

MPI: MESSAGE PASSING INTERFACE

Podklady k přednášce kurzu PRL
2018, 2019

PROSTŘEDKY PRO TVORBU PARALELNÍCH SYSTÉMŮ

- Mezi současné systémy, které umožňují tvorbu paralelních systémů patří **openMP**, **PVM** (Parallel Virtual Machine) či **MPI** (Message Passing Interface)
 - openMP je systém založený na **komunikaci sdílenou pamětí**
 - PVM a MPI jsou systémy, které jsou založené na **komunikaci předáváním zpráv**

MPI VS. OPENMP

- Ale ... MPI primárně podporuje komunikaci předáváním zpráv, od verze 3 i komunikaci sdílenou pamětí, zatímco openMP pouze komunikaci sdílenou pamětí
- MPI podporuje pouze paralelismus na úrovni procesů, později vláken, openMP pouze na úrovni vláken
- OpenMP může mít vedle privátních i sdílené proměnné
- Realizace MPI může být od více vydavatelů pro různé systémy s vlastními překladači, openMP je překládán specifickým překladačem

PVM, MPI: HISTORIE

- Vývoj PVM byl započat v roce 1989 na Oak Ridge National Laboratory.
- Vývoj MPI probíhá od dubna 1992, je koordinovaný MPI Fórem
- **PVM Verze 2** 1991
- **MPI Verze 1** 1992 –
První verze, základní koncept,
statické procesy (nemění se během výpočtu)
- **PVM Verze 3** únor 1993, protokol TCP
- **MPI Verze 2** 2008 – Dynamická správa procesů
- **MPI Verze 3** 2012 – Neblokující kolektivní operace,
sdílená paměť
- **MPI Verze 4** ???, viz <http://mpi-forum.org/mpi-40/>

MPI VS. PVM

- PVM je **distribuovaný operační systém**, kdežto MPI je **knihovna** pro vytváření paralelních aplikací
 - PVM běží jako démon na různých pracovních stanicích a abstrahuje je do jednoho operačního systému
- Oba systémy jsou **portabilní**, tj. mohou být použity pod různými hostitelskými operačními systémy
- Oba systémy jsou **heterogenní**, tj. mohou pracovat s více různými architekturami procesorů
- PVM umožňuje **dynamickou správu procesů** a automatické zotavování se z chyb či dynamickou správu prostředků
- MPI umožňuje vytvářet procesy za běhu až od verze 2. Pád jednoho procesu může zapříčinit nefunkčnost celého systému (například při kolektivních operacích)

PRINCIPY MPI

- Knihovna pro implementaci paralelních systémů komunikujících předáváním zpráv. Umožňuje vytvářet aplikace pro **paralelní architektury**, **počítačové clustery** i **heterogenní sítě**.
- Nezávislé na hostitelském jazyce (implementace primárně pro **C++**, **Fortran**, ale i další)
- Uživateli odstíňuje hardware a umožňuje jej využívat **knihovnými funkcemi**.
- Umožňuje komunikaci v modelech **dvoubodových** (P2P, Send / Receive) a **kolektivních** (Broadcast / Reduce)

MPI, PŘEKLAD A SPUŠTENÍ

- Na serveru Merlin – openMPI verze 1.8.7
- openMPI od verze 1.8 podporuje MPI - 3

- Překlad:

```
mpicc hw.cc
```

// parametr -np udává počet procesů

- Spuštění:

```
mpirun -np 1 hw
```

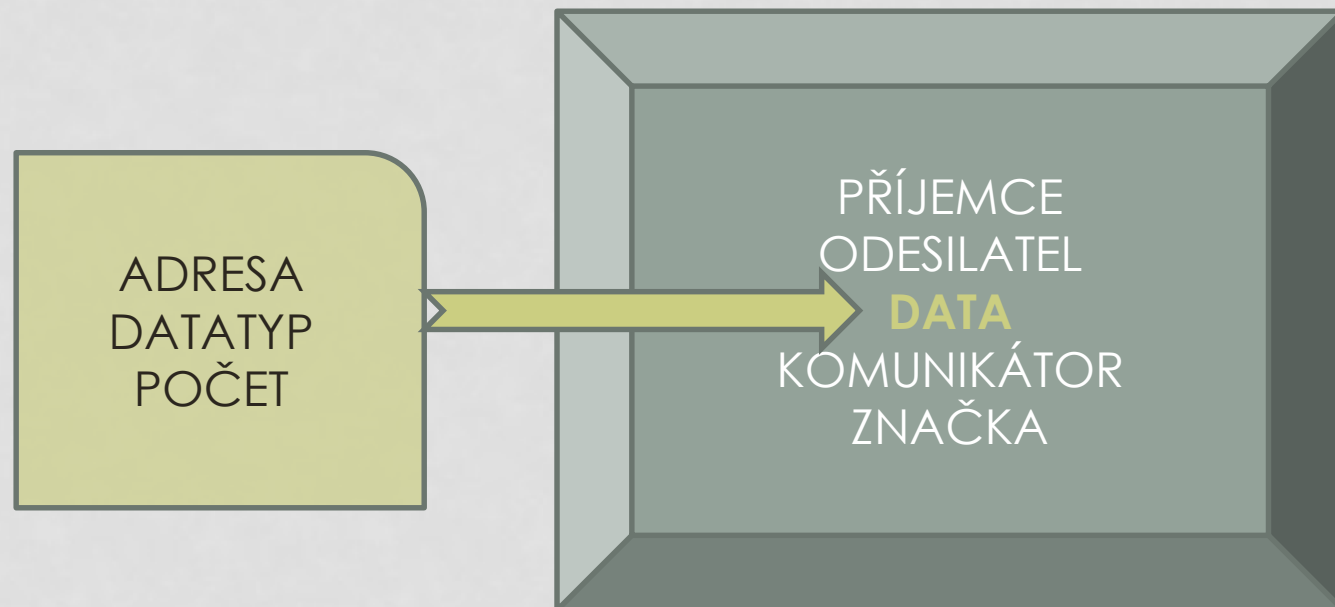
MPI INICIALIZACE A UKONČENÍ

- Korektně musí být kód používající MPI inicializován a ukončen operacemi Init a Finalize

```
MPI_Init (&argc, &argv);  
printf("Hello, world! \n");  
MPI_Finalize();
```

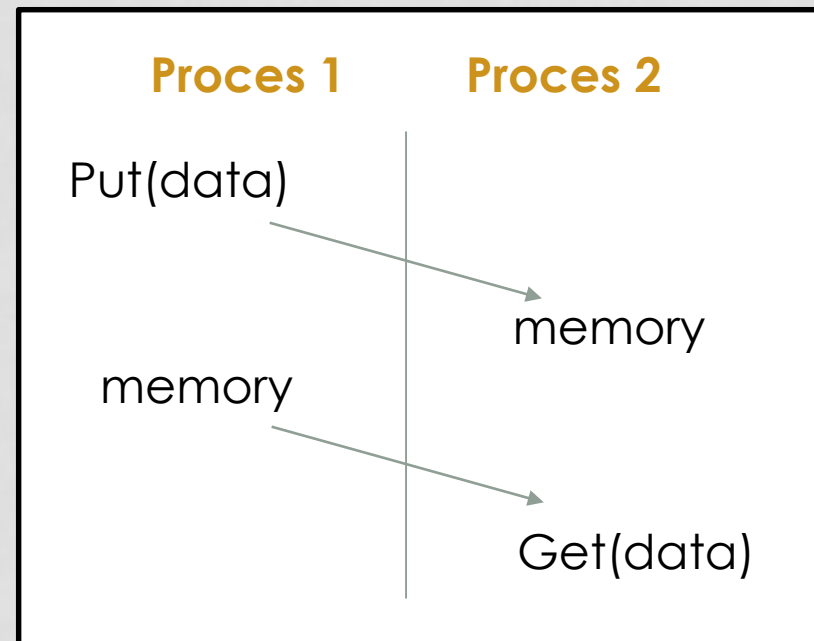
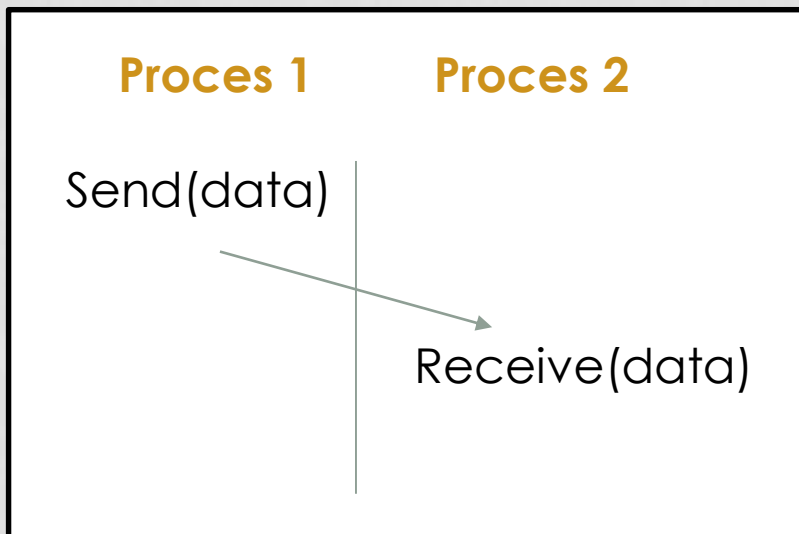

DVOUBODOVÁ KOMUNIKACE

- Pro dvoubodový spoj (Point-to-point, P2P) zasílají zprávu komunikační primitiva typu **Send** a přijímají ji operace typu **Receive**.
- Data jsou dána adresou, datovým typem a počtem
- Data jsou umístěna v obálce, která zahrnuje položky:



KOOPERATIVNÍ VS NEKOOPERATIVNÍ KOMUNIKACE

- **Send / Receive** operace jsou operace kooperativní, spolupracující. Oproti tomu od MPI-2 je možné jednostraně přistupovat operacemi **Put** a **Get**



DVOUBODOVÝ SPOJ - OPERACE

```
int MPI_Send(const void *buf, int count, MPI_Datatype  
    datatype, int dest, int tag, MPI_Comm comm)
```

- Zasílají se data z vyrovnávací paměti (bufferu) daného typu a daného počtu
- Příjemce je uveden pozicí (rankem) v komunikátoru
- Zpráva může být opatřena značkou

```
int MPI_Recv(void *buf, int count, MPI_Datatype  
    datatype, int source, int tag, MPI_Comm comm,  
    MPI_Status *status)
```

- Přijímají se data do vyrovnávací paměti (bufferu) daného typu a daného počtu
- A to jen od odesilatele uvedeného pozicí v komunikátoru a odpovídající značce (pomocí komunikátoru a značek oddělujeme a abstrahujeme kontext komunikace)
- Stav (v tomto případě po ukončení) komunikace lze vyčíst ze struktury status
- Výstupem je kód signalizující úspěch, nebo důvod neúspěchu operace

INTER / INTRA KOMUNIKÁTOR

- Komunikátor definuje kontext předávání zpráv. Pouze procesy uvnitř komunikátoru mohou spolu komunikovat
- Komunikátorem se obvykle (a ve verzi MPI-1 výhradně) myslí **intrakomunikátor**, který zahrnuje navzájem komunikující procesy
- **Interkomunikátor** zahrnuje dvě skupiny procesů, které spolu mohou navzájem komunikovat
- Druhý typ komunikátorů se používá převážně u kolektivních operací

KOMUNIKÁTOR

- Komunikátor je abstrakce pro skupinu procesů
- Proces může komunikovat s jedním nebo více procesy v rámci komunikátoru
- **MPI_COMM_WORLD** zahrnuje všechny procesy aplikace
- Za běhu lze zjistit, kolik zahrnuje komunikátor procesů a který z procesů v rámci komunikátoru je daný proces (který operaci vykoná)

```
MPI_Comm_rank (MPI_COMM_WORLD, &mytid)
```

```
MPI_Comm_size (MPI_COMM_WORLD, &size)
```

- Komunikátory lze vytvářet vlastní, pro část procesů původního komunikátoru

KOMUNIKÁTOR

```
#include "mpi.h"

#include <stdio.h>

int main( int argc, char *argv[] )
{
    int rank, size;

    MPI_Init( &argc, &argv );

    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();

    return 0;
}
```

KOMUNIKACE – ZNAČKA (TAG)

- Kontext je dán komunikátorem, ale uživatel může vytvořit vlastní **značku** (tag), která určí dosah komunikace (kontext je značkou řízenou systémem, tag uživatelem)
- Intrakomunikátor využívá jednu značku, interkomunikátor dvě – pro každou z obou skupin procesů
- Značka může být přirozené číslo hodnotou z intervalu 0 ... **MPI_TAG_UB**
- **ANY_TAG** zobecňuje na celý prostý značek, tzn. že rozhodování o přijetí či nepřijetí zprávy značka neovlivňuje

KOMUNIKAČNÍ KONTEXT – KOMUNIKÁTOR, NEBO ZNAČKA?

- Důvodu pro použití či nepoužití tagu je řada, například:
 - (- proti tagu) Není vhodné používat, pokud hrozí, že v rámci jednoho procesu může dojít k použití stejné značky pro různé komunikace
 - (+ pro tag) Naopak je výhodná k identifikaci komunikace, kdy nemusíme znát *rank* příjemce, např. při duplikování komunikátorů, kdy může dojít k přeuspořádání procesů a změně jejich ranků
 - (+ pro tag) Práce s komunikátorem zabere čas, vytvoření nového komunikátoru konzumuje více prostředků, než označení zprávy značkou.

KOMUNIKACE - DATOVÉ TYPY

- Datové typy společně s adresou a počtem udávají obsah, který se má odeslat, nebo přijmout během komunikace
- Odstiňuje reprezentaci datových typů v konkrétním programovém prostředí (PASCAL vs. C++)
- Předdefinované, např **MPI_BYTE**
- Základní, obecně používané programovacími jazyky
 - PASCAL: **MPI_CHAR**, **MPI_INT**, **MPI_DOUBLE**
 - FORTRAN: **MPI_CHARACTER**, **MPI_COMPLEX**
- které odpovídají datovým typům z výše uvedených jazyků
 - Vektor MPI datotypů
 - Indexované pole MPI datotypů
 - Libovolný soubor bloků výše uvedených
- Lze definovat i vlastní (ukážeme si příklad později)

KOMUNIKACE – STATUS

- Struktura **MPI_Status** obsahuje položky

```
int count;  
int cancelled;  
int MPI_SOURCE;  
int MPI_TAG;  
int MPI_ERROR;
```

... využijeme převážně později u asynchronní komunikace

Pozn: operace **MPI_Probe** – funguje podobně jako operace **receive**, ale nepřijímá zprávu jako takovou, pouze počká do přijetí nějaké zprávy v daném kontextu. Po jejím přijetí pozmění patřičně **status**.

P2P KOMUNIKACE

- **Synchronní**
- (MPI_Ssend, MPI_Srecv) – operace jsou blokuující, dokud nejsou dokončeny obě z nich
- **Asynchronní**
 - Blokuující (MPI_Send, MPI_Recv) – operace je ukončena, pokud je buffer použitý ke komunikaci bezpečně k dispozici
 - Neblokuující, (MPI_Isend, MPI_Irecv) – operace zahájí zpracování komunikace a je ukončena

PŘÍKLAD 1, P2P ZASÍLÁNÍ ZPRÁV, BLOKUJÍCÍ SEND

```
MPI_Init(&argc, &argv);
```

```
int i, j, sndrdata = 10, rcvdata = 20, sndr=1, rcvr=2, rank, data=0;  
MPI_Status status;  
MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
for(i=1; i<12; i++){  
    if(rank == sndr){  
        data++;  
        MPI_Send(&data, 1, MPI_INT, rcvr, MY_TAG,  
                MPI_COMM_WORLD);  
        printf("Sent from source %d / data %d\n", rank, data);  
        for(j=1; j<10000; j++); // delay  
    }  
    if(rank == rcvr){  
        MPI_Recv(&data, 1, MPI_INT, sndr, MY_TAG,  
                MPI_COMM_WORLD, &status);  
        printf("Received from source %d / data %d\n", rank, data);  
    }  
}
```

```
MPI_Finalize();
```

PŘÍKLAD 1, P2P ZASÍLÁNÍ ZPRÁV, BLOKUJÍCÍ SEND, VÝSTUP

```
Sent from source 1 / data 1
Sent from source 1 / data 2
Sent from source 1 / data 3
Sent from source 1 / data 4
Sent from source 1 / data 5
Sent from source 1 / data 6
Sent from source 1 / data 7
Sent from source 1 / data 8
Sent from source 1 / data 9
Sent from source 1 / data 10
Sent from source 1 / data 11
Received from source 1 / data 1
Received from source 1 / data 2
Received from source 1 / data 3
Received from source 1 / data 4
Received from source 1 / data 5
Received from source 1 / data 6
Received from source 1 / data 7
Received from source 1 / data 8
Received from source 1 / data 9
Received from source 1 / data 10
Received from source 1 / data 11
```

PŘÍKLAD 2, P2P ZASÍLÁNÍ ZPRÁV, SYNCHRONNÍ SEND

```
MPI_Init(&argc, &argv);
```

```
int i, j, sndrdata = 10, rcvdata = 20, sndr=1, rcvr=2, rank, data=0;  
MPI_Status status;  
MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
for(i=1; i<12; i++){  
    if(rank == sndr){  
        data++;  
        MPI_Ssend(&data, 1, MPI_INT, rcvr, MY_TAG,  
                MPI_COMM_WORLD);  
        printf("Ssent from source %d / data %d\n", rank, data);  
        for(j=1; j<10000; j++); // delay  
    }  
    if(rank == rcvr){  
        MPI_Srecv(&data, 1, MPI_INT, sndr, MY_TAG,  
                MPI_COMM_WORLD, &status);  
        printf("Sreceived from source %d / data %d\n", sndr, data);  
    }  
}
```

```
MPI_Finalize();
```

PŘÍKLAD 2, P2P ZASÍLÁNÍ ZPRÁV, SYNCHRONNÍ SEND, VÝSTUP

Ssent from source 1 / data 1
Sreceived from source 1 / data 1
Sreceived from source 1 / data 2
Ssent from source 1 / data 2
Ssent from source 1 / data 3
Sreceived from source 1 / data 3
Sreceived from source 1 / data 4
Ssent from source 1 / data 4
Sreceived from source 1 / data 5
Ssent from source 1 / data 5
Ssent from source 1 / data 6
Sreceived from source 1 / data 6
Ssent from source 1 / data 7
Sreceived from source 1 / data 7
Sreceived from source 1 / data 8
Ssent from source 1 / data 8
Ssent from source 1 / data 9
Sreceived from source 1 / data 9
Sreceived from source 1 / data 10
Ssent from source 1 / data 10
Ssent from source 1 / data 11
Sreceived from source 1 / data 11

KOMUNIKACE, KOLEKTIVNÍ

- Kolektivní operace jsou blokující, dokud je neprovedou **všechny** procesy v uvedeném komunikátoru
- Pro komunikaci je k dispozici řada operací, nejdůležitější (ve smyslu 'obvykle používané') jsou
- **Broadcast** – Kromě dat a obálky je třeba znát proces, který vysílá, tj. předá data všem ostatním procesům v komunikátoru.

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

- **Reduce** – Uvedený proces provede redukci uvedenou operací nad daty – prvky z každého procesu v komunikátoru

```
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype  
datatype, MPI_Op op, int root, MPI_Comm comm)
```

- Stejně jako u P2P operacím je výstupem úspěch – neúspěch operace
- Broadcast / Reduce model je alternativou pro P2P Send / Receive

OPERACE PRO REDUKCI

- [MPI_MAX] maximum
- [MPI_MIN] minimum
- [MPI_SUM] sum
- [MPI_PROD] product
- [MPI_LAND] logical and
- [MPI_BAND] bit-wise and
- [MPI_LOR] logical or
- [MPI_BOR] bit-wise or
- [MPI_LXOR] logical xor
- [MPI_BXOR] bit-wise xor
- [MPI_MAXLOC] max value and location
- [MPI_MINLOC] min value and location

PŘÍKLAD 3: – REDUKCE, SUMA

- **MPI_Init(&argc, &argv);**
- int rank,size,data;
- **MPI_Comm_rank(MPI_COMM_WORLD, &rank);**
MPI_Comm_size(MPI_COMM_WORLD, &size);
- int buf=(rank*142)%128;
- printf("Jsem rank:%d, mám data:%d \n",rank, buf);
- **MPI_Reduce(&buf, &data, 1, MPI_INT, MPI_SUM, 0,**
MPI_COMM_WORLD);
- if(rank==0)
- printf(„SUM rank:%d - data:%d \n",rank, data);
- **MPI_Finalize();**

Reukce operací SUM lokálních 'buf' délky 1, typu INT všech procesů v komunikátoru MPI_COMM_WORLD. Výsledek v 'buf' procesu 0

PŘÍKLAD 3: – REDUKCE, MAXIMUM

```
MPI_Init(&argc, &argv);
```

```
int rank,size,data;
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```
int buf=(rank*142)%128;
```

```
printf("Jsem rank:%d, mám data:%d \n",rank, buf);
```

```
MPI_Reduce(&buf,&data, 1, MPI_INT, MPI_MAX, 0  
          MPI_COMM_WORLD);
```

```
if(rank==0)
```

```
    printf(„SUM rank:%d - data:%d \n",rank, data);
```

```
MPI_Finalize();
```

PŘÍKLAD 4: REDUKCE (V KTERÉ POLOVINĚ JE MAXIMUM?)

Vynulujeme **buf2** pro první polovinu a **buf** pro druhou polovinu

```
MPI_Init(&argc, &argv);
int rank, size, data, rdc, mx1, mx2, rdc=0;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
int buf=(rank*1234)%33; // nebo něco jiného
int buf2=buf;
if (rank>=(size/2)) buf=0; else buf2=0;
printf("Jsem rank:%d data:%d data2:%d\n",rank, buf, buf2);
MPI_Reduce(&buf, &mx1, 1, MPI_INT, MPI_MAX, rdc, MPI_COMM_WORLD);
MPI_Reduce(&buf2, &mx2, 1, MPI_INT, MPI_MAX, rdc, MPI_COMM_WORLD);
if(rank==rdc)
printf("HOTOVO rank:%d max1: %d max2:%d \n",rank, mx1, mx2);
MPI_Finalize();
```

PŘÍKLAD 4: REDUKCE (V KTERÉ POLOVINĚ JE MAXIMUM?)

Pro INT čísla, bez konstant MPI_INT_MIN

```
int buf=(rank*1234)%33;
int buf2=buf;
int min = buf;
MPI_Reduce(&min, &mx1, 1, MPI_INT, MPI_MIN, rdc,
           MPI_COMM_WORLD);

if (rank>=(size/2))
    buf=min;
else
    buf2=min;
```

PŘÍKLAD 5: UŽIVATELEM DEFINOVANÁ OPERACE PRO REDUKCI

Definice operace

```
b[i] = a[i] + b[i] ; a[i] >= 0  
      b[i]          ; jinak
```

```
MPI_Op myOp;
```

```
void myProd(int *a, int *b, int *len,  
            MPI_Datatype *dptr )  
{  
    if (a[0] < 0) b[0] = b[0];  
    else b[0] = a[0] + b[0];  
    return;  
}
```

PŘÍKLAD 5: UŽIVATELEM DEFINOVANÁ REDUKCE

```
MPI_Init(&argc, &argv);
```

```
MPI_Op_create( (MPI_User_function *) myProd, 0, &myOp );
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```
int buf=rank-(size/2);
```

```
printf("Jsem rank:%d data:%d \n",rank, buf);
```

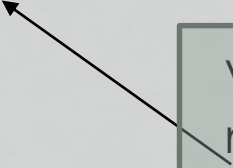
```
MPI_Reduce(&buf,&data, 1,MPI_INT, myOp ,0 ,MPI_COMM_WORLD);
```

```
if(rank==0)
```

```
printf("HOTOVO rank:%d buf: %d data:%d \n",rank, buf, data);
```

```
MPI_Finalize();
```

Vytvoření
nekomutativní
operace z funkce
myProd



PŘÍKLAD 5: UŽIVATELEM DEFINOVANÁ REDUKCE / VÝSTUP

Pro 9 procesů je výstup

```
Jsem rank:6 data:2  
Jsem rank:1 data:-3  
Jsem rank:4 data:0  
Jsem rank:7 data:3  
Jsem rank:8 data:4  
Jsem rank:0 data:-4  
Jsem rank:2 data:-2  
Jsem rank:3 data:-1  
Jsem rank:5 data:1  
HOTOVO rank:0 buf: -4 data:10
```


PŘÍKLAD 5: UŽIVATELEM DEFINOVANÁ REDUKCE / VÝSTUP

Ale pokud změníme data takto: `int buf= - rank+(size/2)`

Dostaneme PROČ?

```
Jsem rank:0 data:4  
Jsem rank:5 data:-1  
Jsem rank:1 data:3  
Jsem rank:2 data:2  
Jsem rank:3 data:1  
Jsem rank:4 data:0  
Jsem rank:6 data:-2  
Jsem rank:8 data:-4  
HOTOVO rank:0 buf: 4 data:6  
Jsem rank:7 data:-3
```

pi.f - compute pi by integrating $f(x) = 4/(1 + x^2)$

```
#include "mpi.h"
```

```
#include <math.h>
```

```
int main(int argc, char *argv[])
```

```
{
```

```
    int done = 0, n, myid, numprocs, i, rc;
```

```
    double PI25DT = 3.141592653589793238462643;
```

```
    double mypi, pi, h, sum, x, a;
```

```
    MPI_Init(&argc, &argv);
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
```



**** CORE**

```
    mypi = h * sum;
```

```
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,  
               MPI_COMM_WORLD);
```

```
    if (myid == 0)
```

```
        printf("pi is approximately %.16f, Error is %.16f\n",  
               pi, fabs(pi - PI25DT));
```

```
}
```

```
MPI_Finalize();
```

```
return 0;
```

**** CORE**

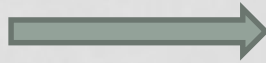
```
while (!done) {
    if (myid == 0) {
        printf("Enter the number of intervals: (0 quits) ");
        scanf("%d",&n);
    }
    MPI_Bcast(&n, 1, MPI_INT , 0, MPI_COMM_WORLD);
    h = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
        x = h * ((double)i - 0.5);
        sum += 4.0 / (1.0 + x*x);
    }
}
```

DALŠÍ KOLEKTIVNÍ OPERACE

- Mezi další kolektivní operace patří
 - **Gather**: data jsou sesbírána a seřazena do posloupnosti ze všech procesů v komunikátoru do kořenového procesu
 - **Scatter**: Data z kořenového procesu jsou rozprostřena mezi ostatní procesy v komunikátoru
 - **Alltoall**: Všechny procesy v komunikátoru rozprostřou svá data navzájem mezi sebou
 - **Scan**: Provede operaci Scan, každý proces dostane svoji hodnotu podle uspořádání v komunikátoru.
- Varianty většiny těchto operací jsou – **toAll** (výsledek je distribuován všem procesům), **I** (asynchronní), **V** (pro Gather, Scatter, proměnná délka distribuovaných dat u každého z procesů)

A1	A2	A3	A4
B1	B2	B3	B4
C1	C2	C3	C4
D1	D2	D3	D4

MPI_BCAST

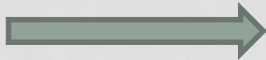


A1			
A1			
A1			
A1			

MPI_GATHER

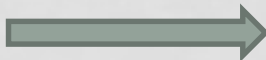


MPI_SCATTER



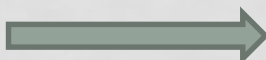
A1			
A2			
A3			
A4			

MPI_ALLGATHER



A1	B1	C1	D1
A1	B1	C1	D1
A1	B1	C1	D1
A1	B1	C1	D1

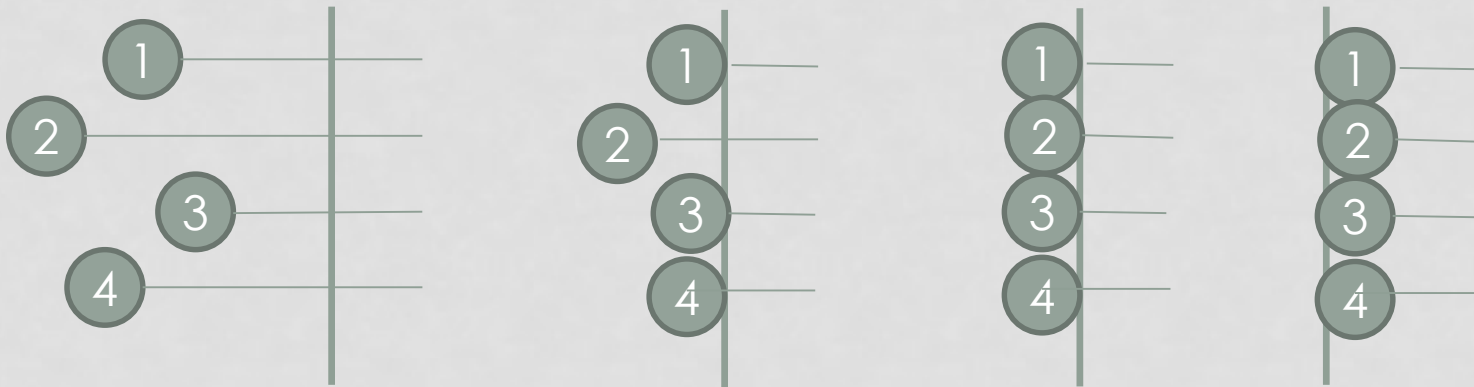
MPI_ALLTOALL



A1	B1	C1	D1
A2	B2	C2	D2
A3	B3	C3	D3
A4	B4	C4	D4

BARIÉRA

- Mezi kolektivní operace patří i **bariéra**.
- Bariéra je synchronizační mechanismus, který je založen na tom, že všechny/část procesů musí tuto operaci provést, než mohou pokračovat.
- Operace pracuje s jediným parametrem a to komunikátorem.



OPERACE S KOMUNIKÁTOREM

- Komunikátor může být
 - zdvojen (duplikován)
 - vytvořen odstraněním některých procesů z existujícího komunikátoru
 - nebo vytvořen jako podmnožina skupin procesů z existujícího komunikátoru

PŘÍKLAD 6: ZDVOJENÍ KOMUNIKÁTORU

```
int rank, size, one, two, sum1, sum2;
```

```
MPI_Comm comm_two;
```

```
MPI_Init(&argc, &argv);
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```
MPI_Comm_dup(MPI_COMM_WORLD, &comm_two);
```

```
one = rank + 2; two = (rank + 2) * 2;
```

```
MPI_Allreduce(&one, &sum1, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
```

```
MPI_Allreduce(&two, &sum2, 1, MPI_INT, MPI_SUM, comm_two);
```

```
printf("%d: Reduced data1=%d and data2=%d to red1=%d and  
red2=%d\n", rank, one, two, sum1, sum2);
```

```
MPI_Comm_free(&comm_two);
```

```
MPI_Finalize();
```


PŘÍKLAD 6: VÝSLEDNÁ REDUKCE

5: Reduced data1=7 and data2=14 to red1=44 and red2=88
6: Reduced data1=8 and data2=16 to red1=44 and red2=88
7: Reduced data1=9 and data2=18 to red1=44 and red2=88
0: Reduced data1=2 and data2=4 to red1=44 and red2=88
1: Reduced data1=3 and data2=6 to red1=44 and red2=88
2: Reduced data1=4 and data2=8 to red1=44 and red2=88
3: Reduced data1=5 and data2=10 to red1=44 and red2=88
4: Reduced data1=6 and data2=12 to red1=44 and red2=88

PŘÍKLAD 7: VYTVOŘENÍ NOVÉHO KOMUNIKÁTORU / ROZDĚLENÍ KOMUNIKÁTORU

- Každý proces se před rozdělením obarví. To znamená, že si jednotlivé skupiny zvolí unikátní hodnotu typu int, a tu přiřadí proměnné zvoleného jména (např int **barva**).

- Po provedení

**MPI_Comm_split(komunikator, barva, rank,
&novy_komunikator);**

- Jsou vytvořeny samostatné komunikátory pro procesy stejných barev. Každý proces má uložen svůj nový komunikátor.

PŘÍKLAD 7: VYTVOŘENÍ NOVÉHO KOMUNIKÁTORU / ROZDĚLENÍ KOMUNIKÁTORU

```
MPI_Init(&argc, &argv);
```

```
int color, rank, nrank;
```

```
MPI_Comm ncomm;
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
color=rank % 4;
```

```
MPI_Comm_split(MPI_COMM_WORLD, color, rank,  
&ncomm);
```

```
MPI_Comm_rank(ncomm, &nrank);
```

```
printf("My rank is %d, my color is %d and new rank id %d\n",  
rank, color, nrank);
```

```
MPI_Finalize();
```

PŘÍKLAD 7: VYTVOŘENÍ NOVÉHO KOMUNIKÁTORU ROZDĚLENÍ KOMUNIKÁTORU, VÝSTUP


- *My rank is 0, my color is 0 and new rank id 0*
- *My rank is 1, my color is 1 and new rank id 0*
- *My rank is 2, my color is 2 and new rank id 0*
- *My rank is 3, my color is 3 and new rank id 0*
- *My rank is 4, my color is 0 and new rank id 1*
- *My rank is 5, my color is 1 and new rank id 1*
- *My rank is 6, my color is 2 and new rank id 1*
- *My rank is 7, my color is 3 and new rank id 1*
- *My rank is 8, my color is 0 and new rank id 2*
- *My rank is 9, my color is 1 and new rank id 2*

VYTVOŘENÍ NOVÉHO KOMUNIKÁTORU / SKUPINY PROCESŮ

- Nový komunikátor lze vytvořit z procesů, které se v systému nachází v rámci nějakého komunikátoru
- Postup
 1. Vytvoření skupiny procesů z existujících komunikátorů
 2. Přidání, odstranění procesů z/do skupiny
 3. Vytvoření nového komunikátoru ze skupiny

PŘÍKLAD 8: VYTVOŘENÍ SKUPINY PROCESŮ

```
MPI_Comm comm_world, comm_worker;  
MPI_Group group_world, group_worker;  
comm_world = MPI_COMM_WORLD;  
MPI_Comm_group(comm_world, &group_world);
```




Vytvoří skupinu
MPI_Group ze všech
procesů v daném
komunikátoru

PŘÍKLAD 8: MODIFIKACE SKUPINY PROCESŮ

```
int lst[]={0,1,2,3};
```


```
int sz=sizeof(lst)/sizeof(int);
```

```
MPI_Group_excl(group_world, sz, lst, &group_worker);
```



Odstranění procesů s ranky
0,1,2 a 3 ze skupiny
procesů

```
MPI_Comm_create(comm_world, group_worker, &comm_worker);
```



Vytvoření komunikátoru ze
zbývajících procesů ve
skupině

PŘÍKLAD 8: VYTVOŘENÍ KOMUNIKÁTORU ZE SKUPINY, VÝSTUP

HOTOVO rank:7 data:104 nrank: 3
HOTOVO rank:8 data:104 nrank: 4
HOTOVO rank:9 data:104 nrank: 5
HOTOVO rank:10 data:104 nrank: 6
HOTOVO rank:11 data:104 nrank: 7
HOTOVO rank:0 data:100 nrank: 999
HOTOVO rank:1 data:101 nrank: 999
HOTOVO rank:2 data:102 nrank: 999
HOTOVO rank:3 data:103 nrank: 999
HOTOVO rank:4 data:104 nrank: 0
HOTOVO rank:5 data:104 nrank: 1
HOTOVO rank:6 data:104 nrank: 2

Nejsou v novém komunikátoru, nrank je na defaultní 999, v datech mají svůj rank +100, Ostatní mají nový rank a v datech původní rank + 100 jejich roota

INTERKOMUNIKÁTOR

- Máme skupiny procesů v komunikátorech, vzniklé dejme tomu z `MPI_COMM_WORLD`
- Tyto skupiny lze propojit Interkomunikátorem tak, že
- Každá skupina
 - pasuje jeden ze svých procesů za 'lídra' pro interkomunikaci
 - vytvoří interkomunikátor příkazem

int MPI_Intercomm_create(

lokální intrakomunikátor,

pověřený lídr,

partnerský intrakomunikátor, // `MPI_COMM_WORLD`

lídr partnerské skupiny,

tag,

& výsledný interkomunikátor)

INTERKOMUNIKÁTOR

Vytvoříme nové
komunikátory pro tři
skupiny procesů

0	1	2	3	4	5	6	7	8	9	10	11
---	---	---	---	---	---	---	---	---	---	----	----

0	1	2	3	4	5	6	7	8	9	10	11
---	---	---	---	---	---	---	---	---	---	----	----

0

3

6

9

1

4

7

10

2

5

8

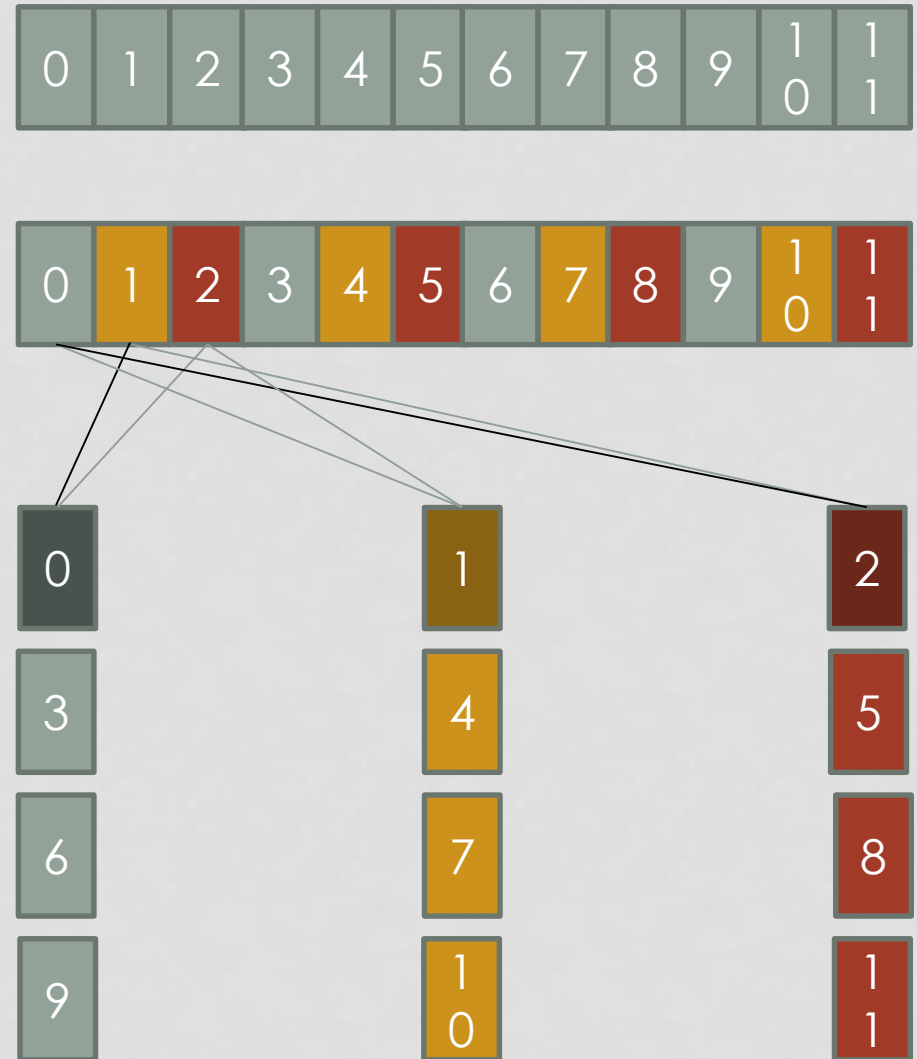
11

INTERKOMUNIKÁTOR

Vytvoříme interkomunikátory mezi každou dvojicí skupin procesů:
Každý proces uvede svého **lídra**, partnerský komunikátor a lídra partnerského komunikátoru.
V tomto případě je partnerský komunikátor **MPI_COMM_WORLD**

Např mezi skupinou první a druhou takto:

```
MPI_Intercomm_create(  
  myComm, // první komunikátor  
  0,       // lídr prvního  
  MPI_COMM_WORLD,  
  1,       // lídr druhého ve 'WORLD'  
  1,       // tag  
  &myFirstComm); // interkomunikátor
```



PŘÍKLAD 9: INTERKOMUNIKÁTOR – KOLEKTIVNÍ OPERACE

- Kolektivní operaci musí provést všechny procesy z obou skupin
- Root je proces 0 v první skupině.
- Root se uvede jako **MPI_ROOT**
- Ostatní v jeho skupině jako **MPI_PROC_NULL**
- Všechny procesy v druhé skupině uvedou číslo roota jak jej má v první skupině (tedy **0**)

```
if(group==0){
    if(rank==0){
        MPI_Reduce(&value,&res,1,MPI_INT, MPI_SUM, MPI_ROOT,myFirstComm);
        printf("%u\n",res);
    }
    else
        MPI_Reduce(&value,&res,1,MPI_INT, MPI_SUM, MPI_PROC_NULL, myFirstComm);
}

if(group==1)
    MPI_Reduce(&value,&res,1,MPI_INT, MPI_SUM, 0,myFirstComm);
```

ASYNCHRONNÍ KOMUNIKACE

- Asynchronní (neblokující) komunikaci, tj. operace typu `send` nezastaví činnost procesu do přijetí zprávy adresátem
- Mimo jiné asynchronní komunikace zamezí uvážnutí z důvodu nesprávného použití blokujících komunikačních primitiv –

proces p1

```
send(p2, &mssg1, l1)  
receive(p2, &bufp1)
```

proces p2

```
send(p1, &mssg2, l2)  
receive(p1, &bufp2)
```

ASYNCHRONNÍ KOMUNIKACE

- Primitiva odeslání a přijetí zprávy **isend**, **ireceive** provádí neblokující zasílání a přijímání zpráv. Jejich struktura je následující:

int MPI_Isend(**const void** *buf, **int** count, **MPI_Datatype** datatype, **int** dest, **int** tag, **MPI_Comm** comm, **MPI_Request** *request)

(Položka Request identifikuje komunikaci zahájenou touto operací)

int MPI_Irecv(**void** *buf, **int** count, **MPI_Datatype** datatype, **int** source, **int** tag, **MPI_Comm** comm, **MPI_Status** *status)

ASYNCHRONNÍ KOMUNIKACE

Po zahájení asynchronní komunikace mohou procesy přizpůsobit svůj běh jejímu průběhu. K tomu slouží operace **Wait** a **Test**.

- **Wait** - operace čekání na zprávu. Pozastaví činnost, než je komunikace dokončena a naplní **status** podle výsledku komunikace.

```
int MPI_Wait(MPI_Request *request, MPI_Status  
             *status);
```

- **Test** - testování stavu neblokuje činnost procesu, pouze naplní **status** podle výsledku komunikace

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


PŘÍKLAD 10: ASYNCHRONNÍ KOMUNIKACE / ČÁST 1

```
MPI_Status status;  
MPI_Request send_request,recv_request;  
int rank, size, ierr;  
static int bufsize=65535000;  
double *sendbuff,*recvbuff, inittime, sendtimef, sendtime, recvtime,  
recvtimef, totaltime;
```

```
MPI_Init(&argc, &argv);
```

```
MPI_Comm_rank(MPI_COMM_WORLD,&rank);  
MPI_Comm_size(MPI_COMM_WORLD,&size);
```

```
sendbuff=(double *)malloc(sizeof(double)*bufsize);  
recvbuff=(double *)malloc(sizeof(double)*bufsize);
```

```
srand((unsigned)time( NULL ) + rank);
```

```
for(int i=0;i<bufsize;i++){  
    sendbuff[i]=2*i;  
    recvbuff[i]=-1;  
}
```

```
inittime = MPI_Wtime();
```

PŘÍKLAD 10: ASYNCHRONNÍ KOMUNIKACE / ČÁST 2

```
if ( rank == 0 ){
    ierr=MPI_Isend(sendbuff,buffsize,MPI_DOUBLE,
                  1,0,MPI_COMM_WORLD,&send_request);
    sendtime = MPI_Wtime();
    printf("send started (%f):\n\t [500]...%f\n\t[500000]...%f\n\t\n", sendtime, sendbuff[500], sendbuff[500000]);

    ierr=MPI_Wait(&send_request, &status);
    sendtimef = MPI_Wtime();
    printf("send finished (%f):\n\t [500]...%f\n\t [500000]...%f\n\t\n", sendtimef, sendbuff[500], sendbuff[500000]);
}

else if( rank == 1 ){
    ierr=MPI_Irecv(recvbuff,buffsize,MPI_DOUBLE,
                  0,MPI_ANY_TAG,MPI_COMM_WORLD,&recv_request);
    recvtime = MPI_Wtime();
    printf("recv started (%f):\n\t [500]...%f\n\t [500000]...%f\n\t\n", recvtime, sendbuff[500], sendbuff[500000]);

    ierr=MPI_Wait(&recv_request, &status);
    recvtimef = MPI_Wtime();
    printf("recv finished (%f):\n\t [500]...%f\n\t [500000]...%f\n\t\n",recvtimef, sendbuff[500], sendbuff[500000]);

    totaltime = finishtime - inittime;
    printf(" Communication time : %f - %f = %f seconds\n\n",finishtime, inittime, totaltime);
}

free(recvbuff);
free(sendbuff);

MPI_Finalize();
```

PŘÍKLAD 10: ASYNCHRONNÍ KOMUNIKACE, VÝSTUP

- recv started (1489934608.484541):
 - [500]...1000.000000
 - [500000]...1000000.000000
- send started (1489934608.556497):
 - [500]...1000.000000
 - [500000]...1000000.000000
- recv finished (1489934608.649075):
 - [500]...1000.000000
 - [500000]...1000000.000000
- send finished (1489934608.649075):
 - [500]...1000.000000
 - [500000]...1000000.000000
- #####
- Communication time : 1489934608.649075 - 1489934608.484532 = 0.164543 seconds
- #####

OPERACE ČEKÁNÍ NA VÍCERÉ UKONČENÍ ASYNCHRONNÍCH OPERACÍ

- Operace čekání mohou být použity ve formě, kdy čekají na ukončení skupny asynchronních operacích, daných v poli 'requestů'.
- Mohou také čekat na ukončení jen jedné takové operace z uvedených

MPI_Waitall(count, array_of_requests, array_of_statuses)

MPI_Waitany(count, array_of_requests, &index, &status)

MPI_Waitsome(count, array_of_requests, array_of indices, array_of_statuses)

// statusy všech ukončených operací z pole requestů

NEBLOKUJÍCÍ KOLEKTIVNÍ OPERACE

- Neblokující operace, stejně jako asynchronní P2P komunikace, je provedena bez dalšího čekání.
- Neblokující broadcast a reduce vypadají následovně:
- `MPI_IBCAST(buffer, count, datatype, root, comm, request)`
- `MPI_IREDUCE(sendbuf, recvbuf, count, datatype, op, root, comm, request)`
- Request identifikuje započatou operaci
- Pro řízení běhu na základě kolektivního dokončení těchto operací se opět používají `Wait` a `Test`
- Dokonce i bariéra má svojí neblokující verzi!

`int MPI_IBARRIER(comm, request)`

... a to si zaslouží příklad

PŘÍKLAD 11: NEBLOKUJÍCÍ BARIÉRA

```
MPI_Status status;  
MPI_Request request;  
int rank, done;  
MPI_Init(&argc, &argv);  
MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
MPI_Ibarrier(MPI_COMM_WORLD, &request);  
printf("jsem tu po bariere!! %i\n", rank);  
MPI_Test(&request, &done, &status);  
printf("To jsem jeste ja, %i, to done je %i! \n", rank, done);  
MPI_Wait(&request, &status);  
MPI_Test(&request, &done, &status);  
printf("To jsem zase jeste ja, %i, to done je nyní %i!\n", rank, done);  
MPI_Finalize();
```

PŘÍKLAD: NEBLOKUJÍCI BARIÉRA, VÝSTUP

jsem tu po bariere!! 1

To jsem jeste ja, 1, to done je 0!

jsem tu po bariere!! 2

To jsem jeste ja, 2, to done je 0!

jsem tu po bariere!! 3

To jsem jeste ja, 3, to done je 0!

jsem tu po bariere!! 0

To jsem jeste ja, 0, to done je 0!

jsem tu po bariere!! 4

To jsem jeste ja, 4, to done je 0!

To jsem zase jeste ja, 4, to done je vcil 1!

To jsem zase jeste ja, 3, to done je vcil 1!

To jsem zase jeste ja, 1, to done je vcil 1!

To jsem zase jeste ja, 2, to done je vcil 1!

To jsem zase jeste ja, 0, to done je vcil 1!

SPUŠTĚNÍ NOVÝCH PROCESŮ

- Lze vytvořit interkomunikátor mezi skupinou již běžících procesů a skupinou procesů, které budou spuštěny. Příkaz je kolektivní a blokující, dokud všechny potomkovské procesy neprovedou inicializaci
- **MPI_Comm_Spawn**(příkaz, &argv[], maxproc, info, root, comm, &ncomm, errs[])
 - **Vstupy**
 - **příkaz** – './slave'
 - **maxproc** – kolik procesů (pokud možno) spustit
 - **info** – parametry, kolekce dvojic klíč / hodnota, nebo **MPI_INFO_NULL**
 - **root** – od kterého procesu se mají brát předchozí argumenty
 - **comm** – intrakomunikátor s původními procesy
 - **Výstupy**
 - **ncomm** – interkomunikátor mezi původními a novými procesy
 - **errs** – pole chybových kódů