- Petriho sítě -

- Temporální logika - věčeření filozofů, p čeká dokud ne q, automat, např. v kritické sekci vždycky jeden proces
	- večeřící filozofové - rebel, aloha, ostraha, prioritní fronta

- Granualita - poměr objemu výpočtů k objemu komunikace, velký zprávy - vlákna strašně dlouho čekají, malý - velká komunikace (režie)

- Par. výpočty, musím synchronizovat - analýza uvíznutí, výkonnosti, ověření chování

-precedenční graf - jak na sebe navazují jednotlivé činnosti, uzel doba na výpočet, hrana - náklady na komunikaci

- paralelizace cyklů - určit co je lok proměnná, sdílená paměť, závislá(proměnná ke který přistupuje/zapisuje víc vláken, nezávislá (co čteme)

- uspořádaná proměná - pro výsledek bere předešlou hodnotu co jsme vypočítali, nejde paralelizovat

- cpu komunikuje s gpu přes sběrníci - úzké místo

- Multiprocesor s distribuovanou pamětí
	- každý cpu jenom svojí lokální pamět, není globální -> není sběrnice rychlé, komunikace pouze se sousedem pomocí zasílání zpráv, mřížka, na speciální úlohy

- vektorový par. počítač - operace s vektory čísel na úrovni instrukcí strojového kodu -> cíl zpracovat najednou co nejvíc dat, AVX
- Flynnova taxonomie - zavádí pojmy
	- MIMD - program co zpracovává datové toky
		- SPMD - program co zpracovává daotvé toky
		- SIMT - na gpu, snaží se aby všechny jádra na tom gpu vykonávali ty samý instrukce toho programu, sníží režii
	- SISD - není paralelismus, všechno seriově
	- SIMD - vektory, 4 double ve vektrou, jednou instrukcí přičtu k nim jiný vektor o 4 dablech
	- MISD (MPSD)- pipeline, MI - lineární urychlení, pro výpočty odolné vůči poruchám, jeden program počítá, nějaký ověřuje
	- MIMD - procesory pracují nezávisle na sobě, na lokalhostu ok, distribuovaně problém se synchronizací

- Flow a logically correct
	- flow program se chová deterministicky,
	- logically correct - dá správný výsledek, neznamená, že bude výsledek vždy stejný (3x stejný prvek v poli, vrátí index ten který vlákno najde první (náhodný))

- Synchronizace - ne sleep ale dělat kritický sekce

- Cena - závislost jak dlouho to trvalo na počet procesoru

- urychlení

	- SISD - nemělo by být, ale existuje TURBOBOOST
	- SIMD - odpovídá počtu vektorů v prvku
	- MISD/MPSD - pipeline urychlí se Nkrat n je počet procesoru v pipeline
	- MIMD/MPMD - záleží na datech, programu HW

?
- Perfektvní - urychlení < 1
- Anomální urychlení - urychlení = počet procesorů ?
- Superlineární urychlení - použije cache procesoru
?

- Vliv cache - rychlost jako blázen

- RAII - stará se o správné uvolnění paměti, vytvořím objekt ve scope, po opuštění scope se zavolá destruktor

- out of order execution - procesor načítá instrukce dopředu a hledá mezi nimi závislosti, pokud nezávislý, tak si je přehází tak aby je mohl dělat paralelně

- Registry Renaming - procesor má registry navíc, který skrývá pro programátory, pokud dojdou, přistupuje se do paměti

- pipeline
	- paralelismus na úrovni instrukcí, zpracování jedné instrukce lze rozdělit do několika fází jako načtení, dekodování, vykonání, přístup do paměti, v každém cyklu se dokončí instrukce, paralelismus zvětším počtem urovní pipeliny

- Specilative execution - aby procesor neztrácel čas s načítáním instrukcí snaží se začít vykonávat instrukce už v době kdy se předchozí instrukce teprve zpracovávají.. tipuje si u podmínek kam půjde skok

- Predikce skoku - snažím si tipovat jestli provedu skok nebo ne u podmínky, pro urychlení, pokud špatně tipnu zahazuju pipelinu
	- Branch Target Buffer - buffer kde se udržují statistiky predikce skoků, dvoubitové číslo
	- Static Branch Prediction Algorith - pokud se procesor ještě nesetkal s podmínkou předpokládá že je ok

- i & 1 - vrací jestli je číslo sudý nebo lichý

- rekurze proč není dobrá - zanořuju se od nějaký hloubky, špatně tipnu adresu kam se skočí z podprogramu, musím zahodit pipeline

- Falešná závislost - eax návratová hodnota, často nuluju, zabírá 5 byte, u xor jenom 2 bity

- Falešné sdílení (cache line) - 2 cpu každé svoje cache, pamětový systém zajišťuje nějakou konzistenci, nastane pokud procesory modifikuji svoji cache line a proměnné jsou lokální -> celá cache line je zneplatněna, musí na načíst znovu, pokud více vláken zapisuje do jedný proměnný

-Vektorové instrukce
	- vektor čísel se kterým dělám operaci najednou, šetší cache, autovektorizace

	- Loop Unrolling - rozepíšu sčítání ve smyšce - moderní překladače to umí

- Expression Templates -> FMA
	- třída reprezentující vektor, ukládá data, drží paměť, rychlost, umožnuje výpočet jednou instrukcí

- Transakční pamět
	- HW podporuje konkurenční přístup do paměti aby nedošlo k porušení dat
	- zámky paměti pomocí HW
	- 2 způsoby
		- velokost hlídané paměti je záležitost aktuálního procesoru
	- HLE
	- RTM - bez cmpxchg8b (je to u HLE), používá xbegin

- profilace
	- seriový kod -> par. -> chci vědět jestli jsem to naspsal dobře -> usoudím dobrý/špatný řešení. Procesor má HW počítadla kolikrát jsem čekal na načtení dat, kolikrát byl problém atd..... jsou i SW počítadla
	- HW počítadlo nedokáže překladač eliminocat ani předvypočítat
		- CPI - Cycles Per instructions - uniprocesor = 1, superskalarni může najednou zvádnout 2 instrukce -> CPI = 0.5
		- LLC Miss - Last Level Cache Miss - alokuju příliš velký blok paměti, dlouho čekám
		- Instruction Starvation - procesor čeká na instrukci kterou by mohl vykonat
		- Branch Misprediction - poměr špatně predpovězených skoků ke všem skokům

==== Vlákna =====
- obsluha perifeii - testuju vstup HW
- např jedno vlákno gui, druhý dělá výpočty, síť (jedno odesílá jedno čte jedno zpracovává)
- vyvolání dojmu rychlé odezvy
- urychlení výpočtu
- Atomická synchronizce
	- kritická sekce - časově drahé, v rámci zrychlení v KS co nejkratší doba
	- spinlock
	- nová hodnota na základě starý - místo toho aby se to celý zamklo v cyklu porovnáváme jestli se změnila puvodní hodnota pokud jo opakuju cyklus pokud ne zapíšu (compare_exchange_weak)
- volatile
	- problém více CPU často používaná proměnná uloží si jí z paměti do registru, každý cpu má tedy vlastní kopii -> není konzistence, proti uložení do registru se dá bránit pomocí volatile
- deadlock vs livelock
	- čeká se na podmínku co nenastane
	- deadlock - vlákno je uspané a čeká na podmínku - filozofové
	- livelock - vklákno aktivně čeká na splnění podmínky (smyčka dokud nebude menší, ale v tom inkrementuju)
- Spurious Wakeup
	- wait() může skončit aniž by někdo zavolal notify(), program mapuje vlákna na vlákna OS. wait se přeloží na systémový volání a OS definuje několik

případů kdy to může probudit bez notify
	- Stolen wakeup - jine vlákno se spustí dřív než vzbuzené
- Futex
	- fronta čekajících procesů je na úrovni jádra, ale počítadlo je v uživ. adresovém prostoru (vyhnutí drahým sys. voláním)
- Monitor - java
	- celá metoda jako synchronize, v jave
	- pokud stop tak se pamět neuvolní
	- semafor musí implemetovat OS
	- monitorenter monitorexit
	- synchronizace pomoci monitoru tam kde je nepotřebný mutex je pomolý
	- v jave používat balík utilities
	- kritické části runtime knihoven jsou v c++ nebo opt. strojový kod
- Escape analýza - kde se objekt musí alokovat stack/zásobník
- Eliminace analýzy - 1B nebo integer zarovnaný na délku strojového slova, dělitelný bezezbytku je na x86 garantováno atomicita r/w i bez lock
- Rendez Vous
	- prostředek pro synchronizaci
	- dvě vláka, jedno chce službu po druhým, pozastaví se, děje se výpočet v druhým až dokončí výpočet dá vědět prvnímu vláknu a to jde do stavu runnable. Výhoda druhý vlákno je extra a může mít různý oprávnění.
	- accept - přidrží vlákno
	- enter - pustí vlákno
	- oproti monitoru 1 vlákno
	- musím umět vzbudit vlákna v požadovaným pořadí (c++, c, ada, NE JAVA!!)

task-stealing plánovač - v Javě nikdy nebude, protože java nemá šablony ale type-ensure generiky

=====Paralelní programování======
- c++ - rychlejší, efektivnější kod který používá míň paměti
- RAII - proměnná ve třídě, alokuje se v konstruktoru, dá se na zásobník, jakmile se proměnná dostane out of scope zavolá se destruktor např. lock_guard
- unique_ptr - díky RAII bude prostředek uvolněn, pamět má právě jednoho vlastíka, zakázáno copy
- shated_ptr - obsahuje čítač kolik proměnných odkazuje na daný raw pointer, pokud na 0 objekt se uvolní
- weak_ptr - drží instanci shared_ptr, nic nevslatní
- type-erasure - pokaždé se vytvoří specializová funkce pro dany typ pokud mám třeba šablonu pro plus u double float string atd, nedělá se přetypování, optimalizuje se
- STL
	- standart template library
	- definuje pole, seznamy, hash, fronty - mají iterátory
- lock_guard - používt místo std::mutex, používá RAII, s koncem scope ukončuje kritickou sekci
- unique_lock - vytvoří zámek který se zamkne/odemkne až mu řekneme
- std::async - vykoná kod funkce nezávisle na vláknu ze kterého byla zavolána
- std::future - vrací hodnotu std::async
- std::promise - oproti async mám jistotu že se ty činnosti vykonají ve vláknech co jsme vytvořili
- thread vs task

	- s promise a future nestaráme se o plánování vláken ale jenom říkáme co se má vykonat paralelně

======= TBB =======
- multiplatformní knihovna, opensource
- cache cooling
	- vlákno běží pracovní data se dostávají do procesoru, když se přepíná nahrávají se nový data dalšího vlákna, předchozá data odstraňována, takže až původní vlákno dojde zase na řadu, tak bude cache-miss - chybí mu jeho data a musí se čekat než si je natáhne (data byly cooled)
	- tbb nepíše vlákna ale úlohy
	- TBB - pracuje na jedný úloze dokud není dokončená -> redukuje cache-cooling
- mám strom požadavků (precedenční graf), rozházím to na jádra tak aby ty výpočty co spolu závisí byli u sebe -> bude to pořád v cache a nebudu muset čekat, ty úlohy budou mít nějakou prioritu jak je zpracovávat, pokud jeden procesor dokončí svoje výpočty dřív může druhýmu procesoru ukrást jeho úlohy s nejnižší prioritou (task stealing), protože se nepředpokládá že budou potřeba pro další výpočty -> nebude cache-cooled
- Task stealing scheduler - jeden procesor bere naplánované úlohy druhému procesoru, aby se neflákal
- parallel_reduce dělí úlohy na 2 -> kvůli cache -> čím menší tím větší šance že se to dostane do cache -> rychlejší
- parallel_do - pokud nevím počet položek k paralelnímu zpracování, _feed - umožní přidávat
- parallel_pipeline
	- linkám se říká filtr, možnost jim nastavit jak se budou zpracovávat (seriově popořadě, seriově bez pořadí, paralelně)
	- vytváří n instancí paralelně
- flow graph
	- popíšeme výpočet jak se mají vykonávat -> strom, zaručí to, že se to počítá vždycky stejně,

======== GPU =========
- výpočty kde je málo podmínek, good na počítání

- OpenCL
	- kernel co vrací void
- rekurze na gpu nedělat
- Globální paměť
	- nejpomalejší paměť celého GPU
	- výkon závisí na tom jak se s ní pracuje
	- pamatovat: vyplatí si pamatovat, že se má omezit konkurenční přístup ke stejné paměti z několika různých work-items, které jsou vykonávány současně -> nahradit konstantami
- privátní paměť
	- rychlá při výpočtu jednoho work-item (thread na gpu)
	- velikost na konkrétním HW
	- použít co nejméně, pokud přesáhne uloží se do globální -> sníží výkon
- Lokální paměť a konstanty
	- lokální - sdílení dat mezi jednotlivými work-items v jedné work-group
	- konstantní - readonly, pro optimalizaci konkurenčního přístupu
- Barrier
	- synchronizace všech vláken ve work-group

	- work-groups mezi sebou nelze synchronizovat
- Fance
	- pouze pro jeden work-item
	- zajistí že se load/store instrukce dokončí před fence

CUDA
- C+ AMP použiju restrict abych rozlišil kam se kod bude kompilovat
- array_view - to samý jako u OpenCL (vytvoření bufferu, přesun paměť na GPU)
- index - index do work-group
- Coalesced memory - technika co se používá pro rychlost, pokud například přistupuju na nějaký offset na pozadí si do bufferu přenačtu hodnoty na dalším indexu, tato optimalizace ztratí smysl pokud v paměti skáču

==== Par. programování
- WIN
- winApi proces
	- virtuální adresový prostor
	- bezpečnostní kontext
	- pokud píšu runtime knihovny
	- rozšířený dřív když nebyla v jazyku podpora vláken
- Job object - možnost sloučení několik procesů dohromady a spravovat je
- winApi Thread - entita v rámci procesu, kterou plánuje OS
- Thread local storage (TLS) - data která jsou specifická pro daný thread
	- global - proměnný mezi vlákny
	- local - každé vlákno svou paměť
	- stack - každé volání rutiny má vlastní proměnnou
- thread pool - dostupný vlákna, můžu si z toho půjčovat vlákna nemusím je spouštět a ukončovat
- fiber
	- vlákno co je plánovaný v uživatelském adresovém prostoru, kvůli snížení režie přepínání do kontektu jádra OS a zpět
	- udělám klasický vlákno a pak ho předělám na fiber
- user mode scheduler - modernější než fiber
	- tbb využívá knihovnu co používá tohle
- synchronizace
	- semafor, monitor, časovač, proces, thread (čekáme než skončí)
- APC
	- beží vlákno, můžu mezi to naspecifikovat APC, který může vykonávát nějakou speciální činnost,
	- další forma synchronizace
- slim lock
	- optimalizovaný pro příady kdy kritická sekce nebo mutex představuje velkou režii
	- u vstupu dávám info co se v KS děje
	- vyplatí se pokud se chráněná data více čtou
	- možnost nastvit read/write
- inicializační fiasko
	- datová struktura co chce inicializovat několik vláken, ale smí se inicializovat pouze jednou
	- objekty se vytváří v jiném pořadí než jsme očekávaly
- zprávy
	- fronta zpráv abych ovládál GUI, asynchronní z pohledu příjemne
	- post message neblokující

	- send message odesílatel blokován
	- a pošle zprávu b a b je v deadlocku, a je taky v deadlocku, protože čeká na odpověď
- sdílená sekce DDL
	- jednu ddl i může načíst více procesů
	- podle flagů se pozná že se má vytvořit sdílený segment
	- data jsou potom sdílený pro všechny procesy co si danou ddl načtou

- POSIX
	- OS
	- api jádra, příkazová řádka a validace

- posix vs winApi - úplně stejný, celá funcionalita pomocí jednoho h souboru

========== distibuovaný výpočet ==============
- skládá se z N uzlů, komunikační kanál -> počítačová síť
- 3 realizace
	- univerzální poč. síť - kompy přes switch
	- univertální par. počítač - cluster, MPI
	- jednoučelový par. počítač - postavený na výpočet jedný konkrétní aplikace

- systém s distribuovanou pamětí
	- větší urychlení než systém se sdílenou pamětí -> díky paralelizaci komunikace (zatímco se data přenáší může počítat)
	- závisí taky na objemu interakce, hw architektuře, jak je sw optimální

- architektury
	- pravidelná - popíšu grafem
	- nepravidelná - internet

- směrování
	- pevná topologie
	- podle příjemce
	- podle odesílajícího

- systologické pole
	- několik propojených uzlů které si postupně s každým taktem hodin předávají data k dalšímu výpočtu
	- z jedný strany dávám data z druhý jdou výsledky
	- simplex, synchronní

- wormhole switching
	- packet je rozdělen na menší jednotky
	- dá se zmenšit komomunikační zpoždění

- SPMD
	- do všech uzlů v síti zavedu stejný program
	- do uzlů se zavedou specifická data
	- podle id provádí konkrétní činnost
	- princip farmer-worker
		- hodí se když výpočet je náročnej, ale zpráva ne

- Velikost zprávy
	- příliš velké objemy dat jsou rychlejší než maly ale zase se zbytečně čeká
	- příliš male objemy dat nevedou k urychlení
	- závisí na výpočetním výkonu uzlů

- lamportovy hodiny
	- počítám v událostech programu, příjmu data, odešlu data inkrementuju o 1 -> neřeším správně nastavený hodiny v systému
	- časové značky které umožňují částečně uspořádání
	- jedna zpráva hello, druhá world, chci asi to i tak přišlo

- vekrotové hodiny
	- lamport jenom pro 2 procesy
	- každý proces si zavede hodiny s procesem s kterým komunikuje
	- když něco udělá inkrementuje si události, pokud přijde zpráva z jinýho tak si pamatujeme jinou dobu a pak to zaktualizujeme na větší hodnotu co jsme dostali
	- schopnej detekovat co porušilo příčinu mezi něčím a následkem

- Komunikační schéma
	- držíme si sousedy u sebe v clusterech
	- inkrementujeme procesy který komunikují spolu nejvíce aby hsme zjistili clustery -> vyplatí se je držet u sebe kvůli rychlosti

========== MPI =========
- framework pro distribuované aplikace
- metody
	- MPI_init - proces definuje že se bude používat MPI
	- MPI_Comm_Size - počet procesů používající MPI
	- MPI_Comm_Rank - vrací vlastní indentifikační číslo v MPI
	- MPI_Send - odeslání zprávy
	- MPI_Recv - příjem zprávy
	- MPI_Finalize - proces říká, že už nebude používat MPI

- Komunikátor
	- zasílání pzrávy, message tag (typ zprávy), skupina (seznam procesů, pořadí, ...)
	- typy
		- intra - komunikace uvnitř skupiny
		- inter - komunikace mezi dvěma MPI procesy

- Komunikační režim
	- standartní
		- zpráva se kopíruje do bufferu, můžou nastat dvě situace. Zprava se vejde do bufferu není blokující, zpráva se nevejde, pošlu část, zablokuju se a čekám než to příjemce udělá recieve potom odesílám zbytek

	- buffer communication mode
		- můžu odesílat zprávy tak velký jako je buffer, pokud je zpráva

větší, vrací chybu

	- synchronní
		- send je vždy blokující pokud je zpráva přijata, příjemce udělal recieve

	- ready
		- zprávu odešlu jenom pokud na ní příjemce čeká, něco jako handshake

	- režimy liší se v názvu funkce

- Datové typy
	- popsány pomocí adresy, délky a id datového typu
	- můžu dělat struktury

- persistentní komunikace
	- periodicky něco dělám
	- obdoba TCP
	- princip
		- udělám si MPI_SEND_INIT / RECV
		- udělám požadavky
		- když to všechno mám zavolám MPI_START a MPI_WAIN, MPI odešle data a samo si řídí obsluhu

- skupinová komunikace - Complete Exchange
	- broadcast - jeden proces všem procesův ve skupině
	- all-to-all
		- každý proces odešle zprávu jinému procesu a příjme svojí zprávu
		- zpracovávám pole, každá skupina zpracovává MPI proces, až to všichni dodělají udělají all-to-all a tím se sesynchronizují (poskládají si to z ostatních procesů)

	- scatter - jedno velký pole a musím si ho rozdistribuovat všem procesům

	- geather - stáhnu si jednotlivý dílky od ostatních procesů a seskládám si výsledek

- Redukce
	- umožňuje udělat globální operaci - min/max prvek, průměr ....
	- u vektoru dělá vektorovou operaci

- transformace obrazu
	- načetl jsem obraz stupně šedi 1B a chci obraz přeškálovat na 0-255,
	- najdu pomocí redundace min a max
	- udělám scatter - každá část dostane podmnožinu vstupu
	- když to skončí stáhnu to pomocí gather
	- v MPI se předpokládá že všechny uzly jsou stejně výkoný

- práce se soubory
	- MPI umí použít C/C++
	- podporuje souborový operace -> cílem je odstranit úzká místa,

optimalizace
	- typy
		- displacement - kde začínají data
		- filetype - rozdělení souborů
		- etype - podmnožina filetype, menší bloky dat
	- práce se soubory
		- int MPI_FILE_read (MPI_File fh, void* buf, int count, MPI_Datatype datatype, MPI_Status *status)
		- int MPI_FILE_write (MPI_File fh, void* buf, int count, MPI_Datatype datatype, MPI_Status *status)

====== Metody urychlení distribuovaného výpočtu ========
- faktory ovlivňující rychlost
	- virtuální topologie (jak se vidí), jak to je zapojený, komunikační schéma, ne každý uzel stejně výkonný,

- obecné řešení
	- chceme to udělat tak aby uzly jeli co nejrychleji a nejmenší komunikační zpoždění

- typy úloh
	- s konečným časem výpočtu - rychle něco vypočte a skončí
	- s teoreticky nekonečným časem výpočtu - aplikace poskytuje nějakou službu, uzel vytížen jenom když něco chci
	- processing while in transit - zpracování dat během přenosu, výpočet nad daty během komunikace

- load sharing
	- předpokládá se prostředí pracovních stanit které nemusejí být vždy plně vytíženy
	- jeden uzel jako master, ostatní jako slave,
	- pokud master vytížen na 100, najde slave kam předá kus výpočet

- červ
	- jednotlivé procesy se mohou replikovat na nevytížených uzlech
	- komponenty
		- inicializační kod ke spuštění master
		- inicializační kod ke spuštění slave
		- výpočetní program
	- pokud stanice vytížena přejde na slave přehraje kod a spustí výpočet
	- nemůže růst donekonečna -> čím větší červ, tím rychlejší výpočet, ale synchronizace, stabilita velký špatný
	- nalezení nevytíženého uzlu - uzel nezná globální stav sítě a proto každý segment jede sám za sebe
	- musí umět uvolnit uzel

- Condor
	- snaží se o fér vyrovnání všech uzlů
	- centrální uzel arbitr -> rozhoduje co se kde pustí, sám hledá uzly
	- checkpoints - možné uložit stav procesu a ten odmigrovat na jiný uzel a tam ho restartovat, ukládá se periodicky, dobrý pro handly souborů

- Load-balancing
	- na rozdíl od load-sharingu se předpokládá že celá síť je dostupná pro

výpočet (u load-sharing nějakej uzel mohl pracovat pro někoho jinýho)
	- jak vyvažovat zátěž
	- statický výpočet
		- znám úlohu dopředu jak dlouho to počítá, zdroje, můžu vypočítat teoretický rozdělení sítě
	- dynamický
		- mění se v čase, umí se vyrovnat
	- pre-emptivní
		- procesy lze přerušit během výpočtu a přemístit na jiný uzel (nějaký proces mohl skončit svojí činnost)
	- centralizovaný
		- mají centrální prvek - arbit - rozděluje zátěž na uzly -> jeden uzel se všemi přetížení ?
	- distribuované
		- rozhodování o rozdělování zátěže provádí několik procesů až všechny, pokud jeden selže tak se restartuje
		- možnost rozdělit procesy na specialisty a výpočtáře
	- adaptivní
		- metody berou do úvahy i několik předchozích stavů při rozhodování o přidělení zátěže
	- kooperativní
		- může rozhodovat sám za sebe nebo může na rozhodnutí spolupracovat
		- přímo - procesy spolupracují nad konkrétním rozhodnutím
		- nepřímo - procesy dávají informace o svých rozhodnutích k dispozici ostatním a ty je použijí při svých rozhodnutích

	Techniky:
	- kdy uzel hledá jiný uzel kam přemístit část své zátěže
		- sender-initiated:
			- uzel když zjistí, že vytěžuje uzel na přes čáru, kouká jestli je v sítí jiný uzel kam by mohl předat část své práce, je to režie navíc

		- receiver-initiated
			- pokud zátěž uzlu klesne pod čáru, začne v síti hledat uzly odkud by mohl převzít práci
			- režie je ok, protože neplýtvá ničím, protože se fláká
			- něco jako task stealing

- Load Redistribution
	- neměří zátěž v počtu procesů jako load-balancing
	- měří jí ve skutečném vytížení (spočítá si na začátku benchmark)
	- poměr/cena/výkon bráško

- Programovatelná síť (Aktivní sítě)
	- nahrazení tradiční IP sítě
	- prog. síť dokáže spustit na uzlu kod v závislosti na obsahu přijatého PDU
	- potřeba
		- programový kod - execution environment
		- code distribution protocol - to co posílá kod
	- capsule
		- to co se posílá, hlavička, co se má spustit
	- init node načte kod do keše, načte si execution environment, udělá req

do target node, to zjistí, že tenhle kod nemá, vyžádá si ho, uloží do cache, vykoná, odešle data

- Load redistribution - Case Study
  - nezávislá na konkrétní aplikaci a síťový topologii, výkonnosti uzlů atd. - dělá si všechno sama za běhu
  - princip
    - procesy se rozdělují samy za sebe
    - periodicky zkoumají své okolí
    - nepřímo spolupracují
  - Discovery
    - PerformaceScouts - kapsule
      - u procesu, zjistí jak vypadá okolo uzlu z hlediska topologie, výkonnost, zatížení, a další info o zatížených uzlech
      - tyto informace se potom využívají při hledání uzlů -> kam odmigrovat proces, protože by tam mohl jít rychleji

    - Communication cluster -> pokud aplikace dobře napsaná, procesy vytváří skupiny mezi sebou
      - může být tolik clusterů co je procesů
      - vyplatí se držet procesy z daného clusteru blízko sebe, aby se snížilo komunikační zpoždění
      - během migrace si mohou procesy uzly i značkovat (modrý trojuhelník -> kudy je nejlepší cesta)

  - výkon a rychlost běhu procesu
    - lze změřit kolik proc času proces konzumuje na lokálním uzlu a následně porovnat, zda by mu mohl poskytnout více času
    - pomocí benchmarku se dá porovnat výkon

  - výběr nového uzlu
    - jdeme tam kam odesíláme už nějaký data, protože je blíž k uzlům s kterými se komunikuje

  - kolektivní rozhodování
    - synchronizace
      - každý proces sám za sebe ale nepřímo spolupracuje s ostatními pomocí backboards
    - masová migrace
      - několik procesů migruje u různých uzlů na stejný uzel
    - oscilace
      - migrace aniž by nic dělal, zavedení kreditů, nejdeš nikam dokud nemáš kredity (kredity budou stačit)
    - zbytečná migrace
      - viz oscilace
      - pokud dva přibližně stejné množství procesorového času může dojít k tomu, že se udělá migrace bez toho aby došlo k urychlení -> papírově zvýším procesorový čas

- Virtuální instrukce
  - diskrétní simulace
    - pseudo paralelní simulace
    - event-driven

		- takhle pc ale nefunguje

	- virtuální instrukce (VI)
		- vyjádříme výkon uzlu ve VI za simulovaný čas (ST) obdoba FLOPS
		- místo toho abychom proces uspali, musí odpracovat nějakou množství VI
		- takže doba po kterou bude proces spát bude záviset na tom kolik běží zrovna procesů

- Time Warp Simulace
	- Distribuovaná simulace
		- optimistická simulace, která pokračuje kupředu a předpokládá že vše probšhlo v pořádku
		- procesy spolu komunikují pomocí zpráv
	- existují troje hodiny
		- reálné
		- wallclock - jak dlouho simulace běží
		- simulační

- AVNMP
	- vytvoří se simulace sítě pomocí algoritmu z rodiny TimeWarp, překryje skutečnou síť
	- pro predikci co se stane se sítí
	- využití
		- při rozdělování zátěže chci predikovat stav které mu by bylo dobrý se vyhnout
		- prevence proti útoku DDoS

===== Big data & clound ========
- big-data
	- říká že jde o takový objem dat který nezpracujete se stávajícím vybavením v rozumně krátkém čase
	- nikdo neřeší jak je to napsaný, jaké programy s tím pracují, jestli jsou efektivní struktury
	- má smysl schraňovat tak evlký obsah dat ?

- Distribuovaný mapReduce
	- v pc se sdílenou pamětí map plánovač, a reduce odpovídá redukční funkci
	- v distribuovaném map odpovídá procesu rozdělení dat na části a jejich distribuci na jednotlivé uzly, reduce pak je provedení nějaké operace nad danou částí dat na příslušném uzlu, pak sloučím výsledek
	- součet prvků v poli -> kolik mám jader tak rozdělím pole, spustím jádru vlákno, řeknu kde má počítat sloučím výsledky

- Cloud
	- distribuovaný systém
	- infrastruktura jako služba
		- pronajmu si HW, nainstaluju SW co chci používat

- platforma jako služba
    - webhostin, nezajíma mě hw co jsem pronajal
    - zajímá mě jenom nějaká standarizovaná konfigurace, kde si můžu něco spouštět
- sw jako služba
    - office 365, je to někde na webu, nemám nad tím kontrolu
- víceméně nemám nad ničím kontrolu, když jsem firma v cloudu
-

```
locked:                     ; The lock variable. 1 = locked, 0 = unlocked.
     dd      0

spin_lock:
     mov     eax, 1         ; Set the EAX register to 1.
     xchg    eax, [locked]  ; Atomically swap the EAX register with
                            ;  the lock variable.
                            ; This will always store 1 to the lock, leaving
                            ;  the previous value in the EAX register.
     test    eax, eax       ; Test EAX with itself. Among other things, this
will
                            ;  set the processor's Zero Flag if EAX is 0.
                            ; If EAX is 0, then the lock was unlocked and
                            ;  we just locked it.
                            ; Otherwise, EAX is 1 and we didn't acquire the
lock.
     jnz     spin_lock      ; Jump back to the MOV instruction if the Zero Flag
is
                            ;  not set; the lock was previously locked, and so
                            ; we need to spin until it becomes unlocked.
     ret                    ; The lock has been acquired, return to the calling
                            ;  function.

spin_unlock:
     xor     eax, eax       ; Set the EAX register to 0.
     xchg    eax, [locked]  ; Atomically swap the EAX register with
                            ;  the lock variable.
     ret                    ; The lock has been released.
```