Praktické paralelní programování (PPP 2025)
Počítačové cvičení č. 5: Jednostranné komunikace v MPI

Jiří Jaroš (jarosjir@fit.vutbr.cz)

1 ÚVOD

Cílem dnešního cvičení bude vyzkoušet si práci s jednostrannými komunikacemi v MPI. Nejprve se zaměříme na triviální příklad transpozice matice, kdy každý proces vystaví svůj řádek a následně si sesbírá požadovaný sloupec od ostatních procesů. Ve druhém příkladě si vyzkoušíme práci s atomickými operacemi při tvorbě distribuovaného histogramu. Poslední příklad nám ukáže jednu variantu distribuovaného násobení matic.

2 PŘIHLÁŠENÍ NA BARBORU/KAROLINU A ALOKACE VÝPOČETNÍHO UZLU

Pokud používáte cluster Karolina:

1. Zažádejte o jeden uzel v interaktivním módu:

```
$ salloc -A DD-24-108 -p qcpu_exp -N 1 --ntasks-per-node 128 -t 01:00:00
```

2. Natáhněte modul s OpenMPI:

```
$ ml GCC/12.2.0 OpenMPI/4.1.4-GCC-12.2.0 CMake/3.24.3-GCCcore-12.2.0
```

Pokud používáte cluster Barbora:

1. Zažádejte o jeden uzel v interaktivním módu:

   ```
   $ salloc -A DD-24-108 -p qcpu_exp -N 1 --ntasks-per-node 36 -t 01:00:00
   ```

2. Natáhněte modul s OpenMPI:

   ```
   $ ml GCC/12.2.0 OpenMPI/4.1.4-GCC-12.2.0 CMake/3.24.3-GCCcore-12.2.0
   ```

2.1 Překlad

Vygenerujte překladový skript pomocí `cmake` a spusťte překlad:

```
$ cmake -Bbuild -S.
$ cmake --build build
```

3 Příklad 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.

Otázky k zamyšlení/zodpovězení:

1. Proč nemohu použít něco jako operaci swap?
2. Proč je nutné používat paměťové zábrany?
3. Která varianta je dle vás efektivnější Put + Get nebo kolektivní komunikace?

4 Příklad 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.

5 Příklad 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.

Otázky k zamyšlení/zodpovězení:

1. Sestrojte graf silného škálování (závislost času na počtu jader, obě osy $\log_2$ měřítko), zrychlení a efektivity. Testujte na 2, 4, 8, 16, 32 a 64 jádrech a využijte i větší počet uzlů.

2. Proč dosahujete tak nízké efektivity?

3. Proč efektivita prudce padá jakmile zapojíte více než jeden uzel?

6 Příklad 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.

Otázky k zamyšlení/zodpovězení:

1. Sestrojte graf silného škálování (závislost času na počtu jader, obě osy $\log_2$ měřítko), zrychlení a efektivity. Testujte na 2, 4, 8, 16, 32 a 64 jádrech a využijte i větší počet uzlů.

2. Srovnejte efektivitu s předchozím příkladem?

7 PŘÍKLAD 5. - DISTRIBUOVANÉ NÁSOBENÍ MATIC

Uvažujme situaci, kdy máme tři čtvercové matice a chceme spočítat maticový součin $C = A \times B$. Velikost matic je $nRows \times nCols$. Matice jsou distribuované po blocích řádků $localRows \times nCols$.
Pro každý prvek matice C tedy musíme provést skalární součin řádku matice A se sloupcem matice B. Z matic A a C máme všechny potřebné elementy, pro matici B ale budeme muset stahovat z ostatních ranků (vždy sloupec, který potřebujeme).
Zadání se nachází pod sekcí `case 5:` ve funkci `main`:

1. Nejprve si vytvoříme okno a alokujeme paměť pro lokální část matice B (`localB`).
2. Následně alokujeme paměť pro `localA` a `localC`.
3. Nyní vytvoříme řádkový datový typ, který bude sloužit pro rozptýlení původních matic.
4. Následně vytvoříme sloupcový datový typ, který bude popisovat jeden sloupec `localB`.
5. Rozptýlíme pole `globalA` a `localB` a vynulujeme `localC`. Použijte neblokující kolektivní komunikaci a překryjte čas komunikace s nulováním pole `localC`.
6. Nyní provedeme samotné násobení matic. Aby byl výpočet efektivnější řadíme smyčky jako `(j, i, k)`. Každý sloupec nejdříve sesbíráme od ostatních ranků, k čemuž využijeme vhodný datový typ. Následně provedeme násobení s lokálními daty a uložení do výsledku.
7. **Nezapomeňte použít paměťové zábrany na důležitých místech, které vhodně omezíte asserty**.
8. Na závěr matici `localC` sesbíráme zpět do `globalC`, abychom mohli porovnat výsledky.
9. Přeložte soubor.
10. Spusťte výslednou binárku:

```
$ mpiexec -np 2 ./one 5
$ mpiexec -np 4 ./one 5
$ mpiexec -np 8 ./one 5
$ mpiexec -np 16 ./one 5
```

11. Porovnejte výsledky všech operací.

Otázky k zamyšlení/zodpovězení:

1. Sestrojte graf silného škálování (závislost času na počtu jader, obě osy $\log_2$ měřítko), zrychlení a efektivity. Testujte na 2, 4, 8, 16, 32 a 64 jádrech a využijte i větší počet uzlů.
2. Jakmile použijeme více uzlů, efektivita jde razantně dolů. Zamyslete se, jak by šlo udělat násobení lépe (s menším množstvím komunikace, při zachování distribuce).