FITCoin & ČVUTCoin

Termín odevzdání:	**15.04.2018 23:59:59**	3122824.807 sec
Hodnocení:	**0.0000**	
Max. hodnocení:	**30.0000** (bez bonusů)	
Odevzdaná řešení:	0 / 60	
Nápovědy:	0 / 0	

Úkolem je realizovat třídu, která bude umožňovat rychle těžit kryptoměnu.

Boom kryptoměn zaznamenalo i ČVUT a naše fakulta a probíhá příprava na ICO této virtuální měny. Pro zajištění transakcí se používají výpočetně náročné operace, které zároveň umožňují takovou měnu těžit. Vaším úkolem je implementovat program, který dokáže tuto těžbu pomocí vláken rozkládat na všechny dostupné procesory.

Protože nedošlo k dohodě mezi ČVUT a FIT, má plánovaná měna dva různé algoritmy těžení. Vznikají tak různé varianty měny, které budeme pracovně nazývat fitCoin a čvutCoin.

fitCoin
: dostane na vstupu pole 32 bitových neznaménkových celých čísel - bitových vektorů. Dále má parametrem zadanou vzdálenost `dist`. Algoritmus těžení hledá, kolik existuje různých 32 bitových čísel `x` takových, že se liší od každého ze zadaných vektorů nejvýše v `dist` bitech. Výpočet je demonstrován na příkladu, pouze pro přehlednost uvažujeme pouze 4 bitová čísla. Pokud budou zadané vektory:

```
0011
0101
0110
```

Pak dostaneme pro čísla `x` následující počty odlišných bitů:

```
        ≤0     ≤1     ≤2     ≤3     ≤4
0000                 *      *      *
0001                        *      *
0010                        *      *
0011                 *      *      *
0100                        *      *
0101                 *      *      *
0110                 *      *      *
0111                 *      *      *
1000                        *      *
1001                               *
1010                               *
1011                        *      *
1100                               *
1101                        *      *
1110                        *      *
1111                 *      *      *
Celkem:  0      0      6     13     16
```

Bude-li tedy vstupem algoritmu předložená sada vektorů a `dist` nastaveno na 3, je výsledkem výpočtu číslo 13 (platí pro ukázku se 4-bitovými vektory).

čvutCoin
: dostane na vstupu pole bajtů. Algoritmus výpočtu toto pole chápe jako posloupnost bitů počínaje LSB (nejméně významným bitem) prvního bajtu a konče MSB (nejvíce významným bitem) posledního bajtu. Pro vstup délky `n` bajtů tedy máme posloupnost délky `8n` bitů. Dále jsou parametrem dvě celá čísla `distMin` a `distMax`. Úkolem je vyzkoušet všechny neprázdné předpony (prefixy) této bitové posloupnosti a všechny neprázdné suffixy této posloupnosti, pro každou dvojici (prefix x suffix) je potřeba určit jejich editační vzdálenost. Výsledkem je počet dvojic, jejichž editační vzdálenost patří do zadaného uzavřeného intervalu `< minDist ; maxDist >`. Editační vzdálenost dvojice bitových řetězců chápeme jako nejmenší počet operací mazání/vkládání/změny bitu tak, aby z jednoho bitového řetězce vznikl druhý.

Vaším úkolem je realizovat třídu `CRig`, která dokáže takové problémy řešit. Oba uvedené problémy jsou výpočetně náročnější a oba problémy je potřeba řešit rychle. Proto bude využito vláken k rozdělení výpočetní zátěže na více CPU a asynchronního modelu výpočtu.

Třída `CRig` má modelovat opakované výpočty obou výše popsaných problémů v těžebním poolu. Problémy zadávají zákazníci (instance třídy `CCustomer`, vytvořené testovacím prostředím a předané `CRig`). Zákazníci předávají problémy (instance tříd `CFITCoin` a `CCVUTCoin`), Vaše implementace `CRig` si instance problému převezme, zpracuje je a vyřešené je zadávajícímu vrátí.

Vaše implementace si vytvoří pracovní vlákna, jejich počet je předán při spouštění výpočtu. Dále, pro načítání problémů si vytvořte tři pomocná vlákna pro každého zákazníka. Tato vlákna budou volat odpovídající metody instance `CCustomer`, jedno vlákno bude volat funkci pro

doručování problémů typu fitCoin, druhé pro doručování problémů čvutCoin a třetí vlákno bude doručovat zpět oba druhy vyřešených problémů. Zadaný problém má podobu instance třídy `CFITCoin` nebo třídy `CCVUTCoin` (podle typu úlohy, instance jsou předané jako smart pointery - `shared_ptr<CFITCoin>` a `shared_ptr<CCVUTCoin>`, pro zkrácení zápisu jsou pro smart pointery vytvořené aliasy `AFITCoin` a `ACVUTCoin`). Vlákna, která přebírají zadávané problémy, nejsou určena k tomu, aby počítala řešení, jejich úkolem je pouze předání problémů dále k pracovním vláknům.

Pracovních vláken vytvoříte více (podle parametrů při inicializaci). Pracovní vlákna vyřeší zadanou instanci problémů a podle výsledků vyplní příslušné složky instance `AFITCoin` / `ACVUTCoin`. Po vyplnění předají instanci vyřešeného problému vyčleněnému předávacímu vláknu daného zákazníka. Předávací vlákno zajistí zavolání příslušné metody rozhraní zákazníka a zajistí serializaci odevzdávaní vyřešených úloh.

Jak již bylo řečeno, načítací a odevzdávací vlákna slouží pouze k odebírání požadavků od zákazníků a k předávání těchto požadavků pracovním vláknům. Celkový počet načítacích vláken bude dvojnásobkem počtu zákazníků, celkový počet odevzdávacích vláken bude roven počtu zákazníků. Pokud by načítací/odevzdávací vlákna rovnou řešila zadávané problémy a zákazníků bylo mnoho, vedlo by takové řešení k neefektivnímu využívání CPU (mnoho rozpracovaných problémů, časté přepínání kontextu, ...). Proto požadované řešení ponechává výpočty pouze na pracovních vláknech, kterých je pouze zadaný fixní počet.

Rozhraním Vaší implementace bude třída `CRig`. V této třídě musí být k dispozici metody podle popisu níže (mohou existovat i další privátní metody potřebné pro Vaši implementaci):

`implicitní konstruktor`
: inicializuje instanci třídy.

`destruktor`
: uvolní prostředky alokované instancí `CRig`.

`Start (thr)`
: metoda spustí vlastní výpočet. V této metodě vytvoříte potřebná pracovní vlákna pro výpočty. Pracovních vláken vytvořte celkem `thr` podle hodnoty parametru. Tím se spustí obsluha požadavků od zákazníků. Metoda `Start` se po spuštění pracovních vláken okamžitě vrací (tedy nečeká na doběhnutí výpočtů).

`Stop ()`
: metoda informuje, že se mají ukončit výpočty. Tedy je potřeba převzít zbývající požadavky od existujících zákazníků, počkat na jejich vypočtení a vrácení výsledků. Metoda `Stop` se vrátí volajícímu po doběhnutí a uvolnění jak načítacích, předávacích, tak pracovních vláken. Metoda `Stop` se musí vrátit do volajícího. Neukončujte celý program (nevolejte `exit` a podobné funkce), pokud ukončíte celý program, budete hodnoceni 0 body.

`AddCustomer ( c )`
: metoda přidá dalšího zákazníka do seznamu zákazníků obsluhovaných touto instancí. Parametrem je smart pointer (`shared_ptr<CCustomer>`, zkráceně `ACustomer`) začleňovaného zákazníka. Metoda `AddCustomer` musí mj. vytvořit dvě pomocná načítací vlákna, která budou tohoto nového zákazníka obsluhovat a předávací vlákno pro odevzdávání vyřešených zadání. Pozor: metodu lze zavolat ještě před voláním `Start` (tedy zákazníci jsou obsluhovaní, ale výpočetní vlákna ještě neexistují), tak i po spuštění `Start` (nový zákazník je přidán k existujícím a je zahájena jeho obsluha).

`Solve (fitCoin)`
: - metoda vypočte sekvenčně jeden zadaný problém typu `AFITCoin` (parametr). Testovací prostředí nejprve zkouší sekvenční řešení, abyste případně snáze odhalili chyby v implementaci algoritmu.

`Solve (cvutCoin)`
: - metoda vypočte sekvenčně jeden zadaný problém typu `ACVUTCoin` (parametr). Testovací prostředí nejprve zkouší sekvenční řešení, abyste případně snáze odhalili chyby v implementaci algoritmu.

Třída `CCustomer` definuje rozhraní jednoho zákazníka. Zákazník je implementován v testovacím prostředí a je předán Vaší implementaci v podobě smart pointeru (`shared_ptr<CCustomer>` alias `ACustomer`). Rozhraní `CCustomer` má následující metody:

destruktor
: uvolňuje prostředky alokované pro zákazníka,

`FITCoinGen() / CVUTCoinGen()`
: metoda po zavolání vrací další instanci problému fitCoin/čvutCoin ke zpracování. Návratovou hodnotou je smart pointer (`shared_ptr<CFITCoin>` alias `AFITCoin` případně `shared_ptr<CCVUTCoin>` alias `ACVUTCoin`) s popisem problému k vyřešení. Pokud je vrácen prázdný smart pointer (obsahuje NULL), znamená to, že daný zákazník již nemá žádný další problém tohoto typu k vyřešení (ale stále může dodávat problémy typu druhého typu). Pokud metoda vrátí prázdný ukazatel, lze ukončit příslušné načítací vlákno.

`FITCoinAccept ( fitCoin ) / CVUTCoinAccept (cvutCoin)`
: metodou se předá vyřešený problém typu fitCoin/čvutCoin zpět zákazníkovi. Je potřeba vrátit vyřešený problém tomu zákazníkovi, který problém zadal. Dále, je potřeba vrátit tu samou instanci problému, kterou dříve předala metoda `FITCoinGen / CVUTCoinGen`, pouze je v ní potřeba vyplnit vypočítané hodnoty. Metody `FITCoinAccept / CVUTCoinAccept` je potřeba volat serializovaně z odevzdávacího vlákna, toto vlákno musí být pro daného zákazníka stále stejné.

Třída `CFITCoin` je deklarovaná a implementovaná v testovacím prostředí. Pro testování Vaší implementace je k dispozici v bloku podmíněného překladu (ponechte jej tak). Význam složek je následující:

`m_Vectors`
: seznam 32 bitových vektorů zadání úlohy. Tato hodnota je vyplněna při vytváření zadání.

`m_DistMax`
: maximální vzdálenost od zadaných bitových vektorů. Tato hodnota je vyplněna při vytváření zadání.

`m_Count`
: je výsledkem výpočtu - počet 32 bitových hodnot `x` takových, že jejich vzdálenost od zadaných vektorů je nejvýše `m_DistMax`. Tuto hodnotu má vyplnit pracovní vlákno.

`implicitní konstruktor`
: existuje pro usnadnění vytváření instance problému.

Třída `CCVUTCoin` je deklarovaná a implementovaná v testovacím prostředí. Pro testování Vaší implementace je k dispozici v bloku podmíněného překladu (ponechte jej tak). Význam složek je následující:

`m_Data`
: bajty tvořící bitovou posloupnost k analýze (bity čteme směrem od nejnižšího k nejvyššímu). Tato hodnota je vyplněna při vytváření zadání.

`m_DistMin, m_DistMax`
: Rozmezí hodnot odlišnosti bitových řetězců při porovnávání. Tyto hodnoty jsou vyplněné při vytváření zadání.

`m_Count`
: je výsledek výpočtu, který má vyplnit pracovní vlákno. Hodnota má udávat počet dvojic (předpona, přípona) zadané bitové posloupnosti takových, že jejich editační vzdálenost patří do intervalu hodnot `< m_DistMin ; m_DistMax >`.

`implicitní konstruktor`
: existuje pro usnadnění vytváření instance problému.

Odevzdávejte zdrojový kód s implementací požadované třídy `CRig` s požadovanými metodami. Můžete samozřejmě přidat i další podpůrné třídy a funkce. Do Vaší implementace nevkládejte funkci `main` ani direktivy pro vkládání hlavičkových souborů. Funkci `main` a hlavičkové soubory lze ponechat pouze v případě, že jsou zabalené v bloku podmíněného překladu.

Využijte přiložený ukázkový soubor. Celá implementace patří do souboru `solution.cpp`, dodaný soubor je pouze mustr. Pokud zachováte bloky podmíněného překladu, můžete soubor `solution.cpp` odevzdávat jako řešení úlohy.

Při řešení lze využít pthread nebo C++11 API pro práci s vlákny (viz vložené hlavičkové soubory). Dostupný kompilátor g++ verze 4.9, tato verze kompilátoru zvládá většinu C++11 konstrukcí.

Nápověda:

- Nejprve implementujte sekvenční funkce řešení problémů fitCoin/čvutCoin. Správnost implementace lze ověřit lokálně pomocí infrastruktury v přiloženém archivu. Až budete mít funkce lokálně otestované, můžete je zkusit odevzdat na Progtest (pro tento pokus nechte ostatní metody třídy `CRig` s prázdným tělem). Takové řešení samozřejmě nedostane žádné body, ale uvidíte, zda správně projde sekvenčními testy.
- Abyste zapojili co nejvíce jader, zpracovávejte několik problémů najednou. Vyzvedněte je pomocí opakovaného volání `FITCoinGen/CVUTCoinGen` jednotlivých zákazníků, zajistěte si odevzdávací vlákno a zprovozněte komunikaci mezi přebíracími/pracovními a odevzdávacími vlákny. Není potřeba dodržovat pořadí při vracení řešení. Pokud budete najednou zpracovávat pouze jeden problém, nejspíše zaměstnáte pouze jedno vlákno a ostatní vlákna budou čekat bez užitku.
- Instance `CRig` je vytvářená opakovaně, pro různé vstupy. Nespoléhejte se na inicializaci globálních proměnných - při druhém a dalším zavolání budou mít globální proměnné hodnotu jinou. Je rozumné případné globální proměnné vždy inicializovat v konstruktoru nebo na začátku metody `Start`. Ještě lepší je nepoužívat globální proměnné vůbec.
- Nepoužívejte mutexy a podmíněné proměnné inicializované pomocí `PTHREAD_MUTEX_INITIALIZER`, důvod je stejný jako v minulém odstavci. Použijte raději `pthread_mutex_init()` nebo C++11 API.
- Testovací prostředí samo o sobě nevytváří žádná vlákna, tedy metoda `Start` sama o sobě nemusí být reentrantní (může používat globální proměnné, s omezením výše).
- Instance tříd CFITCoin / CCVUTCoin alokovalo testovací prostředí při vytváření příslušných smart pointerů. K uvolnění dojde automaticky po zrušení všech odkazů. Uvolnění těchto instancí tedy není Vaší starostí, stačí zrušit všechny odkazy na takto předané smart pointery. Váš program je ale zodpovědný za uvolnění všech ostatních prostředků, které si alokoval.
- Problémy musíte načítat, zpracovávat a odevzdávat průběžně. Postup, kdy si všechny problémy načtete do paměťových struktur a teprve pak je začnete zpracovávat, nebude fungovat. Takové řešení skončí deadlockem v prvním testu s více vlákny. Musíte zároveň obsluhovat požadavky typu fitCoin i čvutCoin. Řešení, které se bude snažit nejprve vyřešit všechny problémy jednoho typu a pak začne obsluhovat problémy druhého typu, skončí taktéž deadlockem.
- Musíte najednou obsluhovat více přidaných zákazníků. Pokud se budete snažit nejprve obsloužit zákazníka A, následně pouze zákazníka B, ..., skončíte taktéž v deadlocku.
- Volání metod `FITCoinAccept/CVUTCoinAccept` není reentrantní. Je potřeba volání obsluhovat pouze z jediného vlákna, toto vlákno musí být vytvořené zvlášť pro každého zákazníka.
- Neukončujte metodu `Stop` pomocí `exit`, `pthread_exit` a podobných funkcí. Pokud se funkce `Stop` nevrátí do volajícího, bude Vaše implementace vyhodnocena jako nesprávná.
- Využijte přiložená vzorová data. V archivu jednak naleznete ukázku volání rozhraní a dále několik testovacích vstupů a odpovídajících výsledků.
- V testovacím prostředí je k dispozici STL. Myslete ale na to, že ten samý STL kontejner nelze najednou zpřístupnit z více vláken. Více si o omezeních přečtěte např. na **C++ reference - thread safety.**
- Testovací prostředí je omezené velikostí paměti. Není uplatňován žádný explicitní limit, ale VM, ve které testy běží, je omezena 4 GiB celkové dostupné RAM. Úloha může být dost paměťově náročná, zejména pokud se rozhodnete pro jemné členění úlohy na jednotlivá vlákna. Pokud se rozhodnete pro takové jemné rozčlenění úlohy, možná budete muset přidat synchronizaci běhu vláken tak, aby celková potřebná paměť v žádný okamžik nepřesáhla rozumný limit. Pro běh máte garantováno, že Váš program má k dispozici nejméně 500 MiB pro Vaše data (data segment + stack + heap). Pro zvídavé - zbytek do 4GiB je zabraný běžícím OS, dalšími procesy, zásobníky Vašich vláken a nějakou rezervou.
- Pokud se rozhodnete pro všechny bonusy, je potřeba velmi pečlivě nastavovat granularitu řešeného problému. Pokud řešený problém rozdělíte na příliš mnoho drobných podproblémů, začne se příliš mnoho uplatňovat režie. Dále, pokud máte najednou rozpracováno příliš mnoho problémů (a každý je rozdělen na velké množství podproblémů), začne se výpočet dále zpomalovat (mj. se začnou hůře využívat cache CPU). Aby se tomu zabránilo, řídí referenční řešení počet najednou rozpracovaných úloh (navíc dynamicky podle velikosti rozpracované úlohy).

Co znamenají jednotlivé testy:

Test algoritmu (sekvencni)

Testovací prostředí opakovaně volá metody `Solve(fitCoin)/Solve(cvutCoin)` pro různé vstupy a kontroluje vypočtené výsledky. Slouží pro otestování implementace Vašeho algoritmu. Není vytvářena instance `CRig` a není volaná metoda `Start`. Na tomto testu můžete ověřit, zda Vaše implementace algoritmu je dostatečně rychlá. Testují se náhodně generované problémy, nejedná se o data z dodané ukázky.

Základní test/test několika/test mnoha thready
Testovací prostředí vytváří instanci `CRig` pro různý počet vláken a zákazníků.

Test zahlcení
Testovací prostředí generuje velké množství požadavků a kontroluje, zda si s tím Vaše implementace poradí. Pokud nebudete rozumně řídit počet rozpracovaných požadavků, překročíte paměťový limit.

Test zrychleni vypoctu
Testovací prostředí spouští Vaši implementaci pro ta samá vstupní data s různým počtem vláken. Měří se čas běhu (wall i CPU). S rostoucím počtem vláken by měl wall time klesat, CPU time mírně růst (vlákna mají možnost běžet na dalších CPU). Pokud wall time neklesne, nebo klesne málo (např. pro 2 vlákna by měl ideálně klesnout na 0.5, existuje určitá rezerva), test není splněn.

Busy waiting - pomale pozadavky
Do volání `FITCoinGen/CVUTCoinGen` testovací prostředí vkládá uspávání vlákna (např. na 100 ms). Výpočetní vlákna tím nemají práci. Pokud výpočetní vlákna nejsou synchronizovaná blokujícím způsobem, výrazně vzroste CPU time a test selže.

Busy waiting - pomale odevzdani
Do volání `FITCoinAccept/CVUTCoinAccept` je vložena pauza. Pokud jsou špatně blokována vlákna načítající vstup, výrazně vzroste CPU time. (Tento scénář je zde méně pravděpodobný.)

Busy waiting - complex
Je kombinací dvou posledně jmenovaných testů.

Velmi mnoho zákazníků
Testovací prostředí zkouší přidávat mnoho zákazníků, každý zákazník má pouze několik málo požadavků a skončí. Po skončení zákazníka je potřeba průběžně ukončovat a uvolňovat načítací vlákna. Pokud se uvolnění načítacích vláken neděje průběžně, enormně vzroste paměťová náročnost a program spadne. Vzhledem k použitému HW je rozumné najednou obsluhovat přibližně 5-10 zákazníků. Jedná se o test bonusový.

Rozlozeni zateze FITCoin
Testovací prostředí zkouší, zda se do řešení jednoho problému typu fitCoin dokáže zapojit více dostupných vláken. Pokud chcete v tomto testu uspět, musíte Váš program navrhnout tak, aby bylo možné využít více vláken i při analýze jedné instance problému. Jedná se o test bonusový.

Rozlozeni zateze CVUTCoin
Testovací prostředí zkouší, zda se do řešení jednoho problému typu čvutCoin dokáže zapojit více dostupných vláken. Pokud chcete v tomto testu uspět, musíte Váš program navrhnout tak, aby bylo možné využít více vláken i při analýze jedné instance problému. Jedná se o test bonusový.

Jak to vyřešit - pozor, SPOILER

Pokud se nechcete obrat o dobrý pocit, že jste úlohu vyřešili zcela sami, nečtěte dále.

- Oba problémy jsou výpočetně náročnější a vyžadují používání bitových operátorů.
- Problém typu fitCoin je v principu potřeba řešit hrubou silou, tedy odzkoušet všechny možnosti a spočítat, ty, které vyhovují podmínce vzdálenosti. Pro `n` vektorů velikosti 32 bitů to dává $n\ 2^{32}$ porovnání. To je pro praktické nasazení příliš, takový výpočet by trval desítky sekund pro jedno zadání. Proto je potřeba zadané vektory předzpracovat. Pokud je ve všech vektorech hodnota nějakého bitu stejná, lze tento bit v testování vyloučit. Například pokud je ve všech vektorech nastaven bit č. 17 na hodnotu 0, pak bit 17 nebude ovlivňovat výsledek. Ze zadané sady vektorů tedy nejprve odstraňte fixní bity a teprve na takto omezeném vstupu spusťte hledání hrubou silou. Většinou se tím zmenší rozsah úlohy na cca $n\ 2^{20}$, což je časově mnohem lepší.
- Pokud provedete úpravu zadání z minulého odstavce, dostanete mnohem menší rozmezí testovaných hodnot. Zároveň ale tímto postupem nezískáte přímo výsledek. Například máme zadané bitové vektory se 18 fixními bity (tedy budeme testovat celkem 2^{32-18} hodnot) a máme zadáno `distMax=6`. Pokud zpracujeme hodnotu `x`, která se od upravených bitových vektorů liší v 6 bitech, pak se do výsledku započte právě 1x. Pokud by se ale lišila v méně bitech, např. v 5 bitech, pak se do výsledku započte celkem 19 krát (4 odlišnosti - 172x, 3 odlišnosti - 988x, ...). Rozmyslete si, o jaký kombinatorický problém zde jde.
- Problém typu čvutCoin je postaven na problému porovnání editační vzdálenosti řetězců. Tedy na problému LCS, který znáte jako aplikaci dynamického programování. Protože prefixů i suffixů existuje celkem $8n$ (n je počet bajtů problému, prázdný řetězec netestujeme), lze problém vyřešit testováním $64n^2$ párů řetězců, každé porovnání spotřebuje řádově n^2 operací. Celkem tedy řešení jednoho problému čvutCoin má složitost n^4, vhodnou úpravou se dá složitost snížit na n^3.
- Testovací prostředí si nejprve sekvenčně otestuje rychlost odevzdaného řešení, podle toho pak upraví rozsah zadávaných dat. Úprava rozsahu testovacích dat ale není neomezená, odevzdané řešení musí být rozumně efektivní. Výpočet fitCoinu musí redukovat fixní bity, naivní řešení s $n\ 2^{32}$ operacemi časově neprojde. Pro čvutCoin lze těsně projít i s řešením se složitostí n^4.
- Vzhledem k heterogennímu charakteru vstupních dat se hodí objektový návrh s polymorfismem.

Další nápověda - SUPERSPOILER

Podle potřeby v průběhu řešení úlohy zveřejníme další nápovědy pro tápající studenty.

Vzorová data: Download

Odevzdat: Vybrať súbor | Nie je vybratý žiadny súbor — Odevzdat

---

Referenční řešení

- **Hodnotitel: automat**
  - Program zkompilován

- Test 'Test algoritmu FITCoin (sekvencni)': Úspěch
  - Dosaženo: 100.00 %, požadováno: 100.00 %
  - Celková doba běhu: 0.003 s (limit: 5.000 s)
  - Úspěch v závazném testu, hodnocení: 100.00 %
- Test 'Test algoritmu CVUTCoin (sekvencni)': Úspěch
  - Dosaženo: 100.00 %, požadováno: 100.00 %
  - Celková doba běhu: 0.000 s (limit: 4.997 s)
  - Úspěch v závazném testu, hodnocení: 100.00 %
- Test 'Zakladni test (1x thread, 1x customer)': Úspěch
  - Dosaženo: 100.00 %, požadováno: 100.00 %
  - Celková doba běhu: 0.682 s (limit: 30.000 s)
  - Úspěch v závazném testu, hodnocení: 100.00 %
- Test 'Test nekolika thready (1x customer)': Úspěch
  - Dosaženo: 100.00 %, požadováno: 100.00 %
  - Celková doba běhu: 0.199 s (limit: 29.318 s)
  - CPU time: 0.760 s (limit: 29.316 s)
  - Úspěch v závazném testu, hodnocení: 100.00 %
- Test 'Test mnoha thready (n customers)': Úspěch
  - Dosaženo: 100.00 %, požadováno: 80.00 %
  - Celková doba běhu: 0.391 s (limit: 29.119 s)
  - CPU time: 3.076 s (limit: 28.556 s)
  - Úspěch v závazném testu, hodnocení: 100.00 %
- Test 'Test zahlceni': Úspěch
  - Dosaženo: 100.00 %, požadováno: 80.00 %
  - Celková doba běhu: 0.899 s (limit: 28.728 s)
  - CPU time: 3.636 s (limit: 25.480 s)
  - Úspěch v závazném testu, hodnocení: 100.00 %
- Test 'Test zrychleni vypoctu': Úspěch
  - Dosaženo: 100.00 %, požadováno: 50.00 %
  - Celková doba běhu: 1.665 s (limit: 27.829 s)
  - CPU time: 3.736 s (limit: 21.844 s)
  - Úspěch v závazném testu, hodnocení: 100.00 %
- Test 'Busy waiting test (pomale pozadavky)': Úspěch
  - Dosaženo: 100.00 %, požadováno: 50.00 %
  - Celková doba běhu: 3.012 s (limit: 30.000 s)
  - Úspěch v nepovinném testu, hodnocení: 100.00 %
- Test 'Busy waiting test (pomale akceptace)': Úspěch
  - Dosaženo: 100.00 %, požadováno: 50.00 %
  - Celková doba běhu: 2.372 s (limit: 26.988 s)
  - Úspěch v nepovinném testu, hodnocení: 100.00 %
- Test 'Busy waiting test (complex)': Úspěch
  - Dosaženo: 100.00 %, požadováno: 50.00 %
  - Celková doba běhu: 3.105 s (limit: 24.616 s)
  - Úspěch v nepovinném testu, hodnocení: 100.00 %
- Test 'Velmi mnoho zakazniku': Úspěch
  - Dosaženo: 100.00 %, požadováno: 100.00 %
  - Celková doba běhu: 1.939 s (limit: 10.000 s)
  - Úspěch v bonusovém testu, hodnocení: 120.00 %
- Test 'Rozlozeni zateze FITCoin': Úspěch
  - Dosaženo: 100.00 %, požadováno: 100.00 %
  - Celková doba běhu: 0.223 s (limit: 10.000 s)
  - CPU time: 0.820 s (limit: 10.000 s)
  - Úspěch v bonusovém testu, hodnocení: 120.00 %
- Test 'Rozlozeni zateze CVUTCoin': Úspěch
  - Dosaženo: 100.00 %, požadováno: 100.00 %
  - Celková doba běhu: 0.179 s (limit: 9.777 s)
  - CPU time: 0.700 s (limit: 9.180 s)
  - Úspěch v bonusovém testu, hodnocení: 120.00 %
- Celkové hodnocení: 172.80 % (= 1.00 * 1.00 * 1.00 * 1.00 * 1.00 * 1.00 * 1.00 * 1.00 * 1.00 * 1.00 * 1.20 * 1.20 * 1.20)

- Celkové procentní hodnocení: 172.80 %
- Bonus za včasné odevzdání: 3.00
- Celkem bodů: 1.73 * (30.00 + 3.00) = 57.02

SW metriky:

	Celkem	Průměr	Maximum	Jméno funkce
Funkce:	**1**	--	--	--
Řádek kódu:	**9**	**9.00 ± 0.00**	**9**	`CCVUTCoin(const vector &,int,int)`
Cyklomatická složitost:	**0**	**0.00 ± 0.00**	**0**	