# Implementace algoritmu "Odd-even transposition sort"

Jakub Zárybnický

25. března 2020

## 1 Algoritmus

Algoritmus pracuje s lineárně propojenou sadou procesorů, pro uspořádání N hodnot použijeme N procesorů, kde hodnoty jsou distribuované v přípravném kroku algoritmu.

Pokud se v jednom kroku přesune jedna hodnota nejvýše o jeden procesor doleva či doprava, potřebujeme právě N kroků pro seřazení opačně uspořádané sekvence - tj. pro přesunutí nejmenší hodnoty např. z pravého konce doleva. Pokud nazveme jednotlivé kroky algoritmu "distribute", "sort" a "collect", pak

1. "distribute" je v odevzdané implementaci $O(n)$, odesíláme hodnoty po jedné. Při využití sdílené paměti by ale krok "distribute" mohl mít $O(1)$, kdy každý procesor by si načetl svou hodnotu z paměti v jednom kroku;

2. "sort" je jádro algoritmu. Za použití lineární topologie a $N$ procesorů pro $N$ hodnot je nutný počet kroků lineární, $O(N)$, za předpokladu opačně seřazené sekvence;

3. "collect" má stejně jako "distribute" časovou složitost $O(n)$, nejlevější procesor přijímá hodnoty po jedné. Za použití sdílené paměti by ale složitost mohla být $O(1)$.

Celková časová složitost je tedy lineární, $t(n) = O(n)$. Celková prostorová složitost je také lineární, n procesorů (s jedním paměťovým slotem), $p(n) = O(n) = O(n)$. Celková cena algoritmu je ale $c(n) = t(n) * p(n) = O(n^2)$, kdežto optimální složitost pro algoritmus paralelního řazení je $O(logn)$.

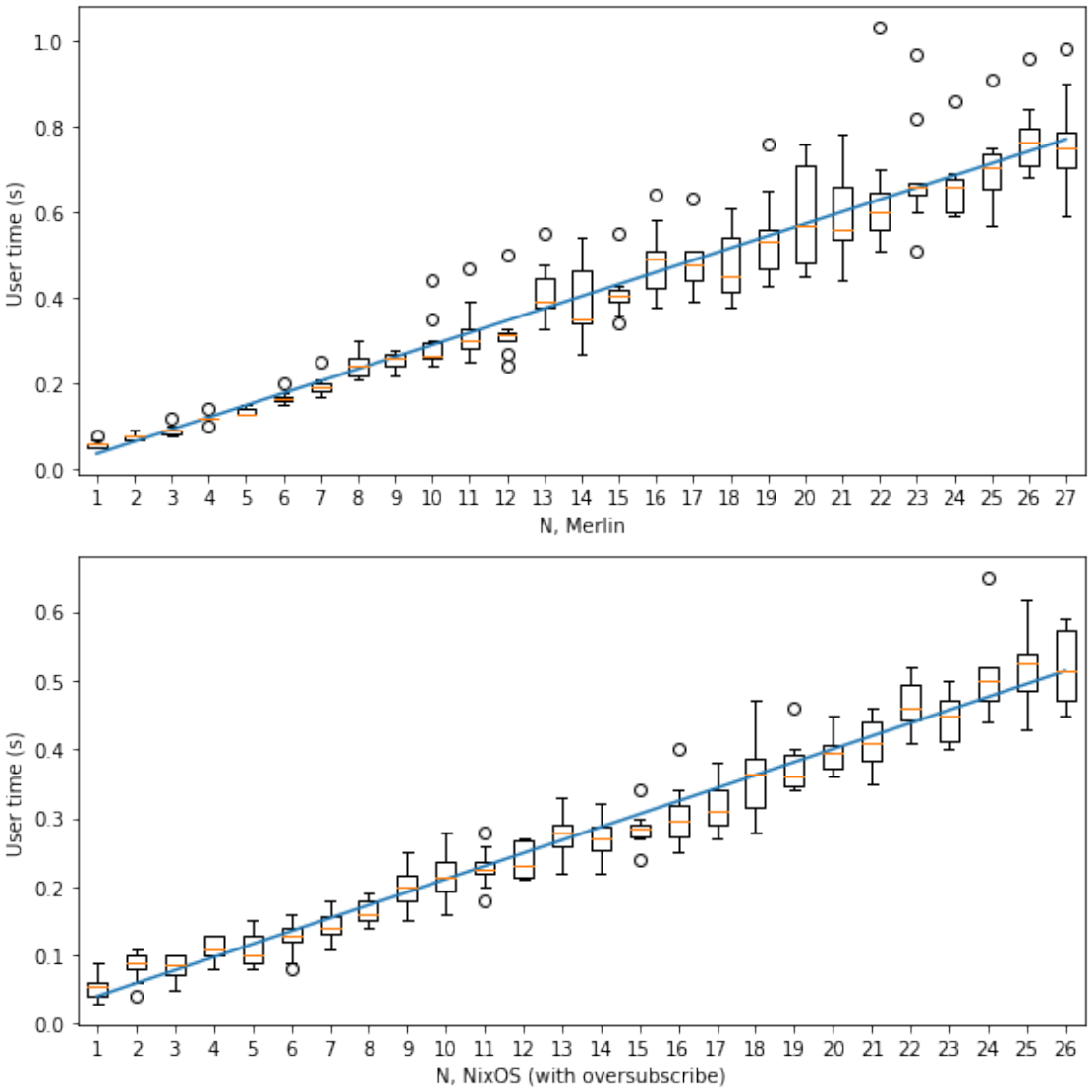
## 2 Implementace

Implementovaný program také pracuje ve třech krocích, ve fázi "distribute" se načtou data ze souboru v procesu s rankem 0 a pomocí primitivní operace `MPI_Scatter` se jednotlivé položky dostanou i k ostatním procesům.

Fáze "sort" probíhá různě pro sudé/liché procesy, kdy v sudých cyklech dostanou sudé procesy hodnotu svého levého souseda a rozhodnou se, jestli transponovat nebo ne (kombinace operací `MPI_Send` a `MPI_Recv`), v lichých cyklech toto provádějí zase procesy liché.

V závěrečné fázi "collect" se pro sesbírání dat od jednotlivých procesů použije `MPI_Gather`, a proces s rankem 0 vypíše už uspořádaná data.

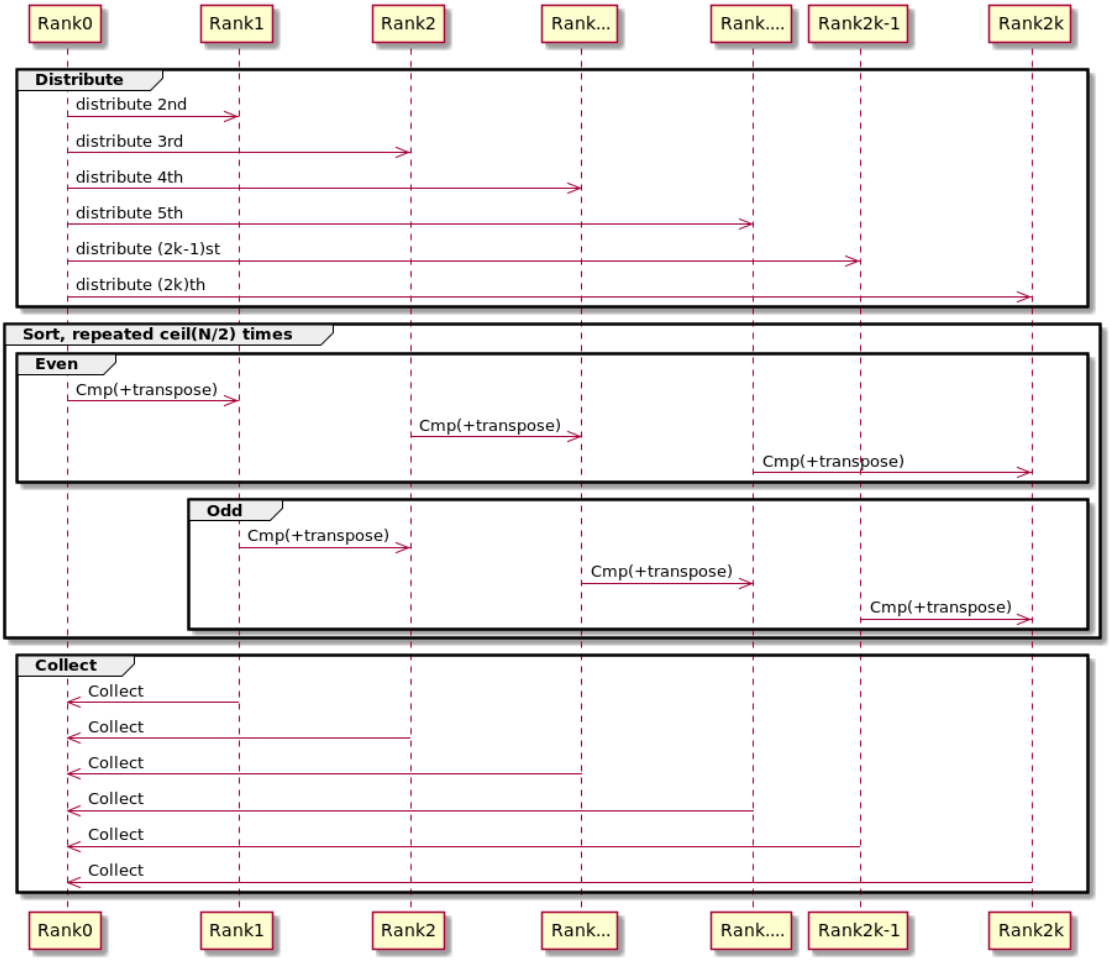
# 3 Experimenty




Experimenty byly provedeny pomocí shell skriptu, ve kterém bylo pro N z intervalu od 1 do 26, resp. 27 pro Merlina, provedeno deset měření pomocí příkazu `time` (`/usr/bin/time`, ne vestavěný příkaz terminálu). Pro každé měření byla vygenerována nová sada N vstupních hodnot spuštěn program pomocí `mpirun`, měřena byla celková doba běhu příkazu `mpirun`. Výstupem tohoto skriptu byl CSV soubor s jednotlivými naměřenými hodnotami (parametr N, `wall time`, `user time`, `kernel time`).

Měření jsem provedl na Merlinovi a na svém osobním počítači (za použití `--oversubscribe`), pro vytvoření grafu jsem použil sloupec `user time` a knihovnu `matplotlib`. Z těchto hodnot je zakreslen výše přiložený boxplot (za použití výchozího nastavení), včetně regresní přímky proložené získanými hodnotami.

# 4 Protokol




# 5 Závěr

Z výsledného grafu experimentu je zřejmé, že lineární časová náročnost implementovaného algoritmu odpovídá teoretické časové složitosti. Počet hodnot *outliers* naměřených na Merlinovi je sice nečekaně mnoho, předpokládám ale, že to bylo způsobené procesy jiných uživatelů, při dalších měřeních se stejný problém už neopakoval.

Absolutní časové hodnoty, např. 0.8s pro seřazení 27 čísel na merlinovi, jsou sice pro program s touto funkčností nepřijatelné - nejen algoritmus není optimální svou celkovou cenou jakožto algoritmus pro paralelní řazení, ale implementační volba OpenMPI způsobuje, že čas potřebný pro režii komunikace mezi procesy zcela dominuje času spotřebovaný pro řazení samotné.