Úvod do MPI

Úvod do MPI

Programování architektur založených na posílání zpráv:

- výpočet se skládá ze skupiny procesů běžících na jednotlivých procesorech
- synchronizace probíhá pomocí posílání zpráv
- to je vhodné pouze pro nepříliš úzce synchronní výpočty

Standard MPI

Standard pro posílání zpráv - MPI = Message Passing Interface
Dostupné implementace

- OpenMPI - `http://www.open-mpi.org/`
- LAM-MPI - `http://www.lam-mpi.org/`
- MPICH - `http://www-unix.mcs.anl.gov/mpi/mpich/`
- Intel, HP, ...

Zdroje na internetu:
`http://www-unix.mcs.anl.gov/mpi/`
Wikipedia

Základ kódu pro MPI

```
1  #include <mpi.h>
2
3  int main( int argc, char* argv[] )
4  {
5     MPI_Init( argc, argv );
6
7     ...
8
9     MPI_Finalize();
10 }
```

Funkce `MPI_Init` a `MPI_Finalize` musí být volány právě jednou a všemi procesy. Vrácená hodnota by měla být `MPI_SUCCES`.

Komunikační skupiny

Komunikační skupiny určují, které procesy se budou účastnit zvolené operace.

Jde o tzv. COMMUNICATORS s typem `MPI_Comm`.

Všechny běžící procesy jsou obsaženy ve skupině `MPI_COMM_WORLD`.

Informace o procesech

Počet procesů (`size`) v dané skupině lze zjistit pomocí:

```
1  int MPI_Comm_size( MPI_Comm comm, int* size );
```

Identifikační číslo procesu (`rank`) vůči dané skupině lze zjistit pomocí:

```
1  int MPI_Comm_rank( MPI_Comm comm, int* rank );
```

Operace send a recieve

Jde o základní operace pro odeslání a přijetí jedné zprávy.

- `send( void *sendbuf, int nelems, int dest)`
  - `sendbuf` - ukazatel na pole dat, jež mají být odeslána
  - `nelems` - počet prvků pole
  - `dest` - ID procesu/uzlu, kterému mají být data zaslána
- `receive( void *recvbuf, int nelems, int source)`
  - `recvbuf` - ukazatel na pole, do nějž mají být přijatá data uložena
  - `nelems` - počet prvků, které budou přijaty
  - `source` - ID procesu/uzlu, od kterého mají být data načtena

Operace send a recieve

Příklad:

P_1	P_2
`a=100;`	
`send(&a,1,2);`	`receive( &a,1,1);`
`a=0;`	`printf("%d",a);`

Jaký výsledek vypíše proces P_2?

Operace send a recieve

Příklad:

P_1	P_2
`a=100;`	
`send(&a,1,2);`	`receive( &a,1,1);`
`a=0;`	`printf("%d",a);`

Jaký výsledek vypíše proces P_2?
Výsledek je `100` nebo `0`.

Operace send a recieve

- síťové rozhraní může využívat DMA a pracovat nezávisle na CPU
- pokud má P_2 zpoždění ve zpracování kódu, mohou být data z P_1 odeslána až po provedení příkazu `a=0;` na P_1
- tím pádem bude procesu P_2 odesláno číslo `0`

Blokující a neblokující posílání zpráv

Existují dva způsoby posílání zpráv:

- blokující
  - funkce `send` nevrátí řízení programu, dokud to není sémanticky bezpečné
- neblokující
  - komunikační funkce vrací řízení programu dříve, než je to sémanticky bezpečné

Blokující posílání zpráv

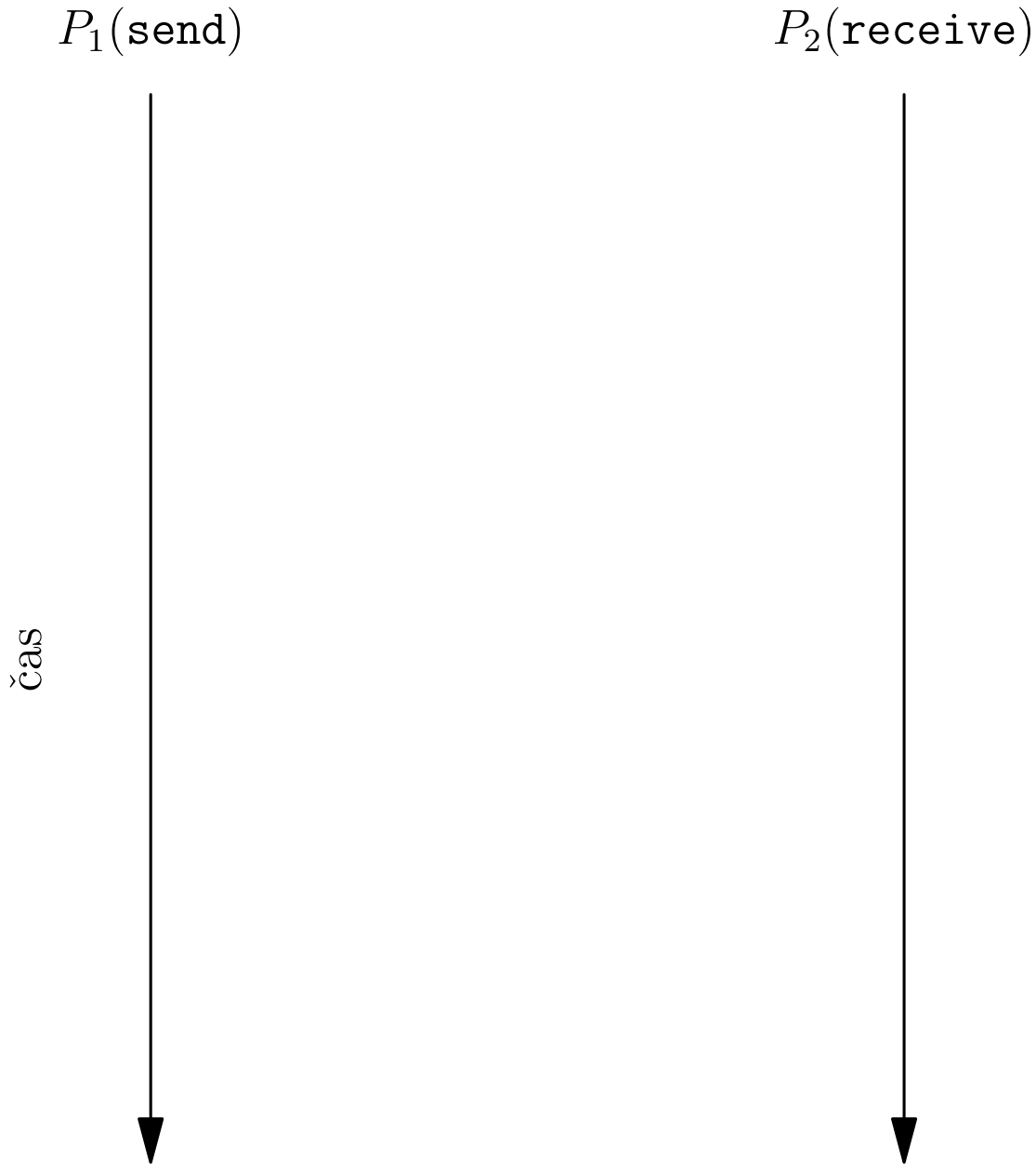
Blokující posílání zpráv je možné provést dvěma způsoby:

- bez bufferu
- s bufferem

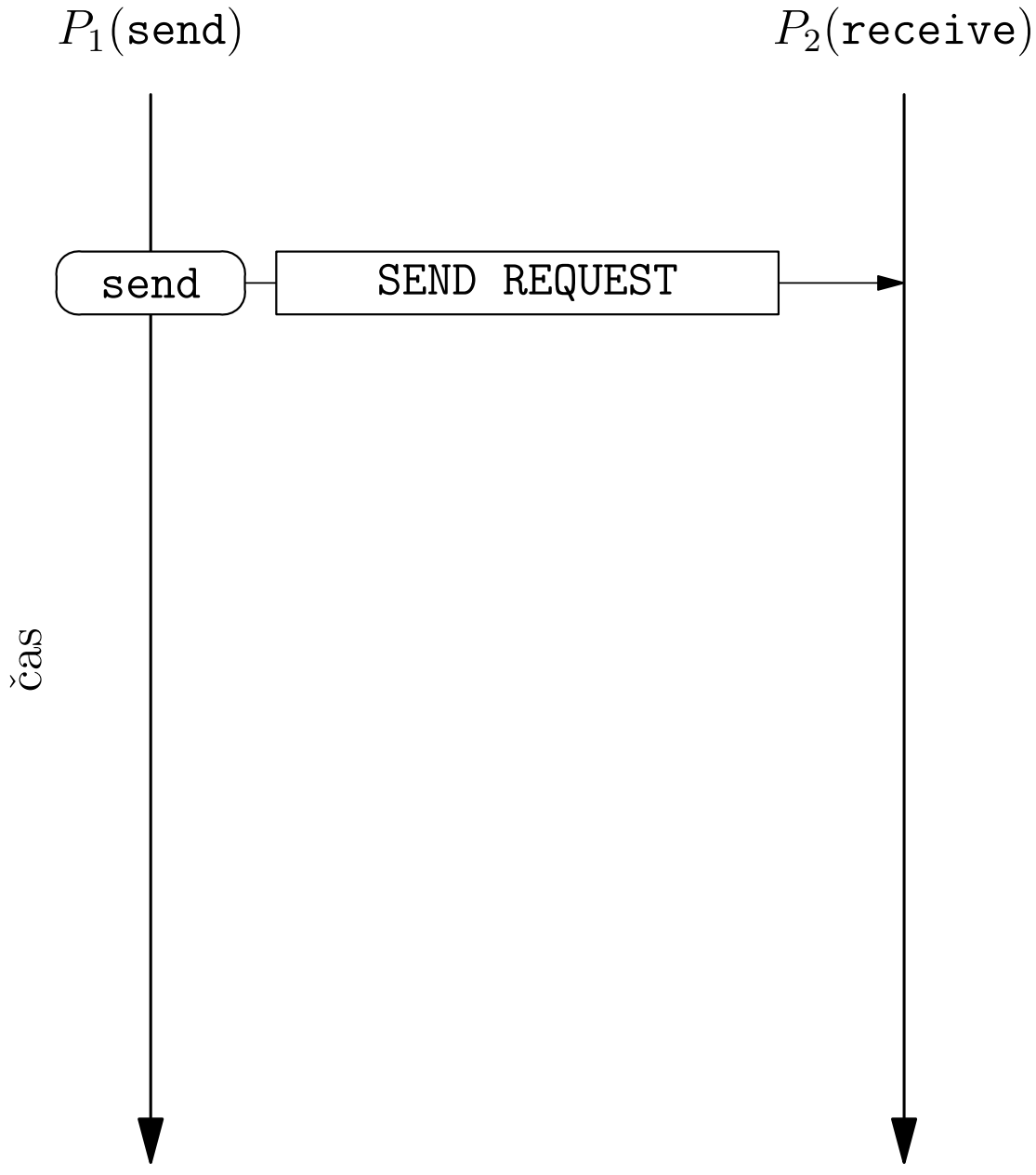
Blokující posílání zpráv bez bufferu

- odesílatel pošle požadavek k příjemci
- odesílatel potvrdí ve chvíli, kdy ve zpracování kódu dospěje k místu pro příjetí zprávy
- následně dojde k přenosu dat

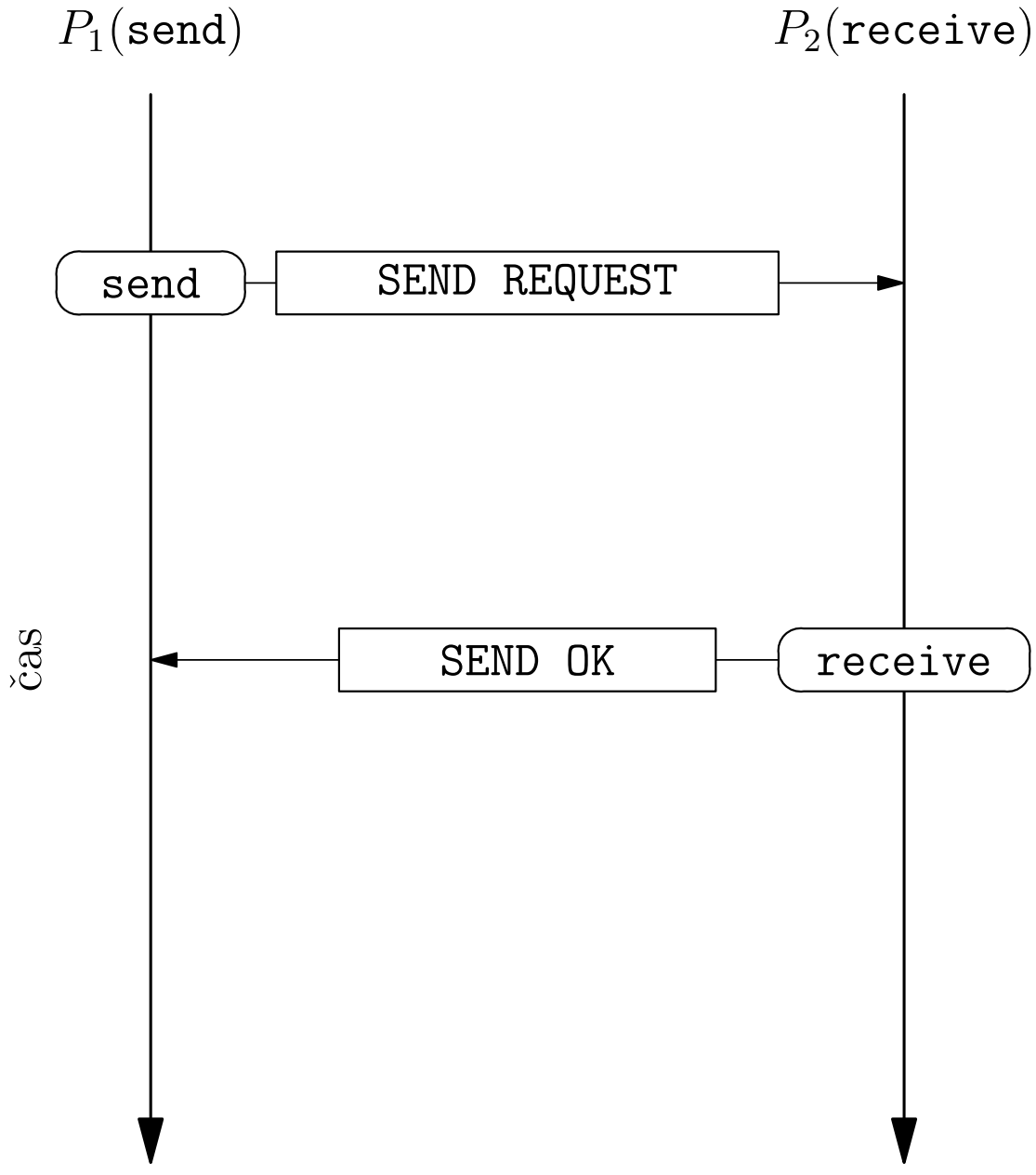
Blokující posílání zpráv bez bufferu



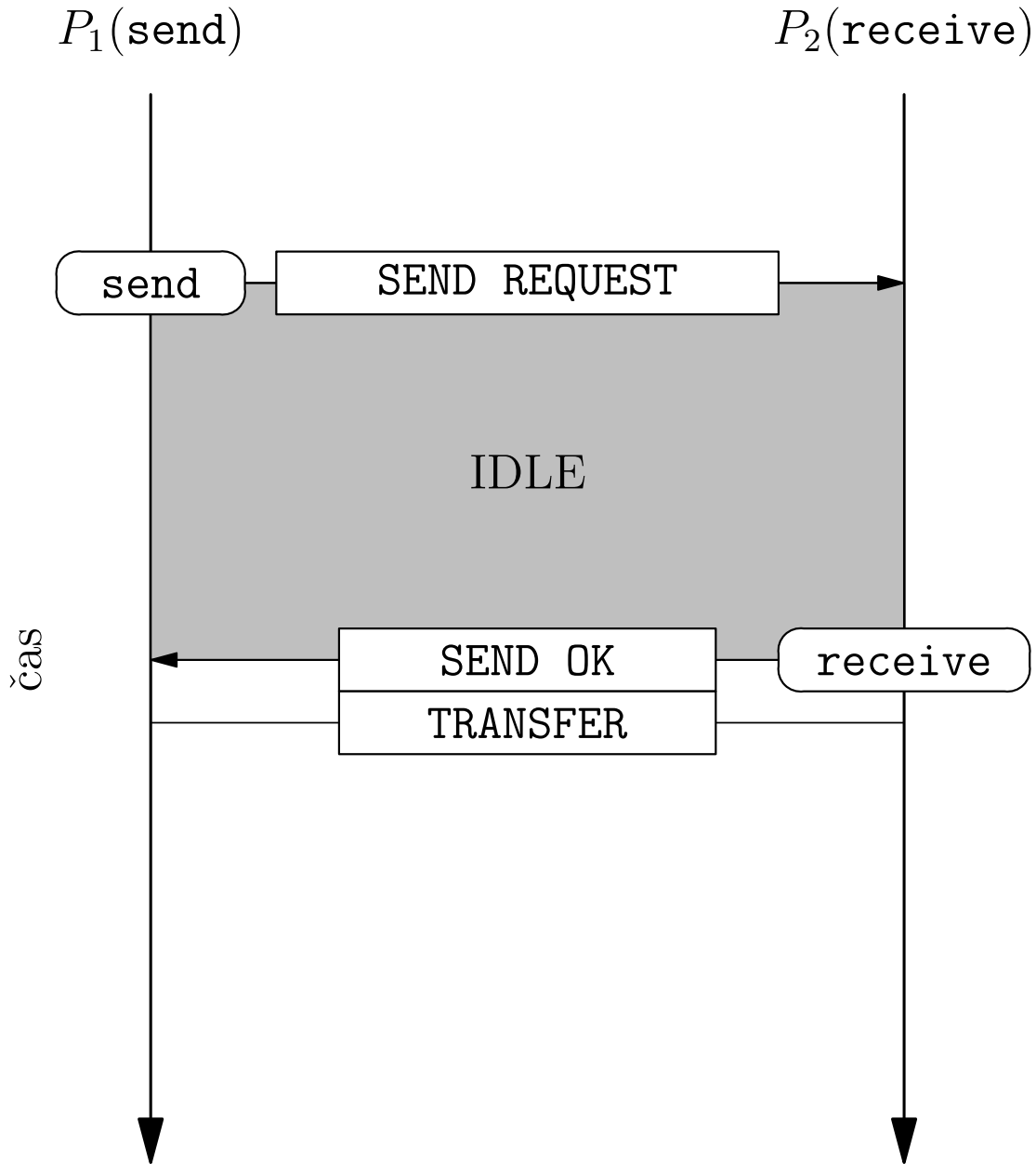
Blokující posílání zpráv bez bufferu



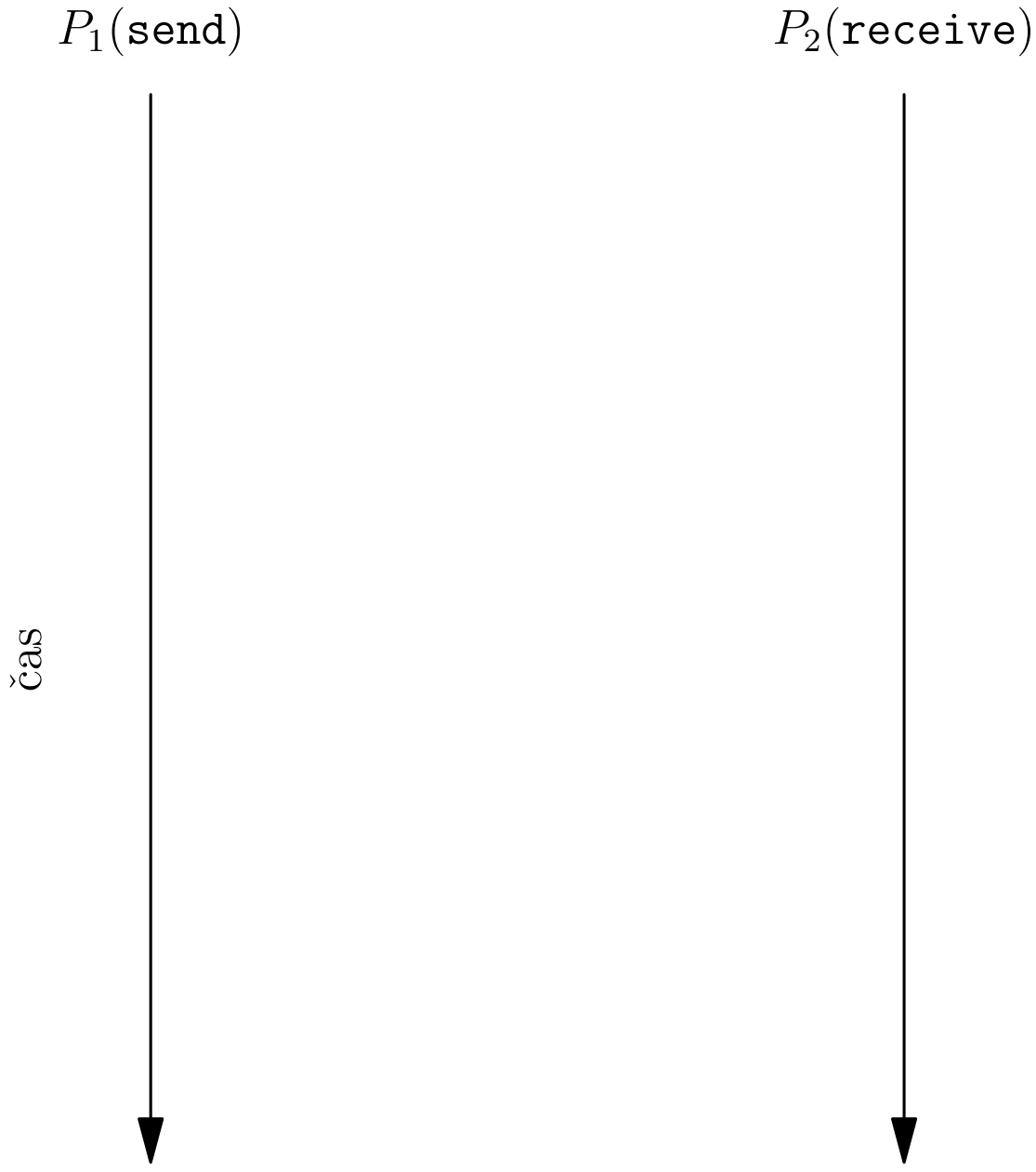
Blokující posílání zpráv bez bufferu



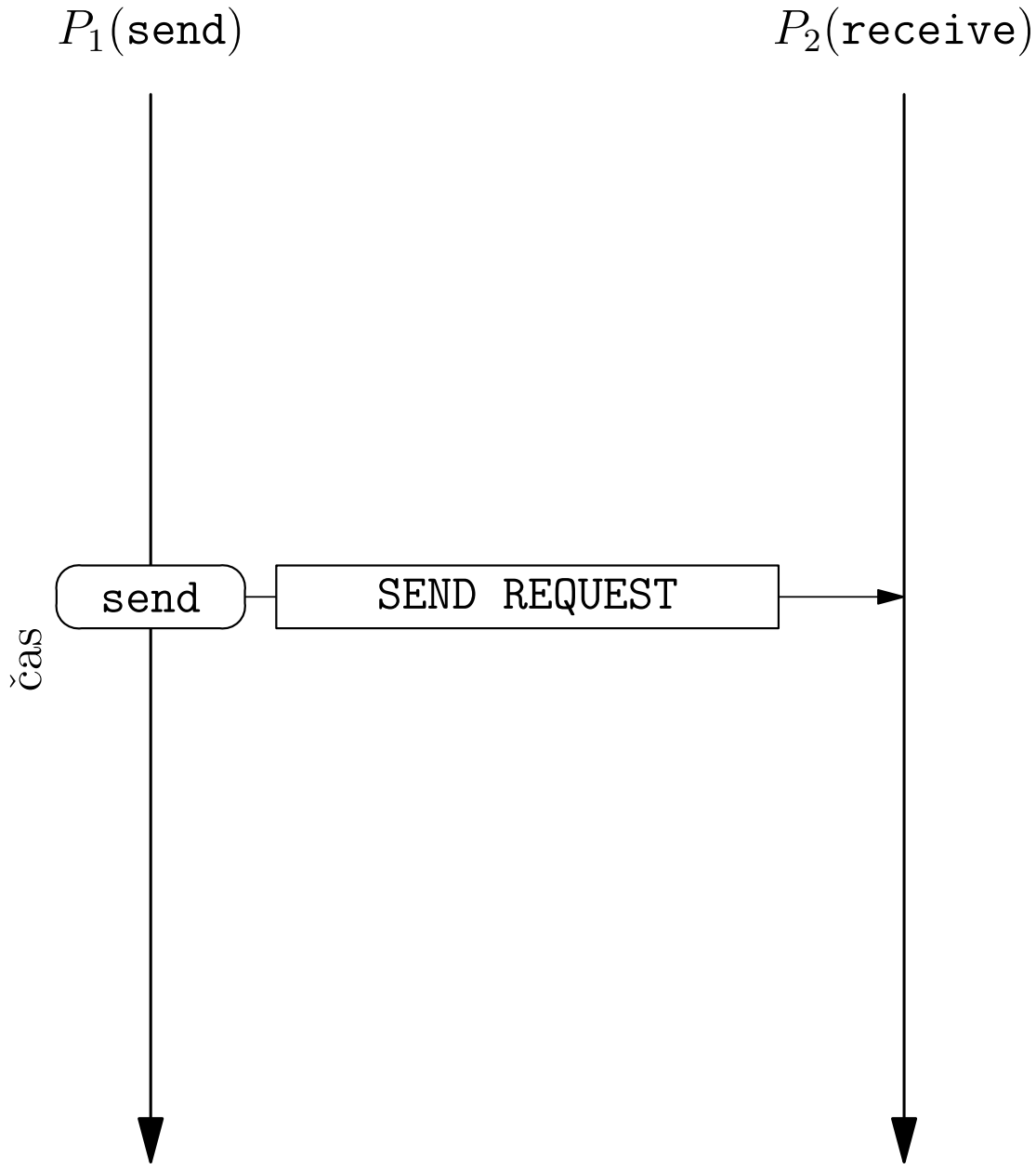
Blokující posílání zpráv bez bufferu



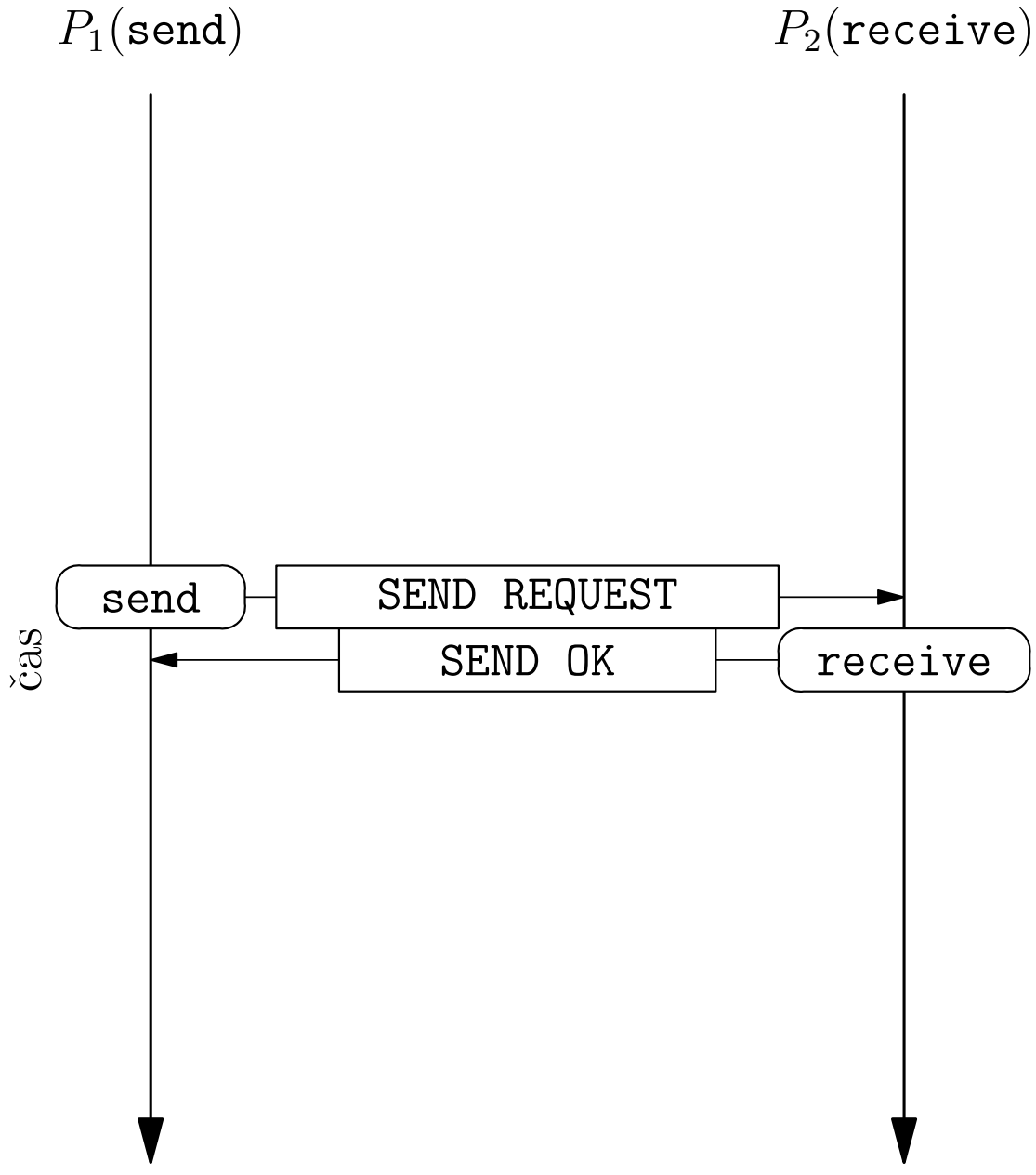
Blokující posílání zpráv bez bufferu



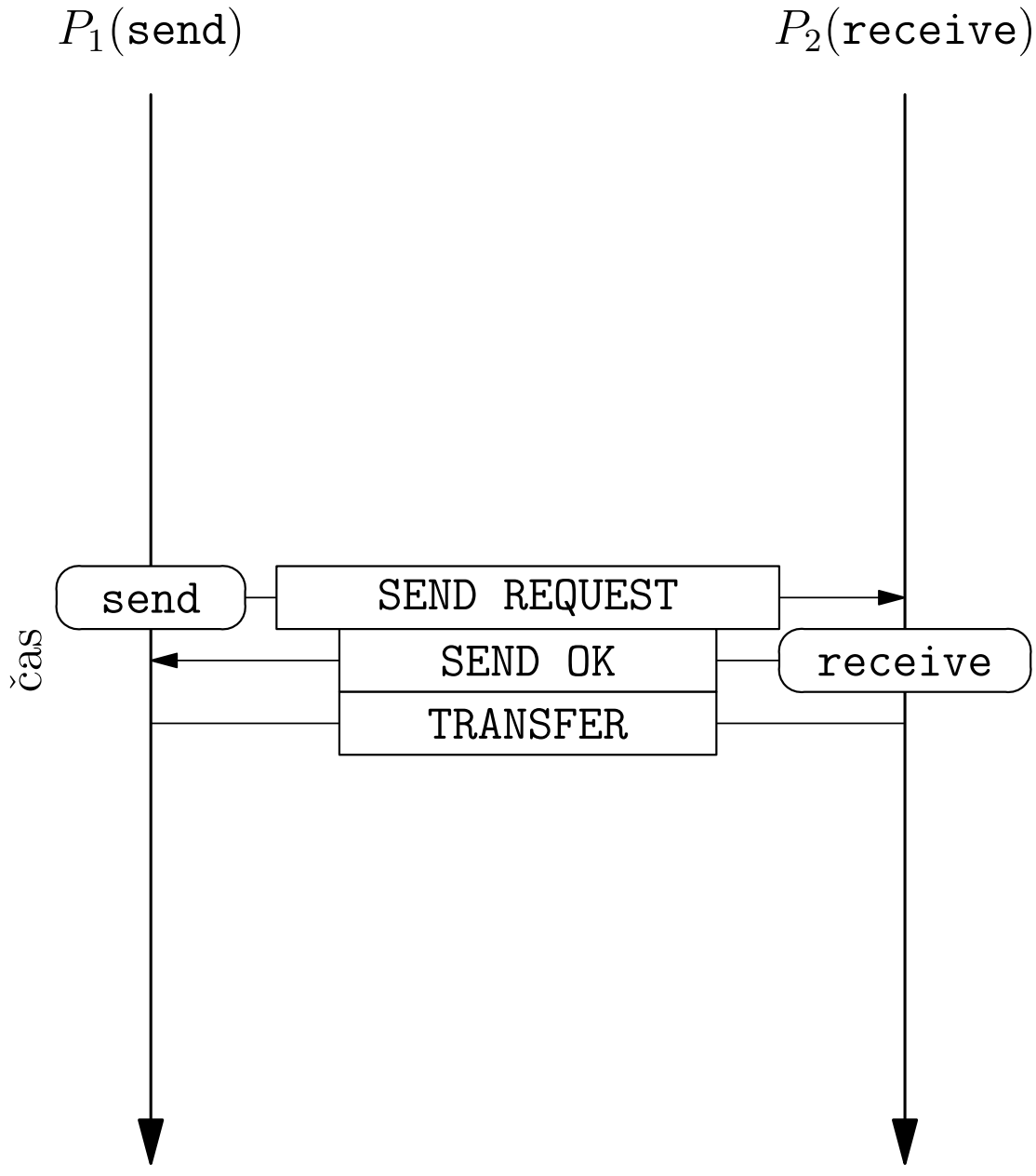
Blokující posílání zpráv bez bufferu



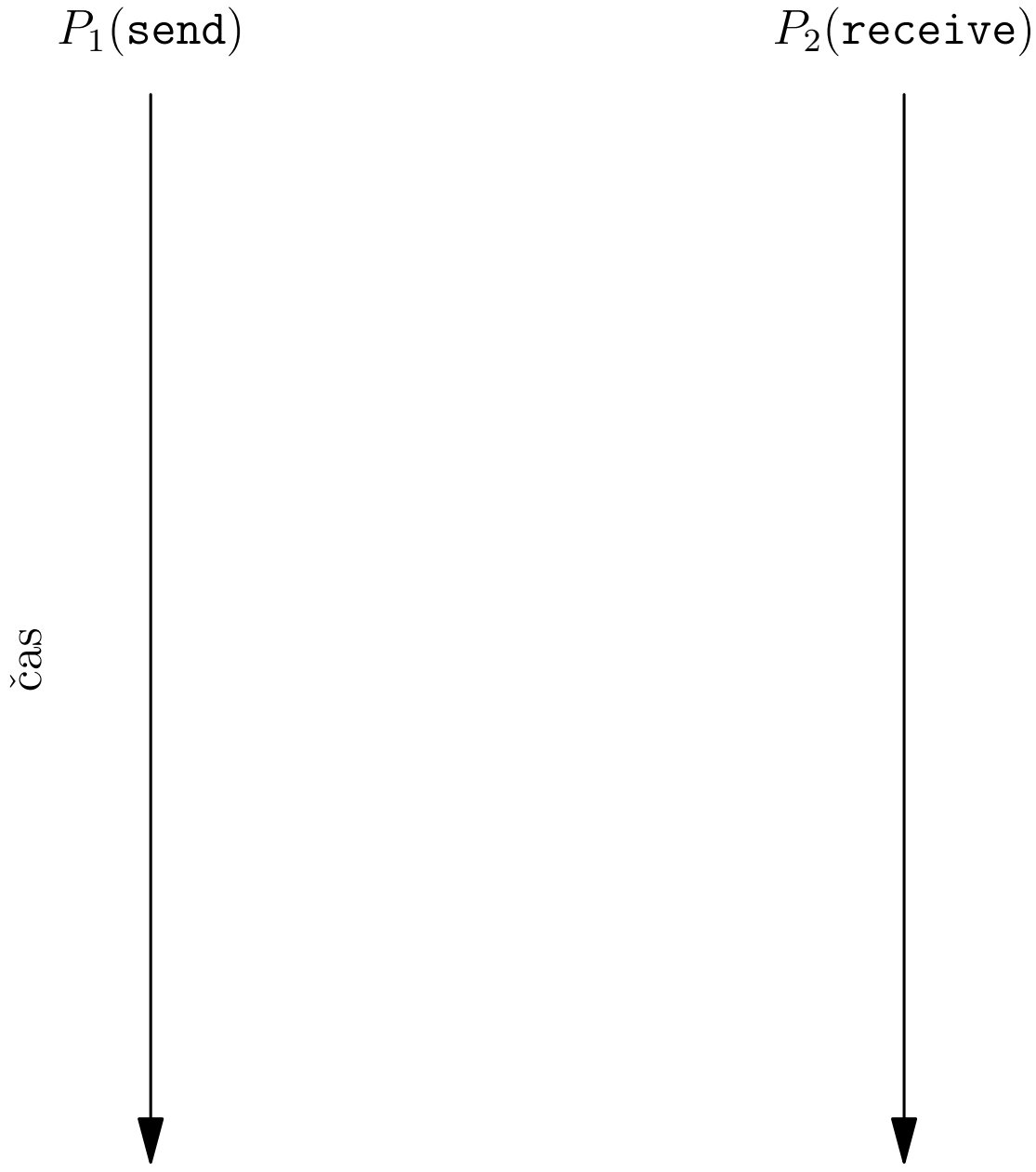
Blokující posílání zpráv bez bufferu



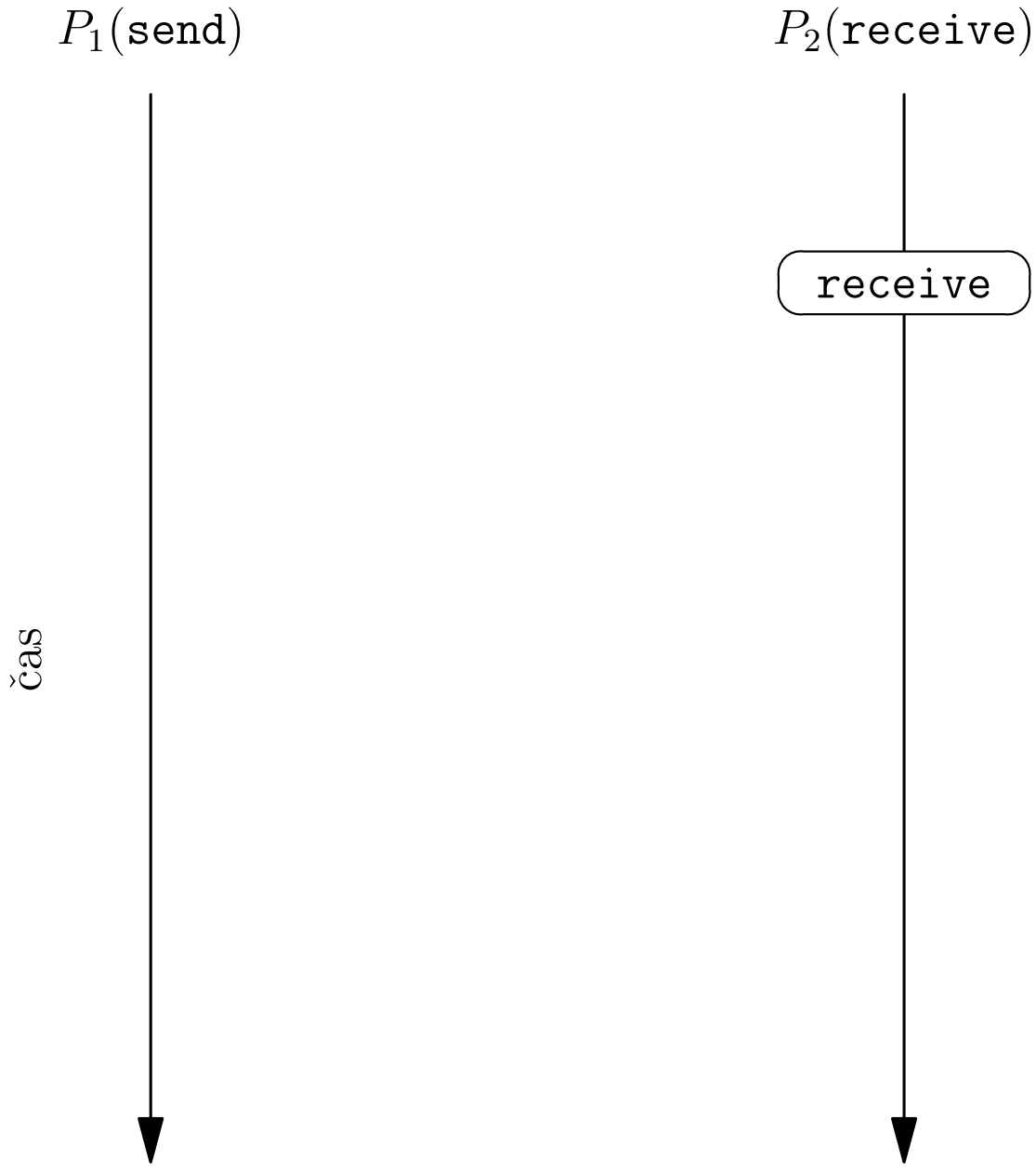
Blokující posílání zpráv bez bufferu



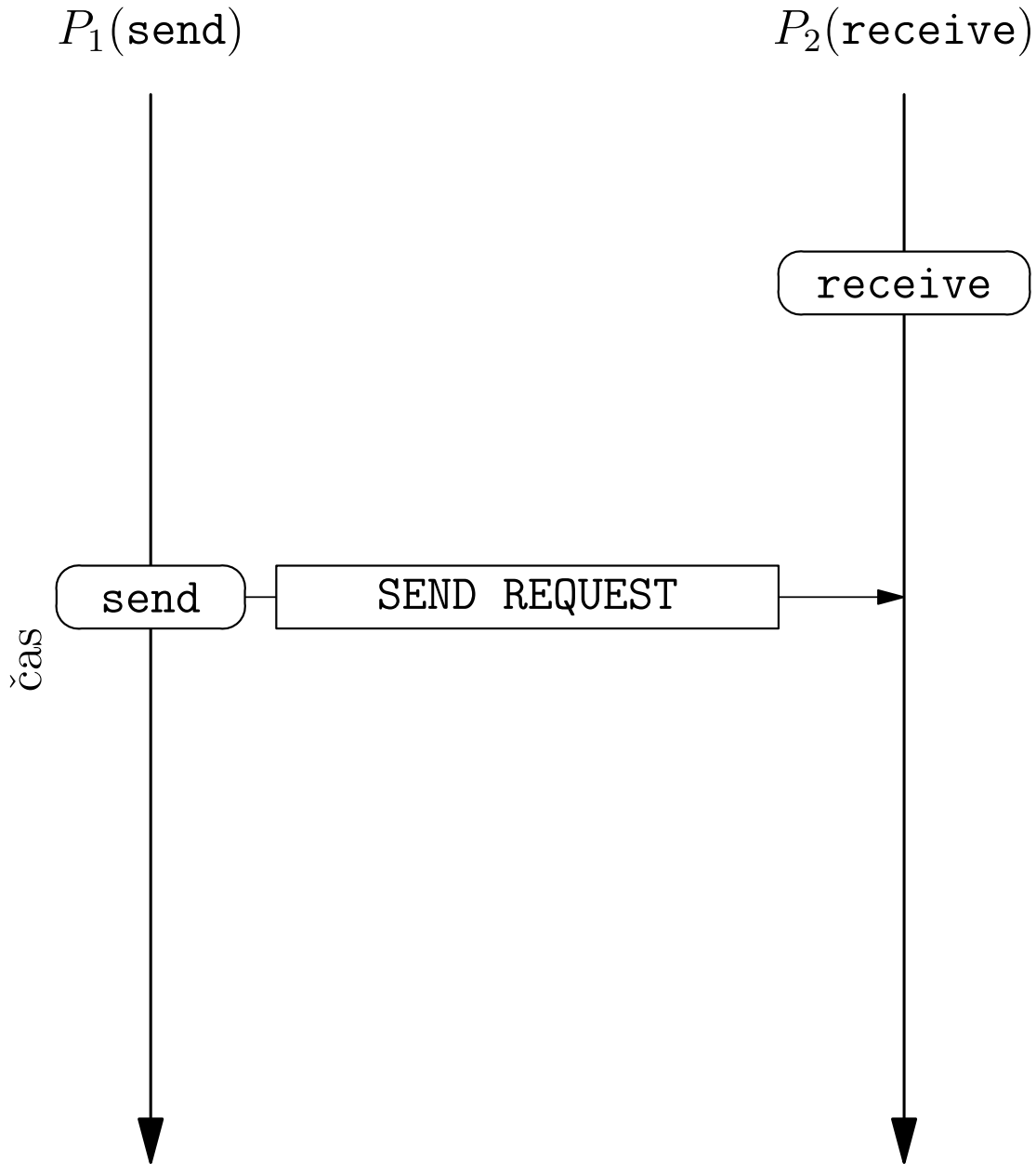
Blokující posílání zpráv bez bufferu



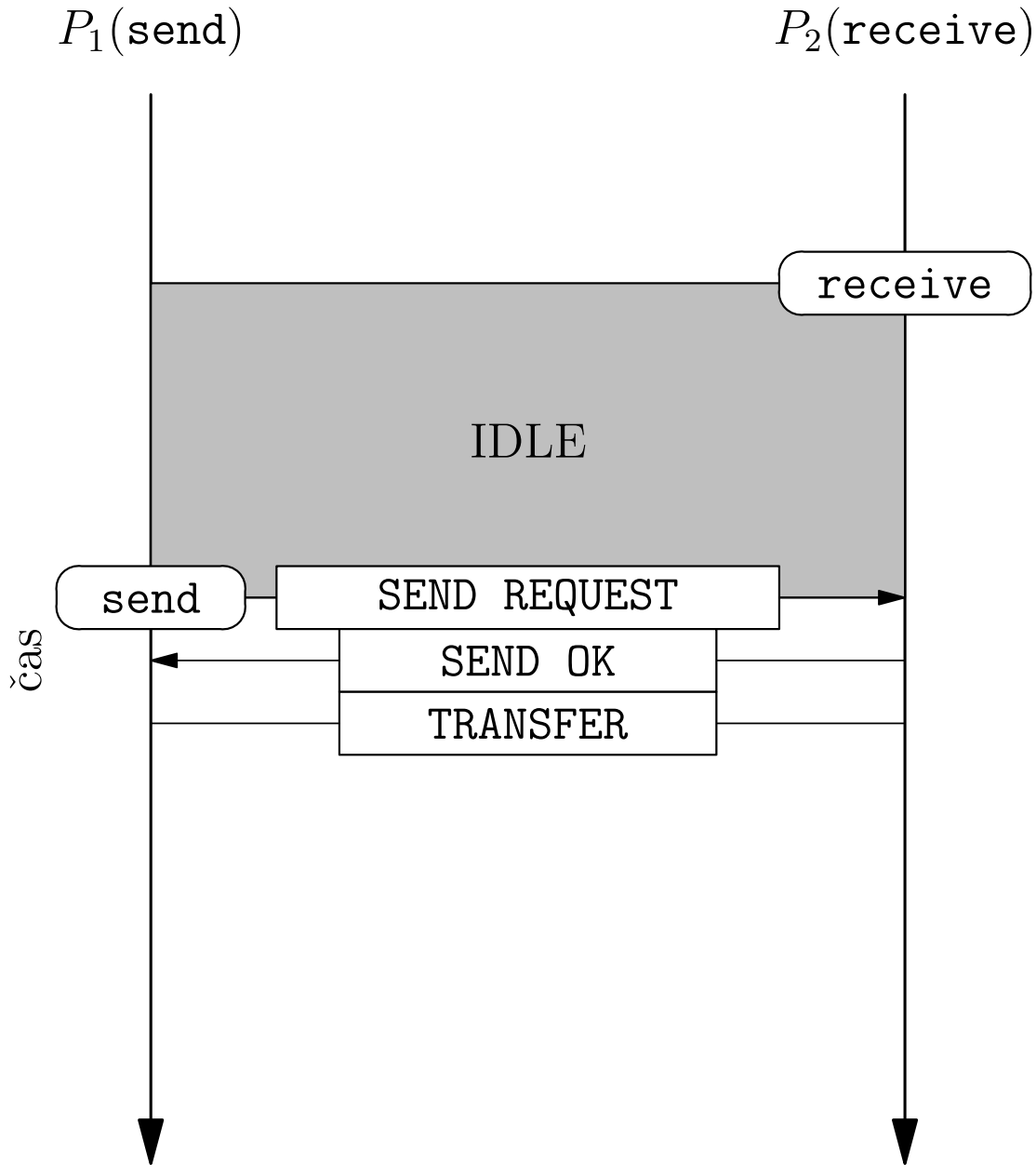
Blokující posílání zpráv bez bufferu



Blokující posílání zpráv bez bufferu



Blokující posílání zpráv bez bufferu



Blokující posílání zpráv bez bufferu

Možnost deadlocku:

P_1	P_2
`send(&a,1,2);`	`send( &a,1,1);`
`receive(&b,1,2);`	`receive( &b,1,1 );`

- oba procesy posílají *send request* a oba pak čekají na potvrzení *send ok*

Blokující posílání zpráv s bufferem

- snažíme se odstranit nutnost čekání na `SEND OK`
- při volání funkce `send` se data kopírují do pomocného bufferu
- program pak může okamžitě pokračovat
- přenos dat může proběhnout bez účasti CPU
- na straně příjemce se data také kopírují do pomocného bufferu
  - někdy je buffer jen na jedné straně

Blokující posílání zpráv s bufferem

- čekání odesílajícího procesu se podařilo zredukovat
- přibyla ale režie nutná při kopírování dat z/do bufferů
- pokud jsou procesy často synchronizovány, může být komunikace bez bufferů efektivnější
- pokud se buffery zaplní, dochází k čekání také
- příjem je vždy blokující
  - nesmí se pokračovat dál, dokud nejsou data zkopírována z bufferu příjemce

Blokující posílání zpráv s bufferem

Možnost deadlocku:

P_1	P_2
`receive(&b,1,2);`	`receive( &b,1,1 );`
`send(&a,1,2);`	`send( &a,1,1);`

- oba procesy čekají na příjem zprávy

Neblokující posílání zpráv

- komunikační funkce vrací řízení programu dříve, než je to sémanticky bezpečné
- programátor musí sám ohlídat, zda již byla data přenesena
- mezi tím ale může zpracovávat jiný kód, který napracuje s posílanými daty
- existují funkce, které řeknou, zda již komunikace úspěšně proběhla
- i neblokující komunikace může používat buffery

Funkce `send` a `receive` v MPI

```
1  int MPI_Send( void* buf, int count,
2                MPI_Datatype datatype, int dest,
3                int tag, MPI_Comm comm );
```

```
1  int MPI_Receive( void* buf, int count,
2                   MPI_Datatype datatype, int source,
3                   int tag, MPI_Comm comm,
4                   MPI_Status* status );
```

- `buf` ukazatel na pole dat typu `datatype` o velikosti `count`
- `datatype` může být: `MPI_CHAR`, `MPI_INT`, `MPI_FLOAT`, `MPI_DOUBLE`, ...
- `dest` ID příjemce
- `source` ID odesílatele
- `tag` určuje typ zprávy
- `comm` udává komunikační skupinu

Struktura `MPI_Status`

```
1  typedef struct MPI_Status
2  {
3      int MPI_SOURCE;
4      int MPI_TAG;
5      int MPI_ERROR;
6  };
```

- `MPI_SOURCE` = odesílatel
- `MPI_SOURCE` = typ zprávy
- `MPI_ERROR` = chybové hlášení

Funkce `send` a `receive` v MPI

U odesílatele i příjemce musí být `count`, `datatype` a `tag` stejné.

Funkce

```
1  int MPI_Get_count( MPI_Status* status,
2                     MPI_Datatype datatype,
3                     int* count );
```

udává skutečný počet přijatých dat.

- obě funkce jsou vždy blokující
- `send` může být implementováno bufferově (nelze s tím počítat, pozor na deadlock)

Funkce `send` a `receive` v MPI

Příklad **s deadlockem**:

```
1  int a[ 10 ], b[ 10 ], myrank;
2  MPI_Status status;
3  ...
4  MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
5  if( myrank == 0 )
6  {
7      MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
8      MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
9  }
10 else if( myrank == 1 )
11 {
12     MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
13     MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
14 }
```

Funkce `send` a `receive` v MPI

Příklad **bez deadlocku**:

```
 1  int a[ 10 ], b[ 10 ], myrank;
 2
 3  MPI_Status status;
 4  ...
 5  MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
 6
 7  if( myrank == 0 )
 8  {
 9      MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
10      MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
11  }
12  else if( myrank == 1 )
13  {
14      MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
15      MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
16  }
```

Simultání `send` a `receive` v MPI

Za účelem zabránění deadlocku je často lepší použít funkce pro simultání `send/receive`.

```
1  int MPI_Sendrecv( void* sendbuf, int sendcoutn,
2                    MPI_Datatype senddatatype, int dset,
3                    int sendtag, void* recvbuf,
4                    int recvcount, MPI_Datatype recvdatatype,
5                    int source, int recvtag,
6                    MPI_Comm comm, MPI_Status* status )
```

```
1  int MPI_Sendrecv_replace( void* buf, int count,
2                            MPI_Datatype datatype, int dest,
3                            int sendtag, int source,
4                            int recvtag, MPI_Comm comm,
5                            MPI_Status* status )
```

V případě `MPI_Sendrecv_replace` jsou odeslaná data přepsána přijatými.

Neblokující `send` a `receive` v MPI

```
1  int MPI_Isend( void* buf, int count,
2                 MPI_Datatype datatype, int dest,
3                 int source, int tag,
4                 MPI_Comm comm, MPI_Request* request )
```

```
1  int MPI_Irecv( void* buf, int count,
2                 MPI_Datatype datatype, int source,
3                 int tag, MPI_Comm comm,
4                 MPI_Request* request )
5  }
```

Obě funkce vracejí řízení programu dříve, než jsou data skutečně přenesena.

- `request` - slouží k ověření, zda byla data již přenesena

Neblokující `send` a `receive` v MPI

Funkce pro ověření stavu přenosu dat při neblokujícím send a receive.

```
1  int MPI_Test( MPI_Request* request,
2                int* flag,
3                MPI_Status* status )
```

```
1  int MPI_Wait( MPI_Request* request,
2                MPI_Status* status )
```

- `MPI_Test` vrací v proměnné `flag` nenulovou hodnotu, pokud již operace skončila
- `MPI_Wait` čeká na ukončení operace

Komunikační operace

- jde o komunikační vzory, které se v paralelním programování vyskytují velice často
- komunikující procesy mohou běžet na jednom nebo i na více výpočetních uzlech
- jejich speciální implementace má následující výhody
  - zjednodušuje práci programátora
  - mohou využívat hardwarovou podporu dané paralelní architektury
  - udržují aktivní síťové spojení komunikační sítě
    - tím vlastně několik malých zpráv "shlukuje do jedné"
- k většině těchto operací existují i operace duální, které "běží v opačném směru"

One-to-all broadcast/All-to-one reduction

One-to-all broadcast

- jeden proces má identická data o velikosti m a rozešle je všem ostatním
  - používá se např. při rozesílání konfiguračních parametrů

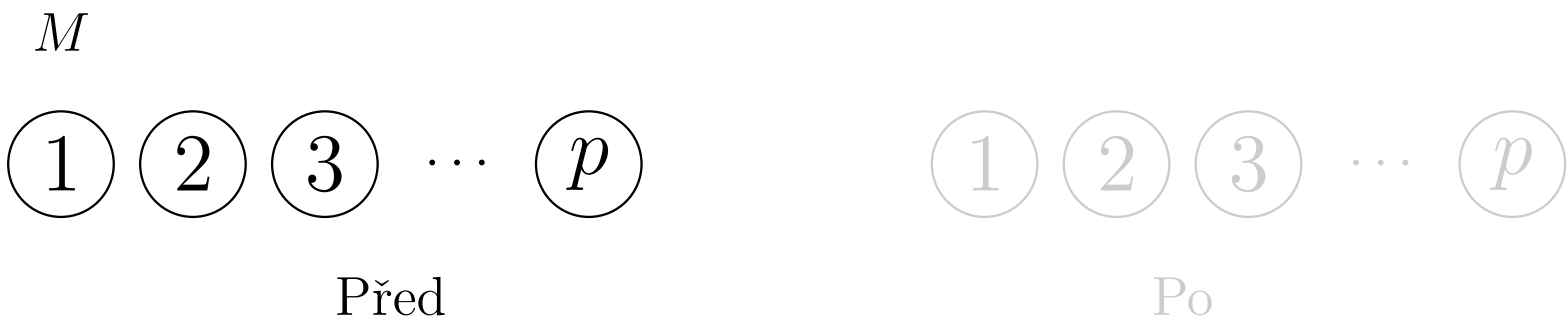
One-to-all broadcast/All-to-one reduction

All-to-one reduction

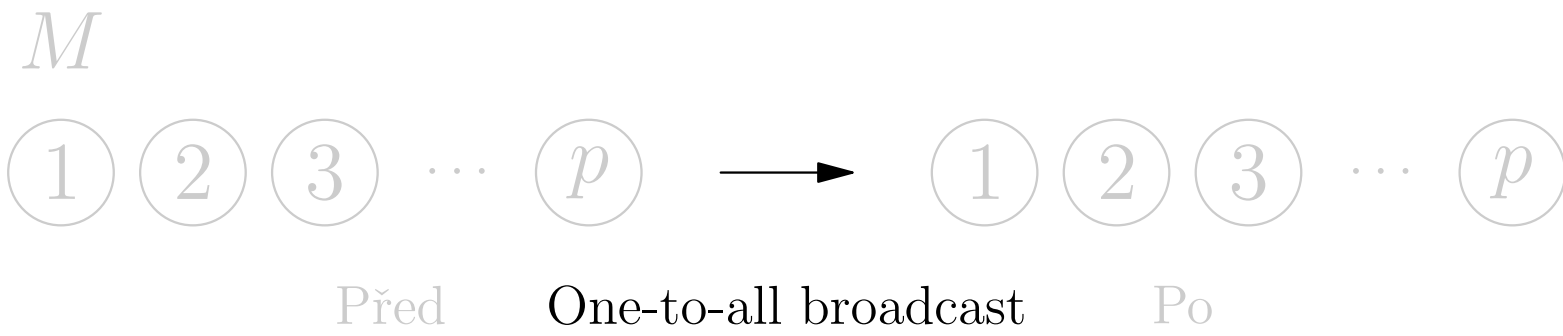
- každý proces má data o velikosti m a obecně jsou různá (různé hodnoty)
- data jsou poslána jednomu procesu a současně sloučena dohromady pomocí nějaké **asociativní operace** do velikosti m

Příklad: Máme 100 reálných čísel umístěných na 100 procesech (každý proces má jedno reálné číslo). Výsledkem redukce je součet všech čísel a výsledek má proces číslo 0.

One-to-all broadcast/All-to-one reduction

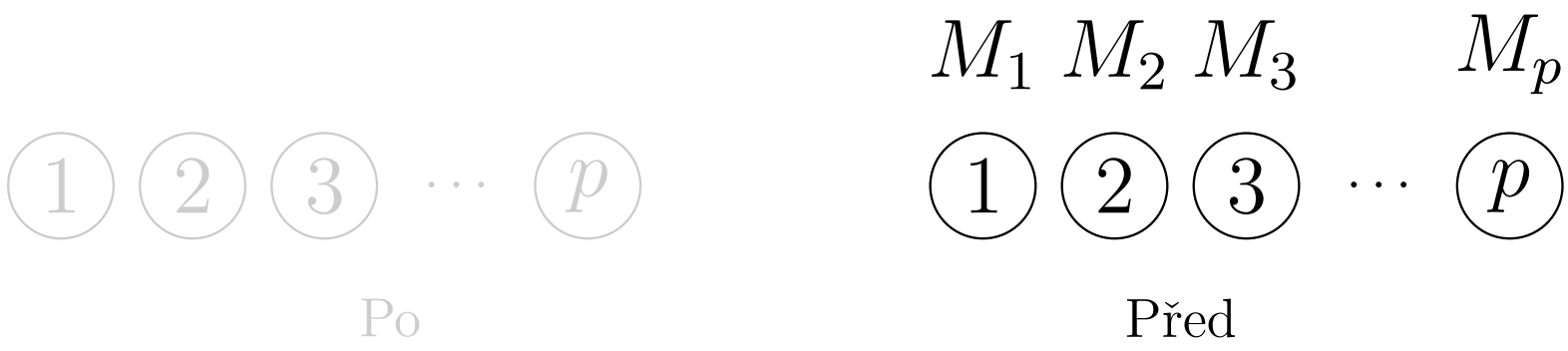


One-to-all broadcast/All-to-one reduction

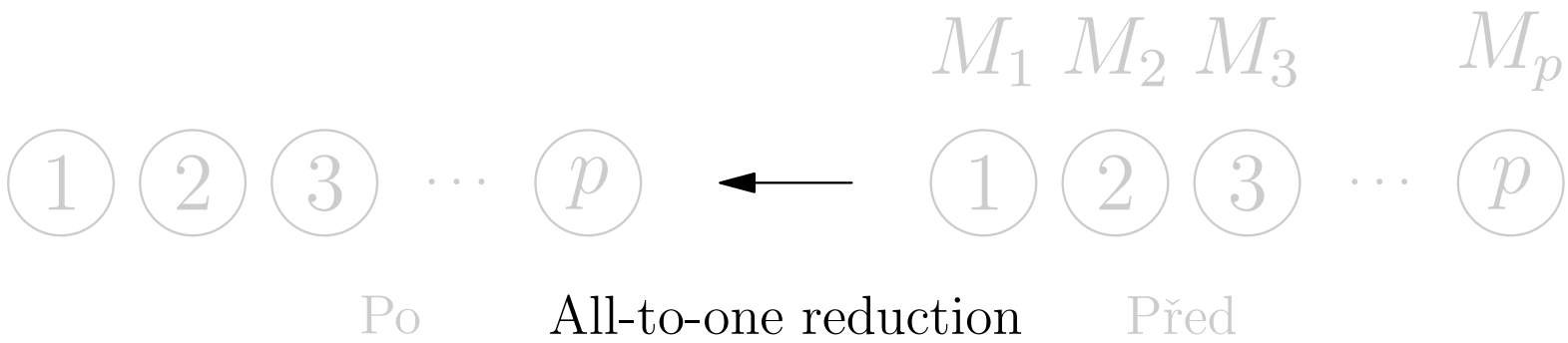


One-to-all broadcast/All-to-one reduction

One-to-all broadcast/All-to-one reduction



One-to-all broadcast/All-to-one reduction



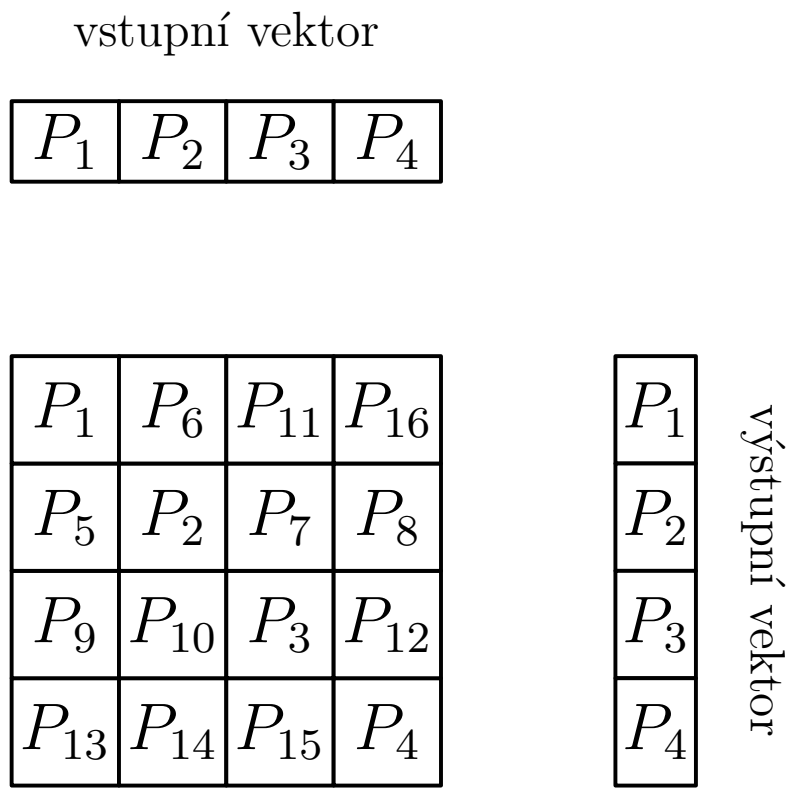
One-to-all broadcast/All-to-one reduction

$\sum_{i=1}^{p} M_i$

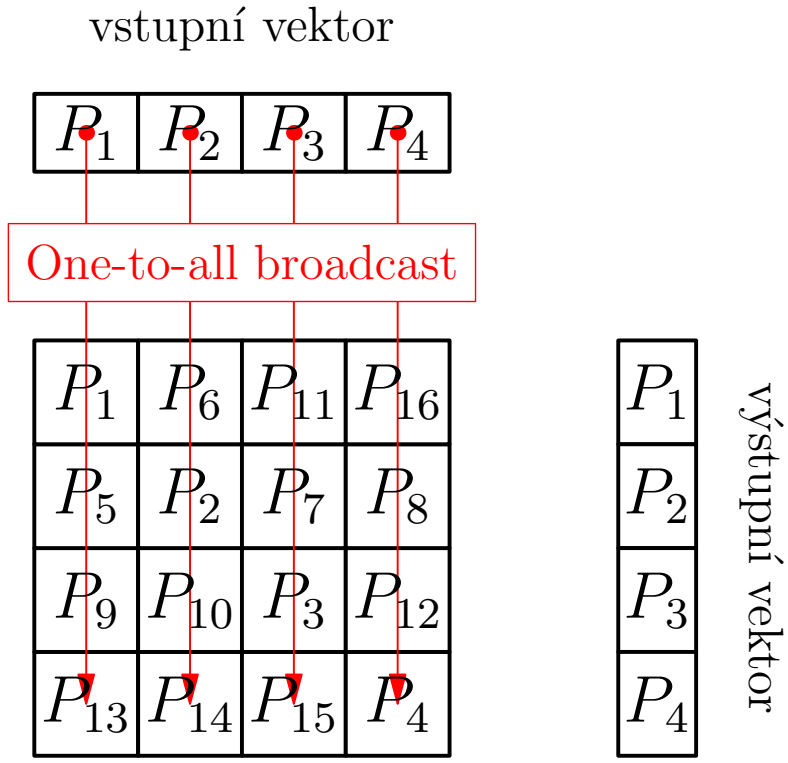
Po

Před

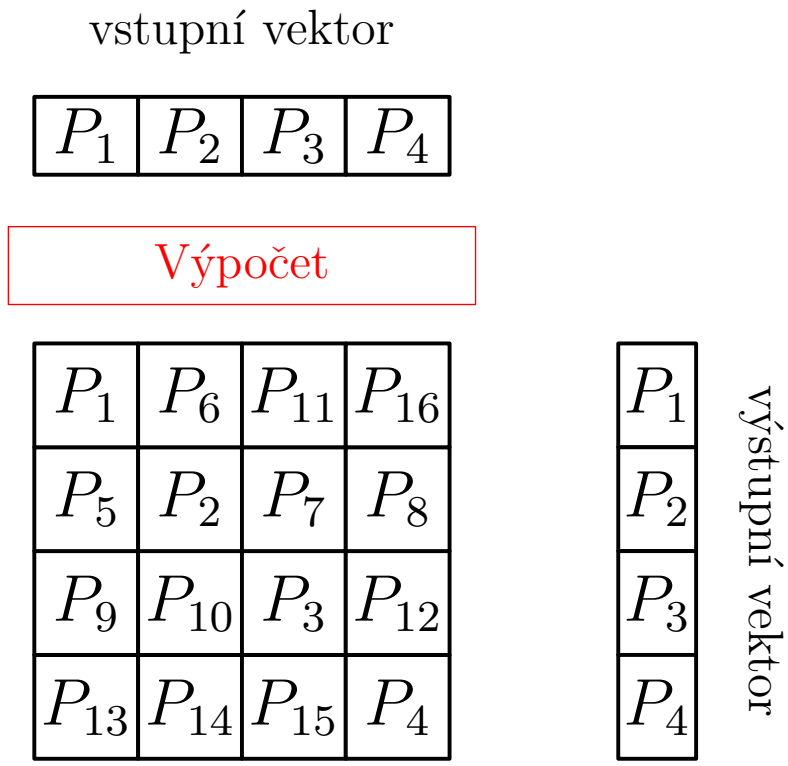
One-to-all broadcast/All-to-one reduction - příklad



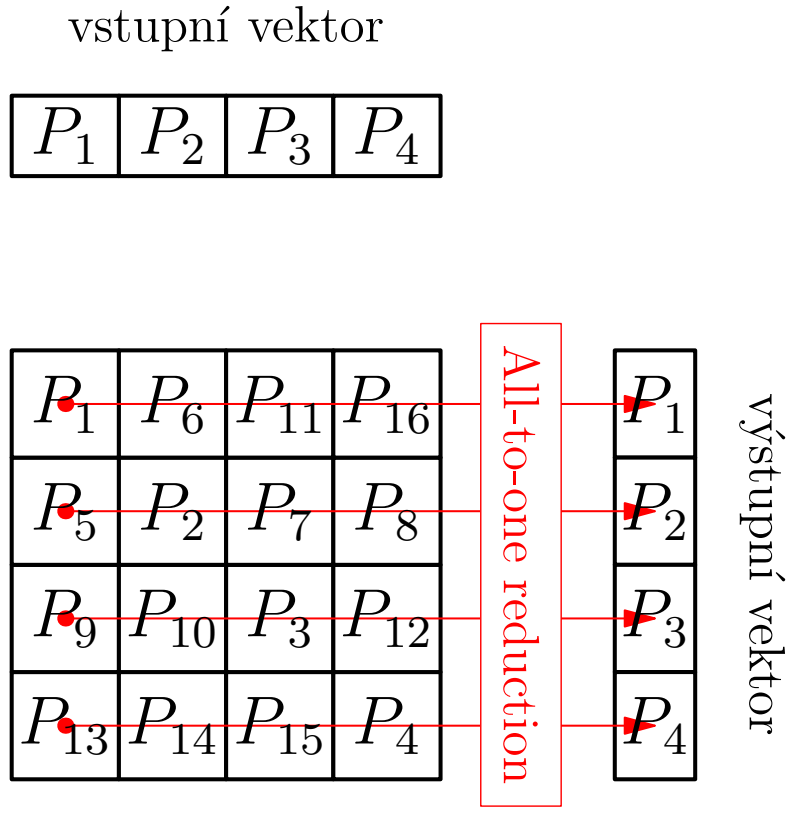
One-to-all broadcast/All-to-one reduction - příklad



One-to-all broadcast/All-to-one reduction - příklad

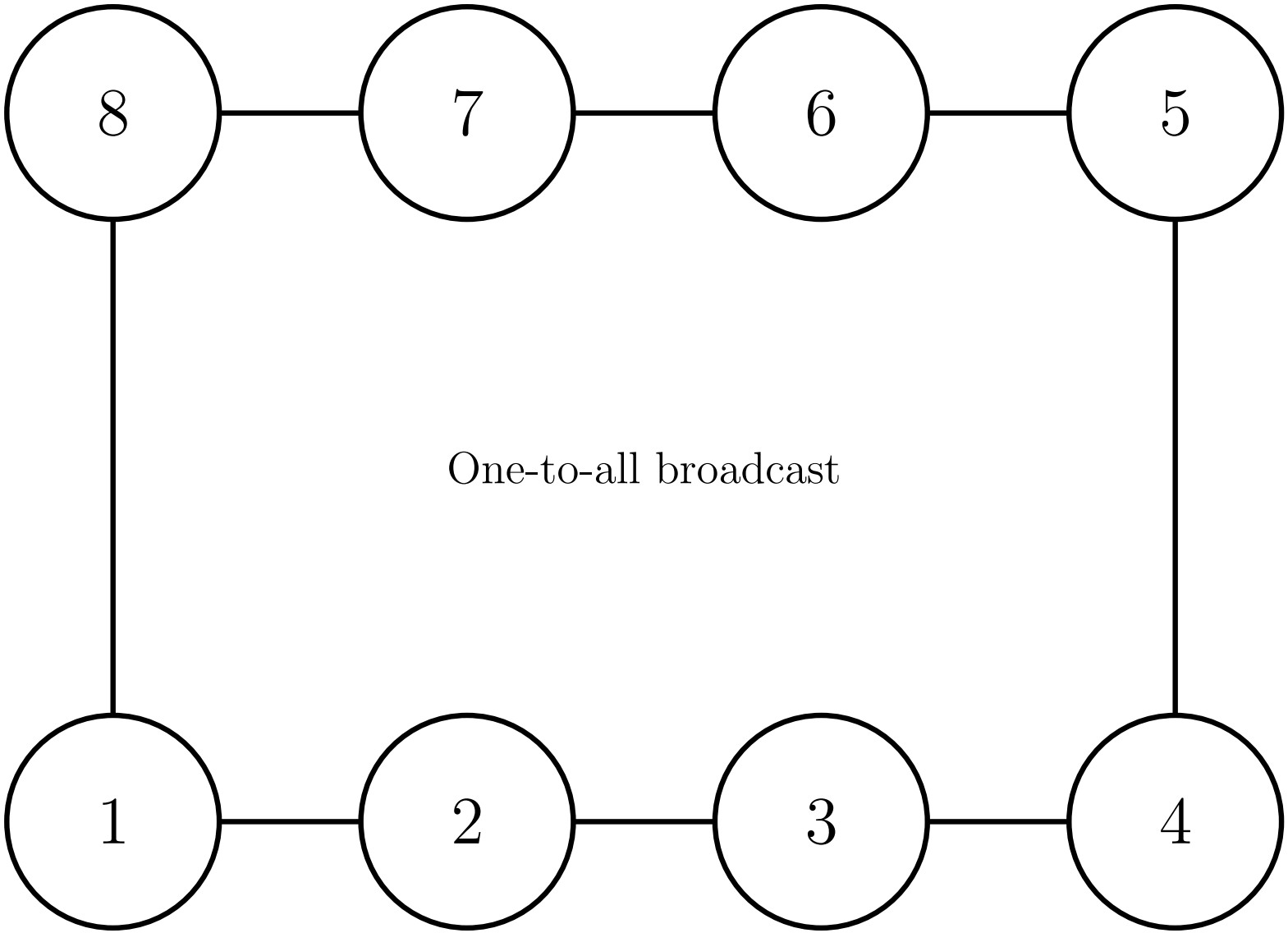


One-to-all broadcast/All-to-one reduction - příklad



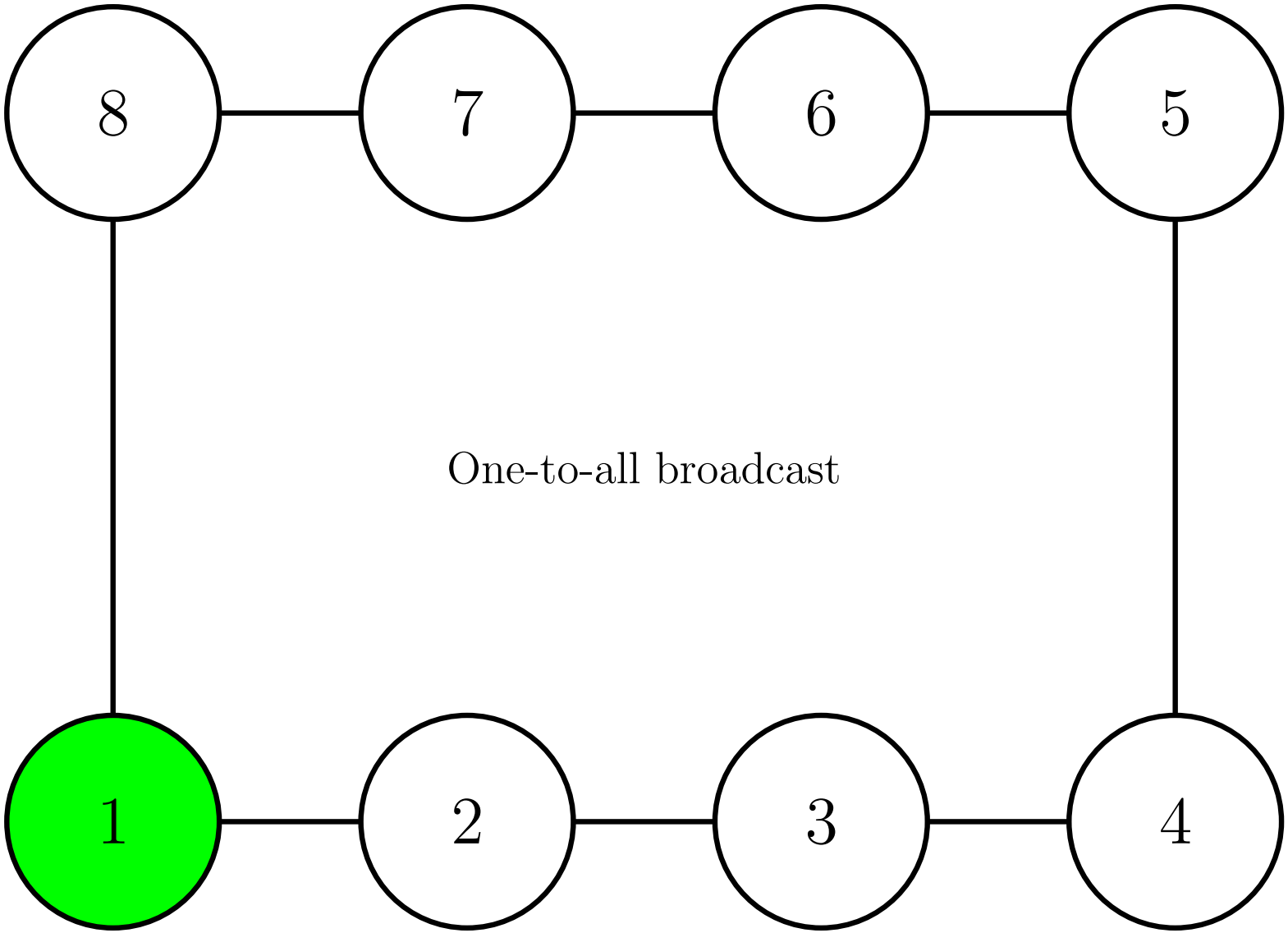
One-to-all broadcast/All-to-one reduction

Realizace v případě sítě typu **kruh**



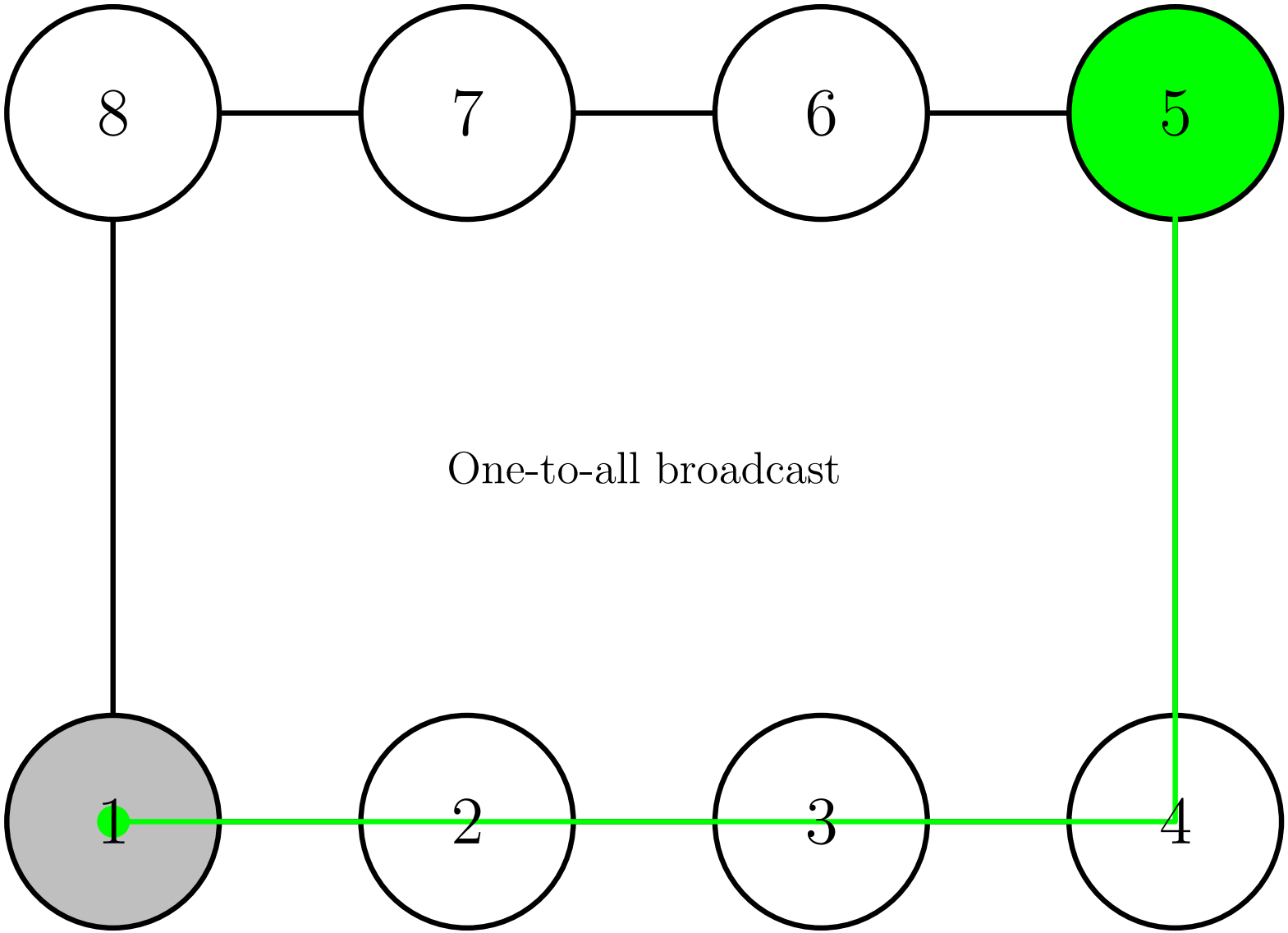
One-to-all broadcast/All-to-one reduction

Realizace v případě sítě typu **kruh**



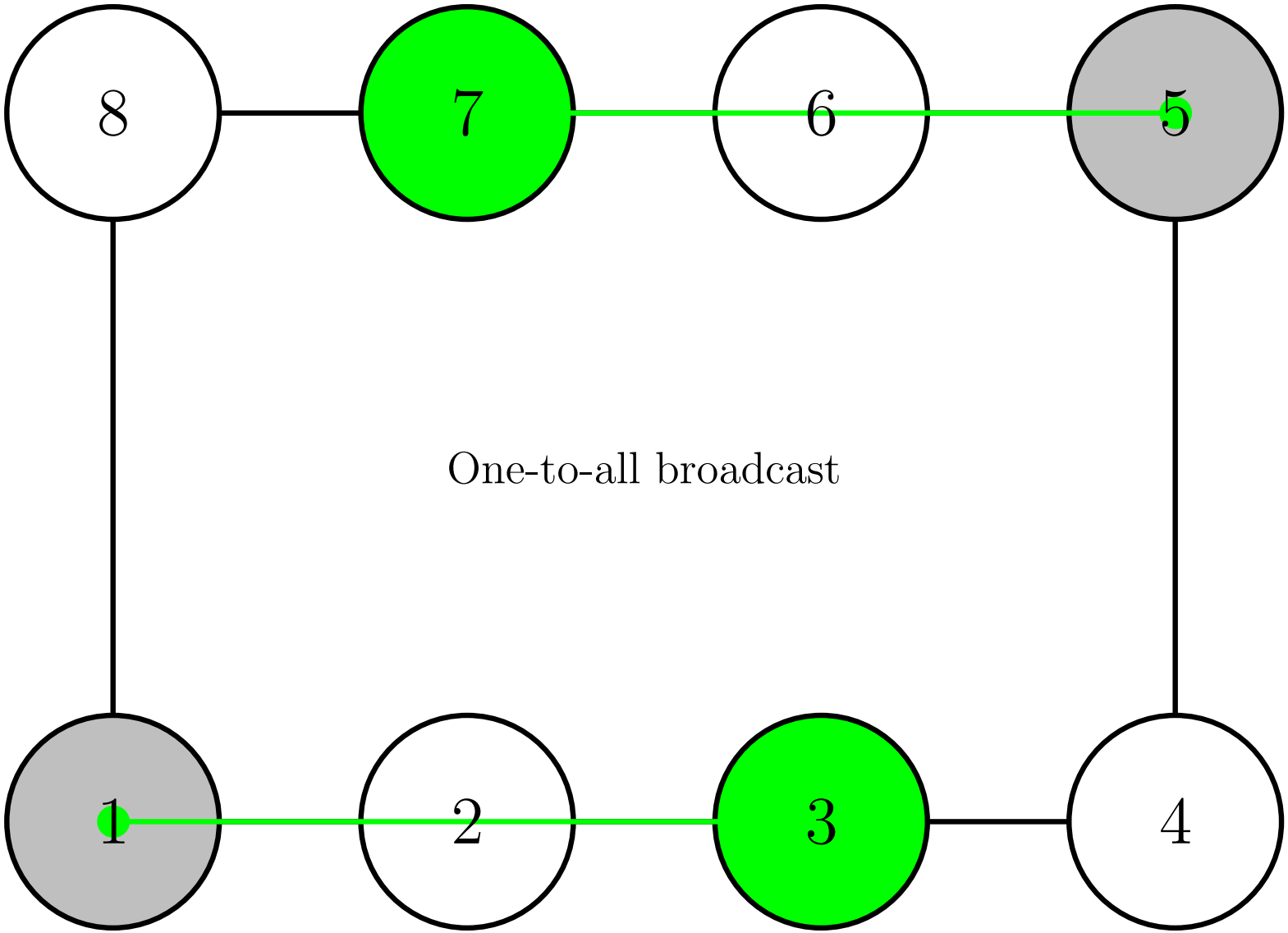
One-to-all broadcast/All-to-one reduction

Realizace v případě sítě typu **kruh**



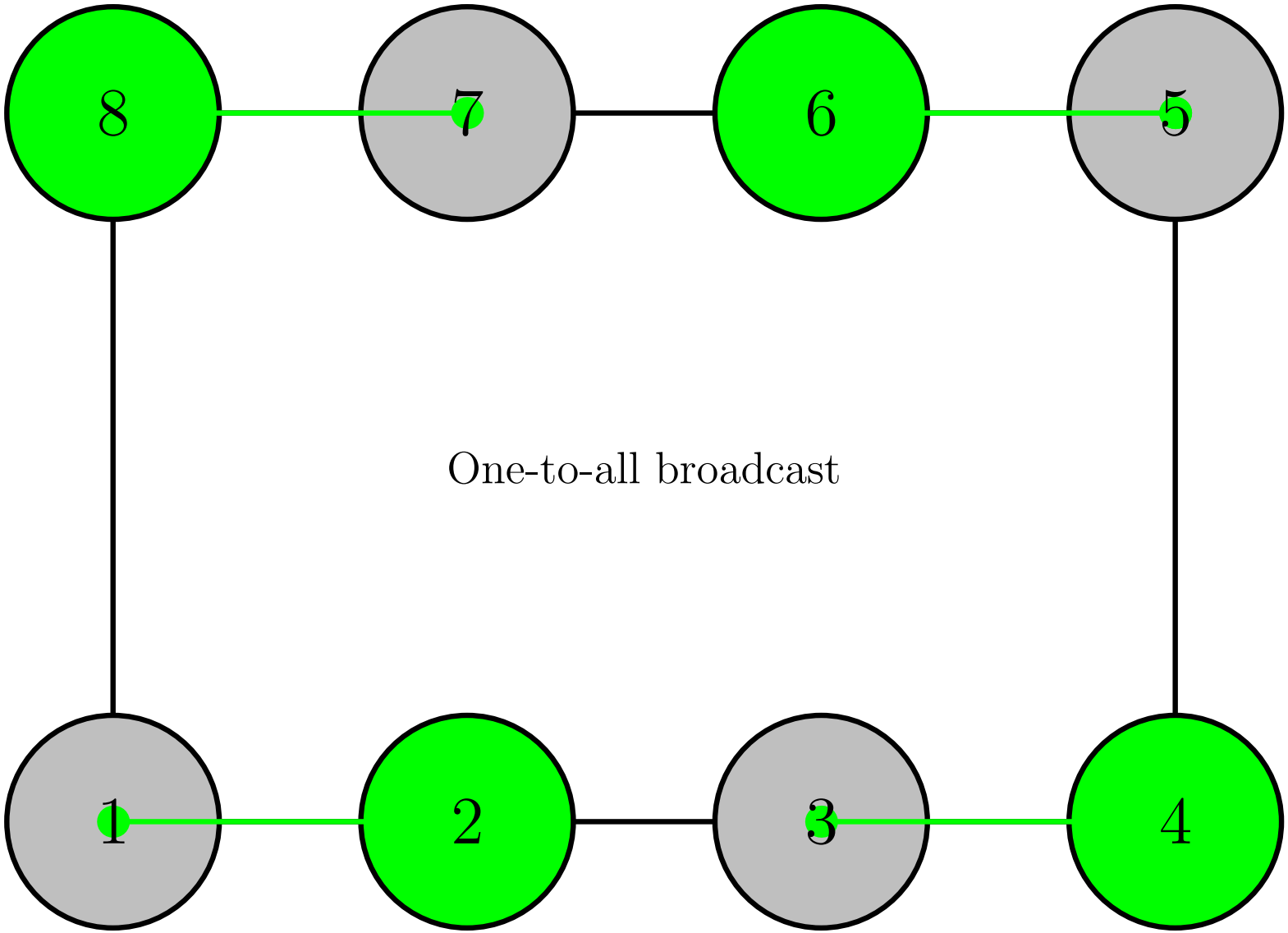
One-to-all broadcast/All-to-one reduction

Realizace v případě sítě typu **kruh**



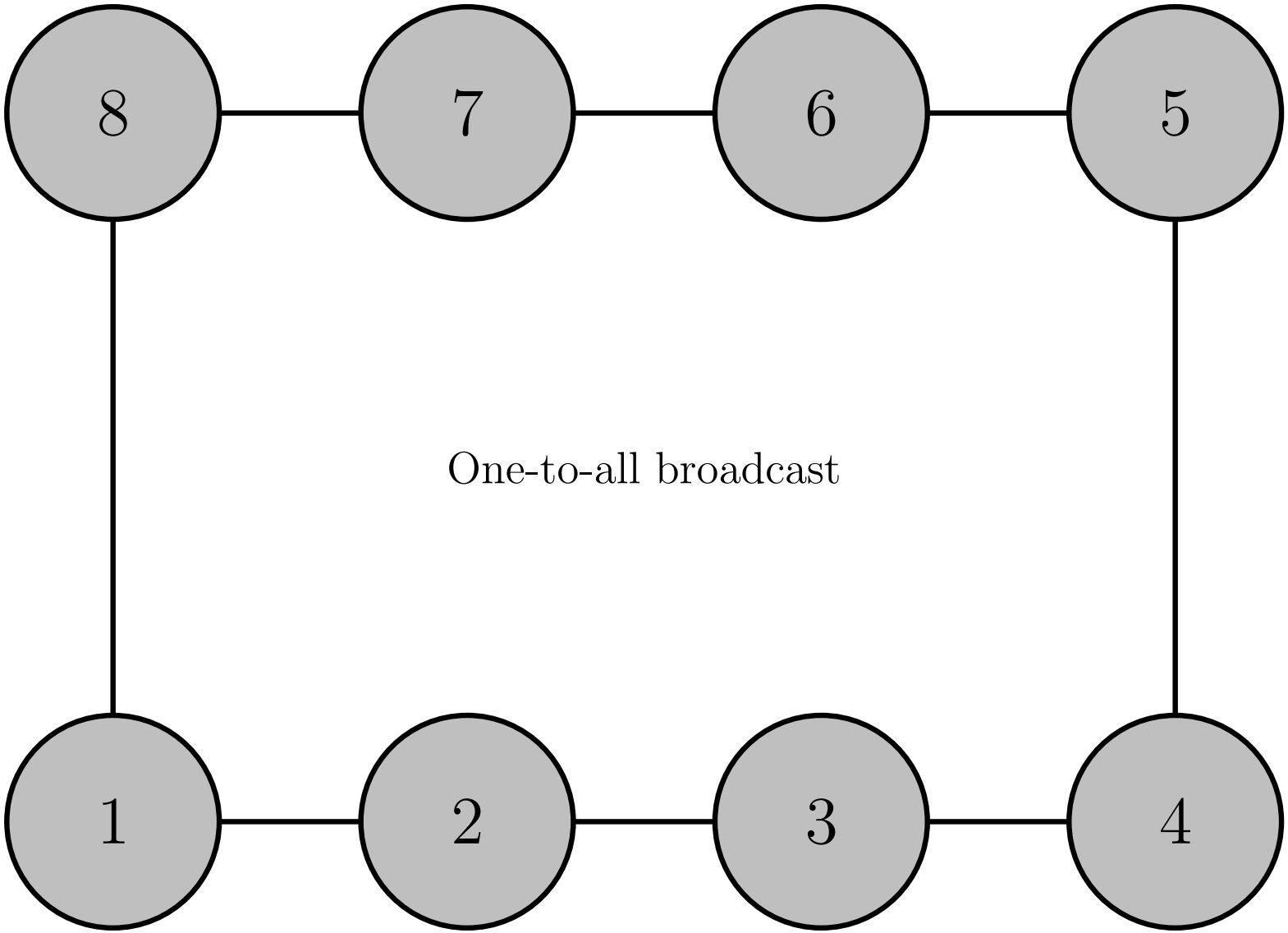
One-to-all broadcast/All-to-one reduction

Realizace v případě sítě typu **kruh**



One-to-all broadcast/All-to-one reduction

Realizace v případě sítě typu **kruh**



One-to-all broadcast/All-to-one reduction

Realizace v případě sítě typu **lineární řetězec** je stejná jako u kruhu.

- poslední spoj nebyl potřeba

Realizace v případě **ortogonální sítě**:

- nejprve se provede *one-to-all broadcast* podél jedné souřadnice, následně podél další

One-to-all broadcast/All-to-one reduction v MPI

One-to-all broadcast

```
1  int MPI_Bcast( void* buf, int count, MPI_Datatype,
2                 int source, MPI_Comm comm )
```

```
1  MPI_Bcast( &x, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD );}
```

All-to-one reduction

```
1  int MPI_Reduce( void* sendbuf, void* recvbuf,
2                  int count, MPI_Datatype datatype,
3                  MPI_Op op, int target, MPI_Comm comm)
```

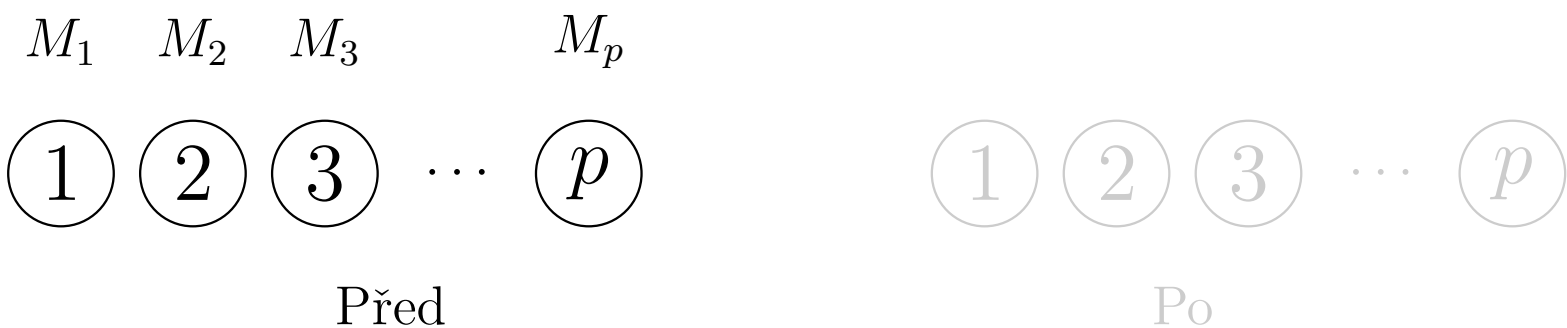
- op = operace: `MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD`
- výsledek se uloží do `recvbuf` procesu `target`

```
1  MPI_Reduce( &x, &max_x, 1, MPI_DOUBLE, MPI_MAX,
2              0, MPI_COMM_WORLD );
```

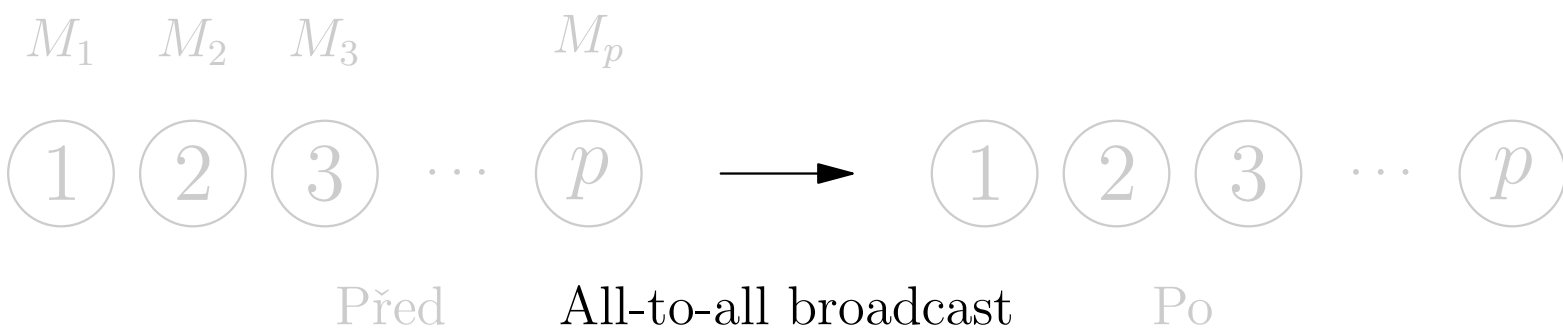
All-to-all broadcast/All-to-all reduction

Všech p procesů provádí současně *one-to-all broadcast* nebo *all-to-one reduction* s různými daty.

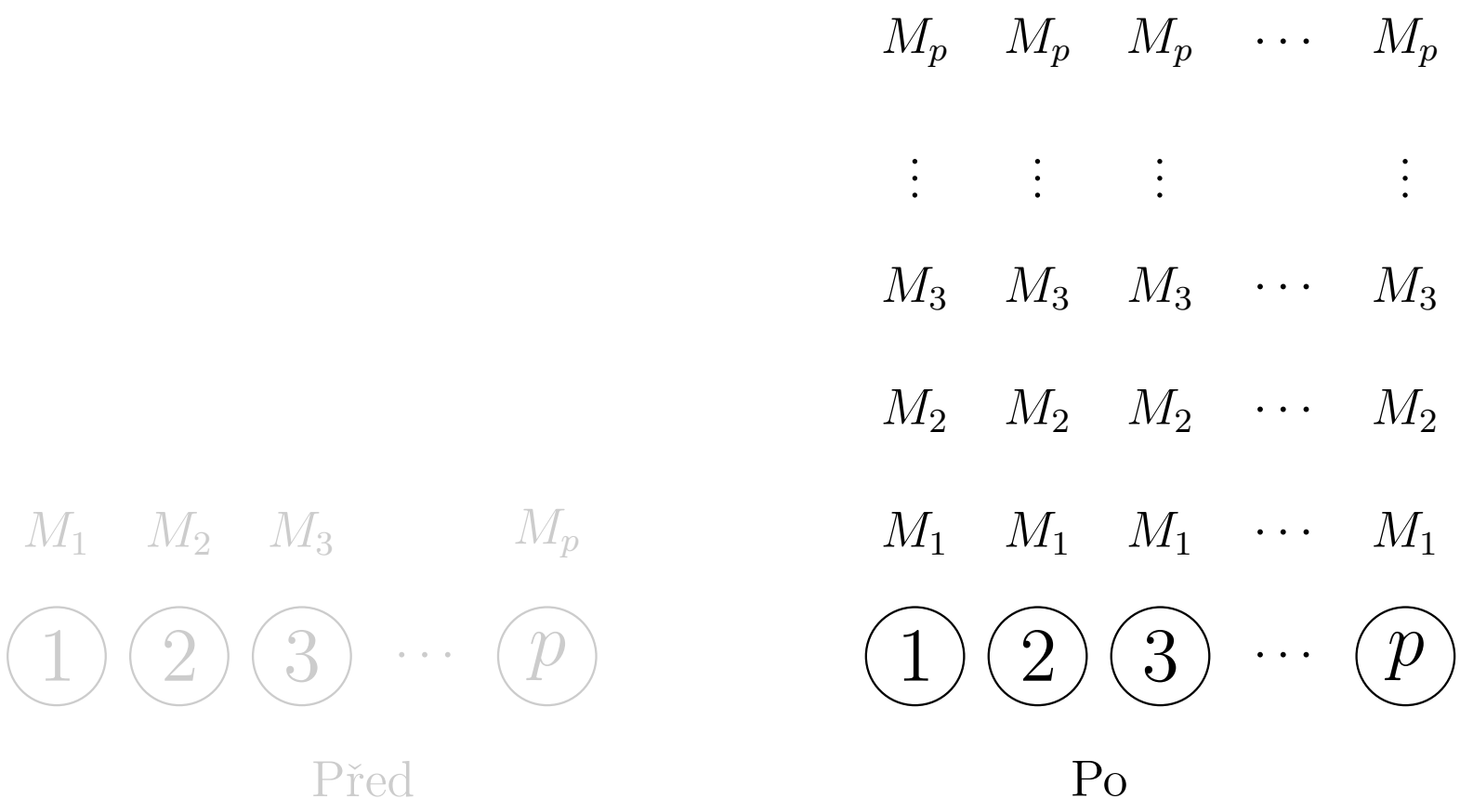
All-to-all broadcast/All-to-all reduction



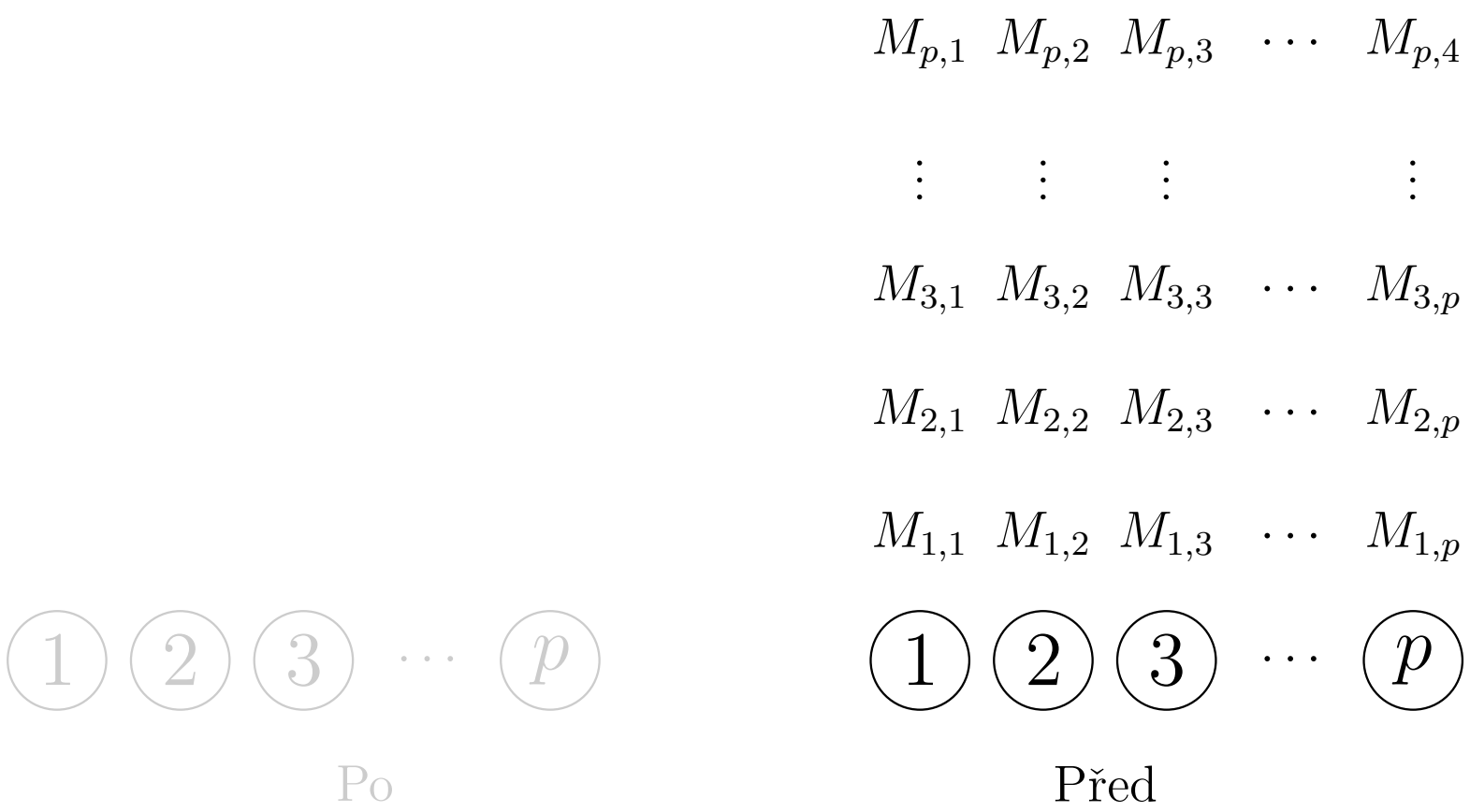
All-to-all broadcast/All-to-all reduction



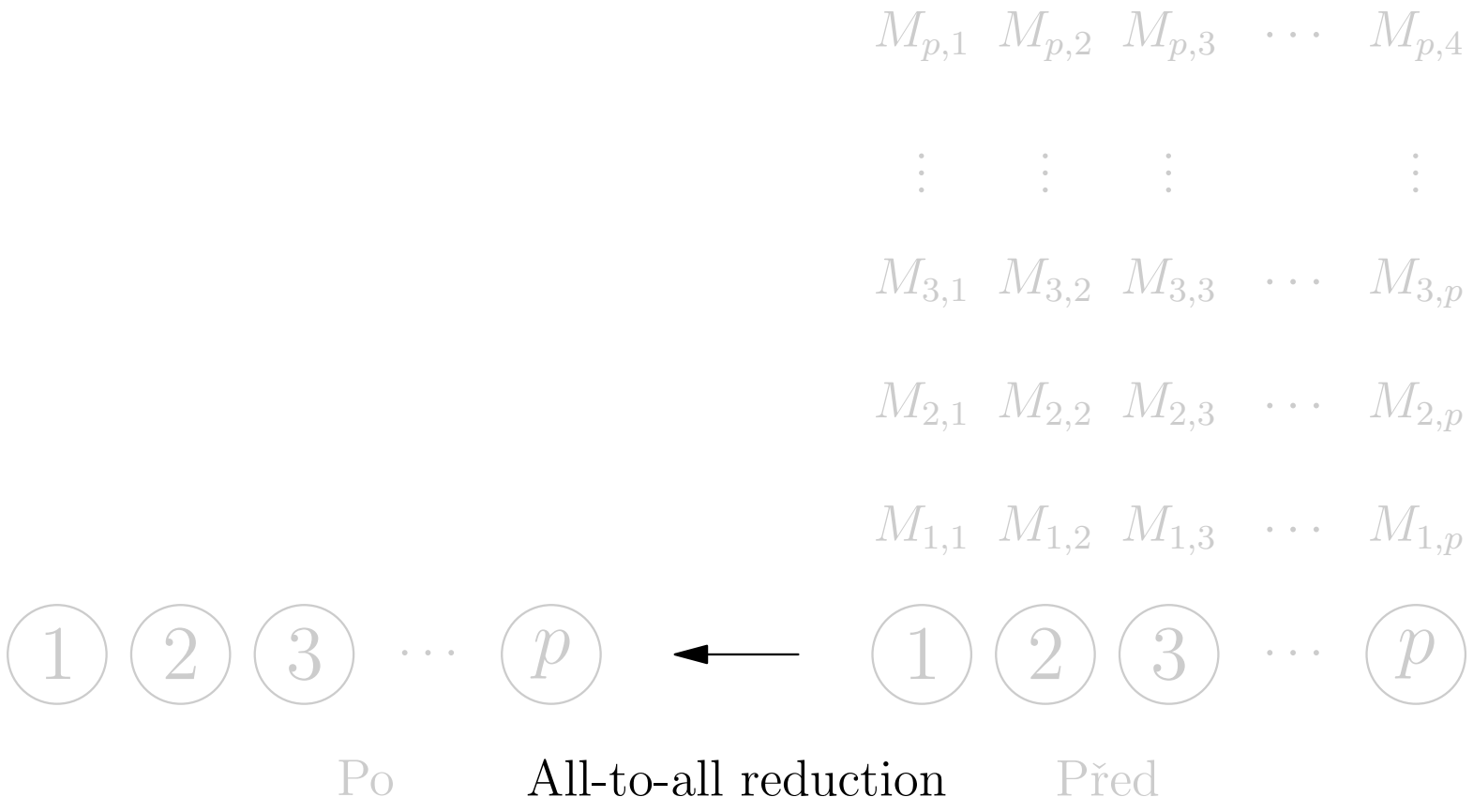
All-to-all broadcast/All-to-all reduction



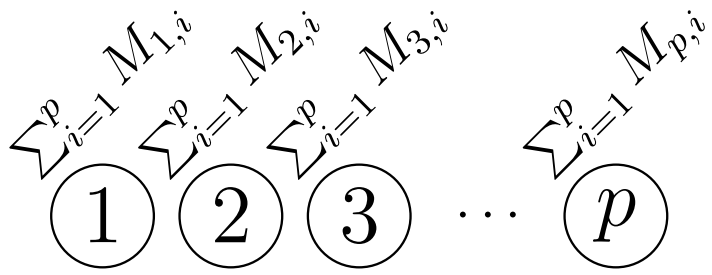
All-to-all broadcast/All-to-all reduction



All-to-all broadcast/All-to-all reduction



All-to-all broadcast/All-to-all reduction



Po

$$\begin{matrix} M_{p,1} & M_{p,2} & M_{p,3} & \cdots & M_{p,4} \\ \vdots & \vdots & \vdots & & \vdots \\ M_{3,1} & M_{3,2} & M_{3,3} & \cdots & M_{3,p} \\ M_{2,1} & M_{2,2} & M_{2,3} & \cdots & M_{2,p} \\ M_{1,1} & M_{1,2} & M_{1,3} & \cdots & M_{1,p} \\ 1 & 2 & 3 & \cdots & p \end{matrix}$$

Před

All-to-all broadcast/All-to-all reduction

Realizace *all-to-all broadcast* v případě sítě typu **kruh**:

- v každém kroku každý uzel pošle svoje data sousedovi vpravo a přijme data od souseda vlevo
- v následujícím kroku posílá data, která v předchozím kroku přijal
- vše se opakuje $p - 1$-krát

V případě **ortogonálních sítí** se opět postupuje podél jednotlivých souřadnic.

All-to-all broadcast/All-to-all reduction

Realizace *all-to-all reduction* v případě sítě typu **kruh**:

- v první kroku pošle uzel i data $M_{j,i}$ pro $j = 1, \cdots p$ a $j \neq i$ svému sousedovi vpravo a příjme data $M_{j,i-1}$ pro $j = 1, \cdots p$ a $j \neq i - 1$ od svého souseda vlevo
- v druhém kroku pošle uzel i data $M_{j,i-1}$ pro $j = 1, \cdots p$ a $j \neq i \wedge j \neq i - 1$, svému sousedovi vpravo a příjme data $M_{j,i-2}$ pro $j = 1, \cdots p$ a $j \neq i - 1 \wedge j \neq i - 2$ od svého souseda vlevo
- vše se ještě opakuje $p - 2$-krát za současného provádění redukční operace

All-to-all broadcast/All-to-all reduction v MPI

All-to-all broadcast

```
1  int MPI_Alltoall( void *sendbuf, int sendcount,
2                    MPI_Datatype sendtype,
3                    void *recvbuf, int recvcount,
4                    MPI_Datatype recvtype,
5                    MPI_Comm comm )
```

- `sendcount` a `recvcount` je počet elementů poslaných jednomu procesu

All-reduce a Prefix-sum

All-reduce

- na počátku má každý proces různá data M_i o velikosti m
- na konci mají všechny procesy stejný „součet“ dat $\sum_{i=1}^{p} M_i$
- All-reduce lze provést jako
  - All-to-one reduction + One-to-all broadcast
  - All-to-all broadcast, kde se ale provede součet $\sum_{i=1}^{p} M_i$
- pro $m = 1$ lze All-reduce použít jako bariéru [1]
  - redukci nelze dokončit, dokud k ní nepřispěje každý proces svým podílem

[1] **Bariéra** je místo v programu, které nesmí žádný proces překročit, dokud ho nedosáhnou všechny ostatní procesy. Má význam synchronizace procesů.

All-reduce a Prefix-sum

Prefix-sum

- jde o modifikaci all-reduce
- pro data $M_1, \cdots M_p$ chceme najít $S_k = \sum_{i=1}^{k} M_i$ pro $k = 1, \cdots p$
- postup je stejný jako u all-reduce, ale každý proces k si "přičítá"jen data od procesů s ID $< k$
- některá komunikace je tu tady zbytečná, ale celkovou složitost této operace to neovlivní

All-reduce a Prefix-sum v MPI

All reduce

```
1  int MPI_Allreduce( void* sendbuf, void* recvbuf,
2                     int count, MPI_Datatype datatype,
3                     MPI_Op op, MPI_Comm comm )
```

Prefix sum

```
1  int MPI_Scan( const void *sendbuf, void *recvbuf, int count,
2                MPI_Datatype datatype, MPI_Op op, MPI_Comm comm )
```

Bariéra

```
1  int MPI_Barrier( MPI_Comm comm )
```

Scatter/Gather

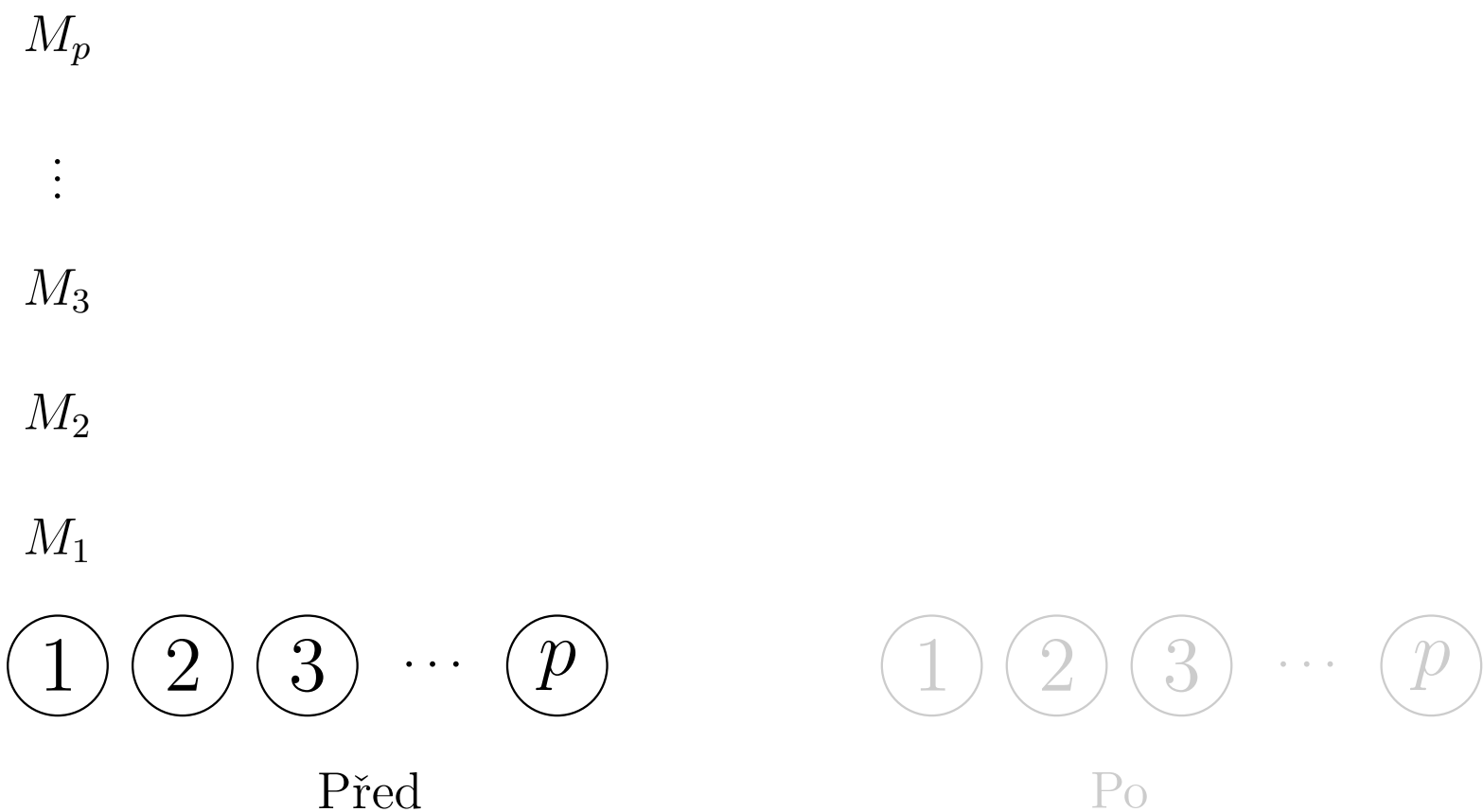
Scatter

- jeden proces má na počátku p různých zpráv $M_1, \cdots M_p$
- proces i má na konci zprávu M_i
- někdy se tato zpráva nazývá **one-to-all personalized communicatio**

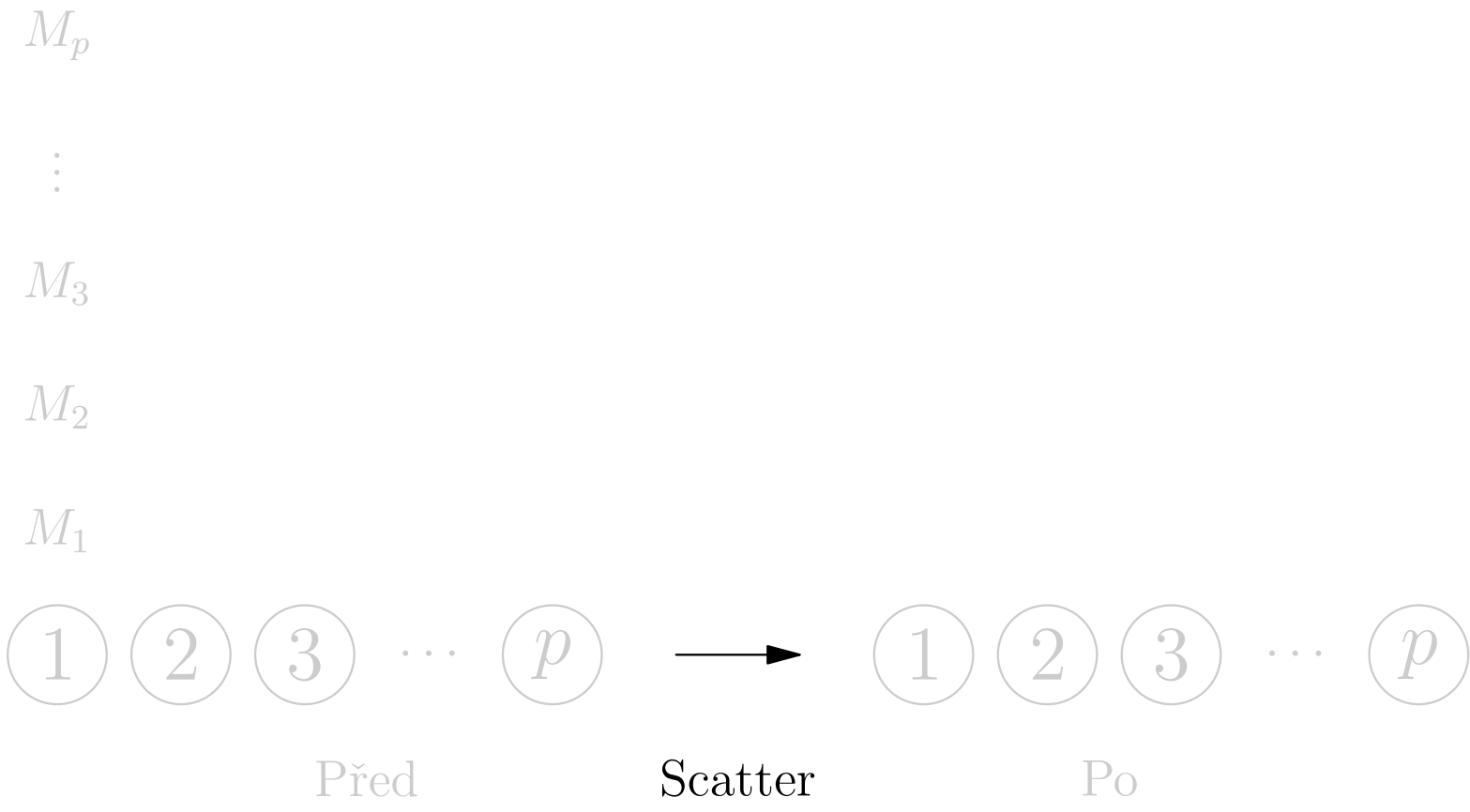
Gather

- je to duální operace ke *scatter*
- na počátku má každý proces i zprávu M_i
- na konci má jeden proces všechny zprávy $\cap_{i=1}^{p} M_i$

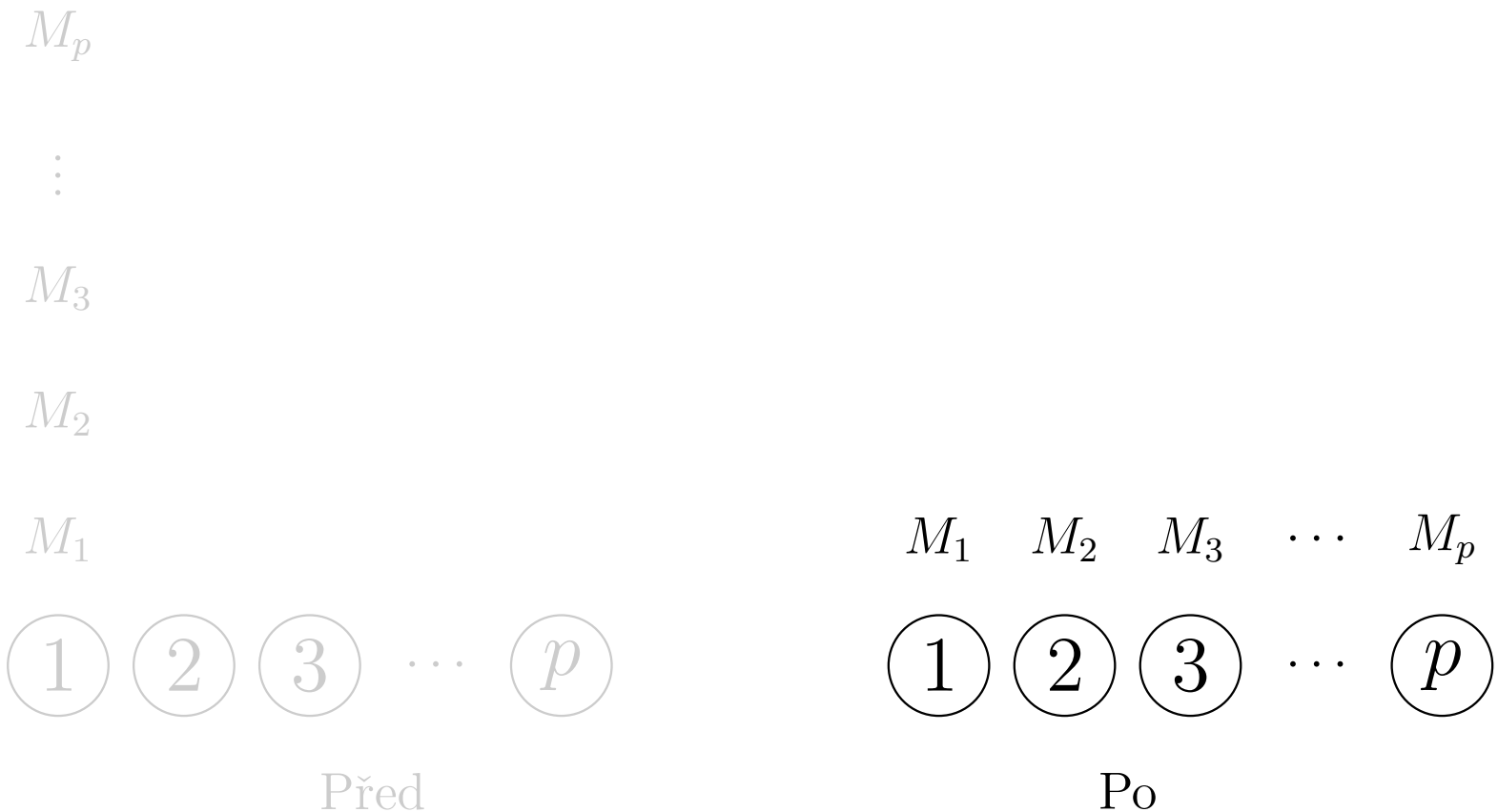
Scatter/Gather



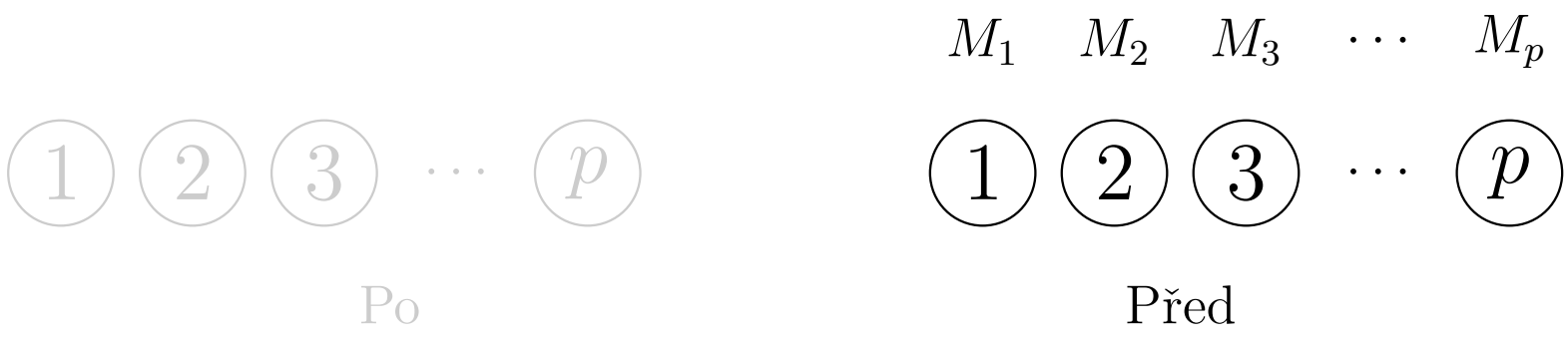
Scatter/Gather



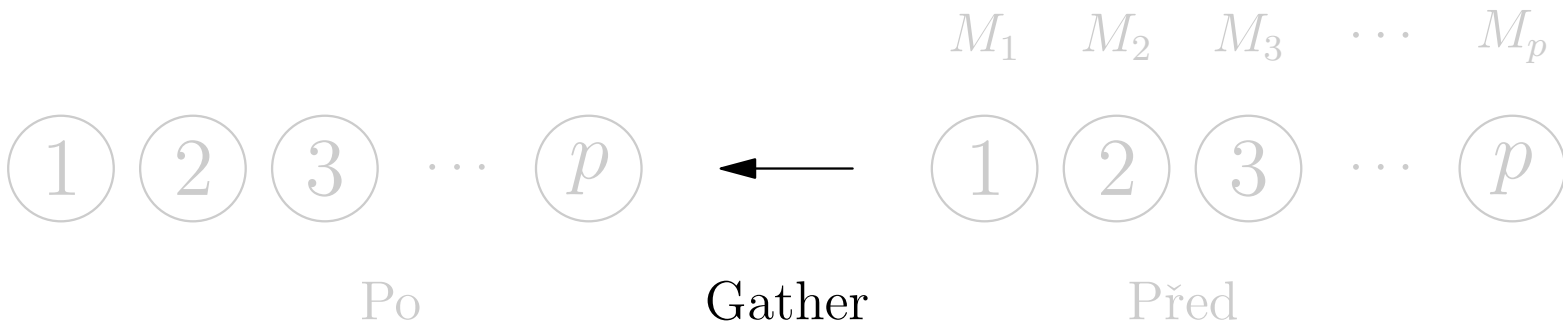
Scatter/Gather



Scatter/Gather



Scatter/Gather



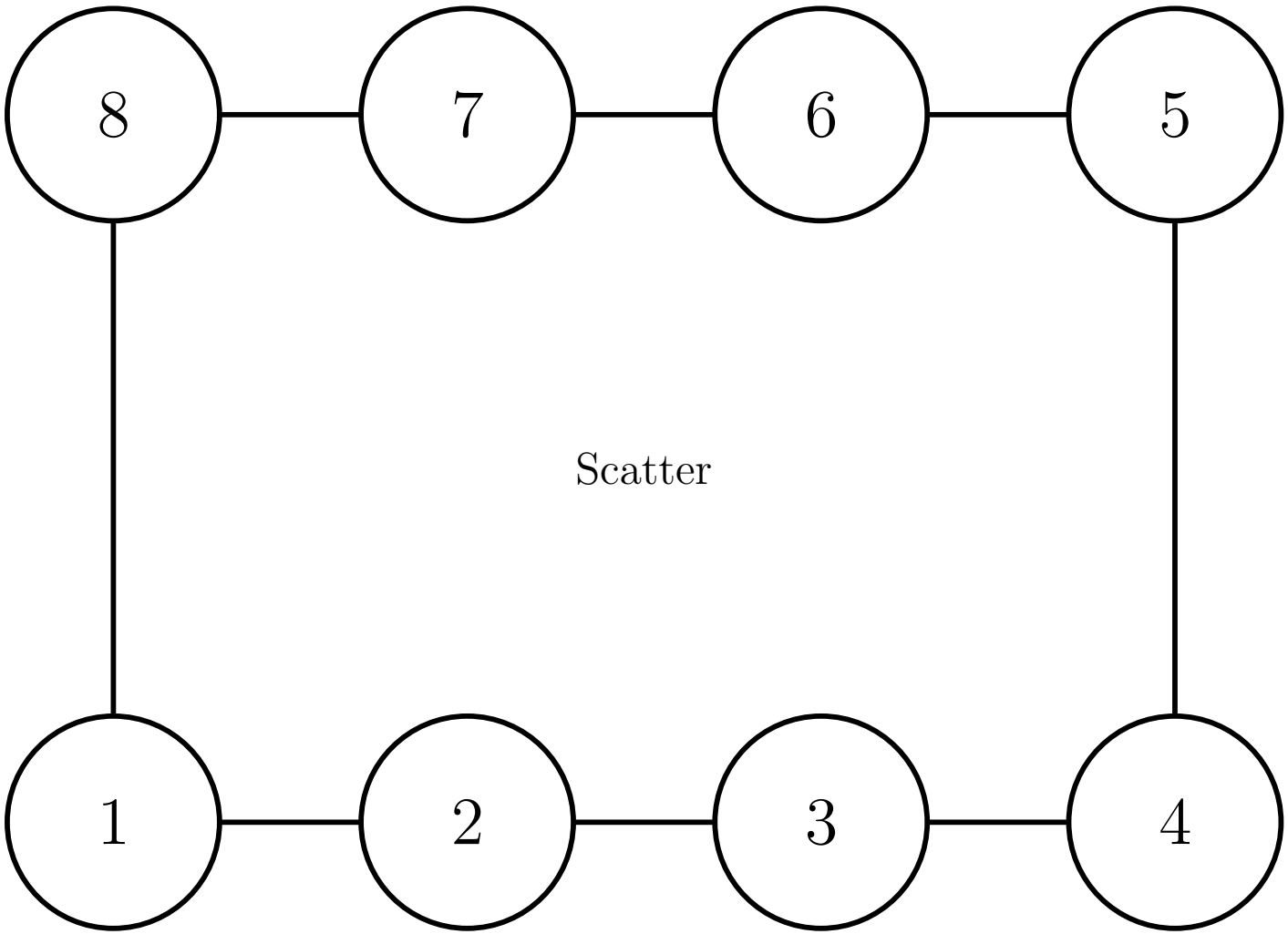
Scatter/Gather

Scatter/gather

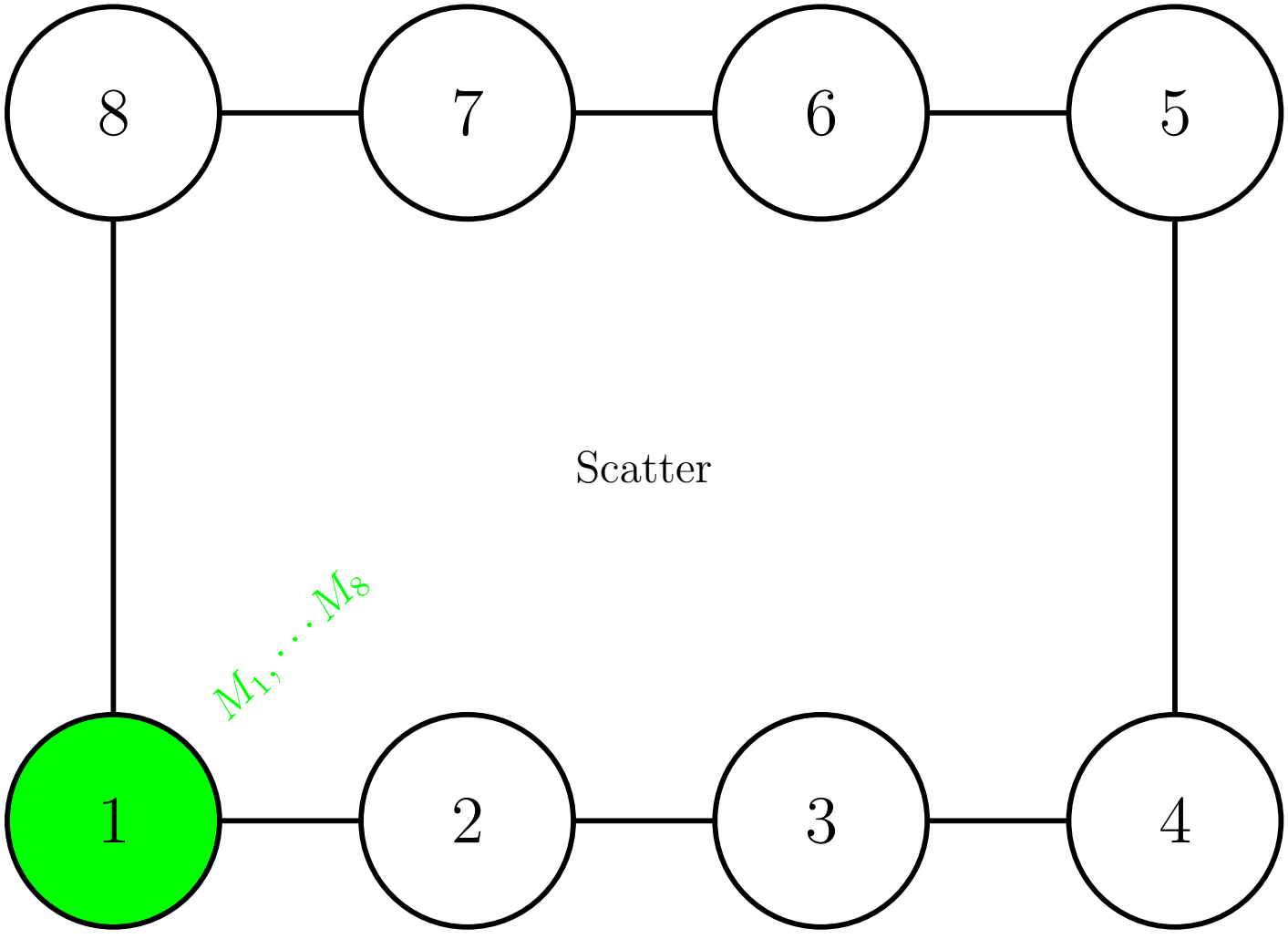
Realizace *scatter* v případě sítě typu **kruh**:

- je stejná jako u *one-to-all broadcast* ale s jinými objemy dat

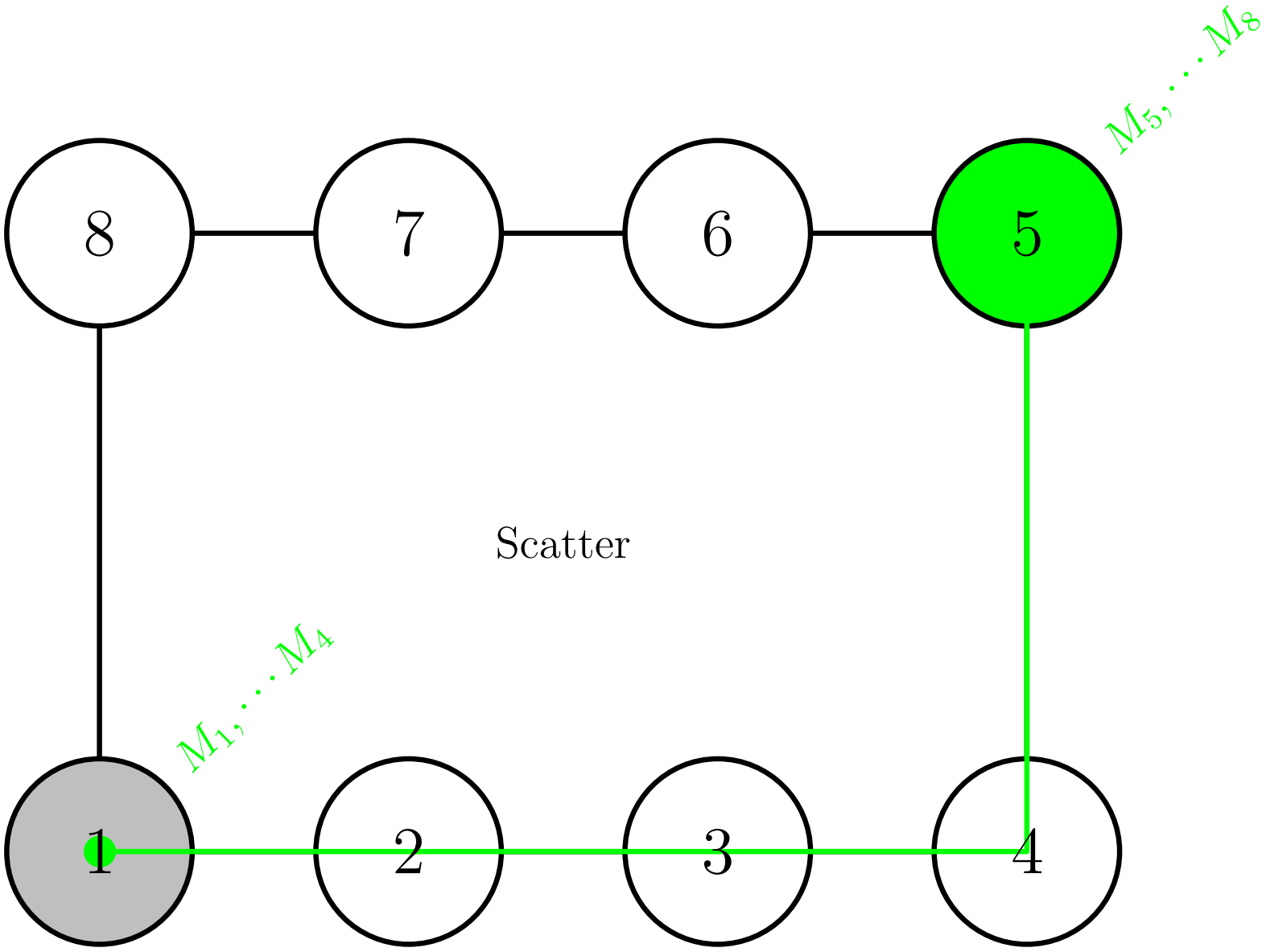
Scatter/gather



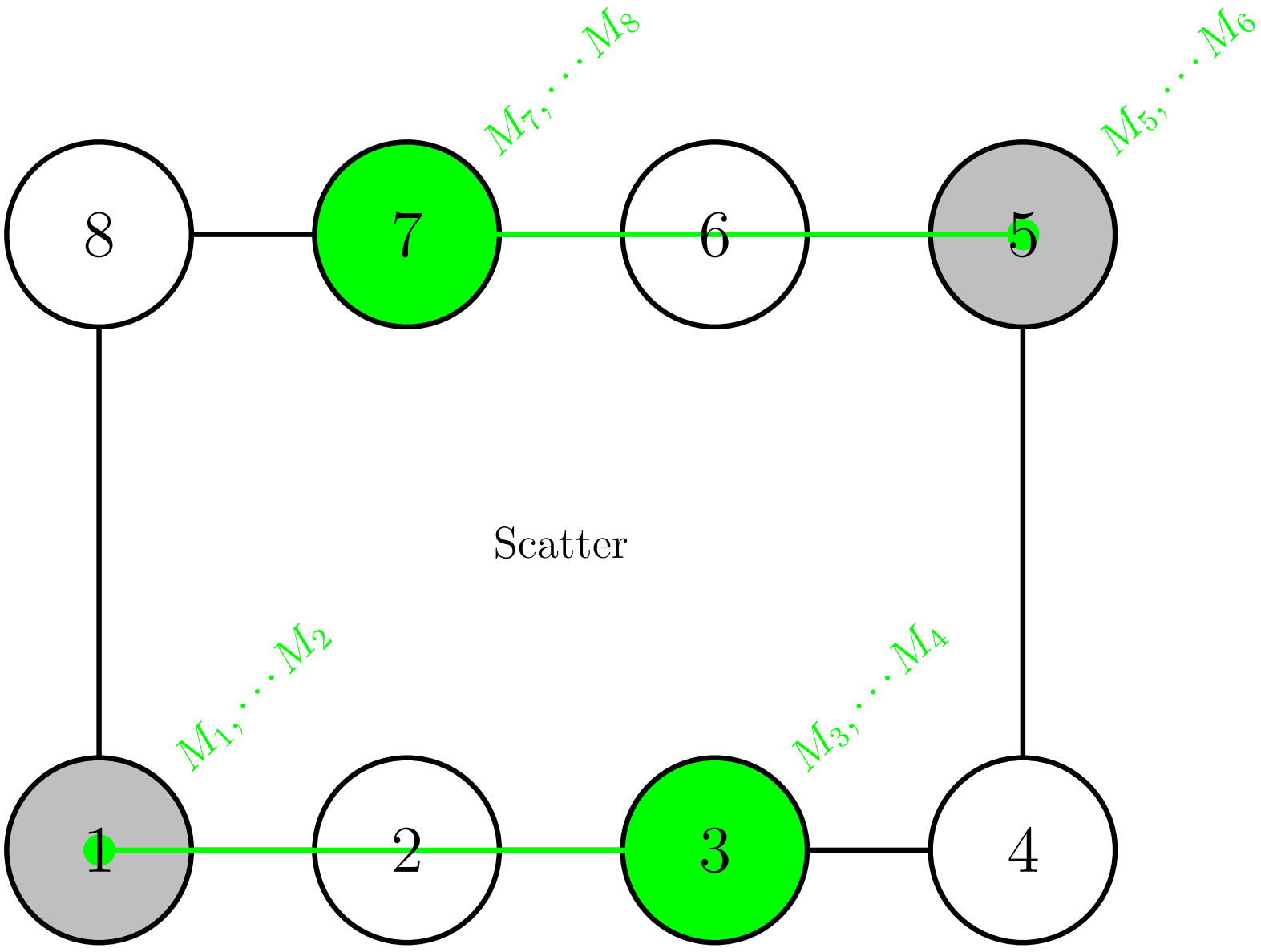
Scatter/gather



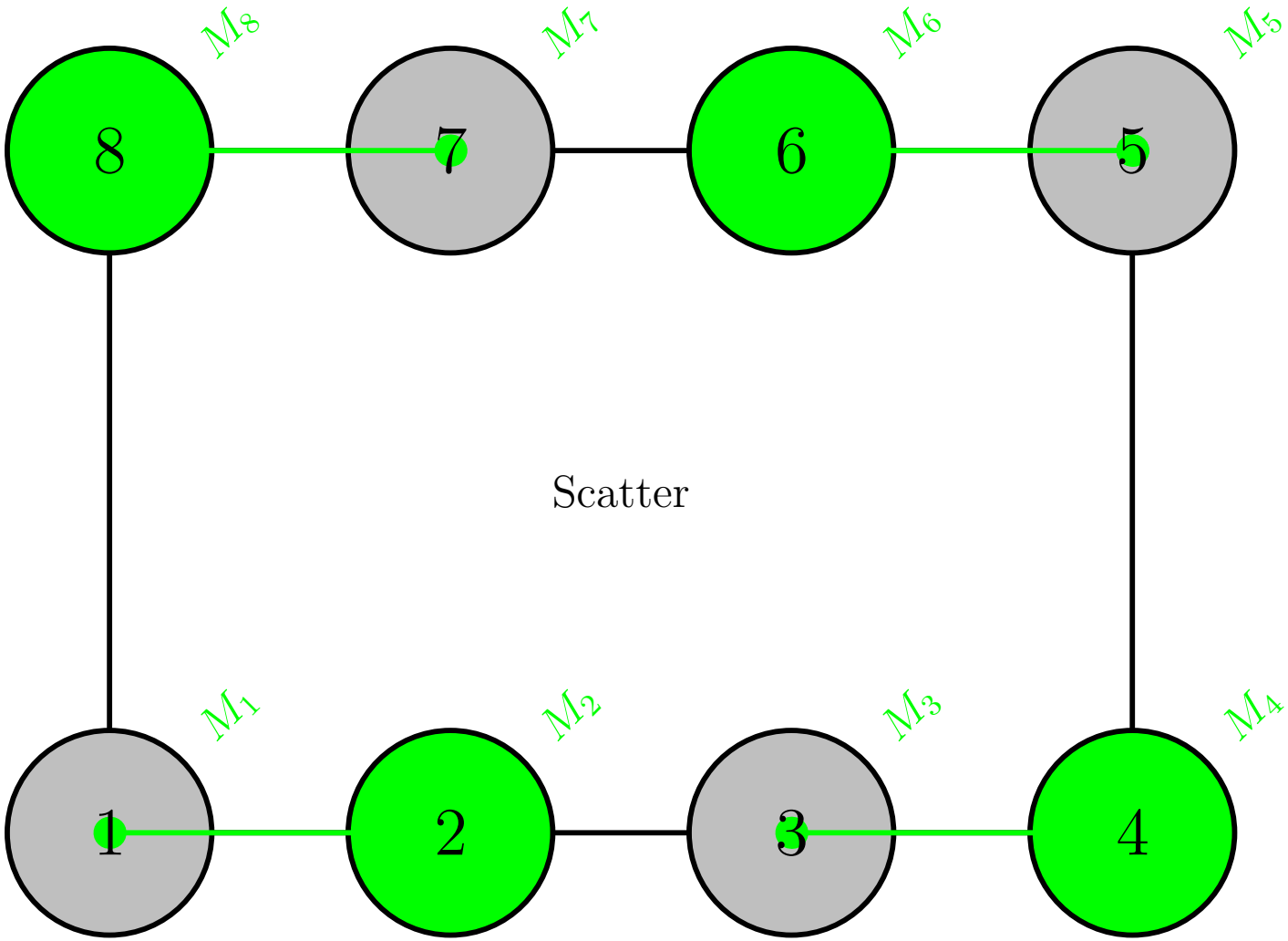
Scatter/gather



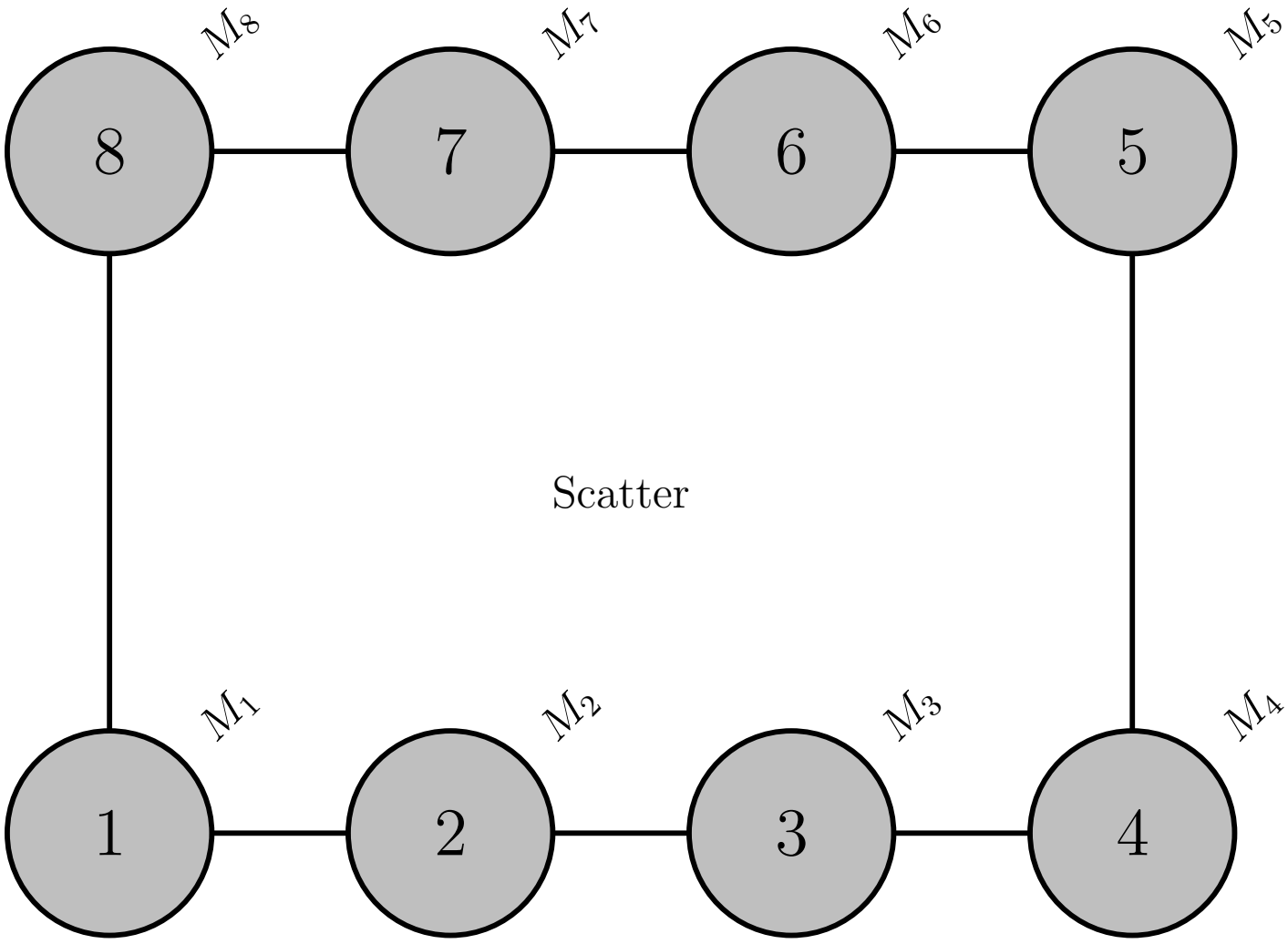
Scatter/gather



Scatter/gather



Scatter/gather



Scatter/gather v MPI

Scatter

```
1  int MPI_Scatter( void* sendbuf, int sendcount,
2                   MPI_Datatype senddatatype,
3                   void* recvbuf, int recvcount,
4                   MPI_Datatype recvdatatype,
5                   int source, MPI_Comm comm )}
```

- `sendcount = recvcount` udává počet prvků posílaných jednomu procesu

Scatter vektorově - každý proces dostane jiný objem dat

```
1  int MPI_Scatterv( void* sendbuf, int* sendcounts,
2                    int* displs,
3                    MPI_Datatype senddatatype,
4                    void* recvbuf, int recvcount,
5                    MPI_Datatype recvdatatype,
6                    int source, MPI_Comm comm )
```

- `sendcounts` ukazatel na pole udávájící počet prvků posílaných danému procesu
- `displs` ukazatel na pole udávající pozici dat pro daný proces

Scatter/gather v MPI

Gather

```
1  int MPI_Gather( void* sendbuf, int sendcount,
2                  MPI_Datatype senddatatype,
3                  void* recvbuf, int recvcount,
4                  MPI_Datatype recvdatatype,
5                  int target, MPI_Comm comm )
```

- `recvcount` udává počet prvků získaných od daného procesu

Gather vektorově - každý proces dostane jiný objem dat

```
1  int MPI_Scatterv( void* sendbuf, int sendcount,
2                    MPI_Datatype senddatatype,
3                    void* recvbuf, int* recvcount,
4                    int* displs,
5                    MPI_Datatype recvdatatype,
6                    int target, MPI_Comm comm )}
```

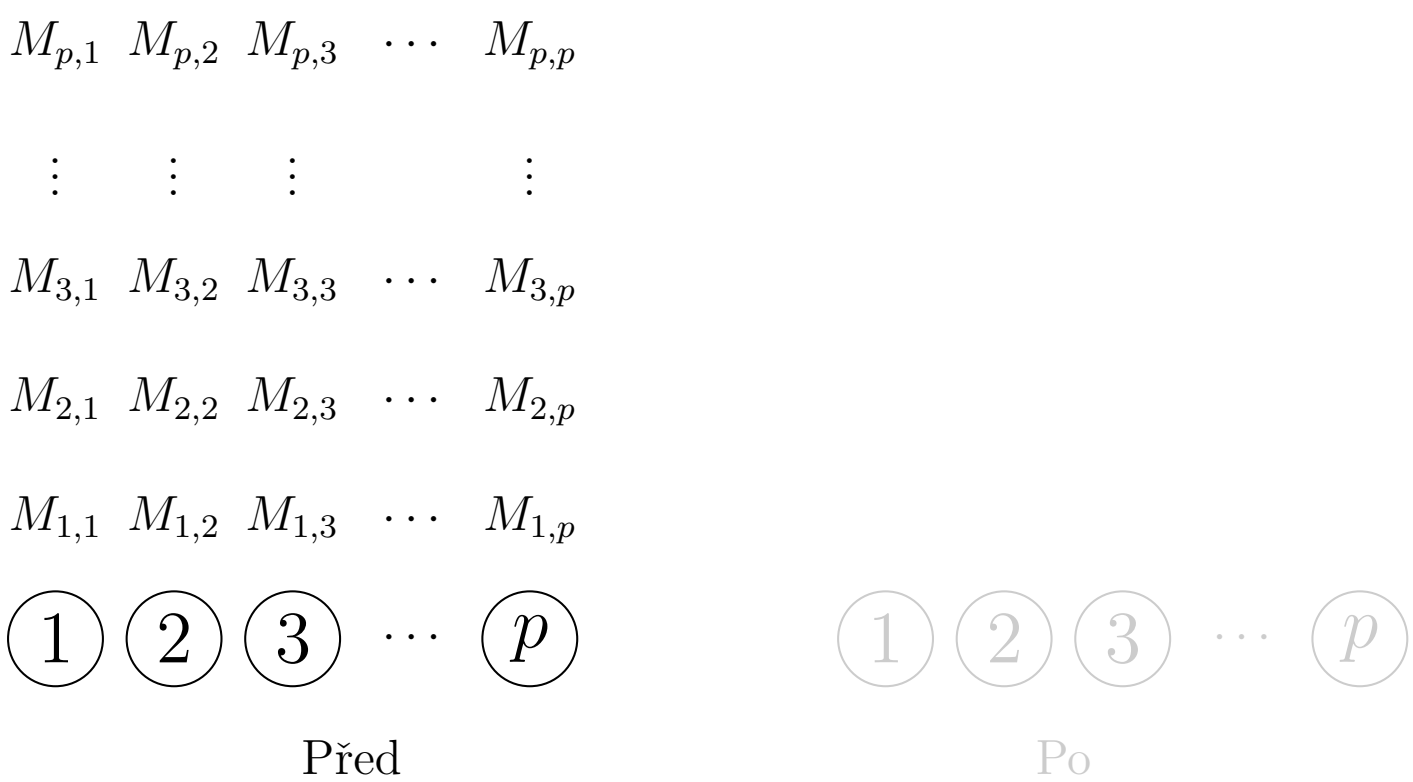
- `recvcounts` ukazatel na pole udávájící počet prvků získaných od daného procesu
- `displs` ukazatel na pole udávající pozici dat od daného procesu

All-to-all personalized communication

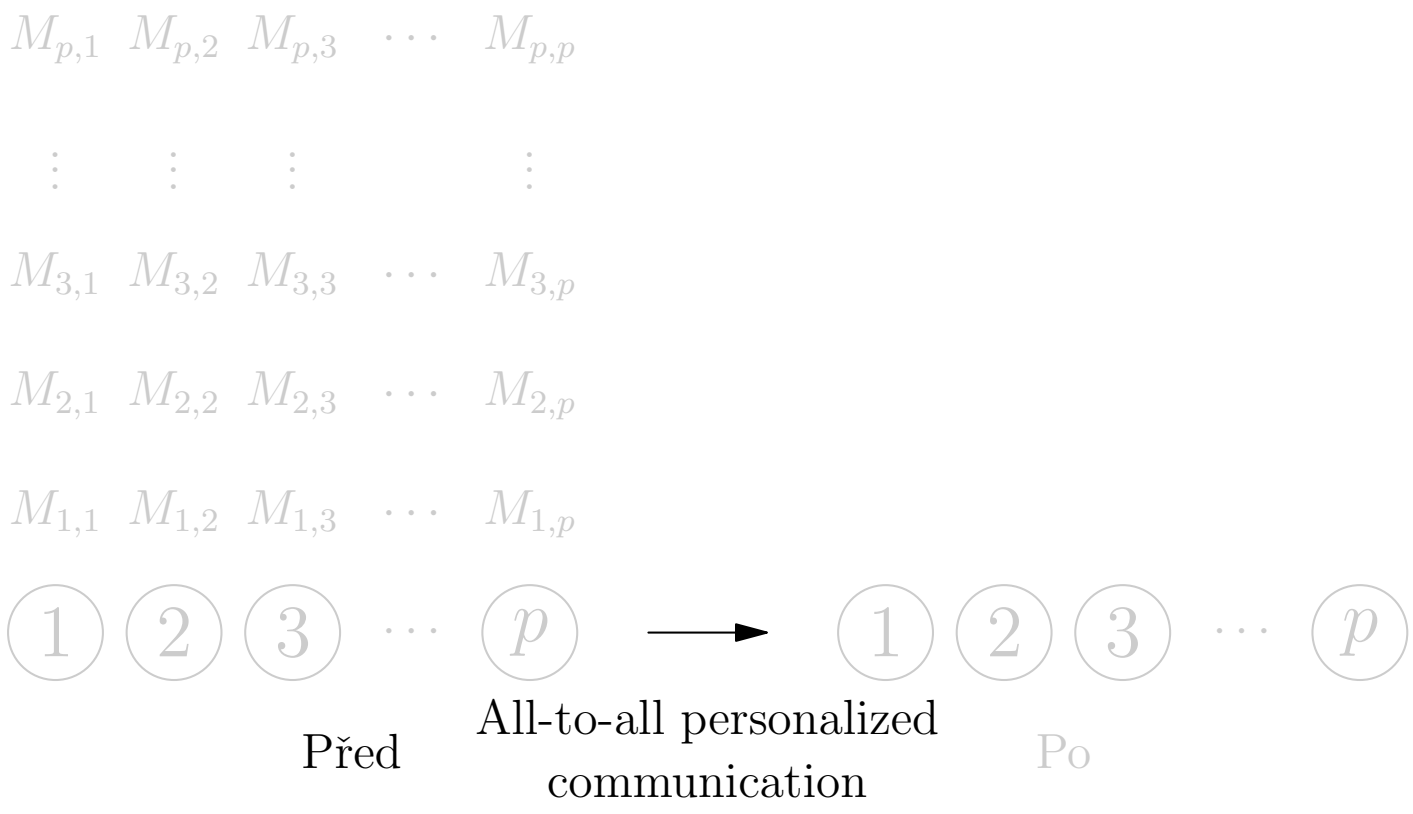
All-to-all personalized communication

- každý proces i má data $M_{i,j}$, která pošle procesu j.

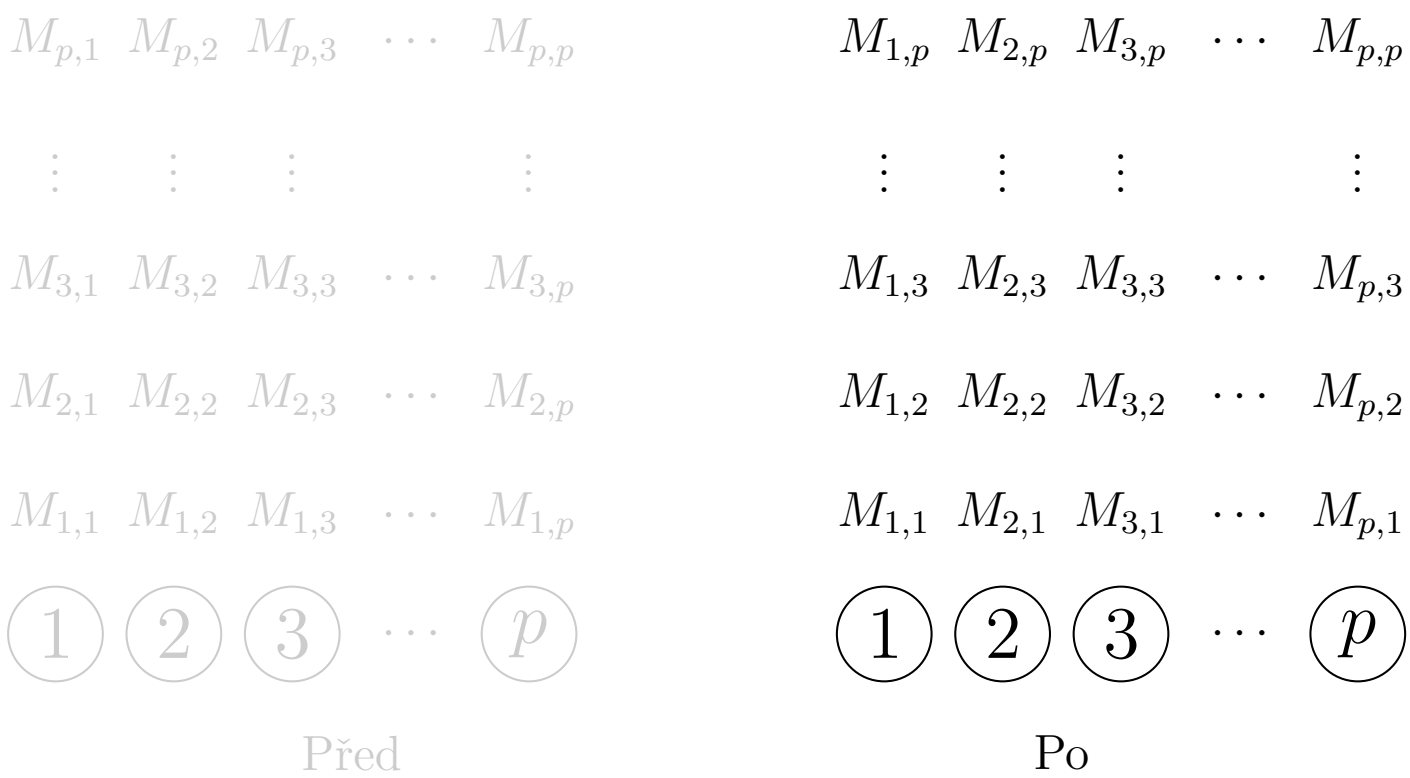
All-to-all personalized communication



All-to-all personalized communication



All-to-all personalized communication



All-to-all personalized communication

- jde vlastně o maticovou transpozici
- stejná analogie jako mezi *scatter* a *one-to-all broadcast* platí i mezi *all-to-all personalized communication* a *all-to-all broadcast*
- z toho plyne i realizace *all-to-all personalized communication* na síti typu **kruh** nebo na ortogonálních sítích

All-to-all personalized communication v MPI

All-to-all personalized communication

```
1  int MPI_Alltoall( void* sendbuf, int sendcount,
2                    MPI_Datatype senddatatype,
3                    void* recvbuf, int recvcount,
4                    MPI_Datatype recvdatatype,
5                    MPI_Comm comm)
```

```
1  int MPI_Alltoallv( void* sendbuf, int* sendcounts,
2                     int* sdispls,
3                     MPI_Datatype senddatatype,
4                     void* recvbuf, int* recvcounts,
5                     int* rdispls,
6                     MPI_Datatype recvdatatype,
7                     MPI_Comm comm)
```

Circular shift

Circular shift

- patří mezi tzv. **permutační komunikační operace**
- každý proces pošle jednu zprávu o velikosti m slov některému jinému uzlu

Příklad: Circular q-shift

- proces i pošle svá data procesu $(i + q) \mod p$

Kostra jednoduché aplikace

```
 1  #include <mpi.h>
 2
 3  int main( int argc, char* argv[] )
 4  {
 5      int nproc, iproc;
 6      Config config;
 7      InputData input_data;
 8      OutputData output_data;
 9      MPI_Init(&argc, &argv);
10      MPI_Comm_size(MPI_COMM_WORLD, &nproc);
11      MPI_Comm_rank(MPI_COMM_WORLD, &iproc);
12      if(iproc == 0)
13      {
14          ParseConfigurationParameters(&config,&argc,&argv);
15          GetInputData(&config,&input_data);
16      }
17      Broadcast(&config, 0);
18      Scatter(&input_data, 0);
19      Compute(&input_data, &output_data);
20      Gather(&output_data, 0);
21      if(iproc == 0) WriteOutput(&output_data);
22      MPI_Finalize();
23  }
```

Spouštění MPI aplikací

Programy pro MPI většinou překládáme pomocí `mpicc`

```
1  mpicc -o foo foo.c
```

a spouštíme pomocí

```
1  mpirun -v -np 2 foo
```

- program `mpirun` spustí `foo` ve dvou procesech

Spouštění MPI aplikací

- MPI aplikace lze dobře provozovat i na vícejádrových procesorech
- v případě výpočtu na klastru nebo superpočítači je často potřeba udat více parametrů

```
1  mpirun -np 256 -npernode 4  -machinefile file foo
```

- `-npernode` udává počet procesů na jeden uzel
- `-machinfile` udává adresy jednotlivých uzlů

```
1  node328
2  node328
3  node328
4  node328
5  node329
6  node329
7  node329
8  node329
9  node330
10 node330
11 node330
12 node330
13 ...
```