Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Ing. Petr Lukašík

Využití paralelních výpočtů a technologie Gridu pro rozsáhlé vědeckotechnické výpočty

Use of parallel computations and Grid computing technology for large-scale scientific computing

Disertační práce

Studijní program:	Inženýrská informatika
Studijní obor:	Inženýrská informatika
Školitel:	doc. Ing. Martin Sysel, Ph.D.

Zlín, červen 2016

Vydala **Univerzita Tomáše Bati ve Zlíně.**

Publikace byla vydána v roce 2016.

Klíčová slova:

Intranet Grid, optimalizace zátěže, plánování , JSDL, distribuovaný výpočetní systém, paralelní výpočetní systém, GPGPU, checkpointing, systém odolný poruchám

Keywords:

Intranet Grid, Optimizing the computing load, Scheduling , JSDL, Distributed computing, Parallel computing, GPGPU, Checkpointing, Fault Tolerant System

Disertační práce je dostupná v Knihovně UTB ve Zlíně.

Rád bych poděkoval všem, kteří kdy se mnou měli svatou trpělivost.

Prohlášení

Prohlašuji, že jsem disertační práci na téma Využití paralelních výpočtů a technologie Gridu, pro rozsáhlé vědeckotechnické výpočty vypracoval samostatně pod vedením doc. Ing. Martina Sysla, Ph.D. za použití literatury uvedené na konci mé dizertační práce v seznamu použité literatury.

Ve Zlíně dne:

Abstrakt

Práce je zaměřena na problematiku využití grid computingu jako nástroje pro rozsáhlé vědeckotechnické výpočty v průmyslové praxi a také jako prostředku pro optimalizaci zátěže hlavního serveru, který je nasazen v oblasti plánování a řízení výroby. Motivací pro využití gridu v této oblasti byla skutečnost velmi nerovnoměrného rozložení zátěže hlavního serveru a koncových stanic uživatelů.

Problém je řešen tak, že některé z algoritmů systému pro plánování a řízení výroby (ERP) jsou distribuovány na koncové stanice uživatelů. Zlepšení bylo dosaženo zejména tím, že systém dávkových úloh byl nahrazen událostmi řízenou distribucí objektů, které řeší některé standardní úkoly plánovacího procesu. Výsledkem byla optimalizace rozložení zátěže hlavního serveru v čase.

Cílem bylo také prozkoumat možnosti a navrhnout řešení pro využití metodiky paralelních výpočtů s využitím grafických akcelerátorů pro řešení úloh v oblasti distribuovaných výpočtů a technologie GRIDu.

Klíčová slova:

Intranet Grid, optimalizace zátěže, plánování, JSDL, distribuovaný výpočetní systém, paralelní výpočetní systém, GPGPU, checkpointing, systém odolný poruchám

Abstract

The article focuses on the problem of usage a grid computing as a tool for optimizing the load of the master server that is deployed in planning and production control. The motivation for the use of the grid is a known fact of the uneven load of the master server and the workstations.

Problem is solved so that some algorithms of the Enterprise Rresource Planning (ERP) system are distributed to the end-user workstations. The improvement was mainly achieved by that the system batch jobs has been replaced as the event-driven distribution of objects, which solves some of the standard tasks of the planning process on the user's workstation. The result was optimal load distribution of the server in a time.

Another objective was to investigate and propose solutions to use the methodology of parallel algorithms for solving problems of distributed computing and grid technology.

Keywords:

Intranet Grid, Optimizing the computing load, Scheduling , JSDL, Distributed computing, Parallel computing, Checkpointing, Fault Tolerant System

Obsah

Seznam obrázků

Seznam tabulek

Seznam použitých zkratek

Zkratky

2DFFT	2D Fast Fourier Transform algoritmus
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
cuBLAS	nVIDIA CUDA Basic Linear Algebra Subroutines
CUDA	Compute Unified Device Architecture
cuFFT	nVIDIA CUDA Fast Fourier Transform library
DirectX	aplikační rozhraní Microsoft pro GPU
ERP	Systém pro řízení a plánování výroby
FCFS	First-come, first-served
FFT	Fast Fourier Transform
FIFO	First-In First-Out
GLSL	OpenGL Shading Language
GPGPU	General-purpose computing on graphics processing units
GPU	Graphics processing unit
HIL	Hardware-in-the-loop
HLSL	High Level Shading Language

http	Hypertext Transfer Protocol
I/O	Input/Output
J2EE	Java Platform, Enterprise Edition
JSDL	Job Submission Description Langruage
JVM	Java Virtual Machine
LLC	Low-Lewel Code
MRP	Material Requirement Planning
NTP	Network Time Protocol
OpenCL	Open Computing Library
OpenGL	Open Graphics Library
POSIX	Portable Operating System Interface
QoS	Quality of Service
RMI	Java Remote Method and Invocation
RMI/IOOP	Internet Inter-Orb Protocol
RPC	Remote Procedure Call
RPG III	Report Program Generator
SQL	Structured Query Language
UNIX	Remote Procedure Call
XML	Extensible Markup Language
XSD	XML Schema Definition

Symboly

$\mathcal{T}(e_1),\ldots,\mathcal{T}(e_n)$	timestampy (časová razítka)událostí $e_1,\ldots,e_n$
$\rightarrow_i$	příčina a směr sledu událostí - závislost mezi odesílatelem a příjemcem zprávy

C	kapacita výpočetního zdroje
cp_i	chechpoint - pomyslná čára časového úseku obnovy pro všechny právě běžící procesy i
cp_i^j	časová značka (timestamp) checkpointu j, procesu p_i
e_i	událost i - výpočet, I/O operace…
e_i^j	událost na procesu p_i a pořadí operace j, v rámci procesu p_i
$e_i^1, \ldots, e_i^x$	sled událostí, produkovaných procesem p_i
GS	globální stav systému
h_i	množina událostí v čase, produkovaných procesem p_i
J	množina úloh $J := (1, \ldots, j)$
j	úloha - proces běžící na jednom výpočetním zdroji s jehož výstupem je dílčí výsledek
LS_i^x	lokální stav procesu P_i
M	makespan úlohy [s]
m_i	zpráva i
N	počet úloh uvolněných do zpracování
P	priorita úlohy $P \in \{1, \ldots, 5\}$ - zadáno uživatelem
p_i	proces i
$rec(m)$	příjem zprávy m
rm_i^j	okamžik příjmu zprávy procesem p_i a pořadí operace j, v rámci procesu p_i
S	množina výpočetních zdrojů $S := (1, \ldots, s)$
s	výpočetní zdroj

$SC_{ij}^{x,y}$	stav komunikačního kanálu, tj.- zprávy od události e_i^x, které proces p_i odeslal, a které proces p_j do události e_j^ya dosud nezaznamenal
$send(m)$	odeslání zprávy m
sm_i^j	okamžik odeslání zprávy procesem p_i a pořadí operace j, v rámci procesu p_i
T_i	transakce i - posloupnost všech událostí v intervalu mezi dvěma checkpointy, $T_x \in \langle cp_x, cp_{x+1})$
T_J	celkový čas běhu úlohy,$t_J = t_{scd} + M[s]$
t_j	čas aktivity úlohy[s]
$t_{k,l}$	čas spotřebovaný úlohou j_l na výpočetním zdroji s_k [s]
t_n	předpokládaný čas běhu úlohy - zadáno uživatelem [s]
t_q	nejdelší možný časový úsek pro aktivitu úlohy [s]
t_{scd}	čas délky běhu optimalizačního plánovače
y	událost y z množiny událostí $e_i^1, \ldots, e_i^x$ produkovaných procesem p_i

1 Úvod

Technologický vývoj umožňuje firmám instalovat stále větší výpočetní výkon, při vynaložení stejných nebo dokonce menších pořizovacích nákladů. Trend nárůstu výkonu a snižování koncové ceny zařízení je patrný, zejména v oblasti personální výpočetní techniky. Páteřní prvky a investice do nich, však podléhají jiné filozofii. Zde je nutná (v mnoha případech díky licenční politice dodavatelů softwaru, která se odvíjí ve většině případů od instalovaného hardwarového vybavení), velmi pečlivá úvaha o výkonnostních parametrech jednotlivých systémů. Výsledkem je poměrně vysoké zatížení serverových služeb, proti téměř zanedbatelnému zatížení výpočetní techniky na straně uživatele. Snaha o řešení tohoto problému, byla primárním podnětem pro myšlenku nástroje, který by dokázal část zátěže serverových služeb přenést na stranu koncových zařízení síťové infrastruktury. Proto byla navržena služba Gridu na úrovni intranetu, která do jisté míry tuto zátěž lépe rozloží a zajistí tak lepší využití systémových prostředků.

Současný stav řešené problematiky

Zajímavou oblastí pro využití služeb gridu, jsou výpočty související s návrhem výrobku a definicí jeho technických vlastností. Zde se obvykle řeší velké množství podobných úloh s rozdílnými vstupními parametry. Proto se nabízí možnost distribuovat jednotlivé části výpočtů do Gridu, a tím snížit časovou náročnost. Současná praxe je taková, že se výpočty obvykle provádějí na jediném výpočetním zdroji, nebo v lepším případě na více zdrojích, ale bez možnosti jakéhokoliv řízení a plánování jednotlivých částí výpočtu. Paralelní přístup k úloze (na úrovni CPU, nebo grafického procesoru GPGPU) se provádí v případě, že to umožňuje konkrétní softwarové vybavení. Pro rozsáhlé projekty připadá v úvahu propojení jednotlivých organizačních struktur do výpočetního Gridu a využít vysokého potenciálu této služby.

Výpočetní Grid [16] je primárně budován pro poskytování služeb v oblasti distribuovaných výpočtů. Předpokládá se, že tato koncepce umožní uživateli navrhnout, odladit a spustit úlohu, jejíž distribuce do výpočetního Gridu výrazně zkrátí celkový čas nutný na její zpracování.

Datový Grid [16] je budován především jako rozsáhlé datové úložiště, které poskytuje data napříč virtuálními organizacemi. Základní výhodou je sdílení datových struktur a jejich údržbu. To má velmi pozitivní vliv na jednotnou správu dat a správu bezpečnostních politik. Uživatel datového Gridu obecně nemá povědomí, kde jsou tato data uložena. Například European Grid Infrastructure je jedním z příkladů rozsáhlých datových gridů.

Cloud Computing. Výsledkem *masivního marketingu* je oblast datového a výpočetního Gridu spojována do obecného pojmu Cloud Computing. V praxi se jedná o rozsáhlé gridové služby, jejichž primárním úkolem je poskytování hardwarových a softwarových služeb za úplatu. To má výrazně pozitivní dopad na správu IT infrastruktury v řadě organizací. Data v oblacích však mohou na druhou stranu znamenat výrazně vyšší zranitelnost a existenční závislost na poskytovateli cloudových služeb.[1] Pravidlo pro nasazení služeb Cloud computingu by mělo vycházet

[1] Vždy existuje bezpečnostní riziko, že data mohou zůstat v oblacích navždy, pokud nad nimi nemám vybudovánu vlastní správu a vlastní infrastrukturu.

z pečlivého uvážení, které oblasti správy informačních technologií svěřit do péče poskytovatelů těchto služeb.

Obecně lze říci, že rozdíl mezi gridovou a cloudovou službou je v charakteru zpracovávaných úloh. Cloud je především využíván pro on-line uživatelské služby a transakce. Jako typický příklad lze uvést například službu Google Apps nebo SAP S/4HANA cloud pro poskytování služeb v oblasti řízení organizací - služby Software as a Service (SaaS).

Služby Gridu jsou naopak využívány v oblasti rozsáhlých dávkových úloh, které jsou náročné na spotřebu výpočetních zdrojů, zejména v oblasti vědy a výzkumu.

Důvody pro budování výpočetních gridů

Současný stav výpočetní techniky lze charakterizovat jako oblast, která doznala za velmi krátkou dobu výrazného pokroku na poli výkonových a cenových poměrů. Současný standardní stolní počítač nebo chytrý mobilní telefon má mnohem více paměti, a dokáže tak zpracovat násobně více instrukcí za jednotku času, nežli velké sálové počítače před dvaceti lety. To samozřejmě vyvolává myšlenky, jak více zužitkovat výpočetní kapacitu, která za prvé není plně využita, [2] za druhé přestože cena koncových zařízení [3] neustále klesá, je škoda aby nabízená výpočetní kapacita ležela ladem, a konečně za třetí, existují problémy, na které ani současně nabízená infrastruktura výpočetních prostředků nestačí, nebo je příliš drahá.

Lze zmínit například rozsáhlé úlohy vlivů rozvoje civilizace a její dopady na přírodní prostředí, dlouhodobé předpovědi počasí nebo návrhy složitých chemických a biologických struktur léčiv a netradičních materiálů. To všechno předpokládá masivní nasazení všech dostupných výpočetních prostředků.

Vývoj, přestože zaznamenal výrazný vzestup, dosud neposkytl zařízení, které by dokázalo využít skutečného paralelismu při výpočtech. Současný svět paralelních výpočtů doposud nepřekonal omezení, které vyplývá ze stávajících konstrukčních vlastností každého výpočetního stroje. A tím je, že každý výpočet probíhá

[2] Přestože zkušenost s řadou běžných programových vybavení personálních počítačů může evokovat myšlenku, že žádný výpočetní výkon, který máme právě k dispozici není dostačující.

[3] Zařízení, která lze propojit do sítě jejímž základním úkolem je zajistit spolehlivou komunikaci mezi těmito prostředky.

sekvenčně v časové ose postupným zpracováním jednotlivých instrukcí, které má integrovány centrální procesorová jednotka, [4] a která i přes neuvěřitelně rychlý rozvoj mikroelektroniky stále odpovídá návrhu sekvenční výpočetní jednotky Alana Turinga [97].

Proto jakákoliv snaha o nasazení masivního paralelismu naráží na dvě hlavní omezení. A tím je:

Za prvé nutnost časové synchronizace běhu paralelně pracujících výpočetních jednotek, a s tím související časová synchronizace zpracovávaných instrukcí [58].

Za druhé omezení vyplývající z vlastního principu sekvenčního zpracování. To znamená, že existuje hranice, za níž dále není účelné rozšiřovat paralelní výpočetní jednotky [50].

V důsledku těchto omezení, v případě časové synchronizace dochází k časovým prodlevám, které mají výrazný vliv na čas zpracování úlohy, a výrazně zvyšují režii řídicích a synchronizačních operací. [5]

Základní myšlenky pro návrh gridových služeb rozpracovali Ian Foster a Carl Kesselman v práci Computational Grids [34]. Prvotní myšlenka Gridu však vznikla v době budování elektrických silových rozvodů a snahy poskytnout vysoce spolehlivý systém distribuce elektrické energie. Propojení generátorů elektrické energie a spotřebičů do Gridu je skvělým testem spolehlivosti a nadčasovosti těchto koncepcí. [6]

Proto je snaha o propojení výpočetních zdrojů po vzoru elektrické rozvodné sítě přirozenou myšlenkou. Přínosem je koordinace práce všech zařízení tak, abychom vytěžili maximum z poskytované výpočetní kapacity. A to i přes známá omezení, která plynou z konstrukčních vlastností těchto zařízení.

[4] V roli centrální procesorové jednotky může být myšleno cokoliv, na čem lze provádět výpočty, grafickou kartou počínaje, moderní ledničkou zapojenou v internetu konče.

[5] Což vyplývá z Amdahlova zákona [13].

[6] Zatímco v rozvodech elektrické energie platí jasná hranice mezi spotřebičem a zdrojem, u výpočetního Gridu tato hranice není úplně jasně vymezená. Výpočetní zdroj může být také v roli spotřebiče, u něhož může sedět netrpělivý uživatel gridové služby čekající na výsledek.

Odolnost proti poruchám

Tolerance chyb (Fault tolerance) je schopnost systému plnit svou funkci správně i za přítomnosti poruch. Odolnost proti chybám je základní vlastností, která vede k vyšší spolehlivosti. Primární metodikou pro zvýšení spolehlivosti je prevence. Kontrolní mechanismy, jejichž smyslem je odstranit okolnosti, kterými vznikají poruchy, jsou velmi důležité při provozu rozsáhlých gridových služeb [38]. Každý systém má parametry, kterými je definována jejich odolnost proti chybám. Základním parametrem z pohledu Quality of Service (QoS) je definovaná šířka přenosového pásma. Jakákoliv odchylka od normálního chování má vliv na šířku přenosového pásma soustavy. K selhání tedy dochází, když se skutečné chování systému odchyluje od definice standardních parametrů. Chyba představuje neplatný stav, který není v souladu se specifikací konkrétní aplikace. Jinými slovy, chyba je příčinou selhání systému.

Diskutovanými pojmy v oblasti fault tolerance jsou odolnost systému proti chybám, a vysoká dostupnost systému. Rozdíl mezi oběma pojmy lze definovat následovně [16] [8].

- Systém odolný proti chybám má velmi malou pravděpodobnost vzniku poruchy, ale výrazně vyšší režii provozu.
- Vysoce dostupný systém má vyšší pravděpodobnost vzniku poruchy a zároveň nižší režii provozu.

Požadavky na systém odolný proti chybám.

- Porouchaná část nesmí způsobit výpadek systému (No single point of failure).
- Izolace chyb (fault containment) – chyba se nesmí dále v systému šířit.
- Dostupnost režimů návratu – systém musí umět obnovit svoji činnost od jistých definovaných bodů návratu (checkpoint).
- Pokud má systém poruchu, musí pokračovat v činnosti i během rollback procesu.

Prostředí distribuovaných systémů a Gridu je více náchylné ke vzniku poruchy. Za prvé rozsáhlá topologie gridů klade vysoké nároky na kvalitu síťových služeb ve smyslu výměny informací, ale také na odolnost proti výpadkům soustav, a za druhé vyšší riziko neodborného zásahu z hlediska poskytovatele výpočetních

zdrojů. Provozovatel a uživatel Gridu prakticky nemůže ovlivnit například vypnutí prvku na němž právě probíhá výpočet.

Odborné články a literatura [24] [51] [52] [57] [59] naznačují principy zvýšení odolnosti systému proti poruchám. Obecně jsou zmiňovány dvě techniky. První přístup je založen na principu definice kontrolních bodů zpracování (checkpointů) a zasílání zpráv o stavu systému. Druhý přístup využívá vysoké dostupnosti výpočetních zdrojů, a přeposílá do zpracování vícenásobné repliky úlohy. Obě tyto techniky mají vliv na vyšší zátěž výpočetních zdrojů.

Vlastnosti checkpointingu

Princip checkpointingu je rozšířený pro svoji odolnost proti chybám. Vlastnosti checkpointingu přímo závisí na určení délky intervalu jednotlivých checkpointů. Velmi krátký interval mezi checkpointy výrazně zhoršuje výkonové vlastnosti systému, a to zejména díky nárůstu režie systému v důsledku množství zasílaných zpráv a nárůstu synchronizačních operací jednotlivých komponent. Na druhou stranu dlouhý interval mezi kontrolními body může vést ke ztrátě informací potřebných pro obnovu systému po poruše [38].

Systémy založené na principu checkpointu také musí řešit časovou synchronizaci komunikace. Problém časové synchronizace je výrazný u geograficky rozsáhlých gridů. Ta probíhá na úrovni protokolu synchronizace času Network Time Protocol (NTP), a je závislá právě na geografické vzdálenosti komunikujících uzlů [88].

Z tohoto důvodu je definice vlastností a časového intervalu operací checkpointu poměrně náročný úkol. Při návrhu systému je třeba zvážit několik hledisek a rozhodnout o nejvhodnější strategii obnovy systému po chybě. Z hlediska mechanismu zaznamenávání zpráv a výjimek se jedná ve většině případů o plný (full) checkpointing nebo přírůstkový (inkrementální) checkpointing. Full checkpointing. je tradiční mechanismus obnovy chyby. Má však vysokou režii na registraci a záznam všech logovaných kontrolních bodů (checkpointů), tedy i těch bodů u nichž nedošlo v průběhu záznamu k žádné změně [10]. Alternativou pro snížení režie je návrh inkrementálního checkpointingu. Ten v jistých časových intervalech zaznamenává celkový stav aplikace. Následně ukládá jen ty kontrolní body (checkpointy), u nichž

došlo k nějaké změně. Tento typ checkpointingu má výrazně nižší režii na záznam kontrolních bodů, má však vyšší režii při obnově systému po havárii [10].

Z hlediska synchronizace časových značek systému je možno definovat synchronní (koordinovaný), nebo asynchronní (nekoordinovaný) checkpointing.

V případě nekoordinovaného checkpointingu každý proces zaznamenává své průběžné stavy nezávisle na ostatních spuštěných procesech. Výhodou je nízká režie a malá zátěž systému. V případě, že spuštěné procesy mají definovány precedenční vazby může v případě obnovy po poruše dojít k tzv. domino efektu, což znamená návrat na úplný začátek výpočtu. Díky této vlastnosti je tak možné ztratit celý výpočet. Proto se nekoordinovaný checkpointing v praxi příliš nepoužívá [37].

V práci [21] je uveden podrobný průzkum vlastností koordinovaného checkpointingu. Autoři se zaměřují na minimalizaci počtu synchronizačních zpráv a počtu kontrolních bodů v průběhu checkpointingu. Algoritmy koordinovaného checkpointingu nutí všechny procesy zablokovat výpočet (tzv. blokovací algoritmy) v okamžiku kdy probíhá operace checkpointu. To vede ke snížení výkonu celého systému.

Práce [70] [37] se zabývají neblokujícími algoritmy, které tento problém řeší. Velmi zajímavý je článek [37], který se tímto problémem zabývá v mobilních výpočetních systémech, které se stávají plnohodnotnými výpočetními nody (Mobile Hosts) díky ceně, vysoké dostupnosti mobilních sítí, a v neposlední řadě skvělému instalovanému výpočetnímu výkonu a softwarovému vybavení dnes nabízených chytrých mobilních telefonů.

Pro vlastní realizaci systému odolnosti proti chybám lze zvolit řadu softwarových metod a knihoven, které lze budovat v různých vrstvách aplikačního softwarového vybavení. Princip checkpointingu je uveden v kapitole 5.1.

Checkpointing na úrovni jádra systému (low-level checkpointing)

Zde jsou mechanismy checkpointingu vystavěny přímo v jádře operačního systému. Je k dispozici knihovna the Berkely Labs Checkpoint/Restart for Linux. Knihovna využívá systémové signály a zprávy (SIG...) linuxového kernelu. Nevýhodou je, že

ji nelze portovat na jiný systém [7] než linux[7]. Integrace checkpointingu na úrovni jádra systému je velmi účinná, nikoliv však snadná [37].

Checkpointing na úrovni knihovny – DMTCP)

Je nástroj pro transparentní checkpointing souběžných aplikací, včetně aplikací distribuovaných aplikací. Tato knihovna spolupracuje přímo s binárními spustitelnými soubory. Nevyužívá ke své činnosti žádný z modulů linuxového jádra. Knihovna DMTCP podporuje řadu aplikací na bázi: Matlab, Java, Python, Perl, Ruby, PHP, Ocaml, GCL (GNU Common Lisp), emacs, vi/cscope, Open MPI, MPICH-2, OpenMP [24].

Checkpointing na úrovni programu

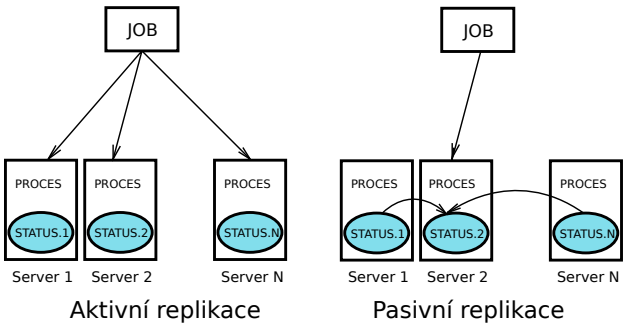
Využívá se u aplikací, u nichž je k dispozici zdrojový kód. Princip zachytávání chybových stavů (try, catch) pak lze využít pro rozesílání zpráv a signálů. Výhodou je, že vlastnosti aplikace jsou plně pod kontrolou svých tvůrců. Nevýhodou je nutnost vynaložit programátorské úsilí [62].

V návrhu Intranet Gridu je použita technika checkpointingu na úrovni programového vybavení a restartu aplikace. V případě, že systém vyhodnotí chybu, je tato úloha přeposlána stejnému zdroji, (z důvodu již existujícího výsledku rozplánování úloh) a v případě, že se tato chyba opakuje, je úloha přeposlána zdroji jinému. Zhorší se však optimalizovaný návrh rozvrhu úloh, protože v tomto případě nedochází k přeplánování. Toto má výhodu snazší implementace a nižších nároků na přeplánování úloh. K přeplánování dochází jen v případě trvalého výpadku jednoho nebo několika výpočetních zdrojů. Mechanismus checkpointingu v prostředí Intranet Gridu je popsán v kapitolách 5 a 9.2.5.

[7] Naskýtá se však otázka, zda má být distribuovaný výpočetní systém budován i na jiných systémech, a to zejména z důvodu otevřenosti standardů (POSIX), a v neposlední řadě nákladů na softwarové licence

Techniky replikace

Replikace je technika založená na předpokladu, že každý jednotlivý výpočetní zdroj je mnohem náchylnější k poruchám, než současné selhání více zdrojů naráz. Principem replikace je spuštění několika kopií stejného úkolu na více než jednom výpočetním zdroji. Počet replikací je přímo úměrný požadavku nárůstu spolehlivosti a nepřímo úměrný požadavku na výkonnost výpočetního systému. To znamená, že čím více je definováno replik jedné a téže úlohy, tím více narůstá odolnost systému vůči poruchám, ale tím také na druhou stranu klesá výkonnost systému, a to nejen násobně vyšší potřebou výpočetních uzlů, ale také nárůstem režie systému a zatížení síťového provozu. Stanovení optimálního počtu replik představuje relativně obtížnou disciplínu a je předmětem mnoha technických a teoretických úvah. Nárůst replik samozřejmě nezvyšuje odolnost systému lineárně. Nevhodná nebo zbytečně předimenzovaná volba počtu replikovaných uzlů vede k vyšší režii na správu a provoz celého systému [22].



Obrázek 1.1: Princip aktivní a pasivní replikace [53].

Aktivní replikace

Je definována tak, že každá úloha je zadána do zpracování více výpočetním uzlům [63]. Aktivní replikace vyžaduje, aby procesy běžící ve výpočetním uzlu byly deterministické. To znamená, že s ohledem na stejný počáteční stav, budou všechny procesy produkovat stejnou sekvenci odezvy a budou končit ve stejném koncovém stavu. Pro aktivní replikaci je důležité, aby všechny operace byly atomické. To znamená, že se provedou správně a oznámí správný výsledek, nebo se neprovedou

vůbec. Aktivní replikace je vhodná pro real-time aplikace, kde deterministické chování systému je podmínkou. Typickým příkladem pro aktivní replikaci jsou disková pole, kdy dochází k online replikaci dat. Podmínkou pro tuto replikaci jsou hardwarové prvky stejných vlastností [60].

Pasivní replikace

V případě pasivní replikace je definován pouze jeden server (primární), který zpracovává požadavky klientů. Primární server po zpracování žádosti, aktualizuje stavy na straně záložních serverů. Výsledek operace pošle zpět zadavateli úlohy. V případě, že primární server selže, sekundární server automaticky nastoupí na jeho místo. Výhodou pasivní replikace je, že může být využita i pro nedeterministické procesy. Nevýhodou pasivní replikace je ve srovnání s aktivní replikací, delší odezva systému na vzniklou poruchu [38].

Vícenásobné zpracování jediné úlohy má také vyšší nároky na systém i celý síťový provoz. V případě, že v průběhu žádná chyba nevznikne, jsou nadbytečné větve výpočtu zahozeny. Výhodou proti checkpointingu je snadnější mechanismus obnovy vzniklé chyby. Představitelem tohoto typu replikace je například Hadoop cluster [28]. Princip aktivní a pasivní replikace je znázorněn na obrázku 1.1.

Plánování úloh a plánovací strategie v prostředí Gridu

Plánovač je zodpovědný za optimální a spravedlivé rozložení zátěže v prostředí gridu. Obecně lze říct, že se jedná o zásadní komponentu, která spolu s počtem a výkonností výpočetních zdrojů, určuje celkový výpočetní výkon gridové služby. Stěžejní úlohou plánovače je nalezení co nejvíce optimálního postupu při rozdělování úloh pro všechny dostupné zdroje. Bohužel neexistuje univerzální algoritmus, který by řešil optimalizaci rozvrhu, pro všechny typy úloh.

Úkolem plánovače je navrhnout a optimalizovat posloupnost akcí, a umožní dosáhnout požadovaný cíl. Plánovací techniky jsou využívány v širokém spektru činností a představují významný nástroj pro optimalizaci. Neexistuje přístup, který by byl schopen tuto velmi širokou oblast řešit univerzálně. Obecně jsou plánovací metody zahrnuty do oblasti $\mathsf{NP}-\mathsf{úplných}$ problémů [97]. Proto existuje velké množství principů jak tuto úlohu řešit pro konkrétní situaci. Žádná z úloh však nevede

k úplně optimálnímu řešení. Vždy se bude vyskytovat určité pásmo neurčitosti, které určuje kvalitu plánovacího algoritmu [20]. Pro výpočetní Grid představuje princip plánování procesů klíčovou funkcionalitu. Ta je zodpovědná za přidělování výpočetního času jednotlivým úkolům, prováděným v rámci jednoho výpočetního zdroje. Výpočetní zdroj může zahrnovat široké spektrum definic, které budou vycházet z různých typů výpočetních struktur. Například CPU je chápáno jako základní prvek, kterému jádro operačního systému přiděluje v určitých časových kvantech jednotlivé úkoly. Plánovač jádra operačního systému zodpovídá za spravedlivé přidělování času jednotlivým úkolům, přičemž například priorita úlohy představuje jen jedno z kritérií, které musí zahrnout do své činnosti.

V rozsáhlých (distribuovaných) systémech však může být výpočetní jednotka chápána obecněji a může být začleněna do vyšších struktur. Grid obecně představuje velmi heterogenní výpočetní systém. Jeho topologie má obvykle strukturu typu hypercube [30]. Jednotlivé podmnožiny této struktury mohou představovat výpočetní jednotku typu výpočetní zdroj gridu nebo skupinu výpočetních zdrojů gridu.

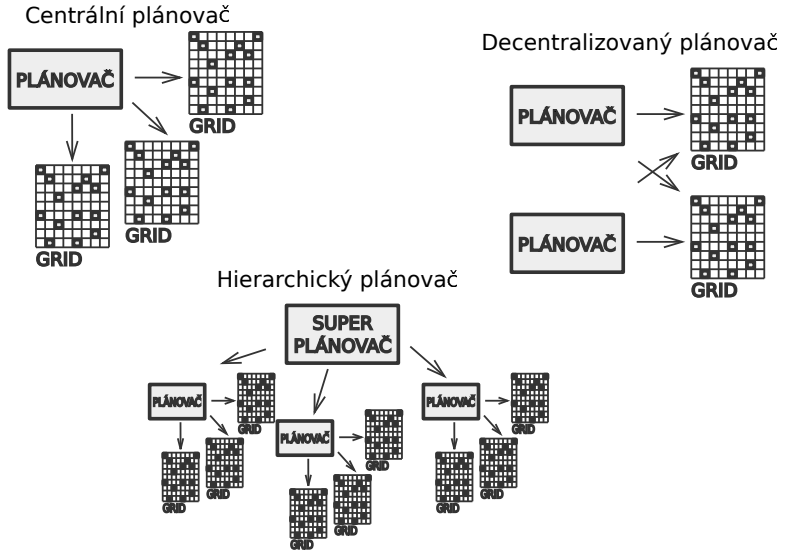
Plánovač Gridu v rozsáhlých topologiích typu hypercube může představovat několikaúrovňový hierarchický systém od centrálního plánovače, až po lokální plánovače v jednotlivých větvích systému, které optimalizují rozvrh v rámci své skupiny zdrojů [47].

Kvalita plánovacího procesu

Plánovač úloh musí být navržen tak, aby splňoval určité kvalitativní parametry. Zvolený plán a jeho metodika určuje pravděpodobnost úspěšné realizace zadané úlohy. Kvalita plánovacího procesu je závislá na volbě a rozměru kompromisních řešení. To je také důvod pro volbu různých alternativních přístupů k plánovacímu procesu. Hlavní požadavek optimalizace, tedy rozvržení úloh, na co nejvíce zdrojů s cílem dosažení nejkratšího času, musí jednoznačně respektovat kritérium spolehlivosti. Samotný algoritmus plánovače musí také splňovat určitá kritéria proveditelnosti a časové realizovatelnosti návrhu. To znamená, správnou volbu kompromisu mezi přesností návrhu a délkou výpočtu tohoto návrhu.

Topologie plánovačů

K začlenění plánovače úloh do struktury Gridu je věnována řada výzkumných prací a studií [14] [23] [47]. Plánovač Gridu je významná komponenta, jejíž kvalita, robustnost a spolehlivost určuje vlastnosti celé gridové služby. Topologii plánovačů gridových služeb lze rozdělit do tří nejčastěji se vyskytujících kategorií: *centrálního plánovače, decentralizovaného plánovače a hierarchického plánovače.*



Obrázek 1.2: Plánovače úloh, a jejich topologie.

Centrální plánovač řeší návrh plánu a časový rozvrh úloh v rámci celého systému. Výhodou je, že může poskytnout vyšší optimalizaci plánovacího rozvrhu, protože díky této koncepci má veškeré dostupné informace o všech úlohách i výpočetních zdrojích. Centralizovaný systém není příliš škálovatelný se zvyšujícím se počtem výpočetních zdrojů. Tato koncepce je vhodná pro malé gridové služby v rámci intranetu jedné organizace. Další nevýhodou je, že jeho výpadek způsobí výpadek celé gridové služby [100].

Systém decentralizovaných plánovačů rozděluje plánovací role každé lokalitě. To znamená, že úkoly směřované do určitého geografického uzlu soustředěných výpočetních zdrojů jsou obvykle distribuovány také na tento lokální plánovač. Díky decentralizované topologii je systém méně náchylný na úzká místa. Tato koncepce také nabízí vyšší škálovatelnost a rozšiřování systému. Výhodou je také to,

že při výpadku některé z komponent je možno přesměrovat výpočet jinam. To výrazně zvyšuje dostupnost a odolnost systému proti poruchám. Nevýhodou je, že takto navržený systém plánování může navrhnout méně optimální rozvrhy úloh, díky decentralizovanému rozložení informací o jednotlivých procesech. Zjednodušeně řečeno, systém nemusí zahrnout do plánovaného rozvrhu nevytížené výpočetní zdroje z jiné lokality [47].

Hierarchická struktura plánovačů se snaží propojit výhody centralizovaného a decentralizovaného přístupu k plánování úloh. Hlavní plánovač, tzv *meta-scheduler* se snaží lépe rozložit výpočetní výkon v jednotlivých větvích gridové služby. Hlavní plánovač není zodpovědný za optimální rozvrh celé soustavy ale za optimální zatížení celé soustavy.

Definice úlohy

Pro definici úlohy je využívána řada skriptovacích jazyků, které umožňují vytvořit interaktivní nebo dávkovou definici úlohy. Souhrnně se nazývají "Application Description Languages", například VDL, Condor DAGMan, JSDL, AGWL, JDL,... Poměrně velké množství jazyků a přístupů k definici úlohy v prostředí distribuovaných výpočtů a Gridu, přineslo jisté komplikace v oblasti interoperability a spolupráce v rozsáhlých topologiích. Proto vznikla snaha definovat jednotnou platformu pro definici úlohy.

JSDL - Job Submission Description Language je specifikace jejíž návrh iniciovalo konsorcium Open Grid Forum. JSDL je vhodné pro definici a popis jednoduchých dávkových úloh. Systém JSDL 1.0 je motivován dvěma hlavními myšlenkami. Za prvé, odstranit již zmíněnou roztříštěnost v oblasti definice úlohy, a za druhé poskytnout jednotný a transparentní nástroj, který usnadní definici úlohy. Je implementováno například v Condor, Globus Toolkit, LoadSharing Facility (LSF), Portable Batch System (PBS) [11].

Z dalších velmi často využívaných skriptovacích jazyků lze zmínit např. **JDL - Job Description Language** jazyk pro definici a parametrizaci dávkových úloh implementovaný například v gLite - Lightweight Middleware for Grid Computing, který je provozován v CERN LHC [83].

V projektu Intranet Gridu byl implementován skriptovací jazyk JSDL 1.0. Komponenta analyzátoru přímo generuje spustitelné skripty v okamžiku definice úlohy. Výhodou tohoto řešení je snadnější implementace služby výpočetního zdroje, který nemusí mít syntaktický analyzátor implementován. V závislosti na typu operačního systému, který je instalován na výpočetním zdroji je také generován typ skriptu pro spuštění úlohy. Popis syntaktického analyzátoru JSDL je uveden v kapitole 8.

Principy distribuce úloh

Push nebo Pull je mechanismus, který určuje, jakým způsobem jsou distribuovány úlohy v prostředí gridu. Volba zda Push nebo Pull, závisí na typu a zaměření gridové služby. Pro metodu **"Push"** platí, že server odesílá informace k výpočetním zdrojům, zatímco při **"Pull"**, výpočetní zdroj požaduje serverovou stranu o přidělení práce. Obecně nelze říci, která ze strategií je lepší [75]. V návrhu Intranet Gridu jsou použity obě metodiky. Při využití Gridu pro distribuci zátěže systém pro řízení a plánování výroby (ERP) systému je server Gridu v roli distributora objektů jednotlivým výpočetním zdrojům (kapitola 9.1). Při návrhu Gridu jako systému pro distribuované výpočty má server roli distributora pracovních příkazů (shellových skriptů) jednotlivým výpočetním zdrojům. Ty si následně vše potřebné k provedení práce obstarají samy. Mechanismus je popsán v kapitole 9.2.

Bezpečnostní politiky v oblasti distribuovaných výpočtů a gridových služeb

Požadavky na nastavení kvalitní bezpečnostní politiky v oblasti distribuovaných výpočtů a gridových služeb jsou umocněny jejich rozsáhlou topologií. Gridová služba vybudovaná v rámci kooperujících organizací vytváří systémovou nádstavbu, tzv. Virtuální organizaci [31], která umožňuje jednotlivým spolupracujícím subjektům využívat společné softwarové a hardwarové zdroje koordinovaným způsobem. To znamená, že jsou definována pravidla, která jasně říkají jakým způsobem a za jakých podmínek lze využít nabízené zdroje, ale také pravidla, která vymezují bezpečnostní požadavky při provozování rozsáhlých systémů.

Definice systému bezpečnosti je dána souhrnem zkušeností, a samozřejmě poměrem výkonu a ceny. Obecně lze říci, že minimalizace bezpečnostních hrozeb a cena vložená do realizace bezpečnostních politik organizace má závislost nepřímé úměry. Platí také, že nejvíce zranitelností a bezpečnostních incidentů je způsobeno útoky uvnitř organizací, tedy vlastními zaměstnanci. V drtivé většině případů neúmyslnými - lidskou chybou, nebo neznalostí. Proto platí, že trvalá osvěta je mnohdy výrazně účinnější než velmi drahá a sofistikovaná zabezpečovací technika. V sekci 9.3 je popsána definice vymezujícího prostoru tzv. *sandboxu*, [8] který eliminuje případné chyby uživatele při práci s gridovou službou.

V článku [98] je formulována myšlenka základních požadavků na provoz služby v rámci Virtuální organizace. To znamená umožnit přístup ke zdrojům, které vlastní konkrétní organizace, a které je tato konkrétní organizace ochotna poskytnout vyšší struktuře – Virtuální organizaci. Z hlediska bezpečnosti je důležité, že členové zastřešení Virtuální organizací se řídí stejnými pravidly bezpečnosti, které definuje poskytovatel konkrétního zdroje.

Obecně škálovatelnost, výkon, různorodost a otevřenost jsou žádoucími vlastnostmi všech distribuovaných systémů. To však může vést k řadě bezpečnostních incidentů. Systémy pro distribuované výpočty a gridové služby mohou vyžadovat některé (nebo všechny) standardní bezpečnostní funkcionality, včetně ověřování, řízení přístupu, integrity, soukromí a nepopiratelnosti [35].

Digitální certifikát X.509 je standardem například pro rozsáhlé gridové služby Globus nebo gLite. Toto řešení umožňuje definovat jednotnou autentizaci uživatele. Použití certifikátu X.509 definuje vzájemný vztah důvěryhodnosti mezi uživatelem a vstupním bodem do prostředí gridové služby [46].

Řada systémů gridových služeb je vybudována na standardních prostředích (např. J2EE nebo .Net), které poskytují standardní a ověřené kontejnery pro definici bezpečnostních politik. Ideální je, když je vývojář gridu odstíněn od implementace bezpečnostních zásad vlastními silami a použije některou z nabízených knihoven [98].

[8] Sandbox - pískoviště které vymezuje prostor, na němž může daná aplikace provádět cokoliv, aniž by něco poškodila.

Intranet Grid byl vybudován v prostředí J2EE, které nabízí rozsáhlou škálu bezpečnostních definic. Ty jsou k dispozici v implementačních knihovnách a rozšířeních programovacího jazyka Java:

- **JAAS** (Java Authentication and Autorization service) rozhraní a služby pro ověřování a autorizaci uživatelů
- **JCE** (Java Cryptography Extention) podpora pro šifrovanou komunikaci
- **JSSE** (Java Secure Sockets Extension) rozhraní pro přenos v zabezpečené vrstvě SSL
- **Security manager** rozhraní ochranné domény, která je definována pomocí zásad zabezpečení (java.security, java.policy)

Systém zabezpečení Intranet Gridu je popsán v kapitole 9.3.

Možnosti netradičních přístupů k paralelním výpočtům

Současný trend masivního paralelního zpracování není samospasitelný, protože naráží díky Amdahlovu zákonu na technologická omezení. Mooreův zákon o růstu tranzistorů v integrovaném obvodu také narazí na svoji technologickou hranici. [9] Kniha autorů Greenlaw, Hoower a Ruzzo [41] definuje hranice současných paralelních výpočetních systémů.[10]

Na scénu proto nastupují další zajímavé obory, které si budují své místo na výsluní informatiky. Kromě systémů založených na paralelismu biologických struktur se výrazně prosazuje obor kvantové informatiky, jejíž prvotní základy položili Richard Feynman a Ed Fredkin [96].

Význam kvantových výpočtů je patrný z následujícího tvrzení. Máme-li klasický osmibitový registr, jsme schopni v něm provést najednou 1^8 výpočtů. Máme-li ovšem k dispozici kvantový osmi qubitový registr, jsme schopni provést v jednom časovém okamžiku 2^8 výpočtů. A to opravdu stojí za pozornost. (Kvantový registr o délce 2^{500} dokáže provést paralelně tolik výpočtů, kolik je zhruba atomů v pozorovatelném vesmíru [73].

Bioinformatika

Charles Bennett z výzkumného centra IBM popsal v roce 1982 pozoruhodnou podobnost mezi využitím DNA struktur přírodními mechanismy a způsobem, jak jsou používány databáze při řešení konkrétních úkolů. Leonard Adleman (jeden z autorů RSA šifry) předvedl, že shluk DNA molekul může řešit jednoduché kombinatorické úlohy. A v roce 2002 předvedl systém, který je schopen řešit úkol s dvaceti proměnnými. Společnou vlastností DNA počítačů je předpokládaný masivní para-

[9] Definice Mooreova zákona je silně závislá na technologických možnostech a fyzikálních limitách polovodičových materiálů. Odhaduje se, že za 20, až 40 let narazí klasická technologie výroby polovodičových čipů na svoji hranici [36].

[10] Vypadá to že jsme narazili na hranici toho, čeho je možné dosáhnout s počítačovými technologiemi. Člověk by si ale měl dávat pozor na takováto tvrzení, protože do 5 let se obvykle ukáží jako pěkná pitomost. (John von Neumann, 1949)

lelismus. Leonard Adleman popisuje v článku [9] princip využití DNA struktur pro masivní paralelní výpočty.

Ehud Saphiro, Tom Ran a Shai Kaplan z Weizmann Institute of Science zveřejnili článek [91], kde představují principy využití biologických molekulárních struktur, vhodných pro řešení paralelních úloh.

Přestože zatím existuje mnoho nevyřešených překážek, lze s velkou pravděpodobností předpovídat velmi úspěšné řešení a perspektivu. Jedná se o nesmírně zajímavou oblast, kterou příroda dovedla k absolutní dokonalosti. No upřímně řečeno, měla dvě obrovské výhody. Za prve dostatek času, a za druhé moc jí do tohoto vývoje nikdo nezasahoval. Proto má zatím v této oblasti absolutní prim. Nám nezbývá nic jiného nežli se úporně snažit, abychom dosáhli také skvělých výsledků. Bohužel však nemáme k dosažení podobné dokonalosti ani zlomek času, který měla k dispozici příroda.

Kvantový výpočetní systém

Využití kvantových vlastností částice (elektronu, fotonu), jako nositele informace. V klasickém výpočetním systému se využívá deterministického přístupu kódování informace na základě binárního kódu, jehož hodnotami jsou předem dohodnuté dvě hladiny napětí, které představují základní informační jednotku - bit. Kvantově mechanický přístup definuje informační jednotku - qubit, která může nabývat nekonečné množství hodnot mezi nulou a jedničkou. Tento přístup připomíná princip analogového počítače, který byl využíván pro modelování dynamických jevů v oblasti regulace. Klasický analogový počítač však postrádá masivní paralelismus, který je ovšem kvantovému počítači vlastní.

Obecně, každá klasická výpočetní operace $(NAND, XOR, \dots)$ je prováděna nevratným způsobem. [11] To znamená, že se nelze vrátit na začátek výpočtu, nebo dokonce nelze rekonstruovat, z jakých vstupních hodnot vznikl výsledek. Obecně víme, že výsledkem binární operace NAND jsou dvě vstupní hodnoty. Pro logickou jedničku na výstupu hradla nezáleží, zda byla logická nula na jednom ze vstupů a na kterém nebo na obou – důležitý je výsledek. To vede jednoznačně ke ztrátě

[11] I triviální operace součtu je nevratná - z výsledku nelze vyčíst, z čeho byl složen $1+2=3$ nebo $2+1=3$ nebo $1+1+1=3$.

informace a nárůstu entropie systému. Ten se díky tomuto jevu zahřívá [73].[12] Tím je určena technologická hranice klasických výpočetních systémů, která je právě omezena nárůstem vnitřních energetických ztrát křemíkových čipů při snižující se velikosti jednotlivých konstrukčních prvků.

Jakkoliv nalezené kvantové algoritmy vypadají slibně, zásadní problém spočívá v tom, že není vůbec snadné takový kvantový systém sestrojit. Kvantové bity jsou buď velmi citlivé na rušivé vlivy okolí, nebo je značně obtížné je přinutit, aby spolu navzájem komunikovaly. Má-li být kvantovou informací například spin elektronu, musíme jej velmi dobře izolovat od okolí, pokud nechceme, aby se z logické jedničky stala samovolně nula. Protože spin je velmi citlivý na magnetické pole. Dalším problémem jsou fázové posuvy při interakci atomů s okolím. Kvantové superpozice jsou velmi křehké. V tom také spočívá jedna z odpovědí, proč nepozorujeme Schrödingerovy kočky, tedy superpozice dvou různých stavů (živá a zároveň mrtvá) u makroskopických objektů. Vlivem i nepatrné interakce s okolím, se systém dostává do jednoho z normálních stavů, které jsme zvyklí pozorovat [73].

Toto jsou jen v krátkosti nastíněné základní problémy, kde za každým z nich se skrývá obrovské množství velmi zajímavého vědeckého výzkumu, a které napovídají, že výpočetní systémy založené na biologických a kvantových zákonech jsou velkou výzvou. A to i přes technologické překážky, které představují.

[12] Samozřejmě, že k růstu teploty na hradle přispívá do rovnice energetické bilance soustavy více činitelů. Ztráta informace je však jedním z nich.

Cíl výzkumného záměru disertační práce

Cílem práce je vypracování metodiky, návrh a praktické nasazení systému pro řešení některých náročných výpočetních úloh s využitím metodiky distribuovaných a paralelních výpočtů. Výzkum byl zaměřen na obecný návrh úloh, které se obvykle vyskytují, zejména v komerčním prostředí.

Jsou demonstrovány některé typy výpočtů, jako nástroje pro optimalizaci a rozložení nadměrných zátěží hlavního serveru při výpočtu nákladové kalkulace výrobku. Dále jsou uvedeny možnosti nasazení Gridu jako výkonného výpočetního clusteru pro simulace prováděné při návrhu výrobku.

- Navrhnout a prakticky realizovat gridovou službu, která umožní definovat a zpracovat konkrétní úlohu při využití stávajícího programového vybavení.
- Prozkoumat možnosti nasazení paralelních výpočtů na grafických kartách, (Graphics processing unit (GPU)) a nasadit tuto technologii v prostředí výpočetního gridu.
- Porovnat výkonnost dostupných technologií GPU a podpory programátorských vývojových prostředí ze strany výrobců grafických procesorů.
- Navrhnout systém pro snadnou definici, zaplánování a spuštění úlohy.

Praktické nasazení gridové služby předpokládá nezávislost na hardwarové a softwarové platformě. Bylo počítáno s nasazením na platformách IBM iSeries, Linux a Windows. Snahou bylo také definovat takové prostředí, které nebude uživatele nijak omezovat, a umožní mu snadnou definici úloh a zajistit přímou podporu standardního programového vybavení a systémových prostředků ke komunikaci a výměně dat.

Součástí výzkumu byla také oblast plánování úloh, distribuce vstupů, výstupů a zpracování výsledků. Jedná se zejména o popis plánovacích algoritmů a mechanismů řešení kolizních případně chybových stavů úloh. Principy plánování a optimalizace zpracování nejsou dodnes uspokojivě řešeny, díky neznalosti univerzálního algoritmu pro všechny typy úloh. Proto je diskutováno několik obecných

přístupů k tomuto problému a snaha o návrh nejvhodnějšího řešení pro některé typy těchto zpracovávaných úloh.

Posouzení inženýrského a vědeckého přístupu

Práce na projektu Intranet Gridu umožnila posoudit dva naprosto odlišné přístupy k řešení problému. Inženýrský a vědecký přístup. Inženýrský přístup preferuje přímočará konkrétní řešení, která dávají výsledky v co nejkratším čase. Hlavním kritériem je minimalizace vstupních zdrojů, a pokud možno rychlé uplatnění v praxi. Chybí požadavky na zobecnění a optimalizaci. Inženýrský přístupu lze demonstrovat na zkušenosti se zaváděním systému pro řízení a plánování výroby v dnes již neexistující strojírenské firmě ZPS, a.s. ve Zlíně. Na začátku devadesátých let byla firma postavena před strategický úkol, zavést do praxe interaktivní systém, který by umožnil výrazný posun v oblasti řízení logistiky a výrobního plánování. Nejedná se o typický příklad, protože vstupní investice do systému byla velmi vysoká a příprava byla časově náročná. Výsledek však výrazně převýšil původní předpoklady. Jen zavedení poměrně triviálního modulu řešení skladových zásob přineslo výrazné úspory v rámci logistiky celé firmy. Bylo zdokumentováno, že jen úspory v rámci logistiky skladů zaplatily velmi vysokou vstupní investici do systému. Jednalo se v té době o jednu z prvních instalací systému CIMAPPS firmy IBM. Tento systém byl ve strategii firmy IBM prezentován jako nástupce velmi úspěšného systému COPICS. Jeho nespornou výhodou byla špičková stabilita a spolehlivost, která byla odvozena od vynikající platformy mainframů firmy IBM.[13]

Po letech vysoce spolehlivého provozu (v průběhu deseti let, nebyla zaznamenána žádná závažná porucha ani výpadek systému), byl provoz tohoto systému ukončen a nahrazen systémem jiným. Ukázalo se, že obchodní strategie byla pro tento systém chybná. Firma IBM v tomto případě nezachytila dravou konkurenci, která vsadila na výrazně levnější a méně spolehlivé hardwarové a softwarové platformy. Provoz tohoto systému, který měl nasazeny jen čtyři základní moduly pro

[13] Pozoruhodná byla také hardwarová výbava. Procesor chlazený vodou a 128 MB(!) operační paměti pro cca 800 uživatelů pracujících současně. Rádiové pojítko (wifi) mezi lokacemi ve Zlíně a Malenovicích, bylo ve dvou rackových skříních 19" - to vše deset let bez jediné vážné poruchy.

řízení skladového hospodářství a plánování výroby, však měl zásadní vliv na zkvalitnění logistického řetězce v celé firmě. Inženýrský přístup lze v tomto případě demonstrovat na skutečnosti, že systém řízení firmy byl zaveden a spolehlivě sloužil řadu let, ale nepřinesl zásadní inovativní prvky a vzory, které by ukázaly cestu dalším projektům.

Inženýrský přístup však zcela selhává v okamžiku, kdy je nutná zásadní inovace procesu nebo průmyslového vzoru, jehož výsledkem je etalon, který znamená výrazný kvalitativní posun v dosavadní praxi.

To lze prezentovat na velmi úspěšném projektu vícevřetenového CNC automatu TMZ642 CNC, který představuje strategickou koncepci stroje, v tomto případě již v režii nástupnické firmy TAJMAC-ZPS, a.s. Jedná se o stroj, kde jsou použity velmi zajímavé konstrukční prvky, které výrazně zvýšily užitné vlastnosti vícevřetenového soustružnického automatu. Pro představu má tento stroj 56 řízených os (použity dva systémy SINUMERIK 840D v režimu master-slave), které umožňují provádění všech operací, které souvisejí s obráběním kovů.

Tento projekt nebylo možno úspěšně zvládnout, bez uplatnění vědeckého přístupu, při řešení řady klíčových problémů. Výraznou technologickou komplikací byl například přenos kroutícího momentu na vřetena stroje. Nabízelo se několik řešení, včetně zabudování motorů, přímo uvnitř vřetenového bubnu, včetně nevýhody ztráty přesnosti při zahřátí pohonů uvnitř vřeten a přenosu elektrické energie včetně řídicích signálů pomocí nespolehlivých sběračů. Takto navržená koncepce vyžaduje sběrače přenosu energie kvůli indexu, tedy přetáčení vícevřetenového bubnu při přechodu na další technologickou operaci. Nakonec bylo uplatněno patentované řešení umístění pohonů mimo pracovní prostor a přenos mechanické energie na vřetena, pomocí soustředně uložených centrálních hřídelí, pro přenos kroutícího momentu. Zde se však vyskytl další problém, kdy při vzájemné rotaci soustředných hřídelí docházelo při velmi vysokých otáčkách k rozkmitu a následnému svaření, soustředně uložených radiálních hřídelí. To vedlo k destrukci hlavní pohonné jednotky. Uplatnění vědeckého přístupu ve spolupráci s vysokými školami bylo navrženo řešení s použitím karbonových kompozitů, které díky své hmotnosti a pevnostním vlastnostem dokázaly tento problém úspěšně a velmi spolehlivě vyřešit. Toto řešení však vyžadovalo naprosto odlišný přístup uplatňovaný v běžné praxi.

Prvky vědeckého přístupu při řešení technických problémů umožnily dosáhnout špičkové parametry aplikovatelné v širokém spektru výrobků. Bylo prakticky ověřeno, že spolupráce vědeckých ústavů vysokých škol a praxe je výborná a jediná možná metoda pro uplatnění kvalitních průmyslových vzorů.

TMZ642 CNC byl také motivací pro vznik projektu Intranet gridu, který je popsán a realizován v praktické části této práce. Pro simulace bylo nutné provádět velké množství rozsáhlých výpočtů – byla použita metodika Hardware-in-the-loop (HIL) pro návrh technických vlastností klíčových uzlů stroje. V praxi to znamenalo velké vytížení výpočtového serveru a citelné časové prodlevy mezi jednotlivými úlohami. V konstrukčních kancelářích přitom byla k dispozici velmi výkonná výpočetní technika v podobě grafických pracovních stanic, včetně instalovaných grafických akcelerátorů. Tato technika nebyla v nočních hodinách využita. Proto vznikla snaha navrhnout řešení, které by tento stav pomohlo zlepšit.

Vstupem byl původní projekt gridu, který řešil rozložení zátěže na serveru s aplikací ERP plánovacího systému. Následný požadavek pro vybudování Intranet gridu, jako univerzálního nástroje pro řešení obecných problémů však vyžadoval přehodnotit stávající, do jisté míry povrchní přístup. Stal se tak podnětem a obsahem této práce. Podmínkou praktické realizace bylo uplatnění osvědčených návrhových vzorů a knihoven, které výrazně usnadnily řešení a umožnily vybudování spolehlivé a obecně použitelné aplikace.

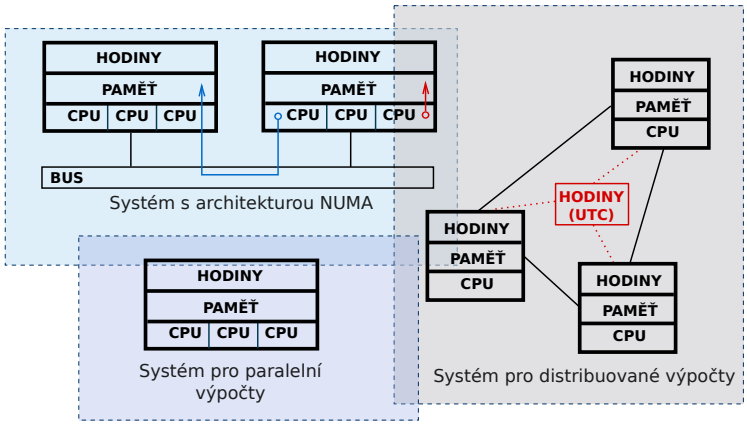
2 Charakteristika výpočetních systémů

Primární motivací architektury paralelních a distribuovaných systémů je snaha urychlit práci, a zvýšit efektivitu výpočetního systému. Každá z těchto technologií vychází ze stejné myšlenky využití souběžného zpracování výpočetního algoritmu. Mají však odlišnou topologii. Zatímco distribuované výpočty mohou probíhat v rozsáhlých sítích a geografických lokalitách, paralelní výpočty jsou směrovány na konkrétní výpočetní jednotku.

Právě toto měřítko je určuje k řešení rozdílných typů úloh, které se liší z hlediska toho, zda jednotlivé větve paralelního výpočtu musí být synchronní (v čase na sebe přesně navazující), nebo ne. Pro úlohy které vyžadují přesnou synchronizaci lze použít pouze paralelní výpočetní systém. Pro úlohy ostatní, kdy je synchronizace očekávána jen v rámci zpracování jednotlivých procesů (nikoliv vnitřních algoritmů) je vhodné nasazení distribuovaného výpočetního systému.

Paralelní výpočetní systém řeší oblast paralelních výpočtů a paralelních algoritmů, jejichž společným rysem je časová synchronizace v rámci algoritmu a paralelní běh více výpočetních jednotek nad jedním programem a společnou pamětí. Dle Flynnovy taxonomie patří do kategorie výpočetních systémů SIMD [74].

Paralelní systém s distribuovanou pamětí (Non-Uniform Memory Architecture, NUMA) je platforma, která dosahuje škálovatelnosti tak, že seskupuje procesory a operační paměť do jednotek. Ty pak mohou samostatně fungovat jako jeden počítač. Každý uzel má vlastní procesory, paměť a sběrnice. Synchronizace času na procesorech je řešena jedinou časovou základnou. Jednotlivé uzly mají z důvodu snadnějšího managementu shodné konstrukční prvky, které jsou vzájemně



Obrázek 2.1: Topologie systémů pro paralelní a distribuované výpočty

propojeny sběrnicí pomocí níž mohou přistupovat k paměti jiného uzlu. Koncepce této paralelní platformy vychází z architektury SMP - symetric multiprocessing, která řeší spolupráci více procesorů nad jednou pamětí na jediné základní desce počítače [65] [69].

Distribuovaný výpočetní systém využívá vysoké dostupnosti a propustnosti sítí WAN a využívá k paralelnímu zpracování počítače právě spojené prostřednictvím této sítě, tedy technologie Grid computingu. Hlavní charakteristikou je souběžné zpracování jednotlivých procesů, kde každý z nich využívá služeb jednoho z plnohodnotných výpočetních zdrojů. Každý z těchto zdrojů má vlastní paměť a vlastní výpočetní jednotku. Ve Flynnově taxonomii patří do kategorie MIMD [74].

S technologií Gridu také souvisí poměrně masivně se rozvíjející technologie Cloud computingu, kterou lze chápat jako nadstavbu Gridové služby. To znamená, že zatímco Grid těží z výpočetního výkonu satelitních počítačů propojených sítí WAN, u Cloudu jsou to satelitní počítače (nebo přesněji terminály), které těží z vysokého výkonu poskytovatele Cloudu, který je ovšem také rozprostřen v různých geografických destinacích propojených sítí WAN. Cloud je však na rozdíl od specializovaných výpočetních Gridů chápán jako platforma pro poskytování rozsáhlých a univerzálních služeb. Paralelní a distribuované systémy lze rozdělit dle této základní charakteristiky [48].

- **Paralelní systém se sdílenou pamětí** - jednotlivé procesory, nebo jádra procesorů mají přístup k jediné společné paměti. Systém provádí paralelní výpočty na úrovni paralelních programů a algoritmů. Časová posloupnost zpracování je řízena jedinými systémovými hodinami, proto existuje centrální synchronizace času.
- **Paralelní systém s distribuovanou pamětí** - jednotlivé výpočetní uzly mají přístup ke své paměti, ale mohou přistupovat také k paměti jiného uzlu. Systém provádí paralelní výpočty na úrovni paralelních programů a algoritmů. Časová posloupnost zpracování je řízena jedinými systémovými hodinami, proto existuje centrální synchronizace času. Přístup k paměti jiného uzlu může znamenat vyšší časové nároky na zpracování
- **Distribuovaný výpočetní systém** - každý procesor má vlastní paměť. Informace jsou vyměňovány formou zpráv. Paralelní zpracování probíhá na úrovni jednotlivých procesů dávkové úlohy. Systém nemá centrální synchronizaci času.

Srovnání technologií paralelních a distribuovaných výpočtů

Paralelní technologie využívají paralelní hardwarovou architekturu jednoho počítače. Proto jsou vhodné pro výpočty, u kterých je nutná časová synchronizace jednotlivých výpočtů. Výhodou je, že obvykle odpadá režie síťových služeb.

Úlohy, které nevyžadují přesnou synchronizaci úloh, například zpracování statistických dat nebo úlohy vyžadující velké množství I/O operací s periferními zařízeními a souborovými subsystémy jsou vhodné pro nasazení v prostředí distribuovaných výpočtů a Grid computingu. Rozsáhlá infrastruktura Gridu umí lépe rozložit zátěž I/O operací a periferních jednotek. Samozřejmě, i zde platí to, že se oba typy technologií vzájemně kombinují a doplňují.

3 Paralelní výpočetní systém

Snahou nasazení technologie paralelních výpočtů, je využití všech dostupných hardwarových zdrojů a snaha o co nejvyšší zužitkování výpočetního výkonu centrálního procesoru a periferních zařízení. Ty jsou určeny k řešení specializovaných úloh a umožňují ve své oblasti dodat mnohem vyšší výpočetní výkon, než vlastní CPU jednotka. Patří sem zejména grafické procesory různých výrobců (ATI, nVIDIA. a speciální signálové procesory (DSP). Společným rysem paralelních systémů je společná sdílená paměť.

Podmínkou pro řízení paralelního zpracování je synchronizace jednotlivých procesů. Přestože je synchronizace jednotlivých fází výpočtu nezbytná, snižuje celkovou výkonnost, protože se z pochopitelných důvodů musí čekat na nejdéle trvající výpočet. To znamená, že výkon neroste lineárně s počtem procesorových jednotek, ale asymptoticky se blíží k hranici, za níž již nemá smysl další rozšiřování paralelních výpočetních jednotek [13].

Projekty a knihovny pro podporu paralelních výpočtů

Pro podporu paralelních a distribuovaných výpočtů je k dispozici velké množství knihoven a vývojových prostředků, dostupných jako open source licence (OpenMP, OpenMPI) tak i komerční produkty například nVIDIA CUDA nebo DirectX.[1]

[1] I komerční produkty jsou volně k dispozici, nejsou však uvolněny zdrojové kódy knihoven. Za zmínku stojí špičkové vývojové prostředí nVIDIA Nsight na bázi Eclipse.

nVIDIA CUDA

Počáteční impuls dala karta nVIDIA GeForce 8800, která definovala obecný shader, který obešel nevyváženost zátěže mezi klasickým bod v prostoru (VERTEX) a picture element, obrazový prvek (PIXEL) shaderem. Tím byla otevřena cesta k obecnému využití shaderů i pro jiné úlohy, než jsou grafické výpočty. Vývoj aplikací pro CUDA je prováděn v CUDA frameworku, který poskytuje potřebné knihovny. Technologie CUDA je velmi dobře propracována. Nevýhodou je zaměření pouze na grafické akcelerátory nVIDIA [77].

DirectX

Technologie firmy Microsoft, která na rozdíl od CUDA a ATI Streamu není úzce vázána na konkrétní hardware, ovšem je zásadně omezena na proprietární technologie zmíněné firmy. To znamená, že vývoj lze provádět pouze na systémech Windows.

OpenCL

Je framework pro využití heterogenních zdrojů výpočetního systému [43]. Byl navržen tak, aby byl schopen zapojit do výpočtu jakýkoliv procesor, který je v systému k dispozici. Obvykle využívá možností více-jádrových procesorů a jader grafických karet. Výhodou je nezávislost na konkrétní konfiguraci. To znamená, že aplikace vyhovující standardu OpenCL je možno ladit a spouštět například na počítači bez speciální grafické karty nebo více-jádrového procesoru. To má například výhodu v tom, že vývoj s pomocí této technologie je přístupný komukoliv, bez nutnosti speciálního hardwarového vybavení.

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. Aktuální je verze 3.1, která byla jako kombinovaná pro jazyky C/C++/FORTRAN uvolněna v roce 2011 [25].

OpenMPI

Message Passing Interface (dále jen MPI) je knihovna implementující stejnojmennou specifikaci (protokol) pro podporu paralelního řešení výpočetních problémů v počítačových clusterech. Konkrétně se jedná o rozhraní pro vývoj aplikací (API) založené na zasílání zpráv mezi jednotlivými uzly. Jedná se jak o zprávy typu point-to-point, tak o globální operace. Knihovna podporuje jak architektury se sdílenou pamětí, tak s pamětí distribuovanou (dnes častější). Z pohledu referenčního modelu ISO/OSI je protokol posazen do páté, tedy relační vrstvy, přičemž většina implementací používá jako transportní protokol TCP.

Toto API je nezávislé na programovacím jazyce, neboť se jedná především o síťový protokol. Nejčastěji se však setkáme s implementací v C, C++, Javě, Pythonu nebo Fortranu. Výjimkou není ani podpora přímo na úrovni hardwaru. Při návrhu celého rozhraní i při jeho implementaci byl vždy kladen důraz především na výkon, škálovatelnost a přenositelnost. K nevýhodám, ale zároveň také výhodám této knihovny patří její nízkoúrovňový přístup. Nehodí se tedy pro rychlý vývoj aplikací (RAD), ale spíše pro aplikace, kde je rozhodující rychlost běhu aplikace, což je ale pro paralelní systémy typické. To je i možná důvodem, proč se stala v této oblasti de-facto standardem. Ke standardizačnímu řízení však zatím nedošlo. Technologie Open MPI zahrnuje podporu paralelních i distribuovaných systémů [82].

Metodika algoritmů paralelních výpočtů

Motivací pro použití paralelismu v oblasti výpočtů je obecně snaha o výrazné zvýšení dosažitelnosti výsledku. Je nutné mít na paměti, že neplatí lineární závislost nárůstu výkonu v závislosti na počtu paralelních větví. Lineárnímu nárůstu výkonu brání další režie, které jsou nezbytné pro synchronizaci výpočtů v paralelním

režimu. Navíc každá úloha obsahuje sekvenční dobu běhu programu, to znamená části programu, které nelze zpracovávat paralelně (například inicializace proměnných, semaforů alokace paměti) a části programu, které paralelně zpracovat lze. Vývoj paralelních algoritmů lze chápat jako optimalizaci.

V prvé fázi se obvykle navrhne standardní sekvenční algoritmus, například rychlá Fourierova transformace, nebo metoda konečných prvků a jiné. V této fázi je výslovně na škodu provádět jakékoliv optimalizace. V tuto chvíli jde o bezchybný, velmi dobře čitelný kód, který lze zpracovat sekvenčními metodami výpočtu.

Máme-li funkční základní sekvenční kód, můžeme přistoupit k optimalizaci. Nejprve je třeba maximálně vyladit sekvenční kód, a teprve následně přistoupit k vyhledání částí kódu, které se vyplatí podrobit paralelnímu zpracování. Nejčastěji se využívá modelu, Task/Channel, který byl navržen Ianem Fosterem a reprezentuje paralelní výpočet jako množinu úloh, které mezi sebou komunikují pomocí komunikačních kanálů [33].

Využití grafické karty pro paralelní výpočty v prostředí Gridu

Současná nabídka grafických akcelerátorů se vyznačuje dobrou podporou pro použití v oblasti rozsáhlých paralelních výpočtů. Obecně známé a dnes i poměrně rozšířené technologie, například nVIDIA CUDA nebo OpenCL, poskytují vývojáři velmi dobrý nástroj pro návrh kvalitních a výkonných aplikací, založených na datovém paralelismu. V současné době je však v provozu také velké množství poměrně velmi výkonné výpočetní techniky, která však ještě nemá instalovány takové grafické akcelerátory, které výše uvedené technologie podporují. Přesto existuje cesta, jak i starší typy GPU přinutit k poskytnutí poměrně masivní paralelní výpočetní platformy. Jsou zde sice jistá omezení, zejména v oblasti přesnosti výpočtu nebo možnosti řízení a synchronizace paralelních větví, ale v oblastech výpočtu rozsáhlých polí nebo iterací lze tohoto přístupu úspěšně využít.

Při návrhu se vycházelo z předpokladu, že v systému pro distribuované výpočty - Gridu, mohou být připojeny velmi různorodé klientské stanice. Hlavním záměrem proto bylo zvolit takovou technologii, která bude podporována na více hardwarových a softwarových platformách.

Průmyslový standard OpenGL, jehož vývoj řídí konsorcium Khronos Group, dává záruku v oblasti kompatibility a přenositelnosti mezi různorodými platformami. Proto byl zvolen jako jedna z alternativ. Ačkoliv tento standard primárně není určen pro oblast paralelních výpočtů, lze aplikace využívající tuto technologii s jistým omezením využít. Výhodou je možnost využít i poměrně staré grafické karty, které se v běžném provozu stále používají.

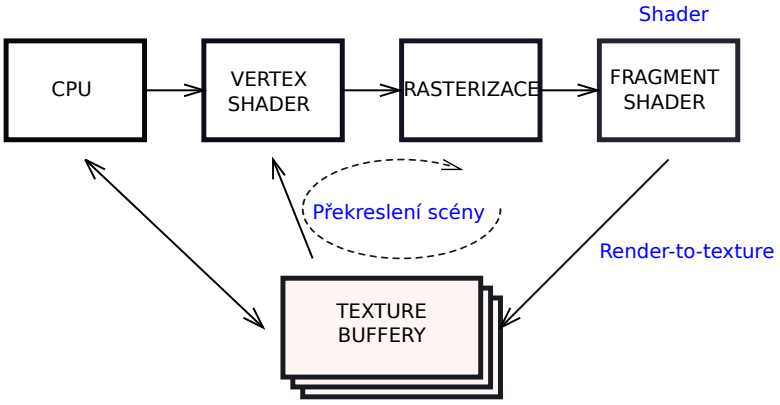
Princip výpočtu je založen na využití základní funkční vlastnosti grafické karty. Tu si lze představit jako velmi kvalitní převodník bodů (vertexů), které tvoří vrcholy 3D scény na 2D matici různě barevných bodů – texelů, (texture element nebo texture pixel). Ty vytvářejí virtuální scénu plastického 3D obrazu [2].

Obvody grafického akcelerátoru jsou proto navrženy tak, že obsluhují dvě základní oblasti. V prvé řadě se jedná o algoritmy, které řeší různé geometrické trans-

[2] princip hry barev a stínů, které vytvářejí dojem plastického obrazu byl znám malířským mistrům již ve středověku

formace nad VERTEXy scény (vertex = bod se souřadnicemi [x,y,z]) a v druhé řadě algoritmy, které řeší vlastní rasterizaci 3D scény do barevné 2D textury. Vzhledem k tomu, že jsou tyto algoritmy spouštěny nad každým bodem 3D modelu, je grafická karta již od prvopočátku konstruována jako velmi výkonná paralelní výpočetní jednotka. Z toho také plynou jisté, výše popsané nevýhody. Tyto nevýhody se projeví zejména u grafických karet starších generací, kdy požadovaná rychlost algoritmu jde na úkor přesnosti výpočtu. Ta však pro rasterizaci grafické scény není zásadní.

Navíc starší typy GPU akcelerátorů nemají implementován paměťový typ double. Nové GPU však již typ double podporují. Ten byl implementován i do OpenCL 1.2, CUDA 1.3 a OpenGL 4.0 [99].



Obrázek 3.1: Koncept grafické pipeline

Pro speciální efekty grafické scény (například pro mlhu nebo různé fluidní procesy, pro virtuální realitu ohně nebo pohybující se vodní hladinu) poskytli výrobci grafických karet vývojářům nástroj, který umožňuje jistými prostředky zasáhnout do výpočetního řetězce grafické karty. A právě zde se otevírá možnost využití grafické karty nikoliv jako grafického akcelerátoru, ale jako velmi výkonné paralelní výpočetní jednotky. Tato podpora je přímou součástí grafických knihoven OpenGL nebo DirectX. V praxi představuje vývojářské rozšíření, které umožňuje psát programy – shadery [45]. Ty jsou schopny ovlivnit řízení zobrazovacího řetězce grafické karty. Standardně jsou podporovány VERTEX, PIXEL a geometry shadery. Nové grafické karty spolu s posledními verzemi OpenGL a DirectX podporují shadery pro realizaci z angl. tesellation, mozaikování, parketování (TESELACE).

Pro paralelní výpočty lze právě tyto shadery využít. Shadery mohou zasahovat do řízení jak v oblasti 3D geometrie scény, tak i v oblasti rasterizace 3D scény na 2D matici texelů. Při návrhu paralelního algoritmu programu v OpenGL nebo DirectX je nutno přistoupit k problému podobně jako k návrhu standardní grafické aplikace. Je tedy nutné počítat s tím, že prostředí OpenGL nebo DirectX neposkytuje nástroje pro synchronizaci paralelních větví programu. To znamená, že současně mohou být prováděny jen opravdu nezávislé části výpočtu.

Technologie OpenCL i CUDA již mají implementovánu synchronizaci mezi tzv. workgroups, (lokální paměťové bloky), takže návrh paralelního kódu výrazně usnadňují a poskytují mnohem větší možnosti [8] [77].

Ovšem i přes řadu nevýhod a omezení využití OpenGL v oblasti paralelních výpočtů, má své opodstatnění. Lze docílit velmi dobrých výsledků, zejména u algoritmů, které vykazují jistou symetrii v návrhu (např. FFT nebo manipulace s maticemi). Druhým přínosem je, že zkušenost s touto poměrně nízkoúrovňovou technologii přináší jistou konkurenční výhodu při implementaci a použití technologií nových, například již zmíněné OpenCL nebo CUDA [3].

Princip výpočtu na grafické kartě

Shader [55] je program určený pro běh na výpočetní jednotce grafické karty. Syntaxe zdrojového kódu shaderu je závislá na použité technologii. Pro OpenGL je určen jazyk OpenGL Shading Language (GLSL), (pro DirectX, High Level Shading Language (HLSL)). Ty mají velmi podobnou (prakticky totožnou) syntaxi s jazykem C [8] [77]. Překladač i linker je součástí OpenGL knihovny. GLSL ani HLSL však nemají implementovánu dynamickou alokaci paměti. Z toho vyplývá nemožnost použití pointerů a rekurze. Shadery lze nasadit v různých částech grafické scény. Z toho také vyplývá jejich speciální zaměření.

[3] Bylo prakticky ověřeno, že technologie OpenGL dávala nejlepší výsledky z hlediska rychlosti výpočtu. Viz kapitola 10.2.2, graf 10.2.5

Vertex shader

Je programovatelná jednotka nad daty vstupních VERTEXů [55] které prostorově popisují 3D grafickou scénu zobrazovaného objektu. Vertex shader je program, který se provede na každém vrcholu (vertexu) vstupní geometrie scény. To znamená, že je zodpovědný za afinní transformace (rotace, posun, zoom) vstupní 3D scény. Programátorovi 3D grafické scény dává možnosti vytváření virtuální reality například vodní hladiny a podobných efektů. Vertex procesor nemá možnost přidávat nebo odebírat jednotlivé vrcholy grafické scény. To v praxi znamená, že shadery nad VERTEXy vždy pracují s předem danou množinou 3D geometrických bodů, kterou neumějí ani zvětšit ani redukovat. Neumějí ovlivnit výslednou geometrii [76] [45].

Geometry shader

Je programovatelná jednotka, která pracuje nad daty vstupních VERTEXů, které prostorově popisují 3D geometrii grafické scény. Program spuštěný na této jednotce, tzv. geometry shader umožňuje přidávat nebo odebírat vrcholy, čímž umí ovlivnit výslednou geometrii. Programátor má takto možnost vytvářet speciální efekty grafické scény. Geometry procesor má podobnou funkci jako VERTEX procesor, ale tím, že umí přidávat nebo odebírat vrcholy z grafické scény, má více možností [55] [45].

Fragment shader

Je programovatelná jednotka, která pracuje nad jednotlivými fragmenty (pixely) [55]. Fragment (nebo také pixel) shader je prováděn na každém PIXELu rasterizované scény. To znamená, že již pracuje nad rasterizovanou maticí 2D zobrazované scény. Využívá se výhradně k aplikaci různých textur nebo programové modifikaci barvy PIXELu. Fragment procesor neumí změnit (x,y) pozici jednotlivých fragmentů. Přístup k vedlejším fragmentům v rámci vykonávání fragment shaderu není povolen.

Shadery pro tesselaci

Grafické knihovny od verze OpenGL 3.2 nebo Direct3D 11 byly doplněny o tesselaci grafického řetězce (pravidelné opakování grafických struktur – dlaždice). Teselace má tři stupně, přičemž dva z nich jsou programovatelné. Tesselační shadery umožňuji podobně jako geometry shadery měnit geometrii zobrazované scény. Dávají však programátorovi další možnosti.

Pro paralelní výpočty na GPU je výhodný především fragment procesor a v některých případech i VERTEX procesor [45]. Ostatní programovatelné jednotky jsou pro tuto oblast méně významné. Fragment procesor, který je konstrukčně umístěn na konci grafického zobrazovacího řetězce již pracuje se 2D rasterizovaným obrazem. Ten je následně zobrazován na výstupním zařízení. Tento procesor je také využíván k aplikaci 2D textury na zobrazovanou grafickou scénu. Právě tato vlastnost je klíčová při použití grafického akcelerátoru v oblasti paralelních výpočtů na GPU.

K aplikaci, která má využít GPU k paralelnímu výpočtu musíme přistoupit jako ke standardnímu grafickému programu s jediným rozdílem, že výsledek výpočtu nemusí (ale může) být následně zobrazen na výstupním zařízení. Vlastní kód aplikace může budit dojem, že je do jisté míry komplikovaný. Ale v praxi je tvořen obecnou kostrou, kterou lze poměrně snadno modifikovat v částech, které se týkají konkrétního řešeného problému, přičemž části kódu, které souvisejí s nastavením vlastností grafického prostředí obvykle zůstávají beze změny

Koncept výpočtu na grafické kartě

Na straně CPU obvykle hovoříme o poli dat, které je alokováno v paměti RAM. Na straně GPU potřebujeme definovat vlastnosti textury, do níž budou data umístěna. Problém nastává v okamžiku, kdy se rozhodujeme napsat aplikaci tak, aby byla co nejvíce přenositelná. V praxi to znamená volbu textury vhodných vlastností. OpenGL nabízí dvě možnosti GL_TEXTURE_2D, což je standardní dvojrozměrná struktura, nebo GL_TEXTURE_RECTANGLE_ARB což je textura, která nemá charakter 2D pole, a která je právě pro aplikace bez grafického výstupu vhodnější. Další důležitou volbou je formát textury. OpenGL nabízí opět dvě

možnosti GL_LUMINANCE a GL_RGBA. GL_LUMINANCE představuje jednodušší formát textury, který mapuje jednu float hodnotu pro jeden texel. (Texel = bod ve 2D rasterizované scéně). GL_LUMINANCE textura spotřebuje na jeden texel 32 bitů (4 bajty) paměti. GL_RGBA textura představuje paralelní (čtyř vektorovou) strukturu, která spotřebuje na jeden texel 4 x 32 bitů paměti. Na základě těchto voleb je také následně nutné přistoupit k programovému kódu shaderu [45].

Mapování vektoru

Vektor o délce N je mapován do textury o rozměru $sqrt(N)$, (v případě, že použijeme formát textury GL_LUMINANCE) nebo $sqrt(N/4)$ v případě formátu GL_RGBA. Optimální je, když je vektor alokován v rozměru 2^n.

Příklad 1.

```
/**   GPU - Get system parameters
 *         1. Open GL version
 *         2. Is system ready for the GPU computing?
 *         3. Dim the FB[tex-size][tex_size]
 */
void gIter::initGL(int argc, char **argv) {
  int max_texsize;
  glutInit(&argc, argv);
  id_window = glutCreateWindow(" GLSL Iteration ");
  int errno = glewInit();

  if (errno != GLEW_OK) {    //ready?
    std::cout << "Error !!!!" << (char*) glewGetErrorString(errno)
        << std::endl;
    exit(-1);
  }

  glGetIntegerv(GL_MAX_TEXTURE_SIZE, &max_texsize);
  std::cout << "Max.size of the texture:" << max_texsize << std::endl;

  if (tex.texFormat == GL_RGBA) {     //Dim of the texture
    tex_size = (int)(sqrt(n / 4));
  } else {
    tex_size = (int)(sqrt(n));
  }

  if (tex_size > max_texsize) {
    Error !!!! - system exit....
  }
```

```
}
```

src/mapovani_vektoru.cpp

Nastavení viewportu pro mapování pixel/texel geometrie

Viewport je 2D obdélník v němž je virtuálně zobrazována 3D scéna. Pro usnadnění situace je nutné zvolit takové grafické nastavení viewportu, které bude 1:1 mapovat PIXELy (popisují 3D scénu, a budou renderovány) a texely (výsledky renderingu), které budou následně interpretovány jako výsledek výpočtu na GPU.

Příklad 2.

```
/**
 *  GPU - FrameBuffer and  Texture init
 *        Dimension of the  Frame Buffer[tex-size][tex_size]-main area for the
 *        GPU computing.
 */
void gIter::initGLFBO(void) {
  // Create FBO (off-screen framebuffer)
  glGenFramebuffersEXT(1, &id_framebuffer);
  // Framebuffer binding
  glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, id_framebuffer);
  // viewport 1:1 -> pixel=texel
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(0.0, tex_size, 0.0, tex_size, -1, 1);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  //std::cout << "Texture size:" << tex_size << " n:" << n << std::endl;
  glViewport(0, 0, tex_size, tex_size);

}
```

src/viewport.cpp

Shader

Programovatelná část GPU obsahuje několik (záleží na typu grafické karty) paralelních výpočetních jednotek. Nad těmito jednotkami nemáme žádnou kontrolu, takže neumíme zajistit pořadí vykonávání vlastního výpočtu. Pro GPU výpočty je nutné přinutit texturu k tomu, aby byla použitelná nejen pro vstup dat (což

je pro grafické aplikace obvyklé), ale také pro výstup vypočtených výsledků. Textura může být definována buď jako „read-ony“ nebo „write-only“. To vyplývá z data-stream (SIMD) architektury GPU. V praxi to znamená, že GPU spouští renderovací úlohy paralelně na jednotlivých stream procesorech. Pokud bychom na této architektuře definovali „read-modify-write“ [45] textury, dostali bychom se do problémů s logickou synchronizací R-W operací. To by mělo vliv na snížení propustnosti grafického řetězce, a tedy snížení výpočetního výkonu. Textura a její vlastnosti (read nebo write) je připojena do framebufferu knihovní funkcí *glFramebufferTexture2DEXT()* v níž jsou definovány její vlastnosti [86].

Jednotka GPU v roli výpočetního zdroje

Pro test využití grafického akcelerátoru k paralelním výpočtům bylo zvoleno řešení velmi jednoduché transcendentní rovnice $cos(x)x = 0;$, kterou lze řešit iterací. K řešení na CPU si vystačíme s triviálním kódem:

Require: $n \geq 0$
1: **while** $n \neq 0$ **do**
2: $X \leftarrow iterace$
3: $n \leftarrow n - 1$
4: **end while**

Algoritmus 1: Výpočet transcendentní rovnice $cos(x)x = 0$

Pro změření a porovnání vlastností výpočtu na GPU v porovnání s vlastnostmi výpočtu na jednotce CPU bylo zvoleno řešení vektoru $[0, \ldots, n]$ transcendentních rovnic $cos(x)x = 0$. V prvé fázi běžel tento algoritmus pouze v režii CPU. Tato část je znázorněna v algoritmu 2.

Pro porovnání vlastností výpočtu na GPU byl kód upraven tak, aby smyčka vektoru $[0, \ldots, n]$ běžela v režii CPU a grafická jednotka GPU, řešila iterační výpočet $cos(x)x = 0$. To znamená, že pro výpočet na GPU byl kód modifikován dle algoritmu 3.

Algoritmus pro porovnání výpočtů na CPU a GPU byl takto navržen proto, aby mohl být následně využit při vyhodnocení kapacity výpočetního zdroje (popsáno v kapitole 9.2.6) , bez vlivu na to, zda je k dispozici grafická výpočetní jednotka.

Require: $n \geq 0$
Require: $m \geq 0$
1: **while** $n \neq 0$ **do** ▷ Vektor $[0,\ldots,n]$ v režii CPU
2: **while** $m \neq 0$ **do** ▷ Iterace $cos(x)x = 0$ v režii CPU
3: $X \leftarrow iterace$
4: $m \leftarrow m-1$
5: **end while**
6: $n \leftarrow n-1$
7: **end while**

Algoritmus 2: Vektor$[0,\ldots,n]$ rovnic $cos(x)x = 0$ v režii CPU

Require: $n \geq 0$
1: **while** $n \neq 0$ **do** ▷ Vektor $[0,\ldots,n]$ v režii CPU
2: $call < GPU() > \leftarrow iterace$ ▷ Iterace $cos(x)x = 0$ v režii GPU
3: $n \leftarrow n-1$
4: **end while**

Algoritmus 3: Vektor $[0,\ldots,n]$ rovnic $cos(x)x = 0$ na GPU

Kód shaderu pro iterační výpočet $cos(x)x = 0$ vypadá následovně. (Pravidla a syntaxe pro psaní shaderů viz: [55])

Příklad 3.

```
const char * glsl_src =
  "#extension GL_ARB_texture_rectangle : enable              \n"
  "uniform sampler2DRect textureY;                           \n"
  "uniform sampler2DRect textureX;                           \n"

  "void main (void) {                                        \n"
  "   vec4 y = texture2DRect(textureY, gl_TexCoord[0].st);   \n"
  "   vec4 x = texture2DRect(textureX, gl_TexCoord[0].st);   \n"
  "   gl_FragColor = cos(y * x);                             \n"
  "}";
```

src/shader.cpp

Shader je vykonáván na každém texelu rasterizovaného obrazce. Jeho úkolem je tedy řešit výše zmíněnou transcendentní rovnici, tentokrát však na GPU. Tento program je v průběhu inicializace hlavního programu kompilován a linkován. Následně je připraven k použití. Inicializace shaderu je znázorněna v níže uvedeném fragmentu kódu.

Příklad 4.

```
/**
 * GPU - GLSL init
 */
void gIter::initGLSLProgram(void) {

        //Program object
  glsl_program = glCreateProgram();
  fragment_shader = glCreateShader(GL_FRAGMENT_SHADER_ARB); //fragment shader

  //Get the source code of the  GLSL program
  const GLchar* src = tex.shader_src;
  glShaderSource(fragment_shader, 1, &src, NULL);

  //Compile to the metacode
  glCompileShader(fragment_shader); //kompilace do metakodu
  shaderLog(fragment_shader); //log
  glAttachShader(glsl_program, fragment_shader); //pripojeni shaderu k programu

  //Link
  glLinkProgram(glsl_program); //link
```

src/init_shader.cpp

Provedení výpočtu na GPU – vykreslení scény

V prvé řadě je nutné připravit datovou oblast na straně CPU a tu poskytnout GPU. To lze provést následovně.

Příklad 5.

```
/**
 * GPU - Data to the texture
 */
data_y = (float*) ::operator new(n * sizeof(float));
.
.
.
void gIter::setTexture(float* data, GLuint texture_id) {
  glBindTexture(tex.texTarget, texture_id);
  glTexSubImage2D(tex.texTarget, 0, 0, 0, tex_size, tex_size, tex.texFormat, GL_FLOAT,
     data);
}
```

src/data_to_shader.cpp

Je také nutné pamatovat na to, že alokace a přenos dat ze strany CPU do GPU patří k nejpomalejším částem výpočetního řetězce. Proto je velmi dobré promyslet a zvážit velikosti paměťových struktur. Obecně platí čím menší a častěji předávaná paměťová oblast, tím horší výpočetní výkon aplikace [45].

Vlastní výpočet na GPU probíhá tak, že vykreslíme obdélník rasterizované scény pomocí následujícího fragmentu kódu. Tím zároveň donutíme GPU, aby vykonala kód shaderu na každém texelu grafické scény. Vlastní paralelní výpočet proběhne mezi body *glBegin()* a *glEnd()*.

Příklad 6.

```
/**
 * GPU - mapping the Y and X texture with the input data
 *       and perform of the calculation
 */
void gIter::gpuComputeFactory() {
  // Setting the color buffer to the GL_COLOR_ATTACHMENT0_EXT value
  glDrawBuffer(attach_point[write_tex_y]);
  // mapping  the id_tex_y[read-tex] texture
  glActiveTexture(GL_TEXTURE0);
  glBindTexture(tex.texTarget, id_tex_y[read_tex_y]);
  glUniform1i(gl_textureY, 0);

  // mapping the  id_tex_x[0] texture
  glActiveTexture(GL_TEXTURE1);
  glBindTexture(tex.texTarget, id_tex_x[read_tex_x]);
  glUniform1i(gl_textureX, 1);

  //setting the rasterization parametrers (GL_FRONT, GL_FILL is default)
  glPolygonMode(GL_FRONT, GL_FILL);

  //rendering of the GL scene (main section to GPU computing)
  glBegin(GL_QUADS);
  glTexCoord2f(0.0, 0.0);
  glVertex2f(0.0, 0.0);
  glTexCoord2f(tex_size, 0.0);
  glVertex2f(tex_size, 0.0);
  glTexCoord2f(tex_size, tex_size);
  glVertex2f(tex_size, tex_size);
  glTexCoord2f(0.0, tex_size);
  glVertex2f(0.0, tex_size);
  glEnd();
  ping_pong();
}
```

src/exec_shader.cpp

Výsledek je k dispozici ve výstupní textuře, kterou je možné vyčíst z paměti RAM na CPU straně pomocí funkce.

Příklad 7.

```
/**
 * GPU - get result from the texture
 */
void gIter::getTexture(float* data) {
  glReadBuffer(attach_point[read_tex_y]);
  glReadPixels(0, 0, tex_size, tex_size, tex.texFormat, GL_FLOAT, data);
}
```

src/data_from_shader.cpp

Ping Pong metoda

Je technika používaná při iteračních výpočtech na GPU. Každý výpočet na GPU požaduje zdrojovou texturu a cílovou texturu. Nelze data číst ze zdrojové textury a zapisovat znovu do zdrojové textury. Tato metoda tento problém řeší. Výhodou je, že data výstupního vektoru výpočtu nemusí být znovu přenesena do vstupního vektoru [45].

Princip Ping Pong metody: V programu jsou definovány dvě textury, textura(read) a textura(write), které si po jedné iteraci svoji roli vymění. Ve skutečnosti je to vyřešeno tak, že nad jediným paměťovým prostorem je definováno více připojovacích bodů (attach pointů) do jediné textury a ping pong metoda pouze přepíná mezi těmito body a určuje který z nich bude read nebo write. Toto omezení grafického akcelerátoru vyplývá z jeho koncepce a taxonomického začlenění do kategorie SIMD [74] [45], tedy do data-stream kategorie. Ping pong techniku řeší triviální rutina, která je uvedena v příkladu 8 [45].

Příklad 8.

```
/**
 * GPU - pingpong R->W; W->R
 */
void ping_pong(void) {
  if (write_tex_y == 0) {
    write_tex_y = 1; read_tex_y = 0;
  } else {
    write_tex_y = 0; read_tex_y = 1;
  }
}
```

src/exec_shader.cpp

Java interface pro výpočty na grafické kartě

Java poskytuje velmi dobrou podporu pro aplikace distribuovaných výpočtů v prostředí Gridu [42]. Neposkytuje však podporu pro výpočty na grafické kartě s podporou OpenGL. K využití této technologie v prostředí Javy je nutný mezičlánek, vybudovaný pomocí JNI-api, které umožňuje integrovat volání nativního kódu z dynamické knihovny. Bohužel tato technologie s sebou přináší jistá omezení a rizika spojená s kvalitou návrhu dynamické knihovny. Například memory leak na straně nativního kódu vždy způsobí pád celého Java Virtual Machine. To v produktivním prostředí může představovat poměrně vážný problém. Proto musí být veškeré funkcionality na straně dynamické knihovny velmi dobře analyzovány a odladěny [66].

Druhou poměrně závažnou nevýhodou je, že nativní kód je plně závislý na operačním systému. Tím ztrácíme nezávislost a plnou přenositelnost aplikace, kterou Java primárně poskytuje. Úkolem interface je částečně tuto nevýhodu odstínit. V prostředí Gridu se předpokládá, že jeho klient může být instalován na značně rozdílných technologiích a operačních systémech, navíc s velmi různou podporou GPGPU výpočtů. Je nutno také počítat s alternativou, že některé technologie výpočty na grafické kartě vůbec nepodporují. Pokud má být výpočet efektivní, musí serverová strana Gridu poskytnout úkol v takové podobě, aby prostředky, kterými klientská strana disponuje, byly k požadovanému úkolu optimálně využity.

Java interface musí stranu serveru požádat o takovou podporu výpočtu, která bude na straně klienta podporována. Obecně lze předpokládat, že na straně kli-

enta může být instalována některá z paralelních technologií (OpenMP, OpenCL, OpenGL nebo CUDA.) Úkolem je otestovat možnosti vzdáleného systému a poskytnout adekvátní prostředky.

V případě, že na hostovaném systému není k dispozici žádná z těchto technologií (například OpenMP, OpenCL, OpenGL nebo CUDA), musí serverová strana poskytnout standardní kód programovacího jazyka Java, který úlohu vyřeší jen s využitím výpočetní jednotky CPU.

JNI-api poskytuje nástroje i rozhraní pro spuštění nativního kódu. Zde jsou nativní funkce implementovány v samostatných knihovních funkcích obvykle psaných v C nebo C++. (C++ poskytuje jednodušší rozhraní s JNI.) Metoda JNI může vypadat například takto:

Příklad 9.

```
/**
 * C++ native function
 *     JNIEnv *env - JNI environment
 *     jobject thisObj - this java
 *     jint ji1 - input parameter 1
 *     jint ji2 - input parameter 2
 *     jcharArray input parameter 3
 */

JNIEXPORT void JNICALL Java_jGLSLiter_iter (JNIEnv *env,
                                            jobject thisObj,
                                            jint ji1,
                                            jint ji2,
                                            jcharArray jc) {

  int iter = ji1;
  int delka_n = ji2;
  char * c = (char*) jc;

  gIter g = gIter(iter, delka_n, *c);
```

src/jni.cpp

Ukazatel JNIenv v příkladu 9 je struktura, která obsahuje rozhraní k Java Virtual Machine (JVM). To obsahuje všechny funkce nezbytné pro interakci s JVM. Jobject je reference na *"this"* [4] Java objekt [66].

[4] odkaz objektu na sebe sama

4 Distribuovaný výpočetní systém

Systémy pro distribuované výpočty jsou technologie, které spojují tisíce běžných počítačů různého výkonu s různými architekturami a operačními systémy1 do jednoho výpočetního clusteru. Jedná se o velmi heterogenní strukturu, která je navíc propojena heterogenní sítí. Velkou výhodou Gridu je vysoká pružnost při komunikaci s jednotlivými satelity. Prakticky nevyžaduje trvalou konektivitu s jednotlivými uzly, ale je naopak konstruována tak, že umí pružně reagovat na nabízenou výpočetní kapacitu.

Základní myšlenkou pro konstrukci distribuovaného systému jsou technologie *RPC (Remote Procedure Call) a objektově orientovaná technologie CORBA (Common Object Request Broker Architecture)*
citefoster-1998 [32], které byly definovány na standard UNIX application interface a staly se součástí většiny známých operačních systémů. V systému Windows je podporován vlastní přístup s pomoci COM.

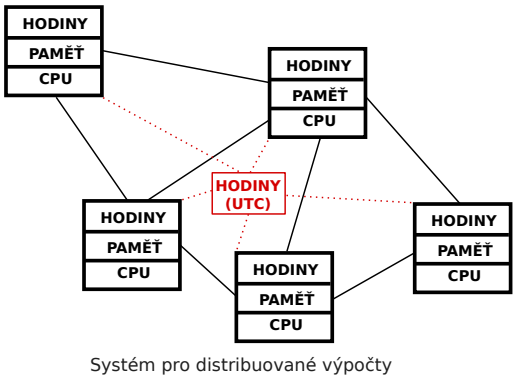
Filozofie distribuovaných systémů může být různá. Pojetí správy prostředků může být velmi striktní, vágní nebo vůbec žádné.

Například gridová služba BOINC Edu se vůbec nezabývá dostupnými prostředky. Pomocí nezabezpečené peer to peer komunikace pověřuje jednotlivé výpočetní uzly drobnými úkoly. Dokonce se ani nezabývá tím, zda se výsledek výpočtu vrátí v pořádku zadavateli. S výhodou využívá vysoké redundance, přičemž jediný výpočet bez problémů pošle více zpracovatelům. Může si to dovolit. Například v projektu SETI@home, který je zastřešen službou BOINC je zapojeno více jak *milion(!)* domácích počítačů [2].

Druhou větví ve filozofii je naopak velmi propracovaná infrastruktura, která umožňuje sdílet kapacity a výpočetní jednotky v rámci jediné nebo více organizací soustředěných do tzv. Virtualní organizace [31], kde je možno zajistit koordinaci

celé sítě spojené do gridu, která je řízena jediným zadavatelem výpočetních úloh. Takto je konstruován například Grid v CERNu v Ženevě na projektu LHC, který jen pro představu výkonu zpracovává *1,5 GB* dat každou sekundu [18].

Distribuované systémy a výpočetní gridy obecně zpracovávají dávkové úlohy, které jsou členěny do jednotlivých procesů. Ty jsou rozesílány jednotlivým výpočetním zdrojům. Synchronizace času v prostředí Gridu probíhá pomocí zasílání zpráv. Jedná se o komplexní problém, který se musí vyrovnat nejen s geografickým rozložením výpočetních zdrojů Gridu, ale také s konečnou rychlostí komunikace v rámci síťových služeb [58] [64].



Obrázek 4.1: Topologie distribuovaného systému

Na obrázku 4.1 je znázorněna topologie distribuovaného systému, kterou lze popsat jako orientovaný graf, jehož vrcholy představují procesory (výpočetní zdroje) a hrany jsou jednosměrné komunikační kanály. Každý z procesorů představuje samostatnou výpočetní jednotku s vlastní pamětí. Distribuované systémy nesdílejí žádnou společnou globální paměť. Každý procesor je spojen prostřednictvím komunikační sítě, která zajišťuje výměnu informací mezi jednotlivými procesory. Systém pro výměnu zpráv (broadcasting) se musí vyrovnat s řadou omezení, která plynou z topologické rozlehlosti distribuovaných systémů. Jedná se především o neexistenci absolutních globálních hodin, které jsou přesně synchronizovány pro všechny účastníky distribuovaného systému a také z omezení rychlosti výměny informací. Komunikační zpoždění je konečné a nepředvídatelné. To následně může představovat riziko časového zpoždění (timeout), ztrátu nebo nečitelnost zprávy.

Systémy výměny zpráv a systémy obnovy po poruše představují důležitou komponentu každé distribuované nebo gridové služby [58] [64].

Projekty a knihovny pro podporu distribuovaných výpočtů

ANTARES	Analýza dat z podvodního detektoru ve Středozemním moři.
BOINC	Nad projektem BOINC je vybudován například SETI@home, které se věnuje výzkumu mimozemských civilizací, Milky-Way@Home - hvězdná kinematika, Einstein@home - gravitace
CESNET	Sdružení založené vysokými školami a Akademií věd České republiky
CODESA 3D	Simulace fyzických procesů ovlivňujících toky spodní vody v hydrologii
EGEODE	Zpracování seismických dat pro zkoumání zemských vrstev a zjišťování zásob ropy a zemního plynu
E-GRID	Finanční simulace
GATE	Simulace reakce tkáně při ozařování
GPS@	Proteinová databáze a prostředky pro analýzu při výzkumu již známých genomů
GOME	Analýza dat a snímků z družic pro výzkum ozónu a ropných skvrn
LHC	Analýza a ukládání dat ze čtyř detektorů urychlovače v CERNu
MAGIC	Simulace kosmických paprsků z teleskopu MAGIC na Kanárských ostrovech
SiMRI3D	Simulace fyziky trojrozměrných zobrazovacích systémů magnetické rezonance
SPLATCHE	Simulace rozprostření osob v zeměpisných oblastech pro výzkum genetického vývoje
WISDOM	Analýza chemických složek při výzkumu nových léků

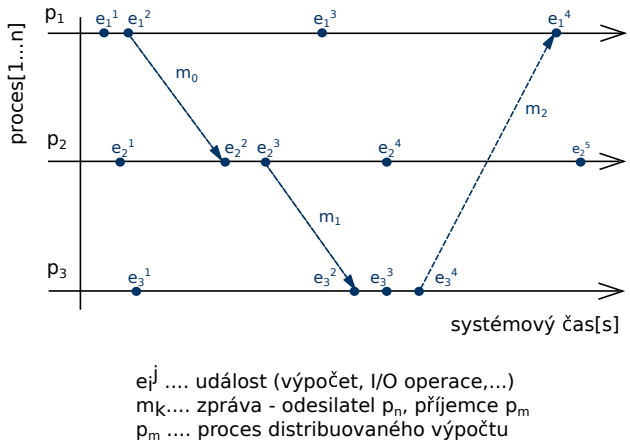
SIMGRID	Simgrid je open-source nástroj ke studiu chování rozsáhlých distribuovaných systémů
HTCondor	open-source pro distribuované výpočty (Zdroj: http://www.distributedcomputing.info/projects.html)
Folding@home	Projekt Stanfordské univerzity pro výzkum mechanizmu skládání proteinů, jehož výstupy by mohly vést k léčení souvisejících nemocí. (Zdroj: http://folding.stanford.edu/)

Charakteristika programů pro distribuované výpočty

Systém pro distribuované výpočty lze definovat jako sadu n asynchronních procesů $p_1, \ldots, p_n$. Každý z procesů může běžet na jiném typu procesoru, procesy **nesdílejí globální paměť** a jejich vzájemná komunikace probíhá na úrovni **předávání zpráv** [58].

Označíme - li C_{ij} jako komunikační kanál mezi procesy p_i a p_j, pak zpráva $m_{i,j}$ určuje výměnu informace mezi procesem p_i a procesem p_j. Prodleva času v komunikaci mezi jednotlivými procesy je konečná a nelze ji předvídat ani zaručit. Procesy nesdílejí globální hodiny. Realizace distribuovaných procesů a přenosu zpráv je tedy asynchronní [58].

Distribuovaný výpočet je popsán stavy procesů a komunikačních kanálů. Stav procesu je charakterizován stavem výpočetního zdroje, tzn. jeho pamětí a typem výpočetní jednotky. Stav komunikačního kanálu je charakterizován množinou zpráv a přenosovými vlastnostmi.



Obrázek 4.2: Časový diagram událostí v distribuovaném systému [58].

Procesy a události - základní subjekty v systémech distribuovaných úloh a výpočtů

V průběhu zpracování procesu se sekvenčně provádějí tři atomické sekvence událostí, jmenovitě:

- události procesu - změna kontextu, obsahu paměti, obsahu registrů
- odesílání zprávy
- příjem zprávy

Výskyt událostí mění stavy jednotlivých procesů a kanálů, což způsobuje přechody v globálním stavovém prostoru systému. Vnitřní událost mění pouze stav procesu, ve kterém se tato událost vyskytuje. Událost příjmu a odeslání zprávy mění stav procesu, který zprávu odesílá (nebo přijímá) a také mění stav kanálu, kterým je tato zpráva přenášena.

Jednotlivé události procesu jsou lineárně seřazené podle pořadí výskytu. Proces p_i produkuje sled událostí $e_i^1, \ldots, e_i^x$, přičemž dolní index označuje číslo procesu a horní index pořadí jednotlivých událostí v rámci procesu. Časový průběh p_i je vyjádřen následovně [58].

$$p_i = (h_i, \rightarrow_i) \tag{4.1}$$

kde h_i je množina událostí v čase, produkovaných procesem p_i. Binární relace $\rightarrow_i$ vyjadřuje příčinu a směr sledu událostí. Události odeslání a příjmu zprávy definují směr toku informací mezi jednotlivými procesy. Relace $\rightarrow_{msg}$ definuje závislosti mezi odesílatelem a příjemcem zprávy. Vývoj distribuované úlohy lze znázornit v diagramu 4.2. Horizontální časová osa znázorňuje směr vývoje procesu, tečky vyjadřují jednotlivé události (eventy) a šipky znázorňují přenosy zpráv [58].

Globální stav a řezy v časovém diagramu distribuovaného systému

Globální stav distribuovaného systému lze definovat jako souhrn lokálních stavů všech jeho komponent, to znamená procesů a komunikačních kanálů v určitém časovém úseku. Stav procesu je definován jako okamžitý stav paměti, registrů procesoru a I/O operací. Stav komunikačního kanálu je popsán souborem zpráv, které v daném časovém okamžiku předmětem přenosu ze vstupu na výstup komunikačního kanálu.

Výskyt jednotlivých událostí mění stavy procesů a komunikačních kanálů. To způsobuje přechody v globálním stavu systému.

Pro aktuální snímek průběhu procesů je nutné, aby stavy všech složek distribuovaného systému byly logovány ve stejném okamžiku. Toto je zajištěno jen v prostředí kde je zaručena synchronizace všech systémových hodin u všech komponent distribuovaného systému.

Tuto skutečnost nelze v rozsáhlých distribuovaných systémech zajistit bezvýhradně. Lze však říct, že i když časová synchronizace není zajištěna pro všechny komponenty distribuovaného systému bezpodmínečně, lze dosáhnout konzistence přechodu od jednoho globálního stavu systému k druhému, za dodržení logické kauzality mezi vysláním a příjmem zprávy. To znamená, že existuje - li záznam, že byla zpráva přijata, musí existovat i záznam, že zpráva byla odeslána. A naopak, nemůže existovat záznam o přijetí zprávy, pokud neexistuje záznam o jejím odeslání. Tento stav se nazývá konzistentní globální stav. Tato skutečnost má zásadní vliv na proces obnovy systému po případné poruše [58] [64].

Nechť LS_i^x označuje lokální stav procesu p_i po vzniku události e_i^x a před událostí e_i^{x+1}. LS_0 je inicializační stav procesu p_i. Pak LS_i^x označuje výsledek procesu a všech jeho kroků od události e_i^0, až po událost e_i^x [58].

Nechť.

$$send(m) \leq LS_i^x \quad \text{označuje stav} \quad \exists y : y \in \langle 1, x \rangle :: e_i^y = send(m). \tag{4.2}$$

A podobně.

$$rec(m) \not\leq LS_i^x \quad \text{označuje stav} \quad \forall y : y \in \langle 1, x \rangle :: e_i^y \neq send(m). \tag{4.3}$$

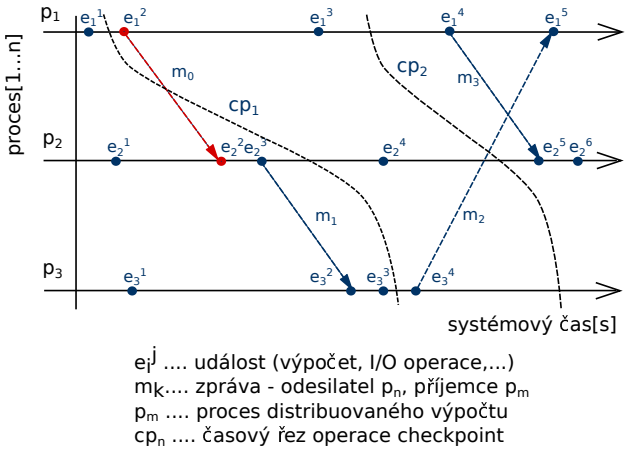
Stav komunikačního kanálu se odvíjí od stavu procesů, které zajišťují komunikaci. Označme $SC_{ij}^{x,y}$ stavy komunikačního kanálu.

$$SC_{ij}^{x,y} = \left\{ m_{ij} | send(m_{ij}) \leq LS_i^x \wedge rec(m_{ij}) \neq LS_j^y \right\} \tag{4.4}$$

Stav komunikačního kanálu $SC_{ij}^{x,y}$ označuje všechny zprávy od události e_i^x, které proces p_i odeslal, a které proces p_j do události e_j^ya dosud nezaznamenal [58].

Globální stav systému lze symbolicky definovat následovně [58].

$$GS = \left\{ \cup_i LS_i^{x_i}, \cup_{j,k} SC_{jk}^{y_j, z_k} \right\}. \tag{4.5}$$



Obrázek 4.3: Konzistentní a nekonzistentní události v časovém diagramu distribuovaného systému [58].

V časovém diagramu systému distribuovaných výpočtů, lze definovat čáru, která spojuje libovolné body v časovém diagramu jednotlivých procesů. Tato čára představuje řez, který odděluje v daném okamžiku sadu událostí v minulosti a budoucnosti probíhajícího výpočtu.

Konzistentní globální stav systému odpovídá řezu, v němž každá zpráva přijatá v minulosti, byla také zaslána v minulosti. O tomto typu časového řezu systému lze

prohlásit, že je konzistentní. O globálním stavu GS lze prohlásit, že je *konzistentní* v případě, že platí tato podmínka:

$$\forall m_{ij} : send(m_{ij}) \nleq LS_i^x \Rightarrow m_{ij} \notin SC_{ij}^{x,y} \wedge rec(m_{ij}) \neq LS_j^y \tag{4.6}$$

Zprávy, které procházejí přes definovaný řez z minulosti do budoucnosti také splňují podmínky konzistence globálního stavu systému.

Řez je nekonzistentní, pokud je zpráva vyslána v okamžiku, kdy překročí definovaný řez z budoucnosti do minulosti. Na obrázku 4.3 je porušena konzistence checkpointu cp_1 zprávou m_0, která byla procesu p_2 doručena dříve, než byl definován chekpoint procesu p_2. Přitom proces p_1, který tuto zprávu vyslal již checkpoint cp_1 zaznamenal. Byla tak porušena časová kauzalita průběhu jednotlivých operací [58].

Časový průběh procesu se skládá z časové posloupnosti provádění jednotlivých činností, například událostí, odeslání nebo příjem zprávy události). Události v procesu, jsou lineárně seřazeny dle pořadí výskytu. Tok informací mezi procesy definuje kauzální (příčinné) závislosti jednotlivých událostí. Globální stavy a přechody mezi nimi popisují stavy distribuovaného systému v konkrétních časových řezech. Jsou významné pro výměnu zpráv mezi jednotlivými procesy a také pro mechanismus checkpointingu, který hraje významnou úlohu při obnově systému po případné poruše. Princip checkpointingu je popsán v kapitole 5.1.

Synchronizace času v prostředí distribuovaných systémů

Princip zachování logické kauzality mezi jednotlivými událostmi má zásadní vliv na funkcionalitu a spolehlivost distribuovaných a paralelních systémů. Logická kauzalita událostí systému je dána směrem toku fyzického času systému, na který jsou všechny tyto události synchronizovány [58].

U paralelních výpočetních systémů je čas jednotlivých událostí odvozen od interního časovače systému. Tím je zaručena velmi dobrá synchronizace všech timestampů (časových razítek) těchto událostí. To umožňuje nasazení úloh, které vyžadují časovou souslednost.

V distribuovaných systémech je tato situace jiná. Zde nelze zajistit přesný časový souběh všech procesů. Je nutné přijmout omezení, která sice nezaručí přesnou synchronizaci, ale umožní alespoň aproximaci tohoto požadavku [58].

Systémové hodiny zajišťují mapování jednotlivých událostí $e_1, \ldots, e_n$ distribuovaného systému na prvky $\mathcal{T}(e_1), \ldots, \mathcal{T}(e_n)$ v časové doméně T.

Prvky $\mathcal{T}(e_1), \ldots, \mathcal{T}(e_n)$ se nazývají timestampy (časová razítka) událostí $e_1, \ldots, e_n$. Pro dvě události e_i a e_j na časové ose musí být splněna následující podmínka.

$$e_i \rightarrow e_j \Leftrightarrow \mathcal{T}(e_i) < \mathcal{T}(e_j) \tag{4.7}$$

Pak lze prohlásit, že systémový čas jednotlivých událostí je konzistentní [58].

Definice skalárního času

Princip skalárního času (navržen Leslie Lamportem [61]) vychází z myšlenky synchronizace jednotných hodin a předávání proměnné času mezi jednotlivými procesy v rámci distribuovaného systému. Čas procesu p_i je definován jako nezáporné celé číslo $\mathcal{T}_i$. Jestliže proces p_i příjme zprávu s timestampem $\mathcal{T}_{msg}$, provede následující posloupnost akcí

1. příjme zprávu s časem $\mathcal{T}_{msg}$; $\quad \mathcal{T}_i := max(\mathcal{T}_i, \mathcal{T}_{msg})$

2. inkrementuje čas před provedením jakékoliv události
   $\mathcal{T}_{i+1} := \mathcal{T}_i + d; \quad d > 0; \quad d \in \mathbb{N}$

3. odešle zprávu

Výhodou skalárního času je snadná realizace. Nevýhodou je, že lokální logický čas a globální logický čas představují jednu hodnotu. To vede ke ztrátě informace o časových souvislostech mezi jednotlivými událostmi [58].

Definice vektorového času

Vektorový čas byl navržen proto, aby byla odstraněna výše uvedená nevýhoda skalárního času. Každý proces má přidělen vektor $t[1, \ldots, n]$ který má stejnou dimenzi jako počet spuštěných procesů. Prvek vektoru $t_i[i]$ definuje logický čas procesu p_i.

Vektorový čas zajišťuje konzistenci procesů. Vývoj timestampů jednotlivých událostí procesů umožňuje uchovat časové souvislosti.

Nevýhodou řešení jsou vyšší nároky na systémové zdroje [58].

Systémové služby pro synchronizaci času

Pro časovou synchronizaci událostí v distribuovaném systému lze využít NTP (Network Time Protocol), jehož úkolem je zajistit přesnou časovou synchronizaci všech účastníků síťového provozu. Je navržen tak, aby byl schopen odolat i následkům proměnlivého zpoždění v doručování síťových paketů [58] [88].

5 Fault tolerance

Základním problémem při návrhu distribuovaného systému, je volba vhodného mechanismu pro komunikaci mezi procesy. Z technického hlediska se jedná o propojení jednotlivých uzlů pomocí prostředků počítačové sítě. Výsledkem je definice systému pro přenos a distribuci zpráv mezi jednotlivými procesy.

Tolerance chyb je důležitá vlastnost, zejména v rozsáhlých výpočetních systémech, kde geograficky distribuované uzly spolupracují na provedení úkolu. Vysoká úroveň spolehlivosti a dostupnosti, je podmínkou odolnosti proti chybám. Selhání jakékoliv větve této struktury má zásadní vliv na QoS soustavy. Tolerance chyb je schopnost systému plnit svou funkci správně i za přítomnosti poruch. Odolnost proti chybám je zásadní vlastnost, která vede k vyšší spolehlivosti systému. Primární metodikou pro zvýšení spolehlivosti je prevence. Kontrolní mechanismy, jejichž smyslem je odstranit okolnosti, kterými vznikají poruchy, jsou velmi důležité při provozu rozsáhlých gridových soustav. Každý systém má definovány parametry, které zaručují bezchybnou funkcionalitu. Základním parametrem je šířka přenosového pásma systému. Jakákoliv odchylka od normálního chování systému má vliv na definovanou šířku přenosového pásma.

K selhání tedy dochází, když se skutečný běh systému odchyluje od definice standardních parametrů. Chyba představuje neplatný stav systému, který není v souladu se specifikací systému. Jinými slovy, chyba je příčinou selhání systému.

Standardně používanými technikami pro zvýšení odolnosti proti chybám jsou replikace a checkpointing. Obě tyto techniky však mají vliv na vyšší alokaci výpočetních zdrojů [58].

Checkpointing

Principem checkpointingu je snímkování aktuálního stavu procesu a jeho zaznamenávání do nezávislého úložiště - žurnálu procesů. V případě, že vznikne jakákoliv porucha, systém dokáže restaurovat stav, který byl před poruchou. Následně tuto defektní část znovu pošle do zpracování. Rekonstrukce stavu distribuovaného systému s více procesy, je poměrně velmi náročná úloha [57] [54].

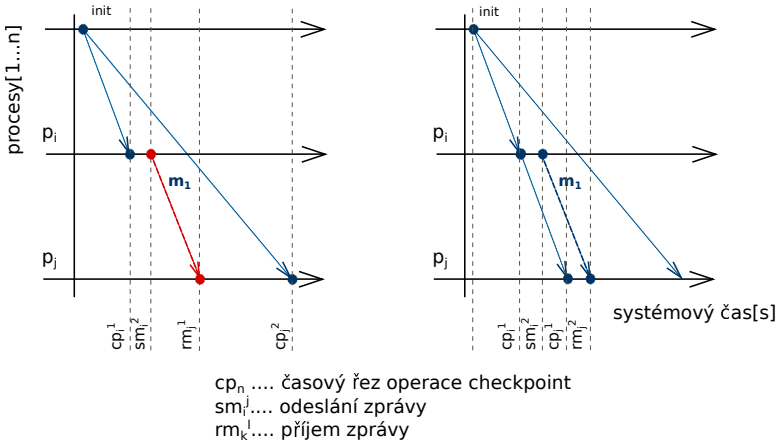
Zejména u procesů, které vzájemně komunikují prostřednictvím zpráv, může rollback způsobit komplikace v integritě úlohy. Tato situace může nastat v okamžiku, kdy je mezi dvěma kontrolními body (checkpointy) odeslána očekávaná zpráva. Po této zprávě dojde k výjimce ve zpracování [17].

Rollback a recovery jsou techniky určené pro eliminaci transientních (dočasných) chyb systému [58] [64]. Transientní chyba může vzniknout kdykoliv v průběhu výpočtu. Tato chyba nemusí být detekována okamžitě. Příčinou vzniku může být řada výjimek, jako například nestabilní síťové spojení, hardwarová nebo softwarová chyba. Transientní chyba je definována jako chyba dočasná a opravitelná (recoverable). Vznik transientní chyby znamená, že systém musí obnovit svůj stav do okamžiku posledního checkpointu a musí provést restart [57].

Checkpointing neřeší vznik permanentní chyby, která je definována jako neopravitelná (irrecoverable), a ve většině případů znamená výpadek celého systému [58]. V distribuovaných systémech, je systém posílání zpráv jedinou možností, jak zajistit komunikaci mezi jednotlivými procesy. Zpráva, která je generovaná odesílatelem, vyvolá determinované akce na straně příjemce. Úkolem checkpointingu je zajistit synchronizaci bodů obnovy (check pointů) a synchronizaci systému předávání zpráv. Tyto činnosti následně zajistí konzistenci prováděných transakcí.

Nezávislý (nekoordinovaný) checkpointing

V případě nezávislého checkpointingu jsou pro všechny procesy periodicky generovány jednotlivé kontrolní body (checkpointy) nezávisle na jiných procesech. Při výpadku je nutné nalézt tzv. konzistentní řez.



Obrázek 5.1: Nekoordinovaný a koordinovaný checkpointing [58].

Problém: V případě, že poslední checkpoint vytvoří nekonzistentní řez, pak se musí systém vracet zpět, dokud není nalezen poslední konzistentní řez. Kaskádové vrácení provedených změn může vést k domino efektu. To znamená, že se celý výpočet může vrátit, až na samý počátek. Proto se tento typ checkpointingu příliš nepoužívá [37].

Synchronní (koordinovaný) checkpointing

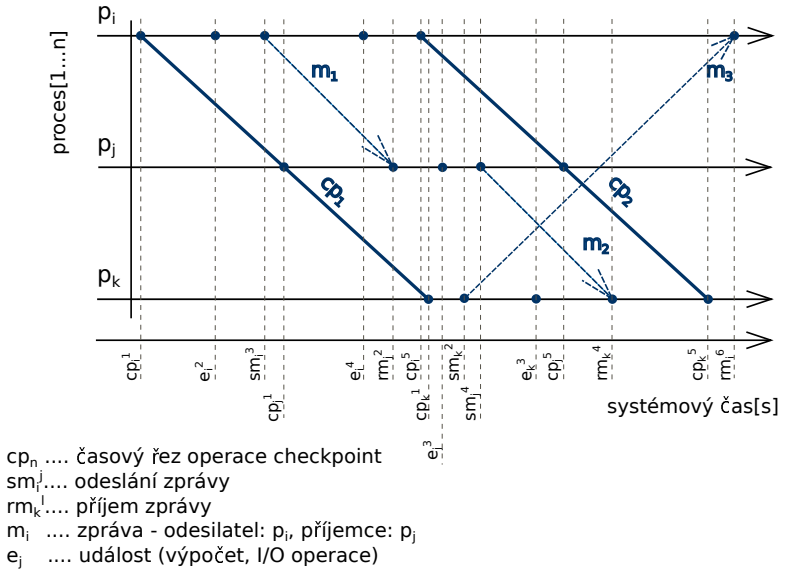
Komunikace v distribuovaných systémech probíhá prostřednictvím výměny zpráv. Zpráva generovaná odesílatelem, spouští u příjemce zprávy jisté procesy, vyplývající z obsahu této zprávy [21] [58]. Proces distribuovaného checkpointingu řeší tři základní úkoly:

- časová synchronizace kontrolních bodů - checkpointů
- proces přenosu zpráv - messaging
- operace obnovy - rollback

Primární úlohou synchronního checkpointingu je zajistit synchronizaci zpráv a checkpointů (kontrolních bodů) mezi jednotlivými procesy tak, aby byla zachována konzistence.

Transakce synchronního checkpointingu

Obrázek 5.2 znázorňuje průběh jednotlivých procesů p_i, p_j a p_k. V procesech se vyskytují tři základní objekty. Checkpointy $cp_1, \ldots cp_x$, zprávy $m_1 \ldots m_y$ a události $e_1 \ldots e_z$. Dále je definován také okamžik odesání zprávy m_j procesem p_i, který je značen sm_i^x a okamžik příjmu zprávy m_j procesem p_j, značen jako sm_j^y. Dolním index představuje číslo procesu, horní index znamená pořadí operace v rámci spuštěného procesu. Například příjem zprávy procesem p_2, který v časové ose představuje pátou operaci, je značen rm_2^5. Časová značka checkpointu je označena cp_i^j, přičemž dolní index znázorňuje číslo procesu a horní index číslo operace checkpointu [58].



Obrázek 5.2: Transakce konzistentního checkpointu [58] [64].

Tyto události jsou promítnuty na časovou osu systému. Sekvence operací na ose systémového času představují záznamy do systémového logu [58] [64].

$$\begin{aligned} log \;=\; & cp_i^1 \rightarrow e_i^2 \rightarrow sm_i^3 \rightarrow cp_j^1 \rightarrow e_i^4 \rightarrow \\ & rm_j^2 \rightarrow cp_i^5 \rightarrow cp_k^1 \rightarrow e_j^3 \rightarrow sm_k^2 \rightarrow \\ & sm_j^4 \rightarrow e_k^3 \rightarrow cp_j^5 \rightarrow rm_k^4 \rightarrow cp_k^5 \rightarrow rm_i^6 \end{aligned} \tag{5.1}$$

V žurnálovacím logu je zaznamenáno souběžně několik událostí. Hodnoty v závorce reprezentují index jednotlivých procesů, které proběhly ve znázorněném časovém úseku.

$$\begin{aligned} &\text{Checkpoint:} \\ &\qquad cp_1 \;=\; cp_i^1 \rightarrow cp_j^1 \rightarrow cp_k^1 \\ &\qquad cp_2 \;=\; cp_i^5 \rightarrow cp_j^5 \rightarrow cp_k^5 \\ &\text{Zpráva:} \\ &\qquad m_1 \;=\; sm_i^3 \rightarrow rm_j^2 \\ &\qquad m_2 \;=\; sm_j^4 \rightarrow rm_k^4 \\ &\qquad m_3 \;=\; sm_k^2 \rightarrow rm_i^5 \\ &\text{Událost:} \\ &\qquad e_1 \;=\; e_i^2 \\ &\qquad e_2 \;=\; e_i^4 \\ &\qquad e_3 \;=\; e_j^3 \\ &\qquad e_4 \;=\; e_k^3 \end{aligned} \tag{5.2}$$

Nazvěme událost checkpointu cp_x operací comit. Comit vytyčuje oblast mezi minulostí a budoucností procesů $p_1, \ldots, p_n$ a definuje takzvané body obnovy.

$$cp_x \;:=\; \{cp_1^x, cp_2 1^x, \ldots, cp_n^x, \} \tag{5.3}$$

Dále všechny události od $comit_x$ včetně, až po $comit_{x+1}$ vyjma, nazvěme transakcí T_x.

$$T_x \;:=\; \{cp_x, m_x, e_x\} \quad cp_{x+1} \notin T_x \tag{5.4}$$

Transakcí T_x je definována posloupnost všech událostí v intervalu:

$$T_x \in \langle comit_x, comit_{x+1}) \tag{5.5}$$

O transakci lze prohlásit, že je konzistentní, pokud splňuje podmínky definice konzistentního globálního stavu systému [58]. Ten je definován ve vztahu 4.5.

Operace rollback

Rollback je definován jako operace, která při vzniku chyby v průběhu zpracování transakce T_x, vrátí systém do posledního známého konzistentního stavu systému, v tomto případě do okamžiku poslední operace $comit_x$.

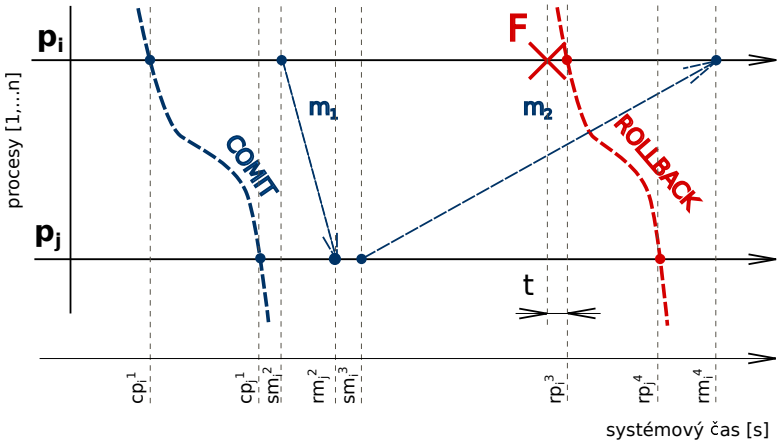
Rollback je v distribuovaných systémech poměrně komplikovaná operace [58], protože proces, který operaci rollback vyvolal, mohl poslat mnoho různých zpráv jiným procesům. Proto i účinky těchto zpráv a procesů, které byly ovlivněny, musí být vráceny do posledního známého konzistentního stavu systému. Konzistenci operace rollback lze zajistit prostřednictvím sériového zápisu jednotlivých událostí transakce, což je znázorněno v rovnici 5.1. Nutnou podmínkou pro úspěšný rollback je přesné dodržení časové posloupnosti návratu všech událostí od okamžiku vyvolání operace *rollback* do poslední operace *comit* (metoda FIFO). Toto lze demonstrovat na následujícím příkladu.

Pořadí závislostí mezi souběžně prováděnými transakcemi je určeno pořadím jejich konfliktních operací. Operace jsou konfliktní, pokud nejsou komutativní. V tomto případě může žurnálovací systém restaurovat rozdílné výsledky, pokud se zamění pořadí. Příklad konfliktních operací:

$$x := x + a \qquad x := x * b$$

Operace inkrement $x = x + a$ a násobek $x = x * b$ nejsou komutativní.

Příklad konzistentního rollbacku



Obrázek 5.3: Konzistentní rollback [58] [64]

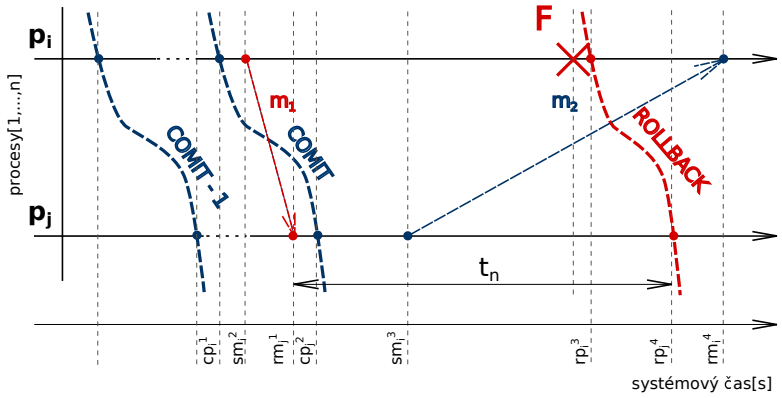
.

Na obrázku 5.3 je znázorněn průběh operací checkpointingu. Oblast znázorněná body cp_i^1, cp_j^1 definuje oblast obnovy - recovery line, která představuje časové okamžiky procesů p_i a p_j. do nichž bude v případě poruchy provedena obnova. Body rp_i^3, rp_j^4 definují oblast obnovy, odkud bude po detekci chyby F prováděn rollback všech událostí do posledního comitovaného (potvrzeného) checkpointu, tedy do oblasti cp_i^1, cp_j^1.

Obrázek 5.3 znázorňuje konzistentní průběh transakce. protože časová posloupnost jednotlivých událostí umožňuje provést rollback do definovaných bodů cp_i^1 a cp_j^1. Čas t, představuje zpoždění systému, při reakci na chybu F.

Příklad nekonzistentního rollbacku

Obrázek 5.4 znázorňuje nekonzistentní rollback, kdy zpráva m_1 poruší časovou souslednost a protne definovanou oblast cp_i^1 a cp_j^2.



Obrázek 5.4: Příklad nekonzistentního rollbacku [58] [64]

.

To znamená, že v časové souslednosti je zpráva doručena dříve, než proces p_j stačí definovat checkpoint cp_j^2. Nekonzistentní stav nastává v okamžiku, kdy odesílatel zprávy, v tomto případe proces p_i, vyšle příkaz k rollbacku, ale příjemce této zprávy nemůže rollback provést, protože ještě nemá definován výše zmíněný checkpoint. cp_j^2. Časový úsek t_n znázorňuje interval, po který je proces checkpointingu nekonzistentní. V tomto případě se musí systém zachovat tak, že provede rollback až do stavu $COMIT - 1$. U nekoordinovaného checkpointingu (viz obrázek 5.1) však vzniká nebezpečí domino efektu, kdy není při návratu v časové ose nalezena žádná oblast konzistentního checkpointu. Výsledkem je návrat výpočtu na úplný začátek. Situace je popsána v kapitole 5.1.1.

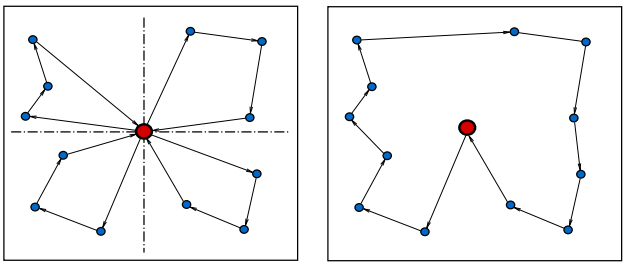
6 Plánování procesů

Plánování procesů je v prostředí gridové služby základní úloha, která je zodpovědná za přidělování výpočetního času a výpočetních zdrojů jednotlivým úkolům. Výpočetní zdroj může zahrnovat široké spektrum definic, které budou vycházet z různých typů výpočetních technologií. Například CPU jako výpočetní zdroj je chápán jako základní prvek, kterému jádro operačního systému přiděluje v určitých časových kvantech jednotlivé úkoly. Plánovač jádra operačního systému je zodpovědný za spravedlivé přidělování času jednotlivým úkolům, přičemž například priorita úlohy představuje jen jedno z kritérií, které musí tento plánovač zahrnout do své činnosti. Plánovač na úrovni jednoho CPU vytváří pseudo paralelní (vícevláknové) prostředí, které umožní víceúlohové řízení aplikací v rámci jediného výpočetního zdroje. V rozsáhlých (paralelních systémech) však může být výpočetní zdroj hierarchicky začleněn do vyšších struktur. Například pro grid může být výpočetním zdrojem gridový výpočetní zdroj nebo skupina gridových výpočetních zdrojů. Pro rozsáhlé gridové struktury může být výpočetní zdroj chápán jako jedna větev gridové topologie. Důležité je, že každá takto definovaná výpočetní jednotka má svůj plánovač úloh.

Plánování v prostředí Gridu lze převést na úlohu diskrétní optimalizace jejíž úkolem je najít nejlepší sadu parametrů pro optimalizaci cílové funkce. Úlohy diskrétní optimalizace a konkrétně plánování procesů patří do kategorie úloh s výpočetní složitostí $\mathsf{NP}-\mathsf{úplných}$ problémů, to znamená problémů pro něž není znám algoritmus, který by tuto úlohu vyřešil v deterministicky polynomiálním [1] čase. Zjednodušeně lze tento problém definovat tak, že není znám algoritmus, který by našel optimum v rozumném časovém intervalu.

[1] Obecně platí, že úlohy řešitelné v rozumném čase spadají do třídy složitosti maximálně $\mathcal{O}(n^k)$ to znamená úlohy, jejichž výpočetní čas roste nejvýše polynomiálně

Otázka rovnosti nebo nerovnosti časové složitosti třídy P, kam spadají úlohy pro něž je znám algoritmus vyčíslení v rozumném [2] čase a třídy NP, pro něž algoritmus, který by ji vyřešil v čase polynomiálním, znám není, je zařazena do kategorie matematických problémů tisíciletí [27].



Obrázek 6.1: Dekompozice úlohy

V případě, že bude proveden důkaz, že NP problém lze zredukovat na P problém (tzn. že platí $\mathsf{N} = \mathsf{NP}$), bude to mít dalekosáhlé dopady do řady odvětví současné informatiky a kryptografie.

Úlohy hledání nejkratší cesty v Hamiltonově kružnici, není vhodné řešit paralelně, a to i přesto, že se tato možnost pro svou jednoduchost přímo nabízí. Paralelní úloha dává obvykle horší výsledky než sekvenční zpracování. To vyplývá z následující úvahy.

Rozdělíme-li limitně řešení úlohy na tolik paralelních větví jako je počet hran grafu, obdržíme nejhorší možné řešení úlohy. Obrázek 6.1 tuto skutečnost snadno demonstruje. K výše uvedenému typu patří také úlohy v kategorii Job-Shop problém, tedy úlohy, které se zabývají plánováním a rozvrhováním procesů. Toto je úloha, která je pro aplikace Gridu klíčová[3].

Úlohy, které lze převést na problém Hamiltonovy kružnice patří do množiny $\mathsf{NP}-\mathsf{úplných}$. Ač není znám efektivní exaktní algoritmus, jedná se o třídu úloh, které mají velké praktické dopady (plánování výrobních zdrojů, hledání optimální cesty). Takže je nutné hledat takové postupy a algoritmy, které dávají v rozumném čase, alespoň suboptimální řešení [20] [102].

[2] přesněji v čase nejvýše polynomiálním

[3] V tomto případě může převod úlohy na paralelní zpracování dávat chybné výsledky v tom smyslu, že v jednom časovém okamžiku bude naplánováno více úloh na jeden zdroj

Metody soft-computingu a metaheuristické algoritmy

Řešení úloh, které nemají efektivní exaktní algoritmus je podmíněno určitými znalostmi a zkušenostmi, které je nutné uplatnit, abychom docílili snížení výpočetní náročnosti úlohy i za cenu ztráty exaktnosti řešení. Do této kategorie jsou zařazeny algoritmy, které využívají bázi zkušeností prověřených praxí, nebo obecných postupů pozorovaných v rámci živé či neživé přírody. Obecně se dá rozdíl mezi exaktním postupem a heuristickým přístupem deklarovat jako známá zkušenost, že například drtivé procento všech babiček jsou vynikající kuchařky,[4] protože díky svým dlouholetým zkušenostem a propracovaným postupům dokážou vytvořit mistrovská gastronomická díla. Samozřejmě i v oblasti gastronomie lze definovat exaktní pravidla a postupy, obvykle s tlakem na optimalizaci nákladů. Výsledek je ovšem stejně uniformní jako přístup a rozhodně nepřináší nic, co by konzumenta takovéhoto produktu něčím obohatilo.

U heuristických metod není hlavním kritériem použitá metoda, to znamená na jakém principu tyto úlohy fungují. Důležité je to, že fungují a poskytují dobré výsledky. Tyto metody se však hodí jen na určité specifické úlohy. Nelze je proto použít ve všech oblastech diskrétní optimalizace [102].

Metody soft-computingu a metaheuristických algoritmů jsou třídy úloh, jejichž vlastnosti je předurčují k řešení problémů zařazených v kategorii $\mathsf{NP-úplné}$ [90]. Důležitými vlastnostmi jsou obecnost a tolerance k nepřesnostem v zadání úlohy. Patří sem systémy založené na fuzzy logice, evoluční algoritmy, genetické algoritmy, algoritmy využívající vlastností neuronových sítí a strojového učení. Jedná se o velmi zajímavou a rozsáhlou oblast.

Při návrhu plánovače Gridu však bylo primárně zkoumáno, zda existuje obecná softwarová komponenta nebo knihovna, která problém diskrétní optimalizace řeší a zda bude možno ji využít v optimalizačním plánovači. Druhým úkolem bylo také navrhnout metodiku pro určení délky běhu optimalizačního algoritmu.

[4] Samozřejmě, existují i výjimky, ale v tomto případě spíš neobvyklé

Klasifikace plánovacího rozvrhu

Úloha plánování procesů spadá do kategorie úloh Job-Shop scheduling, které řeší optimální rozložení práce pro jednotlivé dostupné zdroje.

Pro klasifikaci úlohy je nutné definovat jistá kritéria definovaná Grahamovou klasifikací, $(\alpha|\beta|\gamma)$ [20], která umožní určit charakter prostředí, typ úloh a vlastnosti kritérií pro optimalizaci. Grahamova klasifikace především usnadňuje přesnější definici řešeného problému.

Job-Shop úlohy představují velmi rozsáhlou množinu problémů. Peter Brucker [20] uvádí odkaz `http://www.mathematik.uni-osnabrueck.de/research/OR/class`, v němž je možno, na základě této metodiky definovat problém a získat více informací o článcích a vědeckých statích, které se konkrétním typem problému zabývají.

Grahamova klasifikace Job-Shop problémů

Vstupními hodnotami pro klasifikaci Job-Shop problému, jsou zdroje α, omezující podmínky úlohy β a optimalizační kritéria γ [20].

Klasifikace zdrojů - α

- **jediný zdroj *{1}*** - Všechny úlohy jsou zpracovávány na jediném výpočetním zdroji
- **identické paralelní zdroje *{Pm}*** - čas zpracování konkrétní úlohy je na každém zdroji stejná
- **paralelní zdroje s různou rychlostí zpracování *{Qm}*** - čas zpracování konkrétní úlohy je na různých zdrojích různá. Podmínkou však je, že jakákoliv úloha může být zpracována na jakémkoliv zdroji, který je k dispozici
- **neidentické zdroje s různou rychlostí zpracování *{Rm}*** - čas zpracování konkrétní úlohy je na různých zdrojích různá, a navíc platí omezení, že ne každá úloha může být zpracována na jakémkoliv zdroji. Příklad z praxe - operaci výpočet na grafické kartě nelze provést na stroji, který ji nemá instalovánu.

Klasifikace úloh a omezujících podmínek - β

- **precedenční podmínky *{prec}, {tree}, {chains}*** - prec - omezení pořadí zpracování úloh, tree - omezení stromovou strukturou, chains - omezení zřetězení zpracování
- **přerušení úlohy *{pmtn}*** - preempce - při příchodu úlohy s vyšší prioritou je stávající úloha přerušena.
- **vhodnost zdroje *{*** (S_j) ***}*** - podmnožina zdrojů které vyhovují požadavkům úlohy
- **čas startu úlohy - release time** $s(a)$
- **čas ukončení úlohy - deadline** $e(a)$
- **čas očekávaného ukončení úlohy - due date** $\sigma(a)$
- **celková doba zpracování úlohy - total processing time** $p(a)$

Klasifikace optimalizačních kritérií - γ

- **makespan** $M = max\{M_i; i = 1, \ldots n\}$ - Maximální čas konce úloh - cílem je minimalizace makespanu.
- **zpoždění - lateness** $L = max\{L_1, \ldots, L_n\}$ - Minimalizace maximálního zpoždění.
- **nezáporné zpoždění - earlines** $E = max\{\sigma(a) - e(a), 0\}$ - minimalizace celkového zpoždění
- **zdlouhavost úlohy - tardiness** $T = max\{e(a) - \sigma(a), 0\}$ - minimalizace celkového času

Volba optimalizační strategie

Pro volbu vhodné plánovací strategie je nutné určit charakter úlohy. V navrženém Intranet Gridu jsou propojeny výpočetní zdroje typu běžných personálních počítačů a pracovních stanic, které se od sebe odlišují pouze výpočetní kapacitou . Tato zjednodušující podmínka umožňuje řešit jakoukoliv část zpracovávané úlohy, na kterémkoliv z výpočetních zdrojů. Bude se lišit pouze čas zpracování. Proto je lze dle Grahamovy klasifikace zařadit do kategorie paralelních zdrojů s různou rychlostí zpracování *{Qm}*.

Charakter úloh, které jsou ve výpočetním Gridu zpracovávány, však zahrnuje více kritérií Grahamovy specifikace. Posloupnost zpracování úloh může obsahovat podmínku časové závislosti, to znamená, že vstupní parametry úlohy následující mohou být závislé na výstupních parametrech úlohy předcházející. Jisté části úlohy mohou být zpracovávány paralelně a také mohou být zřetězeny. To znamená, že úloha řešená plánovačem Gridu patří do kategorie precedenčních podmínek $\{prec, tree, chains\}$.

Situace se dále komplikuje tím, že v důsledku zajištění spravedlivého rozdělování výpočetního času různým úlohám s různou prioritou, také dochází k přepínání kontextu a rozdělení strojového času mezi jednotlivé nezávisle zpracovávané úlohy. To znamená, že do klasifikace tohoto typu úlohy musí být také zahrnuto kritérium přerušení úlohy $\{pmtn\}$.

Kritérium vhodnost zdroje v tomto případě nebylo uvažováno, protože se v této verzi výpočetního Gridu předpokládala jednotná výbava (blíže vysvětleno v kapitole 6.4). V další vývojové fázi Gridu se však předpokládá zahrnout i toto kritérium do návrhu plánovače. K tomu vede praktický požadavek například možnosti využití speciálního softwarového vybavení, které z důvodu nákladů, nebude instalováno na každém výpočetním zdroji. Dle Grahamovy klasifikace, $\{\alpha|\beta|\gamma\}$ je tato úloha zařazena do kategorie:

$$\{Qm \mid prec, tree, chains, pmtn \mid M\}$$

Toto kritérium představuje pro plánovač Gridu velmi náročnou úlohu. V důsledku toho bylo zvoleno takové řešení, které tuto úlohu rozdělí na více problémů. Ty jsou následně řešeny speciálně navrženými komponentami optimalizačního plánovače. Zodpovědnost za kategorii $\{pmtn\}$ byla předána prioritnímu plánovači, (popsán v kapitole 7.1). Za kategorii $\{prec\}$ zodpovídá precedenční plánovač (popsán v kapitole 7.2). Optimalizační plánovač, (viz. kapitola 7.3) následně řeší zjednodušenou úlohu:

$$\{Qm \mid tree, chains \mid M\}$$

Pro tuto optimalizační úlohu jsou vstupními parametry množina výpočetních zdrojů $S := (1, \ldots, s)$, které se liší výpočetní kapacitou 6.5 a množina úloh $J :=$

$(1, \ldots, j)$. Dále se předpokládá, že úloha j_l spotřebuje $t_{k,l}$ časových jednotek, pokud je naplánována na výpočetním zdroji s_k. Úkolem plánovacího procesu je nalezení optimálního rozvrhu pro všechny zdroje a úlohy.

Pro optimalizační kritérium je nutno vzít v úvahu fakt, že celkový čas úlohy se skládá z času potřebného pro vytvoření rozvrhu úlohy a času běhu vlastní úlohy - časový rozdíl mezi začátkem a koncem posloupnosti sekvencí dávkové úlohy (MAKESPAN)u.

$$t_J = t_{scd} + M[s] \tag{6.1}$$

Proto při návrhu parametrů plánovače je nutno vzít do úvahy dvě protichůdné hodnoty a nalézt jejich optimální velikost. Kvalita plánovacího procesu závisí na algoritmu a čase t_{scd}, po kterou tento algoritmus běží. Delší čas plánovacího běhu t_{scd}, má příznivý vliv na optimalizaci MAKESPANu M. Čas t_{scd} však nepříznivě ovlivňuje celkový čas úlohy. V kapitole 7.3 je uvedena metodika návrhu délky běhu času t_{scd}, který má vliv na kvalitu návrhu rozplánování úloh a celkový čas zpracování.

Určení kapacity výpočetního zdroje

Každý výpočetní zdroj musí splňovat jisté minimální hardwarové požadavky, aby byl schopen řešit úlohy v prostředí distribuovaných výpočtů a gridu. Pro popis hardwarových a výkonových vlastností výpočetního zdroje byla navržena metodika benchmarkových testů, které tyto vlastnosti vyhodnotí a zveřejní v komunikačním rozhraní každého výpočetního zdroje. Tato hodnota - kapacita výpočetního zdroje je následně k dispozici optimalizačnímu plánovači gridu. Ten následně přiděluje úlohy tak, aby optimalizoval rozložení zátěže, dle výkonových vlastností jednotlivých uzlů gridu.

Pro určení kapacity, jsou zahrnuty tyto parametry výpočetního zdroje.

- Výkon CPU
- Velikost paměti
- Vlastnosti síťového propojení

Kapacita výpočetního zdroje (přesněji řečeno ocenění výpočetního zdroje, protože se jedná o bezrozměrné číslo) má vliv na spravedlivé rozložení zátěže pro jednotlivé výpočetní zdroje. Tato plánovací strategie vede na optimalizaci hodnoty MAKESPANu M. Makespan je významným indikátorem propustnosti gridové služby, proto je také často používán jako jedno z hlavních optimalizačních kritérií [20].

Definice kapacity výpočetního zdroje je uvedena v článcích [93, 94]. Je zde uveden matematický popis kapacity jednotky CPU. Publikovaná metodika dává přesný popis chování CPU v závislosti na jeho instrukční výbavě, ale nehodnotí výpočetní zdroj jako celek. Nezapočítává další oblasti, které mají na celkové vyhodnocení kapacitních vlastností také výrazný vliv.

Pro návrh a zápočet všech vlivů na vyhodnocení celkové kapacity výpočetního zdroje, byla v našem případě zvolena metoda experimentálních měření a benchmarkových testů, které výpočetní zdroj ocení jako celek se započtením vlivu všech konstrukčních částí tzn. CPU, paměti, I/O subsystému a síťových služeb. Návrh a princip této experimentální metody je uveden v kapitole 9.2.6.

7 Plánovač

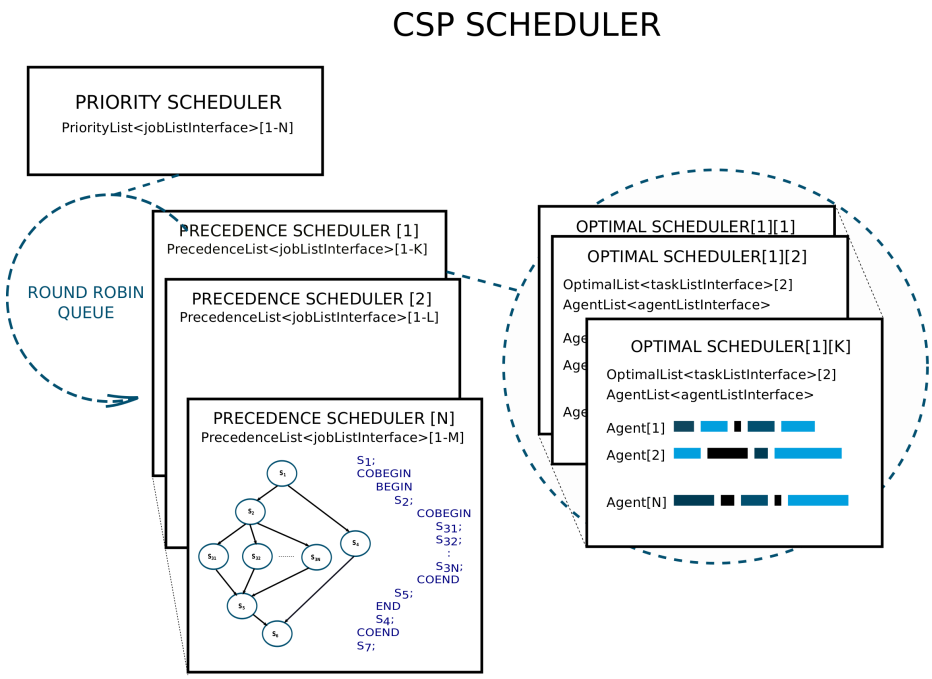
Plánování a distribuce úloh jsou základními prvky gridových služeb. Podmínkou je nástroj, který umožní snadnou definici úlohy, její distribuci na výpočetní zdroje a také snadné sledování průběhu vlastního zpracování. Další velmi významnou vlastností je spolehlivost a snadná správa. Uživatel Gridu by měl mít k dispozici určitou svobodu, a neměl by být, pokud možno svazován například licenčními podmínkami nebo podobnými překážkami při definici svých úloh [68].

Cílem byl návrh plánovače, jehož základní funkcionalitou je vytvořit takový rozvrh úloh, který bude splňovat prioritní, precedenční a optimalizační požadavky pro distribuci a zpracování dávkových úloh v prostředí výpočetního gridu. Výsledkem je rozvrh optimálního plánovacího času v souvislosti s počtem úloh, které jsou uvolněny do zpracování a také porovnání jednotlivých plánovacích strategií. V prostředí Intranet Gridu byla použita specifikace JSDL, určená pro pro řízení a distribuci dávkových úloh. Současná verze 1.0 má také definovánu podporu Portable Operating System Interface (POSIX). JSDL má pro řízení úloh také podporu, pro definici rozsahů výpočetních zdrojů (délka zpracování, maximální počet spuštěných vláken na procesoru, velikost diskového prostoru). Povinně definované parametry (typ systému, typ procesoru, paměť, propustnost sítě) jsou základními parametry vstupními parametry plánovače [11] [72].

Plánovač představuje důležitou komponentu výpočetního Gridu. Je zodpovědný za dodržení požadované posloupnosti zpracování a optimalizovaný rozvrh paralelních částí zpracovávané dávkové úlohy. Dlouhodobý (long term) plánovač v prostředí Intranet Gridu , je navržen jako soustava tří hlavních modulů.

Prioritní plánovač má za úkol optimalizovat řízení úloh, na základě priority. Priorita procesu je vstupním parametrem, který je povinně zadán uživatelem. Za pořadí zpracování v rámci spuštěné úlohy je zodpovědný *precedenční plánovač*.

Úkolem *optimalizačního plánovače* je minimalizace délky zpracování - makespanu. Vstupem pro optimalizaci je časový údaj, který zadá uživatel při plánování úlohy. Výhodou této koncepce je vyšší přehlednost a transparentnost algoritmu plánovače, protože byly logicky rozčleněny role jednotlivých plánovačů.



Obrázek 7.1: Blokové schéma optimalizačního plánovače

Prioritní plánovač

Je použita modifikovaná strategie cyklické fronty obsluhy plánovače (Round-robin). Priorita úloh je řešena přiděleným časovým intervalem t_j , kterým je určena aktivita konkrétní úlohy. Po uplynutí dané doby je úloha pozastavena. Řízení je následně předáno úloze další. Čas aktivity úlohy t_j je odvozen od předem určeného časového kvanta t_q, které představuje nejkratší možný úsek pro aktivitu úlohy. Je zaručeno, že úloze s nejnižší prioritou bude přidělována aktivita alespoň v čase t_q

[s]. Priorita úlohy P je jako povinný parametr zadán uživatelem $P \in \{1, \ldots, 5\}$, přičemž platí, že úloha s $P = 1$ má nejnižší možnou prioritu[1].

Mechanismus cyklického přidělování pracovního času není náchylný k zanedbávání úloh s nižší prioritou (process starvation). Podmínkou pro správnou funkci takto navržené plánovací strategie je určení optimálního t_q pro přepínání kontextu jednotlivých úloh.

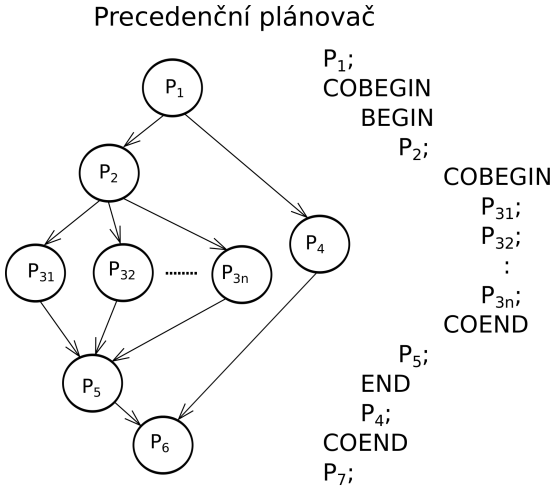
Příliš krátký interval, znamená příliš časté přepínání kontextu. To v praxi znamená výrazný nárůst režie systému. Příliš dlouhý interval časového kvanta t_q, naopak znamená degradaci round robin plánovače na FCFS plánovač. Role FCFS plánovače je v tomto případě nevýhodná z hlediska strategie obsluhy jednotlivých úloh. Dochází k nespravedlivému rozdělení času v případě dlouhých úloh. Úlohy s kratší dobou zpracování musí zbytečně dlouho čekat.

Precedenční plánovač

Precedenční plánovač je zodpovědný za řízení datového toku v rámci jednotlivých úloh. Plní roli střednědobého (mid-term) plánovače v rámci jedné úlohy.

Je řešen jako prioritní FIFO fronta, se zpětnou vazbou. Sleduje úspěšné ukončení poslední úlohy, která poskytuje výsledky pro úlohu následující. Plánovač řeší obecný acyklický graf (Directed Acyclic Graph), který popisuje workflow úlohy, které je definováno zadavatelem. Precedenční plánovač také řeší problém checkpointingu systému. V případě, že vznikne jakákoliv porucha, systém dokáže restaurovat stav, který byl před poruchou. Následně tuto defektní část znovu pošle do zpracování. Umí řešit jen transientní typy chyb, které jsou definovány jako chyby dočasné a opravitelné (recoverable). Vznik transientní chyby znamená, že systém musí obnovit svůj stav do okamžiku posledního checkpointu a následně restartovat defektní část zpracování. Neumí řešit permanentní typy chyb, které ve většině případů znamenají výpadek celého systému [58].

[1] Pro přidělení priority byla zvolena opačná filozofie než je tomu u operačních systémů (kde platí že nižší číslo má vyšší prioritu) z praktických důvodů. Prioritu úlohy pro gridovou službu přiděluje uživatel. Pro něj je mnemonicky lépe zapamatovatelné, že větší číslo, znamená vyšší prioritu.



Obrázek 7.2: Precedenční plánovač [26].

Optimalizační plánovač

Úkolem optimalizačního plánovače je nalezení optima rozložení výpočetní zátěže na jednotlivé výpočetní zdroje gridu. Snahou je minimalizovat výpočetní čas úlohy - MAKESPAN.

Optimalizační plánovač spolu s precedenčním plánovačem řeší rozložení paralelních a sekvenčních větví dávkové úlohy na jednotlivé výpočetní zdroje.

Úkolem optimalizačního a prioritního plánovače je také vyřešité problém dynamického přeplánování úloh. To se uplatňuje při doplnění další úlohy do již spuštěného procesu, nebo při výpadku některého z výpočetních zdrojů. Primárním vstupním parametrem pro optimalizační plánovač je kapacita výpočetního zdroje. Metodika určení tohoto parametru je popsána v kapitole 9.2.6.

Pro návrh vhodných parametrů plánovače byly zkoumány některé metaheuristické algoritmy. Úkolem bylo nalezení nejvhodnějšího algoritmu a určení rozumně dlouhého času t_{scd} pro plánovací běh. Časový úsek pro nalezení optimálního řešení má výrazný vliv na celkový čas zpracování. Je nutné zvolit kompromis mezi kvalitou rozvrhu a časem potřebným k naplánování úlohy.

Čas t_{scd}, byl určen z výsledků měření. Byl měřen celkový čas pro 1 000, 3 000 a 10 000 úloh, pro časový interval plánovacího běhu $0, 10, 60$ a 300 [s]. Z naměře-

ných hodnot byla metodou lineární regrese určena závislost optimální délky běhu plánovacího algoritmu a počtu úloh uvolněných do zpracování, pro které má být připraven plánovací rozvrh.

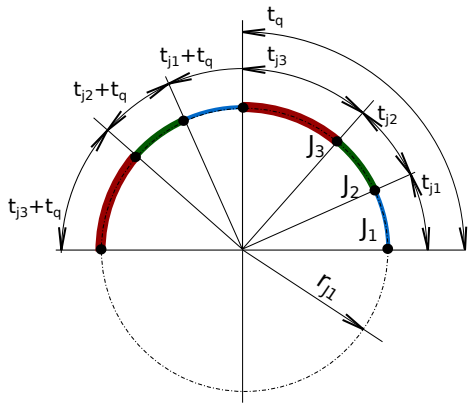
$$t_{scd} = aN + b \tag{7.1}$$

Závislost času t_{scd} a počtu úloh uvolněných do zpracování je popsána rovnicí přímky. Konstanty $a = 0.0122$ a $b = 20.69$ vycházejí z měření MAKESPANu úloh pro jednotlivé plánovací strategie. Metodika a postup měření je popsán v kapitole 10.4.

Určení časového intervalu aktivity úlohy t_j

Pro spravedlivé rozdělení systémových prostředků jednotlivým úlohám je nutné definovat vlastosti prioritního plánovače. Ten je řešen jako cyklická (Roud Robin) fronta. Hlavním parametrem této fronty je určení času t_j pro přepínání kontextu jednotlivých současně běžících úloh a určení strategie pro férovou obsluhu všech běžících procesů.

Časový interval přidělený úloze má vliv na chování prioritního plánovače. Volba krátkého časového intervalu výrazně zvyšuje režii na přepínání kontextu plánovače a také zvyšuje riziko vyhladovění procesů. To může nastat tak, že následujícímu procesu nejsou po dobu času t_j k dispozici žádné výpočetní zdroje, protože jsou zaneprázdněny zpracováním předchozích úloh. Volba dlouhého časového intervalu t_j limitně degraduje cyklickou frontu na FIFO, která bude jednotlivé dávkové úlohy zpracovávat sekvenčně.



$t_{j1},\ldots,t_{jn}$	čas aktivity úlohy [1,...,n]
t_q	nejdelší možný časový úsek aktivity úlohy
$t_j = t_q$	pro úlohu s nejvyšší prioritou
r_{j1}	rádius cyklické (round robin) fronty
$J_1,\ldots,J_n$	fronta úloh uvolněných do zpracování

Obrázek 7.3: Přidělení času aktivity úlohy

Vstupní parametry:

t_n	předpokládaný čas běhu úlohy – zadáno uživatelem
$P \in (1,\ldots,5)$	priorita úlohy, $P = 1$; nejnižší priorita úlohy
t_q	nejdelší možný časový úsek pro aktivitu úlohy
N	počet úloh uvolněných do zpracování

Výstupní parametry:

t_j	čas aktivity úlohy

Časový interval pro přepínání úloh:

$$t_q = \frac{\sum_{n=1}^{N} t_n}{N} \tag{7.2}$$

Čas přidělený úloze:

$$t_j = t_q \left(\frac{1}{P_{max} - P + 1} \right) \tag{7.3}$$

Zátěž gridové služby je přímo úměrná počtu aktivních úloh. Toto je znázorněno na obrázku 7.3, zvětšováním poloměru r_{jN} kružnice cyklické fronty prioritního plánovače.

Cyklus prioritního plánovače je řešen tak, že délka vlákna (konstanta *thread_q*) obsluhujícího proces cyklické fronty je násobena časem t_j. Tím je definován časový interval, po který je aktivní konkrétní úloha n z množiny aktivních úloh $\{1, \ldots, N\}$. Konstanta *thread_q* je načtena z konfiguračního souboru aplikace a její povolený rozsah je 6000 až 60000 milisekund. Fragment kódu v příkladu 1, řízení prioritního plánovače znázornuje.

Příklad 1.

```
/**
       * metoda pro rizeni casoveho kvanta thread_loop
 * obsluhy prioritniho planovace
       *
 * @throws Exception
 * @throws RemoteException
 */
public void notifyOrSleepThread(String message) throws RemoteException,
    Exception {
  if (t_precedence == null) {
    return;
  }
  synchronized (t_precedence) {
    if (message.equals(IConstants.WAKE_UP_THREAD)) {
      t_precedence.notify();
    }

    if (message.equals(IConstants.GET_TASKS_AND_SLEEP)) {
      if (cspQueue.size() != 0) {
            /* Uvolni job dle precedencniho grafu */
        DBJobListPrecedenceGetterInterface jobListPrecedence = cspQueue.poll();
        /* Dle priority a casu tq nastav rychlost threadu
         * ----------------------------------------------
         * tj = tq (1 / Pmax - P + 1)
         * thread_loop = tj * thread_q
         */
        this.thread_loop = threadLoopSpeed(jobListPrecedence.getJob_priority());
        this.initCSPSchedulerStrategy(jobListPrecedence
            .getJob_key0());
        if (cspQueue.size() > 0) {
          t_precedence.wait(0);
        }
      }
    }

    if (message.equals(IConstants.SLEEP_THREAD)) {
      t_precedence.wait(0);
    }
  }
  if (message.equals(IConstants.KILL_THREAD)) {
    if (cspQueue.size() == 0) {
      t_precedence.interrupt();
    }
  }
}

/**
       * @param job_priority
```

```
     * @return
50   */
    private int threadLoopSpeed(int job_priority) {
52    int thread_speed;
      int maxPriority;
54    try {
        thread_speed = Integer.parseInt(config.getThreadLoopInMilisecond());
56      maxPriority  = Integer.parseInt(config.getMaxPriority());
      } catch (NumberFormatException e) {
58      thread_speed = this.thread_q; //[ms]
        maxPriority = 5;
60    }

62    if (job_priority == 0) {
        job_priority = 1;
64    }
      thread_speed = 1 / (maxPriority - job_priority + 1) ;
66    // minimalni povolene casove kvantum thread_speed = 6 sekund
      return thread_speed <= (this.thread_q / 10) ? (this.thread_q / 10) : thread_speed ;
68  }
```

src/priority_thread.java

Pro výjimku vyhladovění procesu byla zvolena strategie krátkodobého plánovače operačních systémů (Linux) [95]. Priorita procesu u něhož došlo k překročení intervalu obsluhy (zestárnutí procesu - aging), je zvýšena na $P+1$. Následná obsluha tohoto procesu vrací prioritu na původní hodnotu.

8 Definice a distribuce úloh

Job Submission Description Language - JSDL je obecná specifikace vydaná organizací Open Grid Forum pro řízení a distribuci dávkových úloh v prostředí Gridu. Současná verze 1.0 má také definovánu podporu POSIX.

JSDL skript je textový dokument, jehož syntaktická pravidla odpovídají normě Extensible Markup Language (XML). Tím je dána snadná čitelnost, přenositelnost a možnost úpravy tohoto dokumentu.

Plánování a distribuce úloh jsou základními prvky gridových služeb. Podmínkou kvalitní a uživateli akceptovatelné gridové služby je nástroj, který umožní snadnou definici úlohy, její distribuci do prostředí gridové infrastruktury a také snadné sledování průběhu vlastního zpracování. Velmi významnou vlastností je také co nejméně omezujících podmínek, které musí úloha splněovat, aby mohla být zařazena do zpracování. Uživatel Gridu musí mít k dispozici určitou svobodu, a neměl by být, pokud možno svazován například licenčními podmínkami nebo podobnými překážkami při definici svých úloh [68].

Jednou z možností je využití POSIX normy, která poskytuje rozsáhlé možnosti při využití standardních programů pro komunikační rozhraní a distribuci úloh. Dále zajišťuje velmi dobrou přenositelnost aplikací na různých typech systémů.

Distribuce úloh v prostředí Gridu

Pro distribuci dat, programů a komunikaci mezi výpočetními zdroji a komponentou plánovače gridu, bylo uvažováno o maximálním využití stávajícího programového vybavení systémů, které mají implementováno rozhraní POSIX. Toto rozhraní zajišťuje kromě přenositelnosti programových objektů také kompatibilitu programů

a příkazů systémové konzoly. Bylo tím sledováno to, aby měl uživatel možnost volby, a nebyl p5itom příliš svazován omezujícími požadavky ze strany gridové služby. Toto řešení také umožňuje využití služeb programů (Ansys, MathWorks, Scilab), takže uživatel není nucen psát speciální programy pro řešení konkrétních úkolů. To má výhodu v možnosti využití standardního a uživateli dobře známého prostředí.

Toto řešení je perspektivní právě v prostředí lokálních gridů, budovaných například na univerzitách nebo firmách, kde je zajištěna jistá standardizace výpočetní techniky, programového vybavení a systémových programů.

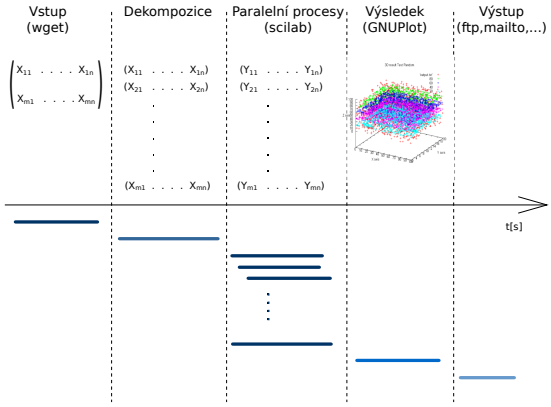
V příkladu níže je navržena úloha, která programem *wget* načte data z různých datových úložišť pomocí protokolu (Hypertext Transfer Protocol (HTTP)), dále využije program *gnuplot* a výsledek v podobě vygenerovaného grafu pošle uživateli do adresáře */home/userid001*. O výsledku běhu úlohy informuje gridový plánovač pomocí webové služby. Průběh této úlohy je znázorněn na obrázku 8.1.

Na tomto konkrétním příkladu je ukázáno, že zadavatel úlohy (uživatel gridu) nemusel psát žádné speciální programy, ale vystačil si se standardním a jemu dobře známým prostředím.

Ve druhém příkladu je znázorněna možnost, jak využít JSDL pro spuštění série dávkových úloh, pro výpočet například rozsáhlé matice. Zde je využito rozšíření specifikace JDSL, která umožňuje danou problematiku řešit. Níže uvedený příklad vygeneruje a připraví plánovači 20 úloh, které jsou evidovány pod unikátním textitjob_id, a které také pod tímto identifikátorem komunikují s plánovačem gridu.

Nástroje pro podporu JSDL

g-Eclipse JSDL PlugIn rozšiřuje Eclipse Project `eclipse.org` a poskytuje podporu gridových služeb. g-Eclipse je rozšíření pro uživatele Gridu, operátory a vývojáře Gridových aplikací. Toto prostředí poskytuje služby, které jsou nezávislé na konkrétním gridovém prostředí. Architektura pluginů umožňuje vývojářům rozšířit funkčnost g-Eclipse o nové funkcionality.



Obrázek 8.1: Příklad úlohy s podporou POSIX specifikace.

g-Eclipse - filozofie podpory

- **uživatel gridu** - nemá podrobnou znalost o Gridových technologiích. Podpora pro tohoto uživatele, umožňuje spustit aplikaci a monitorovat průběh úlohy
- **operátor gridu** - má podrobnou znalost Gridové infrastruktury. Operátor Gridu má k dispozici prostředky, pro řízení lokálních zdrojů, a také zdrojů, které poskytují tzv. virtuální organizace
- **vývojový pracovník** - expert na programování gridových aplikací má k dispozici nástroje pro vývoj, trasování a odladění aplikace.

Knihovna syntaktického analyzátoru JSDL

Součástí projektu Intranet Gridu byla navržena a odladěna knihovna syntaktického analyzátoru JSDL. Ta na základě vstupního JSDL (XML) konfiguračního souboru úlohy, vygeneruje příslušný skript, pro spuštění dávkové úlohy na výpočetním zdroji. Uživatel má možnost v JSDL skriptu určit typ operačního systému. Na základě toho se následně generuje také typ skriptu dávkové úlohy.

Příklad 1.

```
<jsdl:JobDescription>
  <jsdl:JobIdentification>
    <jsdl:JobName>Gnuplot001</jsdl:JobName>
      <jsdl:Description>JSDL - Simple example</jsdl:Description>
  </jsdl:JobIdentification>
  <jsdl:Application>
    <jsdl:ApplicationName>gnuplot</jsdl:ApplicationName>
    <jsdl-posix:POSIXApplication>
 <jsdl-posix:Executable>gnuplot</jsdl-posix:Executable>
 <jsdl-posix:Argument>test_gpu.plt</jsdl-posix:Argument>
 <jsdl-posix:Output>userid@server01.utb.org//home/userid</jsdl-posix:Output>
     <jsdl-posix:Error>http://vm57.os.zps:8080/jgrid/result.do</jsdl-posix:Error>
     <jsdl-posix:WallTimeLimit>1000</jsdl-posix:WallTimeLimit>
     <jsdl-posix:UserName>userid</jsdl-posix:UserName>
     <jsdl-posix:GroupName>utb</jsdl-posix:GroupName>
    </jsdl-posix:POSIXApplication>
  </jsdl:Application>
  <jsdl:Resources>
    <jsdl:TotalCPUCount>
     <jsdl:Exact>1.0</jsdl:Exact>
    </jsdl:TotalCPUCount>
  </jsdl:Resources>

  <!-- data download -->
  <jsdl:DataStaging name="jcuda_iter_k.dat">
  <jsdl:FileName>jcuda_iter_k.dat</jsdl:FileName>
  <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
  <jsdl:DeleteOnTermination>false</jsdl:DeleteOnTermination>
  <jsdl:Source>
      <jsdl:URI>wget --tries=45 http://vm25.os.zps/~plukasik/gnuplot/jcuda_iter_k.dat<
    /jsdl:URI>
  </jsdl:Source>
  </jsdl:DataStaging>


  <!-- data download -->
  <jsdl:DataStaging name="test_gpu.plt">
  <jsdl:FileName>test_gpu.plt</jsdl:FileName>
  <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
  <jsdl:DeleteOnTermination>false</jsdl:DeleteOnTermination>
  <jsdl:Source>
    <jsdl:URI>wget --tries=45 http://vm25.os.zps/~plukasik/gnuplot/test_gpu.plt</
    jsdl:URI>
  </jsdl:Source>
  </jsdl:DataStaging>

  <!-- data upload -->
  <jsdl:DataStaging name="test_gpu.png">
```

```
47    <jsdl:FileName>test_gpu.png</jsdl:FileName>
      <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
49    <jsdl:DeleteOnTermination>true</jsdl:DeleteOnTermination>
      <jsdl:Target>
51      <jsdl:URI>scp test_gpu.png plukasik@vm25.os.zps:/home/plukasik/test_gpu.png</
        jsdl:URI>
      </jsdl:Target>
53    </jsdl:DataStaging>
      </jsdl:JobDescription>
55  </jsdl:JobDefinition>
```

src/priklad01.jsdl

V příkladu 1 je generován skript, který pro manipulaci s daty používá standardní systémové programy (wget, scp, curl, gnuplot). Výsledek zpracování oznámí webové službě gridu. Při ukončení úlohy zruší v pracovním adresáři soubor test_-gpu.png.

Příklad 2.

```
#! /bin/bash
2 echo "Gnuplot001 - jsdl script generator"
wget --tries=45 http://vm25.os.zps/~plukasik/gnuplot/jcuda_iter_k.dat
4 wget --tries=45 http://vm25.os.zps/~plukasik/gnuplot/test_gpu.plt
gnuplot test_gpu.plt
6 scp test_gpu.png useridk@server04.utb.org:/home/userid/test_gpu.png
rc=$?
8 curl -d jobname=Gnuplot001 -d rc=$rc http://vm57.os.zps:8080/jgrid/result.do
rd test_gpu.png
```

src/priklad02.sh

Příklad 2 znázorňuje, jak využít JSDL pro generování více instancí jedné úlohy. Zde je patrný mechanismus, jak jsou jednotlivé úlohy identifikovány a následně posílány do zpracování.

V příkladu 3 je demonstrována možnost generování paralelních úloh, které jsou následně rozplánovány a poslány do zpracování.

Příklad 3.

```
1   <jsdl:JobDescription>
      <jsdl:JobIdentification>
3       <jsdl:JobName>MatrixMult</jsdl:JobName>
        <jsdl:Description> Simple example for parallel matrix multiplication</
      jsdl:Description>
```

```
      <jsdl:JobAnnotation>0001</jsdl:JobAnnotation>
      <jsdl:JobProject>PROJEKT_A001</jsdl:JobProject>
    </jsdl:JobIdentification>
    <jsdl:Application>
      <jsdl:ApplicationName>Multiplication of the Matrix</jsdl:ApplicationName>
      <jsdl:ApplicationVersion>001</jsdl:ApplicationVersion>
      <jsdl-posix:POSIXApplication>
        <jsdl-posix:Executable>java -jar matrixmult.jar</jsdl-posix:Executable>
        <jsdl-posix:Argument>-infile</jsdl-posix:Argument>
        <jsdl-posix:Argument>inputNNN.dat</jsdl-posix:Argument>
        <jsdl-posix:Argument>-outfile</jsdl-posix:Argument>
        <jsdl-posix:Argument>outputNNN.dat</jsdl-posix:Argument>
        <jsdl-posix:Error>
            http://vm57.os.zps:8080/jgrid/result.do
        </jsdl-posix:Error>
      </jsdl-posix:POSIXApplication>
    </jsdl:Application>
    <jsdl:DataStaging name="matrixmult.jar">
      <jsdl:FileName>matrixmult.jar</jsdl:FileName>
      <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
      <jsdl:DeleteOnTermination>false</jsdl:DeleteOnTermination>
      <jsdl:Source>
        <jsdl:URI>
    wget -tries=45  http://vm25.os.zps/~plukasik/matrixmult/matrixmult.jar
        </jsdl:URI>
      </jsdl:Source>
    </jsdl:DataStaging>
    <jsdl:DataStaging name="inputNNN.dat">
      <jsdl:FileName>inputNNN.dat</jsdl:FileName>
      <jsdl:CreationFlag>dontOverwrite</jsdl:CreationFlag>
      <jsdl:DeleteOnTermination>false</jsdl:DeleteOnTermination>
      <jsdl:Source>
        <jsdl:URI>
            wget --tries=45 http://server01.os.org/~userid001/inputNNN.dat
        </jsdl:URI>
      </jsdl:Source>
    </jsdl:DataStaging>
    <jsdl:DataStaging name="outputNNN.dat">
      <jsdl:FileName>outputNNN.dat</jsdl:FileName>
      <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
      <jsdl:DeleteOnTermination>true</jsdl:DeleteOnTermination>
      <jsdl:Target>
        <jsdl:URI>
            scp test_gpu.png useridk@server04.utb.org:/home/userid001/outputNNN.dat
        </jsdl:URI>
      </jsdl:Target>
    </jsdl:DataStaging>
  </jsdl:JobDescription>
  <sweep:Sweep>
    <sweep:Assignment>
```

```
55        <sweep:Parameter>substring(/*//jsdl-posix:Argument[2], 6, 3</sweep:Parameter>
          <sweepfunc:Loop sweepfunc:end="20" sweepfunc:start="1" sweepfunc:step="1"/>
57      </sweep:Assignment>
    </sweep:Sweep>
59    <sweep:Sweep>
      <sweep:Assignment>
61        <sweep:Parameter>substring(/*//jsdl-posix:Argument[4], 7, 3</sweep:Parameter>
          <sweepfunc:Loop sweepfunc:end="20" sweepfunc:start="1" sweepfunc:step="1"/>
63      </sweep:Assignment>
    </sweep:Sweep>
65</jsdl:JobDefinition>
```

src/priklad03.jsdl

Příklad 3 vygeneruje 20 úloh (viz příklad 4 , připravených pro distribuci do prostředí výpočetního Gridu.

Příklad 4.

```
#! /bin/bash
echo "job_010 - jsdl script generator"
wget --tries=45 http://vm25.os.zps/~plukasik/matrixmult/matrixmult.jar
wget --tries=45 http://server01.os.org/~userid001/input010.dat
java -jar matrixmult.jar -infile input010.dat -outfile output010.dat
```

src/priklad04.sh

9 Praktická část

V oblasti strojírenské výroby se setkáváme se dvěma základními úlohami, které ve své podstatě mají zásadní vliv na konečný úspěch při uplatnění výrobku na trhu.

První úlohou je návrh a vývoj výrobku. Zde jsou definovány zásadní vlastnosti a funkcionality, které určují kvalitu a spolehlivost. To následně určuje obchodní úspěch výrobku.

Druhá úloha řeší organizační stránku. To znamená, jak navržený výrobek co nejlépe a co nejvíce efektivně vyrobit. Společným jmenovatelem pro obě tyto základní úlohy je požadavek na spolehlivou a výkonnou výpočetní podporu.

V oblasti návrhu výrobku, jsou konstrukční kanceláře postaveny před poměrně náročnou úlohu, která využívá metodiku virtuálního prototypu k definici optimálních vlastností výrobku, aniž by byl předem fyzicky realizován. Tento nový přístup představuje podobnou revoluci, jako CAD/CAM systémy na začátku masivního rozvoje personální výpočetní techniky v osmdesátých a devadesátých letech minulého století. Stejnou výzvu představuje výše zmíněná perspektivní oblast virtuálního prototypu. Zde jsou výrazně zastoupeny dvě disciplíny. A to využití netradičních materiálů a HIL simulace [104].

Oblast netradičních materiálů, například využití karbonových vláken pro konstrukci speciálních uzlů strojů (rychlostní vřetena, rotační hřídele), nebo speciální hmoty tlumící vibrace významných konstrukčních prvků (základny, portály, vřetenové jednotky), vyžadují mnohem vyšší dimenzi znalostního inženýrství, ale také nové přístupy k výpočtům pevnostních a dynamických vlastností těchto komponent.

Oblast HIL svým specifickým přístupem (pevnostní výpočty v přímé zpětné vazbě s dynamikou pohonů a řídicích systémů) klade velmi vysoké nároky na dovednosti a znalosti.

HIL je základní technikou principu virtuálního prototypu. Na matematickém modelu klíčového prvku stroje jsou virtuálně testovány jeho mechanické a dynamické vlastnosti. Výstupy z virtuálního prototypu umožňují optimalizovat návrh a korigovat případné defektní prvky. Zásadní přínos této metody spočívá v tom, že umožňuje testovat a optimalizovat vlastnosti výrobku před jeho realizací. Metodika virtuálního prototypu také umožňuje simulace v extrémních podmínkách, které při testování skutečného výrobku obvykle nepřicházejí v úvahu díky možnosti destrukce a velkých materiálních škod. Současná praxe ukazuje, že vývojové a konstrukční kanceláře se zaměřují při využití metodiky HIL pouze na některé klíčové uzly výrobku, nikoliv na model výrobku jako celku. To vychází z praktického poznatku, že výsledná kvalita (například tuhost nebo přesnost) závisí jen na některých základních komponentech, které lze do jisté míry podrobně matematicky popsat a definovat tak jejich vlastnosti. Limitujícím faktorem je také výpočetní kapacita systému [103] [104].

Podmínkou pro úspěšné zvládnutí problematiky netradičních materiálů i HIL simulace je přístup k dostatečně dimenzovanému výpočetnímu systému.

Činnost, která je pro výrobní firmu neméně důležitá, je navržený výrobek optimálně vyrobit a uplatnit na trhu. Zde představuje klíčovou roli ERP. Myšlenka využití Gridu v této oblasti vychází z faktu, že stávající výpočetní kapacita nebyla schopna zajistit dostatečnou podporu pro všechny úlohy plánování a řízení. To se projevilo zejména tím, že řada úloh, zařazených do dávkového zpracování nebyly včas dokončeny, což výrazně snižovalo kvalitu a spolehlivost celého plánovacího procesu. Tato situace navíc vznikla v době, kdy strojírenská výroba procházela vleklou krizí. Veškeré investice byly výrazně sníženy a nákup výkonnější techniky nepřipadal v úvahu. Tato zkušenost také ukázala hlavní příčinu nedostatečného výpočetního výkonu. Jedním z důsledků krize je výrazné snížení výroby, které by se mělo proporcionálně projevit také na snížení potřeby výpočetních zdrojů. Praxe naopak ukazuje, že značně narůstá aktivita v oblasti návrhu výrobku a také snaha uplatnit výrobek na trhu i za cenu improvizace a zásadních technických úprav. To výrazně zvyšuje zátěž všech systémů.

Proto bylo navrženo, aby koncepce některých dávkových úloh byla přepracována tak, aby tato úloha byla v čase rozložena na dílčí části. To následně umožní distribuci do do výpočetního gridu. Toto ukázalo, že Grid je vhodný nástroj pro optimalizaci celkové zátěže systému. Díky časovému rozložení jednotlivých částí úlohy, optimalizuje nárazové zatížení, které je pro dávkové úlohy typické.

Výsledkem bylo, že řešení plánování v okamžiku vzniku požadavku, také poskytovalo přesnější výsledky. Umožnilo okamžitou reakci na jakoukoliv změnu. Na druhé straně však také ukázalo jistou nevýhodu. Díky dekompozici Material Requirement Planning (MRP) algoritmu do režimu on-line plánovacích fragmentů úlohy, docházelo k příliš častým změnám, na které nebyli schopni pružně reagovat lidé do plánovacího procesu zapojení. Plán se doslova měnil pod rukama. Praxe ukázala, že dávkový režim, který navržený plán po jistou dobu zafixuje je výhodnější. Proto se následně od tohoto řešení upustilo a ponechala se v on-line režimu jen úloha kalkulace nákladů výrobku, která nemá žádné dopady pro následné procesy plánování, nákupu a distribuce materiálů a komponent pro výrobu.

Grid - nástroj pro optimalizaci zátěže ERP systému

Cílem, pro tuto oblast nasazení gridové služby, bylo navrhnout a realizovat řešení, které nahradí základní dávkové úlohy. Ty jsou spouštěny v určitých cyklech, přičemž v průběhu pracovního dne je velmi nesnadné najít vhodný plán pro jejich práci. Tyto úkoly obvykle sekvenčně skenují celou databázi a snaží se nalézt rovnovážné stavy mezi požadavky a skutečností. To obvykle představuje vysoké požadavky na výkonovou rezervu systému. Mezi základní a velmi náročné úlohy na zatížení systému, patří například.

- **Plánování výroby** – MRP algoritmus, jehož hlavním úkolem je bilance požadavků (plánu výroby), výrobních a materiálových zdrojů [89].
- **Kalkulace výrobků** – výpočet plánované ceny výrobku, na základě znalosti struktury výrobku (kusovníku) a ceny komponent vstupujících do výrobku.

- **Low-Lewel Code (LLC)** – je definován jako stupeň nejnižšího umístění komponenty ve stromové struktuře kusovníku [101].

Koncepce Gridu samozřejmě musí splňovat podmínku možnosti dalšího rozšíření i na jiné typy úloh. Hlavními kritérii pro návrh, byly následující požadavky.

- Nezávislost na platformě hardwaru a operačního systému.
- Nezávislost na databázové platformě.
- Snadná implementace klienta.
- Snadná definice jiného typu úlohy.
- Možnost řešení více různých typů úloh najednou.
- Možnost instalace na IBM iSeries.

Návrh Gridu počítal s nasazením v rámci firemního intranetu. Předpokládá se, že případné útoky z vnějšku budou zachyceny firewallem. Bezpečnostní pravidla vlastního systému jsou definovány pomocí Java Security Manageru, který umožňuje detailně konfigurovat veškeré restriktivní politiky pro konkrétní aplikaci [81]. Pravidla a definice bezpečnostní politiky aplikace je popsána v kapitole 9.3

Grid, jako nástroj pro optimalizaci zátěže byl navržen tak, aby byl hardwarově nezávislý. Proto byla zvolena Java s využitím Java Remote Method and Invocation (RMI) protokolu pro vzájemnou komunikaci mezi jednotlivými uzly Gridu a také pro možnost volat vzdálené metody objektů z jednoho virtuálního prostředí JVM na jiném počítači. Výhodou RMI je, že lze přistoupit ke vzdálenému objektu, jako k místnímu [80] [79] [78].

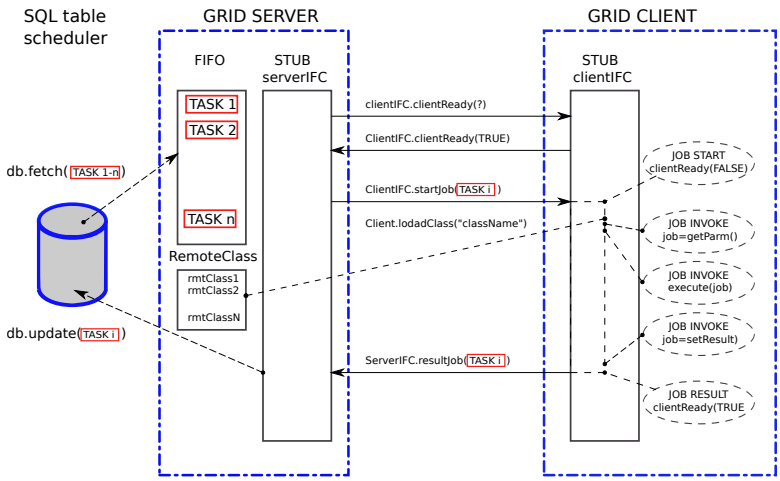
Nad daty systému plánování a řízení výroby byly definovány Structured Query Language (SQL) triggery [49] [87], které zachytávají události s vlivem na změnu konzistence dat.

Zachytávání změn pomocí triggerů, umožnilo nahradit dávkové plánovací úlohy, on-line přeplánováním konkrétní položky, u níž došlo k jakékoliv změně, která má vliv na stavovou rovnici plánovacího procesu (výdej, příjem, uzavření výrobní

dávky). To znamená, že toto řešení udržuje plánovací databázi trvale v konzistentním stavu.

Událost, která je zachycena triggerem je zaznamenána do plánovače gridu, který se následně postará o předání úkolu prvnímu klientovi, který je k dispozici. Výpočetní zdroj Gridu provede výpočet a výsledek vrátí plánovací databázi (obr. 9.1).

Nevýhodou tohoto řešení je zdánlivé vyšší zatížení síťového provozu. Výpočetní zdroj Gridu má v tomto případě vlastnosti „tlustého klienta“, kdy vlastní výpočet se provádí nikoliv na serveru, ale právě na jeho straně. Toto, řešení však umožnilo optimalizovat rozložení zátěže hlavního serveru. Jednak tím, že se část výpočetního výkonu přenesla mimo server, ale hlavně proto, že jednotlivé výpočty byly lépe rozloženy v časové ose. Měřením zátěže sítě (systémový program iptarf), nebyl prokázán dramatický nárůst síťového provozu.



Obrázek 9.1: Schema komunikace v gridu.

Pro tuto verzi Gridu byl zvolen plánovač, jehož algoritmus uplatňuje prioritní rozvrhování úloh. V rámci jednotlivých prioritních stupňů se následně uplatňuje metoda FIFO fronty. Smyčka plánovače úloh provádí optimalizaci na systémové zdroje serveru dle následujícího schématu.

- Primární optimalizace výkonových nároků řízením rychlosti smyčky a vybavování jednotlivých úkolů v závislosti na zatížení CPU. Jedná se v tomto

případě o klasickou regulační úlohu proporcionálního regulátoru se zápornou zpětnou vazbou. Pokud naroste zatížení hardwaru, smyčka plánovače úloh se zpomalí.

- Strategie přidělování úloh je řízena jejich prioritami. Například pro plánování výroby musí být nejprve známy LLC, pak teprve může proběhnout plánovací úloha. Proto mají vyšší prioritu než plánovací algoritmus.
- Každý spuštěný výpočetní zdroj má vůči serveru stejnou prioritu. Rozlišuje se jen počet spuštěných vláken jednotlivých procesů, které je menší nebo nejvýše rovno počtu jader CPU výpočetního zdroje.

Důležitým kritériem pro plánovač bylo dodržení jen prioritního řazení jednotlivých typů úloh (LLC, MRP, kalkulace výrobku).

Tato koncepce gridového middleware byla úspěšně použita v praxi a umožnila oddálit investice do nového serveru pro plánování a řízení výroby. Toto řešení však neumožňovalo nasazení jako nástroje pro výpočty a simulace, protože byl jednostranně zaměřen na volání konkrétních metod, které řešily jen určité typy úloh v rámci ERP systému. Toto řešení bylo následně přepracováno a byl navržen jiný princip, který umožní použít tuto službu, an jakýkoliv typ úlohy.

Grid v oblasti experimentálních výpočtů a simulací

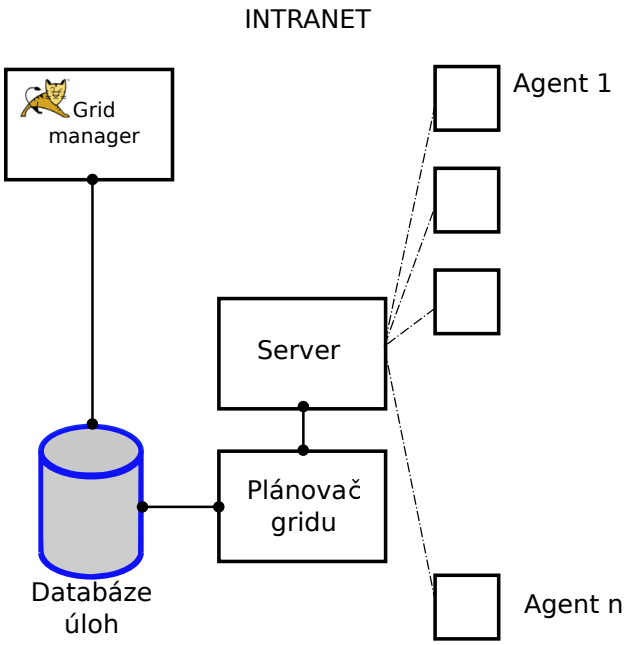
Podmínkou kvalitního návrhu Gridu v oblasti experimentálních výpočtů a simulací je rozhraní, které umožní uživateli snadnou definici úlohy, následně její distribuci, spuštění a také snadné sledování průběhu vlastního zpracování [34] [32]. Uživatel gridu by měl mít k dispozici určitou svobodu a neměl by být, pokud možno svazován například vlastnostmi operačního systému nebo podobnými překážkami při definici svých úloh. Služba Gridu musí umožnit uživateli využití standardních programů, které pro svoji práci běžně používá. Jednou z možností jak toho docílit je POSIX [44], které poskytuje rozsáhlé možnosti pro definici, spuštění a sledování průběhu zpracování úlohy. Dále zajišťuje velmi dobrou přenositelnost aplikací

na různých typech systémů. Proto byla původní koncepce Gridu přepracována dle schématu (obr. 9.2).

Z původního návrhu byla využita pouze transportní vrstva technologie Internet Inter-Orb Protocol (RMI/IOOP) [39] pro komunikaci mezi serverem a jednotlivými zdroji gridové služby. Nově byl přepracován také plánovač, jehož algoritmus je řešen formou pluginu. To umožní zkoumat vlastnosti a chování různých plánovacích strategií.

Role plánovače gridu

Plánovač úloh musí být navržen tak, aby splňoval určité kvalitativní parametry. Míra kvality zahrnuje řadu rozměrů, jejichž význam se může lišit podle oblasti použití (malý Intranet Grid nebo rozsáhlé heterogenní gridové služby), požadavků uživatele (priorita úloh) a dokonce i konkrétní aplikace [29].



Obrázek 9.2: Blokové schéma Intranet gridu.

Zvolený plán a jeho metodika určuje pravděpodobnost úspěšné realizace zadané úlohy. Kvalita plánovacího procesu je závislá na volbě a rozměru kompromisních

řešení. To je také důvod pro volbu různých alternativních přístupů k plánovacímu procesu. Hlavní požadavek optimalizace, tedy rozvržení úloh, na co nejvíce zdrojů s cílem dosažení nejkratšího času, musí jednoznačně respektovat kritérium spolehlivosti. To je vždy v protikladu s primárním požadavkem, tj. optimalizací úlohy na nejkratší čas. Samotný algoritmus plánovače však musí také splňovat určitá kritéria proveditelnosti a časové realizovatelnosti návrhu. To znamená, správnou volbu kompromisu mezi přesností návrhu a délkou výpočtu tohoto návrhu [56].

Úkolem serveru je udržovat konzistentní data o připojených výpočetních zdrojích. Datová struktura popisu zdroje Gridu obsahuje základní informace o jeho vlastnostech. Tato struktura je přímo k dispozici plánovači gridu. Komunikace mezi plánovačem gridu, serverem a grid managerem je řízena pomocí jednoduchých systémových zpráv. Každá změna, která má vliv na konzistenci právě naplánovaných úkolů, přinutí vytvořit nový plán. K těmto změnám patří start nebo vypnutí výpočetního zdroje nebo plánování nové úlohy. V závislosti na zatížení, případně počtu změn je rozhodnuto o přeplánování regenerativním, to znamená nový plán pro všechny úlohy, nebo jen selektivní změna úloh, které jsou zatíženy touto změnou.

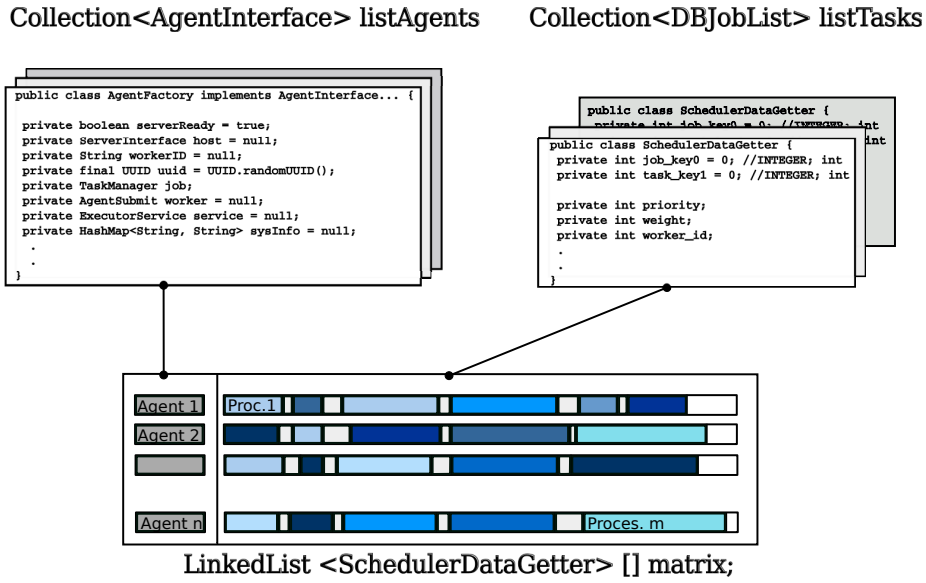
Jádrem plánovače jsou dvě datové kolekce. Kolekce *listAgents* udržuje aktuální stav všech právě přihlášených výpočetních zdrojů (obr. 9.3). Tato datová struktura také udržuje popis hardwarové a softwarové architektury každého aktivního výpočetního zdroje, takže plánovač má k dispozici informace, které používá při přidělování jednotlivých úkolů. Například při požadavku výpočtu na grafickou kartu, přidělí takovou úlohu právě zdroji, který ji má instalovánu.

Druhá datová kolekce *listTasks* obsahuje seznam všech úloh, které jsou uvolněny do zpracování. Kolekce *listAgents* a *listTasks* jsou vstupní parametry pro algoritmus plánovače, který následně vyrobí optimalizovaný plán dávkových úloh. Vlastní plánovací algoritmus je řešen jako plugin objekt, který může být v rámci výzkumu vlastností jednotlivých plánovačů kdykoliv vyměněn.

Pro plánovač jsou k dispozici následující data.

- Priorita úlohy.
- Délka trvání úlohy.
- Pořadí spuštění úlohy.

Všechny tyto tři parametry jsou zadány uživatelem při uvolnění úlohy do zpracování s pomocí aplikace gridmanager, která je součástí Intranet Gridu. Precedenční graf úlohy je řešen přidělením váhy (1 až N), která následně rozhoduje o pořadí zpracování (obr. 9.4).



Obrázek 9.3: Programové interfejsy plánovače gridu.

Synchronizace úloh

Princip dávkových úloh (batch process) je historicky spjat s technologií mainframů, které jsou nasazovány v oblastech, které vyžadují nejvyšší stupeň spolehlivosti, bezpečnosti a výkonu. Požadavkem pro systémy, kde jsou zpracovávány rozsáhlé dávkové úlohy jsou vysoká odolnost proti poruchám – (fault tolerance) a škálovatelnost (scalability). Jádrem dávkové úlohy je časová posloupnost, v níž se sekvenčně vykonávají jednotlivé, na sebe navazující části úlohy. Podmínkou je, že časová posloupnost těchto částí musí být synchronizována. Nelze poslat do zpracování tu část, která očekává výsledek předchozího zpracování, a které dosud nedokončilo činnost.

Parametry dávkové úlohy jsou definovány prioritou, dobou trvání a logickou posloupností jednotlivých vláken (tasků) úlohy. Tyto tři parametry představují

základní informace pro plánovač gridu. Na základě těchto údajů se plánovač snaží vytvořit nejvíce vhodnou posloupnost průběhu celé úlohy. Pro plánování úloh na výpočetní zdroje se používají optimalizační kritéria, jejichž základním parametrem je čas. Hlavním optimalizačním kritériem je minimalizace MAKESPAN úlohy, to znamená optimalizace celkové časové délky plánu. V prostředí reálných systémů, dochází velmi často k dynamickým změnám, které mají zásadní vliv na kvalitu plánovacího procesu. Změny se týkají nejen dostupnosti jednotlivých výpočetních zdrojů, ale také úloh, pro které se má stanovit rozvrh. Další problém představuje stanovení časových nároků plánované úlohy.

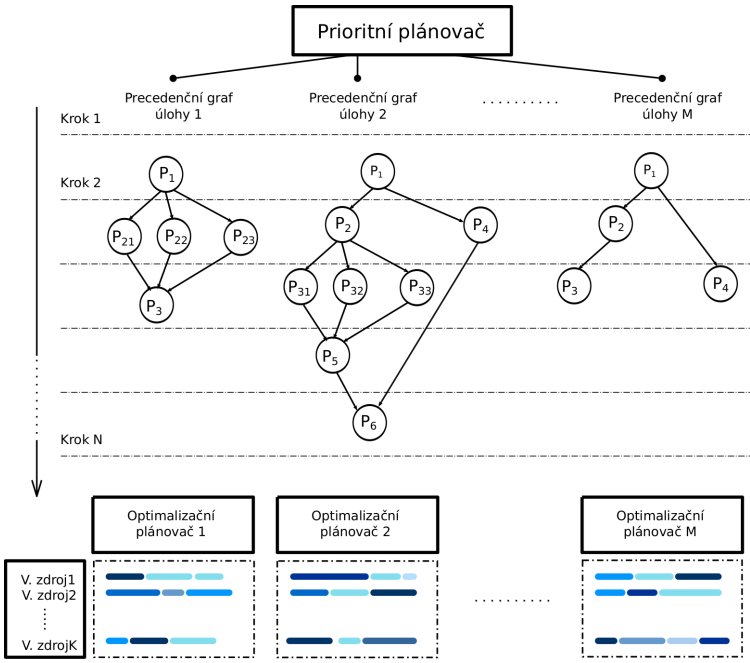
Plánovač Gridu

Pro rozsáhlé úlohy, bylo navrženo jiné řešení, které umožní lépe řídit nejen priority jednotlivých úloh, ale také lépe rozvrhnout MAKESPAN jednotlivých dávkových úloh. Je zde využit hlavní tzv. prioritní plánovač, který má za úkol optimalizovat celý proces rozplánovaných dávkových úloh, na základě priority, která je vstupním parametrem a sekundární tzv. optimalizační plánovače, jejichž úkolem je minimalizace MAKESPANu každé dávkové úlohy. Vstupem pro optimalizaci je časový údaj, který zadá uživatel při požadavku uvolnění úlohy do zpracování, nebo časový údaj z historie transakcí, v případě, že tato úloha již někdy byla zpracována. Jako optimalizační plánovač byla využita knihovna *OptaPlanner* (Constraint Satisfaction Solver) [15].

Výhodou této koncepce je vyšší přehlednost a transparentnost algoritmu plánovače, protože byly logicky rozčleněny role jednotlivých plánovačů.

Očekávaná zlepšení návrhu optimalizovaného návrhu plánovače:

- Rozdělení plánovacích rolí.
  - master - prioritní plánovač
  - slave - optimalizační plánovače
- Vyšší přehlednost plánovacího procesu.
- Možnost použít různé typy plánovacích strategií pro různé typy úloh.



Obrázek 9.4: Optimalizovaný návrh principu plánovacího rozvrhu dávkových úloh.

Definice úlohy

Pro definici a uvolnění úlohy do zpracování, byla použita webová aplikace, která umožňuje uživateli definovat některé důležité (povinné) parametry úlohy a zařadit tuto úlohu do zpracování. Součástí webové aplikace je komponenta JSDL analyzátoru, jejíž vstupní hodnotou je JSDL skript, který definuje vlastnosti úlohy a požadavky na systémové zdroje nutné pro úspěšné zpracování. Výstupem je skript pro spuštění úlohy na výpočetním zdroji gridu.

Pro JSDL analyzátor byla použita knihovna *Apache XMLBeans*, která na základě XML Schema Definition (XSD) schématu XML dokumentu usnadňuje definovat rozhraní jednotlivých elementů.

V příkladu je navržena úloha, která systémovým programem *wget* načte data z různých datových úložišť, spustí program gnuplot, přičemž výsledek v podobě vygenerovaného grafu pošle uživateli do adresáře /home/userid001/.

Příklad 1.

```
<jsdl:JobDescription>
  <jsdl:JobIdentification>
    <jsdl:JobName>RandomTest</jsdl:JobName>
    <jsdl:Description>Random test UNIX</jsdl:Description>
    <jsdl:JobAnnotation>0001</jsdl:JobAnnotation>
    <jsdl:JobProject>PROJECT_RANDOM_TEST</jsdl:JobProject>
  </jsdl:JobIdentification>
  <jsdl:Application>
    <jsdl:ApplicationName>random test</jsdl:ApplicationName>
    <jsdl:ApplicationVersion>001</jsdl:ApplicationVersion>
    <jsdl-posix:POSIXApplication>
      <jsdl-posix:Executable>java -jar TestRandom.jar</jsdl-posix:Executable>
      <jsdl-posix:Argument>100</jsdl-posix:Argument>
      <jsdl-posix:Argument>randomNNN.dat</jsdl-posix:Argument>
      <jsdl-posix:UserName>plukasik</jsdl-posix:UserName>
      <jsdl-posix:GroupName>fai-utb</jsdl-posix:GroupName>
    </jsdl-posix:POSIXApplication>
  </jsdl:Application>
  <jsdl:Resources>
    <jsdl:OperatingSystem>
      <jsdl:OperatingSystemType>
        <jsdl:OperatingSystemName>linux</jsdl:OperatingSystemName>
      </jsdl:OperatingSystemType>
    </jsdl:OperatingSystem>
  </jsdl:Resources>
  <jsdl:DataStaging name="random.jar">
    <jsdl:FileName>random.jar</jsdl:FileName>
    <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
    <jsdl:DeleteOnTermination>false</jsdl:DeleteOnTermination>
    <jsdl:Source>
      <jsdl:URI>wget --tries=45 http://vm21.os.zps/~plukasik/random/TestRandom.jar</
   jsdl:URI>
    </jsdl:Source>
  </jsdl:DataStaging>
  <jsdl:DataStaging name="randomNNN.dat">
    <jsdl:FileName>randomNNN.dat</jsdl:FileName>
    <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
    <jsdl:DeleteOnTermination>true</jsdl:DeleteOnTermination>
    <jsdl:Target>
      <jsdl:URI>mv -v randomNNN.dat ~/temp/random</jsdl:URI>
    </jsdl:Target>
  </jsdl:DataStaging>
</jsdl:JobDescription>
<sweep:Sweep>
  <sweep:Assignment>
    <sweep:Parameter>substring(/*//jsdl-posix:Argument[2], 7, 3</sweep:Parameter>
    <sweepfunc:Loop sweepfunc:end="20" sweepfunc:start="1" sweepfunc:step="1"/>
  </sweep:Assignment>
```

```
  </sweep:Sweep>
</jsdl:JobDefinition>
```

src/rand.jsdl

Na základě této JSDL definice je vygenerován skript pro příkazový interpret bash. Pro manipulaci s daty používá standardní systémové utility a programy (*wget, scp, curl, gnuplot*). Výsledek zpracování oznámí webové službě gridu. Při ukončení úlohy zruší v pracovním adresáři soubor *testgpu.png* protože v definici JSDL je u tohoto souboru požadavek

Příklad 2.

```
#! /bin/bash
echo "rand_001 - jsdl script generator"
wget --tries=45 http://vm21.os.zps/~plukasik/random/TestRandom.jar
java -jar TestRandom.jar 100 random001.dat
mv -v random001.dat ~/temp/random
```

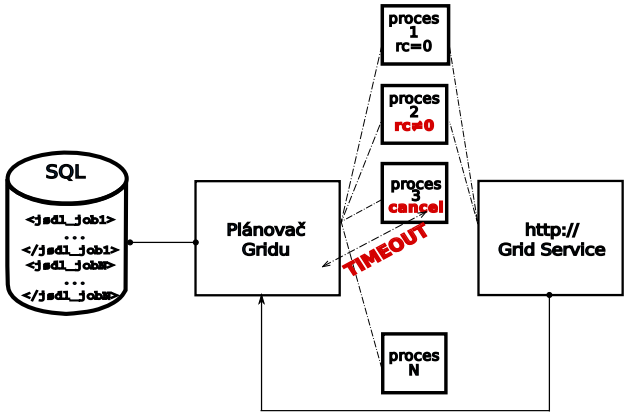
src/rand001.sh

Předkládaný text si klade za cíl popis a netradiční využití JSDL jako generátoru dávkových úloh. Obecně je JSDL nasazováno v oblasti gridových služeb, které mají integrovány všechny funkcionality pro manipulaci s daty a komunikaci s okolím. Primární snahou bylo, co nejvíce využít standardní a existující programové vybavení a systémové služby. To do jisté míry uživateli umožňuje vyšší kreativitu při návrhu a zpracování úlohy. V prvé fázi se také může zdát, že použití JSDL je v této oblasti zbytečně složité. Kód JSDL proti kódů příkazového interpretu bash je na první pohled mnohem složitější a může se zdát, že je neefektivní. Výhoda JSDL je však okamžitě patrná v okamžiku kdy potřebujeme například distribuovat tisíce paralelních instancí jediné úlohy.

Návrh fault tolerance systému v prostředí Intranet Gridu

Pro řešení chybových stavů v prostředí Intranet Gridu byla navržena metodika, která využívá principu zjednodušeného checkpointingu. Systém zpráv checkpointingu využívá vlastnosti POSIXu pro komunikaci mezi procesy. Následně je také řešena výjimka timeout, v případě výpadku některého z výpočetních uzlů.

V průběhu řešení kolizního stavu se neprovádí rollback a recovery od bodu vzniku poruchy, ale do zpracování se posílá zpět celá část úlohy, na níž výjimka vznikla. Například při násobení rozsáhlé matice a výpadku jednoho z prvků C_{ij} se pošle do zpracování znovu řádek A_i a sloupec B_j. Toto řešení je méně vhodné, než klasický rollback mechanismus, výhodou je však snadnost realizace.

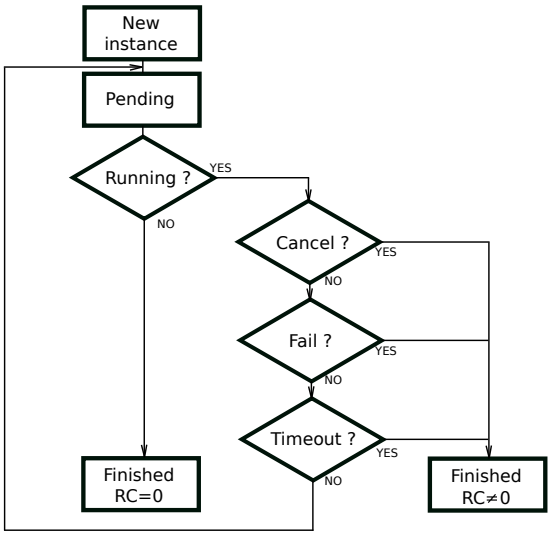


Obrázek 9.5: Princip checkpointingu v prostředí Intranet gridu.

Životní cyklus úlohy představuje aktuální stav instance programu, který je právě vykonáván (nebo doby, než bude vykonán). Každá takto rozplánovaná instance, má přímý vliv na celkový výsledek. Informace o stavu úlohy, které jsou distribuovány jednotlivým výpočetním zdrojům Gridu jsou velmi důležité pro plánovač, který je zodpovědný za úspěšné vyřešení úlohy. Přitom platí, že v řadě poruchových stavů nemusí být tato informace plánovači doručena. Stavy úloh jsou popsány v obrázku 9.5.

Plánovač Gridu musí mít přehled o každém stavu právě zpracovávané instance. Z obrázku 9.6 vyplývá, že výsledek zpracování nabývá tři možné stavy. Tyto stavy lze ještě rozdělit do dvou skupin, tedy stav, kdy úloha dokáže vrátit zadavateli výsledek zpracování (správný nebo chybný) a stav, kdy úloha výsledek zpracování vrátit nedokáže. Například pro přerušení běhu úlohy.

Stav, kdy úloha nedokáže vrátit výsledek zpracování, musí být ošetřen na straně serveru. A to tak, že server dostane informaci o přerušení komunikačního kanálu, nebo pomocí uživatelem zvolené maximální délky zpracování úlohy (timeout), prohlásí úlohu za ztracenou, (obr. 9.6). Následně tuto informaci předá plánovači, který



Obrázek 9.6: Vývojový diagram průběhu dávkové úlohy v prostředí gridu.

tuto instanci pošle ke zpracování znovu, ale jinému zdroji. Existuje samozřejmě nebezpečí, že příčinou nesnází je právě zpracovávaná úloha, která nemá správně řešeny chybové stavy, (memory leak, divide zero). Pak takováto úloha musí být vyřazena ze zpracování. Proto i tento stav musí plánovač Gridu řešit, například tak, že může tuto úlohu poslat do zpracování opakovaně jen několikrát, dle předem nastaveného scénáře. Pokud se tuto úlohu nepodaří vyřešit opakovaně, je následně plánovačem vyřazena ze zpracování.

JSDL má pro řízení úloh implementovánu podporu, která umožní definovat rozsahy výpočetních zdrojů (délka zpracování, počet spuštěných vláken, velikost diskového prostoru), které při překročení některého z parametrů úlohu nespustí. Dále lze definovat způsob komunikace s plánovačem Gridu při zpracování návratových kódů, případně výpadku některé ze spuštěných instancí [11].

Test pro vyhodnocení kapacity výpočetního zdroje

Parametr hodnota kapacity výpočetního zdroje byl zvolen jako jedno ze vstupních kritérií pro plánovač. Při návrhu Gridu se vycházelo z předpokladu, že služba bude pracovat v rámci firemního intranetu s omezeným počtem zdrojů. Proto byla do koncepce návrhu plánovače zahrnuta také možnost paralelního běhu úloh na

jednotlivých výpočetních zdrojích, obvykle kancelářských počítačích. Pro zajištění jisté spravedlnosti při plánování zátěže, je nutná informace o jeho výkonnosti tak, aby některé z uzlů Gridu nebyly přetěžovány nebo naopak zanedbávány.

K základním kritériím pro hodnocení gridové služby patří následující parametry.

- Doba běhu - čas vykonávání konkrétní úlohy.
- Doba odezvy - čas potřebný k získání výsledku dané úlohy.
- Propustnost - množství práce vykonané za jednotku času.

Nejvýznamnější položku pro optimální práci rozsáhlého systému je kritérium propustnosti. Do této kategorie spadá měření propustnosti řady prvků, jako propustnost sítě, Input/Output (I/O) operace nebo odezva systému. Kritérium propustnosti, má zásadní vliv na spolehlivost celého systému [12]. Tyto metriky však ocení celkovou výkonnost infrastruktury gridu, ale neřeknou nic o výkonnosti jednotlivých zdrojů.

Proto byla navržena metodika, která na základě velmi jednoduchého a krátkého testu změří dobu odezvy (response time) konkrétního výpočetního zdroje. V prvé fázi probíhá měření na CPU, a pokud je k dispozici, tak následně i na grafickém procesoru. Tento postup byl zvolen proto, že je velmi obtížné posoudit reálnou výkonnost výpočetního systému jen na základě výčtu hardwarového vybavení. Samozřejmě i takto zvolená metodika není zcela objektivní, protože právě v okamžiku spuštění tohoto testu může testovaný zdroj zpracovávat jinou výkonově náročnou úlohu, která výsledek testu výrazně zkreslí. Jako řešení se nabízí spouštění tohoto testu v jistých intervalech, rozumně navržených tak, aby se zbytečně neplýtvalo strojovým časem výpočetního zdroje a následné vyhodnocení průměrné hodnoty. Tato hodnota se předává serveru v okamžiku startu zdroje. Je součástí datové struktury *AgentInterface* (obr. 9.3). Díky měření na konkrétním zdroji jsou do určení kapacity zahrnuty všechny části výpočetního řetězce, včetně vlivu komunikační sítě a I/O operací. Cílem takto navržené metodiky je nalezení optima pro rozdělení výpočetního výkonu pro potřeby Gridu a pro potřeby uživatele tak,

aby nebyl nijak omezován. To v praxi znamená, že plánovač Gridu nesmí dopustit, aby jakýkoliv zdroj byl zatížen na maximální výkon.

Hodnota kapacity výpočetního zdroje byla zvolena jako jeden z důležitých vstupních parametrů pro optimalizační plánovač popsaný v kapitole 7.3. Jedná se o bezrozměrnou konstantu, která hodnotí výkonové vlastnosti výpočetního zdroje.

Vstupní veličiny pro optimalizaci rozvrhu úloh jsou.

- Velikost paměti.
- Propustnost sítě.
- Parametry výpočetních jednotek CPU a GPU.

Algoritmus pro vyhodnocení kapacity výpočetního zdroje byl zvolen dle následujících kritérií. Počet úloh spuštěných paralelně nesmí přesáhnout počet jader procesoru. Tato úvaha neposuzuje možnost běhu dalších úloh (například úlohy uživatele výpočetního zdroje). Hodnota kapacity CPU je měřítkem četnosti zadávání úkolů konkrétnímu zdroji a hodnota kapacity GPU kvantifikuje vlastnosti výpočetního zdroje pro využití paralelních výpočtů na GPU.

Při návrhu tohoto algoritmu byla brána v úvahu možnost vícevláknového zpracování úkolů a bylo měřeno, jak se výpočetní zdroj bude chovat v případě, že počet požadavků překročí počet jader CPU. Pro tento test byla zvolena aplikace zápisu do vektoru v paměti, přičemž byl porovnáván synchronizovaný i nesynchronizovaný zápis. Z výsledku měření vyplynulo následující. Při překročení počtu úloh na počet jader procesoru samozřejmě dochází k vyčerpání výkonové rezervy a operační systém začne část své režie spotřebovávat na přepínání strojového času pro obsluhu všech spuštěných úloh. S větším počtem vláken tato režie narůstá.

Test vyhodnocení výpočetní kapacity je spuštěn jen jednou v okamžiku startu výpočetního zdroje a byla zvolena metoda popsaná v kapitole 3.5.1. Do měření výpočetní kapacity jsou zahrnuty tři základní parametry.

- Počet jader procesoru.
- výpočetní kapacita CPU.

- výpočetní kapacita GPU, pokud je k dispozici.

Test je navržen tak, že v případě detekce GPU, proběhne celý výpočet pouze v režii CPU jednotky. To znamená, že na CPU běží vektor $[0,\ldots,n]$ a také výpočet $cos(x)-x=0$. Následně proběhne test GPU, a to tak že na CPU běží vektor $[0,\ldots,n]$ a na GPU je řešena iterace $cos(x)-x=0$.

V případě, že není instalována jednotka GPU, je provedena pouze první část testu, to znamená, že vektor $[0,\ldots,n]$ a výpočet $cos(x)-x=0$ běží pouze v režii CPU.

Tabulka 9.2.6 definuje vyhodnocení kapacity výpočetního zdroje na základě časů zpracování na jednotkách CPU a GPU

Určení kapacity výpočetního zdroje

	GPU detekován				GPU nebyl detekován			
CPU time [ms]	1000	500	250	125	1000	500	250	125
C_{CPU}	10	12	13	15	10	12	13	15
GPU time [ms]	100	50	25	15				
C_{GPU}	2	4	10	15				
$C=C_{CPU}+C_{GPU}$	**12**	**16**	**23**	**30**	**10**	**12**	**13**	**15**

Tabulka 9.1: Určení kapacity výpočetního zdroje

Při návrhu tohoto algoritmu byla brána v úvahu možnost vícevláknového zpracování úkolů a bylo měřeno, jak se výpočetní zdroj bude chovat v případě, že počet požadavků překročí počet jader CPU. Pro tento test byl zvolen algoritmus pro zpracování pole $[0,\ldots,n]$ (čtení a zápis), přičemž byl porovnáván synchronizovaný tzv. *safe-thread* a také nesynchronizovaný přístup do paměti. Z měření vyplynulo, že synchronizovaný zápis neměl výrazný vliv na snížení rychlosti vlastního zpracování. Výsledky měření tohoto vlivu jsou popsány v kapitole 10.2.5.

Definice bezpečnostních politik v prostředí Intranet Gridu

Předkládaný projekt Intranet Gridu neočekává rozšíření napříč organizacemi a nemá ambice na budování virtuálních organizací. Hlavní myšlenkou je vybudování, implementace a použití služby bez vysokých administrátorských nároků, která umožní provádět výpočty a plnit výzkumné úkoly, v rámci jediné organizace. Díky tomuto zaměření se také odvíjí nároky na definici a nastavení bezpečnostních politik provozu této služby. Vzhledem k tomu, že se nepředpokládá propojení více organizačních celků, odpadá složitá definice bezpečnostních pravidel napříč těmito organizacemi, ale postačí pravidla definovaná v rámci intranetu.

Pro vstup údajů, definici úlohy a spuštění úlohy slouží intranetová webová aplikace *Gridmanager*. Toto je jediný vstupní bod, který umožňuje uživateli interakci s gridovou službou. Pro autentizaci uživatele byl zvolen Aplikační protokol Adresářových a dotazovacích služeb odvozený od standardu X.500 (OpenLDAP).

Gridová služba je vybudována na Java Platform, Enterprise Edition (J2EE). Javovské objekty disponují abstraktní třídou java.security.Policy. Typicky se pro nastavení bezpečnostní politiky používají dva soubory. Globální, na úrovni JVM, je umístěn v adresářové struktuře instalace <JAVA_HOME> */lib/security/java.policy* a druhý, pro lokální nastavení vlastní aplikace je umístěn přímo v adresářové struktuře této aplikace.

Jádrem objektu Policy je metoda *getPermissions()*, která vrací nastavení bezpečnostní politiky. Pro nastavení rozsahu práv a restrikcí platí pravidlo, že implicitně není povoleno nic. Je na administrátorovi systému aby zvolil pravidla a přístupy k jednotlivým zdrojům [81]. Příklad 1 demonstruje vymezení hranic pro načítání, ukládání a manipulaci s daty tak, aby v případě pokusu o překročení této definice nebo detekce bezpečnostního incidentu byla vyhlášena výjimka, která zamezí této události proniknout dále do systému. Tato výjimka je následně zaznamenána v systémovém logu aplikace.

Příklad 1.

```
grant {
  /* pro ladeni aplikace
   *  permission java.security.AllPermission;
   */

  /* povolen pristup na port 1099 - RMI
  permission java.net.SocketPermission "localhost:1099", "connect,resolve";
  /* povolen pristup na port 80 http */
  permission java.net.SocketPermission "localhost:80", "connect, resolve";
  /* povolen pristup na port 443 https */
  permission java.net.SocketPermission "localhost:443", "connect, resolve";
};

/* povolen pristup do adresaru apache plukasik/jgrid */
grant codeBase "http://vm59.os.zps/~plukasik/jgrid" {
  permission java.io.FilePermission "<<ALL FILES>>", "read";
};

/* povolen pristup do adresaru /home/plukasik/jgrid - podepsano certifikatem */
grant signedBy "e-simgrid",  codeBase "file:/home/plukasik/jgrid",
      principal javax.security.auth.x500.X500Principal "cn=Petr Lukasik, ou=Faculty of
    applied informatics, o=Thomas Bata University, c=CZ" {
  permission java.io.FilePermission "<<ALL FILES>>", "write";
};
```

src/jgrid.policy

Výše uvedený soubor bezpečnostních politik povolí aplikaci Intranet Grid komunikovat na portech 1099 (port pro RMI), 80 (http port) a 443 (http secure port). Dále umožní provádět download z url http://vm59.os.zps/plukasik/jGrid a přístup do adresáře file:/home/plukasik/jgrid. Objekt *Java.Policy* umožňuje transparentně natavit rozsáhlý soubor bezpečnostních pravidel, která lze dynamicky měnit i za provozu aplikace [81].

Při startu aplikace, se musí JVM říct kde, a jaký konkrétní soubor bezpečnostních politik má načíst. To je znázorněno v příkladu 2

Příklad 2.

```
JAVA_POLICY="-Djava.rmi.server.codebase=file://$FILE_PATH/e-simgrid_stubs-2.0.1.jar -
    Djava.security.policy=$FILE_PATH/conf/e-simgrid.policy"
EXEC="/usr/bin/java"
```

src/server.sh

10 Experimentální část

Grid jako podpora ERP systému - distribuce výpočetní zátěže

Měřením zatížení jednotlivých složek síťové infrastruktury byly ověřeny značné rozdíly v zatížení serverů a pracovních stanic. To je znázorněno na obrázcích 10.1 a 10.2.

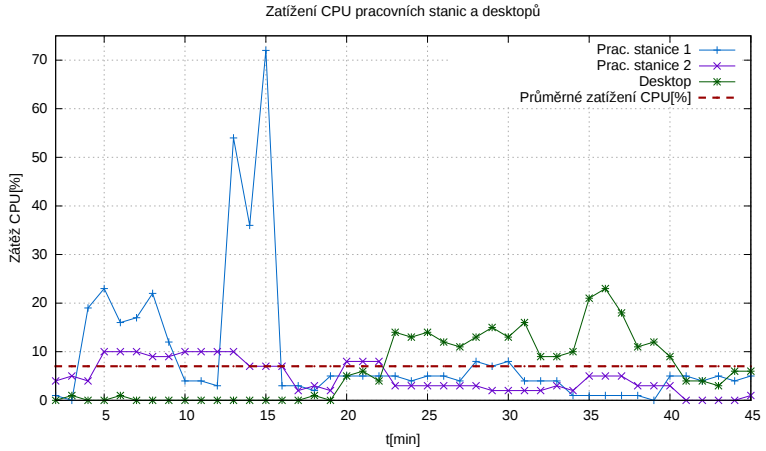
Proto bylo navrženo, aby byla část výpočetní zátěže distribuována prostřednictvím Gridu na stranu jednotlivých výpočetních zdrojů. Bylo zkoumáno, zda se distribucí úloh sníží nároky na výpočetní výkon hlavního serveru, a tím dojde k k lepšímu rozložení zátěže systému.

Měření byla provedena pomocí standardních programů, které jsou součástí každého z operačních systémů. Bylo sledováno zatížení procesoru. Pro IBM iSeries byla využita programová utilita *System i Navigator*. Pro koncové pracovní stanice s OS Linux byl použit *sar* (program pro diagnostiku zátěže CPU) a pro pracovní stanice s OS Windows, systémový program *typeperf*. Byla provedena řada měření, která prokázala nízké průměrné zatížení pracovních stanic. Testy byly prováděny ve standardním uživatelském provozu. Pro získání vzorku zátěže bylo prováděno několik dlouhodobých měření. Prezentován je reprezentativní vzorek grafů 10.1, 10.2 a 10.3.

Naměřené hodnoty

Nejmenší zatížení vykazovaly běžné kancelářské počítače. Měření bylo také provedeno na počítači vývojáře softwaru. Například v průběhu překladu projektu, byly nároky na výpočetní výkon stanice vyšší. Z dlouhodobého sledování stavu však

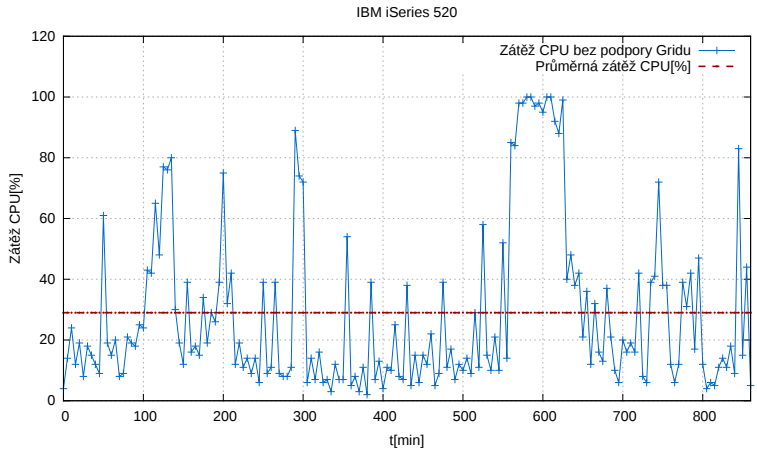
bylo zjištěno, že průměrné hodnoty zatížení v dlouhých časových úsecích nepřekročí 8% celkového výkonu, který má stanice k dispozici (obr. 10.1). U počítačů, které jsou využívány pro běžnou kancelářskou práci, byla hodnota zátěže ještě nižší a pohybovala se v řádu nejvýše 5% [67].



Obrázek 10.1: Zátěž pracovních stanic.

Ovšem výsledky na serveru ukázaly výrazně vyšší hodnoty zátěže. Měření bylo prováděno na serveru IBM iSeries 520. Jeho základní parametry jsou v tabulce 15.1.

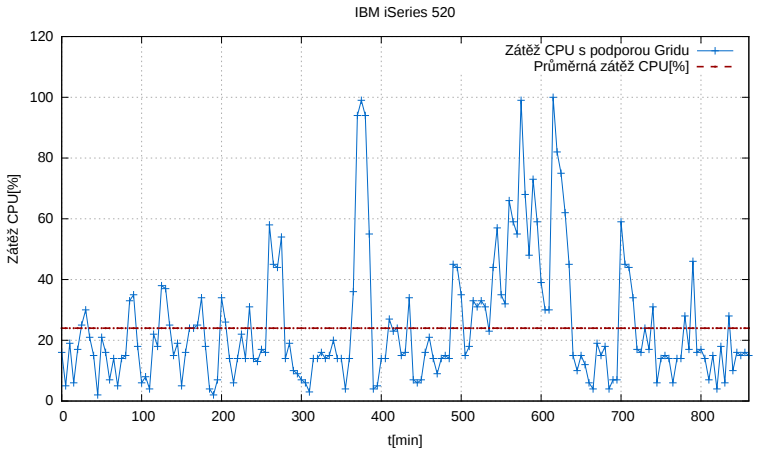
V praxi byla naměřena průměrná hodnota zátěže cca 29%. To je z hlediska ekonomiky provozu serveru velmi dobrá hodnota. Problém však představují úseky, kdy server je zatížen na hranici svých výkonových možností. Tato situace nastává zejména v době zpracování dávkových úloh. Výsledkem je, že výrazně klesá celková dostupnost systému. To představuje zásadní kolizní stav, který je patrný na obrázku 10.2.



Obrázek 10.2: Zátěž serveru bez podpory gridu.

Cílem řešení bylo dosáhnout lepšího časového rozložení výpočetních operací hlavního serveru. Část dávkových úloh, které měly výrazný vliv na zatížení systému, byla přepsána z Report Program Generator (RPG III) do objektů programovacího jazyka Java. To umožnilo fragmentaci dávkových úloh a následnou možnost předání k řešení výpočetním zdrojům gridu. Výhodou tohoto konceptu, je kromě výše zmíněného časového rozložení výkonu, také dosažení on line konzistence sledovaných datových struktur v ERP. Výsledkem bylo lepší rozložení celkové zátěže systému. To vedlo ke zvýšení výkonové rezervy a tím i vyšší dostupnosti služby. Průměrné zatížení hlavního serveru sice pokleslo o relativně malou hodnotu cca 3 až 4 %, ale výrazně se zmenšila plocha výskytu výkonových špiček, a to o hodnotu více jak 30%. Prezentovaná série měření ukázala snížení výkonové zátěže z 29.08% na 24.98% a snížení plochy výkonových špiček o 39.4 % (obr. 10.2, 10.3).

Po instalaci nového serveru IBM iSeries 720 byla ponechána v tomto režimu úloha kalkulace výrobku. Tato metoda prokázala, že gridová služba je dobrý nástroj při řešení výkonových problémů tohoto systému.



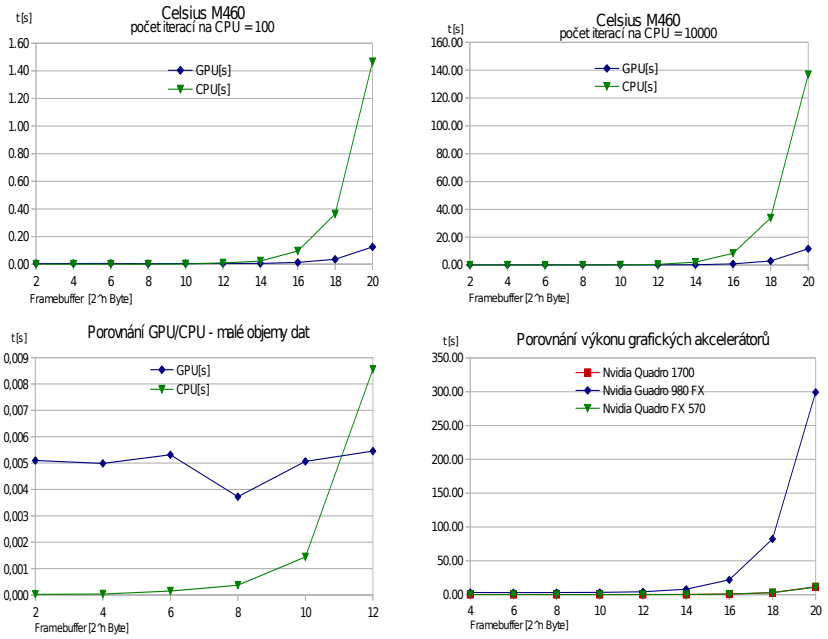
Obrázek 10.3: Zátěž serveru s podporou gridu.

Využití GPU pro distribuované výpočty

Testy byly prováděny na třech různých typech počítačů s různými typy grafických akcelerátorů, které byly k dispozici v letech 2004, až 2008.

Výsledky měření na GPGPU

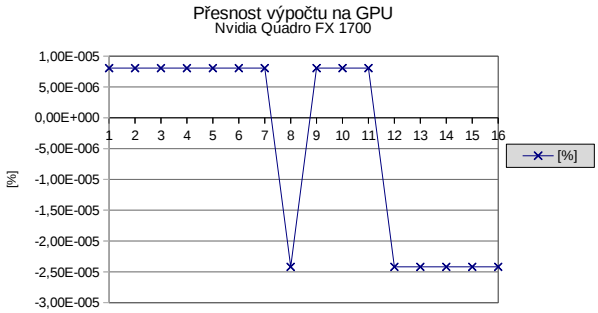
Z výsledku vyplývá nejen dosti značná závislost výpočetního výkonu na konfiguraci GPU jednotky, ale také dosti velká závislost na ovladačích grafické karty. Velmi důležitý závěr vyplývá z faktu, že přesun dat do paměti GPU je relativně časově náročná operace, proto výpočty malých objemů dat jsou na GPU velmi málo efektivní. V těchto případech dokonce CPU vykazuje lepší výsledky. To je patrné z grafu 10.4



Obrázek 10.4: Porovnání výkonnosti GPU/CPU

Menší přesnost výpočtu, která se pochybuje v řádu ±0.00002% (viz tabulka 15.2), je pro většinu technických aplikací naprosto vyhovující. Další problém může představovat relativně obtížný přístup k návrhu paralelní aplikace a také velmi nesnadné ladění shaderu. Bez speciálních nástrojů nemožné, protože GLSL nemá

k dispozici obvyklou ladicí funkci “printf”. (Tato funkce je již například u OpenCL implementována). Rozptyl přesnosti výpočtu na GPU je patrný také v grafu 10.5



Obrázek 10.5: Přesnost výpočtu na GPU

Avšak výsledek měření benchmarkových testů a porovnání výpočetní výkonnosti CPU a GPU jednoznačně ukazuje, že paralelní výpočty na GPU mají své opodstatnění. Již bylo zmíněno, že nad malými objemy dat, dává CPU lepší výsledky, ale situace se dramaticky mění při velkém objemu dat. Tehdy výpočetní výkon GPU výrazně převyšuje CPU. Samozřejmě každá grafická karta má své hardwarové limity, takže následným rostoucím objemem dat bychom museli vstupní vektor zpracovávat po více fragmentech. Pak bychom narazili na řadu omezení, které vyplývají z nárůstu režie na řízení úlohy [13] čímž bychom narazili na výkonový strop klasického výpočetního řetězce. Tato úloha však tento strop nezkoumá. Z výsledků měření vyplývá, že tento typ výpočtu je na grafické kartě zhruba 10 x rychlejší než prostý výpočet na CPU. Výsledek samozřejmě velmi kolísá s hardwarovým vybavením.

Nejhorších výsledků bylo dosaženo na Siemens Celsius 420 s grafickou kartou nVIDIA Quadro 980, kdy poměr výkonu CPU a GPU byl zhruba 1:3. Tento nepříliš přesvědčivý výsledek byl dán především typem ovladače, protože pro tento typ grafické karty již není k dispozici novější verze (rok výroby 2004). Zdánlivě nejlepších výsledků bylo dosaženo na Siemens Celsius W370, kde poměr výkonu mezi CPU a GPU byl u velkých vzorků dat, až 1:100. Tento velmi dobrý výsledek však byl dán poměrně nevyváženou konfigurací GPU a CPU, kdy CPU patřil do méně

výkonné řady. To také vyplývá z výsledků měření, kdy pro velmi malé vzorky dat tento počítač nedokázal vyhodnotit časovou náročnost úlohy (čas počátku a konce úlohy byl stejný). Výsledek měření je v tabulce 15.3

Nejlépe vyvážená a relativně výkonná konfigurace Siemens Celsius M 460 vykazovala poměrně velmi stabilní poměr výkonu CPU a GPU 1:11. Na tomto počítači bylo provedeno více měření, které probíhalo na pozadí, přičemž počítač řešil další uživatelské úlohy. Nutno poznamenat, že s nárůstem objemu dat (maximální délka vektoru 2^{24}, která byla na tomto počítači vyzkoušena) se poměr výkonu GPU proti CPU ještě výrazně zvýšil. Výpočet na CPU v tomto případě trval $2191,1$ sekund a na GPU trval $20,2$ sekundy. To však nebylo zahrnuto do výsledné analýzy, protože tak velký objem dat již nebylo možné na jiných systémech zpracovat aniž bychom museli vstupní vektor zpracovávat po více částech. To by samozřejmě díky nárůstu režie na tuto fragmentaci, zkreslilo výsledek.

Porovnání technologií OpenGL, OpenCL a CUDA

Pro nasazení paralelních výpočtů v prostředí distribuovaných výpočtů byly také zkoumány vlastnosti technologií OpenGL 4.3, OpenCL a CUDA.

Pro porovnání jednotlivých technologií a přístupů byly zvoleny dva testy. První vychází z algoritmu iteračního testu 3 uvedeném v kapitole 3.5.1. Ten umožní test na straně GPU, a přitom také využívá CPU, kde běží smyčka iterace. Byly testovány knihovny OpenGL, OpenCL a CUDA, které výpočty na straně GPU podporují. Hlavním kritériem pro volbu knihovny, byla především snadná aplikovatelnost na většině grafických akcelerátorů různých výrobců a nezávislost na operačním systému.

Dále byl pro porovnání výsledků testován také výpočet 2D Fast Fourier Transform algoritmus (2DFFT) transformace, kde však nebyl dodržen jednotný přístup k algoritmizaci úlohy. Pro OpenGL byl zvolen poměrně složitý shader, kdežto pro OpenCL a CUDA byly využity Fast Fourier Transform (FFT) knihovny, které jsou součástí instalace vývojového prostředí.

Pro hodnocení výpočetní kapacity hardwaru byla navržena metodika, která umožní specifikovat parametry pro plánovač Gridu. Tyto parametry zvýší kvalitu plánovacího procesu a zajistí spravedlivější distribuci zátěže v gridové infrastruk-

tuře. Test výkonnosti hardwaru byl primárně zaměřen na malé gridové služby využívané v rámci firemního nebo univerzitního intranetu, kde je do jisté míry usnadněna jednotnost hardwarové platformy a unifikace programového vybavení.

Výsledky měření a praktické testy vedly také k přehodnocení původní podmínky snadné aplikovatelnosti pro většinu grafických akcelerátorů různých výrobců. Pro podporu výpočtů na GPU byla zvolena technologie CUDA jako nejvíce perspektivní, přestože OpenCL standard je otevřený a multiplatformní.

Důvody byly tyto.

- Velmi dobrá podpora vývoje ze strany výrobce.
- Snadná realizovatelnost úloh paralelních výpočtů v oblasti lineární algebry (knihovna*nVIDIA CUDA Basic Linear Algebra Subroutines (*cuBLAS*)*).
- podpora v oblasti analýzy signálů (knihovna *nVIDIA CUDA Fast Fourier Transform library (*cuFFT*)*).
- Velmi kvalitní vývojové prostředí nVIDIA Nsight (na bázi Eclipse), které má velmi dobře implementovány funkce debuggeru a profileru.

Samozřejmě, nevýhodou technologie CUDA je její jednostranné zaměření na hardwarovou platformu nVIDIA. To však pro gridové služby v rámci firemního intranetu nemusí představovat výrazný problém [77].

Bohužel původně zamýšlená a velmi perspektivní technologie OpenCL, nemá takovou vývojářskou podporu. V testech na kartách nVIDIA, OpenCL vykazovala nejhorší výkonové parametry. Tento závěr však nelze zobecnit, protože nebyly prováděny testy na grafických kartách jiných výrobců, například akcelerátorů ATI od Advanced Micro Devices. Díky rychlému rozvoji v této oblasti však může dojít ke změnám ve prospěch OpenCL a tato volba může být přehodnocena.

OpenGL byla testována proto, aby se dalo posoudit její využití na grafických kartách, které ještě nemají podporu technologií OpenCL a CUDA. Závěr je takový, že pro starší grafické karty je toto řešení jedinou alternativou. Nevýhodou starších grafických akcelerátorů je, že nemají implementován datový typ *double*. Ten je podporován, až v OpenGL 4.3. Podmínkou je však novější grafický akcelerátor. Ten však již má také podporu technologií OpenCL nebo CUDA.

Nevýhodou OpenGL je poměrně složitý návrh a realizace programů pro paralelní výpočty na GPU jednotce. Neexistují žádné knihovny, které by přímo podpo-

rovaly výpočty na této platformě. Proto musí být každá úloha pečlivě analyzována a navržena.

Knihovna OpenGL však dávala nejlepší výsledky z hlediska výkonových parametrů u obou typů testů. Proto může pro jisté speciální úlohy nebo jednoúčelové aplikace představovat velmi dobrou alternativu, knihovnám *cuBLAS* a *cuFFT*.

Realizace výpočtů na GPU

Projekt byl napsán v Eclipse Indigo verze 3.7.1. (OS Ubuntu 12.04 LTS). Pro podporu výpočtů na OpenGL je nutná knihovna *libGLEW* (OpenGL Extension Wrangler Library). Pro měření času je v prostředí Linuxu nutná ještě knihovna *librt*. Snahou bylo prověřit vlastnosti a možnosti GPGPU v systému Linux a Windows. Mac OS X nebyl k dispozici, ale poskytuje stejné možnosti. Seznam grafických stanic, na nichž byly prováděny testy je v tabulce 15.1.

Přesnost výpočtu v aritmetice plovoucí desetinné čárky

Principy výpočtů v plovoucí desetinné čárce jsou definovány v normě IEEE 754. Norma definuje pravidla pro výpočetní operace s formáty typu float 32 bitů a double 64 bitů. Je zaručeno, že tyto operace budou dávat stejné výsledky na všech výpočetních platformách, kde je tento standard implementován. V článku Davida Goldberga [40] je uvedena problematika výpočtů s plovoucí desetinnou čárkou.

Z grafu 10.5 a tabulky 15.2 vyplývá, že výsledek výpočtu na grafické kartě je odlišný, méně přesný než výsledek v případě výpočtu na CPU. Nabízí se otázka, proč se liší přesnost výpočtů na GPU a CPU.

Grafické akcelerátory jsou primárně používány k vykreslování grafické scény. Primárním požadavkem pro vykreslení scény je rychlost výpočtu, přesnost v tomto případě není zásadní.

Výpočty na grafických kartách jsou novým konceptem, který vyžaduje naprosto odlišný přístup pro návrh výpočetních algoritmů, než při využití klasického CPU s integrovaným matematickým koprocesorem. GPU nemá implementovánu virtuální paměť a přerušení. Proto přenos dat a komunikace mezi GPU a CPU probíhá na základě speciálních knihovních funkcí, jejichž princip je popsán v kapitole 3.5.

Nejedná se však o klasické I/O operace. Není také zaručeno, že funkce matematické knihovny CPU a GPU jsou totožné. CPU a GPU mají významně odlišné architektury, které je předurčují pro různé typy úloh.

Grafická procesorová jednotka je vhodná pro zpracování velkých objemů dat, nad nimiž jsou prováděny poměrně jednoduché výpočetní operace. Výpočetní jednotka CPU je obecně mnohem rychlejší (ve smyslu počtu instrukcí na jednu pipeline (instrukční kanál). Proto je výborné právě pro sekvenční zpracování. Grafické akcelerátory GPU, nejsou obecně vhodné pro zpracování úloh, které nemají prospěch z paralelních procesů. [99] Moderní GPU jednotky již mají definován typ *double*, který výpočty na GPU výrazně zpřesňuje.

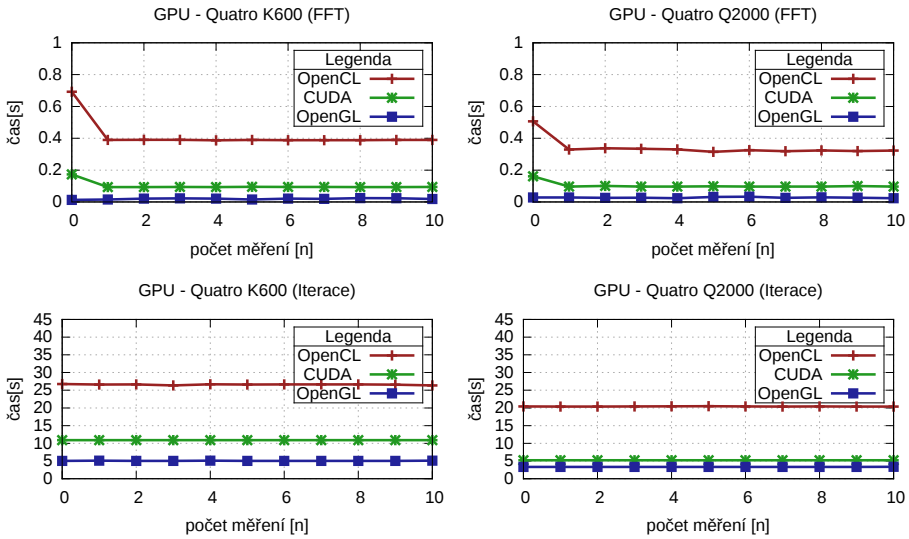
Vyhodnocení testů kapacity výpočetního zdroje

Do testovaných hodnot nebyla zahrnuta významná veličina, a to rychlost I/O operací. Pro zjištění tohoto parametru bychom nevystačili s krátkým testem. Podmínkou byl návrh metody, která zbytečně nezatěžuje testovanou jednotku. Na základě úvahy o hardwarové optimalizaci bylo předpokládáno, že jednotlivé komponenty systému budou proporcionálně vyváženy. To znamená, že výkonnosti CPU, bude odpovídat i dimenze ostatních komponent hardwarového vybavení (diskový subsystém, síťová karta...).

Technologie CUDA a OpenCL jsou přímo pro výpočty na straně GPU navrženy [77] [8]. Knihovna OpenGL má implementovány knihovní funkce, které výpočty na straně GPU podporují také. Ovšem návrh a realizace programu využívající paralelní vlastnosti grafického akcelerátoru je komplikovanější, než přímé použití technologií CUDA nebo OpenCL [71].

Výstupy z jednotlivých měření výpočetní výkonnosti však v těchto případech ukazují, že některé speciální úlohy mohou být při využití OpenGL rychlejší. Test byl prováděn na výše uvedené iterační úloze, jejíž algoritmus byl napsán tak, aby byl pro všechny uvedené technologie stejný.

Pro lepší využití výpočetních zdrojů byla také měřena závislost času úlohy, která je paralelně zpracovávána vícevláknově. Byl porovnáván také synchronizovaný i nesynchronizovaný zápis do paměti. Této úloze byla věnována pozornost také proto, že její algoritmus je použit pro test kapacity při startu výpočetního



Obrázek 10.6: Porovnání výkonnosti GPU/CPU

zdroje. Cílem bylo prozkoumat chování a vlastnosti operačního systému v případě spuštění vícevláknové úlohy, jejichž počet vláken mnohonásobně překročí (limita byla stanovena na 512 vláken) počet jader centrální procesorové jednotky CPU. Pro tento test byla použita úloha popsaná v kapitole 3.5.1. Byl počítán vektor $[1,\ldots,n]$ rovnic $cos(x)-x=0$, kde $n=2^{20}$ a při počtu iterací 100.

V příkladu 1 je uvedena část kódu se synchronizovaným zápisem do paměti a v příkladu 2 je znázorněn kód bez synchronizace zápisu.

Příklad 1.

```
1  /* (non-Javadoc)
   * @see java.lang.Runnable#run()
3  */
  synchronized public void run() { // (4)
5    this.len = this.len/this.threads;
     for (int j = 0; j < this.iter; j++) {
7      int k = this.len * this.thread + this.len;
       int i = this.len * this.thread;
9      for (; i < k; i++) {
         pole[i] = (float)Math.cos(pole[i]);
11     }
     }
13   System.out.println("Thread " +  this.thread + " exit.");
     active = false;
15 }
```

```
17  /**
     *
19   */
    synchronized public void start() {
21    System.out.println("Start:" + this.thread);
      if (t == null) {
23      t = new Thread(this, "Thread-" + Integer.toString(this.thread));
        t.start();
25    }
    }
```

src/src/ref/iter/j_ter.java

Příklad 2.

```
/* (non-Javadoc)
 * @see java.lang.Runnable#run()
 */
public void run() { // (4)
  this.len = this.len/this.threads;
  for (int j = 0; j < this.iter; j++) {
    int k = this.len * this.thread + this.len;
    int i = this.len * this.thread;
    for (; i < k; i++) {
      pole[i] = (float)Math.cos(pole[i]);
    }
  }
  System.out.println("Thread " +  this.thread + " exit.");
  active = false;
}

/**
 *
 */
public void start() {
  System.out.println("Start:" + this.thread);
  if (t == null) {
    t = new Thread(this, "Thread-" + Integer.toString(this.thread));
    t.start();
  }
}
```

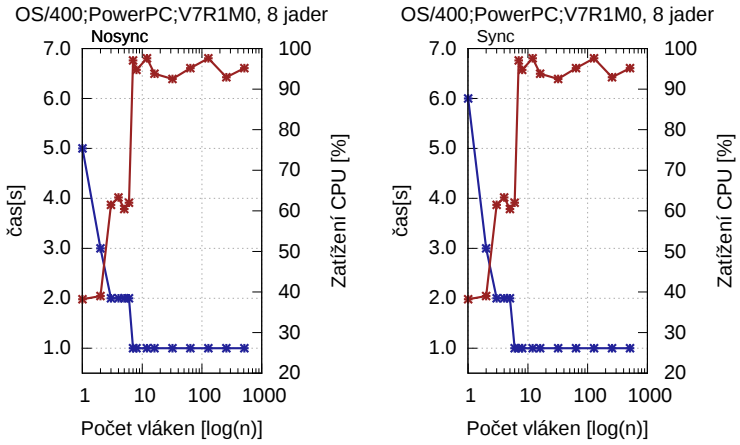
src/src/ref/iter/j_ter.java

Z výsledku měření vyplynulo následující zjištění. Neprojevuje se příliš velký časový nárůst úlohy při synchronizovaném zápisu do paměti. Tento typ zápisu, sice klade vyšší časové nároky na řízení jednotlivých vláken, ale nárůst těchto hodnot není dramatický.

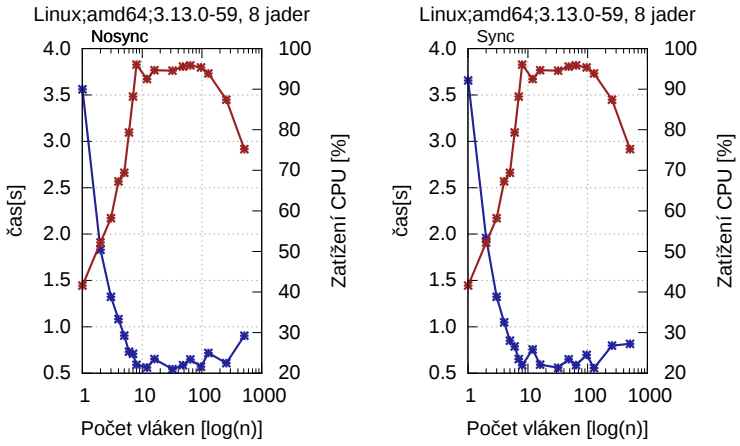
Při měření na systémech Linux a Windows docházelo k mírnému nárůstu času zpracovávané úlohy v případě, že bylo spuštěno více procesů než bylo k dispozici jader procesoru. Při dalším zvyšování vláken, se tento nárůst dále neprojevoval. K následnému nárůstu režie na správu vláken došlo až při překročení 256 vláken procesu[1].

Při měření stejné úlohy na IBM eServeru 720, byl průběh času zpracování a zátěže CPU jiný. To je znázorněno v grafech na obrázku 10.7. IBM eServer vy-

[1] U procesoru s osmi jádry a paměti 16GiB.



Obrázek 10.7: Test zátěže CPU, IBM OS400, 8 jader



Obrázek 10.8: Test zátěže CPU Linux64, 8 jader

kazoval při měření počtu vláken na zatížení systému nejvyšší stabilitu, přestože toto měření probíhalo v reálném provozu. Měření na systémech Linux a Windows probíhala na nezatížených výpočetních zdrojích 10.8.

Zatížení CPU bylo na systému linux měřeno v rámci spuštěného procesu Jávovským objektem *OperatingSystemMXBean* a metodou *getSystemCpuLoad()*. Zatížení na IBM eServeru bylo měřeno systémovým nástrojem *System I Naviga-*

tor, který má implementovánu podporu systémového monitoringu. Měření zatížení CPU při synchronizovaném i nesynchronizovaném zápisu do paměti bylo prováděno v jediném programovém cyklu. Z důvodu názornosti však bylo rozděleno do dvou grafů, protože naměřené hodnoty u obou metod si byly velmi blízke. Výsledné průběhy grafických závislostí splynuly. Levá strana grafu 10.7 i 10.8, znázornuje nesynchronizovaný zápis a pravá strana grafu, zápis synchronizovaný.

Měření vlastností JSDL analyzátoru

JSDL analyzátor byl navržen jako komponenta uživatelského webového rozhraní pro definici a zaplánování úlohy. Byla použita knihovna Apache XMLBeans a DOM XML parser. Pro změření vlastností JSDL analyzátoru byly definovány dvě úlohy.

První úloha generovala s pomocí jednoduché smyčky *sweep loop (1)* množinu $\{1,\ldots,100000\}$ úloh. Byl měřen čas běhu JSDL analyzátoru na dvou různě výkonných pracovních stanicích. Výsledek je v tabulkách 15.4 a 15.5

.

Příklad 1.

```
<sweep:Sweep>
  <sweep:Assignment>
    <sweep:Parameter>substring(/*//jsdl-posix:Argument[2], 6, 5</sweep:Parameter>
    <sweepfunc:Loop sweepfunc:end="99999" sweepfunc:start="1" sweepfunc:step="1"/>
  </sweep:Assignment>
</sweep:Sweep>
```

src/sweep01.jsdl

Druhá úloha měla definovánu třikrát vnořenou smyčku *sweep loop (3)* a generovala úlohy v množině $\{1,\ldots,87000\}$. Výsledek je opět v tabulkách 15.4 a 15.5.

Příklad 2.

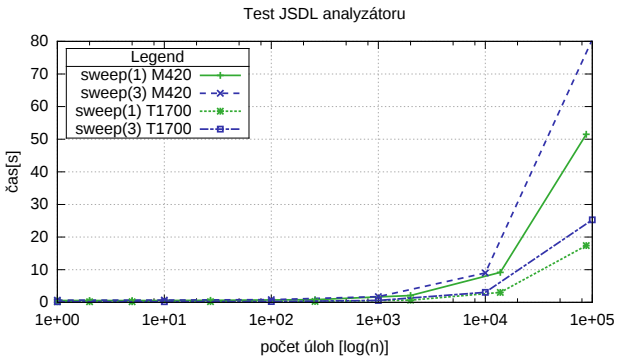
```
<sweep:Sweep>
  <sweep:Assignment>
    <sweep:DocumentNode>
<sweep:NamespaceBinding ns="http://schemas.ggf.org/jsdl/2005/11/jsdl-posix" prefix="
  jsdl-posix"/>
<sweep:Match>
  substring(/*//jsdl-posix:POSIXApplication[1]/jsdl-posix:Argument[4], 5, 2)
</sweep:Match>
    </sweep:DocumentNode>
    <sweepfunc:Values>
<sweepfunc:Value>00</sweepfunc:Value>
<sweepfunc:Value>01</sweepfunc:Value>
<sweepfunc:Value>02</sweepfunc:Value>
    </sweepfunc:Values>
  </sweep:Assignment>

  <!-- LEVEL2.1 -->
  <sweep:Sweep>
    <sweep:Assignment>
<sweep:DocumentNode>
  <sweep:NamespaceBinding ns="http://schemas.ggf.org/jsdl/2005/11/jsdl-posix" prefix=
  "jsdl-posix"/>
  <sweep:Match>
    substring(/*//jsdl-posix:POSIXApplication[1]/jsdl-posix:Argument[6], 7, 2)
  </sweep:Match>
</sweep:DocumentNode>
<sweepfunc:Values>
  <sweepfunc:Value>10</sweepfunc:Value>
  <sweepfunc:Value>11</sweepfunc:Value>
  <sweepfunc:Value>12</sweepfunc:Value>
</sweepfunc:Values>
    </sweep:Assignment>

    <!-- LEVEL3.1 -->
    <sweep:Sweep>
<sweep:Assignment>
  <sweep:DocumentNode>
    <sweep:NamespaceBinding ns="http://schemas.ggf.org/jsdl/2005/11/jsdl-posix"
   prefix="jsdl-posix"/>
    <sweep:Match>
      substring(/*//jsdl-posix:POSIXApplication[1]/jsdl-posix:Argument[8], 9, 2)
    </sweep:Match>
  </sweep:DocumentNode>
  <sweepfunc:Values>
    <sweepfunc:Value>20</sweepfunc:Value>
```

```
        <sweepfunc:Value>21</sweepfunc:Value>
44      <sweepfunc:Value>22</sweepfunc:Value>
    </sweepfunc:Values>
46  </sweep:Assignment>
        </sweep:Sweep>
48    </sweep:Sweep>
  </sweep:Sweep>
```

src/sweep03.jsdl



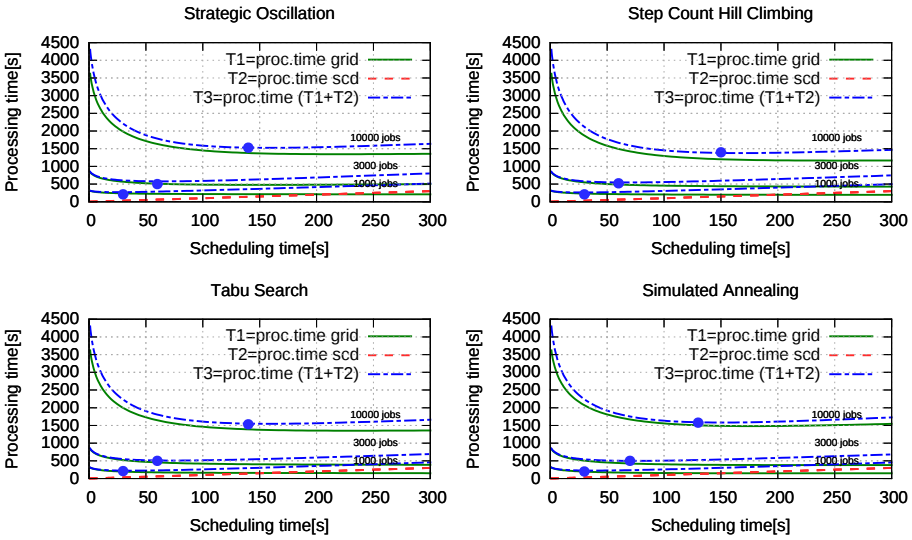
Obrázek 10.9: Zatížení systému při testu JSDL cyklu Sweep

Měření prokázalo nízkou časovou a systémovou náročnost nástrojů pro JSDL. Bylo provedeno pět měření pro každou úlohu. Byl zjištěn lineární nárůst času a malé zatížení systému i při použití DOM XML parseru pro analýzu JSDL. Na základě měření obou typů úloh bylo konstatováno, že přístup k definici úlohy s pomocí JSDL specifikace prokazuje velmi dobré výsledky bez výrazného vlivu na výkonovou zátěž i v okamžiku generování velkého počtu dávkových úloh.

Určení doby běhu plánovacího algoritmu t_{scd}

Pro změření vlastností vlivu plánovací strategie byla zvolena úloha, pro kterou lze relativně snadno určit délku zpracování nezávisle na výkonu výpočetního zdroje. To znamená, že byl zvolen takový algoritmus, který umožní optimalizačnímu plánovači navrhnout optimalizovaný rozvrh úlohy.

Čas plánovacího běhu byl určen z výsledků měření celkového času úlohy. Byl měřen celkový čas pro 1000, 3000 a 10000 naplánovaných procesů a zvoleném časovém intervalu plánovacího běhu 0, 10, 60 a 300 s. Bylo provedeno pět měření pro každý typ algoritmu a časový interval. V grafech na obrázku 10.10 jsou znázorněny průměrné hodnoty z těchto měření. Pokud nebyl plánovací běh uplatněn, optimalizační rozvrh úloh degradoval na frontu FIFO.



Obrázek 10.10: Makespan úlohy pro různé typy optimalizačních algoritmů

Nalezení optimálního času vychází z naměřených hodnot a odečtených z grafu 10.10. Z tabulky 10.1 vyplývá, že naměřené hodnoty času t_{scd} [s] se u jednotlivých algoritmů příliš nelišily. Lineární regresí byla nalezena funkční závislost - přímka, která s dostatečnou přesností popisuje závislost počtu plánovaných úloh a času k návrhu plánovacího rozvrhu úloh. t_{scd}.

Čas běhu plánovacího algoritmu

Počet tasků v úloze	1 000	3 000	10 000
		$t_{scd}[s]$	
Strategic Oscillation	30	60	140
Step Count Hill Climbing	30	60	150
Tabu Search	25	60	140
Simulated Annealing	30	70	135

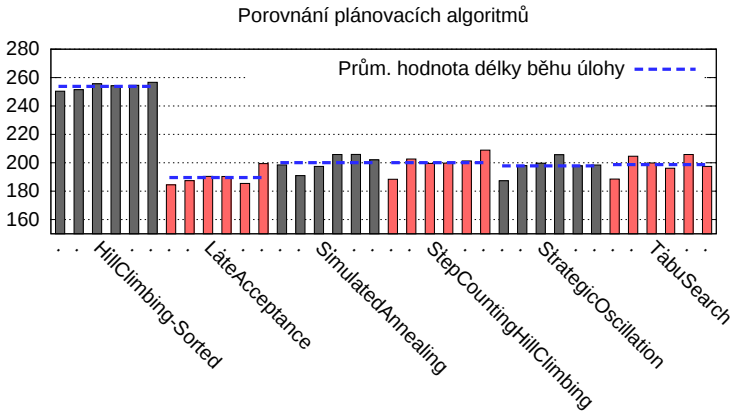
Tabulka 10.1: Čas běhu plánovacího algoritmu $t_{scd}[s]$

$$t_{scd} = 0.0122J + 20.69 \tag{10.1}$$

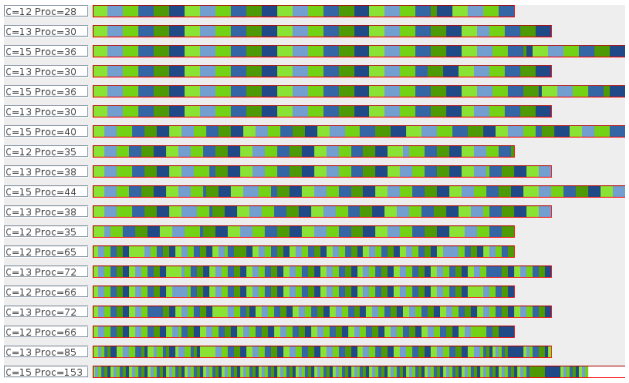
V průběhu měření byl také hodnocen MAKESPAN úloh pro jednotlivé optimalizační algoritmy. Měření probíhalo tak, že každý optimalizační algoritmus měl za úkol navrhnout rozvrh pro 1 000 úloh. Pro každý algoritmus bylo provedeno pět měření. Vyhodnocoval se průměr. Nejlepších výsledků dosáhl Late Acceptance algoritmus. Špatné výsledky vykazoval HillClimbing-Sorted algoritmus, který nad optimalizovaným návrhem prováděl sort dle délky úlohy (znázorněno na obrázku 10.12). Výsledkem bylo, že některým výpočetním zdrojům byl přidělen velký počet krátkých úloh. Díky tomu byly tyto výpočetní uzly zatíženy velkou režií V/V operací. Algoritmy s náhodným rozložením délky úlohy vykazovaly mnohem lepší výsledky. To je znázorněno na obrázku 10.13.

Z grafu 10.11 a obrázků 10.12, 10.13 je patrný vliv optimalizace rozložení všech vstupních hodnot, z nichž je určena kapacita výpočetního zdroje. V případě použití algoritmu HillClimbing-Sorted bylo na relativně výkonný výpočetní zdroj rozplánováno velké množství velmi krátkých úloh. To znamenalo výrazně vyšší režii na síťový provoz a I/O operace tohoto zdroje. Ve výsledku to znamenalo výrazně vyšší čas ke zpracování celé úlohy.

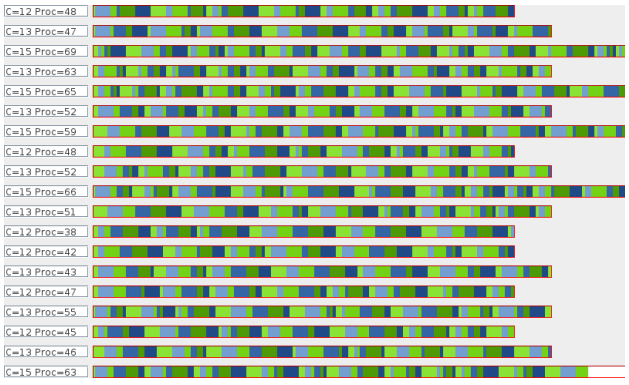
Proto byla pro určení kapacity výpočetního zdroje volena metodika, uvedená v kapitole 9.2.6. Metodika uvedená v [93, 94] je nedostačující. Zabývá se pouze kapacitou CPU a nezahrnuje ostatní důležité komponenty výpočetního zdroje.



Obrázek 10.11: Porovnání plánovacích algoritmů a makespanu úlohy pro 1000 úloh a délku běhu plánovacího algoritmu 20s.



Obrázek 10.12: Rozvrh úlohy pro plánovací strategii HillClimbing - Sorted



Obrázek 10.13: Rozvrh úlohy pro plánovací strategii HillClimbing - UnSorted

11 Závěr

Práce se zabývá návrhem a praktickou realizací gridové služby pro podporu rozsáhlých výpočtů. Současná praxe ukazuje, že se jedná o velmi perspektivní nástroj, který umožní lépe využít stávajících výpočetních zdrojů. Praktická realizace ukázala na řadu velmi zajímavých a rozsáhlých témat z oblasti synchronizace úloh, plánovacích algoritmů a řešení fault tolerance subsystému. Některé z částí, zejména oblast plánování bude předmětem dalšího výzkumu. Dobrým výsledkem je poznatek, že gridová služba může být využita jako podpora pro rozložení špičkových zátěží informačních a plánovacích systémů.

Definice fault tolerance subsystému a jeho vlastností má zásadní vliv na spolehlivost celé gridové služby. Úkolem je nejen zvýšit odolnost systému, ale i rozpoznat různé chybové scénáře a na základě nich zvolit vhodnou strategii obnovy. V předloženém projektu byl navržen a realizován systém, který řeší specifické výpadky a chyby, vznikající v reálném provozu, například vypnutí výpočetního zdroje v průběhu výpočtu nebo chyby programového vybavení. Z důvodu požadavku nižších nároků na systémové prostředky, byla zvolena metodika nesynchronizovaného checkpointu, která řeší případný výpadek nebo chybu tak, že posílá znovu do zpracování celou část úlohy, která chybovou výjimku vyvolala.

Jako podpora vyšší spolehlivosti systému, bylo využito vlastností JSDL specifikace, která ve své definici pamatuje na některé kolizní a chybové stavy a umožňuje úloze předem nastavit některé limitní hodnoty, například velikost paměti, velikost souboru, počet vláken nebo maximální dobu běhu úlohy. To do jisté míry umožňuje plánovači gridu, případně spuštěné úloze, lépe vyhodnotit mezní nebo chybové stavy.

Předkládaný text si dále klade za cíl popis netradičního využití specifikace JSDL, jako generátoru dávkových úloh. Obecně je JSDL nasazováno v oblasti

gridových služeb, které mají integrovány všechny funkcionality pro manipulaci s daty a komunikaci s okolím [3] [4] [5].

Primární snahou bylo, co nejvíce využít standardní a existující programové vybavení a systémové služby. To do jisté míry uživateli umožňuje vyšší kreativitu při návrhu a zpracování úlohy. V prvním náhledu při studiu této problematiky, lze nabýt dojmu, že použití JSDL je pro tuto oblast poměrně složité. Kód JSDL, proti kódů výsledného skriptu interpreteru *bash* je složitější. Výhoda JSDL je však okamžitě patrná v okamžiku kdy potřebujeme například distribuovat tisíce paralelních instancí jediné úlohy.

Výše popsaná metodika byla prakticky vyzkoušena na experimentálním intranetovém gridu, jehož základem byla zvolena Java RMI technologie. Pro generátor dávkových jobů bylo využito knihoven projektu Apache XML Beans. V tuto chvíli však tento generátor ještě nemá implementovánu kompletní specifikaci JSDL. Chybí například rozšíření o specifikaci JSDL *HPC Profile Aplication Extension*. Toto rozšíření je velmi užitečné, protože umožňuje například spouštění úloh jako součást systémových procesů.

Dále bylo ukázáno, že pro řešení úloh typu Job-Shop, které patří do kategorie $\mathsf{NP}-\mathsf{úplných}$ problémů, je třeba zvolit jisté kompromisy mezi kvalitou výsledku a celkovou délkou zpracování. Exponenciální závislost kvality výsledného plánu a času nutného pro výpočet definuje mezní bod, který určuje hranici, za kterou je další zkvalitnění výsledného rozvrhu plánu neefektivní a neúměrně prodlužuje čas zpracování úloh. Toto je znázorněno na obrázku 10.10 v kapitole 10.4. Navržená aproximace závislosti počtu plánovaných úloh a času běhu plánovacího algoritmu poměrně přesně vystihuje tuto závislost.

Použitou koncepcí modulárního plánovače je umožněn další výzkum v oblasti optimalizace úloh typu Job-Shop, přidáním vlastních komponent bez nutnosti zasahovat přímo do zdrojového kódu plánovače.

Návrh využití grafických procesorů v režimu výkonného paralelního výpočetního zdroje ukázalo velkou perspektivu a zajímavý směr dalšího rozvoje systému. Bohužel nevýhodou tohoto řešení je menší přenositelnost aplikace mezi různými typy počítačových architektur. Zajímavým řešením by představoval návrh vrstvy, která by tuto nevýhodu odstínila. V prostředí Intranet Gridu však toto omezení

není významné, protože v rámci jedné organizace (kde se nasazení Intranet Gridu předpokládá), je možno lépe zajistit jednotné hardwarové a softwarové vybavení. Pro uživatele Gridové služby je výhodné pro tuto oblast využít standardní programy, které tuto službu nabízejí [6] [92].

Nasazení gridové služby v reálném prostředí ukázalo velký potenciál této koncepce pro řadu praktických a výzkumných úkolů, při jejichž řešení je nutná vysoká dostupnost výpočetních zdrojů. Snahou bylo navrhnout koncepci a realizovat systém, který umožní uživateli definovat a spustit úlohu, bez nutnosti detailních znalostí v oblasti gridových služeb a výpočtů. Prakticky bylo také ověřeno, že lze v mnoha případech velmi dobře využít stávající výpočetní potenciál organizace, bez vynaložení velkých investic.

Dosažené výsledky

Úkolem práce je návrh a praktická realizace systému, který umožní distribuci úkolů do výpočetního gridu. Původní realizace vycházela z požadavku optimalizace zátěže ERP systému. Bylo ověřeno, že předložený projekt Intranet Gridu je možné k tomuto účelu využít. Toto řešení umožnilo optimalizovat zatížení hlavního serveru a oddálilo investice do nákupu výkonnějšího hardwaru pro plánování a řízení výroby.

Dále byla tato koncepce využita pro návrh systému, který umožní definovat uživateli obecnou úlohu, a tu následně zpracovat v distribuovaném prostředí gridu. Tato část práce vycházela z myšlenky lepšího využití výkonové rezervy, kterou disponují koncové stanice uživatele. Pro tuto část úlohy byla také zkoumána možnost využití grafických akcelerátorů, jako velmi výkonných paralelních jednotek zapojených v gridové službě.

Bylo ukázáno, že netradiční využití JSDL analyzátoru, jako generátoru spustitelných skriptů, které jsou po rozplánování a spuštění úlohy rozesílány jednotlivým výpočetním zdrojům jako příkazy k práci, které si nesou veškeré informace k získání vstupních dat, spuštění úlohy a odeslání výsledků, usnadnilo celkovou koncepci relativně náročné disciplíny distribuce programových objektů, vstupních a výstupní dat při zadávání výpočetních úkolů.

Přínos práce pro vědu a praxi

Byl ověřen a prakticky realizován systém, který umožní provádět rozsáhlé distribuované výpočty, ale také umožní další rozvoj zkoumané problematiky. Při realizaci tohoto projektu vznikla řada nových zajímavých podnětů, zejména v oblasti synchronizace úloh a návaznosti jejich jednotlivých částí.

Další velmi zajímavou oblastí, která zůstává otevřená je princip plánovacích algoritmů. Navržený plánovač umožňuje velmi snadnou modifikaci. Tím dává dobrý prostor k dalšímu studiu a optimalizaci.

Přínosem byl popis a praktické ověření výpočetního systému, který umožní řešit širokou škálu problémů, které se vyskytují v každodenní praxi, bez nutnosti výrazných investic, ale s využitím prostředků, které jsou běžně k dispozici.

Další směry výzkumu

Autor tezí disertační práce má v plánu rozšířit znalosti v následujících oblastech:

- Praktická implementace v oblasti mechatronických výpočtů
- Rozšíření plánovače o perspektivní optimalizační strategie založených na strategiích inspirovaných přírodou

Další perspektivní témata:

- Ověřit vlastnosti cloudu Apache Hadoop [84], který zahrnuje velmi zajímavou metodiku (Map/Reduce) pro distribuci vzdálených paralelních výpočtů.
- Praktická realizace distribuce výpočtů v prostředí Apache Hadoop.
- Rozšířit perspektivu této oblasti pro praktické využití u některých typů úloh, zejména v oblasti mechatronických výpočtů.
- Výzkum algoritmů pro plánování a rozvrhování úloh v prostředí Gridu [29].

Další snahou je vybudovat experimentální miniGrid, který bude tvořen farmou malých, dostupných a velmi levných miniPC. Jejich výpočetní výkon je naprosto dostačující pro nasazení v roli výpočetních zdrojů. Dokonce se dodávají i s plnohodnotnou grafickou kartou. Výhodou této koncepce je odstínění výpočetních zdrojů od běžného uživatele, a tím výrazné zvýšení spolehlivosti provozu celé soustavy. Výpočetní zdroj instalovaný na běžně používaném kancelářském počítači, který je nasazený v běžném provozu je zatížen poměrně vysokým rizikem výpadku. Navíc cena běžného miniPC je natolik zajímává, že i z ekonomického hlediska původní záměr využití nevyužitého výpočetního výkonu na počítačích v běžném provozu začíná ztrácet smysl. Takto vybudovaný miniGrid může být velmi zajímavou volbou pro další vývoj a výzkum v oblasti gridových služeb.

Literatura

[1] (c) Copyright Policy - open-access License.
URL https://openi.nlm.nih.gov/detailedresult.php?img=PMC2703572_d-65-00659-fig2&req=4

[2] BOINC - Open-source software for volunteer computing.
URL http://boinc.berkeley.edu

[3] GridWay Metascheduler.
URL http://www.gridway.org/doku.php

[4] IBM Platform Computing.
URL http://www-03.ibm.com/systems/services/platformcomputing/lsf.html

[5] IBM Workload Scheduler.
URL http://www-03.ibm.com/software/products/en/workload-scheduler

[6] Parallel Computing In Scilab.
URL https://wiki.scilab.org/Documentation/ParallelComputingInScilab

[7] A U.S. Department of Energy National Laboratory Operated by the University of California: *Berkeley Lab Checkpoint/Restart (BLCR) User's Guide*. 11/07/2014.
URL http://crd.lbl.gov/departments/computer-science/CLaSS/research/BLCR/

[8] Aaftab, M.; Benedict, R. G.; Timothy, G. M.; aj.: *OpenCL Programming Guide*. Addison-Wesley Professional ISBN-13: 978-0-321-74964-2 ISBN-10:0-321-74964-2, July 2011.

[9] Adleman, L. M.: Computing with DNA. *Scientific american*, ročník 279, č. 8, 1998: s. 34–41.

[10] Agarwal, S.; Garg, R.; Gupta, M. S.; aj.: Adaptive Incremental Checkpointing for Massively Parallel Systems. In *Proceedings of the 18th Annual International Conference on Supercomputing*, ICS '04, New York, NY, USA: ACM, 2004, ISBN 1-58113-839-3, s. 277–286, doi:10.1145/1006209.1006248.
URL `http://doi.acm.org/10.1145/1006209.1006248`

[11] Ali, A.; Fred, B.; Michel, D.; aj.: *Job Submission Description Language (JSDL) Specification*. Open Grid Forum (2003-2005 2007-2008) © Open Grid Forum All Rights Reserved., 28 July 2008.
URL `http://forge.gridforum.org/projects/jsdl-wg`

[12] Altameem, T.: Fault Tolerance Techniques in Grid Computing Systems. *T. Altameem / (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 4 (6) , 2013, 858-862*, 2013.

[13] Amdahl, G. M.: Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS spring joint computer conference, 1967*, IBM Sunnyvale California, 1967.

[14] Anjum, A.: *Data Intensive and Network Aware (DIANA) Grid Scheduling*. Disertační práce, The Faculty of Computing, Engineering and Mathematical Sciences, University of the West of England, Bristol, 2007.

[15] Apache Group: *OptaPlanner User Guide Version 6.2.0.Final*.
URL `http://www.optaplanner.org/community/team.html`

[16] Bart, J.; Michael, B.; Kentaro, F.; aj.: *Introduction to Grid Computing*. IBM RedBook, December 2005.

[17] Bharat, B.; Shy-Renn, L.: Independent Checkpointing and Concurrent Rollback for Recovery in Distributed System - An Optimistic Approach. *Purdue University, Purdue e-Pubs Computer Science Technical Reports, Department of Computer Science*, 1987.

[18] Bird, I.; Jones, B.; Kee, K. F.: The Organization and Management of Grid Infrastructures. *Computer*, ročník 42, č. 1, 2009: s. 36–46, ISSN 0018-9162, doi:http://doi.ieeecomputersociety.org/10.1109/MC.2009.28.

[19] BREEBAART, L. C.: *Rule-based Compilation of data-parallel programs.* Technische Universiteit Delft, 2003.

[20] Brucker, P.: *Scheduling Algorithms, Fifth Edition.* © Springer-Verlag Berlin Heidelberg 2001, 2004, 2007, 2007.

[21] Cao, G.; Singhal, M.: On coordinated checkpointing in distributed systems. *Parallel and Distributed Systems, IEEE Transactions on*, ročník 9, č. 12, Dec 1998: s. 1213–1225, ISSN 1045-9219, doi:10.1109/71.737697.

[22] Casanova, H.; Robert, Y.; Vivien, F.; aj.: Combining Process Replication and Checkpointing for Resilience on Exascale Systems. Research Report RR-7951, INRIA, Květen 2012.
URL `https://hal.inria.fr/hal-00697180`

[23] Chtepen, M.; Dhoedt, B.; Turck, F. D.; aj.: Evaluation of Replication and Rescheduling Heuristics for Grid Systems with Varying Resource Availability. *Parallel and Distributed Computing and Systems (PDCS 2006)*, 2006.

[24] Cooperman, G.: Checkpointing using DMTCP, Condor, Matlab and FReD. *High Performance Computing Laboratory College of Computer and Information Science Northeastern University, Boston*, 1998.

[25] Copyright © 1997-2011 OpenMP Architecture Review Board.: *OpenMP Application Program Interface.* Version 3.1 July 2011.

[26] Cytron, R.; Hind, M.; Hsieh, W.: Automatic Generation of DAG Parallelism. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming*

Language Design and Implementation, PLDI '89, New York, NY, USA: ACM, 1989, ISBN 0-89791-306-X, s. 54–68, doi:10.1145/73141.74823.
URL http://doi.acm.org/10.1145/73141.74823

[27] Devlin, K.: *Problémy pro třetí tisíciletí*. Argo, Dokořán, 2002.

[28] Dhruba, B.: HDFS Architecture Guide. *Copyright © 2008 The Apache Software Foundation. All rights reserved.*, 2008.

[29] Fangpeng, D.; Selim, G. A.: *Scheduling Algorithms for Grid Computing, State of the Art and Open Problems*. School of Computing; Queen's University Kingston; Ontario, January 2006.

[30] Figueira, S. M.; Reddi, V. J.: *Euro-Par 2005 Parallel Processing: 11th International Euro-Par Conference, Lisbon, Portugal, August 30 - September 2, 2005. Proceedings*, kapitola Topology-Based Hypercube Structures for Global Communication in Heterogeneous Networks. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, ISBN 978-3-540-31925-2, s. 994–1004, doi: 10.1007/11549468_109.
URL http://dx.doi.org/10.1007/11549468_109

[31] Foster, I.: The anatomy of the grid: enabling scalable virtual organizations. In *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, 2001, s. 6–7, doi:10.1109/CCGRID.2001.923162.

[32] Foster, I.: *The Grid: a new infrastructure for 21st century science*. Argonne National Laboratory, Argonne, Illinois, © 2002 American Institute of Physics., 2002.

[33] Foster, I.: *Designing and Building Parallel Programs*. 2003.
URL http://www-unix.mcs.anl.gov/dbpp/text/book.html

[34] FOSTER, I.; KESSELMAN, C.: *Computational Grids*. Mathematics and Computer Science Division Argonne National Laboratory, Argonne, IL 60439 Information Science Institute University of Southern California, Marina del Rey, CA 90292, 1998.

[35] Foster, I.; Kesselman, C.; Tsudik, G.; aj.: A Security Architecture for Computational Grids. In *Proceedings of the 5th ACM Conference on Computer and Communications Security*, CCS '98, New York, NY, USA: ACM, 1998, ISBN 1-58113-007-4, s. 83–92, doi:10.1145/288090.288111.
URL `http://doi.acm.org/10.1145/288090.288111`

[36] Fritze, M. P.; Patrick, C.; Jennifer, L.; aj.: The Death of Moore's Law. *STEPS: SCIENCE, TECHNOLOGY, ENGINEERING, AND POLICY STUDIES*, 2016.

[37] Garg, R.; Kumar, P.: A Nonblocking Coordinated Checkpointing Algorithm for Mobile Computing Systems. *IJCSI International Journal of Computer Science Issues*, ročník 7, č. 3, May 2010.

[38] Garg, R.; Singh, A. K.: Fault Tolerance in Grid Computing: State of the art and open issues. *International Journal of Computer Science & Engineering Survey (IJCSES) Vol.2, No.1, Feb 2011*, 2011.

[39] Gokhale, A.; Schmidt, D. C.: Principles for Optimizing CORBA Internet Inter-ORB Protocol Performance. *gokhale@cs.wustl.edu and schmidt@cs.wustl.edu Department of Computer Science Washington University St. Louis MO 63130 USA*, January 9th 1998.

[40] Goldberg, D.: What Every Computer Scientist Should Know About Floating-point Arithmetic. *ACM Comput. Surv.*, ročník 23, č. 1, Březen 1991: s. 5–48, ISSN 0360-0300, doi:10.1145/103162.103163.
URL `http://doi.acm.org/10.1145/103162.103163`

[41] Greenlaw, R.; Hoover, H. J.; Ruzzo, W. L.: *Limits to Parallel Computation: P-completeness Theory*. New York, NY, USA: Oxford University Press, Inc., 1995, ISBN 0-19-508591-4.

[42] Grosso, W.: *Java RMI*. O'Reilly ISBN: 1-56592-452-5, First Edition, October 2001.

[43] GROUP, K.: *The OpenCL Specifications*. Khronos OpenCL Working Group, 2009.
URL www.khronos.org

[44] Group, O.: POSIX.1-2008. *The Open Group Base Specifications Issue 7 IEEE Std 1003.1™, 2013 Edition Copyright © 2001-2013 The IEEE and The Open Group*, 2013.
URL http://pubs.opengroup.org/onlinepubs/9699919799/

[45] Göddeke, D.: GPGPU – Basic Math Tutorial Lehrstuhl für Computergrafik. 2005.

[46] Habib, M. A.; Krieger, M. T. (editoři): *Security in Grid Computing*, Johannes Kepler University, A-4040 Linz, Austria, 2015.

[47] Hamscher, V.; Schwiegelshohn, U.; Streit, A.; aj.: Evaluation of Job-Scheduling Strategies for Grid Computing. In *Proceedings of the First IEEE/ACM International Workshop on Grid Computing*, GRID '00, London, UK, UK: Springer-Verlag, 2000, ISBN 3-540-41403-7, s. 191–202.
URL http://dl.acm.org/citation.cfm?id=645440.652831

[48] Herlihy, M.; Kozlov, D.; Rajsbaum, S.: *Distributed Computing Through Combinatorial Topology, 1st Edition*. Boston: Morgan Kaufmann, 20 Jan 2014, doi:http://dx.doi.org/10.1016/B978-0-12-404578-1.00017-6.
URL http://www.sciencedirect.com/science/article/pii/B9780124045781000176

[49] Hernando, B.; Fredy, C.; Daniel, L.; aj.: *Stored Procedures Triggers and User-Defined Functions on DB2 Universal Database for iSeries*. IBM RedBook, October 2006.

[50] Hill, M. D.; Marty, M. R.: Amdahl's Law in the Multicore Era. *Computer*, ročník 41, č. 7, 2008: s. 33–38, ISSN 0018-9162, doi:http://doi.ieeecomputersociety.org/10.1109/MC.2008.209.

[51] Hwang, S.; Kesselman, C.: A Flexible Framework for Fault Tolerance in the Grid. *12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, 2003.

[52] Hwang, S.; Kesselman, C.: A Generic Failure Detection Service for the Grid. *12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, 2003.

[53] jaksa: .
URL `https://jaksa.wordpress.com/2009/05/01/active-and-passive-replication-in-distributed-systems`

[54] Johnson, D. B.: *Distributed System Fault Tolerance Using Message Logging and Checkpointing*. Disertační práce, Department of Computer Science; Rice University; P.O. Box 1892; Houston, Texas 77251-1892, 1989.

[55] Kessenich, J.; Baldwin, D.; Rost, R.: *The OpenGL (R) Shading Language Language Version: 4.30 Document Revision: 6*. Khronos Group Inc., 3-Aug-2012.

[56] Klusáček, D.: *Plánování úloh v paralelním a distribuovaném prostředí*. Disertační práce, MASARYKOVA UNIVERZITA FAKULTA INFORMATIKY, 2006.

[57] Koo, R.; Touegt, S.: Checkpointing and Rollback-Recovery for Distributed Systems. *Department of Computer Science; Cornell University Ithaca, New York 14853*, 1986.

[58] Kshemkalyani, A. D.; Singhal, M.: *Distributed Computing Principles, Algorithms, and Systems*. ISBN-13 978-0-521-87634-6, Cambridge University Press The Edinburgh Building, Cambridge CB2 8RU, UK: © Cambridge University Press 2008, 2008.

[59] Kumar, P.; Setiya, R.; Gahlan, P.: Checkpointing Algorithms for Distributed Systems. *TECHNIA – International Journal of Computing Science and Communication Technologies, VOL. 2, NO. 1, July 2009. (ISSN 0974-3375)*, 2009.

[60] Lampert, L.: Using Time Instead of Timeout for Fault-Tolerant Distributed Systems. *ACM Transactions on Programming Languages and Systems, Vol. 6, No. 2 April 1984, 254-280.*, 1984.

[61] Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, ročník 21, č. 7, 1978.

[62] Lawall, J. L.; Muller, G.: Efficient Incremental Checkpointing of Java Programs. *INRIA Theme 2*, November 1999.

[63] Leslie, L.: Using Time Instead of Timeout for Fault-Tolerant Distributed Systems. *ACM Transactions on Programming Languages and Systems, Vol. 6, No. 2 April 1984, 254-280.*, 1984.

[64] Leu, P.-J.; Bhargava, B.: Concurrent Checkpointing and Recovery in Distributed Systems. *Purdue University Purdue e-Pubs Computer Science Technical Reports*, 1987.

[65] Li, Y.; Pandis, I.; Mueller, R.; aj.: NUMA-aware algorithms: the case of data shuffling. Technická zpráva, University of Wisconsin–Madison, Madison, WI, USA , IBM Almaden Research Center San Jose, CA, USA.

[66] Liang, S.: *The JavaTM Native Interface Programmer's Guide and Specification.* ADDISON-WESLEY ISBN 0-201-32577-2, Jun 1999.

[67] Lukasik, P.; Sysel, M.: An Intranet Grid Computing Tool for Optimizing Server Loads. In *In Advances in Intelligent Systems and Computing. 285. Heidelberg : Springer-Verlag Berlin, 2014, s. 467-474. ISSN 2194-5357. ISBN 978-3-319-06739-1.*, *Advances in Intelligent Systems and Computing*, ročník 285, editace R. Silhavy; R. Senkerik; Z. K. Oplatkova; P. Silhavy; Z. Prokopova, Springer International Publishing, 2014, ISBN 978-3-319-06739-1, s. 467–474, doi:10.1007/978-3-319-06740-7_39.
URL `http://dx.doi.org/10.1007/978-3-319-06740-7_39`

[68] Lukasik, P.; Sysel, M.: A Task Management in the Intranet Grid. In *Software Engineering in Intelligent Systems - Proceedings of the 4th Computer Science On-line Conference 2015 (CSOC2015), Vol 3: Software Engineering*

in Intelligent Systems, 2015, s. 77–85, doi:10.1007/978-3-319-18473-9_8.
URL http://dx.doi.org/10.1007/978-3-319-18473-9_8

[69] Manchanda, N.; Anand, K.: Non-Uniform Memory Access (NUMA). Technická zpráva, New York University.

[70] Mansouri, H.; Badache, N.; Aliouat, M.; aj.: A Non-Blocking Coordinated Checkpointing Algorithm for Message-Passing Systems. In *Proceedings of the International Conference on Intelligent Information Processing, Security and Advanced Communication*, IPAC '15, New York, NY, USA: ACM, 2015, ISBN 978-1-4503-3458-7, s. 34:1–34:5, doi:10.1145/2816839.2816885.
URL http://doi.acm.org/10.1145/2816839.2816885

[71] Mark, S.; Kurt, A.: *The OpenGL Graphics System: A Specification (Version 2.0 - October 22, 2004)*. Copyright c 1992-2004 Silicon Graphics, Inc., 2004.

[72] Marty, H.; Chris, S.; Marvin, T.; aj.: *JSDL HPC Profile Application Extension Version 1.0*. Open Grid Forum (2006-2007) Copyright © Open Grid Forum All Rights Reserved., October 2 2006.

[73] Michael, A. N.; Isaac, L. C.: *Quantum Computing*. Cambridge University Press, Cambridge 2000, 2000.

[74] Michael, F. J.: *Some Computer Organizations and Their Effectivences*. IEEE Trans.Comput. C-21:948, 1972.

[75] Minnebo, W.; Van Houdt, B.: Pull Versus Push Mechanism in Large Distributed Networks: Closed Form Results. In *Proceedings of the 24th International Teletraffic Congress*, ITC '12, International Teletraffic Congress, 2012, ISBN 978-1-4503-1896-9, s. 9:1–9:8.
URL http://dl.acm.org/citation.cfm?id=2414276.2414288

[76] Neider, J.; Davis, T.; Woo, M.: *OpenGL Programming Guide (c) 1993 by Silicon Graphics, Inc.* Addison-Wesley Professional ISBN 0-201-63274-8, January 2006.

[77] nVIDIA: *CUDA C PROGRAMMING GUIDE PG-02829-001 v5.5 July 2013 Design Guide*.

[78] Oracle: *Ken Baclawski Java RMI Tutorial Northeastern University.* URL `http://www.eg.bucknell.edu/~cs379/DistributedSystems/rmi_tut.html`

[79] Oracle: *ORACLE Java Remote Method Invocation - Distributed Computing for Java.* URL `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html`

[80] Oracle: *ORACLE Java RMI.* URL `http://docs.oracle.com/javase/6/docs/technotes/guides/rmi/index.html`

[81] Oracle: *ORACLE, The Java EE 6 Tutorial; Part No: 821–1841–16; Januar 2013.*

[82] Pacheco, P. S.: *An Introduction to Parallel Programming.* University of San Francisco 2011, Elsevier Inc., 2011.

[83] Pacini, F.: *Job Description Language Attributes Specification.* https://edms.cern.ch/document/590869, 2011.

[84] Pavlech, M.: Framework for Development of Distributed Evolutionary Algorithms Based on MapReduce. *Annals of DAAAM & Proceedings. DAAAM International Vienna. 2011.*, 2011.

[85] Petr, L.; Martin, S.: Fault tolerance systém v prostředí výpočetního Gridu. *Trilobit, Univerzita Tomáše Bati ve Zlíně, Fakulta aplikované informatiky, ISSN: 1804-1795*, 2015.

[86] Pharr, M.; Fernando, R.: *GPU Gems 2: Programming Techniques, Tips and Tricks for Real-Time Graphics.* Addison-Wesley Professional ISBN-10:0321335597; ISBN-13: 978-0321335593, March 13 2005.

[87] PostgreSQL: *PostgreSQL 9.2.2 Documentation, Copyright © 1996-2012, The PostgreSQL Global Development Group.*

[88] Pravda, M.: *Možnosti zlepšení přesnosti synchronizace času v paketovýyc sítích*. Disertační práce, ČVUT v Praze Fakulta elektrotechnická Katedra telekomunikačcní techniky, 2015.

[89] Ptak, C.; Smith, C.: *Orlicky's MRP 3rd edition*. McGraw Hill New York ISBN 978-0-07-175563-4., 2011.

[90] Ramík, J.: *Soft Computing: Overview and Recent Developments in Fuzzy Optimization*. Ústav pro výzkum a aplikace fuzzy modelování, Ostravská univerzita, 2001.

[91] Ran, T.; Kaplan, S.; Shapiro, E.: Molecular implementation of simple logic programs. *Nat Nano*, ročník 4, č. 10, Oct 2009: s. 642–648, ISSN 1748-3387, doi:10.1038/nnano.2009.203.
URL `http://dx.doi.org/10.1038/nnano.2009.203`

[92] Reese, J.; ; Zaranek, S.: GPU Programming in MATLAB.
URL `http://www.mathworks.com/company/newsletters/articles/gpu-programming-in-matlab.html`

[93] Ryabko, B.: Using Information Theory to Study Efficiency and Capacity of Computers and Similar Devices. *Information*, , č. ISSN 2078-2489, 2010.

[94] Ryabko, B.: An information-theoretic approach to estimate the capacity of processing units. *Performance Evaluation*, ročník Volume 69, Issue 6, č. ISSN: 0166-5316, June 2012: str. 267–273.

[95] Salah, K.; Manea, A.; Zeadally, S.; aj.: On Linux starvation of CPU-bound processes in the presence of network I/O. *Computers & Electrical Engineering*, ročník 37, č. 6, 2011: s. 1090 – 1105, ISSN 0045-7906, doi: http://dx.doi.org/10.1016/j.compeleceng.2011.07.001.
URL `http://www.sciencedirect.com/science/article/pii/S0045790611001030`

[96] Tony, H.; Patrick, W.: *Nový kvantový vesmír*. Cambridge University Press, Cambridge 2003, 2003.

[97] Vaníček, J.; Papík, M.; Pergl, R.; aj.: *Teoretické základy informatiky*. ISBN 978-80-903962-4-1, Praha, Kernberg Publishing, 2007, 2007.

[98] Welch, V.; Siebenlist, F.; Foster, I.; aj.: Security for Grid services. In *High Performance Distributed Computing, 2003. Proceedings. 12th IEEE International Symposium on*, June 2003, ISSN 1082-8907, s. 48–57, doi: 10.1109/HPDC.2003.1210015.

[99] Whitehead, N.; Fit-Florea, A.: Precision and Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs. 2015.

[100] Zhu, Y.: A survey on grid scheduling systems. *Department of Computer Science, Hong Kong University of science and Technology*, 2003: str. 32.

[101] ZPS Zlin: *CIMAPPS MANUAL - Podrobný popis řídicího systému Kolektiv autorů ZPS, a.s. Zlín 1997.*

[102] Čeněk Šandera: *Hybridní model metaheuristických Algoritmů.* Vysoké učení technické v Brně fakulta strojního inženýrství, Ústav matematiky, 2013., 2013.

[103] Švéda, J.; Machálka, M.: Nové metody a postupy při využívání mechatronických prvků v konstrukci a stavbě obráběcích strojů. *MM Spectrum*, 2011.

[104] Švéda, J.; Machálka, M.: Využívání mechatronických prvků v konstrukci a stavbě obráběcích strojů. *MM Průmyslové spektrum 6 / 2011*, 2011.

Publikační aktivity autora

Články v časopisech

[1] Machálka Martin and Lukašík Petr. Vizualizace výrobních procesů. *MM Průmyslové spektrum 10*, 2010.

[2] Lukašík Petr and Sysel Martin. Analýza vlastností 3d grafických programů a jejich nasazení při modelování mechatronických prvků při vývoji obráběcích strojů. *Trilobit, Univerzita Tomáše Bati ve Zlíně, Fakulta aplikované informatiky, ISSN: 1804-1795*, 2/2011.

[3] Lukašík Petr. Možnosti faktorizace velkých čísel - Shorův algoritmus. *Trilobit, Univerzita Tomáše Bati ve Zlíně, Fakulta aplikované informatiky, ISSN: 1804-1795*, 1/2012.

[4] Lukašík Petr and Sysel Martin. Základní principy kvantového výpočetního systému. *Trilobit, Univerzita Tomáše Bati ve Zlíně, Fakulta aplikované informatiky, ISSN: 1804-1795*,2/2012.

[5] Lukašík Petr. Základní pravidla pro definici bezpečnostní politiky organizace v oblasti informačních technologií. *Trilobit, Univerzita Tomáše Bati ve Zlíně, Fakulta aplikované informatiky, ISSN: 1804-1795*,1/2014.

[6] Lukašík Petr and Sysel Martin. Využití gpu pro paralelní výpočty v prostředí gridu. *Trilobit, Univerzita Tomáše Bati ve Zlíně, Fakulta aplikované informatiky, ISSN: 1804-1795*,2/2013.

[7] Lukašík Petr. Parametry plánovače gridu. *Trilobit, Univerzita Tomáše Bati ve Zlíně, Fakulta aplikované informatiky, ISSN: 1804-1795*, 2014.

[8] Lukašík Petr and Sysel Martin. Fault tolerance systém v prostředí výpočetního gridu. *Trilobit, Univerzita Tomáše Bati ve Zlíně, Fakulta aplikované informatiky, ISSN: 1804-1795*, 2015.

Konference

[1] Petr Lukasik and Martin Sysel. Aplying mechatronic elements in developing and construction work centres. In *Proceedings of the 15th WSEAS International Conference on Computers*, pages 141–144, Stevens Point, Wisconsin, USA, 2011. World Scientific and Engineering Academy and Society (WSEAS).

[2] Petr Lukasik and Martin Sysel. An Intranet Grid computing tool for optimizing server loads. In Radek Silhavy, Roman Senkerik, Zuzana Kominkova Oplatkova, Petr Silhavy, and Zdenka Prokopova, editors, *In Advances in Intelligent Systems and Computing. 285. Heidelberg : Springer-Verlag Berlin, 2014, s. 467-474. ISSN 2194-5357. ISBN 978-3-319-06739-1.*, volume 285 of *Advances in Intelligent Systems and Computing*, pages 467–474. Springer International Publishing, 2014.

[3] Petr Lukasik and Martin Sysel. The distribution of tasks in a Grid environment, Tool to optimize load. In *DAAAM International Scientific Book 2014; Publisher DAAAM International Vienna, Vienna*, pages 401–408. DAAAM International Vienna, 2014.

[4] Petr Lukasik and Martin Sysel. A task management in the Intranet grid. In *In Advances in Intelligent Systems and Computing. Heidelberg : Springer-Verlag Berlin, 2015,*, 2015.

[5] Petr Lukasik and Martin Sysel. An Optimization Scheduler in the Intranet Grid. In *Advances in Intelligent Systems and Computing - ISSN 2194-5357.,*, 2016.

[6] LUKAŠÍK Petr and JAŠEK Roman. Process optimalization of information logistics – subject for increase in information security. In *MEB051024 Information logistics of transport, production and storing systems; ISBN 978-80-7318-989-1*,

pages 90–96. TOmas Bata University in Zlin, Faculty of Applied Informatics, november 10-12 2010.

Publikovaný software

[1] Simulace iteračního výpočtu na grafické kartě

[2] Grid - aplikace pro rozsáhlé výpočty a podporu ERP systému

Odborný životopis autora

Vzdělání

1975 – 1979 Závody Přesného Strojírenství, učební obor strojní zámečník

1979 – 1982 Střední průmyslová škola, obor strojírenství

1982 – 1987 VUT Brno, Fakulta elektrotechnická, obor Technická kybernetika

1987 Státní závěrečná zkouška, obhájena diplomová práce na téma Návrh a realizace klíčových obvodů zdroje střídavého napětí s číslicovým řízením

2010 – dosud Univerzita Tomáše Bati ve Zlíně,Fakulta Aplikované Informatiky doktorandské studium, téma Využití Gridu pro rozsáhlé výpočty.

Profese

1988 – 1989 ZPS Elektrorozvodna revize ochran VN, VVN

1989 – 1991 ZPS Slévárna a.s. programátor analytik

1991 – 2000 ZPS,a.s. programátor analytik, systém CIMAPPS

2000 – 2004 Tajmac ZPS a.s. programátor analytik systém SME-UP

2004 – dosud Tajmac ZPS a.s. zodpovědný za provoz IT infrastruktury

Odborné znalosti

Programování C,C++, Java, Assembler (Z80, Intel86), COBOL, RPGIII, ILERPG

Databáze DB2, ORACLE, PostgreSQL

Webové aplikace JSP, Jakarta Struts, J2EE, Tomcat

Hardwarové platformy iSeries, ES9000, Intel

Operační systémy OS/400, VSE/ESA, Linux

Vývojová prostředí Eclipse, VisualC, SEU

Dokončené projekty

COPICS/CIMAPPS (1993 – 2000) Systém pro řízení výroby
- dodavatel IBM
- Platforma ES9000, DB2, CICS
- cca 800 uživatelů

SME-UP (2001 – dosud) Systém pro řízení výroby
- dodavatel SMEA
- platforma iSeries, DB2
- cca 500 uživatelů

PCC (2004 – 2005) MIS – porovnání skutečných a kalkulovaných nákladů výrobku - platforma iSeries, DB2, Jakarta Tomcat, Jakarta Struts - nadstavba nad SME-UP

Servis Servisní databáze - platforma Linux, Jakata Tomcat, Jakarta Struts

Grid Grid jako podpora ERP systému - platforma iSeries, Java

ProI Interface ProIntralink SME-UP Online propojení CAD a ERP systému na úrovni kusovníkových vazeb.
- platforma iSeries, DB2, ORACLE, CAD ProEngineer

CREO Interface CREO 3D, Windchill SME-UP. Interface CAD a ERP systému na úrovni databází Master-Slave.
- platforma iSeries, DB2, ORACLE, CREO, Windchill

Digitální archív Dokument management systém výkresové dokumentace
- platforma iSeries, DB2, ORACLE

Projekty rozvoje IT infrastruktury

2006 Spolupráce na projektu centralizace tisků

2007 Vybudování nezávisle dislokované záložny dat

2008 Participace na projektu obnovy páteřní sítě 10Gb

2009 Virtualizace serverů – VMware na platformě DELL

Projekty podporované prostředky EU

TM-ND Projekt virtuálního prototypu – část vizualizace pracovního prostoru stroje. Využití Virtual NC Kernelu firmy Siemens, OpenGL

Použité softwarové projekty a knihovny

Apache Ant™ 1.9.4 Nástroj pro sestavování softwarových aplikací. Princip Antu je shodný s unixovým nástrojem Make. Kompilační skripty jsou psány v XML. Výhodou ANTu je platformní nezávislost. `http://ant.apache.org/`

Apache Struts Open-source, MVC framework pro tvorbu webových aplikací technologií JSP. Framework Struts je rozšiřitelný pomocí pluginů AJAX a JSON `https://struts.apache.org/`

Apache Tomcat Apache Tomcat je open source implementace softwarového kontejneru technologie Java Servlet a JSP. `http://tomcat.apache.org/`

Apache Common DBCP™ 2.0.2 Knihovna pro databázovou konektivitu serverových aplikací. Jedná se o wrapper, který umožňuje hospodárně přidělovat databázovou konektivitu pro více uživatelů. `http://commons.apache.org/proper/commons-dbcp/`

Apache XMLBeans XMLBeans usnadňuje manipulaci s XML dokumenty. `http://xmlbeans.apache.org/`

PostgreSQL PostgreSQL je výkonný, open source objektově-relační databázový systém. Je portován na všechny hlavní operační systémy, včetně systémů Linux, UNIX (AIX, BSD, HP-UX, SGI IRIX, Mac OS X, Solaris, Tru64) a Windows. `http://www.postgresql.org`

Apache Logging™ Log4j je výkonný nástroj pro logování stavů aplikací. Je nezbytný pro sledování aplikací běžících v režimu daemon. `http://logging.apache.org/`

GPGPU wrappery Jogl,Jocl,Jcuda jsou knihovny, které usnadňují přístup javovských objektů ke grafickým knihovnám OpenCL, OpenGL a CUDA. `http://www.jcuda.org/` `http://www.jocl.org/` `http://jogamp.org/jogl/www`

CUDA (Compute Unified Device Architecture) je hardwarová a softwarová architektura, která umožňuje na GPU spouštět programy napsané v jazycích C/C++, FORTRAN, Java nebo programy postavené na technologiích OpenCL. `https://developer.nvidia.com/cuda-zone`

OpenGL (Open Graphics Library) je průmyslový standard specifikující multiplatformní rozhraní (API) pro tvorbu aplikací počítačové grafiky. Poskytuje rozhraní pro paralelní výpočty. `https://www.opengl.org/`

OptaPlanner (Constraint Satisfaction Solver) OptaPlanner je optimalizační software pro řešení problémů v oblasti CSP (Constraint Satisfaction Problems). OptaPlanner je uvolněn pod licencí Apache Software. `http://www.optaplanner.org/`

Slovník

2DFFT 2D Fast Fourier Transform algoritmus.

bash je jedním z unixových shellů, který interpretuje příkazový řádek terminálu..

batch process dávková úloha (batch process) je soubor posloupnosti úloh, které které jsou spouštěny v definovaném časovém sledu bez možnosti intervence obsluhy. Obsluha má možnost dávkovou úlohu pouze přerušit (cancel) nebo pozastavit (pause).

Cloud Cloud computing je metoda poskytování komerčních služeb prostřednictvím internetových aplikací.

comit úspěšné ukončení transakce.

CPU Centrální procesorová jednotka (central processing unit) základní součást, která umí vykonávat strojové instrukce, ze kterých je tvořen počítačový program a obsluhovat jeho vstupy a výstupy.

cuBLAS nVIDIA CUDA Basic Linear Algebra Subroutines.

CUDA Compute Unified Device Architecture - hardwarová a softwarová architektura, která umožňuje spouštět programy napsané v jazycích C/C++, FORTRAN na grafických kartách nVIDIA.

cuFFT nVIDIA CUDA Fast Fourier Transform library.

divide zero dělení nulou - výsledkem je kladné nebo záporné nekonečno, které dle IEEE 754 musí generovat chybovou zprávu a ukončení programu.

DOM Document Object Model XML Parser je standard W3C konsorcia, pro analýzu a parsování XML dokumentů.

ERP systém pro řízení a plánování výroby.

Fault tolerance odolnost proti selhání je vlastnost systému (nejen počítačového), která mu umožňuje vykonávat činnost i v případě selhání některé jeho části. Důsledkem je obvykle snížení výkonu a tedy i kvalita služby, ale důležité funkce zůstanou zachovány. Toto je zásadní podmínka služeb a systémů, které zajišťují vysokou dostupnost nebo životně kritické funkce.

FCFS First-come, first-served - přišel jako první, je jako první obsloužen, jedná se o spravedlivou politiku obsluhy požadavků (zákazníků, objednávek) v takovém pořadí v jakém přišly, bez dalších předsudků či preferencí.

FFT Fast Fourier Transform.

FIFO First-in, first-out - požadavky jsou vyřízeny dle pořadí jak přicházejí.

framebuffer je část paměti, která obsahuje rastr dat, která mají být zobrazena na displeji počítače. Framebuffer obsahuje právě jeden kompletní rámec dat, který obsahuje informace o číselných hodnotách barev (odstíny šedi, RGB) pro každý zobrazovaný bod (pixel) na rastru displeje..

GLSL OpenGL Shading Language.

GPGPU General-purpose computing on graphics processing units (zkratka GPGPU) je způsob využití paralelizace na grafické kartě k výpočtu obecných algoritmů.

GPU Graphics processing unit.

Grid Grid computing - je metoda pro zapojení různých výpočetních zdrojů propojených sítí tak, aby mohly být využity pro řešení rozsáhlých úkolů. Rozložením zátěže do jednotlivých uzlů gridu se docílí výrazně rychlejšího zpracování úlohy, než kdyby byla zpracovávána na jediném výpočetním zdroji.

HIL Hardware-in-the-loop.

HLSL High Level Shading Language.

http Hypertext Transfer Protocol.

checkpoint kontrolní bod, checkpointing je technika která zvyšuje odolnost proti chybám. V praxi se jedná o snímkováni běhu aplikace v časových úsecích. V případě poruchy se systém vrací k poslednímu bezchybnému okamžiku - checkpointu.

I/O Input/Output.

J2EE Java Platform, Enterprise Edition.

JSDL Job Submission Definition Language je rozšiřitelná specifikaci XML od Global Grid Forum pro popis dávkových úkolů v prostředí distribuovaných systémů.

JVM Java Virtual Machine.

LLC Low-Lewel Code.

makespan časový rozdíl mezi začátkem a koncem posloupnosti sekvencí dávkové úlohy.

memory leak - únik paměti označuje v informatice situaci, kdy počítačový program neúmyslně alokuje operační paměť a není ji schopen uvolnit poté, co ji již dále ani nepotřebuje ani nevyužívá. Jedná se o častou programátorskou chybu, způsobující v řadě případů nestabilitu aplikace. V syntaxi jazyka C se jedná o funkce malloc() a free(), které jsou plně v režii programátora. Pamětový únik je možno vyvolat i u programovacích jazyků (například Java), které mají vlastní správu alokace a dealokace paměti (garbage collector). A to tak, že programátor například neuzavře datový stream, soubor nebo konektivitu do sítě při vyvolání výjimky.

MRP Material Requirement Planning.

NTP Network Time Protocol.

OpenCL (Open Computing Language) je průmyslový standard pro paralelní programování heterogenních počítačových systémů včetně počítačů vybavených GPU.

OpenGL (Open Graphics Library) průmyslový standard pro tvorbu aplikací počítačové grafiky. Používá se při tvorbě počítačových her, CAD programů, aplikací virtuální reality či vědecko-technické vizualizace. Standard OpenGL je spravován konsorciem ARB (Architecture Review Board). Členy jsou NVIDIA, SGI, Microsoft, AMD.

OpenLDAP Aplikační protokol Adresářových a dotazovacích služeb odvozený od standardu X.500.

permanentní chyba je chyba, která obvykle znamená dlouhodobou poruchu typu výpadek serveru, napájení a podobné. Obvykle má dopad na celou soustavu.

pixel picture element, obrazový prvek.

polynomiální časová složitost algoritmů patří do třídy složitosti P. Tato složitost je považována za třídu problémů, které jsou efektivně řešitelné.

POSIX Portable Operating System Interface.

QoS Quality of Service.

rasterizace je proces při němž dochází k vyjádření 3D vektorové scény ve dvourozměrné rastrové scéně. Jedná se o matematickou metodu, která využívá optickému klamu hry barev a stínů pro vyjádření plastického obrazu na ploše. V praxi je využívána například pro zobrazení prostorového objektu na monitoru počítače..

RMI Java Remote Method and Invocation.

RMI/IOOP Internet Inter-Orb Protocol.

rollback neúspěšné ukončení transakce - návrat do stavu před započetím transakce.

RPG III Report Program Generator.

scalability rozšiřitelnost procesu – schopnost reagovat na změny a zlepšovat sledované parametry.

shader počítačový program sloužící k řízení jednotlivých částí programovatelného grafického řetězce grafické karty.

SQL Structured Query Language.

task jednotka zpracování v rámci dávkové úlohy, obvykle programu, který lze samostatně spustit na výpočetním agentu Gridu. Task se považuje za dále nedělitelnou sekvenci příkazů a programových cyklů, jehož výstupní hodnota je považována za výsledek řešení.

teselace z angl. tesellation, mozaikování, parketování.

texel je základní jednotkou textury (tapety) používané v počítačové grafice.

transakce proces změny z jednoho konzistentního stavu do cílového konzistentního stavu. Transakce se provede buď jako celek, nebo se neprovede vůbec - tzv. atomicita transakce.

transientní chyba je chyba dočasná, například krátkodobý výpadek konunikace. Normální stav je po krátkém čase obnoven.

trigger definuje činnosti, které se mají provést v případě definované události nad databázovou tabul-

kou. Definovanou událostí je jakákoliv manipulace s daty, například insert, update nebo delete.

vertex bod v prostoru.

viewport výřez obrazu (okno) grafické scény..

výpočetní zdroj je základní výpočetní entita (výpočetní zařízení nebo služba), na kterou jsou plánovány úlohy a aplikace, které jsou následně odpovídajícím způsobem zpracovány. Výpočetní zdroje mají své vlastní charakteristiky, CPU paměť, softwarová výbava, operační systém, což je limitně předurčuje k řešení úloh, pro které jsou tyto prostředky dostačující. V rozsáhlých gridech můžou být také limitujícím faktorem přístupová práva k výpočetním zdrojům. Podmínkou je komunikační a softwarové rozhraní (např. POSIX, Java RMI), které umožní připojení do gridové služby.

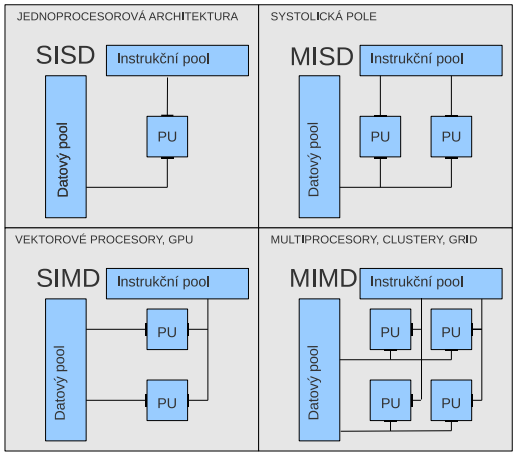
XML Extensible Markup Language.

XSD XML Schema Definition.

Dodatky

Pojednání o Flynnově taxonomii

Flynnova taxonomie je *klasifikace architektur počítačů definovaná již v roce 1966 Michaelem J. Flynnem* [74]. Byly definovány čtyři typy architektur, které vycházejí z principu propojení datových a instrukčních toků a topologie propojení procesorových jednotek.



Obrázek 15.1: Flynnovo rozdělení počítačových architektur [1] [58].

V [19] se vyskytuje také alternativní značení Flynnovy taxonomie, kdy termín **instrukce** je nahrazen slovem **program**. Pak jsou zkratky SISD, SIMD, MISD, MIMD nahrazeny zkratkami SPSD, SPMD, MPSD, MPMD.

Na základě zkušeností s návrhem služby Intranet Gridu byl ověřen zajímavý trend, který jednotlivé architektury počítačů spojuje. V prostředí Grid computingu a Cloud computingu lze s úspěchem propojit všechny typy architektur popsaných Flynnovou taxonomií do fungujícího celku, aniž by uživatel služby musel příliš promýšlet, jaký typ architektury má k dispozici. A v důsledku toho následně promýšlet, jakou metodiku návrhu úlohy zvolí.

To má zajímavý dopad na využívání takovýchto služeb. Namátkou lze uvést například služby **Google Apps**, které jsou určeny pro široké spektrum, od lokálních uživatelů, až po rozsáhlé aplikace v korporátních sférách. Druhou velmi zajímavou oblastí může být uveden **Apache Hadoop** cluster, který je úspěšně nasazován v oblasti big data aplikací a jejich dolování. Další velmi zajímavou vlastností Hadoop je oblast distribuce výpočtu s pomocí programovacího modelu Map/Reduce. Uživatel je těchto případech úplně odstíněn od problému, zda úlohu navrhnout jako paralelní, nebo distribuovanou nebo úlohu pro systémy s distribuovanou pamětí. Jeho jedinou starostí je návrh úlohy dle Map/Reduce programovacího modelu a odeslání této úlohy k vyřešení do (pro uživatele anonymního) prostoru Hadoop clusteru. Tedy, nabízí se otázka, zda Model Flynnovy taxonomie není, alespoň z pohledu uživatele těchto služeb do jisté míry překonán. Tento trend je patrný z obrázku 2.1, kde je znázorněn průnik množin distribuovaných, paralelních a NUMA systémů.

Seznam techniky na níž bylo prováděno měření

	Celsius M460 Fujitsu Siemens
PAMĚŤ	8GiB RAM
CPU	Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz 64 bits
DISPLAY	G84GL [Quadro FX 1700] nVIDIA 64 bits 33MHz, 512Mb
OS	Ubuntu 12.4 LTS
OpenGL verze	OpenGL 2.1.2 nVIDIA 173.14.35
	Celsius W360 Fujitsu Siemens
PAMĚŤ	1.96 GiB RAM
CPU	Intel(R) CPU E 6750 @ 2.66GHz 32 bits
DISPLAY	nVIDIA Quadro FX 570 128Mb
OS	Win XP 32, SP2
OpenGL verze	2.1.1
	Celsius M420 Fujitsu Siemens
PAMĚŤ	1.96 GiB RAM
CPU	Intel(R) CPU E 6750 @ 2.66GHz 32 bits
DISPLAY	nVIDIA Quadro FX 980 128Mb
OS	Ubuntu 12.4. LTS
OpenGL verze	2.1.1
	DELL T1700
Pamět	16GiB RAM
CPU	Intel(R) Xeon(R) CPU E31225 @ 3.10GHz 64 bits
OS	Ubuntu 14.4 LTS
Java	OpenJDK 1.7
	Fujitsu Siemens Celsius M430
Pamět	1.96 GiB RAM
CPU	Intel(R) CPU Intel Pentium 4 @ 3.6GHz 32 bits
DISPLAY	nVIDIA Quadro FX 1400 128Mb
OS	Ubuntu 12.4. LTS
Java	OpenJDK 1.6
	IBM eServer 520
OS	OS/400
CPU	64-bit 1.65GHz POWER5 processor
Počet CPU	2
Oživeno jader	1
Operační pamět	16GiB DDR1
Diskový subsystém	150GiB
Počet současně pracujících uživatelů	300 - 500

Tabulka 15.1: Seznam techniky na níž bylo prováděno měření

Naměřené hodnoty na GPU

Porovnání přesnosti výsledku výpočtu na GPU

Výsledek na GPU	Výsledek na CPU	Rozdíl [%]
0,739085197449	0,739085137844	0,00000806
0,739085197449	0,739085137844	0,00000806
0,739085197449	0,739085137844	0,00000806
0,739085197449	0,739085137844	0,00000806
0,739085197449	0,739085137844	0,00000806
0,739085197449	0,739085137844	0,00000806
0,739085197449	0,739085137844	0,00000806
0,739084959030	0,739085137844	-0,00002419
0,739085197449	0,739085137844	0,00000806
0,739085197449	0,739085137844	0,00000806
0,739085197449	0,739085137844	0,00000806
0,739084959030	0,739085137844	-0,00002419
0,739084959030	0,739085137844	-0,00002419
0,739084959030	0,739085137844	-0,00002419
0,739084959030	0,739085137844	-0,00002419
0,739084959030	0,739085137844	-0,00002419

Tabulka 15.2: Přesnost výpočtu GPU (Porovnáno s výsledkem na CPU)

Měření byla prováděna na dvou pracovních stanicích Siemens Celsius 420 s grafickou kartou nVIDIA Quadro FX 980 128Mb, Siemens Celsius 460 s grafickou kartou nVIDIA Quadro FX 1700 512MB a kancelářském počítači Siemens W370 s grafickou kartou nVIDIA Quadro FX 570 128MB [1]

[1] Měření na počítači Siemens W370 bylo pro úlohu o 100 iteracích zkresleno tím, že systém nedokázal vyhodnotit časové zpoždění mezi startem a stopem úlohy. Jinými slovy, systém při startu i konci úlohy vyhodnotil stejný čas. Bylo to dáno nevyváženou konfigurací počítače - malá výpočetní kapacita CPU. Viz tabulka 15.3

Celsius M460	Vel. FB 2^n	GPU[s]	CPU[s]	CPU/GPU
CPU = 100 iterací	4	0.004987	0.000035	0.00702
	6	0.005315	0.000149	0.02803
	8	0.003725	0.000370	0.09933
	10	0.005064	0.001441	0.28456
	12	0.005454	0.008557	1.56894
	14	0.005648	0.022743	4.02674
	16	0.013153	0.095274	7.24352
	18	0.035416	0.363140	10.25356
	20	0.125273	1.465508	11.69851
CPU = 10000 iterací	4	0.184835	0.002119	0.01146
	6	0.189958	0.008283	0.04360
	8	0.186358	0.033080	0.17751
	10	0.195238	0.132567	0.67900
	12	0.192389	0.529084	2.75007
	14	0.244419	2.117371	8.66287
	16	0.859119	8.461165	9.84865
	18	2.961488	33.849385	11.42986
	20	11.724060	136.872366	11.67449

Celsius M420	Vel. FB 2^n	GPU[s]	CPU[s]	CPU/GPU
CPU = 100 iterací	4	0.236610	0.000166	0.00070
	6	0.212628	0.000594	0.00279
	8	0.211633	0.002303	0.01088
	10	0.217105	0.009066	0.04176
	12	0.224709	0.039944	0.17776
	14	0.259449	0.156229	0.60216
	16	0.405128	0.588821	1.45342
	18	1.006351	2.341569	2.32679
	20	3.178956	9.347348	2.94038
CPU = 10000 iterací	4	2.184835	0.032119	0.01467
	6	2.999719	0.063861	0.02129
	8	3.107196	0.235031	0.07564
	10	3.325038	0.915986	0.27548
	12	4.229010	3.665210	0.86668
	14	7.949982	14.612609	1.83807
	16	21.884199	58.699273	2.68227
	18	82.274248	234.853428	2.85452
	20	299.218033	945.309907	3.15927

Siemens W370	Vel. FB 2^n	GPU[s]	CPU[s]	CPU/GPU
CPU = 100 iterací	4	0.000000	0.000000	
	6	0.000000	0.000000	
	8	0.000000	0.000000	
	10	0.000000	0.015625	
	12	0.000000	0.046875	
	14	0.015625	0.187500	12.00000
	16	0.000000	0.718750	
	18	0.031250	2.890625	92.50000
	20	0.125000	11.531250	92.25000
CPU = 10000 iterací	4	0.109375	0.015625	0.14286
	6	0.109375	0.078125	0.71429
	8	0.093750	0.281250	3.00000
	10	0.093750	1.125000	12.00000
	12	0.093750	4.484375	47.83333
	14	0.203125	17.984375	88.53846
	16	0.750000	72.000000	96.00000
	18	2.812500	287.984375	102.39444
	20	11.046875	1153.781250	104.44413

Tabulka 15.3: Porovnání výkonnosti GPU/CPU

Naměřené hodnoty JSDL analyzátoru

(sweep01) M420

počet úloh	čas[s]				
1	0.555	0.555	0.554	0.557	0.558
2	0.559	0.558	0.564	0.559	0.560
5	0.537	0.533	0.536	0.536	0.537
27	0.606	0.606	0.608	0.604	0.606
256	0.842	0.830	0.810	0.828	0.819
2000	2.067	2.070	2.142	2.056	2.030
13824	9.218	9.088	9.209	9.233	9.177
87808	51.354	51.962	51.625	51.082	51.574

(sweep03) M420

počet úloh	čas[s]				
1	0.553	0.559	0.550	0.548	0.548
10	0.559	0.557	0.600	0.556	0.566
100	0.672	0.661	0.679	0.676	0.706
1000	1.445	1.425	1.404	1.454	1.489
10000	7.412	7.524	7.438	7.501	7.625
100000	67.213	67.114	67.111	67.222	67.049

Tabulka 15.4: Výsledek sweep-loop(1), sweep-loop(3), prac. stanice M420

(sweep01) T1700

počet úloh	čas[s]				
1	0.156	0.155	0.158	0.157	0.156
2	0.160	0.160	0.175	0.158	0.156
5	0.149	0.152	0.150	0.151	0.151
27	0.168	0.170	0.170	0.170	0.170
256	0.233	0.230	0.228	0.228	0.226
2000	0.595	0.552	0.571	0.539	0.482
13824	2.613	2.355	2.591	2.546	2.482
87808	14.675	14.281	14.443	14.655	14.283

(sweep03) T1700

počet úloh	čas[s]				
1	0.155	0.152	0.155	0.156	0.151
10	0.158	0.158	0.158	0.157	0.157
100	0.197	0.193	0.193	0.191	0.190
1000	0.457	0.497	0.483	0.506	0.455
10000	2.534	2.607	2.504	2.590	2.493
100000	21.017	21.299	21.298	20.886	21.097

Tabulka 15.5: Výsledek sweep-loop(1), sweep-loop(3) prac. stanice DELL T1700

Naměřené hodnoty optimalizační plánovač

Měření porovnání vlastností optimalizačních algoritmů bylo prováděno pro dávku 1000 úloh. Pro každý algoritmus bylo provedeno 6 měření. Byl hodnocen celkový čas zpracování pro 1000 úloh a 20 výpočetních zdrojů.

Porovnání optimalizačních algoritmů

Algoritmus	Čas[s] 1000 úloh	Průměr[s]	Sm.Odch.
hillClimbing	250.441		
	251.511		
	255.662		
	254.294		
	254.561		
	256.662	253.855	2.408
lateAcceptance	184.506		
	187.440		
	190.414		
	190.354		
	185.447		
	199.407	189.595	5.390
simulatedAnnealing	198.485		
	190.935		
	197.482		
	205.854		
	205.944		
	202.027	200.121	5.735
stepCountHillClimb	188.431		
	202.697		
	199.567		
	199.810		
	201.320		
	208.953	200.130	6.680
strategicOscillation	187.379		
	198.020		
	199.740		
	205.681		
	197.869		
	198.472	197.860	5.915
tabuSearch	188.562		
	204.655		
	199.854		
	196.165		
	205.842		
	197.422	198.750	6.303

Tabulka 15.6: Výsledek pro sweep loop (1) a sweep loop (3) pracovní stanice M420

Měření pro vyhodnocení délky běhu plánovacího algoritmu bylo prováděno pro algoritmy hill climbing, simulated annealing, stepcounthill climbing, strategic oscillation, tabu search a late acceptance. Pro každý z algoritmů byla prováděna čtyři měření pro čas plánovacího běhu 10, 60 a 300 sekund, a pro 1000, 3000, a 10 000 úloh. Vzhledem k rozsáhlé databázi naměřených hodnot je zde uvedena pouze tabulka algoritmu late acceptance, který vykazoval nejlepší hodnoty.

StepCountHILLClimbing					
Počet úloh	Čas plán.alg[s] T	Čas zpracování.[s] T	Průměr	T1+T2	Stdev
1000	10	202.697 199.567 199.810 201.320	200.849	200.849	1.456
1000	60	208.953 198.572 195.228 201.011	200.941	200.941	5.844
1000	300	196.293 195.773 201.529 194.396	196.998	196.998	3.125
3000	10	418.812 405.283 421.216 429.378	418.672	418.672	10.006
3000	60	433.914 438.381 438.591 438.220	437.277	437.277	2.247
3000	300	447.854 438.831 465.018 427.355	444.765	444.765	15.896
10000	10	1205.503 1193.696 1200.166 1156.643	1189.002	1189.002	22.106
10000	60	1204.021 1169.482 1193.917 1154.361	1180.445	1180.445	22.641
10000	300	1154.090 1181.719 1171.164 1166.117	1168.273	1168.273	11.474

Tabulka 15.7: Výsledek něření algoritmu StepCountHILLClimbing