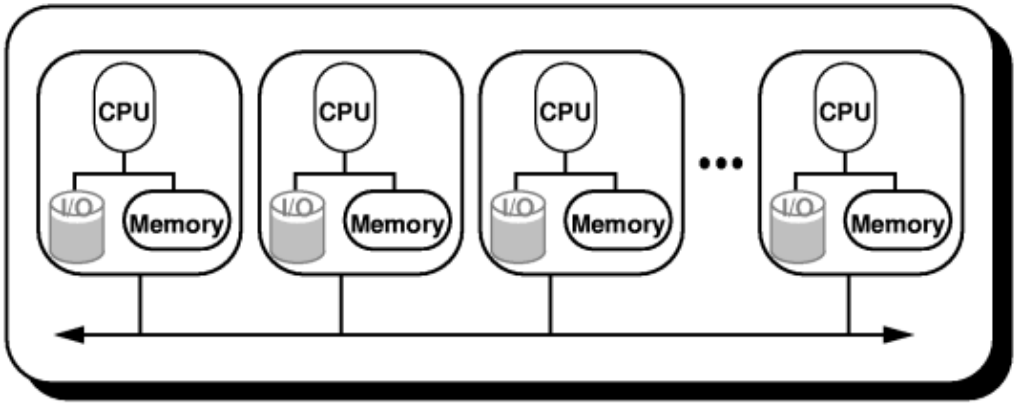
MPI

Message Passing Interface

Systémy s distribuovanou pamäťou

- Autonómne procesory s vlastnou pamäťou prepojené komunikačnou sieťou



- Komunikácia realizovaná posielaním správ
- Procesory sú typicky od seba fyzicky vzdialené (clustre, BOINC, „superpočítače“, …)

- HPC = **High Performance Computing**

Portabilita softvéru?

- Špecializovaný hardvér –
  rôzne architektúry, rôzne špecifiká
- Ethernet vs. Infiniband
- Sieťová topológia vs. sieťový overhead

Message Passing Interface

- API, ktoré umožňuje procesom komunikovať prostredníctvom odosielania a prijímania správ
- používa sa pri vývoji paralelných programov pre superpočítače a clustre
- ciele:
  - použiteľnosť, portabilita, efektívnosť, flexibilnosť
- vznikla v roku 1994 ako výsledok dohody 40 organizácií združených v MPI fórum
- www.mcs.anl.gov/research/projects/mpi/
- www.mpi-forum.org

Message Passing Interface

- MPI nie je implementácia, resp. nejaká knižnica, je to len špecifikácia API
- štandardizované API umožňuje vývoj portabilných paralelných programov
- výrobcovia hardvéru vytvárajú implementácie MPI optimalizované pre svoj hardvér
- známe implementácie:
  - MPICH [`www.mpich.org`], LAM/MPI, WMPI, OpenMPI [`www.open-mpi.org`], Intel MPI
  - MS MPI (optimalizované pre Win HPC Server 2008)
  - MPJ Express [www.mpj-express.org]

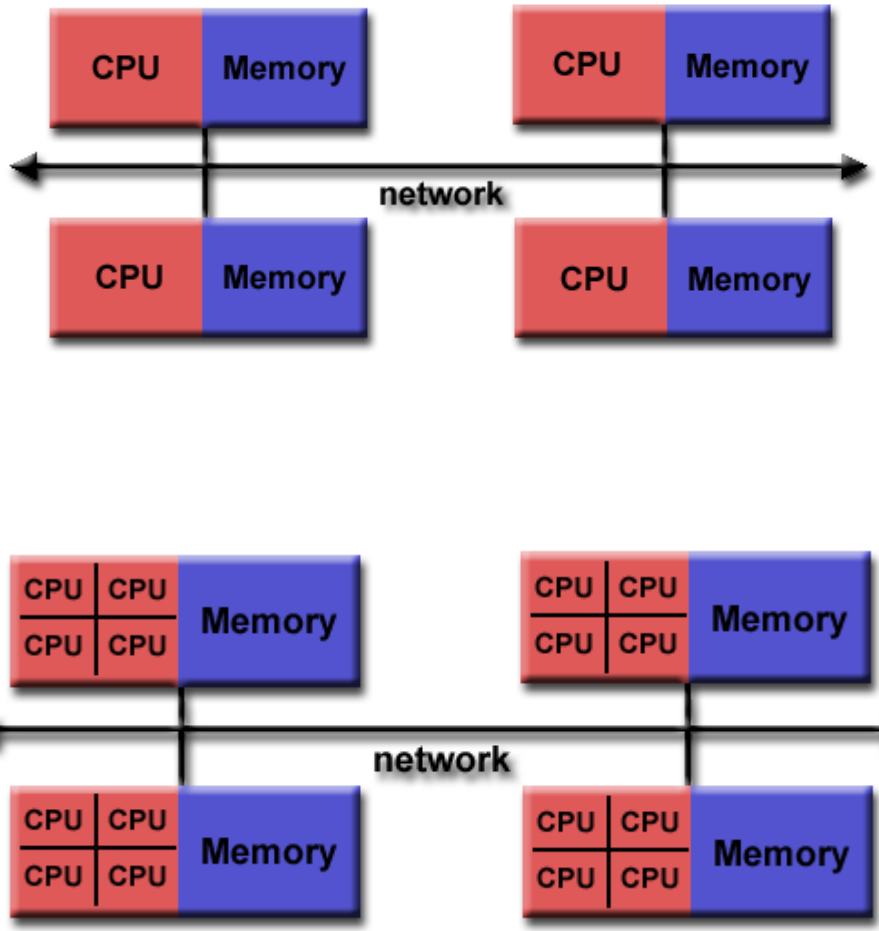
MPI & Java

- objektovo-orientované Java rozhranie pre štandardizované MPI (1997)
- implementácie:
  - mpiJava – Java wrapper (cez JNI) na MPICH implementáciu
    - `www.hpjava.org/mpiJava.html`
  - MPJ Express – čistá Java implementácia
    - `mpj-express.org`
      - multicore konfigurácia
      - cluster konfigurácia (Java NIO, Myronet)
  - F-MPJ – fast MPJ
    - `gac.udc.es/~rreye/fastmpj`

MPI & Python

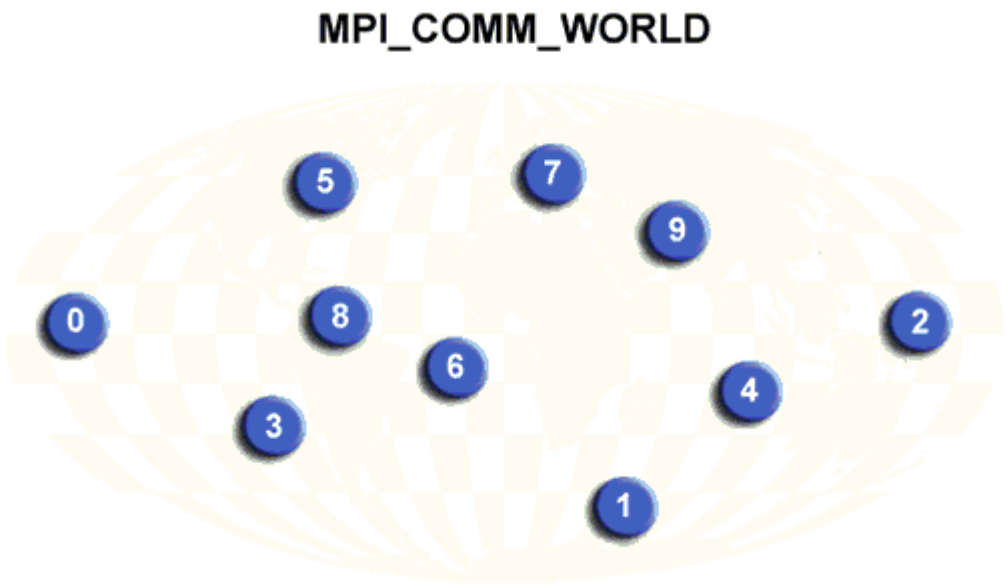
- *Distributed Parallel Programming for Python* (PyMPI*)*
  - sourceforge.net/projects/pympi/ *(2013)*
- *Scientific Python*
  - github.com/khinsen/ScientificPython *(2014)*
- *MPI for Python* package (mpi4py)
  - mpi4py.readthedocs.io/en/stable/ (2022)

Podporované modely



Základné koncepty

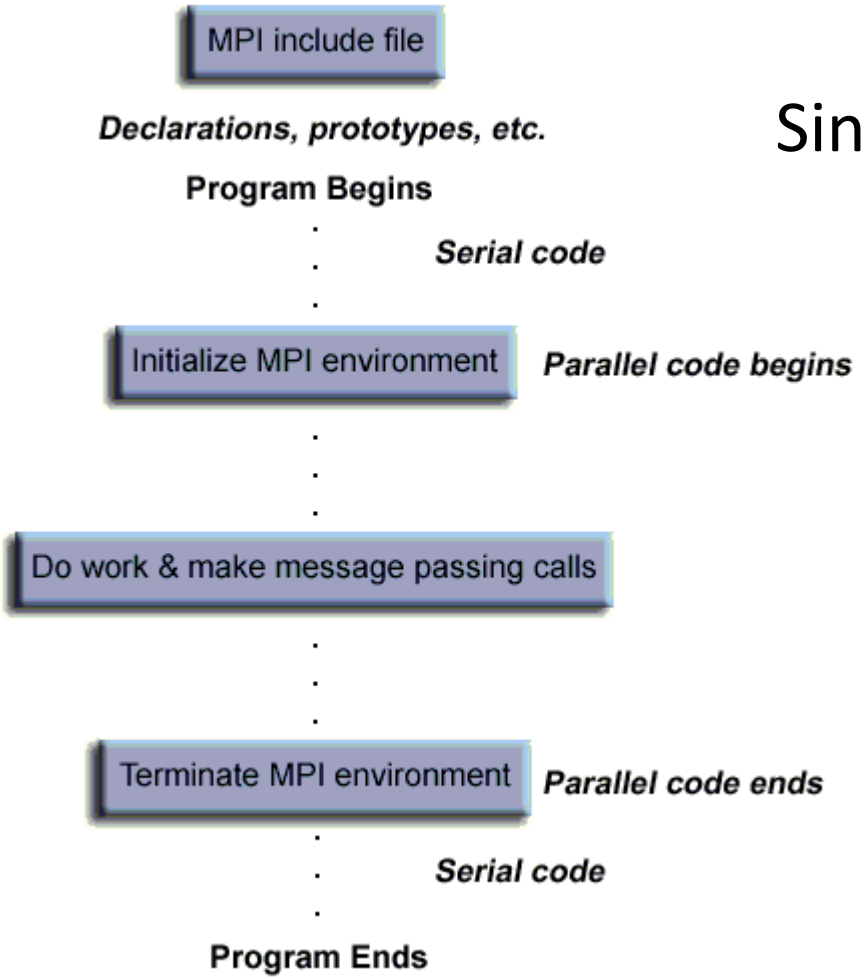
- **Job** – úloha, ktorá má byť realizovaná paralelným výpočtom zadaným počtom taskov (procesov)
- **Task** – proces, ktorý realizuje nejaký výpočet
  - každý proces má ID: 0, …, N-1



Komunikátory a skupiny

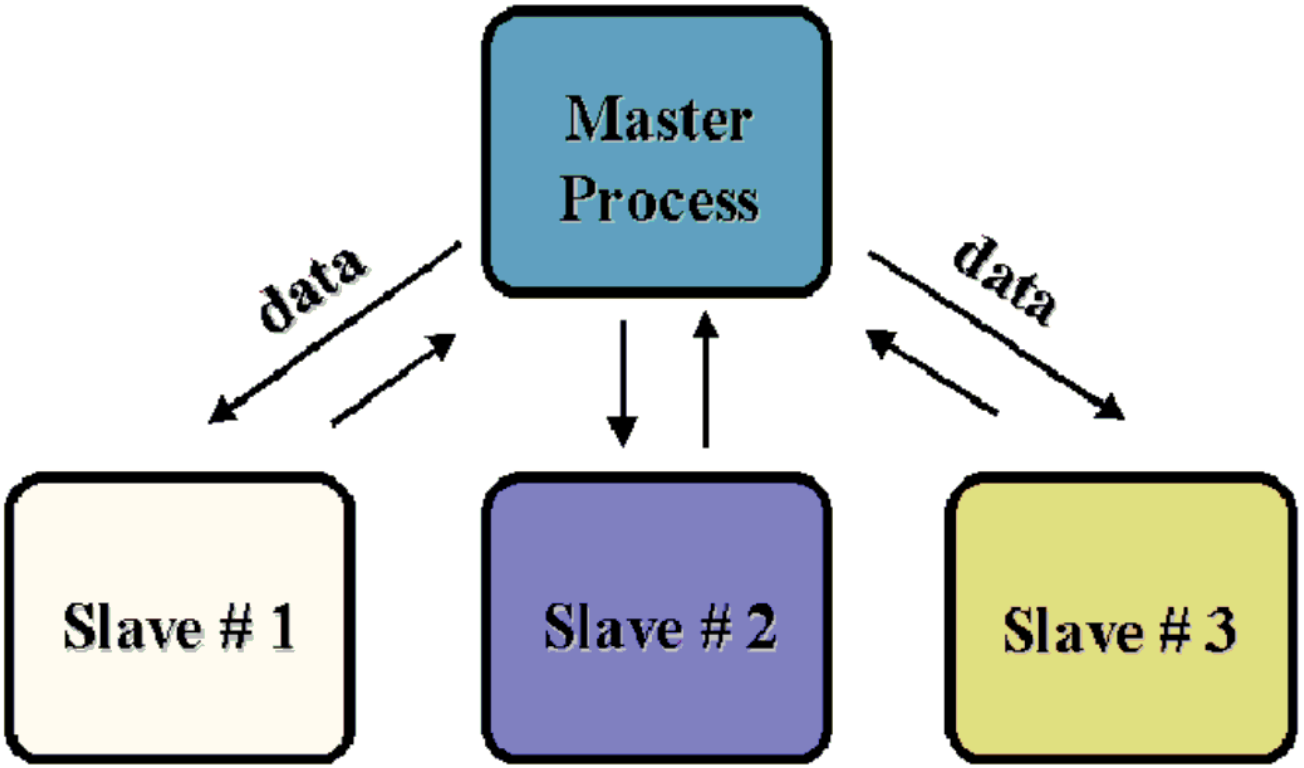
- **Skupina** (group)
  - nejaká podmnožina vytvorených procesov
- **Komunikátor** (Comm)
  - umožňuje komunikáciu v rámci skupiny procesov
  - každý proces je v rámci komunikátora identifikovaný svojim rank-om
  - MPI.COMM_WORLD je komunikátor pre skupinu tvorenú všetkými procesmi

Štruktúra programu



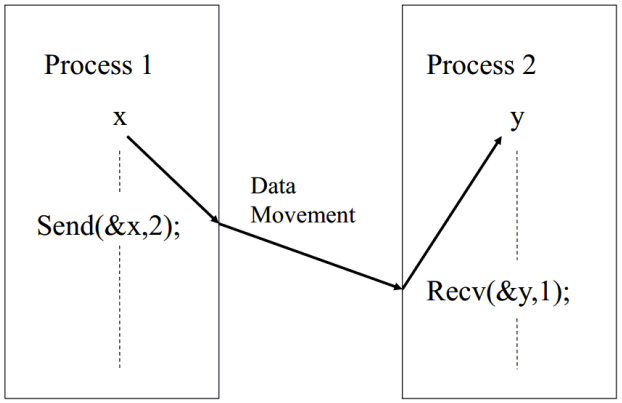
SPMD =
Single Program, Multiple Data

Master / Slave-Worker



Komunikácia

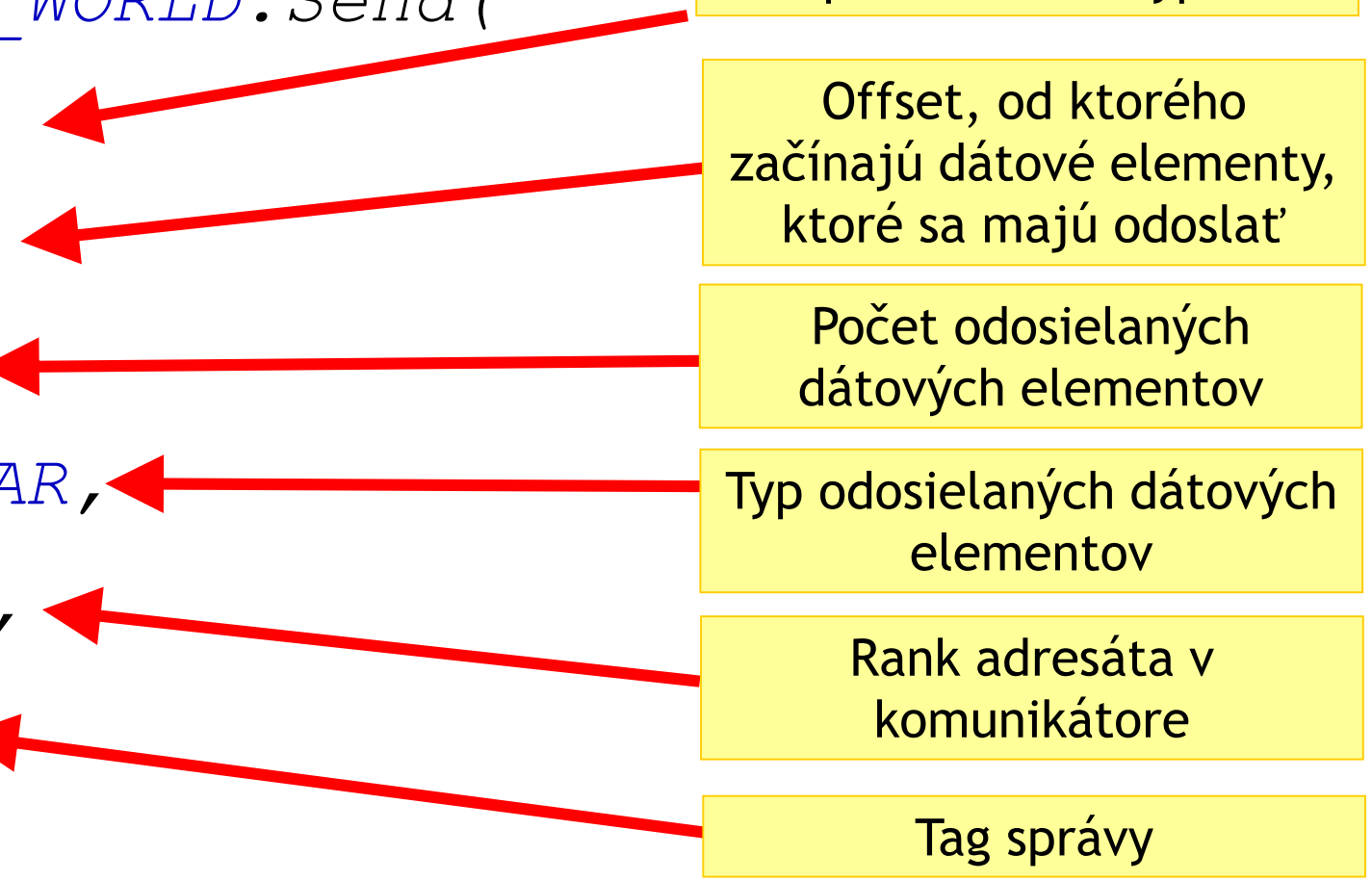
- Point-to-point komunikácia
  - proces pošle správu inému procesu v rámci komunikátora



- Kolektívna komunikácia
  - uskutoční sa komunikačná rutina, na ktorej participujú všetky procesy komunikátora
  - broadcasting, redukcie, ...

Point-to-point komunikácia

```
char[] buffer = ...
MPI.COMM_WORLD.Send(
   buffer,
   offset,
   count,
   MPI.CHAR,
   target,
   tag);
```



Mapovanie typov elementov

MPI datatype	C++	Java	Python (numpy.dtype)
MPI.BYTE	signed char	byte	'b'
MPI.CHAR	char	char	'|S1'
MPI.BOOL	bool	boolean	'?'
MPI.SHORT	signed short	short	'i2' int16
MPI.INT	signed int	int	'i4' int32
MPI.LONG	signed long	long	'i8v int64
MPI.FLOAT	float	float	'f4' float32
MPI.DOUBLE	double	double	'f8', 'd' float64
MPI.COMPLEX			'c'

www.mpi-forum.org

Príjem správy

```
Status Comm.Recv(Object buf,
    int offset,
    int count,
    Datatype datatype,
    int source,
    int tag)
```

Prijímací buffer je určený ako časť (určená offsetom a parametrom count) poľa hodnôt primitívneho typu

Typ odosielaných dátových elementov

Rank odosielateľa (alebo MPI.SOURCE_ANY)

Tag prijímanej správy (alebo MPI.TAG_ANY)

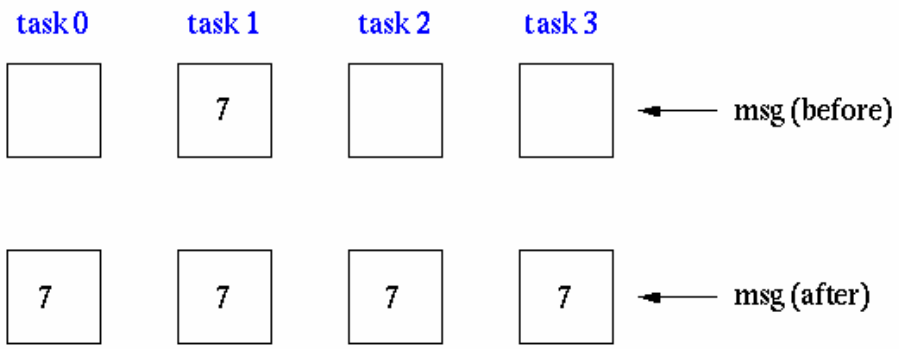
Zo Status objektu sa vieme dozvedieť skutočný počet prijatých dátových elementov, odosielateľa a tag správy.

Typy point-to-point komunikácie

- blokované vs. neblokované
  - pri blokovaných operáciach sa čaká na dokončenie operácie (napr. pri odosielaní sa čaká, kým správa bude odoslaná; po ukončení blokovanej operácie je možné znovupoužívať buffre) … (napr. `MPI_Isend`)
- synchrónne vs. asynchrónne
  - pri synchrónnych operáciach máme potvrdenie prijatia správy príjemcom (napr. `MPI_Ssend`)
- `Send` a `Recv` sú asynchrónne blokované.

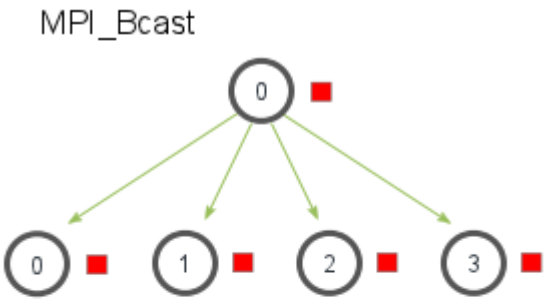
Kolektívna komunikácia (1)

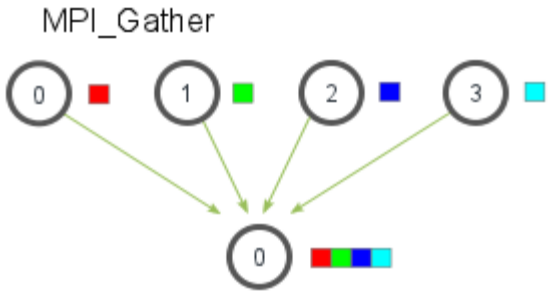
- MPI_Barrier (Barrier)
  - blokuje vykonávanie procesu dokiaľ všetky procesy v komunikátore nevykonajú túto operáciu
- MPI_Bcast (Bcast)
  - pošle správu všetkým procesom v komunikátore, resp. prijme broadcastovanú správu

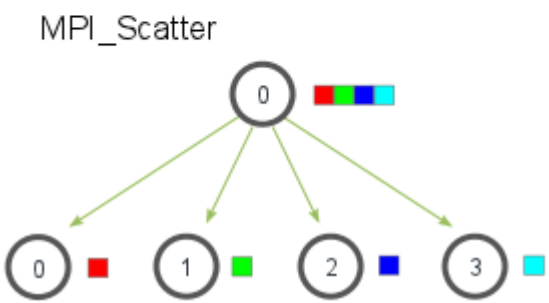


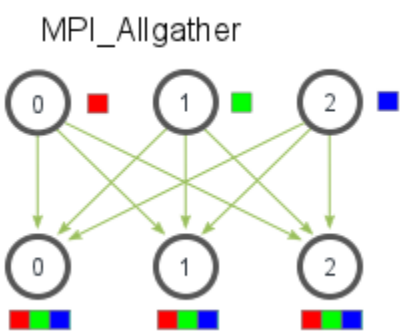
Kolektívna komunikácia (2)

- Broadcast
- Scatter
- Gather
- Allgather
- **Reduce**
- Allreduce
- Alltoall
- Scan



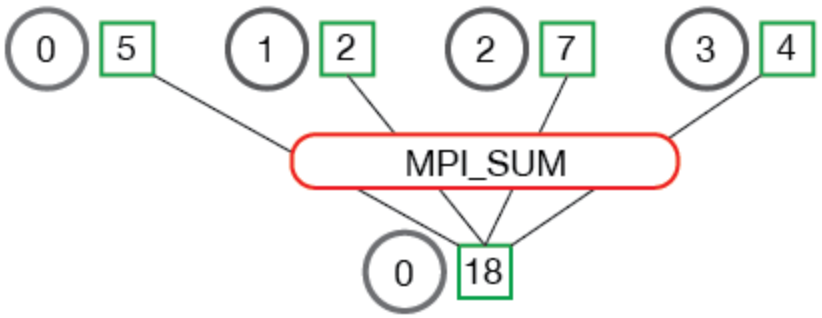




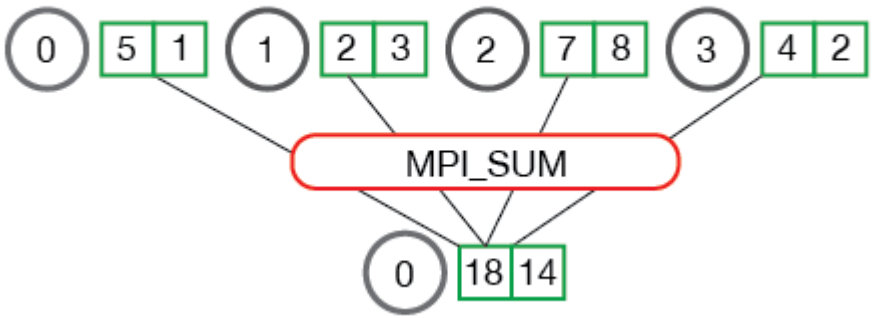


MPI Reduce

MPI_Reduce



MPI_Reduce



SUM, PROD,

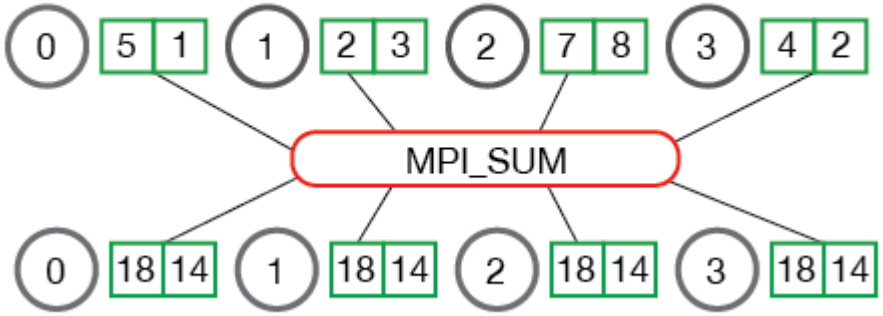
MIN, MAX,

MINLOC, MAXLOC

LAND, LOR, LXOR

BAND, BOR, BXOR

MPI_Allreduce

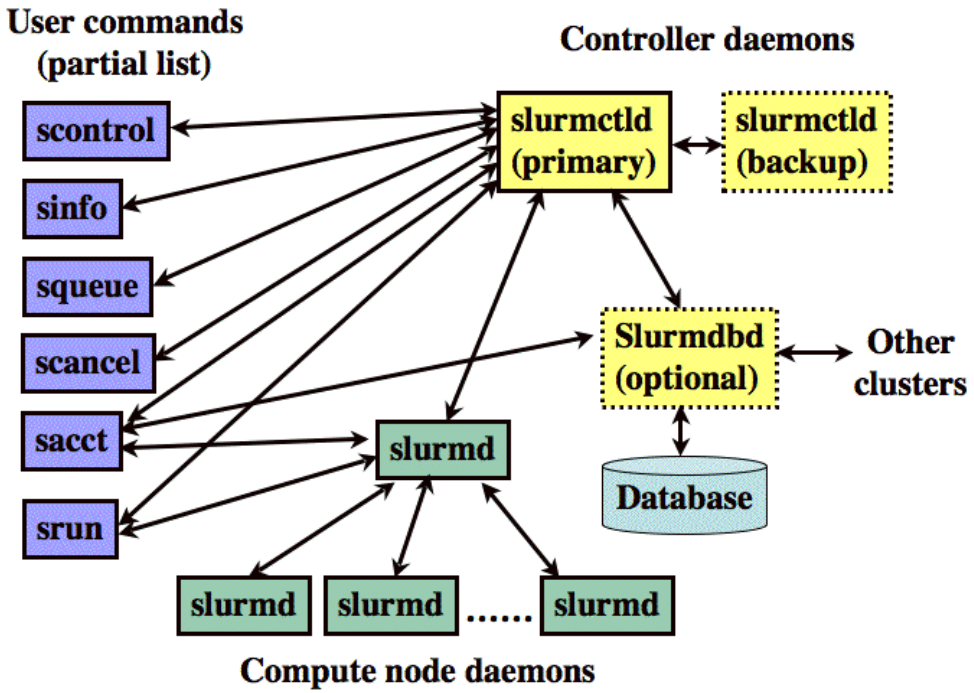


Ďalšie charakteristiky

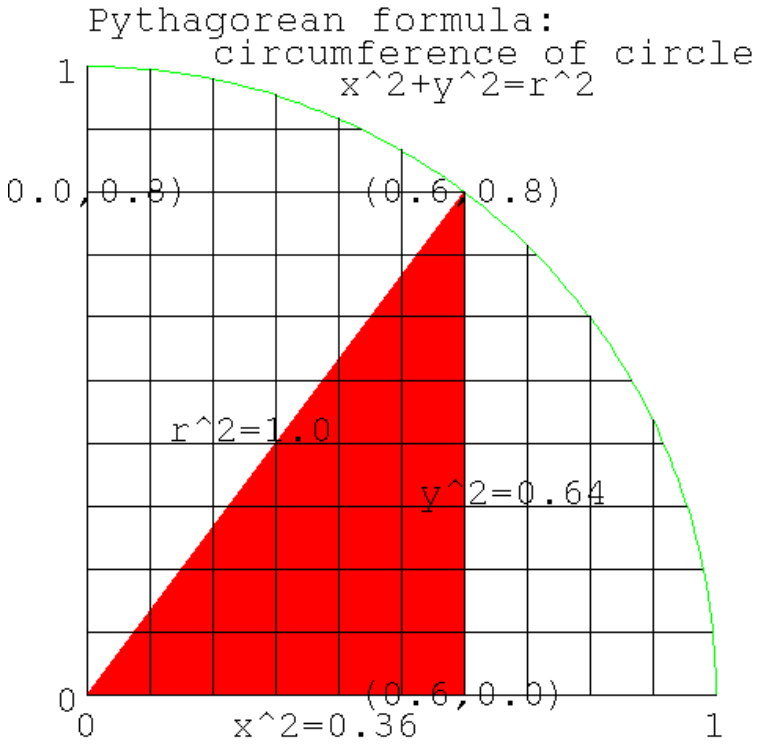
- MPI umožňuje vytvárať vlastné skupiny z existujúcich skupín
  - ku každej skupine možno vytvoriť komunikátor, ktorý je užitočný v prípade kolektívnej komunikácie
- MPI umožňuje definovať virtuálne topológie procesov
  - mriežka, graf
  - MPI implementácia môže túto informáciu využiť na mapovanie procesov na fyzické procesory

Manažment zdrojov

- **SLURM** - Simple Linux Utility for Resource Management
  - Štartovanie, vykonávanie a monitorovanie paralelných jobov (napr. MPI)
  - Prideľovanie výpočtových uzlov používateľom



Numerické integrovanie a PI



Ukážka v Jave

```
import mpi.*;

public class AhojSvet {
   public static void main(String[] args) {
      MPI.Init(args);
      int size = MPI.COMM_WORLD.Size();
      int me = MPI.COMM_WORLD.Rank();
      System.out.println("Hi from <"+me+">");
      MPI.Finalize();
   }
}
```

- počet procesov je určený pri spustení programu (argument –np)

```
export MPJ_HOME=~/mpj-v0_44
javac -cp ".:${MPJ_HOME}/lib/mpj.jar" AhojSvet.java
java -jar "${MPJ_HOME}/lib/starter.jar" -np 4 AhojSvet
```

Ukážka v Pythone

```
from mpi4py import MPI

size = MPI.COMM_WORLD.Get_size()
rank = MPI.COMM_WORLD.Get_rank()
pname = MPI.Get_processor_name()

print(f"Hello from {pname} - {rank}/{size}")
```

- spustenie

```
mpiexec -n 4 python3 -m mpi4py mpi.py3
```

Ukážka v Pythone

- ***Broadcast Python object***

```
data = MPI.COMM_WORLD.bcast(
    {'pds': 7} if rank == 2 else None,
    root = 2)
print(data)
```

- ***Broadcast NumPy array***

```
n = np.full(1,
            7 if rank == 2 else 0,
            dtype = int32)
MPI.COMM_WORLD.Bcast((n, MPI.INT),
             root = 2)
print(n[0])
```

Ďakujem za pozornosť !