Dokumentácia k projektu do predmetu PRL
Implementácia úlohy *Viditeľnosť*

Matúš Liščinský
xlisci02@stud.fit.vutbr.cz

Úvod

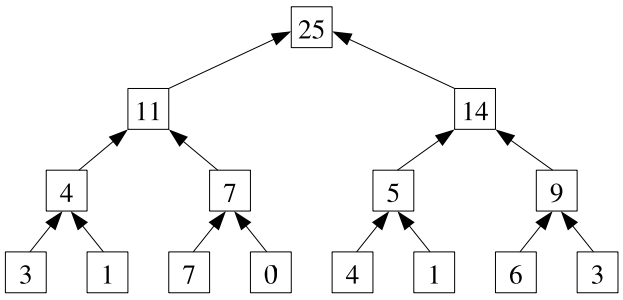
Úlohou tohto projektu bolo implementovať paralelný algoritmus pre riešenie úlohy *Viditeľnosť* v jazyku C/C++ pomocou knižnice Open MPI. Okrem samotného algoritmu bolo potrebné vytvoriť testovací shell skript, ktorého cieľom je riadiť testovanie projektu.

Popis algoritmu

Pred samotným popisom algoritmu, ktorý rieši úlohu Viditeľnosti si potrebujeme ozrejmiť riešený problém. Vstupom je nadmorská výška pozorovacieho bodu a vektor reprezentujúci nadmorské výšky bodov pozdĺž paprsku vychádzajúceho z pozorovacieho bodu. Úlohou je rozhodnúť, ktoré body pozdĺž paprsku sú viditeľné. Daný bod je viditeľný, ak žiaden bod medzi pozorovateľom a ním nemá väčší vertikálny uhol.

Je zrejmé, že na začiatku potrebujeme vypočítať vertikálny uhol každého bodu vektora. Kľúčom k riešeniu úlohy je však využitie operácií *reduce* a *prescan*. Operácia *reduce* má ako vstup postupnosť prvkov $[a_0, a_1, \ldots, a_{n-1}]$, binárny asociatívny operátor $\oplus$ a vracia hodnotu $a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1}$. Podobnou operáciou je *prescan*, s tým rozdielom, že žiada na vstupe aj neutrálny prvok I a jej výsledkom je postupnosť $[I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})]$. Binárnym asociatívnym operátorom pre účely riešenia úlohy je operátor `max`.

Operáciu *reduce* počítame v paralelnom prostredí pomocou stromu procesorov. Berúc v úvahu EREW PRAM , každá úroveň stromu môže byť počítaná paralelne, takže výpočet môže postupovať od listov stromu ku koreňu a preto tomu hovoríme aj *up-sweep*. Ukážka fungovania operácie na strome procesorov je znázornená na obrázku 1.



Obr. 1: Operácia *reduce* s operátorom + a využitím stromu procesorov. [1]

Časová zložitosť operácie *reduce* je rovná výške stromu, teda $t(n) = \mathcal{O}(log(n))$. Pre každú dvojicu prvkov potrebujeme jeden procesor z čoho plynie $p(n) = n/2$ a celková cena je $c(n) = t(n) \cdot p(n) = \mathcal{O}(n \cdot log(n))$. V prípade, že počet procesorov $N < n$, každý procesor musí pred výpočtom vykonať sekvenčný reduce pre svoju časť postupnosti o dĺžke n/N. Tým sa mení aj jeho časová zložitosť na $t(n) = \lceil n/N \rceil + \lceil log(N) \rceil = \mathcal{O}(n/N + log(N))$. Avšak ak je dodržaná podmienka $log_2(N) \leq n/N$ (ideálne ak $log_2(N) = n/N$), tak časová zložitosť $t(n) = \mathcal{O}(n/N)$ a algoritmus má cenu $c(n) = \mathcal{O}(n/N) \cdot N = \mathcal{O}(n)$. Za tejto podmienky je algoritmus optimálny, pretože jeho cena sa rovná časovej zložitosti optimálneho sekvenčného riešenia, teda $\mathcal{O}(n)$.

Operácia *prescan* na strome procesorov predstavuje postupnosť troch operácií: *up-sweep*, *clear*, a *down-sweep*. Operácia *up-sweep* je spomínaná operácia *reduce* s tým, že každý uzol si pamätá svoju hodnotu. Ulože-

nie neutrálneho prvku do koreňového uzlu, alebo aj operácia *clear* je predpríprava pre operáciu *down-sweep*. Tá postupuje od koreňa k listom tak, že každý uzol aktuálnej hĺbky zanorenia si vymení hodnotu s ľavým synom, a získanú hodnotu pošle svojmu pravému synovi, ktorý na ňu a svoju hodnotu aplikuje daný operátor, v našom prípade `max`. Po dokončení *down-sweep* obsahuje každý uzol maximálnu hodnotu všetkých listov, ktoré ho predchádzajú [1].

Výsledky z *prescan*-u sú následne použité na sekvenčný *prescan* v rámci $\lceil n/N \rceil$ prvkov každého procesora. Je zrejmé, že zložitosť i cena operácie *down-sweep* je rovnaká ako pri *reduce* spolu s podmienkou optimálnosti ($log_2(N) \leq n/N$).

Na záver získavame výsledky o viditeľnosti/neviditeľnosti jednotlivých bodov z porovnania uhla bodu s odpovedajúcim výsledkom operácie *max-prescan*.

Analýza algoritmu

Algoritmus pre riešenie úlohy viditeľnosti sa skladá z niekoľkých častí:

1) Každý procesor počíta uhly $\lceil n/N \rceil$ bodov, zložitosť $\mathcal{O}(n/N)$.
2) Operácia *max-reduce*, zložitosť $\mathcal{O}(n/N + log(N))$.
3) Operácia *max-prescan*, zložitosť $\mathcal{O}(n/N + log(N))$.
4) Každý procesor porovná $\lceil n/N \rceil$ uhlov s výsledkami *max-prescan* operácie, zložitosť $\mathcal{O}(n/N)$.

Z toho vyplýva, že celková časová zložitosť algoritmu je $t(n) = 4 \cdot \lceil n/N \rceil + 2 \cdot \lceil log(N) \rceil = \mathcal{O}(n/N + log(N))$. Optimálnosť algoritmu závisí od počtu použitých procesorov, rovnako ako pri operácii *reduce* či *prescan*. Snažíme sa o dodržanie vzťahu $log_2(N) \leq n/N$, s cieľom dosiahnuť čo najmenší rozdiel medzi pravou a ľavou stranou nerovnice. Vtedy sa mení časová zložitosť na $t(n) = \lceil n/N \rceil + \lceil n/N \rceil = \mathcal{O}(n/N)$ čoho dôsledkom je cena $c(n) = \mathcal{O}(n)$. Za splnenia tejto podmienky môžeme konštatovať, že algoritmus **je optimálny** pretože jeho cena sa rovná časovej zložitosti optimálneho sekvenčného algoritmu $t_{seq}(n) = \mathcal{O}(n)$.

Implementácia

Tento algoritmus je implementovaný v jazyku C++ za pomoci knižnice pre paralelné výpočty Open MPI. Pred samotným riešením úlohy je potrebné určiť počet procesorov, s ktorým bude program spúšťaný. Výpočet prebieha v rámci skriptu `test.sh` za pomoci matematickej knižnice jazyka `python3` nasledovne:

1. Vypočíta sa optimálny počet procesorov N_{opt} tak, aby platila nerovnica $log_2(N_{opt}) \leq n/N_{opt}$ a zároveň pravá a ľavá strana nerovnice mali čo najmenší rozdiel. Tento problém je preformulovaný na úlohu hľadania takého N_{opt}, aby výsledok funkcie $log_2(N_{opt}) - n/N_{opt}$ bol čo najmenšie nezáporné číslo. Berie sa ohľad aj na na limitáciu školského servera *merlin* a preto je optimálny počet procesorov N_{opt} zhora obmedzený na hodnotu 25.
2. Keďže implementovaný algoritmus predpokladá počet procesorov $N = 2^k$, kde $k \in \mathbb{N}_0$, počet procesorov N v našom prípade predstavuje najbližšiu mocninu čísla 2 ku N_{opt}, pre ktorú platí: $N \leq N_{opt}$. Výsledné N získame jednoducho vyhodnotením výrazu: $2^{\lfloor (log_2(N_{opt})) \rfloor}$.

Ďalej už budeme opisovať implementáciu riešenia úlohy *Viditeľnosť* . Na začiatok je pre inicializáciu MPI prostredia nevyhnutné zavolať knižničnú funkciu `MPI_Init`, po ktorej si už každý procesor uloží celkový počet procesorov a svoje id (rank) v rámci skupiny procesorov. Procesor s $id = 0$ následne prečíta sekvenciu nadmorských výšok z poľa vstupných argumentov, uloží si ju do vektora typu `int` a pomocou funkcie `MPI_Bcast` informuje ostatné procesory o dĺžke sekvencie a nadmorskej výške pozorovacieho bodu. Procesor s $id = 0$ následne distribuuje časti sekvencie medzi jednotlivé procesory pomocou funkcie `send_int_to` a ostatné procesory svoje sekvencie prijímajú funkciou `recv_int_from`. Funkcie `send_int_to` a `recv_int_from`

iba obaľujú funkcie `MPI_Send` a `MPI_Recv` s implicitnou hodnotou počtu prvkov (1) a bez vyžadovania konštantných parametrov, ako je dátový typ (`MPI_INT`), tag, komunikátor (`MPI_COMM_WORLD`), resp. adresa `MPI_Status` štruktúry. Počet prvkov pridelených pre procesor počíta funkcia `get_proc_value_count` podľa id procesora s ohľadom na to, aby bolo rozloženie čo najrovnomernejšie. Prepočet nadmorských výšok na uhly už rieši každý procesor samostatne, no na to potrebuje index daného bodu v pôvodnej sekvencii, ktorý získa funkciou `get_global_alt_index`. Keďže výpočet uhla na pozícii pozorovateľa nemá zmysel, na túto pozíciu sa nastaví hodnota uhla na $-\pi/2$ a to z toho dôvodu, že funkcia $arctan$ na výpočet uhlov zľava konverguje ku hodnote $-\pi/2$. Z vypočítanej sekvencie uhlov si procesory paralelne spočítajú jednu hodnotu `myangle` funkciou `seq_reduce`, ktorá obsahuje sekvenčnú implementáciu operácie *max-reduce*. Za ňou už nasleduje operácia *max-prescan*. Tú predstavuje funkcia `up_sweep`, kópia hodnoty z `up_sweep`, nastavenie neutrálneho prvku koreňového procesora (opäť na $-\pi/2$) a posledne funkcia `down_sweep`.

Vo funkcii `up_sweep` je implementovaný paralelný algoritmus operácie *max-reduce* postupujúci od listov stromu procesorov ku jeho koreňu. V aktuálnej hĺbke stromu pošle každý ľavý syn istého uzla svoju hodnotu pravému synovi tohto uzla, ten ju prijíma a aplikuje na ňu a svoju hodnotu funkciu `fmax` a výsledok si uloží. Komunikácia prebieha pomocou funkcií `send_double_to` a `recv_double_from`, ktoré majú obdobnú funkciu ako už spomínané `send_int_to` a `recv_int_from`, ale pracujú nad typom `double`, resp. `MPI_DOUBLE`. Identifikátory ľavých synov (odosielateľov) a pravých synov (prijímateľov) sa počítajú podľa id procesora a aktuálnej hĺbky zanorenia prostredníctvom funkčných makier `up_sweep_sender` a `up_sweep_reciever`.

Funkcia `down_sweep` obsahuje prechod stromu procesorov od koreňa k listom s tým, že na aktuálnej úrovni si každý rodičovský uzol vymení hodnotu so svojim ľavým synom (kde sa ešte nachádza hodnota z funkcie `up_sweep`), aplikuje na ňu a svoju uloženú hodnotu funkciu `fmax` a výsledok uloží do jeho pravého syna, čo v praxi znamená že si ju uloží on sám. Komunikácia prebieha opäť pomocou funkcií `send_double_to` a `recv_double_from` s tým, že identifikátory dvojice aktuálne komunikujúcich procesorov sa overujú v makre `down_sweep_comm_node` a identifikátor rodičovského procesora makrom `down_sweep_parent`.

Výsledky sú následne použité na sekvenčný *max-prescan* v rámci $\lceil n/N \rceil$ prvkov každého procesora, ktorý zabezpečuje funkcia `seq_prescan`. Na záver si každý procesor otestuje viditeľnosť priradených uhlov a uloží výsledky do poľa vo forme celočíselných konštánt 0 a 1, kde 1 znamená, že bod je viditeľný a 0 indikuje, že bod viditeľný nie je. Procesor s $id = 0$ prijíma tieto sekvencie od ostatných procesorov a postupne ich pridáva do vektora výsledkov. Komunikácia prebieha za pomoci funkcií `send_int_to` a `recv_int_from`. Po prijatí všetkých výsledkov vektor obsahuje sekvenciu čísel, ktoré sa dekódujú funkčným makrom `decode` a vypíšu na štandardný výstup. Následne sa po výpise použité polia uvoľnia z pamäte a zavolá sa funkcia `MPI_Finalize`, ktorá ukončí prostredie vykonávania MPI.

Experimenty

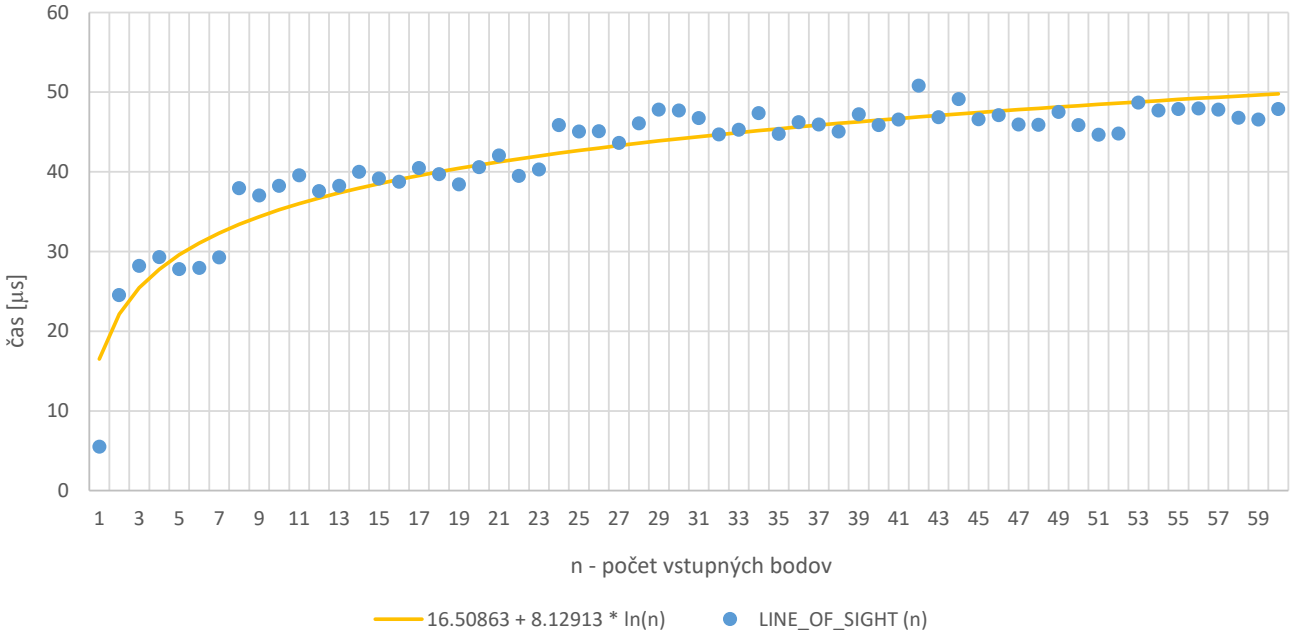
Pre overenie časovej zložitosti bol implementovaný algoritmus experimentálne otestovaný. Zaujímal nás predovšetkým výkon algoritmu bez okolitej réžie s ním spojenej, čo zahŕňa napríklad čítanie hodnôt, výpis výsledkov na štandardný výstup, a podobne. Experimentom sa podrobila časť kódu od prepočtu bodov na uhly až po testovanie viditeľnosti jednotlivých bodov, teda kód funkcie `LINE_OF_SIGHT` z prednáškových textov [1]. Režim testovania je v kóde možné zapnúť definovaním makra `TEST`.

Pred začiatkom testovania je volaním funkcie `MPI_Barrier` vytvorená explicitná bariéra, aby boli všetky procesy pred spustením testovania zosynchronizované na rovnakej pozícii. Jednotlivé časy behu programu sú získané vďaka funkcii `MPI_Wtime`, rozdielom časových údajov pred a po vykonaní meraného úseku kódu. Výsledná hodnota pri jednom behu je zo všetkých procesorov zredukovaná na jednu (maximálnu) hodnotu funkciou `MPI_Reduce`. Testovaná časť kódu neobsahuje žiadne I/O operácie, preto nebolo potrebné zabezpečiť ich preskočenie pri behu programu v rámci experimentálneho testovania.

Pre účely testovania bol vytvorený bash skript, ktorý automatizovane spúšťal testovací skript so sekvenciou 1 až 60 nadmorských výšok a pre každý prípad sa algoritmus otestoval 15-krát. Testovací skript `test.sh`

je ošetrený tak, že berie práve jeden argument v podobe sekvencie prirodzených čísel oddelených čiarkou, v inom prípade končí chybovou hláškou. Pri výsledných časoch sa objavila istá miera variability, preto bolo odstránených 5 najvyšších časov a zo zvyšných 10 výsledkov sa vypočítal priemerný čas. Do výsledkov za nezahŕňali behy s výrazne odľahlými časovými výsledkami presahujúce prah $200\mu s$. Pre lepšiu predstavu bola z nazbieraných dát pomocou logaritmickej regresie[1] získaná funkcia $16.50863 + 8.12913 \cdot ln(n)$, ktorou sú preložené namerané dáta na obrázku 2.

Implementácia bola priebežne testovaná na lokálnom systéme *Ubuntu 19.10*, na školskom serveri *merlin*, a na superpočítači *salomon*, na ktorý sme ako študenti predmetu AVS dostali prístup. Výsledky experimentov sú práve z testovania na superpočítači, z dôvodu väčšej stability výsledných časov.



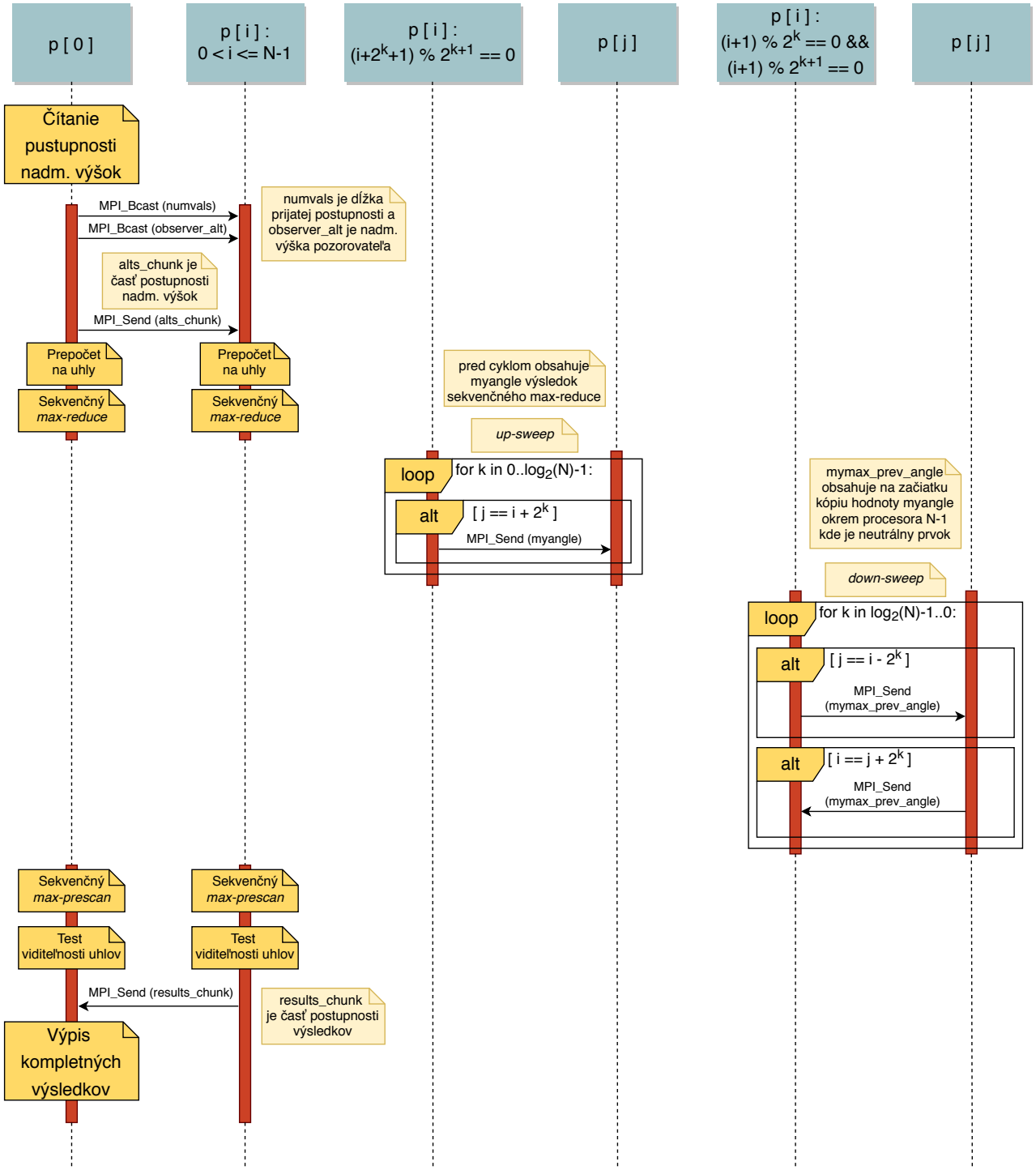
Obr. 2: Výsledky experimentálnych meraní spolu s výsledkom logaritmickej regresie z nameraných dát.

Záver

Na základe teoretických znalostí o princípe fungovania paralelných algoritmov pre operácie *reduce* a *prescan* a ich použitia pre riešení úlohy *Viditeľnosti* sme vyvodili časovú zložitosť $t(n) = \mathcal{O}(n/N) \sim O(log(N))$ za podmienky $log_2(N) \leq n/N$ a cieľom experimentov bolo prakticky ju overiť. Z grafu pozorujeme, že dáta aproximujú logaritmickú závislosť a tým potvrdzujú teoretický podklad o časovej zložitosti tohto algoritmu. Výsledky môžeme rozdeliť do celkovo štyroch celkov. V tom prvom sa nachádza beh programu s jedným vstupným bodom ($n = 1$), ktorý ako jediný pracoval iba nad jedným procesorom. Pri vstupe o veľkosti $2 \leq n \leq 7$ boli do práce zapojené 2 procesory čomu odpovedajú aj podobné časové údaje blížiace sa k $30\mu s$. V poradí tretí celok predstavuje vstup o veľkosti $8 \leq n \leq 23$, využívajúci 4 procesory a opäť vidíme, že dosiahnutý čas cca $40\mu s$ je pre každý vstup z tejto skupiny rovnaký. Posledným celkom sú časové údaje, kde bol použitý vstup $24 \leq n \leq 60$. Tu pracovalo až 8 procesorov a beh algoritmu dosahoval čas zásadne nad hranicou $40\mu s$ blížiac sa ku $50\mu s$. Konštatujeme, že v dôsledku správne zvoleného počtu procesorov sa časy behov algoritmu líšili hlavne pri zmene počtu procesorov N. Navyše pozorujeme, že časové rozdiely sa držia logaritmickej závislosti, čo odpovedá časovej zložitosti algoritmu. Komunikačný protokol vo forme sekvenčného diagramu (obrázok 3) sa nachádza na ďalšej strane spolu s použitou literatúrou.

[1] https://keisan.casio.com/exec/system/14059930226691

Komunikačný protokol



Obr. 3: Sekvenčný diagram znázorňujúci komunikáciu medzi N procesormi.

Literatúra

[1] Hanáček, P.: *Model PRAM, suma prefixů*. online, 2019.
URL https://wis.fit.vutbr.cz/FIT/st/cfs.php.cs?file=%2Fcourse%2FPRL-IT%2Flectures%2FPDA06_PRAM_MNG.pdf