

**IA039**

**Parallel Virtual Machine, PVM**

# Základní vlastnosti

- Parallel Virtual Machine (PVM)
  - Vyvinut koncem na přelomu 80. a 90. let minulého století
  - Primární motivace: tvorba paralelního virtuálního superpočítače
    - \* Nedostatečný výkon
    - \* Původně pouze superpočítače byly propojeny vysokorychlostní sítí
  - Postupně propojování pracovních stanic
  - Distribuované prostředí pro vývoj a spouštění distribuovaných programů

# Komponenty

- Sít pvmd démonů
  - Odpovídají za vlastní prostředí virtuálního paralelního (distribuovaného) počítače
- Knihovna funkcí
  - API pro meziproceso(ro)vou komunikace
  - API pro manipulaci s jednotlivými procesy
  - Přilinkována k programu

# Programování

- PVM podporuje primárně programy tvořené *kooperujícími úlohami*
  - Samostatné procesy na různých procesorech
  - Komunikace pomocí výměny zpráv
- Přirozená podpora *task paralelismu*
  - Odpovídá MPMD (Multiple Programs Multiple Data)
  - Vhodný i pro SPMD výpočetní model
  - Je možné realizovat i *data paralelní* programy
- API dostupné pro jazyky
  - C, C++, Fortran 77 (a Fortran 90)

# Základní třídy příkazů

- Řízení procesů
- Posílání zpráv
- Přijímání zpráv
- Správa bufferů
- Skupinové operace
- Informace
- Signály

# Řízení procesů

- Každý proces má vlastní identifikátor
  - **TID** –Task Identifier
  - Jedinečný v rámci celé PVM
- Nese informace o úloze i o jejím umístění v PVM
  - Umístění – uzel, na němž proces běží
- Vydáván master PVM démonem
  - Každá PVM má jeden master démon
  - Potenciální slabé míst
    - \* Nebezpeční přetížení
    - \* Single point of failure

# Příkazy řízení procesů

```
int tid = pvm_mytid(void)
```

```
int info = pvm_exit(void)
```

```
int info = pvm_kill(int tid)
```

```
int info = pvm_addhosts(char **hosts, int nhosts,  
                        int *infos)
```

```
int info = pvm_delhosts(char **hosts, int nhosts,  
                        int *infos)
```

# Spuštění nového procesu

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

flag: PvmTaskDefault

PvmTaskHost

PvmTaskArch

PvmTaskDebug

PvmTaskTrace

PvmMppFront

PvmHostCompl

# Posílání a přijímání zpráv

- Předávání zpráv zprostředkováno `pvmd` démony
  - Odpovídají za spolehlivý přenos a doručení
  - Zprávu vždy přebírá lokální (procesu příslušný) `pvmd`
  - Následně z TID cílové procesu určí jeho umístění a zprávu zašle vzdálenému `pvmd` procesu.
- Musí zajistit převod dat mezi různými architekturami

# Jednoduchý příklad

- Vlastní zaslání dat může realizovat např. následující jednoduchý příklad: `int bufid = pvm_initsend(int encoding)`
- Encoding udává způsob překódování dat
  - PvmDataDefault
    - \* Data jsou překódována do systémového bufferu
  - PvmDataRaw
    - \* Data se průchodem sítí nemění (vhodné pokud víme, že používáme identickou architekturu a prostředí)
  - PvmDataInPlace
    - \* Data jsou konvertována přímo v bufferu

# Zasílání zpráv

```
int info = pvm_bufinfo(int bufid, int *bytes,  
                       int *msgtag, int *tid)
```

```
int info = pvm_send(int tid, int msgtag)
```

```
int info = pvm_psend(int tid, int msgtag,  
                    char *buf, int len, int datatype)
```

```
int info = pvm_mcast(int *tids, int ntask,  
                    int msgtag)
```

# Přijímání zpráv

```
int info = pvm_recv(int tid, int msgtag)
```

```
int info = pvm_precv(int tid, int msgtag,  
                    char *buf, int len, int datatype,  
                    int atid, int atag, int alen)
```

```
int info = pvm_nrecv(int tid, int msgtag)
```

```
int info = pvm_trecv(int tid, int msgtag,  
                    struct timeval tmout)
```

# Ověření dostupnosti zprávy

```
int info = pvm_probe(int tid, int msgtag)
```

# Příprava a rozebírání dat

pvm\_pk<ptype>(<type> \*data, int cnt, int stride)

pvm\_upk<ptype>(<type> \*data, int cnt, int stride)

<ptype>	<type>
---------	--------

byte	char
------	------

short	short
-------	-------

int	int
-----	-----

long	long
------	------

float	float
-------	-------

double	double
--------	--------

cplx	float
------	-------

dcplx	double
-------	--------

str	char
-----	------

# Skupinové operace

- PVM podporuje tvorbu *skupin procesů*
  - Proces se může kdykoliv ke skupině přidat
  - Proces může skupinu kdykoliv opustit
- **Není nutná žádná synchronizace**
  - Je implementován model slabé konzistence

# Příkazy skupinové komunikace

```
int inum = pvm_ingroup(char *group)
int info = pvm_lvgroup(char *group)
int info = pvm_bcast(char *group, int msgtag)
int info = pvm_barrier(char *group, int count)
int info = pvm_reduce(void *op, void *data, int cnt,
                    int datatype, int msgtag, char *group,
                    int root)
void op(int *datatype, void *x, void *y,
        int *num, int *info)
int size = pvm_gsize(char *group)
int tid = pvm_gettid(char *group, int inum)
int inum = pvm_getinst(char *group, int tid)
```

# Množina informačních příkazů

```
int tid = pvm_parent(void)
int dtid = pvm_tidtohost(int tid)
int info = pvm_config(int *nhost, int *narch,
                    struct pvmhostinfo **hostp)
int info = pvm_tasks(int which, int *ntask,
                    struct pvmtaskinfo **taskp)
int oldval = pvm_setopt(int what, int val)
int val = pvm_getopt(int what)
```

# Pokročilé funkce

```
int info = pvm_reg_hoster()
```

```
int info = pvm_reg_tasker()
```

```
int info = pvm_reg_rm(struct hostinfo **hip)
```

- Všechny tři uvedené procedury slouží k nahrazení defaultních funkcí PVM při spouštění nových uzlů (pvmd) i nových úloh.
- `pvm_reg_hoster()` a `pvm_reg_tasker()` zajišťují konkrétní činnosti (addhost či spawn)
- `pvm_reg_rm()` umožní definovat nový plánovač (scheduler)

# Koordináční jazyky – Linda

## ■ Základní principy

- Sdílená tabule (tuple space)
- Matching (asociativní paměť)
- Minimální počet primitivních operací:
  - \* zápis (out)
  - \* blokující a neblokující a destruktivní (in) a nedestruktivní (rd) čtení
  - \* test
  - \* zápis (pod)úlohy

## ■ Persistentní tabule

## ■ Hierarchické tabule

# Piranha

- Aktivní vyhledávání „volných“ strojů
- Podpora fault-tolerance
- Základní vlastnosti
  - **feeder** – vytváří a distribuuje úlohy, sbírá výsledky
  - **piranha** – provádí vlastní výpočet
  - **retreat** – volán pokud piranha musí předčasně skončit