Dokumentace projektu do předmětu Paralelní a distribuované systémy
Implementace algoritmu Bucket sort

Tomáš Aubrecht

1. Úvod

Cílem tohoto projektu bylo implementovat řadící algoritmus Bucket sort v jazyce C++ s využitím knihovny Open MPI. Bylo potřeba také vytvořit řídící skript, který vypočítá počet procesorů na základě vstupního parametru počtu hodnot v řazené posloupnosti a nad touto posloupností spustí implementovaný algoritmus.

2. Rozbor a analýzu algoritmu

Algoritmus Bucket sort je paralelní řadící algoritmus, který pracuje na stromové architektuře. Při řazení postupuje tak, že v prvním kroce rovnoměrně rozdělí hodnoty vstupní posloupnosti mezi listové procesory. Ty je v dalším kroce seřadí optimálním sekvenčním řadícím algoritmem. Následující krok představuje spojení dvou seřazených posloupností od svých synů nelistovými procesory. Toto spojování probíhá iterativně od největší úrovně nelistových uzlů ($log_2(log_2(n)) - 1$) směrem ke kořenu. V posledním kroce uloží kořenový procesor výslednou seřazenou posloupnost do paměti.

Asymptotická časová složitost tohoto algoritmu je $O(n)$, kde n je počet řazených prvků. Časová složitost vychází ze složitosti jednotlivých kroků. Načtení a rozdělení řazených prvků mezi procesory v prvním kroce probíhá v lineárním čase. Každý listový procesor čte $n/log_2(n)$ prvků, kde při použití optimálního sekvenčního řadícího algoritmu se složitostí $O(x \cdot log_2(x))$ dostaneme složitost $O((n/log_2(n)) \cdot log_2(n/log_2(n)))$, která odpovídá lineární časové složitosti $O(n)$. Spojování dvou posloupností každým procesorem na úrovni i, které mají každá délku $n/2^i$, trvá při použití například straight merge sort $k \cdot n/2^i$ kroků, kde k je konstantní. Spojování lze tedy provést v lineárním čase. Pro seřazení n prvků je potřeba $2 \cdot log(n) - 1$ procesorů. Výsledná cena algoritmu Bucket sort je tedy $O(n \cdot log(n))$, což je optimální.

3. Implementace

Bucket sort je implementovaný v jazyce C++ za použití knihovny OpenMPI pro paralelní výpočty, kde tento algoritmus pracuje v úplném binárním stromě procesorů. Na začátku programu je nutné volat funkci MPI_Init(), která nainicializuje paralelní prostředí. Dále je třeba si uložit celkový počet procesorů vykonávající daný program a ID konkrétního procesoru. Každý procesor má tři vektory integerů ze standartní knihovny C++ *vector*. Dva vektory jsou určeny pro posloupnosti čísel od svých synů a jeden vektor představuje spojení těchto dvou posloupností. Listové procesory využívají pouze jeden z těchto vektorů.

Pro reprezentaci stromové struktury byla naimplementovaná třída *Node*, která na základě ID procesoru vypočítá ID svých dětí a rodiče. Nechť x je číslo aktuálního procesoru. Pokud je x rovno nule, pak je daný procesor kořenový. Jinak je číslo rodiče v uplném binárním stromě $(x - 2)/2$ pokud je x sudé, $(x - 1)/2$, pokud je x liché. Číslo levého dítěte je $2x + 1$ a číslo pravého

dítěte je *2x + 2*. Pokud je číslo dětí větší než je celkový počet procesorů, pak daný procesor nemá děti, protože je listový.

Zpracování začíná načtením řazených hodnot ze vstupního souboru kořenovým procesorem. Ten je neseřazené vytiskne na konzoli a vypočítá celkový počet listových procesorů, mezi které tyto hodnoty rozdělí. Nechť *p* je celkový počet procesorů. Potom počet listových procesorů v úplném binárním stromě je roven *(p + 1)/2* zaokrouhleno nahoru. Při počnu *N* vstupních hodnot pak každý listový procesor řadí *N/p* hodnot zaokrouhleno nahoru. Kořenový procesor tedy postupně rozešle všem listovým procesorům *N/p* hodnot pomocí metody *sendVector()* třídy *Node*. Tato funkce představuje odeslání vektoru hodnot jinému procesoru, kde pomocí funkce *MPI_Send()* se nejprve odešle počet hodnot ve vektoru a následně se postupně odesílají jednotlivé hodnoty. Pro rozlišení počtu hodnot a hodnoty ve vektoru jsou definovány dva tagy: *TAG_COUNT* a *TAG_VALUE*. Listové procesory příjmou své posloupnosti pomocí obdobné metody *recvVector().* V této metodě nejprve příjmou pomocí funkce *MPI_Recv()* s tagem *TAG_COUNT* počet hodnot, které budou přijímat a následně řadit, a poté v cyklu pomocí stejné funkce, ale s tagem *TAG_VALUE*, příjme daný počet hodnot, které ukládá do vektoru.

Samotný proces řazení začíná seřazením vektoru hodnot ve všech listových procesorech pomocí sekvenčního řadícího algoritmu. Pro tento účel byla použita funkce *sort(),* která je součástí standartní knihovny C++. Po seřazení odešlou listové procesory posloupnost svým rodičům, kteří na ni čekají. K tomu jsou použity již výše zmíněné funkce *sendVector()* a *recvVector()*. Nelistové procesory po přijetí spojí dané posloupnosti pomocí funkce *merge(),* která je součastí knihovny *algorithm* a odešlou je opět svým rodičům, dokud se postupně spojované posloupnosti nedostanou až ke kořenu. Ten provede poslední spojení posloupností z levého a pravého podstromu a výslednou seřazenou posloupnost vytiskne na konzoli po jedné hodnotě na řádek. Na konci programu je volána funkce *MPI_Finalize()*, která uvolní a ukončí paralelní prostředí.

4. Experimenty

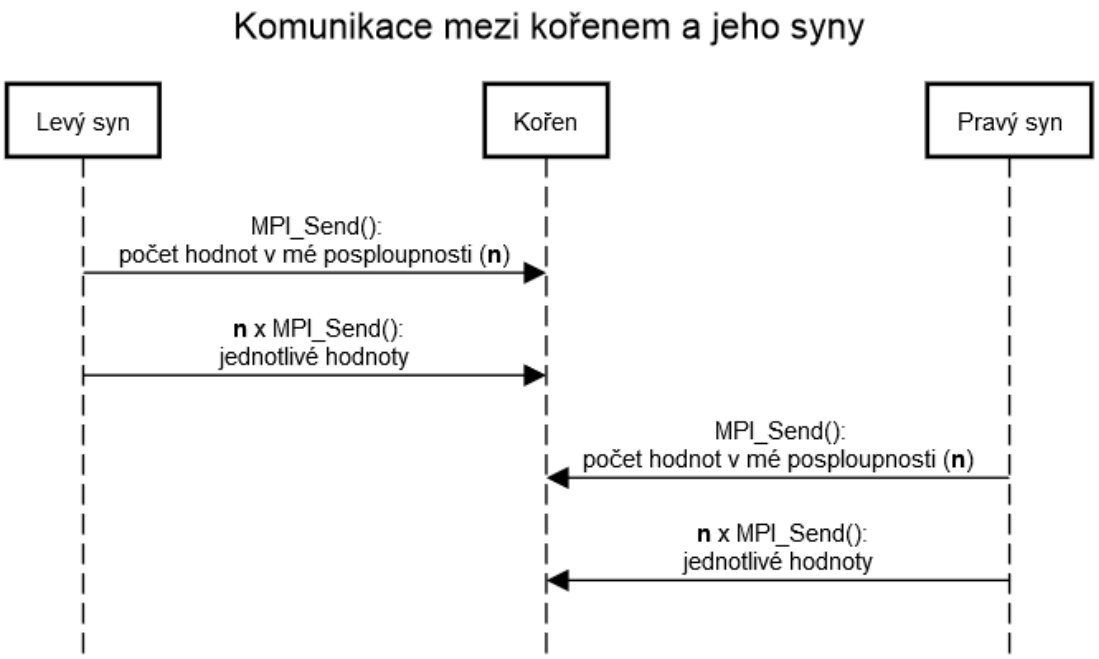
Na naimplementovaném algoritmu bylo provedeno několik experimentů s různě velkými počty vstupních hodnot pro ověření časové složitosti Bucket sortu. Knihovna *Open MPI* neobsahuje žádné implicitní metody pro měření složitosti algoritmů, proto byl naimplementován vlastní způsob měření času pomocí knihovny *time.h*.

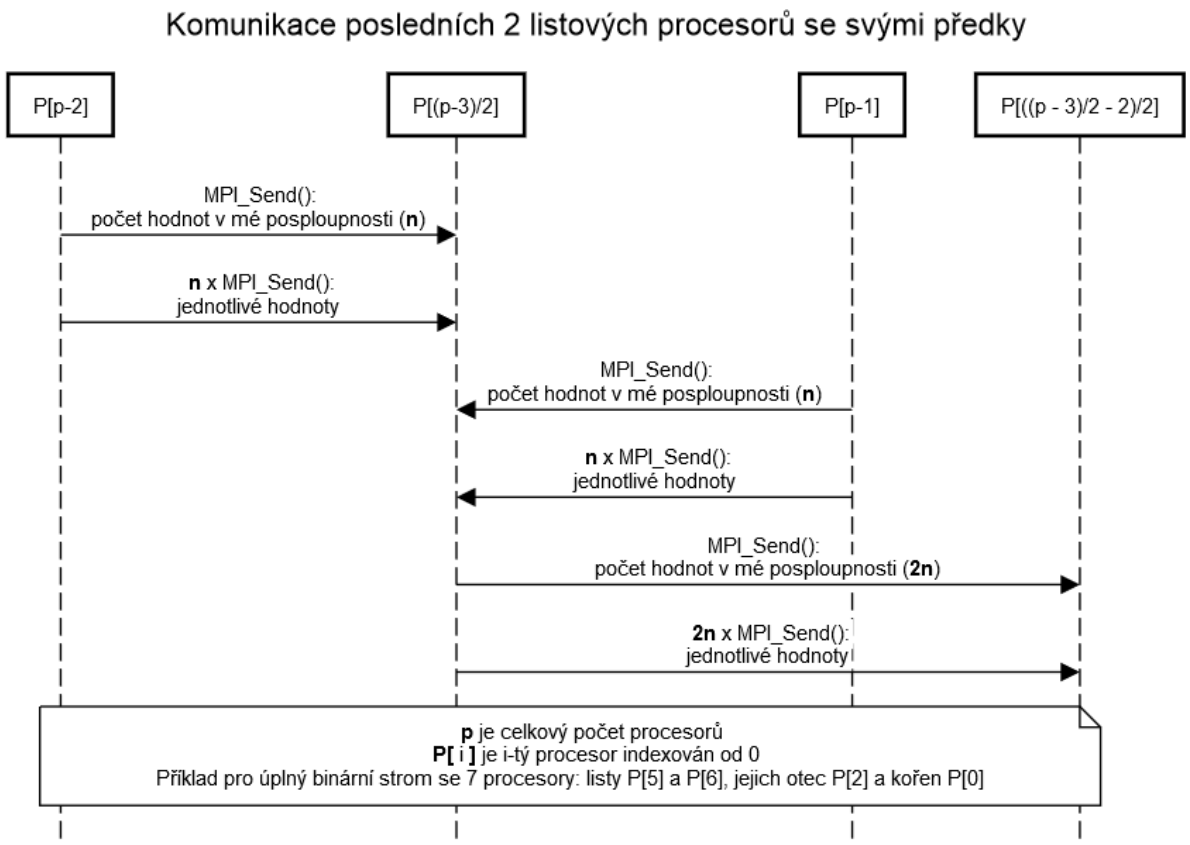
Algoritmus byl testován na několika sadách vstupů o velikosti 8, 16, 24, 32, 40, 48, 56 a 64 prvků, kde každý počet prvků měl 10 různých posloupností a každá z nich byla testována 12 krát. Minimální a maximální hodnoty na každé posloupnosti byly zahozeny a zbylých 10 se zprůměrovalo. Dohromady tedy bylo provedeno 960 testů.

Pro účely ověření časové složitosti algoritmu není do měření započtena inicializace MPI prostředí, načtení hodnot ze vstupního souboru a veškeré výpisy do konzole. Pro spuštění programu za účelem měření stačí předefinovat macro *MEASURE_PERFORMANCE* ve zdrojovém kódu na hodnotu *true*. Jednotlivé naměřené hodnoty můžete vidět v grafu uvedeném níže.

5. Komunikační protokol

Na dvou sekvenčních diagramech uvedených níže můžete vidět komunikaci mezi kořenovým procesorem a jeho syny a následně komunikaci mezi dvěma posledními listovými procesory, jejich otcem a otcem jejich otce. Procesory mezi sebou komunikují pomocí funkcí *MPI_Send()* a *MPI_Recv()*.





6. Závěr

Po provedení experimentů bylo zjištěno, že teoretická časová složitost algoritmu Bucket sort popsána v kapitole 2 odpovídá grafu naměřených hodnot, na kterém lze vidět, že s rostoucím počtem řazených hodnot lineárně roste i celkový čas řazení.

Testování bylo provedeno na referenčním serveru merlin s operačním systémem CentOS pomocí výše popsané testovací sady. Veškeré testy skončili úspěchem.