

Optimalizace a profilování

Tomáš Oberhuber

`tomas.oberhuber@fjfi.cvut.cz`

11. února 2024

Video na Youtube

Optimalizace

Vývoj paralelního algoritmu je nutné chápat jako **optimalizaci**.

1. nejprve vždy vyvíjíme co **nejjednodušší sekvenční algoritmus bez optimalizací**
 - ▶ **za každou cenu se vyhýbáme předčasné optimalizaci**
 - ▶ ta může zcela zbytečně poničit čistý návrh algoritmu
 - ▶ bez základní jednoduché verze kódu nemůžeme poměřit přínos optimalizace
2. máme-li základní funkční kód, který není dostatečně výkonný, přistupujeme k optimalizacím
 - ▶ pomocí profilování kódu určíme kritické části, kde se tráví nejvíce CPU času
 - ▶ následně většinou optimalizujeme sekvenční kód
 - ▶ pokud to nepostačuje, přikročíme k paralelizaci
3. během implementace optimalizací provádíme průběžné testy a kontrolujeme, zda optimalizovaný kód dává stále správné výsledky
4. nakonec poměříme přínos optimalizace/paralelizace tj. výslednou **efektivitu paralelizace**

Profilování kódu

- ▶ jde o proces, kdy se snažíme zjistit, kolik z celkového času běhu programu zaberou jednotlivé části kódu
- ▶ to lze provést buď přidáním měřicího kódu nebo pomocí tzv. profilerů
- ▶ čím přesnější měření chceme mít, tím více ovlivníme samotný výpočet

Profilery

Jsou dva základní přístupy:

- ▶ **statistické profilování** (sampling)
 - ▶ v pravidelných intervalech se sonduje, jaká část kódu se momentálně provádí
 - ▶ výsledek není přesný, obzvlášť krátké funkce nemusí být vůbec zachyceny
 - ▶ nezpomaluje příliš běh profilovaného programu
- ▶ **instrumentace** (*instrumentation*)
 - ▶ do kódu se přidají pomocné instrukce, kterými se sleduje zpracování kódu
 - ▶ ani krátké funkce se tak nepřehlédnou
 - ▶ běh programu se tím značně zpomalí
- ▶ **dynamická instrumentace**
 - ▶ instrumentaci neprovádí překladač, ale profiler těsně před spuštěním
 - ▶ profiler vlastně funguje jako virtuální stroj s JIT překladem
 - ▶ výpočet se hodně zpomalí, ale výsledek je podrobný

GNU gprof

<https://sourceware.org/binutils/docs/gprof/>

- ▶ tento profiler spolupracuje s překladačem gcc, který dělá instrumentaci
- ▶ program nejprve musíme přeložit s volbou `-pg` tj.
 - ▶ `g++ -O3 -pg -o program main.cpp`
 - ▶ provádíme-li linkování zvlášť, je potřeba přidat tento přepínač i linkeru
- ▶ následně program spustíme
- ▶ po ukončení běhu máme v pracovním adresáři soubor `gmon.out`
- ▶ ten slouží jako základ analýzy programem `gprof`

Gprof

- ▶ *flat profile*
 - ▶ `gprof -p program gmon.out`
 - ▶ u každé funkce ukáže, kolik času s v ní strávilo výpočtem
- ▶ *call graph*
 - ▶ `gprof -q program gmon.out`
 - ▶ vidíme tabulku, kde jsou jednotlivé položky oddělené řádkou pomlček
 - ▶ v každé položce je primární řádka ta, která obsahuje index v hranatých závorkách
 - ▶ udává primární funkci, kterou tato část popisuje
 - ▶ řádky nad primární řádkou popisují funkce, které danou funkci volají
 - ▶ položka `called` říká, kolikrát z celkového počtu proběhnutí daná funkce volala primární funkci
 - ▶ řádky pod primární řádkou popisují funkce volané danou funkcí
 - ▶ položka `called` říká, kolikrát z celkového počtu proběhnutí primární funkce volala danou funkci
- ▶ *source code annotations*
 - ▶ `g++ -O3 -g -pg -o program main.cpp`
 - ▶ `gprof -A program gmon.out`

GNU Gcov

- ▶ tento program umí napočítat, kolikrát byla daná řádka kódu provedena
- ▶ nejprve přeložíme náš program
 - ▶ `g++ -O0 -fprofile-arcs -ftest-coverage -o program main.cpp -lgcov`
- ▶ následně program spustíme
- ▶ v adresáři se zdrojovým kódem se objeví soubory s příponami `.gcda` `.gcno`
- ▶ použijeme příkaz
 - ▶ `gcov main.cpp`
- ▶ otevřeme si soubor `main.cpp.gcov`
- ▶ nebo použijeme program `lcov`
 - ▶ `lcov --capture --directory . --output-file coverage.info`
 - ▶ `genhtml coverage.info --output-directory coverage`
 - ▶ `cd coverage`
 - ▶ `firefox index.html`
- ▶ pozor, `lcov` udává pokrytí jednotlivých souborů, což neodpovídá času zpracování
- ▶ řádky s největším počtem provedení budou ale zřejmě odpovídat kritickým částem aplikace

Oprofile

Oprofile

<http://oprofile.sourceforge.net/about/>

- ▶ jde o statistický (samplovací) profiler
- ▶ není nutné dělat zvláštní překlad kódu
- ▶ zadáme příkazy
 - ▶ `perf program arguments`
 - ▶ `opreport -l`
- ▶ chceme graf volání funkcí, pak
 - ▶ `perf program arguments`
 - ▶ `opreport -l`
- ▶ chceme-li propojení se zdrojovým kódem
 - ▶ `g++ -O0 -g -o program main.cpp`
 - ▶ `perf program arguments`
 - ▶ `opannotate --source --search-dirs . --output-dir=annotated`

Callgrind

Callgrind

<http://valgrind.org/docs/manual/cl-manual.html>

- ▶ je součástí nástroje **Valgrind** – `valgrind.org`
- ▶ jde o skupinu dynamických instrumentačních nástrojů
- ▶ callgrind generuje velice přesný graf volání
- ▶ náš kód přeložíme s přepínačem `-g`
- ▶ a spustíme pomocí příkazu
 - ▶ `valgrind --tool=callgrind program argumenty`
- ▶ výpočet je výrazně pomalejší
- ▶ v průběhu výpočtu lze valgring sledovat pomocí
 - ▶ `callgrind_control -b -e`
 - ▶ `callgrind_control -s`
- ▶ po skončení běhu programu vznikne soubor `callgrind.out.<pid>`
- ▶ ten lze zpracovat pomocí příkazů
 - ▶ `callgrind_annotate callgrind.out.<pid>`
 - ▶ `kcachegrind callgrind.out.<pid>`

Intel Vtune a AMD μ Prof

Intel Vtune

<https://software.intel.com/content/www/us/en/develop/tools/oneapi/base-toolkit.html>

- ▶ jde o profiler s dynamickou instrumentací
- ▶ nejprve provedeme
 - ▶ `sudo echo 0 > /proc/sys/kernel/yama/ptrace_scope`
- ▶ profiler spustíme příkazem
 - ▶ `/opt/intel/vtune_amplifier_xe_2015/bin64/amplxe-gui`

AMD μ Prof

<https://developer.amd.com/amd-uprof/>

- ▶ profiler spustíme příkazem
 - ▶ `/opt/AMDuProf_3.3-462/bin/AMDuProf`

Cvičení:

- ▶ vyberte si některý svůj kód a proveďte na něm profilování pomocí zmíněných nástrojů
- ▶ najděte kritické části kódu, kde procesor tráví nejvíce času
- ▶ zamyslete se nad možnou optimalizací kódu
- ▶ než začnete jakékoliv optimalizace provádět, založte si gitovský repozitář pro možnost ukládání různých verzí kódu
 - ▶ <http://git-scm.com/>
 - ▶ <http://kmlinux.fjfi.cvut.cz/~zabkavit/git/>