

**IA039**

**Message Passing Interface, MPI**

# Paralelní programování

## ■ Data paralelismus

- Stejné instrukce na různých procesorech zpracovávají různá data
- V podstatě odpovídá SIMD modelu (Single Instruction Multiple Data)
  - \* Např. paralelizace cyklu

## ■ Task paralelismus

- MIMD model (Multiple Instruction Multiple Data)
- Paralelně prováděné nezávislé bloky (funkce, procedury, programy)

## ■ SPMD

- Není synchronizován na úrovni jednotlivých instrukcí
- Ekvivalentní MIMD

# Prostředí/Systemy

- Parallel Virtual Machine, PVM
  - Primárně určen pro distribuované prostředí
- Message Passing Interface, MPI
  - Především podpora meziprocesorové komunikace
- Linda (Koordinační jazyk)
  - Expliciní task parallelismus
  - Komunikace přes sdílenou plochu

# Message Passing Interface

- Komunikační rozhraní pro paralelní programy
- Definováno API
  - Standardizováno
  - Řada nezávislých implementací
    - \* Možnost optimalizace pro konkrétní hardware
    - \* Určité problémy se skutečnou interoperabilitou

# Vývoj MPI

- Postupné uvádění verzí
  - Verze 1.0
    - \* Základní, nebyla implementována
    - \* Vazba na jazyky C a Fortran
  - Verze 1.1
    - \* Oprava největších nedostatků
    - \* Implementována
  - Verze 1.2
    - \* Přechodá verze (přes MPI-2)
    - \* Rozšíření standardu MPI-1

# Aktuální stav – MPI-2.0

- Aktuální verze
- Experimentální implementace plného standardu
- Rozšíření
  - Paralelní I/O
  - Jednosměrné operace (put, get)
  - Manipulace s procesy
- Vazba na C++ i Fortran 90

# Cíle návrhu MPI

## ■ Přenositelnost

- Definice standardu (API)
- Vazba na různé programovací jazyky
- Nezávislé implementace

## ■ Výkon

- Nezávislá optimalizace pro konkrétní hardware
- Knihovny, možnost výměny algoritmů
  - \* Např. nové verze kolektivních algoritmů

## ■ Funkcionalita

- Snaha pokrýt všechny aspekty meziprocessorové komunikace

# Cíle II

- Specifikace knihovny pro podporu pedávání zpráv
- Určena pro paralelní počítače, clustery i Gridy
- Zpřístupnění paralelního hardware pro
  - Uživatele
  - Autory knihoven
  - Vývojáře nástrojů a aplikací

# Core MPI

MPI_Init	Inicializace MPI
MPI_Comm_Size	Zjištění počtu procesů
MPI_Comm_Rank	Zjištění vlastní identifikace
MPI_Send	Zaslání zprávy
MPI_Recv	Přijetí zprávy
MPI_Finalize	Ukončení MPI

# Inicializace MPI

- Vytvoření prostředí
- Definuje, že program bude používat MPI knihovny
- Nemanipuluje explicitně s procesy

# Identifikace prostředí

- Paralelní (distribuovaný) program potřebuje znát
  - Kolik procesů se účastní výpočtu
  - Jaká je „moje“ identifikace
- `MPI_Comm_size(MPI_COMM_WORLD, &size)`
  - Vrací počet procesů sdílejících komunikátor `MPI_COMM_WORLD` (viz dále)
- `MPI_Comm_rank(MPI_COMM_WORLD, &rank)`
  - Vrací číslo procesu

# Práce se zprávami

- Primitivní model:
  - Proces A posílá zprávu: operace *send*
  - Proces B přijímá zprávu: operace *receive*
- Celá řada otázek:
  - Jak vhodně popsat data?
  - Jak specifikovat proces (kterému jsou data určena)?
  - Jak přijímající pozná, že data patří jemu?
  - Jak poznáme (úspěšné) dokončení těchto operací?

# Klasický přístup

- Data posíláme jako proud bytů
  - Je úkolem posílajícího a přijímajícího data správně nastavit a rozpoznat
- Každý proces má jedinečný identifikátor
  - Musíme znát identifikátor příjemce/vysílajícího
  - Broadcast operace
- Můžeme specifikovat příznak (tag) zprávy pro snazší rozpoznání (např. pořadové číslo zprávy)
- Synchronizace
  - Explicitní spolupráce vysílajícího a přijímajícího
  - Definuje pořadí zpráv

# Klasický přístup II

- `send(buffer, len, destination, tag)`
  - *buffer* obsahuje data, jeho délka je *len*
  - Zpráva je zaslána procesu s identifikací '*destination*'
  - Zpráva má příznak *tag*
- `recv(buffer, maxlen, source, tag, actlen)`
  - Zpráva bude přijata do paměťové oblasti specifikované položkou *buffer* jehož délka je *maxlen*
  - Skutečná délka přijaté zprávy je *actlen* ( $actlen \leq maxlen$ )
  - Zpráva přijde od procesu s identifikátorem *source* a musí mít příznak *tag*

# Nedostatky klasického přístupu

- **Nedostatečná úroveň definice dat**
  - Heterogenita cíle a zdroje (nekompatibilní reprezentace)
  - Příliš mnoho kopírování
  - Příliš vysoká zátěž programátora
- **Příznaky (tags) globální**
  - Komplikace při realizaci nezávislých knihoven

# Rozšíření v MPI

- Procesy mohou být uspořádány do **skupin**
- Každá zpráva má definovaný **kontext** (ne pouze příznak)
  - Zaslání a přijetí zprávy je možné pouze v rámci stejného kontextu
- Skupina a kontext společně definují **komunikátor**
  - Příznak může být lokální konkrétnímu komunikátoru
- Defaultní komunikátor **MPI\_COMM\_WORLD**
  - Skupina tvořená všemi procesy MPI programu
- Identifikace (rank) procesu je vždy uvnitř kontextu

# Datové typy

- Data nejsou popsána dvojicí (adresa, délka), ale trojicí (adresa, počet, datatyp)
- MPI Datatyp je definován *rekurzivně* jako:
  - Předdefinované datové typy používaného jazyka (např. MPI\_INT)
  - Souvislé pole MPI datatypů
  - Krokované (strided) pole MPI datatypů
  - Indexované pole bloků datatypů
  - Libovolná struktura datatypů
- K dispozici MPI funkce pro definici vlastních datatypů
  - Např. řádek matice která je ukládána po sloupcích

# Příznaky

- Každá zpráva má přiřazený příznaka (tag)
  - Usnadňuje rozpoznání zprávy přímajícím
  - Příznak vždy definován v rámci použitého kontextu
- Přijímající může specifikovat, jaký příznak očekává
  - Alternativně může příznaky ignorovat (specifikací MPI\_ANY\_TAG)

# Komunikační mody

- Standardní mod
  - Blokující
  - Může být bufferován
- Synchronní mod
  - Blokující
- Asynchronní mod
  - Neblokující
- Ready mod
  - Receive musí předcházet send (připraví cílový buffer)

# Základní *send* operace

- Blokující send
  - `MPI_SEND(start, count, datatype, dest, tag, comm)` beginmitemize
  - Trojice (`start, count, datatype`) definuje zprávu
  - *dest* definuje cílový proces, a to relativně vzhledem ke komunikátoru *comm*
- Když operace skončí, znamená to
  - Všechna data byla systémem akceptována
  - Buffer je možné okamžitě znovupoužít
  - Příjemce nemusel ještě data dostat

# Základní *receive* operace

## ■ Blokující operace

- MPI\_RECV(start, count, datatype, source, tag, comm, status)
  - \* Operace čeká dokud nedorazí zpráva s odpovídající dvojicí (source, tag)
  - \* *source* je identifikátor odesílatele (relativně vůči komunikátoru *comm*) nebo MPI\_ANY\_SOURCE
  - \* *status* obsahuje informace o výsledku operace
    - Obsahuje také příznak zprávy a identifikátor procesu při použití specifikací MPI\_ANY\_TAG a MPI\_ANY\_SOURCE)
  - \* Pokud přijímaná zpráva obsahuje méně než *count* bloků, není to identifikováno jako chyba (je poznačeno v *status*)
  - \* Přijetí více jak *count* bloků je chyba

# Krátký Send/Receive protokol

- Plně duplexní komunikace
  - Každá zasílaná zpráva odpovídá přijímané zprávě
- `int MPI_Sendrecv(void *sendbuf, int sendcnt, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int reccnt, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)`

# Asynchronní komunikace

- **Neblokující operace *send***
  - Buffer k dispozici až po úplném dokončení operace
- **Operace *send* i *receive* vytvoří požadavek**
  - Následně lze zjišťovat stav požadavku
- **Volání**
  - ```
int MPI_Isend(void *buf, int cnt, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm,
              MPI_Request *request)

int MPI_Irecv(void *buf, int cnt, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm,
              MPI_Request *request)
```

# Asynchronní operace II

- (Blokující) čekání na výsledek operace
  - `int MPI_Wait(MPI_Request *request, MPI_Status *status)`
  - `int MPI_Waitany(int cnt, MPI_Request *array_of_requests,  
int *index, MPI_Status *status) |`
  - `int MPI_Waitall(int cnt, MPI_Request *array_of_requests,  
MPI_Status *array_of_statuses)`

# Asynchronní operace III

- Neblokující zjištění stavu

- `int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)`

- `int MPI_Testany(int cnt, MPI_Request *array_of_requests, int *flag, int *index, MPI_Status *status)`

- `int MPI_Testall(int cnt, MPI_Request *array_of_requests, int *flag, MPI_Status *array_of_statuses)`

- Uvolnění požadavku

- `int MPI_Request_free(MPI_Request *request)`

# Trvalé (persistentní) komunikační kanály

- Neblokující
- Spojeny ze dvou „půl“-kanálů
- Životní cyklus
  - **Create (Start Complete)\* Free**
  - Vytvořeny, pak opakovaně používány, následně zrušeny

# Persistentní kanál – vytvoření

```
int MPI_Send_init(void *buf, int cnt, MPI_Datatype datatype,  
                 int dest, int tag, MPI_Comm comm,  
                 MPI_Request *request)
```

```
int MPI_Recv_init(void *buf, int cnt, MPI_Datatype datatype,  
                 int dest, int tag, MPI_Comm comm,  
                 MPI_Request *request)
```

# Přenos

- Zahájení přenosu (Start)

- `int MPI_Start(MPI_Request *request)`

- `int MPI_Startall(int cnt, MPI_Request *array_of_request)`

- Zakončení přenosu (Complete)

- Jako u asynchronních (wait, test, probe)

# Zrušení kanálu

- Ekvivalentní zrušení odpovídajícího požadavku

```
int MPI_Cancel(MPI_Request *request)
```

# Kolektivní operace

- Operace provedené současně všemi procesy v rámci skupiny
  - Broadcast: `MPI_BCAST`
    - \* Jeden proces (root) pošle data všem ostatním procesům
  - Redukce: `MPI_REDUCE`
    - \* Spojí data ode všech procesů ve skupině (komunikátoru) a vrátí jednomu procesu
  - Často je možné skupinu příkazů *send/receive* nahradit *bcast/reduce*
    - \* Vyšší efektivita (*bcast/reduce* optimalizováno pro daný hardware)

# Kolektivní operace II

- Další operace
  - `alltoall`: výměná zpráv mezi všemi
  - `bcast/reduce` realizuje tzv. *scatter/gather* model
- Speciální redukce
  - `min`, `max`, `sum`, ...
  - Uživatelsky definované kolektivní operace

# Virtuální topologie

- MPI umožňuje definovat komunikační vzory odpovídající požadavkům aplikace
- Ty jsou v dalším kroku mapovány na konkrétní komunikační možnosti použitého hardware
  - Transparentní
- Vyšší efektivita při psaní programů
- Přenositelnost
  - Program není svázan s konkrétní topologií použitého hardware
- Možnost nezávislé optimalizace

# Datové typy

- Mapa typu

- $\text{Typemap} = \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\}$

- Signatura typu

- $\text{Typesig} = \{type_0, \dots, type_{n-1}\}$

- Příklad:

- $\text{MPI\_INT} == \{(int, 0)\}$

# Rozsah a velikost

- `MPI_Type_extent(MPI_Datatype Type, MPI_Aint *extent)`
- `MPI_Type_size(MPI_Datatype Type, int *size)`
- Příklad:
  - `Type = {(double,0),(char,8)}`
  - `extent = 16`
  - `size = 9`

# Konstrukce datotypů

- Souvislý datový typ

- `int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)`

- Vektor

- `int MPI_Type_vector(int count int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)`
  - `int MPI_Type_hvector(int count int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)`

# Konstrukce datotypů II

## ■ Indexovaný datový typ

- `MPI_Type_indexed(int count, int *array_of_blocklengths, int *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype)`

`MPI_Type_hindexed(int count, int *array_of_blocklength, int *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype)`

## ■ Struktura

- `MPI_Type_struct(int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types, MPI_Datatype *newtype)`

# Konstrukce datatypů III

- Potvrzení definice datového typu

- `int MPI_Type_commit(MPI_Datatype *datatype)`

- Krokové (strided) datové typy

- Mohou obsahovat „díry“

- Implementace může optimalizovat způsob práce

- Příklad: Každý druhý prvek vektoru

- \* Může skutečně „složit“

- \* Může ale také poslat celý vektor a druhá strana vybere jen specifikované typy

# Operace nad soubory

- Podpora až od MPI-2
- „Paralelizace“ souborů
- Základní pojmy

file

etype

- view

file size

file handle

displacement

filetype

offset

file pointer

# Operace nad soubory II

| Umístění                  | Synch                           | Koordinace                                    |                                                                                                                    |
|---------------------------|---------------------------------|-----------------------------------------------|--------------------------------------------------------------------------------------------------------------------|
|                           |                                 | nekolektivní                                  | kolektivní                                                                                                         |
| explicitní<br>offset      | blokující                       | MPI_File_read_at<br>MPI_File_write_at         | MPI_File_read_at_all<br>MPI_File_write_at_all                                                                      |
|                           | neblokující &<br>split collect. | MPI_File_iread_at<br><br>MPI_File_iwrite_at   | MPI_File_read_at_all_begin<br>MPI_File_read_at_all_end<br>MPI_File_write_at_all_begin<br>MPI_File_write_at_all_end |
| individuální<br>file ptrs | blokující                       | MPI_File_read<br>MPI_File_write               | MPI_File_read_all<br>MPI_File_write_all                                                                            |
|                           | neblokující &<br>split collect. | MPI_File_iread<br><br>MPI_File_iwrite         | MPI_File_read_all_begin<br>MPI_File_read_all_end<br>MPI_File_write_all_begin<br>MPI_File_write_all_end             |
| sdílený<br>file ptr.      | blokující                       | MPI_File_read_shared<br>MPI_File_write_shared | MPI_File_read_ordered<br>MPI_File_write_ordered                                                                    |
|                           | neblokující &<br>split collect. | MPI_File_iread_shared                         | MPI_File_read_ordered_begin<br>MPI_File_read_ordered_end                                                           |
|                           | split collect.                  | MPI_File_iwrite_shared                        | MPI_File_write_ordered_begin<br>MPI_File_write_ordered_end                                                         |

# MPI a optimalizující překladače

- Při asynchronním použití se mění hodnoty polí, o nichž překladač nemusí vědět

- Kopírování parametrů způsobí ztrátu dat

```
call user(a, rq)
```

```
call MPI_WAIT(rq, status, ierr)
```

```
write (*,*) a
```

```
subroutine user(buf, request)
```

```
call MPI_IRECV(buf,...,request,...)
```

```
end
```

- V tomto případě se v hlavním programu vypíše nesmyslná hodnota „a“, protože se při návratu z „user“ zkopíruje aktuální hodnota; přitom operace *receive* ještě nebyla dokončena

# Další rozvoj

- MPI-1 standard neobsahuje
  - Explicitní operace nad sdílenou pamětí
  - Specifickou podporu ze strany operačního systému (aktivní zprávy, příjem zpráv řízený událostmi, ...)
  - Ladicí nástroje
  - Explicitní podporu vláken (základy jsou ve verzi 2.0) je však thread-safe
  - Podporu správy úloh (je ve verzi 2.0)
  - I/O (je ve verzi 2.0)