin User Program */
    n = 100;
    for( i=0 ; i<n ; i++ ){
        data[i] = 1.0;
    }

    /* Wysłanie komunikatu z danymi do utworzonych procesów */
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&nproc, 1, 1);
    pvm_pkint(tids, nproc, 1);
    pvm_pkint(&n, 1, 1);
    pvm_pkfloat(data, n, 1);
    pvm_mcast(tids, nproc, 0);

    /* Oczekiwanie na wyniki */
    msgtype = 5;
    for( i=0 ; i<nproc ; i++ ){
        pvm_recv( -1, msgtype );
        pvm_upkint( &who, 1, 1 );
        pvm_upkfloat( &result[who], 1, 1 );
        printf("Odebrałem %f od %d; ",result[who],who);
        if (who == 0)
                        printf( "(spodziewano %f)\n", (nproc - 1) * 100.0);
        else
                        printf( "(spodziewano %f)\n", (2 * who - 1) * 100.0

    }
    /* odłączenie od PVM */
    pvm_exit();
}

/* slave.c */
#include <stdio.h>
#include "pvm3.h"
```

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

    mytid = pvm_mytid();

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

    /* którym procesem jestem? */
    for( i=0; i<nproc ; i++ )
       if( mytid == tids[i] ){ me = i; break; }

    /* wykonanie obliczeń na danych */
    result = work( me, n, data, tids, nproc );
    /* wysłanie wyników */
    pvm_initsend( PvmDataDefault );
    pvm_pkint( &me, 1, 1 );
    pvm_pkfloat( &result, 1, 1 );
    msgtype = 5;
    master = pvm_parent();
    pvm_send( master, msgtype );

    /* odłączenie od PVM */
    pvm_exit();
}

float work(int me, int n, float *data, int *tids, int nproc )
{
    int i, dest;
    float psum = 0.0;
    float sum = 0.0;
    for( i=0 ; i<n ; i++ ){
       sum += me * data[i];
```

```
    }
    /* komunikacja z sąsiednim slave */
    pvm_initsend( PvmDataDefault );
    pvm_pkfloat( &sum, 1, 1 );
    dest = me+1;
    if( dest == nproc ) dest = 0;
    pvm_send( tids[dest], 22 );
    pvm_recv( -1, 22 );
    pvm_upkfloat( &psum, 1, 1 );

    return( sum+psum );
}
```

Zakończenie

Nieniejsze opracowanie zawiera zwięzły wstęp w świat programowania równoległego z PVM. Pokazuje do czego służy, co z PVM można osiągnąć oraz na kilku przykładach prezentuje jak szybko zacząć programować z PVM. Jednakże ze względu na ograniczony czas, który mógł zostać poświęcony temu opracowaniu nie omawia wszystkich tematów wystarczająco szeroko, niektóre takie jak obsługa we/wy, debugowanie, dodatkowe aplikacje wspomagające, bezpieczeństwo danych i programów pomija. Dlatego też zainteresowanego programowaniem z PVM czytelnika odsyłam do umieszczonego na końcu spisu literatury w celu uzupełnienia brakujących informacji.

Spis literatury

[1] Al Geist, Adam Begeuelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaidy Sunderam. *PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing*. The MIT Press. 1994.

[2] PVM Home Page – `http://www.epm.ornl.gov/pvm/pvm_home.html`

[3] An Introduction to PVM Programming – `http://www.epm.ornl.gov/pvm/intro.html`

[4] Obliczenia i programowanie równoległe – `http://k2.pcz.czest.pl/~roman/mat_dyd/prz_rown/prz_rown.html`

[5] Dokumentacja i przykłady dostarczane z pakietem PVM