VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

PDS Paralelní synchronní šifrování

PDS Parallel synchronous encryption

2015

Jiří Cága

Tuto stránku nahradíte v tištěné verzi práce oficiálním zadáním Vaší diplomové či bakalářské práce.

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 11. dubna 2015 .....

Abstrakt

Obsahem této práce je popsat algoritmus synchronního šifrování a převést kod pro paralelní zpracování na více vláknech. S cílem provést porovnání rychlosti při sériovém a paralélním zpracování.

Klíčová slova: Synchronní šifrování, paralélismus, měření

Abstract

The content this thesis is the description synchronous encryption algorithm and devolve this code to parallel process on many thread. The main goal is compare speed serially and parallel process.

Keywords: Synchronous encypher, parallel, measurement

Seznam použitých zkratek a symbolů

C++	–	Objektově orientovaný programovací jazyk nižší úrovně.
JAVA	–	Objektově orientovaný programovací jazyk nezávislý na počítačové platformě.
Proces	–	V informatice název pro spuštěný počítačový program..
Vlákno	–	Označuje v informatice odlehčený proces, pomocí něhož se snižuje režie operačního systému při změně kontextu
Paralelismus	–	zdánlivý nebo skutečný paralelní běh více procesů zároveň.
Bajt	–	Jednotka množství dat v informatice. Označuje osm bitů.
Nibble	–	Jednotka, jež se používá v informatice. Jeho velikost je 4 bity.

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

1.1 O předmetu

Přehledový předmět poskytuje studentům základní orientaci v problematice paralelních a distribuovaných systémů. Podává úvod k architekturám víceprocesorových systémů, jejich využití z pohledu programátora včetně obecné metodiky tvorby paralelních algoritmů a technických prostředků pro jejich realizaci. Dále je podán přehled architektur distribuovaných objektových systémů a předveden způsob implementace distribuovaných algoritmů. Zmiňují se aktuální trendy v předmětné oblasti.

1.2 Cíl práce

Cílem práce bylo naprogramovat algoritmus, který poté bude převeden pro paralelní zpracovaní na více vláknech. V této práci byl zvolen synchronní šifrovací algoritmus založený na blokové šifře. Algoritmus je všestranně použitelný pro šifrování různého typu dat, jelikož pracuje nad poli bajtů. Můžeme ho použít jak pro šifrování souborů na disku tak pro šifrování dat přenášených po počítačové síti, jelikož vždy šifruje pouze určitý blok bajtů.

1.3 Popis použitých technologií

Celý šifrovací nástroj byl původně naprogramován v programovacím jazyce JAVA[1], kvůli své přenositelnosti mezi různými platformami a jednoduchému použíti grafické knihovny Swing. Jelikož cíl bylo provést i paralelní zpracování zdálo se vhodnější přepsat kod do jazyka C++. Tento jazyk umožňuje větší možnosti optimalizace a řízení paralelního zpracování než programovací jazyk Java a taky nárůst výkonu. Programy psané v C++ jsou většinou rychlejší, díky tomu že se překládají přímo pro danou platformu a neběží ve virtuálním stroji jako v případě Javy. Pro paralelní zpracování se zvažovali dvě technologie a to OpenCL a OpenMP. OpenCL technologie nabízí podporu pro paralelní zpracování na grafických kartách tak procesorech a hlavně byla v době vývoje programu nainstalovaná s vývojovým prostředím, ale kvůli jednoduššímu převodu sériového kodu na paralelní byla zvolena technologie OpenMPI.

2 Technologie paralelního zpracování

2.1 Proč paralelismus

Pro mnohé moderní hardwarové architektury se paralelismus stává jedinou cestou k vyšším výkonům. Tváři v tvář fyzikálním limitům, nejčastěji v podobě teplotních omezení, se různé platformy vydaly jednotnou cestou navyšování počtů exekučních jednotek umožňujících paralelní zpracování místo navyšování frekvece procesoru. Zároveň s tím dochází k rychlému nárůstu výkonu a schopnosti i těch nejmenších mobilních zařízení. Množství nosných architektur pro náročné aplikace tak roste poměrně rychlým tempem. Potřeba snadného, efektivního a jednotného zacházení s takovým množstvím hardwarových platforem znamenala vysokou poptávku po nových softwarových pomocnících. Nabídka nástrojů, které jsou schopné jednoduše zprostředkovávat programátorům tento nový způsob myšlení a práce byla dlouho neuspokojivá. Dosavadní prostředí, která se tuto situaci pokoušejí řešit, trpí závažnými nedostatky. Mnohá proprietární řešení jsou totiž vázaná na konkrétní hardware (CUDA) či software (DirectCompute). Další nástroje sice umožňují jistý stupeň přenositelnosti avšak za cenu znatelně pomalejšího běhu aplikací na nich postavených. V dnešní době, ale už začínají být standardizované nástroje pro paralelní zpracovaní jako OpenCL, OpenMPI nebo Corba technologie využívaná spíš v podnikové sféře.

2.2 OpenCL

OpenCL (Open Computing Language) je průmyslový standard pro paralelní programování heterogenních počítačových systémů, jako jsou například osobní počítače vybavené GPU (grafikou), APU, případně DSP (audio). OpenCL definuje abstraktní hardwarové zařízení a k němu ovládací softwarové rozhraní, pomocí kterého aplikace přistupují ke konkrétním výpočetním možnostem různých hardwarových platforem. Jednoduchost modelu abstraktního zařízení usnadňuje jeho implementaci na široké škále existujících i plánovaných hardwarových platforem. Tyto platformy zahrnují klasické procesory (všechny CPU x86 s podporou instrukcí SSE3), grafické procesory (nVidia GeForce řady 8xxx a vyšší, ATI Radeon HD řady 4xxx a vyšší), signálové procesory (DSP), některé novější mobilní čipy, procesory typu Cell a další. Ani softwarové rozhraní standardu není závislé na softwarové platformě, což znamená, že nepotřebuje ke svému chodu žádný konkrétní operační systém. Hlavní výrobci grafických čipů již zahrnuli implementaci OpenCL do grafických ovladačů nabízených pro nejrozšířenější operační systémy jakými jsou různé verze Windows (XP, Vista, 7) či hlavní distribuce Linuxu (Ubuntu, OpenSUSE, Fedora). Podpora samozřejmě nechybí ani v produktech Apple (MacOS X 10.6, prostředí iPhone) a je dokonce dostupná i v některých virtuálních strojích (např. od VMware).[2]

2.3 OpenMP

OpenMP je soustava direktiv pro překladač a knihovních procedur pro paralelní programování. Jedná se o standard pro programování počítačů se sdílenou pamětí. OpenMP

usnadňuje vytváření vícevláknových programů v programovacích jazycích FORTRAN, C a C++. První OpenMP standard pro FORTRAN 1.0 byl publikován v roce 1997. Rok poté byl uvolněn standard pro C/C++. Standard verze 2.0 byl uvolněn pro FORTRAN v roce 2000 a pro C/C++ v roce 2002, verze 3.0 potom v roce 2008. Aktuální je verze 4.0, která byla jako kombinovaná pro jazyky C/C++/FORTRAN uvolněna v roce 2013. Programovací model technologie je následující: Hlavní vlákno (master thread) vytváří podle potřeby skupinu podvláken. Paralelizace programu se pak provádí postupně s ohledem na výkon aplikace, tj. sekvenční program je postupně (podle možností) pararelizován. OpenMP se spouští pomocí tzv. direktiv. K vytvoření skupiny vláken použijeme direktivu pragma.[3]

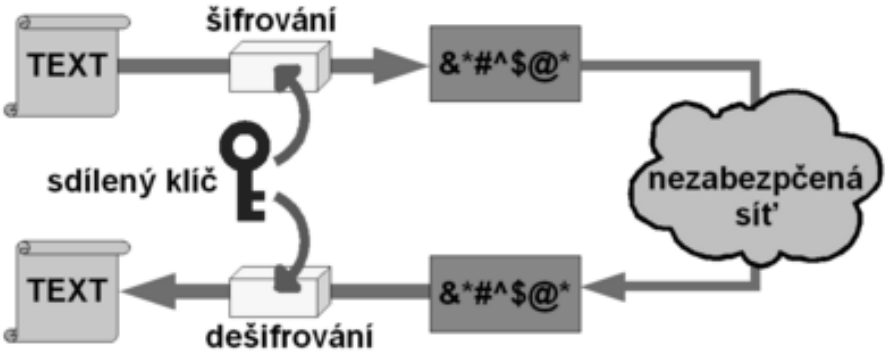
3 Šifrování

3.1 Obecné

Šifrování je věda zabývající se utajováním informace, tato věda se nazýva kryptografie, která je nauka využívající poznatků z matematiky ke kódování a dekódování dat tak, aby je nebylo možné číst bez znalosti určitého tajemství. Naproti tomu existuje kryptoanalýza, která se snaží získat data bez znalosti tohoto tajemství. Kryptoanalytici vlastně ověřují kvalitu výsledků kryptografie a pomáhají tak vytvářet opravdu bezpečné šifry, které se pak používají v praktickém životě. Šifrování dělíme do dvou skupin šifry synchronní a šifry asynchronní. V této kapitole se popisuje princip a dělení synchronní šifrování, které využívá algoritmus našeho šifrovacího nástroje.

3.2 Symetrické šifry

Symetrické šifrovací algoritmy jsou ty, které používají k zašifrování dat i jejich dešifrování stejný klíč (sdílené tajemství). Výhodou těchto algoritmů je jejich vysoká rychlost oproti algoritmům šifrujícím pomocí veřejného klíče. Velkou nevýhodou je však fakt, že komunikující strany se musí nejprve domluvit na sdíleném klíči a tento klíč nesmí být nikým odposlechnut. Díky tomu bývají v praxi méně používané, než asymetrické šifry. Často se však využívá kombinace obou druhů šifer: některá pro bezpečné předání klíče, jiná pro šifrování a jiná pro digitální podpis. Algoritmy symetrického šifrování provádí většinou jednodušší matematické operace typu bitový součet, negace, nonekvivalence atd. Díky tomu jsou jednak rychlé, ale hlavně je možné je jednoduše implementovat hardwarově. Na některých zařízeních proto bývá hardwarová podpora šifrování, ale využití nacházejí také na různých čipových kartách, čtečkách apod. Symetrické šifry se dělí do dvou základních skupin podle toho, jestli se data šifrují postupně bit po bitu, nebo jestli se šifruje ucelený blok dat. Prvním způsobem, tedy postupným zpracováním dat, pracují proudové šifry. Naproti tomu šifry blokové vezmou vždy pevný počet bitů a ten zašifrují, poslední blok se doplní náhodnými daty, které se po dešifrování zase zahodí.



Obrázek 1: Schéma symetrického šifrování

3.2.1 Blokové šifry

Blokové šifry pracují s pevným počtem bitů. Tyto bloky jsou pomocí klíče zašifrovány a výstupem je příslušná část šifrovaného textu. Některé algoritmy provádějí několikanásobné šifrování bloku dokola, aby se zvýšila bezpečnost. Blokové šifry jsou na šifrování dat v praxi využívány častěji než proudové. Mezi blokové šifry patří například algoritmy DES, AES, IDEA, Blowfish a jiné.

3.2.2 Proudové šifry

Proudové šifry jsou rychlejší než blokové. Vhodnější jsou tam, kde není možné využívat buffer. Typickým příkladem je telekomunikace a jiné real-time spojení, kde není vhodné čekat na naplnění bufferu, ale je třeba data šifrovat ihned. Mezi zástupce proudových šifer patří algoritmy RC4, FISH, Helix, SEAL, WAKE a další.

3.3 Asymetrické šifry

Asymetrické šifrovací algoritmy potřebují k zašifrování dat jiný klíč, než k jejich zpětnému dešifrování. Tato šifrovací metoda je novější než symetrické šifrování a v současnosti se také mnohem více používá. K šifrování je třeba dvojice klíčů: veřejný klíč a soukromý klíč. Veřejným klíčem se data zašifrují a dešifrovat je pak lze pouze klíčem soukromým. Odesílatel zašifruje zprávu veřejným klíčem příjemce a odešle ji. Pokud se zprávy zmocní někdo jiný, nemůže ji dešifrovat, protože nezná soukromý klíč příjemce. Jediný, kdo si zprávu může přečíst, je jen skutečný příjemce. Soukromý a veřejný klíč spolu samozřejmě musí matematicky souviset, ale je nesmírně důležité, aby z veřejného klíče nebylo možné zjistit klíč soukromý. Celá bezpečnost dat závisí na dobrém šifrovacím algoritmu a na dobrém klíči. Jednou z důležitých věcí je, aby oba klíče měli dostatečnou délku a jejich zjištění hrubou silou se tak pohybovalo v řádech desítek let nebo i víc.[4][5]

4 Popis algoritmu šifrování

4.1 Nadhled

Jak už bylo výše zmíněno algoritmus se zabývá symetrickým šifrováním. Vždy šifruje určitý blok bajtů pomocí klíče. Klíč je složen z 128 číslic, což umožňuje 128-bitové šifrování. Šifrování probíhá v několika fázích, první se blok šifruje substituční šifrou, poté se nad blokem bajtů provedou permutace a nakonec se provádí maticové operace. Při dešifrování se provádí ty samé operace akorát v opačném pořadí.

4.2 Struktura klíče

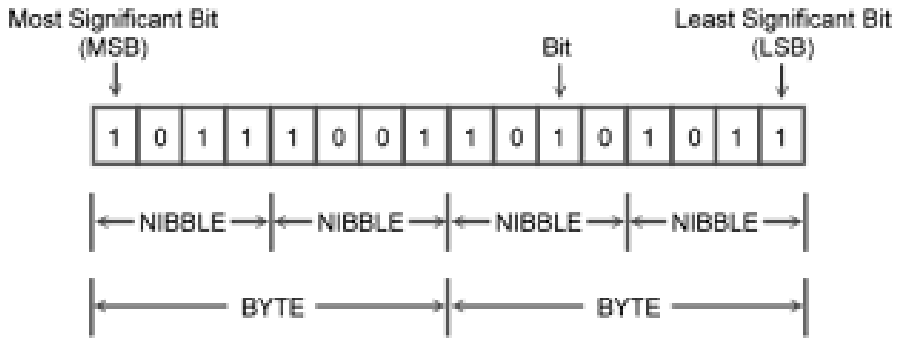
Rozsah hodnot pro klíč je 0 až 127 přičemž žádné číslo se nesmí v klíči opakovat. Klíč v programu může být buď načten z pole bajtů, v případě že uživatel chce dešifrovat soubor, nebo vygenerovaný náhodně. Při generování náhodného klíče se využívá datový kontejner z knihovny STL jazyka C++ a to Vektor. Vektor je šablona jednorozměrného pole, které se v průběhu běhu programu může zvětšovat. Což v našem případě není nutné, ale obsahuje metodu shuffle, jenž promíchá prvky v datovém kontejneru Vector. Jakmile máme prvky promíchané uložíme si je do klasického pole z důvodu rychlosti přístupu k prvkům v poli.

```
void CipherKey::generateKey(){
    /* initialize random seed: */
    srand (time(NULL));

    std::vector<int> myvector;

    // set some values:
    for (int i=0; i<128; ++i) myvector.push_back(i);

    // using myrandom:
    std::random_shuffle ( myvector.begin(), myvector.end());

    int position = 0;
    for (std::vector<int>:: iterator it =myvector.begin(); it !=myvector.end(); ++it)
        numbers[position++] = *it;


}
```

Výpis 1: Ukázka metody pro generování nového klíče

4.3 Pomocná třída BlockBytes

Pro jednodušší programování šifrovacích algoritmů byla vytvořena třída, která přistupuje k poli bajtů jako celku. Nad blokem bajtů lze provádět řadu operací jako nastavování jednotlivých bajtů, niblů či bitů. K realizaci této funkčnosti bylo použito binárních operátorů a bitových posuvů. Třída se vytvoří z pole bajtů a bere pole bajtů jako celistvý blok a k jednotlivým bitům blokům můžeme přistupovat pomocí indexů 0 až (počet bajtů * 8).



Obrázek 2: Blok vytvořený z dvou bajtů.

```
void BlockBytes::setHighNibble(int index,byte value){
    int nibble = ((int) (value))<<4; //posun zadanych 4 bitu na spravne misto
    int byteAsInteger = bytes[index];
    byteAsInteger=byteAsInteger&(INT_MAX&~(240)); //mazani stare hodnoty vrchnich 4 bitu na
        nulu
    byteAsInteger=byteAsInteger|nibble; //nastaveni nove hodnoty vrchnich 4 bitu (nibblu)
    bytes[index] = (byte) byteAsInteger;
}
```

Výpis 2: Ukázka metody nastavující horní nibl v bajtu bloku.

```
byte BlockBytes::getHighNibble(int index){
    int b = bytes[index];
    return (byte) (((b) >> 4) & 0x0F);
}
```

Výpis 3: Ukázka metody pro získání horního niblu bajtu z bloku.

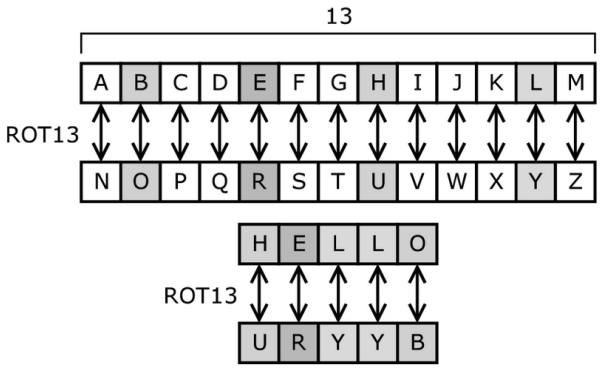
```
/*
     * Metoda zjisteni vnitrni pozice bitu
     * prvni urci index v poli, a pote index prislusneho bitu
     * napr: mame ulozene 2 bajty, chceme bit cislo 9
     * 9/8=1 .. takze budeme prohledavat druhy byte
     * 9&8=1 .. zbytek po deleni 1, coz je index v hledanem bytu
     */
    int indexOfArrayByte = index/8;
    int indexForBit = index%8;

    int byteAsInteger = bytes[indexOfArrayByte];

    /*
     * Bitove operace pro ulozeni bitu v bajtu
     */
    if (value == true) //nastavujeme jednicku na dane misto
    {
        /*
         * Ukazka:
         * Chceme ulozit 1 na 3 pozici v bajtu
         * bajt: [10000000]
         * maska: [00001000]
         * vysledek [10000000] |
         *          [00001000]
         *  =       [10001000]
         */
        int mask = 1<<indexForBit;
        byteAsInteger = byteAsInteger|mask;
    }
    else //nastavuje nulu na dane misto
    {
        /*
         * Ukazka:
         * Chceme ulozit 0 na 3 pozici v bajtu
         * bajt: [10001000]
         * maska: [11110111]
         * vysledek [10001000] &
         *          [11110111]
         *  =       [10000000]
         */
        int mask = ~(1<<indexForBit);
        byteAsInteger = byteAsInteger&mask;
    }

    bytes[indexOfArrayByte] = (byte) byteAsInteger;
}
```

Výpis 4: Ukázka metody pro nastavení bitu v bloku.

5 Druhy blokových šifer

5.1 Substituce

- **Popis:** Substituční šifra je v kryptografii druh šifry, při které dochází k záměně (substituci) nějaké množiny symbolů za jinou množinu symbolů. V našem případě dochází k substituci jednotlivých bajtů v bloku dle šifrovacího klíče.
- **Algoritmus:** Z klíče se vytvoří podklíč obsahující pouze 16 čísel s hodnotami 0 až 15. Tyto čísla jsou vybrána z klíče postupně a v pořadí uložení. Následuje vytvoření hash mapy obsahující čísla podkliče a ke každému se přiřadí pozice v podklíči. Hash mapa urychluje přístup k prvkům podklíče při šifrování. Poté se provádí substituce jednotlivých bajtů. Každý bajt je rozdělený na horní a dolní nible složený ze 4 bitů, což představuje právě čísla 0 až 15. První je vybrán spodní nible, vezme se jeho hodnota a získá se na základě jeho hodnoty substituční hodnota z hash mapy. To samé se provede s horním niblem.



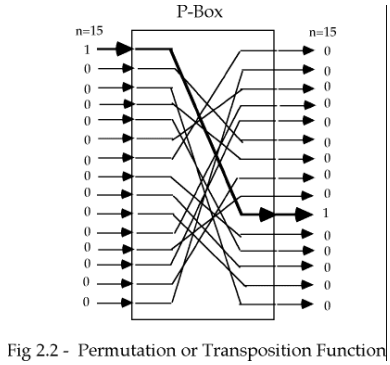
Obrázek 3: Ukázka substituční čifry nad blokem písmen

```
/**
 * @author cag0008
 * Prochazi postupne blok bytu, bere si jejich spodni a horni nibble
 * a ten nahrani za nibble ktery je v hash tabulce.
 */
class SubstitutionEncryptionStrategy:public EncryptionStrategy{

public:
    void execute(BlockBytes *block, CipherKey *key){
        int subkey[16]; //podklic obsahujici pouze cislo od 0-15 z klice

        //vytvoreni subklice
        int positionInSubkey = 0;
        for (int i=0;i<128;i++)
        {
            int number = key->getNumber(i);
            if (number>=0 && (number<16))
                subkey[positionInSubkey++] = number;
        }

        //vytvoreni hash mapy dle klice
        std::map<byte,byte> map;
        for (int i=0;i<16;i++)
        {
            map[(byte) i]= (byte) subkey[i];
        }

        //provadeni substituce jednotlivych niblu
        for (int i =0; i<block->getSize();i++)
        {
            byte lowNible = block->getLowNibble(i);
            byte ciherLowNible = map[lowNible];

            byte highNible = block->getHighNibble(i);
            byte ciherHighNible = map[highNible];

            block->setLowNibble(i, ciherLowNible);
            block->setHighNibble(i, ciherHighNible);
        }
    }
};
```

Výpis 5: Substituční šifrovací strategie.

5.2 Permutace

- **Popis:** Permutační šifra je v kryptografii druh šifry, při které dochází k přehazování pořadí prvků v množině. V našem případě bereme jako množinu blok bajtů a jako prvek jednotlivé bity bloku.
- **Algoritmus:** Postupně se prochází všechny bity v bloku bajtů o velikosti 128-bit, což odpovídá 16 bajtům. Podle indexu právě zpracovávaného bitu se algoritmus podívá na číslo v klíči odpovídající danému indexu. Následně dojde k prohození právě zpracovávaného bitu s bitem na pozici udávané klíčem.
- **Příklad:** (ukázka na bloku o 4 bitech)
  klíč: 4132
  blok: 1010->(přehodím 1 s 4 bitem) 0011->(přehodím 2 s 1 bitem)-> 0011
  -> (přehodím 3 s 3 bitem) -> 0011 ->(přehodím 4 s 2 bitem) 0110



Obrázek 4: Permutační funkce

```
class PermutationEncryptionStrategy: public EncryptionStrategy{
public: void execute(BlockBytes *block, CipherKey *key){
        //cyklus se provede pouze tolikrat kolik je v bloku podbloku obsahujici 128bitu
        for (int count=0;count<block->getSize()/16;count++)
        {
            int position = count*16; //pozice v bloku, urcujici podblok s kterym se ma ted
                pracovat
            for (int i=0;i<key->countNumbers;i++)
            {
                int indexFromKey = key->getNumber(i)+position;
                bool tmp = block->getBit(i+position);
                block->setBit(i+position, block->getBit(indexFromKey));
                block->setBit(indexFromKey, tmp);
            }
        }}};
```

Výpis 6: Permutační šifrovací strategie.

5.3 Maticové operace

- **Popis:** Maticové šifry jsou v kryptografii druhy šifer, které můžou provádět sloupcové, řádkové posuvy v matici či jiné maticové operace dle šifrovacího klíče. V našem případě bereme jako matici blok o velikosti 16 bajtů, což představuje matici 4x4.
- **Algoritmus:** Nad všemi sloupci matice jsou provedeny posuvy prvků ve sloupcí dolů o tolik kolik udává číslo v šifrovacím klíči.
- **Příklad:** Posun prvku v prvnim sloupci o 2
  123 => 423
  456 => 756
  789 => 189

```
class ColumnEncryptionStrategy: public EncryptionStrategy{

public:
    void execute(BlockBytes *block,CipherKey *key){
        for (int i=0;i<(block->getSize()/16);i++)
        {
            int matrixStart = 16*i;
            columnSwift(block, matrixStart, key->getNumber(0));
            columnSwift(block, matrixStart+1, key->getNumber(1));
            columnSwift(block, matrixStart+2,key->getNumber(2));
            columnSwift(block, matrixStart+3, key->getNumber(3));
        }
    }

private:
    void columnSwift(BlockBytes *block,int columnStart,int countSwift){
        for (int i=0;i<countSwift;i++)
        {
            byte firstByteInColumn = block->getByte(columnStart);
            block->setByte(columnStart,block->getByte(columnStart+4));
            block->setByte(columnStart+4,block->getByte(columnStart+8));
            block->setByte(columnStart+8,block->getByte(columnStart+12));
            block->setByte(columnStart+12,firstByteInColumn);
        }
    }
};
```

Výpis 7: Permutační šifrovací strategie.

6 Měření

6.1 Konfigurace počítače

Měření proběhlo na počítači s procesorem intel Core i5 obsahující 2 výpočetní jádra. Dále procesor obsahuje L2 vyrovnávací pamět o velikosti 256 kb pro každé vlákno a vyrovnávací paměť L3 společnou pro obě jádra s kapacitou 3 MB. Operační paměť počítače dosahovala kapacity 8 GB. Jedná se o dvě oddělené moduly DDR3 na frekvenci 1600 Mhz, kdy každý má kapacitu 4 GB. Pro ukládání dat obsahoval počítač SSD disk připojený na rozhraní PCI express.

6.2 Výsledky měření

Měření bylo prováděno nad blokem bajtů vytvořených v operační paměti, cílem měření bylo zjistit rozdíly v rychlosti šifrování a následného dešifrování při sériovém a paralélním zpracování.

Velikost vstupu	Single core	Dual core	Single core Java
10 MB	1.344s	0.948s	2.741s
30 MB	5.598s	4.218s	12.291s
50 MB	7.291s	5.244s	14.543s
100 MB	18.967s	13.754s	37.321s
250 MB	47.218s	33,234s	93.346s
500 MB	93.955s	68.667	185.762s

Tabulka 1: Tabulka měření komprese a dekomprese nad blokem dat.

6.3 Pozorování

Z výsledků měření je patrné, že paralelní zpracování ušetříme okolo 30 procent času. Předpoklad byl, že by se mohlo ušetřit až 50 procent času. Ale musí se počítat i práce pro rozdělení práce mezi dvě vlákna a zpětné složení výsledku, které zabere nějaký čas. Dále tabulka ukazuje velký rozdíl v délce výpočtu mezi jazyky C a Java. Důvod téměř dvojnásobné časové náročnosti je dán tím, že aplikace v programovacím jazyku Java běží nad virtuálním strojem, oproti jazyku C++, kde aplikace běží přímo na daném počítači.

7 Závěr

Cílem této práce bylo pochopení principů paralélního zpracování dat na počítači, jenž v dnešní době má stále větší význam. Firmy vyrábějící procesory už nezvyšují taktovací frekvenci procesoru, ale přidávají do procesoru další jádra. Poté se jeden procesor tváři jako by obsahoval další procesory.Tento trend je nejenom u klasických stolních počítačů, ale taky mobilních telefonů a tabletů, kde nejvýkonnější z nich mají i čtyřjádrové procesory. Naštěstí existuje celá řada hotových frameworků usnadňující programátorovi vývoj, jako například OpenMPI, OpenCL či Corba využívaná spíš pro bssnises aplikace. V práci byla použita technologie OpenMP, která má několik výhod. Jedna z nich je, že jednoduše můžeme převést sériový program na paralelním přidáním direktiv v jazyku C. Druhá výhoda OpenMP je, že pokud překladač nepodporuje OpenMPI, přeloží program jako by se jednalo o klasický program bez paralelního rozšíření.

8 Reference

[1] Pecinovský Rudolf. *Java 7 učebnice objektové architektury pro začátečníky*. České Budějovice: Grada Publishing, a.s., 2012. ISBN 9788025137482.

[2] Developer AMD. Intro OpenCL^{TM} Tutorial. [online]. 2014 [cit. 2015-04-12]. Dostupné z: http://developer.amd.com/tools-and-sdks/opencl-zone/opencl-resources/introductory-tutorial-to-opencl/

[3] Guide into OpenMP: Easy multithreading programming for C++. [online]. 2008 [cit. 2015-04-12]. Dostupné z: http://bisqwit.iki.fi/story/howto/openmp/

[4] Pavel Vondruška. Kryptologie, šifrování a tajná písma. : Albatros, 25.10.2006. ISBN 9788000018881.

[5] Ondřej Bitto . Šifrování a biometrika. : Computer Media, 25.10.2006. ISBN 80-86686-48-5.