Projekt: OpenMP + MPI

Táto dokumentácia opisuje prvý projekt z predmetu Paralelné programovanie. Zapodieva sa prácou s dvomi komunikačnými nástrojmi OpenMP a MPI, pomocou ktorých bola riešená ťažko výpočtová úloha - filtrovanie obrazu s použitím konvolučnej matice.

Dokumentácia obsahuje krátky popis riešenia a spôsob implementácie merajúcu rýchlosti výpočtov pomocou ktorých sa vyhodnocujú výsledky experimentov s rôznymi konfiguráciami paralelného systému. Výsledky sú zaznačované do tabuliek ktoré sa porovnávajú pomocou grafových diagramov. Na záver sa uvádza stručný komentár pri ktorom sa zhrnú výsledky jednotlivých konfigurácií.

Popis riešenia

Ako zdrojový obraz sa použil známy obrázok Lenny s veľkosťou 512x512 pixelov. Na filter obrazu sa pre výpočet konvolučnej matice použil kernel s veľkosťou 75x75 s hodnotami spôsobujúce embessový efekt.

Pred: Po:


MPI

Obrázok sa najprv načíta do buffera v hlavnom procese, následne sa rozpošle ostatným procesom pomocou **MPI_Bcast**. Na to aby sa medzi nimi rozdelili úlohy, každý proces si vypočíta svoju vlastnú časť obrázka v lokálnej premennej, ktorú bude spracovávať cez offset a počet pixelov vyjadrujúcich riadky.

```
63    // sending local data buffer to all processes of the mpi communicator
64    MPI_Bcast(locData, PSIZE*PSIZE*3, MPI_CHAR, 0, MPI_COMM_WORLD);
```

Po výpočte všetkých častí obrázkov sa následne spoja do jedného globálneho buffera pomocou **MPI_Gather**. Nakoniec sa tieto dáta uložia do výsledného obrázku ktorý sa vykreslí cez OpenGL.

```
116   /// grouping up local buffers into global data buffer
117   MPI_Gather(locData, partSize, MPI_CHAR, globData, partSize, MPI_CHAR, 0, MPI_COMM_WORLD);
```

OpenMP

Filtrovanie obrazu s použítím konvolučnej matice prebieha pomocou omp príkazu, prostredníctvom ktorého využijeme paralelizáciu pre každý proces zvlášť. Konvolúcia využíva zrkadlové spracovanie okrajov.

```
81      // computation of convolution
82      #pragma omp parallel for private(x,y,i,j,r,g,b) num_threads(TAMOUNT)
83      for(x = from; x < to; x++) {                        // x rows
84          for(y = 0; y < PSIZE; y++) {                    // y columns
85              for(i = 0; i < KSIZE; i++) {                // i kernel iteration for x (xK)
86                  for(j = 0; j < KSIZE; j++) {            // j kernel iteration for y (yK)
87                      int xK = x+i - (KSIZE / 2);
88                      int yK = y+j - (KSIZE / 2);
89                      if(xK < 0 || yK < 0 || xK >= PSIZE || yK >= PSIZE) {    // edge mirror handling
90                          if(xK < 0)      xK *= -1;
91                          if(yK < 0)      yK *= -1;
92                          if(xK >= PSIZE) xK -= 2 * (xK - PSIZE) + 1;
93                          if(yK >= PSIZE) yK -= 2 * (yK - PSIZE) + 1;
94                      }
95                      r += image[xK][yK].r * kernel[i][j];
96                      g += image[xK][yK].g * kernel[i][j];
97                      b += image[xK][yK].b * kernel[i][j];
98                  }
99              }
100             // saving calculated pixel into resulting image (each process do a portion)
101             r += bias;      imageNew[x][y].r = r;       r = 0;
102             g += bias;      imageNew[x][y].g = g;       g = 0;
103             b += bias;      imageNew[x][y].b = b;       b = 0;
104         }
105     }
```

Premenná x predstavuje pozíciu riadkov na obrázku. Každý proces mal vlastnú čiastku riadkov na spracovanie cez iterácie od premennej from do premennej to. Premenná y predstavovala stĺpce. Ďalšie dva for cykly boli určené pre kernelové iterácie po obrázku. Ich pozície sa vypočítavali do premenných xK a yK. Pred výpočtom sa kontroluje, či kernel neišiel mimo obrázka. Ak išiel, použije sa zrkadlové spracovanie.

Po prejdení všetkých kernelových iterácií pričítame do rgb hodnôt parameter bias ktorý je 128, po ňom tieto hodnoty uložíme do nového obrázka.

Popis použitých PC:

1. PC

- Procesor: Intel(R) Core(™) i7-3610QM CPU
- Frekvencia: 2.3GHz
- Počet jadier: 4

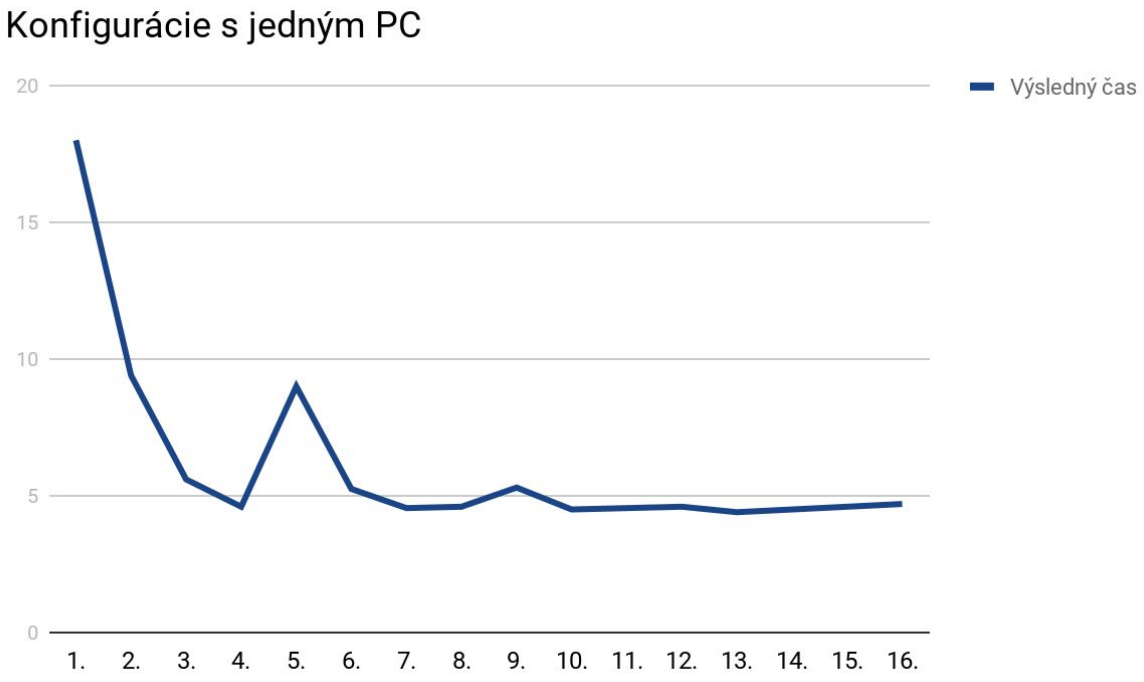
2. PC

- Procesor: Intel(R) Core(™) i7 CPU Q 720
- Frekvencia: 1.6GHz
- Počet jadier: 4

Konfigurácie a meranie ich výkonov:

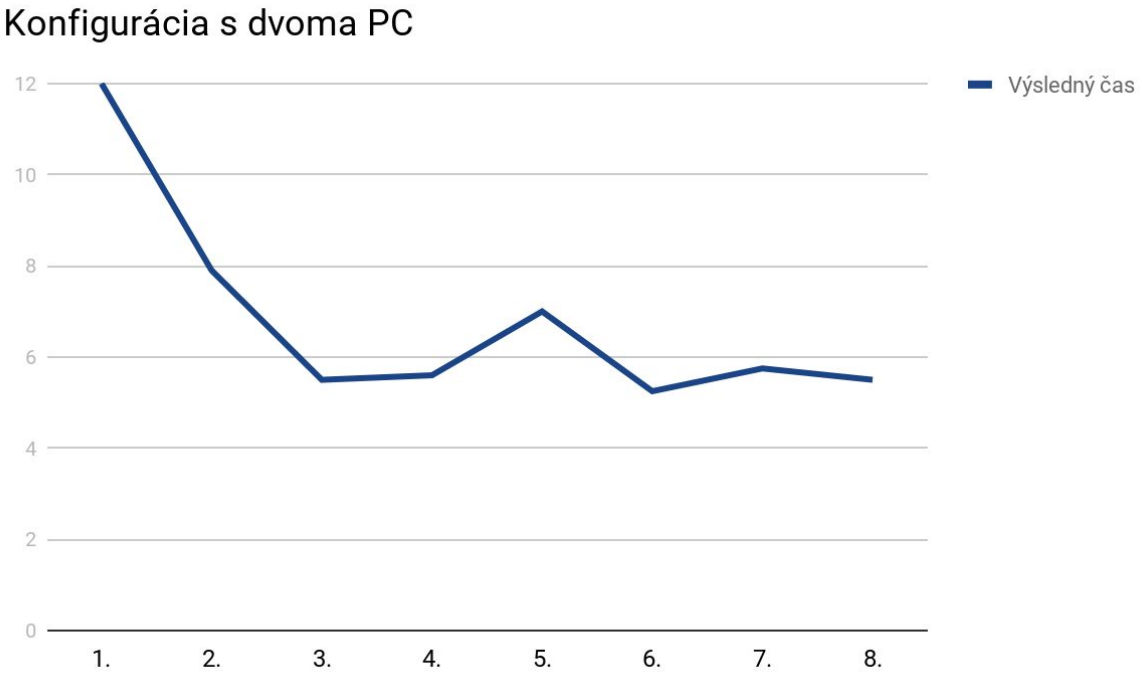
1. Konfigurácia s jedným PC

Číslo merania	Počet MPI procesov	Počet OpenMP vlákien	Výsledný čas v sekundách
1.	1	1	18
2.	1	2	9,4
3.	1	4	5,6
4.	1	8	4,6
5.	2	1	9
6.	2	2	5,25
7.	2	4	4,55
8.	2	8	4,6
9.	4	1	5,3
10.	4	2	4,5
11.	4	4	4,55
12.	4	8	4,6
13.	8	1	4,4
14.	8	2	4,5
15.	8	4	4,6
16.	8	8	4,7



2. Konfigurácia s dvoma PC

Číslo merania	Počet MPI procesov	Počet OpenMP vlákien	Výsledný čas v sekundách
1.	2	1	12
2.	2	2	7,9
3.	2	4	5,5
4.	2	8	5,6
5.	4	1	7
6.	4	2	5,25
7.	4	4	5,75
8.	4	8	5,5



Zhodnotenie

Program pre filtrovanie obrazu s použitím konvolučnej matice zaberie 18 sekúnd výpočtového času bez použitia OpenMP a MPI. Pri tabuľke si môžeme všimnúť, že zapojením MPI procesov vieme tento výpočtový čas výrazne zredukovať. Dopomôže nám k tomu aj OpenMP a to najviac vtedy, keď počet MPI procesov * počet OpenMP vlákien sa rovná 8 pretože keď sa použije menej ako 8 procesov, procesor nemusí ísť naplno, ak je procesov viac ako 8, pridá sa tam len navyše čas pre obsluhu synchronizácie medzi procesmi. Každopádne si taktiež môžeme všimnúť, že pridávaním počtu MPI procesov môžeme pre najlepšie výsledky (pod 4,7 sekúnd) aj postupne odoberať počet OpenMP procesov.

Pri zapojení dvoch PC sa najlepšie výsledky (pod 5,75 sekúnd) dosiahli vtedy keď pri jednom PC boli spustené aspoň 4 procesy (či už MPI alebo OpenMP).

Použitím paralelizácie cez MPI a OpenMP sa dosiahlo zrýchlenie výpočtu konvolučnej matice z 18 sekúnd na 4,4 sekundy. (zrýchlenie o 75%)