Věž a jezdec na šachovnici
Semestrální projekt NI-PDP 2020/2021

Lukáš Litvan
litvaluk@fit.cvut.cz

11. května 2021

1 Definice problému

Vstupní data

- k = přirozené číslo, $20 > k > 5$, reprezentující délku strany šachovnice S o velikosti $k \times k$
- q = přirozené číslo $q < \frac{k^2}{2}$ reprezentující počet rozmístěných figurek na šachovnici S
- $C\,[1..q]$ = pole souřadnic rozmístěných figurek na šachovnici S
- I = souřadnice věže na šachovnici S
- J = souřadnice jezdce (koně) na šachovnici S

Pravidla a cíl hry

Na počátku je na čtvercové šachovnici S rozmístěno q figurek a 1 věž a 1 jezdec. Tomuto rozmístění figurek budeme říkat počáteční konfigurace. Jeden tah je posun věže podle šachových pravidel či posun jezdce podle šachových pravidel. První tah provádí věž. Pokud věž či jezdec táhnou na políčko obsazené některou figurkou, seberou ji. Věž s jezdcem se musejí pravidelně střídat v tazích. Věž nikdy nesebere jezdce a jezdec nikdy nesebere věž, i kdyby tato možnost vznikla. Cílem hry je odstranit všechny figurky pomocí minimálního počtu tahů tak, aby se šachovnice dostala do cílové konfigurace, kdy na ní zůstane samotná věž a jezdec.

Výstup algoritmu

Nejkratší posloupnost střídavých tahů věže a jezdce vedoucí do cílové konfigurace. Posloupnost má formát seznamu souřadnic políček, na které střídavě táhne věž a jezdec s označením hvězdičkou těch políček, kde došlo k odstranění některé figurky.

2 Popis sekvenčního algoritmu

Sekvenční algoritmus, který byl implementován, je typu BB-DFS. Jedná se tedy o prohledávání stavového prostoru do hloubky, přičemž jsou ořezávány větve stavového prostoru, které nemá smysl procházet. Algoritmus byl implementován rekurzivně, přičemž pokud narazí na stav, který nesplňuje podmínku $aktuální\_hloubka + (q - počet\_už\_sebraných\_figurek) < aktuální\_minimum$, je větev ihned ukončena a je proveden návrat. Nejdříve je funkce volána na ty stavy, které mají největší potenciál co nejrychleji se dostat k řešení (je sebrán pěšec nebo se figurka na tahu dostala na místo, odkud dokáže na jeden tah sebrat pěšce), což v důsledku vede k brzkému prořezání stavového prostoru.

```
void dfs(Board board) {
  // if there are no pieces left and the solution is better
  // set current moves as the best solution
  if (board.remaining <= 0 && board.depth < bestSolution.length) {
    bestSolution = board.performedMoves;
    return;
  }

  // branch and bound condition
  if (board.depth + board.remaining >= bestSolution.length) {
    return;
  }

  if (board.depth%2 == 0) {
    // rook on the move
    Moves moves = nextRook(board.rookIndex, board);
    sort(moves.moves, &moves.moves[moves.length], compareMoves);
    for (int i = 0; i < moves.length; i++) {
      Board executed = executeMove(board.rookIndex, moves.moves[i], board);
      dfs(executed);
    }
  } else {
    // knight on the move
    Moves moves = nextKnight(board.knightIndex, board);
    sort(moves.moves, &moves.moves[moves.length], compareMoves);
    for (int i = 0; i < moves.length; i++) {
      Board executed = executeMove(board.knightIndex, moves.moves[i], board);
      dfs(executed);
    }
  }
}
```

Kód 1: Rekurzivní funkce řešící danou úlohu

3 Popis paralelního algoritmu a jeho implementace v OpenMP - taskový paralelismus

Od sekvenčního řešení se paralelní algoritmus při využití taskového paralelismu výrazně neliší. Rozdílem je pouze vhodné použití OpenMP direktiv. Nejdříve je vytvořen paralelní blok pomocí `#pragma omp parallel`(ve kterém je také možno specifikovat počet vláken) a první zavolání rekurzivní funkce je zavoláno pouze jednou (`#pragma omp single`). Veškerá další volání rekurzivní funkce je pak s direktivou `#pragma omp task`, která zapříčiní, že se tato volání budou ukládat do interní fronty, odkud si je budou postupně brát jednotlivá vlákna.

```
#pragma omp parallel num_threads(threadCount)
{
  #pragma omp single
    dfs(startingBoard);
}

.
.
.

// inside the recursive dfs function
#pragma omp task
dfs(partSolution);
```

Kód 2: Paralelní blok, prvotní a další volání rekurzivní funkce v OpenMP

4 Popis paralelního algoritmu a jeho implementace v OpenMP - datový paralelismus

Oproti taskovému paralelismu už tento přístup vyžadoval menší úpravu. U datového paralelismu je cílem si rozdělit práci na části a jednotlivé části nechat vypočítat jednotlivými vlákny. Toto je v OpenMP řešeno paralelním cyklem (`#pragma omp parallel for`), který iteruje přes danou kolekci „podřešení“, která je vypočítána ještě před samotným paralelním výpočtem. Původní expanze je řešena sekvenčně prohledáváním do šířky (BFS), přičemž prohledáváme pouze do určité hloubky nebo do určitého počtu stavů ve frontě.

```
// sequential expansion by bfs
void prepare(Board board, int depthLimit) {
  list<Board> queue;
  queue.push_back(board);

  while((!queue.empty()) && queue.front().depth < depthLimit) {
    board = queue.front();
    queue.pop_front();

    if (board.depth%2 == 0) {
      // rook on the move
      Moves moves = nextRook(board.rookIndex, board);
      for (int i = 0; i < moves.length; i++) {
        queue.push_back(executeMove(board.rookIndex, moves.moves[i], board));
      }
    } else {
      // knight on the move
      Moves moves = nextKnight(board.knightIndex, board);
      for (int i = 0; i < moves.length; i++) {
        queue.push_back(executeMove(board.knightIndex, moves.moves[i], board));
      }
    }
  }

  int queueSize = queue.size();
  for (int i = 0; i < queueSize; i++) {
    prepared.push_back(queue.front());
    queue.pop_front();
  }
}
```

Kód 3: Prvotní expanze pomocí BFS

```
#pragma omp parallel for num_threads(threadCount)
for (int i = 0; i < prepared.size(); i++){
  dfs(prepared[i]);
}
```

Kód 4: Použití paralelního for cyklu v OpenMP

5 Popis paralelního algoritmu a jeho implementace v MPI

U OpenMPI řešení byl využit přístup Master-Slave. Ten funguje tak, že OpenMPI vytvoří daný počet procesů, přičemž jeden z nich je tzv. Master, který rozděluje a předává práci zbylým procesům, tzn. Slave procesům. Na začátku si Master proces předpočítá (expanduje) stavy pomocí BFS (obdobně jako u datového paralelismu), seřadí je a následně posílá (`MPI_Send`) zprávy (stavy k vyřešení) volným Slave procesům a dostává (`MPI_Receive`) zprávy od Slave procesů, které skončili s výpočtem. Co se týče Slave procesů, ty řeší problém datovým paralelismem. Pokud Master proces už nemá na poslání další stavy, pošle terminační zprávu, podle které Slave pozná, že se má ukončit. Po poslání terminační zprávy všem Slave procesům poté interpretuje výsledné řešení.

```
int k, maxDepth;
cin >> k >> maxDepth;
best.length = maxDepth;

Board board = initBoard(k);
int maxPieces = board.remaining;
vector<Board> expanded = prepare(board, parameters.firstExpandDepth);
sort(expanded.begin(), expanded.end(), compareBoards);

// send message to each slave
for (int i = 1; i < numberOfProcesses; i++) {
  Message message;
  message.end = false;
  message.expandDepth = parameters.secondExpandDepth;
  message.threadCount = parameters.threadCount;
  message.board = expanded[i-1];
  message.best = best;
  MPI_Send(&message, sizeof(message), MPI_PACKED, i, i, MPI_COMM_WORLD);
}

// when message from any slave is received, send another message to the same slave
// (until expanded is empty)
for (int i = numberOfProcesses - 1; i < expanded.size(); i++) {
  Moves received;
  MPI_Status status;
  MPI_Recv(&received, sizeof(received), MPI_PACKED, MPI_ANY_SOURCE,
           MPI_ANY_TAG, MPI_COMM_WORLD, &status);
  if (received.length < best.length) {
    best = received;
  }
  Message message;
  message.end = false;
  message.expandDepth = parameters.secondExpandDepth;
  message.threadCount = parameters.threadCount;
  message.board = expanded[i-1];
  message.best = best;
  MPI_Send(&message, sizeof(message), MPI_PACKED, status.MPI_SOURCE, i, MPI_COMM_WORLD);
}

// wait for messages from all slaves
for (int i = 1; i < numberOfProcesses; i++) {
  Moves received;
  MPI_Status status;
  MPI_Recv(&received, sizeof(received), MPI_PACKED, MPI_ANY_SOURCE,
           MPI_ANY_TAG, MPI_COMM_WORLD, &status);
  if (received.length < best.length) {
    best = received;
  }
}

// send end flag to all slaves
for (int i = 1; i < numberOfProcesses; i++) {
  Message message;
  message.end = true;
  MPI_Send(&message, sizeof(message), MPI_PACKED, i, i, MPI_COMM_WORLD);
}

printSolution(best, k, parameters);
```

Kód 5: Kód pro Master proces

```
while (true) {
  Message received;
  MPI_Status status;
  MPI_Recv(&received, sizeof(received), MPI_PACKED, MPI_ANY_SOURCE,
           MPI_ANY_TAG, MPI_COMM_WORLD, &status);
  if(received.end) {
    break;
  } else {
    vector<Board> prepared = prepare(received.board, received.expandDepth);
    best = received.best;

    #pragma omp parallel for num_threads(received.threadCount)
    for (int i = 0; i < prepared.size(); i++){
      dfs(prepared[i]);
    }

    MPI_Send(&best, sizeof(best), MPI_PACKED, 0, processRank, MPI_COMM_WORLD);
  }
}
```

Kód 6: Kód pro Slave proces

6 Naměřené výsledky a vyhodnocení

Prostředí

Měření proběhlo na výpočetním klastru Star. Maximální časový limit pro jeden běh byl 10 minut. Klastr Star má následující hardwarovou specifikaci:

Počet serverů 1

Počet uzlů v jednom serveru 8 (6 uzlu je pro předmět NI-PDP a 2 uzly pro jiný předmět)

Server Supermicro FatTwin F618R3-FT

Motherboard Intel® Xeon® CPU E5-2630 v4 @ 2.20GHz

CPU Intel® Xeon® CPU E5-2630 v4 @ 2.20GHz

#CPU jader 20 (možno až 40 se zapnutým hyperthreadingem)

#sockets, #cores per socket, #threads per socket (2, 10, 1 (2))

CPU Cache L1 - L2 - L3 32KB - 256KB - 25600KB

GPU ASPEED AST2400 BMC

RAM Samsung M393A2G40DB1-CRC, 64GB RAM, 4x 16GB(2Rx4) PC4-21300R ECC registered (RDIMM, DDR4 2400MHz)

HDD WDC WD1003FBYZ, WD RE4 Raid edition, 2x 1TB, 7200, 64MB, SATA 6Gb/s

Datasety

Všechny datasety použité pro měření nejsou z poskytnuté sady vstupů dodávané k úloze, protože nebyly vhodné k měření. K měření měly být použity datasety, u kterých je problém vyřešen sekvenčním algoritmem za 2 - 6 minut. Sestrojil jsem si tedy své vlastní, které vyhovují této podmínce.

```
10
22
-V-----P--
--J-------
----P-----
--P-----P-
---P--P---
------P---
P-------P-
----P--P--
------P---
-P--------

Dataset 1
```

```
12
23
-V----------
--J----P----
P--------P--
---P--------
------------
----P--P----
P-------P---
-P----------
------P-----
---P------P-
------------
---P--------

Dataset 2
```

```
11
24
-V---------
--J----P---
P-----P--P-
---P-------
-----------
----P--P---
P----------
-P---------
-----------
---P-------
--------P--

Dataset 3
```

```
10
22
-V---P-P--
--J-------
-------P--
P---------
-----P--P-
----------
P---------
-P-------P
----------
---P---P--

Dataset 4
```

Kód 7: Datasety

Sekvenční řešení

Hodnoty dob výpočtu naměřené u sekvenčního řešení slouží jako referenční hodnoty pro další měření a výpočet zrychlení a efektivity na vlákno.

Dataset	Doba výpočtu [s]
1	212.742
2	304.302
3	320.11
4	253.398

Tabulka 1: Tabulka sekvenčního řešení

OpenMP task paralelismus

V následujících tabulkách (každá pro jeden dataset) jsou naměřené hodnoty při použití taskového paralelismu a pro různý počet vláken (1, 2, 4, 6, 8, 12, 16, 20). Zrychlení a efektivita je počítána vůči hodnotám naměřených u sekvenčního řešení. Pokud výpočet neskončil do 10 minut (byl „zastřelen“), je tento fakt indikován pomlčkou.

Počet vláken	Doba výpočtu [s]	Zrychlení	Efektivita na vlákno
1	181,05	1,18x	1,18
2	160,282	1,33x	0,66
4	120,132	1,77x	0,44
6	204,969	1,04x	0,17
8	168,597	1,26x	0,16
12	115,832	1,84x	0,15
16	135,014	1,58x	0,10
20	114,104	1,86x	0,9

Tabulka 2: Naměřené hodnoty pro dataset 1 u taskového paralelismu

Počet vláken	Doba výpočtu [s]	Zrychlení	Efektivita na vlákno
1	277,02	1,09x	1,09
2	389,65	0,78x	0,39
4	-	-	-
6	-	-	-
8	-	-	-
12	463,17	0,65x	0,05
16	61,61	4,91x	0,31
20	79,43	3,81x	0,19

Tabulka 3: Naměřené hodnoty pro dataset 2 u taskového paralelismu

Počet vláken	Doba výpočtu [s]	Zrychlení	Efektivita na vlákno
1	269,44	1,19x	1,19
2	421,09	0,76x	0,38
4	354,05	0,90x	0,23
6	180,20	1,78x	0,30
8	-	-	-
12	147,07	2,18x	0,18
16	86,74	3,69x	0,23
20	79,43	4,03x	0,20

Tabulka 4: Naměřené hodnoty pro dataset 3 u taskového paralelismu

Počet vláken	Doba výpočtu [s]	Zrychlení	Efektivita na vlákno
1	216,74	1,17x	1,17
2	53,28	4,76x	2,38
4	17,03	14,88x	3,72
6	58,56	4,33x	0,72
8	15,84	16x	2,00
12	52,03	4,87x	0,41
16	35,43	7,15x	0,45
20	60,73	4,17x	0,21

Tabulka 5: Naměřené hodnoty pro dataset 4 u taskového paralelismu

Jak je vidět, povětšinou je taskový paralelismus rychlejší volbou než samotné sekvenční řešení. V některých případech je ale dokonce i horší, než sekvenční řešení (nedoběhnutí v 10 minutovém limitu na Staru). U datasetů, kde došlo ke zrychlení lze pozorovat, že může být zbytečné využívat nadměrné množství vláken, neboť hodnota efektivity na jedno vlákno je poměrně nízká.

OpenMP data paralelismus

V následujících tabulkách (každá pro jeden dataset) jsou naměřené hodnoty při použití datového paralelismu a pro různý počet vláken (1, 2, 4, 6, 8, 12, 16, 20). Zrychlení a efektivita je počítána vůči hodnotám naměřených u sekvenčního řešení. Hloubka, do které se expandovaly stavy pomocí BFS před samotným vícevláknovým výpočtem, byla nastavena na hodnotu 2.

Počet vláken	Doba výpočtu [s]	Zrychlení	Efektivita na vlákno
1	176,11	1,21x	1,21
2	173,76	1,22x	0,61
4	165,02	1,29x	0,32
6	151,48	1,40x	0,23
8	134,71	1,58x	0,20
12	123,01	1,73x	0,14
16	117,76	1,81x	0,11
20	116,66	1,82x	0,09

Tabulka 6: Naměřené hodnoty pro dataset 1 u datového paralelismu

Počet vláken	Doba výpočtu [s]	Zrychlení	Efektivita na vlákno
1	138,67	2,18x	2,18
2	146,24	2,07x	1,03
4	154,23	1,96x	0,49
6	166,13	1,82x	0,30
8	175,28	1,72x	0,22
12	220,70	1,37x	0,11
16	229,72	1,32x	0,08
20	262,06	1,15x	0,06

Tabulka 7: Naměřené hodnoty pro dataset 2 u datového paralelismu

Počet vláken	Doba výpočtu [s]	Zrychlení	Efektivita na vlákno
1	163,94	1,95x	1,95
2	201,77	1,59x	0,79
4	71,29	4,49x	1,12
6	269,82	1,19x	0,20
8	66,07	4,84x	0,61
12	73,32	4,37x	0,36
16	80,19	3,99x	0,25
20	80,15	3,99x	0,20

Tabulka 8: Naměřené hodnoty pro dataset 3 u datového paralelismu

Počet vláken	Doba výpočtu [s]	Zrychlení	Efektivita na vlákno
1	194,32	1,30x	1,30
2	62,82	4,03x	2,02
4	64,60	3,92x	0,98
6	53,43	4,74x	0,79
8	80,66	3,14x	0,39
12	6,65	38,09x	3,17
16	53,32	4,75x	0,30
20	5,44	46,56x	2,33

Tabulka 9: Naměřené hodnoty pro dataset 4 u datového paralelismu

Obdobně jako u taskového paralelismu je zde vidět zrychlení oproti sekvenčnímu řešení. Ovšem zrychlení je zde celkem stabilní, nestalo se totiž, že by jakýkoliv běh (při libovolném počtu vláken) byl pomalejší než sekvenční řešení. U datasetu 1 je vidět zvětšující se zrychlení vzhledem k rostoucímu počtu vláken. Zároveň ale zrychlení není dostačující, aby neklesala efektivita na vlákno. U datasetu číslo 2 došlo k zajímavému faktu, že s rostoucím počtem vláken dokonce zrychlení klesá. Toto může být zapříčiněno nadměrnou režií OpenMP či architekturou klastru. Na datasetu 4 při počtu 12 a 20 vláken došlo k zajímavé situaci, kdy program doběhl ve velmi krátké době. Domnívám se, že se to stalo z toho důvodu, že OpenMP dělí paralelní cyklus na části různě, a stalo se , že jedno vlákno dostalo podproblém, který rychle vedl k dobrému výsledku a tudíž byl dosti ořezán stavový prostor.

MPI

Měření MPI verze probíhalo na 4 výpočetních uzlech (1x Master, 3x Slave). Zároveň na Slave uzlech je problém řešen datovým paralelismem, počítají tedy vícevláknově. Celkový počet vláken je tedy počet vláken na jednom Slave uzlu vynásobený počtem Slave uzlů. Pro expanzi stavů na Master uzlu byla použita hloubka 2 a expanze na před výpočtem na Slave uzlech hloubka 3 (tedy o jedno zanoření více než po expanzi na Master uzlu).

Počet vláken	Doba výpočtu [s]	Zrychlení	Efektivita na vlákno
18	43,35	4,91x	0,27
24	41,77	5,09x	0,21
36	41,81	5,09x	0,14
48	41,84	5,08x	0,11
60	41,83	5,09x	0,08

Tabulka 10: Naměřené hodnoty pro dataset 1 u MPI řešení

Počet vláken	Doba výpočtu [s]	Zrychlení	Efektivita na vlákno
18	52,43	5,77x	0,32
24	69,74	4,33x	0,18
36	45,23	6,68x	0,19
48	56,01	5,40x	0,11
60	55,03	5,49x	0,09

Tabulka 11: Naměřené hodnoty pro dataset 2 u MPI řešení

Počet vláken	Doba výpočtu [s]	Zrychlení	Efektivita na vlákno
18	19,44	16,47x	0,91
24	15,63	20,48x	0,85
36	13,78	23,23x	0,65
48	13,28	24,10x	0,50
60	12,73	25,15x	0,42

Tabulka 12: Naměřené hodnoty pro dataset 3 u MPI řešení

Počet vláken	Doba výpočtu [s]	Zrychlení	Efektivita na vlákno
18	55,69	4,55x	0,25
24	55,78	4,54x	0,19
36	33,51	7,56x	0,21
48	33,45	7,57x	0,16
60	33,40	7,59x	0,13

Tabulka 13: Naměřené hodnoty pro dataset 4 u MPI řešení

Z naměřených hodnot je vidět, že MPI řešení dává vcelku zajímavé výsledky. U datasetů došlo k velkému zrychlení. U datasetu 3 bylo zrychlení dost markantní a překvapivé. Je ale vidět, že distribuovaný přístup opravdu výpočet zrychlí. Vyšší počet vláken na jednotlivých Slave uzlech nehraje až tak podstatnou roli ve výpočetním čase. Spíše než zvyšování počtu vláken (od určitého počtu vláken začíná být zásadní režie) by bylo lepší přidat další Slave uzel.

Vyhodnocení

Řešení taskovým paralelismem bylo ve velké většině rychlejší než při sekvenčním řešení, ale nastaly i případy, že taskový paralelismus byl výrazně horší (nedoběhl vůbec v časovém limitu). Výpočetní časy nebyly vůbec stabilní při různém počtu vláken. Datový paralelismus už dosáhl větší stability, hodnoty dob výpočtu už tolik neoscilovaly a povětšinou větší počet vláken znamenal i rychlejší běh výpočtu. Od určité chvíle už bylo možné pozorovat režii OpenMP, kdy přidávání vláken už nebylo vůbec smysluplné. V jednom z datasetů se dokonce režie projevovala už od samého počátku a s rostoucím počtem vláken se výpočetní čas zvětšoval. U MPI řešení bylo zrychlení nejstabilnější a počet vláken neměl zas tak velký vliv na dobu výpočtu. Co mělo větší vliv, je počet výpočetních uzlů. Zvyšující počet Slave uzlů by měl implikovat větší zrychlení. Otázkou je poté, do jaké chvíle se přidávání nových uzlů vyplatí (cena stroje vůči potenciálnímu zrychlení).

Možné ovlivnění efektivity

Na efektivitu výpočtu má vliv spousta faktorů, mezi které patří:

Vstupní data různá vstupní data můžou sednout jen určitým přístupům řešení

Řazení a výběr souseda způsob jakým jsou vybíráni postupně sousedé při zanořování se do hloubky

Parametry expanze jak bude probíhat expanze před samotným výpočtem (do určité hloubky, určitý počet stavů, ...)

Hardware a jeho uspořádání na Star klastru

Způsob sdílení nejlepšího řešení u MPI verze nejlepší řešení posíláno s podproblémem nebo se Slave proces na nejlepší řešení neustále ptá Master procesu

Kód programu každý může naprogramovat program, který bude dělat to stejné, ale kód bude vypadat odlišně, například může používat jiné datové struktury.

7 Závěr

Cílem tohoto semestrálního projektu bylo vyzkoušet si naprogramovat řešení jednoho problému (Věž a jezdec na šachovnici) různými způsoby, od sekvenčního, přes taskový a datový paralelismus pomocí knihovny OpenMP, až po distribuované řešení pomocí knihovny OpenMPI. Jednotlivé verze poté otestovat a změřit na školním výpočetním klastru Star a interpretovat výsledky.

Cíl byl splněn, naimplementoval jsem všechny verze řešení a změřil jejich chod na výpočetním klastru Star. Z výsledků jsem mírně překvapen, neboť jsem čekal mnohem lepší výsledky, zejména jsem očekával, že zvyšování počtu vláken bude mít větší vliv a že režie nebude až tak velká. Od taskového paralelismu jsem čekal mnohem více, překvapilo mě, jak různé výsledky (doba výpočtu) byly. S datovým paralelismem jsem vcelku spokojen, ale zde hraje velkou roli, jak je paralelní cyklus rozdělen mezi vlákna. MPI řešení mě také překvapilo, ale vcelku pozitivně, jelikož se mi potvrdilo očekávání, že je lepší rozdělit práci mezi více výpočetních jednotek než zvyšovat počet vláken. Samotné výsledky můžou být ve velké míře ovlivněny velikou řadou faktorů a tak se můžou lišit od ostatních.

Práce na projektu byla pro mě přínosná, vyzkoušel jsem si, jak pracovat s knihovnami pro vícevláknové/-distribuované programy v C++ a dále také měřit výpočetní čas na výpočetním klastru. Co by se podle mého dalo na předmětu zlepšit, je třeba udělat nějaký nekompetentní žebříček časů na několika datasetech, člověk tak může mít třeba větší motivaci zrychlit svůj program (zároveň je ale fakt, že tento předmět je spíše o měření a porovnání vlastních výsledků, než o porovnávání s ostatními). Dále by nebylo špatné si zkusit celý projekt dělat skupinově, bylo by tak více času na měření, což by dávalo relevantnější výsledky.