1 Úvod

Úlohou tohto projektu bolo implementovať v jazyku `C/C++` pomocou knižnice `Open MPI` algoritmus *Line-of-Sight*. Okrem samotnej implemntácie algoritmu bolo nutné vytvoriť `shell` skript, ktorý vypočíta počet procesorov pre zadaný počet prvkov.

2 Viditeľnosť

2.a Popis algoritmu

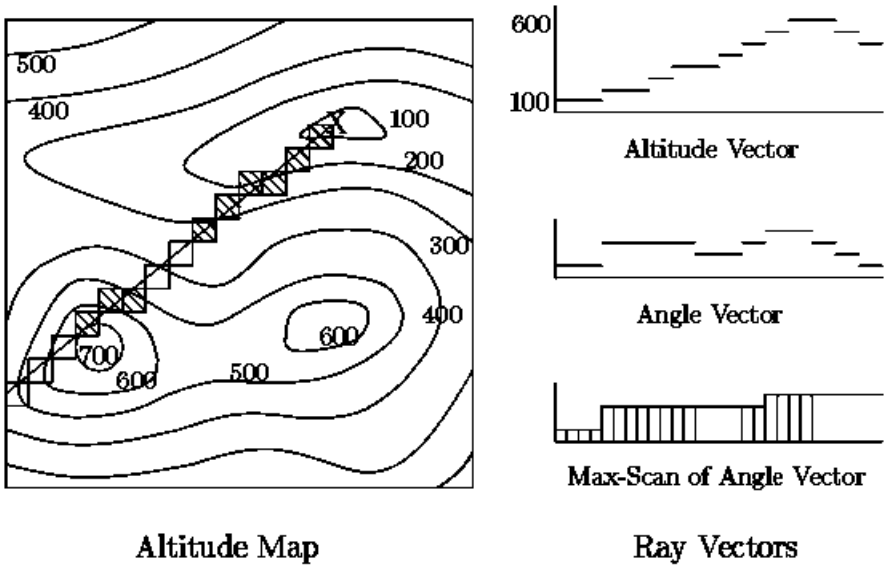
Tento algoritmus je typickým príkladom aplikácie *sumy prefixov*. Jeho vstupom je obvykle matica terénu vo forme matice nadmorských výšok a pozorovací bod `X`. Úlohou algoritmu je potom zistiť, ktoré body pozdĺž paprsku vychádzajúceho z miesta `X` sú viditeľné (viz Obr. 1 [2]).

Bod pozdĺž paprsku je viditeľný v prípade, že žiadny bod medzi ním a pozorovateľom nemá väčší vertikálny uhol. Postup algoritmu:

1. Vytvorí sa vektor výšok bodov pozdĺž pozorovacieho paprsku.
2. Vektor výšok je podľa vzťahu 1 prevedený na vektor vertikálnych uhlov.

$$angle_i = arctan\left(\frac{altitude_i - altitude_{\texttt{X}}}{i}\right) \tag{1}$$

3. Aplikujeme operáciu *prescan* s operátorom `maximum` na získaný vektor uhlov. Táto operácia vráti pre každý bod hodnotu doposiaľ najväčšieho uhla medzi ním a pozorovateľom.
4. Následne musí každý bod porovnať svoj uhol s výsledkom *prescan* a v prípade, že je jeho hodnota väčšia, bod je **viditeľný**.



Obr. 1: *Line-of-sight* algoritmus pre jeden paprsok. `X` označuje miesto pozorovateľa. Bod na paprsku je viditeľný ak žiadny z bodov pred ním nemá väčší uhol (šrafované štvorčeky predstavujú viditeľné body) [2].

2.b Prescan

V tejto sekcii si aspoň v krátkosti priblížime operáciu *prescan*, keďže tvorí hlavnú časť algoritmu Line-of-sight, a budeme z nej vychádzať pri jeho analýze.

Procedúra prescan má ako vstup binárny asociatívny operátor $\oplus$ s neutrálnym prvok I, vektor s n prvkami

$$[a_0, \ldots, a_{n-1}]$$

a výsledkom tejto operácie je vektor:

$$[I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus a_{n-2})]$$

Za predpokladu, že operátor $\oplus$ je asociatívny, môžme na výpočet operácie prescan použiť binárny strom, ktorý možno preložiť vstupným vektorom. Samotný algoritmus potom pozostáva z dvoch fáz a to konkrétne:

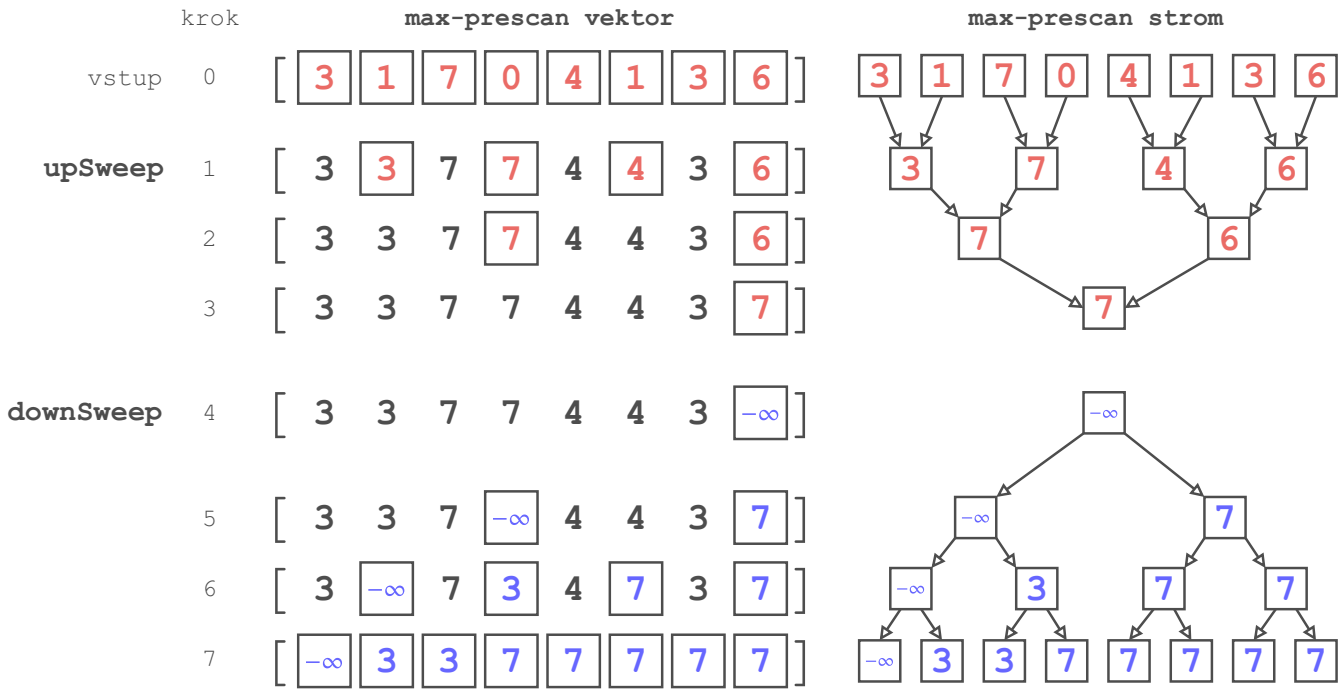
1. `upSweep(...)`

   Konkrétne fáza `upSweep` v každom vrchole stromu aplikuje operátor $\oplus$ na príslušnú dvojicu prvkov. Správnosť výsledku je zabezpečená už skôr spomínanou asociativitou operátoru $\oplus$. Takýmto spôsobom postupujeme až kým nedosiahneme koreň stromu.

2. `downSweep(...)`

   Potom na rad prichádza fáza `downSweep`, v ktorej na začiatku nahradíme hodnotu v koreni stromu neutrálnym prvkom a tentokrát postupujeme od koreňa stromu smerom k listom. Každý vrchol `V` (s hodnotou `V.value`) dá svojmu ľavému synovi svoju hodnotu a pravému synovi hodnotu `V.value` $\oplus$ `V.left` (viz Obr. 2).

Po dokončení týchto dvoch fáz dostaneme $[I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus a_{n-2})]$ (Dôkaz viz. Proof 1.1 [2]). V prípade, že operáciu prescan implementujeme na architektúre EREW PRAM, potom každá úroveň stromu **môže byť vykonaná paralelne**.



Obr. 2: Operácia prescan s asociatívnym operátorom `max`, a neutrálnym prvkom $-\infty$.

2.c Analýza algoritmu

1. *Počet procesorov:*

   Ako vidieť z obrázku 2 pre každú dvojicu prvkov je potrebný jeden procesor a teda: $\mathbf{p(n)} = n/2$.

2. *Časová zložitosť:*

   Z popisu algoritmu v sekcii 2.a si ako výpočet vertikálnych uhlov tak aj záverečné porovnanie (body 2 a 4) vyžadujú konštantný čas, keďže tieto akcie sú vykonané paralelne. Zložitosť týchto krokov je teda konštantná, $\mathcal{O}(1)$.

   Z toho vyplýva, že časová zložitosť algoritmu je daná zložitosťou bodu 3 a teda zložitosťou operácie prescan. Keďže prescan je implementovaný pomocou stromu (viz Obr. 2), a ako sme už spomínali každý krok môže bežať paralelne, potom čas potrebný na vykonanie je $2\lceil log_2(n) \rceil$.

   Celková časová zložitosť nášho algoritmu je teda: $\mathbf{t(n)} = \mathcal{O}(1) + \mathcal{O}\left(log(n)\right) = \mathcal{O}\left(log(n)\right)$.

3. *Celková cena:*

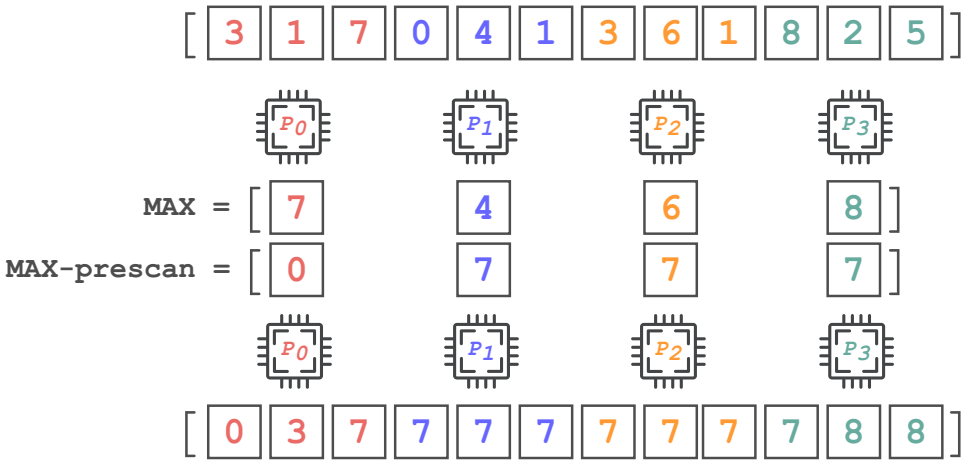
   V našom prípade je cena paralelného riešenia algoritmu Line-of-sight $\mathbf{c(n)} = \mathcal{O}(n \cdot log(n))$. Cena paralelného algoritmu je optimálna v prípade, že $c(n) = t_{seq}(n)$, kde t_{seq} je časová zložitosť optimálneho sekvenčného riešenia a $c(n)$ je cena paralelného riešenia. V našom prípade môžme vidieť, že cena má horšiu ako lineárnu zložitosť a teda paralelné riešenie **nie je optimálne**.

Teraz predpokladajme počet procesorov N, kde $N < n$, potom za určitých podmienok vieme dosiahnuť optimálnosť algoritmu. Každý procesor nájde maximum z n/N prvkov, čím vytvoríme vektor lokálnych maximálnych hodnôt každého procesoru. Na tento vektor sa použije klasická stromová metóda operácie prescan a výsledky sú potom použité ako offset pre finálnu (už sekvenčnú) lokálnu operáciu prescan na každom procesore (viz Obr. 3). Keďže každý procesor má teraz viac ako jednu hodnotu aj body 2 a 4 zo sekcie 2.a si vyžadujú čas n/N. Časová zložitosť teda bude:

$$\mathbf{t(n)} = 4 \cdot n/N + 2 \cdot \lceil log_2(N) \rceil = \mathcal{O}(n/N + log(N))$$

Z tohto vzorca vidíme jednu príjemnú vlastnosť a to, že počtom procesorov dokážeme ovplyvniť časovú zložitosť algoritmu. Pr.: (1) Ak $N = 1$ algoritmus sa degraduje na sekvenčnú verziu a $t(n) = \mathcal{O}(n)$, (2) Ak $N = n$ potom $t(n) = \mathcal{O}(log(n))$, čo je výrazne rýchlejšie oproti $\mathcal{O}(n)$ ale ako sme ukázali vyššie nie optimálne.

Avšak optimálne zrýchlenie oproti sekvenčnej verzii dosiahneme ak platí, že: $\mathbf{log_2(N) \leq n/N}$. Pričom sa snažíme aby obe strany nerovnice mali k sebe čo najbližšie. Potom môžme povedať, že časová zložitosť je $t(n) = \mathcal{O}(n/N) \Rightarrow c(n) = \mathcal{O}(n/N) \cdot N = \mathcal{O}(n) \Rightarrow$ čo **je optimálne** (koľkokrát zvýšime počet procesorov toľko krát sa zrýchli algoritmus).



Obr. 3: Prescan v situácii, keď jeden procesor vlastní viac hodnôt.

3 Implementácia

3.a `test.sh`

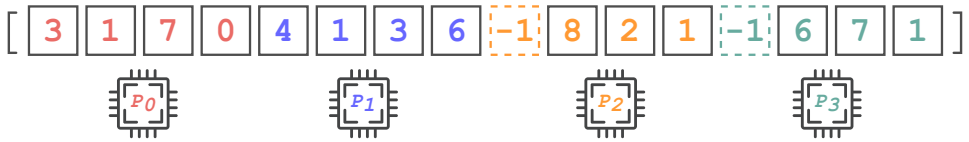
Prvou časťou implementácie je testovací skript v jazyku `shell`, ktorého úlohou je preložiť a spustiť program a tým pádom musí určiť počet potrebných procesorov. Pri určovaní počtu procesorov sme kládli dôraz na optimálnosť implementovaného riešenia a preto sme pri výpočte vychádzali zo vzorca $\mathbf{log_2(N) \leq n/N}$. To znamená, že aby bolo riešenie optimálne musí platiť vyššie uvedená nerovnosť pričom sa snažíme nájsť taký počet procesorov aby obe strany nerovnice mali k sebe čo najbližšie.

Keďže práca s logaritmami a celkovo s desatinnými číslami nie je v `shell`-i práve najlepšia, je na výpočet zavolaný `python` interpret. Konkrétne sme použili funkciu `scipy.optimize.solve`, ktorá nám vráti hodnotu N takú, že $log_2(N)=n/N$. No a keďže implementovaný algoritmus prescan prevzatý z prednášky kurzu PRL [1] predpokladá, že počet procesorov je mocnina čísla 2, výsledná hodnota je ešte naviac zaokrúhlená na najbližšiu nižšiu[1] mocninu dvojky.

3.b `vid.cpp`

Samotný algoritmus je implementovaný v jazyku `C++` za použitia knižnice `Open MPI`. Na samotnom začiatku sa spustí funkcia `MPI_Init` na inicializáciu `MPI` prostredia. Zároveň si každý proces zistí svoje id (`rank`) a celkový počet procesorov.

Následné je volaná funkcia `read_altitudes`, v ktorej procesor s `rank`-om 0 načíta vstup a rozdistribuuje hodnoty ostatným procesorom. Na distribúciu hodnôt je použitá funkcia `MPI_Scatter`. Keďže počet procesorov je mocnina dvojky, ale veľkosť vstupu je ľubovoľná, môže nastať situácia, že `n%N != 0` (`n`-veľkosť vstupu a `N`-počet procesorov). V takom prípade procesor 0 pridá do vstupnej postupnosti „padding" (viz Obr. 4), keďže funkcia `MPI_Scatter` pošle každému procesoru rovnakú časť vstupných dát. Na záver ešte procesor s `rank`-om 0 pomocou funkcie `MPI_Bcast` pošle hodnotu nadmorskej výšky pozorovateľa a taktiež celkovú veľkosť vstupu (`n`).



Obr. 4: Padding vo vstupnom vektore, ktorý zabezpečí rovnomernú distribúciu hodnôt medzi procesory. Pri ďalšom spracovaní údajov je pridaný padding rozpoznaný a ignorovaný.

Potom každý procesor prevedie pridelené hodnoty nadmorských výšok na uhly (`compute_angle`) a v týchto uhloch nájde maximum (`find_local_max`). Po týchto dvoch volaniach prichádza na rad operácia prescan. Jej implementácia je rozdelená do dvoch funkcií:

1. `up_sweep()`

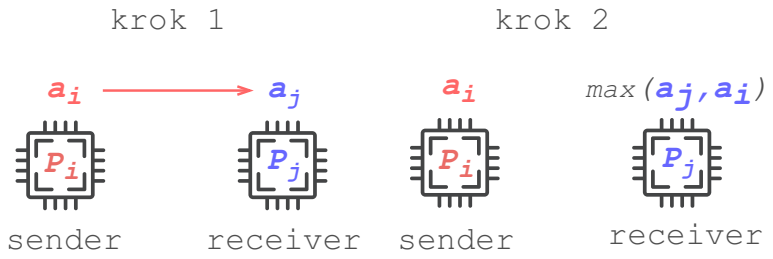
   Túto funkciu už každý procesor volá s jedinou hodnotou (jeho lokálne maximum). Kľúčovou úlohou v tejto funkcii je v každom z `log2(N)` (`N`-počet procesorov) krokov určiť, ktoré procesori budú spolu komunikovať. Procesor môže mať v každom kroku jednu z dvoch rolí: (1) buď bude svoju hodnotu posielať svojmu susedovi `sender` (2) alebo bude hodnotu od svojho suseda príjímať `receiver`.

   Pri určovaní role procesora boli použité vlastnosti binárneho stromu. Uvažujme `rank` $\in [1, 2, ..., \texttt{N}]$ a `step` $\in [1, 2, ..., \texttt{log2(N)}]$. Potom každý procesor spĺňajúci podmienku $\texttt{rank}\%2^{\texttt{step}} == 0$ zastáva v kroku `step` rolu `receiver`. To ktorý procesor má zastávať rolu `sender` v kroku `step` sa vždy

[1] Zaokrúhlením na vyššiu mocninu by prestala platiť nerovnosť $\log_2(N) \leq n/N$.

určí v kroku `step-1`. V prvom kroku budú rolu `sender` zastávať procesory spĺňajúce podmienku $\texttt{rank}\%2 == 1$. Potom sa v kroku `step`, každý `receiver`, ktorý nespĺňa podmienku $\texttt{rank}\%2^{(\texttt{step}+1)} == 0$ stane `sender`-om pre krok `step+1`.

Teraz už každý procesor vie svoju rolu, a ďalšou úlohou je zistiť s kým má komunikovať. Za účelom toho si v každom kroku každý z nich vypočíta `offset` svojho suseda nasledovne: $2^{(\texttt{step}-1)}$. Potom každý `sender` komunikuje s procesorom `sender_rank+offset` a každý `receiver` s procesorom `receiver_rank-offset`. Priebeh samotnej komunikácie je zobrazený na obrázku 5. Na implementáciu zasielania a prijímania správ boli použité funkcie `MPI_Send` a `MPI_Recv`.



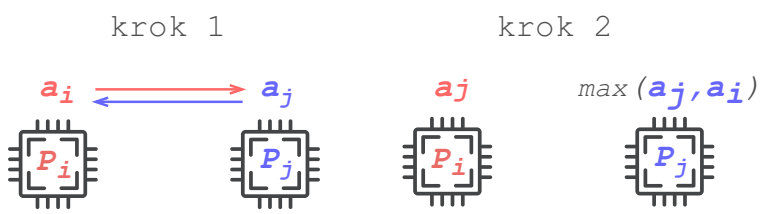
Obr. 5: Komunikácia vo fáze `upSweep` operácie prescan. Vo význame binárneho stromu `sender` predstavuje ľavého syna. `receiver` zastáva rolu pravého syna a zároveň koreňa.

2. `down_sweep()`

   V tejto funkcii už nerozlišujeme role `sender` a `receiver`, keďže každý procesor, ktorý sa podieľa na komunikácii musí svoju hodnotu poslať svojmu susedovi a taktiež prijať jeho hodnotu. To znamená, že jedinou úlohou je rozhodnúť ktoré z procesorov budú v danom kroku komunikovať. Tentokrát uvažujme $\texttt{rank} \in [1, 2, ..., \texttt{N}]$ a $\texttt{step} \in [\texttt{log(N)} - 1, \texttt{log(N)} - 2, ..., 0]$. Potom každý procesor spĺňajúci podmienku $\texttt{rank}\%2^{\texttt{step}} == 0$ bude v kroku `step` komunikovať. `offset` suseda, s ktorým bude daný procesor komunikovať sa spočíta podobne ako vo fáze `upSweep` s malým rozdielom, keďže používame iné číslovanie krokov: $\texttt{offset} = 2^{\texttt{step}}$. V prípade, že je v danom kroku procesor koreňom (`root`) `rank` svojho suseda vypočíta ako `root_rank-offset`. Ostatné procesory na zistenia suseda pripočítajú `offset` k svojmu `rank`-u.

   Poslednou úlohou tak ostáva rozhodnúť, ktorý z procesorov bude v aktuálnom kroku zastávať úlohu koreňa. Na začiatku sa koreňom stane procesor s najvyšším `rank`-om (ktorý zároveň nastaví svoju hodnotu na `-DBL_MAX`). Potom sa každý komunikujúci procesor v kroku `step` stane koreňom pre krok `step-1`.

   Komunikácia potom prebieha tak, že koreň pošle svoju hodnotu susedovi a od neho príjme jeho hodnotu. Z týchto dvoch hodnôt si ponechá `maximum`. Procesor, ktorý nieje koreňom svoju hodnotu pošle susedovi (svojmu koreňu) a zároveň od neho príjme jeho hodnotu, ktorú si ponechá ako aktuálnu (viz Obr. 6). Na implementáciu komunikácie bola tentokrát použitá funkcia `MPI_Sendrecv`.



Obr. 6: Komunikácia vo fáze `downSweep` operácie prescan. Vo význame binárneho stromu $\texttt{P}_\texttt{i}$ predstavuje ľavého syna. $\texttt{P}_\texttt{j}$ zastáva rolu pravého syna a zároveň koreňa.

Po zavolaní týchto dvoch funkcií je teda operácia prescan dokončená. Avšak keďže v našej implementácii má každý procesor viacero hodnôt je potrebné aby si každý z nich, výpočet dokončil lokálne pričom svoju hodnotu použije ako offset (viz Obr. 3). To sa odohráva vo funkcii `compute_local_visibility()`, kde každý procesor spočíta viditeľnosti pre všetky svoje body. Výsledkom je už pole znakov (`char*`), kde `'v'` označuje viditeľné body a `'u'` neviditeľné body.

Poslednou časťou implementácie je funkcia `print_result`, kde procesor s `rank`-om 0 zozbiera výsledky od všetkých procesorov pomocou `MPI_Gather` a následne ich vypíše.

4 Experimenty

Cieľom experimentov bolo overiť časovú zložitosť paralelného algoritmu *Viditeľnosť*. Pre účely zistenia časovej zložitosti bolo potrebné eliminovať rušivé vplyvy, ktoré nesúvisia so samotným algoritmom. Preto sme ignorovali čas potrebný na načítanie a distribúciu hodnôt medzi procesory, ako aj čas potrebný na spätnú propagáciu výsledných hodnôt a výpis. V konečnom dôsledku sme tak merali iba čas strávený v hlavnom tele algoritmu popísanom v sekcii 2.a, na základe ktorého bola vykonaná aj analýza algoritmu v sekcii 2.c.

Pri meraní času paralelných programov, ale musíme byť trochu opatrnejší, pokiaľ ide o spôsob merania času. V našom programe je časť kódu, ktorú chceme odmerať, spúšťaná viacerými procesmi a tým pádom pri meraní času získame množinu niekoľkých časov (každý proces = samostatný čas). Avšak, na to aby sme mohli overiť časovú zložitosť potrebujeme jeden čas a to čas, ktorý uplynie od okamihu, keď prvý proces začne vykonávať meraný úsek kódu až po čas kedy posledný proces skončí vykonávanie meraného kódu. Konkrétnu implementáciu tohto procesu môžte vidieť v Listingu 1.

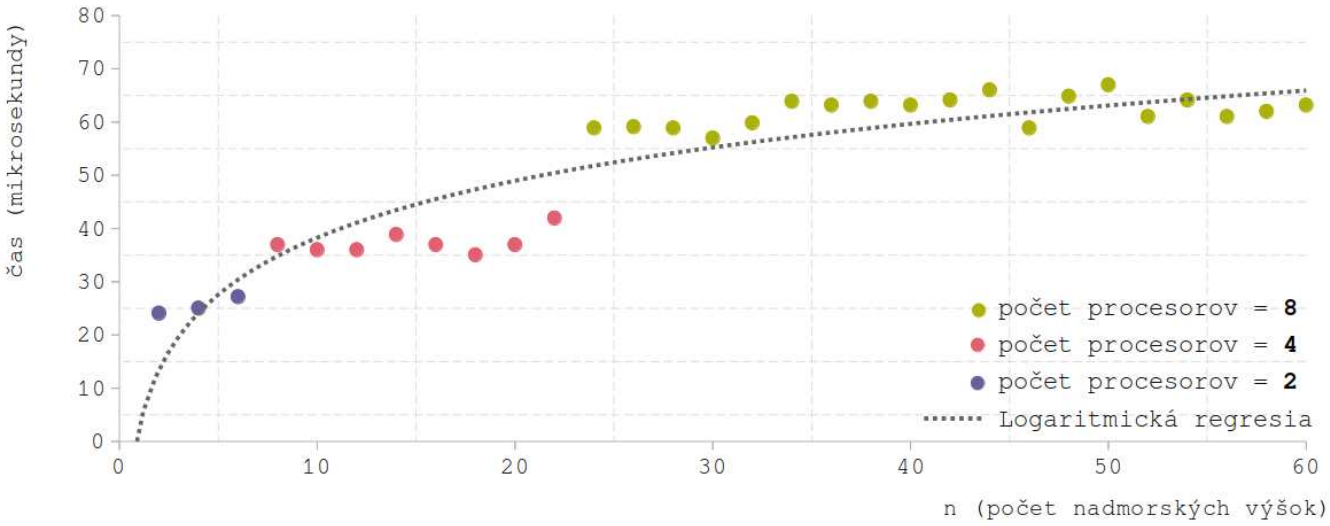
```
1    /* Synchronize all processes */
2    MPI_Barrier(MPI_COMM_WORLD);
3    local_start = MPI_Wtime();
4    /* Code to be timed */
5    ...
6    local_finish = MPI_Wtime();
7    local_elapsed = local_finish - local_start;
8    MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
9
10   if (my_rank == 0)
11       printf("Time = %g us\n", elapsed * 1000000);
```

Listing 1: Meranie času testovaného algoritmu

Na implementáciu sme použili funkcie, ktoré ponúka knižnica `Open MPI`. Prvým krokom je volanie `MPI_Barrier`, ktorá aspoň približne zosynchronizuje všetky procesy a následne každý z nich zistí svoj čas pomocou funkcie `MPI_Wtime`. Na záver každý proces volá fukncu `MPI_Reduce`, ktorá vráti čas najpomalšieho procesu. V prípade, že hodnota `MEASURE_TIME` je `1`, proces 0 vypíše zmeraný čas algoritmu v mikrosekundách.

Posledným úskalím, s ktorým sme sa museli popasovať pri meraní času je variabilita výsledných časov. Ak náš program spustíme niekoľko krát, je veľmi pravdepodobné, že výsledný čas bude zakaždým iný. A to dokonca aj v prípade, že ho spustíme na rovnakom systéme a s rovnakým vstupom. Ako riešenie sa ponúka použiť ako výsledný čas priemer alebo medián z nameraných výsledkov. Avšak je veľmi nepravdepodobné, že rýchlosť nášho algoritmu by mohla byť zlepšená nejakou vonkajšou udalosťou a tak namiesto strednej hodnoty alebo mediánu, ako výsledok reportujeme najnižšiu z nameraných hodnôt [3].

Samotné experimenty prebiehali na školskom serveri `merlin` a to tak, že pre každú zvolenú veľkosť vstupného reťazca nadmorských výšok bolo vykonaných 50 meraní a z nich bola tá najmenšia hodnota vybraná ako výsledná. Výsledky experimentov sú zobrazené v grafe na Obrázku 7.



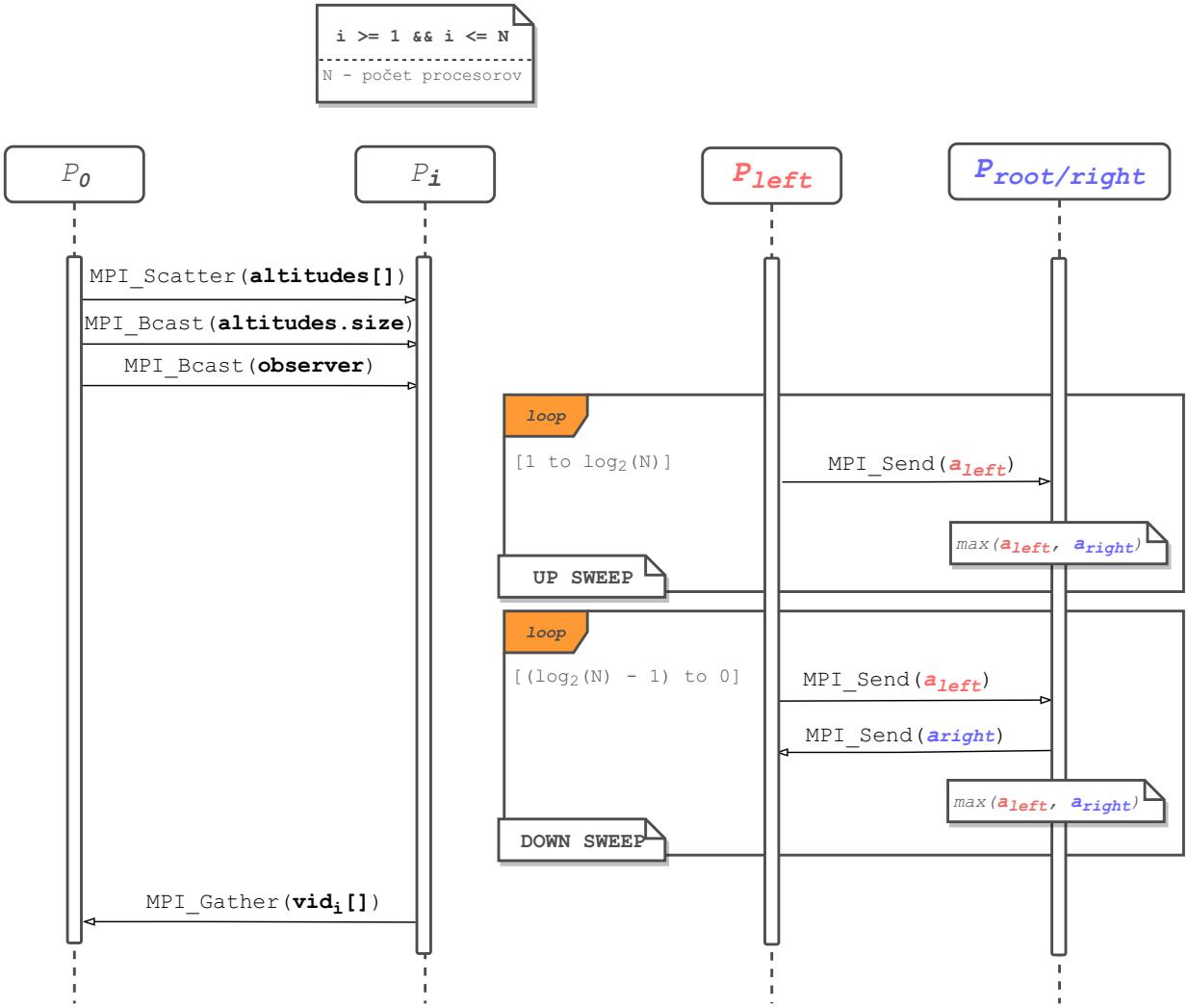
Obr. 7: Experimentálne overenie časovej zložitosti. V počte nadmorských výšok sa neuvažuje nadmorská výška pozorovateľa.

5 Zhodnotenie experimentov

Výsledky meraní poukazujú na fakt, že by sa malo jednať o logaritmickú časovú zložitosť (viz Obr. 7). Z obrázku môžme vidieť skokový nárast času pri vstupe o veľkosti 8 resp. 24 prvkov. Práve pri vstupoch tejto veľkosti sa mení počet procesorov na vyššiu mocninu dvojky a tým pádom sa pridá nová úroveň do binárneho stromu. Ako je aj z obrázku viditeľné časy algoritmu sa pri rovnakom počte procesorov už výrazne nelíšia, pričom mierne výkyvy možno pripísať rôznemu zaťaženiu školského servera v čase, kedy prebiehali experimenty. Z vyššie uvedeného sa teda ukazuje že čas narastá logaritmicky s počtom procesorov.

Toto zistenie podložené vykonanými experimentami je v súlade s odvodenou teoretickou časovou zložitosťou v sekcii 2.c, kde sme odvodili $\mathbf{t}(\mathbf{n}) = \mathcal{O}(n/N + log(N))$. Keďže sa snažíme o optimálnosť a teda o $log(N) \approx n/N$ potom môžme písať $t(n) = \mathcal{O}(log(N))$, čo približne odpovedá nameraným výsledkom.

6 Komunikačný protokol



Obr. 8: Komunikačný protokol zasielania správ. Pole `altitudes[]` už neobsahuje nadmorskú výšku pozorovateľa. O tom či je procesor koreň/pravý syn alebo ľavý syn rozhodne jeho index viz Sekcia 3.b (Obr. 5, Obr. 6). Prvky a_{left}, a_{right} sú už lokálne spočítané uhly jednotlivými procesormi. A pole $vid_i[]$ je už výsledné pole obsahujúce viditeľnosti lokálnych bodov každého procesoru.

Literatúra

[1] *PRAM* (prednáška PRL). 2008.
URL `https://www.fit.vutbr.cz/study/courses/PDA/private/www/h006.pdf`

[2] Blelloch, G. E.: Prefix Sums and Their Applications. Technická zpráva, Synthesis of Parallel Algorithms, 1990.

[3] Pacheco, P.: *An Introduction to Parallel Programming*. Morgan Kaufmann, 2011, ISBN 9780123742605 0123742609.