# Jednostranné komunikace v MPI
# PPP 05

Gabriela Nečasová

Brno University of Technology, Faculty of Information Technology
Božetěchova 1/2, 612 66 Brno - Královo Pole
inecasova@fit.vut.cz

# Úvod

- **Jednostranné komunikace: Hodí se pro projekt – RMA komunikace!**
  - → jeden partner zadává co se má dělat a druhý pouze pasivně otevírá/uzavírá okno,do kterého je povoleno zapisovat
  - Skupina funkcí: MPI_Win_*()
  - MPI_INFO_NULL (pokud se nám nehodí nic pro MPI_Info)
- **Úkoly:**
  - Transpozice matice (Put a Get)
  - Transpozice matice, optimalizace okna a paměťových zábran
  - Distribuovaný histogram pomocích atomických akumulací
  - Distribuovaný histogram s lokálním scratch
  - Distribuované násobení matic

- Kde počítat?
  - Barbora/Karolina/Merlin – možno vypracovat celé cvičení
  - Poznámka: potřebujeme 16 procesů, na Merlinovi je potřeba `--oversubscribe`

- Cvičení jsou dobrovolná, ovšem mohou výrazně usnadnit řešení projektu ☺
- **Nápověda**:
  - Přednáška PPP 5
  - Open MPI dokumentace: https://www-lb.open-mpi.org/doc/v4.1/
    - Seznam všech MPI rutin (MPI_Init,…)

# Úvod

- Připojení na server?
  - `ssh [merlin|barbora|karolina]`
- Ověření, že je k dospozici MPI:
  - `mpicc --help`

- Pokud pracujete na Barboře/Karolině:
  - Karolina
    - `$ salloc -A DD-24-108 -p qcpu_exp -N 1 --ntasks-per-node 128 -t 01:00:00 --x11`
  - Barbora
    - `$ salloc -A DD-24-108 -p qcpu_exp -N 1 --ntasks-per-node 36 -t 01:00:00 --x11`
  - `ml OpenMPI`

- Zobrazení info o MPI instalaci:
  - `ompi_info`

# Překlad a spuštění

- 1. možnost: cmake:
  - `$ cmake -Bbuild -S.`
  - `$ cmake --build build --config Release`
- 2. možnost: mpicc:
  - `mpic++ one.cpp –o one`
- Aplikace má 1 parametr – číslo úlohy (zde 1):
  - `mpirun –np 16 ./one 1`

# Základní funkce

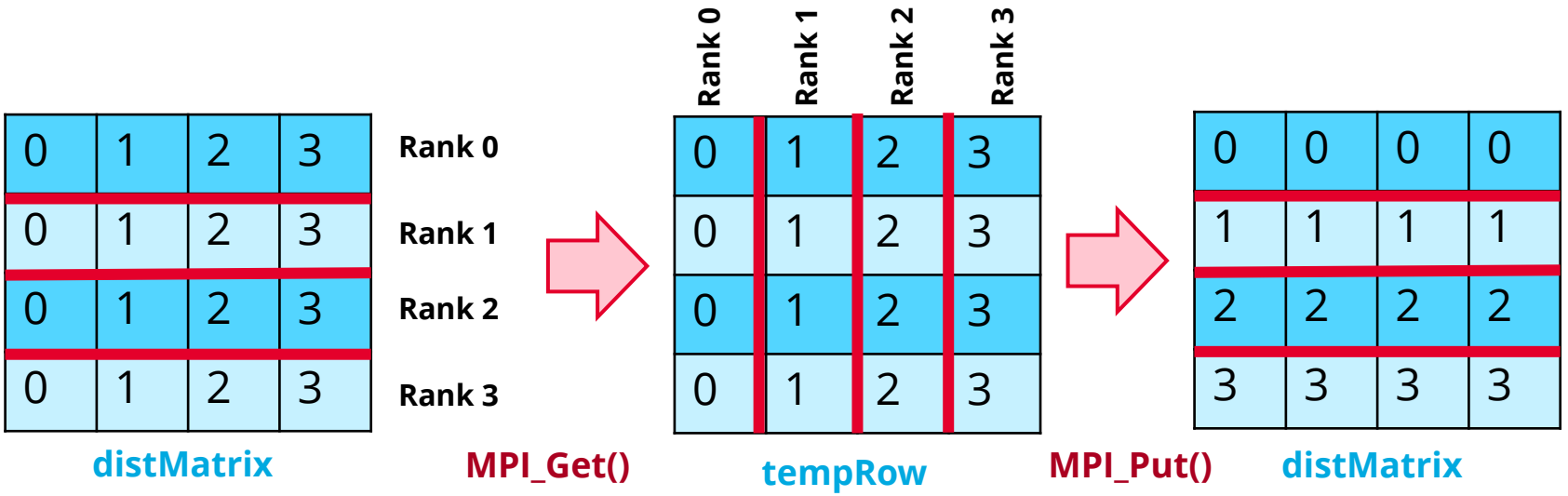
- **initMatrix()**
  - Inicializace: mpiGetCommRank(MPI_COMM_WORLD) * 10000 + i * 100 + j
- **clearMatrix()**
- **printMatrix()**
  - Slouží pro kontrolu distribuce dat

- **histogramHash()**
- **printHistogram()**

```
----------------------- Original matrix ----------------------------
 - Rank  1, row  0 = [   10000,    10001,    10002,    10003,    10004,    10005,    10006,    10007]
 - Rank  2, row  0 = [   20000,    20001,    20002,    20003,    20004,    20005,    20006,    20007]
 - Rank  3, row  0 = [   30000,    30001,    30002,    30003,    30004,    30005,    30006,    30007]
 - Rank  4, row  0 = [   40000,    40001,    40002,    40003,    40004,    40005,    40006,    40007]
 - Rank  5, row  0 = [   50000,    50001,    50002,    50003,    50004,    50005,    50006,    50007]
 - Rank  6, row  0 = [   60000,    60001,    60002,    60003,    60004,    60005,    60006,    60007]
 - Rank  7, row  0 = [   70000,    70001,    70002,    70003,    70004,    70005,    70006,    70007]
 - Rank  0, row  0 = [       0,        1,        2,        3,        4,        5,        6,        7]
```

# ÚKOL 1: TRANSPOZICE MATICE

# 1: Transpozice matice




- Každý rank má řádek matice: **distMatrix** a pomocný řádek **tempRow**
  - **Ranks 0-3:** [0 1 2 3]→ distMatrix
- Každý rank se podívá k sousedům a vytáhne si odpovídající sloupec – pomocí **MPI_Get()** získá data
  - **Rank 0**: [0 0 0 0] → tempRow
  - **Rank 1**: [1 1 1 1] → tempRow
  - **Rank 2**: [2 2 2 2] → tempRow
  - **Rank 3**: [3 3 3 3] → tempRow
- Proč potřebuji tempRow? Protože v případě MPI_Put() a MPI_Get() nemohu pracovat „in place“, došlo by k pomíchání
- 1. Deklarujeme si okno – umožní vystavit všem ostatním rankům každý řádek
- 2. Alokujeme **tempRow** a **distMatrix**. Potom alokujeme řádek distribuované matice, naplníme ji
- 3. Vytvoříme okno **(MPI_Win_create())**
- 4. Ve smyčce voláme **MPI_Get()** – projdeme partnery a získáme dopovídající sloupec - do tempRow
- 5. Zkopírujeme sloupec zpět do okna: **MPI_Put()**
- 6. Abychom ověřili funkčnost, transponujeme matici do původního stavu pomocí **MPI_Alltoall()**

# 1: Transpozice matice

Cílem tohoto příkladu je vyzkoušet si tvorbu okna (`MPI_Win_create` a `MPI_Alloc_mem`), paměťové zábrany (`MPI_Win_fence`), a operace Put a Get (`MPI_Put`, `MPI_Get`). Zadání se nachází pod sekcí `case 1:` ve funkci `main`.
Mějme matici $P \times P$, kde $P$ je počet procesů. Každý rank drží jeden řádek. Vašim úkolem je matici transponovat tak, aby každý rank držel jeden sloupec. Pro ověření pak transponujeme matici zpět pomocí vhodné kolektivní komunikace.

1. Nejprve deklarujte objekt okna, který bude obsahovat váš řádek matice.
2. Následně pomocí vhodné MPI funkce alokujte paměť pro okno. Alokujte ještě další pomocné pole, které bude obsahovat natažené hodnoty z okna.
3. Po inicializaci a vyčištění alokovaných objektů přichází tvorba okna. Pomocí vhodné funkce vytvořte MPI okno a připojte k němu paměť, která drží váš řádek dat.
4. Pomocí operace Get sesbírejte od sousedů prvky ve sloupci, který vám náleží (rank 0 má sloupec 0, rank 1 sloupec 1, atd.). Jednotlivé prvky uložte do svého pomocného pole.
5. Nyní pomocí operace Put vložte data zpět do okna. Stačí jedno volání funkce!
6. Nyní proveďte zpětnou transpozici pomocí odpovídající kolektivní komunikace.
7. Uvolněte všechny MPI objekty a použitá pole.
8. Přeložte soubor:.
9. Spusťte výslednou binárku:

```
$ mpiexec -np 2 ./one 1
$ mpiexec -np 4 ./one 1
$ mpiexec -np 8 ./one 1
$ mpiexec -np 16 ./one 1
```

10. Porovnejte výsledky všech operací.

# 1: Transpozice matice

- // 1. Declare an MPI window for the distMatrix. **MPI_Win distWin;**
- // 2. Allocate tempRow and distMatrix. For the distMatrix, use an appropriate MPI function.
- **MPI_Alloc_mem() nebo: MPI_Win_allocate(nCols *sizeof(int, sizeof(int), MPI_INFO_NULL, Mpi_COMM_WORLD, &distMAtrix, distWin)**
- // 3. Create a window. Every rank exposed its row.
- **MPI_Win_create()**
- **MPI_Win_fence()** // 1.parametr zde nastavíme assert = 0
- // 4. Every rank collects appropriate column from all ranks into tempRow.
- // Rank 0 collects the 1st col, rank 1 the 2nd col, etc.
- // Use MPI_Get operations.
- **for (int i = 0; i < nCols; i++)**
  - **MPI_Get();**
  - **MPI_Win_fence()** // paměť. zábrana zajišťuje konzistenci. Kolektivní operace!
    // ujistíme se, zda všichni už otevřeli okno
- // 5. Copy the row back into the window (although we could use local copy, we'll use MPI_Put).
- **MPI_Put() nebo: memcpy(distMatrix, tempRow,...)**
- **MPI_Win_fence()**
- // 6. Use a collective communication to transpose the matrix back.
- **MPI_Alltoall()** // můžeme použít MPI_IN_PLACE jako 1. parametr
- // 7. Free memory, the window, etc. **MPI_Win_free(), MPI_Free_mem()**

```
int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)
int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info,
                   MPI_Comm comm, MPI_Win *win)
int MPI_Win_fence(int assert, MPI_Win win)
int MPI_Put(const void *origin_addr, int origin_count, MPI_Datatype origin_datatype,int target_rank, MPI_Aint
target_disp, int target_count, MPI_Datatype target_datatype,MPI_Win win)
int MPI_Win_free(MPI_Win *win)
int MPI_Free_mem(void *base)
int MPI_Get(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp,
int target_count, MPI_Datatype target_datatype, MPI_Win win)
```

# 1: Transpozice matice

```
------------------------------------------------------------------------
------------------------------------------------------------------------
                            PPP Lab 5
------------------------------------------------------------------------
 Example 1 - Implement matrix transposition using Put and Get
------------------------------------------------------------------------
----------------------- Original matrix ----------------------------
 - Rank  1, row  0 = [   10000,    10001]
 - Rank  0, row  0 = [       0,        1]
---------------------- Transposed matrix ---------------------------
 - Rank  1, row  0 = [       1,    10001]
 - Rank  0, row  0 = [       0,    10000]
----------------------- Original matrix ----------------------------
 - Rank  1, row  0 = [   10000,    10001]
 - Rank  0, row  0 = [       0,        1]
------------------------------------------------------------------------
```

2 ranky

```
------------------------------------------------------------------------
 Example 1 - Implement matrix transposition using Put and Get
------------------------------------------------------------------------
----------------------- Original matrix ----------------------------
 - Rank  1, row  0 = [   10000,    10001,    10002,    10003]
 - Rank  2, row  0 = [   20000,    20001,    20002,    20003]
 - Rank  3, row  0 = [   30000,    30001,    30002,    30003]
 - Rank  0, row  0 = [       0,        1,        2,        3]
---------------------- Transposed matrix ---------------------------
 - Rank  3, row  0 = [       3,    10003,    20003,    30003]
 - Rank  1, row  0 = [       1,    10001,    20001,    30001]
 - Rank  2, row  0 = [       2,    10002,    20002,    30002]
 - Rank  0, row  0 = [       0,    10000,    20000,    30000]
----------------------- Original matrix ----------------------------
 - Rank  2, row  0 = [   20000,    20001,    20002,    20003]
 - Rank  3, row  0 = [   30000,    30001,    30002,    30003]
 - Rank  1, row  0 = [   10000,    10001,    10002,    10003]
 - Rank  0, row  0 = [       0,        1,        2,        3]
------------------------------------------------------------------------
```

4 ranky

# ÚKOL 2: TRANSPOZICE MATICE, OPTIMALIZACE OKNA A PAMĚŤOVÝCH ZÁBRAN

Příklad č. 2 je totožný s příkladem 1, budeme se ale snažit o další optimalizace. Vyzkoušíme si práci s objektem `MPI_Info` a různými asserty u paměťových zábran. Zadání se nachází pod sekcí `case 2:` ve funkci `main`:

1. Deklarujte objekt `MPI_Info` a nastavte jeho dva parametery `same_size` a `same_disp_unit` na hodnoty `true`. Pozor, klíč i hodnota se zadává jako `string`.
2. Vytvořte okno a předejte mu vlastní objekt typu `MPI_Info`.
3. Projděte jednotlivé paměťové zábrany a zamyslete se, které asserty opravdu potřebujte. Např. jedná se o poslední zábranu, v této zábraně nebyl Put, atd.
4. Přeložte soubor.
5. Spusťte výslednou binárku:

```
$ mpiexec -np 2 ./one 2
$ mpiexec -np 4 ./one 2
$ mpiexec -np 8 ./one 2
$ mpiexec -np 16 ./one 2
```

6. Porovnejte výsledky všech operací.

```
MPI_Win_fence(MPI_MODE_NOPRECEDE, distWin);

// 7. Every rank collects appropriate column from all ranks into tempRow.
//    Rank 0 collects the 1st col, rank 1 the 2nd col, etc.
//    Use MPI_Get operations {MPI_MODE_NOSTORE, MPI_MODE_NOPUT, MPI_MODE_NOPRECEDE, MPI_MODE_NOSUCCEED}.
//    You can use multiple by operator |, eg. MPI_MODE_NOSTORE | MPI_MODE_NOPUT
for (int i = 0; i < nCols; i++)
{
  MPI_Get(&tempRow[i], 1, MPI_INT, i, mpiGetCommRank(MPI_COMM_WORLD), 1, MPI_INT, distWin);
}

MPI_Win_fence(MPI_MODE_NOSTORE | MPI_MODE_NOPUT, distWin);

// 8. Copy the row back into the window (although we could use local copy, we'll use MPI_Put).
MPI_Put(tempRow.data(), nCols, MPI_INT, mpiGetCommRank(MPI_COMM_WORLD), 0, nCols, MPI_INT, distWin);

MPI_Win_fence(MPI_MODE_NOSUCCEED, distWin);
```

- **MPI_Win_fence()** je něco jako bariéra, může to být poměrně náročná operace, někdy se hodí optimalizace
- **MPI_MODE_NOPRECEDE**: žádné okno předtím nebylo otevřeno
- **MPI_MODE_NOSTORE, MPI_MODE_NOPUT**: neprovedli jsme žádnou operaci store ani put, okno bylo pouze pro čtení. Všechno jsou to bitové masky a můžeme s nimi pracovat pomocí bit. operátorů (zde OR)
- **MPI_MODE_NOSUCCEED**: už tu není žádná další operace (the fence does not start any locally issued RMA calls)

- Může se to hodit do projektu – můžete použít tohle vzorové řešení ☺

# ÚKOL 3: DISTRIBUOVANÝ HISTOGRAM POMOCÍ ATOMICKÝCH AKUMULACÍ

Cílem tohoto příkladu je vyzkoušet si atomické operace při práci s oknem `MPI_Accumulate`. Na začátku mějme pole 32M hodnot uložené na ranku 0. Tyto hodnoty rozptýlíme mezi jednotlivé procesy a začneme s výpočtem histogramu. V našem případě se ovšem bude jednat o tzv. distribuovaný histogram. Každý rank tedy drží pouze určitou část košů. Uvažujeme-li 16 jader a 128 košů, pak rank 0 má koše 0–7, rank 1 koše 8–15, atd. Aby byl výpočet histogramu výpočetně náročnější, provede se zařazení do koše ne na základě hodnoty v poli, ale její hash hodnoty (`int histogramHash(int value);`). Výpočet tedy probíhá tak, že nejprve přečteme hodnotu ze své části pole, spočteme její hash a zařadíme do správného koše na odpovídajícím ranku.
Zadání se nachází pod sekcí `case 3:` ve funkci `main`:

1. Nejprve si prostudujte sekvenční implementaci tvorby histogramu.
2. Rozptylte pole `gArray` mezi jednotlivé ranky a uložte zde části do pole `lArray`.
3. Deklarujte MPI okno a volitelně objekt Info.
4. Alokujte nové okno. Tentokrát použijte funkci `MPI_Win_allocate`, kde každý rank vystaví svou část histogramu. Dejte pozor na správné předání pointeru na lokální data `distHist`.
5. Nyní vynulujte svou část histogramu.
6. Následně projděte svojí část pole a pomocí funkce akumulace modifikujte hodnotu správné koše. Dělejte tak pro každou hodnotu zvlášť.
7. Na závěr budeme chtít histogram vypsat na `stdout`. Aby jsme se ve výstupu vyznali, sesbíráme všechny koše histogramu do ranku 0 pomocí kolektivní komunikace.
8. Uvolněte všechny MPI objekty a použitá pole.
9. **Nezapomeňte použít paměťové zábrany na důležitých místech**.
10. Přeložte soubor.
11. Spusťte výslednou binárku:

```
$ mpiexec -np 2 ./one 3
$ mpiexec -np 4 ./one 3
$ mpiexec -np 8 ./one 3
$ mpiexec -np 16 ./one 3
```

12. Porovnejte výsledky všech operací.

- Máme pole nad kterým počítáme histogram **gArray**
- Vytvoříme z toho **lArray** a pomocí **MPI_Scatter()** se hodnoty pole rozptýlí po rancích
- **gHist** obsahuje 128 košů
- **lHist** – distribuovaný:
  - Rank 1: 16 košů
  - …
  - Rank 8: 16 košů
- Cílem je vzít **jednu hodnotu z lokálního pole**, zjistit do kterého koše hodnota padne a zjistit, do kterého koše máme připočítat 1
- Histogram – máme tu hash funkci aby operace trvalo trošku déle
- **koš = histogramHash(lArray[i])**
- Pokud máme např. koš = 72, tak musíme zjistit:
  - Na který rank hodnota patří
  - A v rámci ranku – do které pozice hodnota patří

# 3: Distribuovaný histogram pomocí atomických akumulací

- // 1. Distributed the global array (gArray) into lArray. Every rank has lArraySize elements.
- **MPI_Scatter()**
- // 2. Declare MPI window for the histogram an MPI_Info object.
- **MPI_Win  histWin{MPI_WIN_NULL};**
- **MPI_Info winInfo{MPI_INFO_NULL};**
- // 3.  Create an info set same_size and same_disp_unit to true.
- **MPI_Info_create()**
- **MPI_Info_set()**  // Toto není až tak podstatné, říkal šéf :-D ☺
- // 4. Allocate the window. Use the MPI_Win_allocate routine to create the window and allocate memory for it).
  - V okně máme vystaven globální histogram. Každý rank si naalokuje svou část histogramu.
  - Vynulujeme svoji část histogramu.
  - **MPI_Win_allocate()**
  - **MPI_Info_free()**
- // Clear my part of the histogram, this can be done in local memory without MPI routines.
- // This fence is necessary to ensure the array has been created at the target an initialized.
- **MPI_Win_fence()**  // lze použít MPI_MODE_NOPUT

- // 5. Use MPI_Accumulate routine to increment the appropriate bin. Hist[hash(lArray[i])]++
- **for_each(  lArray.begin(), lArray.end(), [&](int value) {**  // lambda funkce
  - **constexpr int one{1};**
  - **const int  key = histogramHash(value);**  // klíč zjistíme pomocí hash funkce
  - **MPI_Accumulate();  });** // ke kterému ranku to patří:  key / lHistSize, key % lHistSize  (operace = MPI_SUM)
  - **MPI_Win_fence()** // lze použít MPI_MODE_NOSTORE
- // Print histogram
- // 6. Collect all data from all ranks into a single histogram on the root.
- **MPI_Gather()**
- // 7. Free all used variables.  **MPI_Win_free()**

# 3: Distribuovaný histogram pomocí atomických akumulací

```
--------------------------------------------------------------------
 Example 3 - Implement distributed histogram using MPI_Accumulate.
--------------------------------------------------------------------
 Generating data ...
--------------------------------------------------------------------
Seq histogram:
bin =   0: [261793, 523472, 261360,      0,      0, 262013, 524676,      0, 262764, 524939, 262108, 262525, 261485, 262458,      0, 263667]
bin =  16: [524483, 261894, 262097, 524768,      0, 262347, 523607, 261793,      0,      0, 261537, 523862,      0, 523943, 262220, 262520]
bin =  32: [262404, 262577, 262311,      0, 524824,      0, 262582, 262293, 261751, 524239, 261788, 261856, 262381, 261233, 261357,      0]
bin =  48: [524232, 260989, 262964,      0, 785587,      0, 262208,      0, 525754,      0, 263106, 262239, 524191,      0, 524103,      0]
bin =  64: [262815, 261956, 524725,      0,      0, 524485, 260884, 261963,      0, 525149,      0, 524218, 262428, 262439,      0, 521629]
bin =  80: [     0, 262678,      0, 787464,      0, 261371, 261969, 524349,      0, 261087, 262174, 261393, 261686, 262264, 523633, 262395]
bin =  96: [261486, 262805,      0, 524896,      0, 263247, 261772, 261903, 262144, 263349, 522900,      0, 525624, 262128,      0,      0]
bin = 112: [261602, 523924, 262466,      0, 524849, 262846, 262446, 523225, 262570,      0, 262859, 262654, 262577, 261220, 524539, 523947]
--------------------------------------------------------------------
Dist histogram:
bin =   0: [261793, 523472, 261360,      0,      0, 262013, 524676,      0, 262764, 524939, 262108, 262525, 261485, 262458,      0, 263667]
bin =  16: [524483, 261894, 262097, 524768,      0, 262347, 523607, 261793,      0,      0, 261537, 523862,      0, 523943, 262220, 262520]
bin =  32: [262404, 262577, 262311,      0, 524824,      0, 262582, 262293, 261751, 524239, 261788, 261856, 262381, 261233, 261357,      0]
bin =  48: [524232, 260989, 262964,      0, 785587,      0, 262208,      0, 525754,      0, 263106, 262239, 524191,      0, 524103,      0]
bin =  64: [262815, 261956, 524725,      0,      0, 524485, 260884, 261963,      0, 525149,      0, 524218, 262428, 262439,      0, 521629]
bin =  80: [     0, 262678,      0, 787464,      0, 261371, 261969, 524349,      0, 261087, 262174, 261393, 261686, 262264, 523633, 262395]
bin =  96: [261486, 262805,      0, 524896,      0, 263247, 261772, 261903, 262144, 263349, 522900,      0, 525624, 262128,      0,      0]
bin = 112: [261602, 523924, 262466,      0, 524849, 262846, 262446, 523225, 262570,      0, 262859, 262654, 262577, 261220, 524539, 523947]
+------------------------------------------------------------------+
| Seq time:  16.67s                                                |
| Par time:   7.25s (P =  4)                                       |
| Speedup:    2.30x (57.48%)                                       |
+------------------------------------------------------------------+
```

- Pokud komunikační vzor není jasný – jsou výborné jednostranné MPI komunikace, v tomto případě by bylo šílené používat MPI_Send() a MPI_Recv()

Za domácí úkol

# ÚKOL 4: DISTRIBUOVANÝ HISTOGRAM S LOKÁLNÍM SCRATCH

Tento příklad je principiálně totéž co předchozí. Snahou je ovšem eliminovat nutné zápisy do sdílené paměti (okna). Proto si tedy každý rank vytvoří celou kopii histogramu, který zpracuje. Jakmile je výpočet dokončen, každý rank aktualizuje odpovídající koše v distribuovaném histogramu. Tedy, aktualizuji všechny koše na ranku 0, ranku 1, atd. Všechny koše na daném ranku aktualizuji jedním voláním!
Zadání se nachází pod sekcí `case 4:` ve funkci `main`:

1. Vycházejte z předchozího příkladu.
2. Alokujte si dočasný histogram `tmpHist` s velikostí celého histogramu `tmpHistSize`.
3. Do tohoto histogramu proveďte výpočet.
4. Aktualizujte distribuovaný histogram, nezapomeňte ho předtím vynulovat.
5. Uvolněte všechny MPI objekty a použitá pole.
6. **Nezapomeňte použít paměťové zábrany na důležitých místech, které vhodně omezíte asserty.**
7. Přeložte soubor.
8. Spusťte výslednou binárku:

```
$ mpiexec -np 2 ./one 4
$ mpiexec -np 4 ./one 4
$ mpiexec -np 8 ./one 4
$ mpiexec -np 16 ./one 4
```

9. Porovnejte výsledky všech operací.

# ÚKOL 5: DISTRIBUOVANÉ NÁSOBENÍ MATIC

# 4: Distribuovaná histogram s lokálním scratch

- Máme distribuované matice A, B, C
- Každý rank má část řádků A, B, C
- Distribuce je tedy po řádcích
- Když počítáme část C, tak pro jeden prvek potřebujeme
  - 1 řádek z A
  - 1 sloupec z B
- Řádky A a C máme lokálně, jenže B máme uloženo  po řádcích, ale přistupujeme po sloupcích
- Musíme tedy provést nějaké mapování – Bčka musíme získat přes více ranků
- Takže jediné okno bude pro matici B
- **Násobení matic**
  - **for j**
    - // na začátku smyčky si natáhneme Bčka.
      **for** (int rank = 0; rank < mpiGetCommSize(MPI_COMM_WORLD); rank++)
      - MPI_Get()
    - MPI_Win_fence() - musíme použít fences, protože jinak nezajistíme natažení řádku
    - // maticové násobení: localC[i * nCols + j] += localA[i * nCols + k] + oneCol[k];
    - **for i**
      - **for k**
  - MPI_Win_fence()

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

Děkuji vám za pozornost!