Rozdíly mezi PVM a MPI.

Thursday, May 30, 2013 8:34 AM

- PVM a MPI se používají v prostředí s distribuovanou pamětí

Hlavní rozdíly

  - *PVM* - vznik v prostředí *heterogenní* sítě, *MPI* navržen spíše pro clustery - *homogenní* prostředí (u obou je ale běh možný v heteregonním prostředí)
  - MPI - jednodušší a efektivnější abstrakce na vyšší úrovni
  - MPI nabízí bohatší možnosti pro přenos zpráv (např. plně duplexní *sendrecv,* perzistentní komunikace, každý pošle každému - *MPI_Alltoall*)
  - MPI se už v návrhu snaží o omezení kopírování paměťových bloků
  - PVM se nestará o topologii, MPI podporuje logické komunikační topologie
  - MPI nemá žádný config jako PVM
  - MPI se vyhýbá nízkoúrovňovým rutinám kvůli přenositelnosti
  - MPI - možnost definovat vlastní datové typy pro snížení režie při posílání velkého množství dat (vs. PVM - vícenásobné volání *pvm_pack*)
  - MPI - podpora souborových operací (soubor se rozdělí na 1-N bloků, čtení a zápis jako zasílání zpráv)

PVM (Parallel Virtual Machine)

- Univerzální výpočetní model pro **heterogenní** distribuované výpočetní prostředí
- v PVM se musí provádět operace PVM_initsend, PVM_PK* apod.
- PVM umožňuje především provádět distribuované výpočty v heterogenním prostředí (různé architektury)
- PVM se nestará o topologii, MPI podporuje logické komunikační topologie.
- Programátor PVM může využít funkci PVM_Config aby např. určil, kde spustit další proces – MPI nic takového nemá.
- PVM programátorovi umožňuje, aby se do systému napojil pomocí nízko úrovňových rutin, MPI se tomu vyhýbá kvůli vyšší přenositelnosti.

Základní funkce

- Probíhá pomocí zasílání zpráv

pvm_spawn(char *task, char **argv, int flag,char *where, int ntask, int *tids)

- Spustí několik procesů podle zadaného programu
- Procesy se po vytvoření neznají -> proces, který je vytvořil je musí seznámit
  - numt – počet úspěšně spuštěných procesů (*návratová hodnota)*
  - task – spustitelný soubor
  - argv – parametry
  - flag – možnosti spuštění
    - PvmTaskDefault – PVM si rozhodne, kde spustit
    - PvmTaskHost – where bude obsahovat adresu, kde spustit
    - PvmTaskArch – where bude obsahovat typ architektury/platformy
    - Existují i další jako PvmTaskDebug, PvmTaskTrace, PvmMppFront, PvmHostCompl
  - where – kde spustit proces, ignoruje se, pokud nejsou příslušně nastaveny možnosti spuštění
  - ntask – počet procesů ke spuštění
  - tids – identifikátory spuštěných procesů

pvm_initsend(int encoding) – nastavení kódování (defaultně se používá PvmDataDefault)

int pvm_pkint(int *ip, int nitem, int stride) – vloží data do bufferu

int pvm_send(int tid, int msgtag) – pošle data procesu (TID)

int pvm_recv(int tid, int msgtag) – přijme data od procesu (TID)

pvm_upkint(int *ip, int nitem, int stride) – rozbalí data

MPI (Message Passing Interface)

- MPI je postaveno na PVM
- Knihovna pro podporu paralelních výpočtů v systémech s distribuovanou pamětí, vazby (interface) má pro programovací jazyky C a Fortran.

- Proti PVM se jedná o programovací prostředek vyšší úrovně, vztah je asi jako mezi jazykem symb. adres (odpovídá PVM) a vyšším programovacím jazykem (odpovídá MPI), tedy:
  - v PVM jde naprogramovat "skoro cokoliv", ale dá to velkou práci a program pak nejspíš nebude přenositelný,
    - dominantní aplikací pro MPI jsou numerické výpočty s regulárními daty (vektory či matice s číselnými prvky), přičemž pro takové aplikace je programování relativně pohodlné a program je dobře přenositelný do jiné instalace MPI

- MPI je primárně míněno (na rozdíl od PVM) pro **homogenní** výpočetní prostředí (tj. cluster stanic se společnou administrací, superpočítač jako N-Cube, ap.), takže poskytuje prostředky i pro "synchronní" algoritmy (tj. takové, kde se předpokládá přibližně stejná rychlost běhu jednotlivých procesů) a odpovídající statické rozdělení práce mezi procesy.

- MPI primárně využívá SPMD model paralelního výpočtu, tj. vyrobí se (na rozdíl od PVM) jen jeden "exe" soubor programu a ten se zavede do zvoleného počtu (dále N) procesorů. V každém procesoru běží jen jeden proces. Všech N procesů tudíž běží podle téhož programu, zjistí si své číselné ID a podle toho odliší svou činnost. V zásadě je tudíž realizovatelný i MPMD model výpočtu (třeba ve verzi farmer-workers, přičemž v programu je přepínač podle ID, jednu větev realizuje proces s číslem třeba 0 - farmer, druhá větev (jiná čísla než 0) je pro procesy typu worker).

- Komunikace procesů je asynchronní message-passing s přímým adresováním přes číselné ID procesu. Existuje mechanismus skupin procesů (viz dále tzv. komunikátory) a možnost broadcastu zprávy ve skupině procesů. Zprávy není na rozdíl od PVM třeba pracně "pakovat". MPI má svoje "primitivní datové typy" a z nich lze skládat "strukturované typy", sloužící ovšem jen pro účely komunikace (tj. zjednodušený popis toho, co má přijít do zprávy - lze srovnat s náročností "pakování" zprávy v PVM).

- Existuje sada funkcí pro tzv. "globální operace", tj. operace nad daty, jejichž instance jsou "rozprostřeny" ve všech procesech výpočtu. Realizace takových operací v PVM se musí pracně rozepsat do primitivnějších operací pvm_send() a pvm_recv().

Základní funkce

Je jich 6, přičemž už umožňují napsat jednodušší aplikaci. První čtyři z nich je třeba použít v každém MPI programu. Všechny funkce vrací celočíselný kód úspěšnosti provedené operace, přičemž symbolická hodnota MPI_SUCCESS znamená úspěch. Přehled základních funkcí v C-syntaxi:

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

- Inicializace MPI výpočtu, argc a argv jsou argumenty hlavního programu.

int MPI_Comm_size (MPI_Comm comm, int* adr_size)

- Zjištění počtu procesů, počet se dosadí do proměnné odkazované parametrem adr_size. MPI_Comm je typ "komunikátor", pokud při volání dosadíme za parametr comm hodnotu MPI_COMM_WORLD,

zjišťujeme počet všech vytvořených procesů aplikace N.

int MPI_Comm_rank (MPI_Comm comm, int* adr_rank)

- Zjištění čísla procesu v rámci "komunikačního světa" comm. Čísla jsou v rozmezí 0 až M-1, kde M je "rozměr komunikačního světa" reprezentovaného komunikátorem comm (M = N, pokud dosadíme za comm hodnotu MPI_COMM_WORLD).

int MPI_Finalize (void)

- Ukončení výpočtu v MPI, provádí každý proces.

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

- Odeslání zprávy s typem tag v komunikačním světě comm procesu dest. Odesílá se zpráva z bufferu buf obsahující count položek typu datatype.
- Čili buffer pro zprávu je na rozdíl od PVM kdekoliv v datech programu (zadá se adresa příslušného pole). Za datatype se dosazují buď primitivní typy MPI, například MPI_INT, MPI_DOUBLE, MPI_CHAR, nebo strukturované typy vytvořené z primitivních (viz dále).

int MPI_Recv (void* adr_buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *adr_status)

- Blokující příjem zprávy. Parametry mají analogický význam jako u MPI_Send. Navíc je parametr adr_status, odkazující kam se má uložit status příjmu zprávy (výstupní parametr). Status obsahuje položky: status. MPI_SOURCE - od koho zpráva přišla a status.MPI_TAG - jakého typu zpráva přišla. Dosadí-li se za source hodnota MPI_ANY_SOURCE a za tag MPI_ANY_TAG, přijme se jakákoliv zpráva a z uvedených položek výstupního parametru status se dá zjistit co to vlastně přišlo.

Globální operace MPI

- Globální operace jsou operace, do kterých jsou zapojeny všechny procesy patřídí do téhož "komunikačního světa" (tj. jedním parametrem funkcí pro globální operace je příslušný komunikátor). Funkci globální operace volají všechny zúčastněné procesy (což je pochopitelné, protože typicky procesy běží podle téhož programu), přičemž jejich činnost se v obecném případě liší podle čísla procesu.
- Globální operace v PVM nejsou (kromě broadcastu) a musely by se pracně rozepsat do posloupnosti operací send() a receive().

Globální operace MPI lze rozdělit na tři skupiny - synchronizace, přesuny dat a redukční operace.

Synchronizace

Každý komunikátor mj. realizuje bariéru, na které se mohou všechny jeho procesy synchronizovat voláním funkce

int MPI_Barrier (MPI_Comm comm)

  - Volání této funkce je tedy blokující a výpočet každého procesu pokračuje následujícím příkazem až poté, kdy se na bariéře "sejdou" všechny procesy patřící do "komunikačního světa" comm.

Přesuny dat

- Jedná se o operace s charakterem broadcastu, shromáždění či rozptýlení dat a tzv. redukční oprace.

int MPI_Bcast (void* adr_buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

- Operace broadcast. Má v zásadě stejné parametry jako MPI_Send(). Proces s číslem root vysílá, ostatní zprávu přijímají (čili root používá buffer jako vysílací, ostatní jako přijímací). Proti send() chybí tag - pro "globální" komunikační operaci nemá význam - všichni aktéři komunikace prochází tímtéž bodem programu a tudíž vědí, "o co při komunikaci jde".

int MPI_Gather (void* adr_inbuf, int incnt, MPI_Datatype intype, void* adr_outbuf, int outcnt, MPI_Datatype outtype, int root, MPI_Comm comm)

- Proces s číslem root shromažďuje data od všech procesů včetně sebe. Čili všechny procesy vysílají data (count položek typu intype) z bufferu inbuf a proces root je zapisuje do bufferu outbuf - samozřejmě je zapisuje v pořadí podle čísel vysílajících procesů (tj. nikoliv tak, jak zprávy došly). Parametr outbuf (a též outcnt a outtype) využije jen proces root. Za incnt a intype se normálně dosadí stejné hodnoty jako za outcnt a outtype.

int MPI_Scatter (void* adr_inbuf, int incnt, MPI_Datatype intype, void* adr_outbuf, int outcnt, MPI_Datatype outtype, int root, MPI_Comm comm)

- Inverzní operace k Gather(), tj. proces s číslem root "rozptyluje" data z bufferu inbuf všem ostatním procesům včetně sebe. Vstupní buffer musí obsahovat M částí dat (obecně různých, jedna část je pole count položek typu intype), přičemž i-tá část se pošle do bufferu outbuf procesu s číslem i. Parametr inbuf (a též incnt a intype) využije jen proces root. Za incnt a intype se normálně dosadí stejné hodnoty jako za outcnt a outtype.

Redukční operace

Redukční operace má M operandů umístěných ve vstupních bufferech komunikujících procesů. Výsledek operace se zapisuje do výstupního bufferu buď jednoho určeného procesu (root) nebo všech procesů.

int MPI_Reduce (void* adr_inbuf, void* adr_outbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

Za parametr op se dosazuje typ realizované operace, kde použitelné symbolické hodnoty typu jsou

- MPI_MAX, MPI_MIN (maximum, minimum),
- MPI_SUM, MPI_PROD (součet, součin),
- MPI_LAND, MPI_LOR, MPI_LXOR (logické operace)
- MPI_BAND, MPI_BOR, MPI_BXOR (logické operace se všemi bity)

Operandy jsou ve vstupních bufferech procesů, výsledek je ve výstupním bufferu procesu root.

int MPI_Allreduce (void* adr_inbuf, void* adr_outbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

- Funguje jako předchozí operace, ale výsledek dostanou do výstupního bufferu všechny zúčastněné procesy, tudíž chybí parametr root, který pak nemá smysl.

Komunikátor

Komunikátor je interní objekt MPI (typ MPI_comm, jedná se o typ tzv. "handle", čili jakéhosi zobecněného ukazatele reprezentujícího komunikátor).

Komunikátor reprezentuje skupinu komunikujících procesů a komunikační kontext této skupiny.

Dále komunikátor slouží pro paralelní členění programu, či jinak řečeno pro realizaci modelu MPMD (funkční paralelismus), kdy se procesy aplikace rozdělí na skupiny (reprezentované různými komunikátory) a každá skupina "dělá něco jiného" a její procesy "se vybavují jen mezi sebou". Čili uvnitř skupiny procesů (komunikátor) funguje datový paralelismus, kdežto mezi skupinami procesů funguje funkční paralelismus.

Základní funkce nad komunikátory jsou (podrobnosti viz literatura):

MPI_Comm_rank()

- vrací počet procesů komunikátoru, základní funkce MPI - viz dříve v části 2

MPI_Comm_dup()

- vytváří duplikát komunikátoru

MPI_Comm_split()

- "štěpí" kpomunikátor na dva jiné, čili rozděluje jednu skupinu procesů na dvě jiné

  MPI_Comm_free()

- uvolňuje (likviduje) komunikátor (samozřejmě nikoliv procesy, které komunikátor reprezentuje)

  MPI_Intercomm_create()

  vytváří tzv. "interkomunikátor" jakožto prostředek komunikace mezi dvěma skupinami procesů.

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>