# Definice problému

Vyhledávání podřetězce v textu (Substring matching). Cílem je nalezení veškerých výskytů hledaného řetězce v zadaném textu.

## Naivní algoritmus

Datově nezávislý, se zaručenou složitostí Θ(m\*n). Spočívá v průchodu veškerých možností vzájemných pozic řetězců a jejich následném porovnání.

Předčasné ukončení vnitřního cyklu může v ideálním případě urychlit výpočet až na Ω(n), horní složitost zůstává Θ(m\*n).

## Knuth - Morris - Pratt algoritmus

Algoritmus používá předzpracování vzorku pro urychlené vyhledávání. Dobu vyhledávání zkracuje pomocí posunu o více než jednu pozici v závislosti na struktuře dat. Algoritmus nejprve vyplní tabulku posunů a v druhé fázi pak spustí samotné vyhledávání.

V případě neshody znaku se podívá do tabulky a provede předpočítaný posun v závislosti na struktuře vzorku.

Pro výpočet tabulky potřebujeme Θ(m) operací a pro samotné vyhledávání Θ(n). Paměťová náročnosti je Θ(m) = Ω(m) pro tabulku posunů.

# Měření

Měření času bylo provedeno na serveru star.fit.cvut.cz, pomocí knihovní funkce omp\_get\_wtime. Každé měření probíhalo jednotným způsobem, po načtení vstupních hodnot programem byl změřen počáteční čas, spuštěn algoritmus výpočtu a po jeho dokončení spočítán rozdílný čas v sekundách. Přibližná hodnota MIPS byla měřena během zvláštního běhu programu, aby nebyl ovlivněn čas měření.

# Optimalizace

Protože oba algoritmy jsou silně datově závislé a oba mají různou složitost, měřil jsem jejich časy odděleně a na jiných instancích problému. Pro algoritmus KMP jsem používal přibližně 16x větší vstupní řetězce než pro naivní implementaci.

# Výsledky

Po zapnutí optimalizací o3 se průměrný čas běhu zkrátil přibližně na polovinu v důsledku optimalizací kompilátoru. Do srovnávacích cyklů jsem musel vložit nadbytečnou operaci, aby nedošlo k jejich odstranění v důsledku agresivní optimalizace.

Vylepšením, aplikovaným na oba algoritmy bylo rozdělení vstupního textu na části, které se vejdou do cache procesoru. Po změření jsem nezaznamenal větší zrychlení, protože už původní verze měla hodnotu cache-miss velice nízkou. A to i přes značně veliké vstupní řetězce, zatím pouze odhaduji, že vícevláknová verze bude na výpadky trpět více.

Dalším vylepšením použitým na oba přístupy byla změna datového typu řetězce z uint8\_t na uint64\_t. Po té průměrný čas klesl v závislosti na vstupu přibližně 6 – 8 krát pro oba algoritmy.

Na zrychlenou verzi naivního algoritmu jsem aplikoval ruční unrolling nejvnitřnějšího cyklu. Pro některý vstupy se čas téměř nezměnil, u jiných nastalo cca 30% zrychlení.

Mimo tyto úpravy jsem změnil vnitřní cyklus naivního postupu kvůli vektorizaci. Nevýhodou je nalezení všech rozdílů vůči textu místo prvního, kvůli nemožnosti větvení ve vektorizovaném cyklu. Tím se průměrná složitost dostala až na Ω(m^2), urychlení je dosaženo pouze díky vektorizaci, která násobně snížila počet průchodů cyklem. Protože ale došlo k logické změně algoritmu, nedá se výsledek porovnávat s ostatními.

Výsledky jsou vyneseny v přiložených grafech.

Navíc jsem vyhodnotil přibližné zkreslení doby výpočtu během výpočtu MIPS. Měřením došlo k průměrnému nárůstu času o 10 – 15%.

# Paralelní řešení

Pro paralelizaci jsem použil nejrychlejší sekvenční verzi programu s provedenými optimalizacemi. Mezi jednotlivá pracovní vlákna jsem cyklus rozdělil pomocí příkazy **omp parallel for**, se statickým rozdělením práce mezi všechny vlákna. Využívám toho, že jednotlivé výskyty ve vzorcích jsou rozděleny pravidelně. Proto pro můj případ nemá smysl používat dynamické přidělování iterací kvůli zvýšené ceně takového řešení. Každé pracovní vlákno načítá z prohledávaného řetězce určitou část práce, jejíž velikost je určena parametrem příkazové řádky, během měření vycházel pro nejlepší poměr zrychlení na procesor hodnota 16 384 bytů. Protože jsou oba algoritmy datově závislé, jsem měření provedl na více různých typových vstupech a jejich výsledky zanesl do grafů. Každé měření jsem postupně provedl na zvětšujícím se počtu procesorů a vyhodnotil efektivitu algoritmu. Protože velikost zrychlované a měřené části je stejná, mělo by být zrychlení podle amhdalova zákona lineární na počet vláken. Mé měření kopíruje ideální křivku pro nižší počty vláken, až do hodnoty 12. Pro vyšší hodnoty je zrychlení nižší. Myslím, že tento výsledek je způsoben tím, že mé řešení využije všechna fyzická jádra procesorů (12) a pro vyšší hodnoty se použije hyperthreading, který je vhodnější na úlohy, ve kterých se práce vláken často přerušuje. Protože je vyhledávání v řetězci výpočetně náročné a vytíží téměř po celou dobu všechny procesory, potenciál hyperthreadingu nelze využít.

# Závěr

Během tvorby semestrální práce jsem se podrobněji seznámil s architekturou cache pro architekturu CPU x86\_64. Otestoval jsem vliv optimalizací kompilátoru na rychlost práce programu a také jsem si vyzkoušel použít základní optimalizační postupy pro zvýšení rychlosti a efektivity programu. Celkové zrychlení od neoptimální verze až k vícevláknovému zpracování bylo více než stonásobné. Programování semestrální práce hodnotím jako velice zajímavé a užitečné.