UNIVERZITA MATEJA BELA V BANSKEJ BYSTRICI
FAKULTA PRÍRODNÝCH VIED

Algoritmy vyvažovania záťaže v gridovom prostredí

Plánovanie úloh

Diplomová práca

7de16278-58d3-4447-8bb8-1559b0d81524

Študijný program: Aplikovaná informatika
Študijný odbor: 9.2.9 Aplikovaná informatika
Pracovisko: Katedra Informatiky
Vedúci bakalárskej práce: Ing. Jarmila Škrinárová, PhD.

Banská Bystrica 2013 Bc. Michal Krnáč

Čestné vyhlásenie

Čestne prehlasujem, že som túto diplomovú prácu vypracoval samostatne vedený radami a usmernením vedúcej práce Ing. Jarmily Škrinárovej, PhD. a s použitím uvedenej literatúry.

Dátum: 22.4.2013 ….....

ABSTRAKT

KRNÁČ, Michal: *Algoritmy vyvažovania záťaže v gridovom prostredí.* [Diplomová práca] Univerzita Mateja Bela v Banskej Bystrici, Fakulta Prírodných Vied, Katedra Informatiky. Vedúci práce: Ing. Jarmila Škrinárová, PhD.
Stupeň odbornej kvalifikácie: Magister (Mgr.), Banská Bystrica, 2013, 84s

V predkladanej diplomovej práci predstavujeme princípy a technológie súčasných gridových riešení. Analyzujeme súčasné vysokovýkonné a paralelné výpočtové prostriedky ako základ gridových systémov. Poukazujeme na obmedzené možnosti technologického vývoja procesorov a ponúkame paralelné systémy ako riešenia týchto limitov. V teoretickej časti prinášame stručný prehľad histórie vývoja gridov, rôzne gridové modely a softvérové prostriedky na ich správu. Predstavujeme pojem plánovanie a analyzujeme jeho základné fázy, ktoré sú potrebné pre efektívne využívanie prostriedkov systému a zabezpečenie vyváženej záťaže. V praktickej časti navrhujeme plánovací algoritmus, ktorého účelom je minimalizovať čas potrebný na vykonanie skupiny úloh v gridovom prostredí a tým zabezpečiť účinné rozdelenie záťaže v systéme. Navrhovaný algoritmus využíva princípy stochastickej optimalizácie založenej na princípe sociálnych javov správania sa kŕdľa vtákov alebo roja častíc – PSO. Efektivitu algoritmu analyzujeme pomocou simulácie modelového gridu na základe údajov o prostriedkoch v reálne existujúcom gride. Ďalej porovnávame účinnosť navrhovaného algoritmu s ďalšími dvomi algoritmami o ktorých je známe, že sa v súčasnej dobe používajú na plánovanie úloh v gridovom prostredí, konkrétne algoritmus Round Robin a horolezecký algoritmus. Na základe výsledkov simulácie implementujeme princípy navrhovaného algoritmu v plánovači lokálneho správcu zdrojov, ktorý je používaný na klastri Fakulty Prírodných Vied Univerzity Mateja Bela. Pre potreby lokálneho plánovania navrhujeme hybridnú implementáciu PSO a horolezeckého algoritmu. Prezentujeme výsledky dosiahnuté pri použití algoritmu v obidvoch úrovniach gridového plánovania a na základe týchto výsledkov poukazujeme na možnosť využitia heuristických algoritmov pre riešenie rodiny NP-úplných problémov.

Kľúčové slová: grid, vyvažovanie záťaže, plánovanie, optimalizácia, správca zdrojov

ABSTRACT

KRNÁČ, Michal: *Algorithms of load balancing in grid environment.* [Diploma thesis] University of Matej Bel Banská Bystrica, Faculty of Natural Sciences, Department of Informatics. Supervisor: Ing. Jarmila Škrinárová, PhD.
Qualification degree: Master (Mgr.), Banská Bystrica, 2013, 84s

In our diploma thesis we present the principles and technologies of current computing grid solutions. We analyse today's high-performance and parallel computing resources as building blocks of grid systems. We show that the potential of technological development of processing units is quite limited and offer parallel systems as an overcome to this limits. In theoretical part of our research we offer short briefing into the history of computational grids development, several grid models and grid software. We introduce the term scheduling and analyse basic phases that are necessary for effective resource utilization of a parallel system and for load balancing. In practical research we propose a scheduling algorithm in order to minimize the time necessary to execute the group of jobs in grid and by that achieve balanced load in the system. Proposed algorithm uses the stochastic heuristic optimization based on the principles of social behaviour of a birds flock or particle swarm called particle swarm optimization – PSO. We analyse the algorithm efficiency by simulation of a model grid based on the information about the resources of a real grid system. We then compare the efficiency of suggested algorithm with two other algorithms, which are known to be effectively used in scheduling jobs in grid systems, particularly Round Robin and Hill Climbing algorithms. Based on the simulation results we implement the principles of proposed algorithm in local resource manager scheduler used in the cluster of Faculty of Natural Sciences of the University of Matej Bel in Banská Bystrica. For the local scheduler we suggest a hybrid implementation of PSO and HC. We present the findings reached in both grid scheduling levels and based on the results we show the potential of heuristic algorithms in solving the family of NP-complex problems.

Keywords: grid, load balancing, scheduling, optimization, resource utilization

PREDHOVOR

I do not fear computers. I fear the lack of them.
Isaac Asimov

Paralelné počítanie získava v súčasnej dobe čoraz väčšiu popularitu. Dnes už má takmer každý používateľ osobného počítača k dispozícii procesor s viacerými jadrami. To však neznamená, že ich dokáže aj efektívne využiť. Niektorí ľudia to ale vedia. A neuspokoja sa iba s dvoma jadrami procesora. Dokážu riešiť zložité fyzikálne, chemické či astronomické výpočty no potrebujú na to vysoký výkon. Niekto, na druhej strane, môže vlastniť výkonný počítač no pre svoje potreby využíva len malú časť jeho výkonu. Múdri ľudia vymysleli spôsob ako využiť výpočtový potenciál, ktorý poskytujú mnohé výskumné inštitúcie a sprístupniť ho používateľom, ktorí ho potrebujú. Takéto a podobné úvahy viedli k vytvoreniu gridových systémov.

Jednou z mnohých súčastí, ktoré sa podieľajú na správe gridu a zabezpečení vyváženej záťaže je aj plánovač a plánovací algoritmus. Plánovanie v paralelných systémoch je zaujímavý proces, ktorý správne navrhnutý dokáže ušetriť používateľom čas a poskytovateľom peniaze. Gridový plánovač musí spĺňať mnohé podmienky aby mohol efektívne fungovať. Mojím záujmom, ktorého výsledkom je táto práca, bolo preskúmať tieto podmienky a pokúsiť sa navrhnúť plánovací algoritmus, ktorý by sa vyrovnal existujúcim riešeniam. K práci na tejto téme ma motivovala pani Ing. Jarmila Škrinárová, PhD, ktorej by som rád poďakoval za spoluprácu, cenné rady a pripomienky pri mojom bádaní. Ďakujem aj správcom klastrov FPV UMB pánovi Mgr. Michalovi Vagačovi, PhD, a pánovi Mgr. Jozefovi Siláčimu za ochotu a spoluprácu pri testovaní plánovača v reálnom prostredí.

Obsah

Zoznam použitých obrázkov

Zoznam použitých tabuliek

Zoznam pseudo-kódov použitých algoritmov

Zoznam použitých skratiek

Skratka	Význam
AFS	Andrew File System
ATM	Asynchronous Transfer Mode
CPU	Central Processing Unit
CRB	Computational Resource Broker
DIMS	Dynamic Infrastructure Management System
EGEE	Enabling grids for E-science
EGI-InSPIRE	EGI-Integrated Sustainable Pan-European Infrastructure for Research
FAFNER	Factoring via Network-Enabled Recursion
FCSF	First Comes First Served
FIFO	First In First Out
flop/s	Floatin point Operations per Second
GRAM	Globus Toolkit Resource Allocation Manager
HC	Hill Climbing
I-POP	I-WAY Point Of Presence
I-WAY	Information Wide Area Year
JDL	Job Description Language
JSS	Job Submission Service
LDAP	Leightweight Directory Access Protocol
LJF	Longest Job First
LSF	Load Sharing Facility
MDS	Monitoring and Discovery Service
MI	Millions of Instructions
MIMD	Multiple instruction Multiple Data
MIPS	Microprocessor without Interlocked Pipeline Stages
MISD	Multiple instruction Single Data
MPI	Message Passing Interface
NGI	National Grid Initiative
OGF	Open Grid Forum
OS	Operating System
PBS	Portable Batch System
PSO	Particle Swarm Optimization
RAM	Random Access Memory
RR	Round Robin
RSA	Rivest Shamir Adleman algoritmh
SC95	SuperComputing 1995
SGE	Sun Grid Engine
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SJF	Shortest Job First
SMP	Symmetric multi-processing
SQL	Structured Query Language
SSL	Secure Sockets Layer
TCP/IP	Transmission Control Protocol and Internet Protocol
VO	Virtual organization
WRMS	Workload and Resource Management System
XML	Extended Markup Language

ÚVOD

Osobné počítače nás v dnešnej dobe sprevádzajú na každom kroku. Podľa súčasných trendov sa výrobcovia snažia zaujať čo najviac zákazníkov a začínajú sa orientovať na bezdrôtové mobilné zariadenia, tablety či herné konzoly, ktoré ľudia radi využívajú na rýchle a jednoduché aplikácie uľahčujúce každodenný život. Naopak vo vedeckej sfére sa pozornosť upriamuje na efektívne využitie výkonu súčasných počítačov na riešenie výpočtovo náročných fyzikálnych, matematických, chemických, či technologických úloh a problémov. Na tieto účely vzniklo niekoľko architektúr, ktoré umožňujú aplikáciám zdieľať výpočtový výkon viacerých počítačov a tak postavili základ pre paralelné počítanie. V prvej časti práce predstavujeme najbežnejšie paralelné architektúry pre vysokovýkonné počítanie a ich zjednocovanie do vyšších celkov – gridov. Uvádzame stručnú históriu vývoja rôznych gridových infraštruktúr a predstavujeme základné softvérové prostriedky, ktoré boli vyvinuté pre ich správu.

Odhaduje sa, že využívať potenciál paralelných systémov dokáže len malé percento programátorov. To však nie je jediným problémom, ktorý sa vyskytuje v súvislosti s ich používaním. Správa výpočtových prostriedkov a ich efektívne využívanie na vykonávanie paralelných a sekvenčných programov je náročný proces, ktorý vyžaduje starostlivú analýzu a výber správnych rozhodnutí. Priradenie a rovnomerné rozmiestnenie úloh mnohých používateľov medzi výpočtové prostriedky, ktoré grid poskytuje sa nazýva plánovanie. Hoci existuje mnoho prístupov ako zabezpečiť efektívne plánovanie všetky majú isté spoločné charakteristiky. V druhej časti práce predstavujeme základné fázy plánovania, popisujeme teoretické predpoklady pre efektívne plánovanie a naznačujeme rôzne prístupy ku plánovaniu úloh v distribuovaných systémoch.

Tretia časť práce je venovaná návrhu plánovacieho algoritmu, ktorý je založený na princípe stochastickej optimalizácie pomocou častíc roja. Ide o mladú a relatívne málo známu techniku, ktorú môžeme zaradiť medzi evolučné algoritmy. Jej princípy sa pokúsime aplikovať na problém plánovania úloh a prostriedkov v gridovom prostredí. Zároveň predstavujeme plánovaciu technológiu založenú na princípe horolezeckého algoritmu.

Štvrtá časť práce sa zaoberá skombinovaním prístupu obidvoch plánovacích algoritmov a vytvorenie hybridného modelu plánovača využívajúci algoritmus PSO pre usporiadanie úloh a algoritmus HC pre priradenie podúloh výpočtovým zdrojom.

Modelovaním, simuláciou a implementáciou navrhnutých algoritmov sa zaoberá piata časť práce. Naším cieľom tu je porovnať navrhnutý algoritmus PSO s inými algoritmami, ktoré sa bežne používajú na rovnaké účely. Najjednoduchšou alternatívou sú algoritmy Round Robin a FIFO. Vhodným spôsobom ako zistiť efektivitu navrhnutého algoritmu je použiť ho v modelovom virtuálnom prostredí, ktoré dokáže simulovať procesy prebiehajúce v gride. Zároveň popisujeme implementáciu hybridného plánovacieho algoritmu PSO+HC v prostredí lokálneho správcu zdrojov Torque, analyzujeme časovú a pamäťovú náročnosť navrhnutého algoritmu a popisujeme rozdiely medzi plánovaním paralelných úloh vo vysokovýkonnom počítačovom klastri a plánovaním úloh v gride.

Šiesta kapitola je venovaná testovacím údajom, ktoré sme použili pre overenie funkčnosti a simuláciu navrhnutých algoritmov. Počas simulácie využívame náhodne vygenerované úlohy, ktorých vykonávanie simulujeme v modelovom gride. Model je vytvorený podľa reálnych údajov z existujúceho gridového projektu.

V poslednej časti práce sa venujeme vyhodnocovaniu a interpretácii výsledkov dosiahnutých pomocou gridových simulácií a klastrových testovaní. Porovnávame výkon navrhnutého algoritmu s jeho alternatívami a analyzujeme dosiahnuté výsledky. Poukazujeme na fungovanie plánovača na virtuálnom a reálnom klastri a porovnávame jeho efektivitu s pôvodným plánovačom klastra na základe výsledkov, ktoré sme získali počas testovania plánovača.

Veríme, že návrhy a podnety ktoré prináša táto práca budú zaujímavé pre všetkých programátorov, ktorí sa zaoberajú plánovaním, vyvažovaním záťaže, alebo evolučnými algoritmami a ich použitím v rôznych oblastiach.

1 Vysokovýkonné a paralelné počítanie v súčasnosti

Vývoj v oblasti architektúr počítačov nezadržateľne napreduje už od vzniku prvého počítača. Podľa Mooreovho zákona sa počet tranzistorov, ktoré je technologicky možné umiestniť na čip procesora každých 18 mesiacov približne zdvojnásobí [1]. Takýto vývoj sa nazýva exponenciálny a jeho charakteristikou je prudký nárast počtu súčiastok na konštantne veľkej ploche silikónového čipu procesora. Je zrejmé, že takéto súčasné technologické postupy musia mať svoje limity spôsobené najmä minimálnou veľkosťou polovodiča, či obmedzenou rýchlosťou elektrónov, rýchlosťou svetla alebo termodynamickými zákonmi. Ponúkaným riešením je zoskupiť výpočtové jednotky za účelom získania vysokého výkonu pri čo najmenšej strate efektivity. Ďalšou skutočnosťou, ktorá podporuje tieto snahy je fakt, že existujú problémy, ktoré musia dnešné počítače riešiť a ďaleko presahujú ich hardvérové možnosti. To znamená, že ich riešenie trvá neúmerne dlho, alebo sa na jednom procesore jednoducho nedajú realizovať vôbec. Z uvedených dôvodov vznikla potreba paralelných počítačových architektúr.

História paralelného počítania siaha ešte do roku 100 p.n.l., Keď sa na realizovanie náročnejších matematických výpočtov používala tabuľka s tromi nezávislými stĺpcami na vykonávanie výpočtov. Takýto spôsob počítania mohol viesť k zvýšenej kontrole správnosti alebo k schopnosti vykonať niektoré kroky výpočtu naraz. Táto snaha sa začala prejavovať aj vo vývoji digitálnych počítačov už od ich vzniku a mnohí sa správne domnievali, že už koncom minulého storočia sa budú vyrábať výlučne paralelné počítače [2]. Tieto predpoklady sa potvrdili a v súčasnej dobe sú bežne dostupné počítače, ktorých základ tvorí procesor s viacerými jadrami schopnými vykonávať niekoľko úloh naraz v rovnakom čase. Napriek tomu vedci idú ešte ďalej a v snahe získať ešte väčší výpočtový výkon vznikol nápad spojiť viaceré výpočtové jednotky pomocou rýchlych zberníc, alebo sieťových komponentov. Takto vznikli paralelné počítače, ktoré môžeme voľne charakterizovať ako tzv. multiprocesory a multipočítače. Existujú mnohé spôsoby klasifikácie takýchto paralelných počítačov podľa rôznych kritérií:

- v závislosti od spôsobu prepojenia procesorov (tesne prepojené, voľne prepojené),
- podľa prístupu k pamäti (zdieľaná pamäť, distribuovaná pamäť, hybridný model)
- podľa spôsobu spracovania údajov (Flynnova taxonómia – SISD, SIMD, MISD, MIMD)

- podľa štruktúry prepojenia procesorov (hviezdicová, kruhová, mriežka, hyper-kocka)
- podľa úrovne zdieľania pamäte (úroveň hardvéru, úroveň OS, používateľská úroveň)

1.1 Superpočítač

Osadenie viacerých procesorov na jednu matičnú dosku, alebo ich prepojenie rýchlou zbernicou v spoločnom stojane a zabezpečenie prístupu k spoločnej pamäti sú typickými prvkami tzv. *superpočítača*. Jednoznačným efektom je zvýšenie výpočtového výkonu. Podľa súčasného rebríčka TOP 500 najvýkonnejších počítačov na svete sa výkon superpočítačov pohybuje rádovo 2.10^{16} flop/s čo pre názornosť znamená, že takýto superpočítač je schopný vykonať 27 000 miliard operácií s racionálnymi číslami za jedinú sekundu. V porovnaní s bežne dostupnými kancelárskymi počítačmi s výkonom rádovo 10^{10} flop/s, čo zodpovedá 10 miliónom aritmetických operácií s desatinnou čiarkou za sekundu je teda súčasne najrýchlejší superpočítač na svete približne miliónkrát rýchlejší [3]. Architektúra súčasne najvýkonnejšieho superpočítača (obr. č. 1) vďaka viac ako 560 000 jadrám umožňuje vykonávať zložité medicínske, chemické, biologické alebo astronomické výpočty, ktoré by boli na bežnom počítači nerealizovateľné.

Obrázok č. 1: TITAN, súčasný najvýkonnejší superpočítač, nachádzajúci sa v americkom meste Oak Ridge, Tennessee [36].

1.2 Počítačový klaster

Podobným spôsobom efektívneho využitia výkonu viacerých výpočtových jednotiek je prepojiť ich sieťovými prvkami. Vo všeobecnosti sa pre skupinu počítačov

navzájom prepojených rýchlou lokálnou sieťou zaužívalo označenie *počítačový klaster*. Klaster (obr. č. 2) zoskupuje niekoľko výpočtových jednotiek – *uzlov*. Každý uzol má jeden, alebo viacero procesorov s jedným, alebo viacerými jadrami, má dostupné vlastné periférne zariadenia a má prístup ku svojej vlastnej lokálnej pamäti. Na každom uzle beží operačný systém, ktorý je väčšinou jednotný pre celý klaster. Typickým prvkom je jeden hlavný počítač, ktorý zabezpečuje správu zdrojov a vhodné rozdeľovanie úloh na jednotlivé pracovné uzly. Najvýkonnejší klaster podľa TOP 500 ponúka približne 15000 výpočtových uzlov s celkovým výkonom 3.10^{15} flop/s a je efektívne využívaný na riešenie mnohých problémov v oblasti výskumu atmosféry, nanotechnológií, seizmológie, techniky atď. [4]

Obrázok č. 2: Počítačový klaster kedysi a dnes [37].[38]

1.3 Distribuované systémy

Výrazný pokrok v oblasti sieťových technológií v 80-tych rokoch minulého storočia umožnil paradigmy paralelných počítačov posunúť na ďalšiu úroveň. Jednotlivé stroje, ktoré tvoria súčasť paralelného počítača už nemuseli byť umiestnené v jednej miestnosti, ale bolo možné ich rôzne fyzické rozmiestnenie. Zároveň sa dosiahla možnosť prepojiť už existujúce počítače, superpočítače, alebo klastre a ich zjednotenie do jedného systému. Vznikol tak základ pre tzv. *distribuované systémy*.

Je zrejmé, že spojenie viacerých počítačov si vyžaduje správu dostupných prostriedkov s odlišným prístupom ako klasické operačné systémy. Je nevyhnutné, aby na

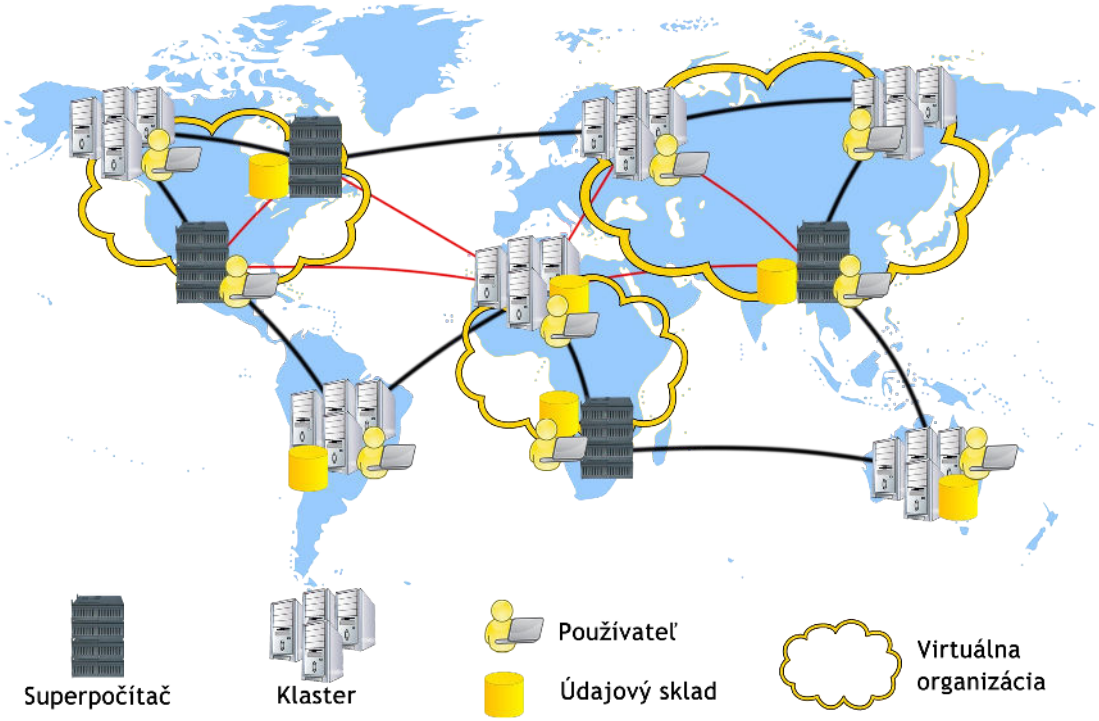
každom uzle boli zabezpečené základné úlohy operačného systému ako sú: správa procesov, správa pamäte, správa periférnych zariadení a správa súborov. Distribuovaný operačný systém zabezpečuje všetky tieto požiadavky a zároveň je rozšírený o softvérové prostriedky, ktoré umožňujú správu viacerých uzlov a ich dostupných zdrojov. Cieľom je integrácia rôznych zdrojov a funkcionality do efektívneho stabilného systému [5]. Užívateľovi sa tak javí ako klasický monolitický operačný systém jedného počítača.

1.4 Gridová infraštruktúra

Prepojením počítačov v rôznych geografických oblastiach s rôznymi dostupnými výpočtovými prostriedkami a zdrojmi vzniká tzv. *grid* (obr. č. 3). Gridová infraštruktúra nám ponúka možnosť dynamicky zoskupovať zdroje za účelom podporiť vykonávanie náročných výpočtových distribuovaných aplikácií [6]. Na rozdiel od jednoduchého klasického distribuovaného systému môže združovať prostriedky z rôznych administratívnych domén s rôznymi operačnými systémami a inými vlastnosťami.

Okrem celkového zvyšovania výkonu existuje viacero výhod spomenutého systému:

- Je možné, že v istom momente je jeden počítačový systém nadmieru vyťažený, zatiaľčo stroje na inom mieste môžu byť nevyužité. Prepojením rôznych počítačov, alebo počítačových systémov je možné dosiahnuť vyváženú záťaž.
- Špecifický softvér môže byť dostupný pre širšie spektrum používateľov.
- Počet uzlov je možné meniť bez nutnosti náročného zásahu do riadiaceho systému.
- Výpadok niektorého uzla nemá vplyv na globálny chod systému.



Obrázok č. 3: Grid - prepojenie výpočtových prostriedkov na rôznych kontinentoch.

1.5 História vývoja gridovej infraštruktúry

Gridové systémy sú staré len niekoľko desaťročí a práve preto majú veľký potenciál a priestor pre rozvíjanie. V histórii vývoja infraštruktúr gridových systémov rozoznávame 3 hlavné generácie, ktoré určili súčasné štandardy.

1.5.1 I generácia – FAFNER, I-WAY

V rokoch 1976 – 1977 páni Rivest, Shamri a Adelman publikovali vtedy jedinečný šifrovací algoritmus založený na princípe verejného kľúča [7]. Algoritmus nazvali podľa iniciál svojich mien – RSA. Je založený na myšlienke, že je veľmi náročné rozdeliť veľké čísla na rozklad ich súčiniteľov (angl. factoring). Hlavne ak ide o čísla, ktoré majú rádovo niekoľko 100 číslic. Dodnes sa tento algoritmus úspešne používa pri komunikácii cez zabezpečený protokol SSL. Aby dotiahli svoj projekt k dokonalosti firma RSA Data Security Inc. spustila v roku 1991 súťaž o nájdenie rozkladu čísel, ktoré považovala za veľmi ťažko rozložiteľné (The RSA Factoring Challenge). Pretože klasické riešenie je výpočtovo veľmi náročné rýchlo vznikli algoritmy pre paralelný rozklad čísla na jeho dva súčinitele. Použité algoritmy sa ukázali ako triviálne paralelné a po počiatočnej inicializácii

nevyžadovali takmer žiadnu komunikáciu. Týmto spôsobom mohol spustiť program s príslušnými vstupmi na svojom osobnom počítači ktokoľvek a prispieť tak k nájdeniu riešenia. Spočiatku medzi ľuďmi zapojenými do projektu fungovala mailová komunikácia, ktorú v roku 1995 nahradil projekt konzorcia Bellcore Labs. univerzity Syracuse v New Yorku, ktorý sa volal FAFNER - Factoring via Network-Enabled Recursion. Pre výpočet rozkladu čísla RSA130 sa začala používať technika sita číselných polí (Number Field Sieve). K spoločnému serveru mohli pristupovať všetci zapojení členovia projektu pomocou webového rozhrania, ktoré umožňovalo distribuovať úlohy, zbierať výsledky, poskytovalo podpornú dokumentáciu a podobné služby. Do projektu sa mohol zapojiť každý anonymne. Zo spomínaného servera prevzal používateľ špeciálny softvér, ktorý po skompilovaní a nainštalovaní vhodne spotreboval nevyužitý čas hostiteľského procesora na potrebné výpočty. Zo serverom komunikoval ako bežný webový prehliadač pomocou GET a POST požiadaviek. Tento projekt je možné považovať za jednu z prvých gridových infraštruktúr.

Druhým podobným projektom, ktorého zámerom bolo spojiť vysokovýkonné výpočtové centrá v USA bol demonštračný projekt I-WAY, ktorý bol, podobne ako FAFNER, súčasťou konferencie o superpočítačoch SC95 v San Diegu. Počas týždňovej konferencie výskumníci a priekopníci v oblasti superpočítačov predstavili testovací projekt 17 výskumných zariadení prepojených pomocou vysoko-rýchlostnej chrbtovej siete, ktorá podporovala vtedy platný štandardný protokol ATM, a zároveň aj protokol TCP/IP. Za účelom správy gridovej infraštruktúry bol vyvinutý špeciálny softvér I-Soft, ktorého úlohou bolo poskytnúť prístup používateľov, zabezpečiť ochranu a bezpečnosť systému, správu zdrojov a iné aktivity. Softvér I-Soft bol nainštalovaný na každom pracovisku na jedenej pracovnej stanici označovanej ako bod prístupu – I-POP (z angl. point-of-presence), ktorá predstavovala vstupnú bránu k projektu I-WAY. Špeciálne pre tento projekt bol vyvinutý plánovač zdrojov – CRB (Computational Resource Broker), ktorý implementoval protokoly pre komunikáciu medzi používateľom a dostupnými hardvérovými prostriedkami. Jeho súčasťou boli lokálne plánovače, zabezpečujúce rozdelenie záťaže medzi jednotlivé uzly systému a jeden centrálny plánovač, ktorý udržoval rad úloh a zároveň tabuľku aktívnych zdrojov v systéme. Pre zdieľanie údajov bol použitý špeciálny súborový systém – AFS (Andrew File System), umožňujúci zdieľanie diskového priestoru nielen na lokálnej sieti, ale aj cez internet. Súčasťou projektu I-WAY

bolo vyše 60 aplikácií vyvinutých a nasadených na I-WAY za účelom testovania. Práve tieto aplikácie sa stali základom dnešných gridových softvérových prostriedkov. Príkladom sú projekty Globus, Legion, Condor, AppLeS, Mars, Prophet a mnohé iné.

1.5.2 II generácia - Otvorené Gridové Fórum

Neskôr v 90-tych rokoch vedci pracujúci na uvedených gridových projektoch vytvorili tzv. Gridové Fórum, z ktorého sa následne stalo Globálne Gridové Fórum (GGF). Vedci a výskumníci z rôznych inštitúcií z krajín celého sveta začali pracovať na spoločnom štandarde, ktorý by definoval základné služby v hlavných oblastiach gridovej infraštruktúry:

- systémový manažment a automatizácia,
- vyvažovanie záťaže a výkonnosť,
- bezpečnosť,
- dostupnosť a správa služieb,
- správa logických a fyzických zdrojov,
- klastrovacie služby,
- správa sieťových prepojení.

V roku 2006 sa Globálne Gridové Fórum zlúčilo s organizáciou Enterprise Grid Alliance a vytvorili spoločnú organizáciu OGF – Open Grid Forum. Cieľom tejto organizácie je vyvíjať štandardy pre prácu v gridovom prostredí. Výrazný technologický pokrok hlavne v oblasti počítačových sietí umožnil napĺňať víziu tvorcov gridovej infraštruktúry. Základné problémy, ktoré bolo treba riešiť pri návrhu prostriedkov toho, čo môžeme označiť ako druhú generáciu vývoja gridov boli:

- Heterogénnosť – grid v sebe zahŕňa množstvo zdrojov, ktoré sú vo svojej podstate heterogénne a môžu byť súčasťou niekoľkých administratívnych domén dokonca vo svetovom meradle [6].
- Rozšíriteľnosť – grid sa môže jednoducho zväčšiť z niekoľkých počítačov na niekoľko miliónov. V súvislosti s týmto sa vynára problém degradácie výkonu. Podobne aplikácie navrhnuté pre veľké množstvo zdrojov, s rôznym geografickým umiestnením musia počítať s latenciou pri komunikácii a správne využívať výhody lokálne pridelených zdrojov. Rozširovanie prináša aj problémy týkajúce sa

štrukturálnej organizácie, autentifikácie a dôveryhodnosti, čo len podčiarkuje problém heterogénnosti.

- Prispôsobivosť – Nedostupnosť zdroja v gride je pomerne bežným javom. Práve vďaka obrovskému počtu gridových prostriedkov je pravdepodobnosť výpadku niektorého zdroja prirodzene vysoká. Systémy vyvinuté na správu a manažovanie týchto zdrojov musia brať tento fakt do úvahy aby dokázali dynamicky prispôsobiť svoje správanie a tak získali z dostupných prostriedkov a služieb čo najvyšší výkon [8].

Za posledné desaťročie sa komunita vývojárov gridovej infraštruktúry zaoberala návrhom a vývojom služieb v spomínaných oblastiach. Vznikol tak vrstvový model gridu, ktorý je znázornený na obrázku č. 4 a možno ho popísať nasledovne:

Obrázok č. 4: Vrstvový model gridu.

Základnou vrstvou sú **hardvérové prostriedky** globálne dostupné pre celý grid. Tvoria ich počítače, sieťové komponenty, dátové úložiská, archivačné zariadenia, zobrazovacie jednotky a pod. Je nutné poznamenať, že množina dostupných prostriedkov je veľmi dynamická a náchylná na zmeny spôsobené rôznymi faktormi – inštaláciou nových zdrojov, dočasným, alebo trvalým výpadkom používaných zdrojov, odstavenie spôsobené plánovanou údržbou, alebo aktualizáciou.

Priamo na hardvérové prostriedky nadväzujú **softvérové služby**, ktorých účelom je zjednotiť heterogénne zdroje gridu aby tak grid virtuálne pôsobil ako jednotná virtuálna platforma pre špecificky zamerané softvérové aplikácie. Je teda dôležité zvoliť štandard, ktorý určuje akým spôsobom sa bude k jednotlivým zdrojom gridu pristupovať. Vývojom

služieb na tejto úrovni sa zaoberá napr. vyššie spomínaný projekt Globus [9].

Ako nadstavbu týchto služieb môžeme chápať ďalšiu vrstvu – **middleware, používateľské nástroje a služby**. Ide o programové vybavenie, ktoré zjednodušuje a zakrýva niektoré nevyhnutné systémové operácie ako napríklad autentifikáciu, prenos súborov a pod. Tu patria aj používateľské rozhrania, plánovacie prostriedky, alebo iné programy, ktoré tvoria most medzi užívateľskými aplikáciami a softvérovými službami gridovej infraštruktúry.

Najdôležitejšou vrstvou je vrstva **používateľských gridových aplikácií**. Práve správna funkčnosť v tejto vrstve je cieľom fungovania gridového systému, preto je potrebné aby všetky vrstvy, ktoré tvoria jej základ efektívne tvorili robustnú stabilnú a použiteľnú platformu pre jej používateľov.

V súčasnej dobe je nevyhnutné prihliadať na prudký rozvoj **nových zariadení**, ktoré tiež môžu zohrať dôležitú úlohu pri vytváraní heterogénnej gridovej infraštruktúry a preto je vhodné počítať s ich prípadnou integráciou. Sem patria napríklad, mobilné zariadenia, bezdrôtové zariadenia, senzory, ktoré tvoria jednu vertikálnu vrstvu gridového modelu.

Práve vďaka neustálemu rozširovaniu rôznorodosti zariadení a prostriedkov dostupných v gridovej infraštruktúre je potrebné navrhnúť správne **politiky používania spoločných prostriedkov**, ekonomického **využitia zdrojov** a **správy siete**. Tieto požiadavky môžeme zhrnúť do poslednej vertikálnej vrstvy v modeli gridovej infraštruktúry.

1.5.3 III generácia, virtualizácia, cloudy a elastické klastre

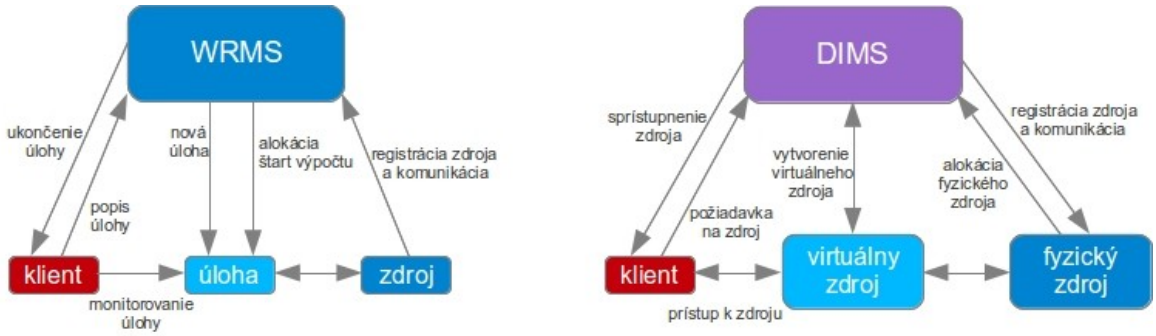
Architektúra gridového počítania je charakteristická spoločným prístupom k výpočtovým zdrojom. Nové prístupy, ktoré sa uplatňujú v súčasnej dobre prinášajú možnosť získať prístup k výpočtovým prostriedkom, infraštruktúram a službám na požiadanie (angl. on-demand computing). Označujú sa ako cloudové počítanie (angl. cloud computing). Charakteristickým prvkom cloudového počítania je virtualizácia. Vďaka súčasným vyspelým virtualizačným technológiám je možné využiť jednotné fyzické prostredie pre rôzne platformy a služby. Zdroje môžu byť buď verejné (angl. public cloud), alebo súkromné (angl. private cloud). Verejné zdroje vlastnia takzvaní cloudoví poskytovatelia (angl. cloud provider) a ponúkajú ich zákazníkom na základe času, ktorý

strávia využívaním týchto prostriedkov. Cloudové počítanie ponúka štyri základné druhy služieb:

- Infraštruktúra ako služba (IaaS z angl. Infrastructure as a Service)
- Platforma ako služba (PaaS z angl. Platform as a Service)
- Softvér ako služba (SaaS z angl. Software as a Service)
- Sieť ako služba (NaaS z angl. Network as a Service)

Každá zo spomenutých technológií (vysokovýkonné počítanie, gridové počítanie, cloudové počítanie) má svoje výhody a nevýhody. V súčasnej dobe začínajú prevládať prístupy, ktoré využívajú výhody týchto troch druhov počítania. Nazývajú sa tzv. **elastické klastre [10]**. Elastické klastre kombinujú prístupy dvoch modelov:

- model správy zdrojov a záťaže (WRMS z angl. Workload Resource Management System). Podľa tohto modelu pracuje väčšina súčasných gridových a klastrových prostriedkov pre správu zdrojov.
- model dynamickej správy infraštruktúry (DIMS z angl. Dynamic Infrastructure Management System). Základom tohto modelu je dynamické vytváranie a prideľovanie virtuálnych výpočtových prostriedkov.



Obrázok č. 5: Základné modely manažéra záťaže a zdrojov (WRMS) a dynamického manažéra infraštruktúry (DIMS)

Elastický klaster využíva obidva modely (obrázok č. 5) a zabezpečuje ich využitie pomocou tzv. agenta, čím dosahuje efektívne plánovanie a správu pracovnej záťaže a zároveň využíva výhody virtualizácie a dynamického prideľovania zdrojov.

1.6 Gridové systémy a projekty v Európe

S nárastom počtu gridových infraštruktúr a gridových projektov vzniklo aj niekoľko medzinárodných skupín vedcov a výskumníkov, ktorí sa rozhodli spojiť tieto komponenty do jedného uceleného systému. Takýchto systémov vzniklo niekoľko príkladom sú projekt DataGrid, ktorý neskôr vyústil do súčasného projektu EGI.

1.6.1 DataGrid

Projekt fungujúci v rokoch 2002 – 2004 bol sponzorovaný Európskou Úniou a bol vyvíjaný primárne pre spracovávanie obrovského množstva údajov, ktoré ročne vyprodukujú simulácie najväčšieho urýchľovača častíc vo výskumnom centre CERN. Projekt bol rozdelený na 12 pracovných balíkov vyvíjaných v 4 skupinách: testovacie prostredie a infraštruktúra, aplikácie, výpočtový a gridový middleware, správa a monitorovanie. DataGrid bol postavený na princípoch a službách projektu Globus a zahŕňal nasledovné komponenty:

- **Job description language** (JDL): skriptovací jazyk určený na popisovanie parametrov úloh
- **Používateľské rozhranie** (UI) – odosiela úlohy plánovaču a získava výsledky
- **Plánovač** (RB) – vyberá vhodné výpočtové jednotky pre úlohy
- **Služba odoslania úlohy** (JSS) – spúšťa úlohu na danej výpočtovej jednotke
- **Logovanie a monitorovanie** (L&B) – udržuje záznamy o stave úloh
- **Informačná služba gridu** (GIS) – register pre monitorovanie dostupných gridových prostriedkov
- **Katalóg** – zoznam údajov a ich záloh na pamäťových jednotkách

V roku 2004 bol projekt DataGrid ukončený no jeho súčasti boli použité v projekte EGEE – Enabling Grids for E-Science, ktorý bol úspešne vyvíjaný do roku 2010. Jeho pokračovaním a zároveň zjednotením viacerých gridových európskych projektov je projekt EGI.

1.6.2 EGI

Už niekoľko rokov po predstavení gridovej infraštruktúry vznikli v rôznych krajinách sveta ich vlastné gridové iniciatívy. Za účelom ich zjednotenia a efektívneho využívania zdrojov, ktoré ponúkajú jednotlivé národné gridové iniciatívy vznikla federácia viac ako 350 výpočtových centier – European Grid Infrastructure (EGI). Za cieľ si kladie vytvoriť celoeurópsku trvalo udržateľnú infraštruktúru pre výskum a inováciu v Európe. Projekt, ktorého predpokladané ukončenie je v roku 2014 má názov EGI-InSPIRE (EGI-Integrated Sustainable Pan-European Infrastructure for Research in Europe) [11].

1.7 Gridové systémy pre správu zdrojov a plánovanie

Existuje niekoľko dostupných systémov, ktorých účelom je dávkovanie a plánovanie, resp. rozvrhovanie zdrojov. Väčšina z týchto systémov boli pôvodne vyvíjané ako programy pre plánovanie a správu úloh na lokálnych distribuovaných platformách.

Condor je softvérový balík, ktorý dokáže spúšťať úlohy v dávkach na rôznych UNIX-ových platformách, obzvlášť dokáže využívať tie prostriedky, ktoré by za iných okolností boli nečinné. Hlavnými charakteristikami systému Condor sú automatická alokácia dostupných zdrojov pre daný zoznam úloh, vytváranie bodov pre obnovu (angl. check-pointing) a migrácia procesov za účelom uvoľnenia vyťaženého zdroja [12]. Tieto súčasti sú všetky dosiahnuté bez modifikácie jadra operačného systému na ktorom Condor beží. Nutnosťou je ale po skompilovaní zdrojového kódu pripojiť knižnice systému Condor. Systém monitoruje aktivity na všetkých dostupných prostriedkoch v distribuovanom systéme a tie stroje, ktoré sú vyhodnotené ako nevyužité sú zaradené do zoznamu voľných zdrojov a následne im priraďuje používateľské programy na vykonanie. Zoznam dostupných zdrojov je veľmi dynamický a jeho stav sa mení podľa stupňa záťaže jednotlivých strojov.

Portable batch system (PBS) je systém na správu zdrojov a spravovania záťaže. Tento systém bol pôvodne vyvinutý pre výpočtové strediská americkej služby NASA. Umožňuje spravovať úlohy a prostriedky na rôznych UNIX-ových platformách [12]. Ponúka na výber niekoľko spôsobov ako spravovať úlohy a využívať prostriedky systému. Toto sa nazýva politika plánovania. V súčasnosti existujú 3 jeho verzie, z ktorých pôvodná verzia OpenPBS už nie je aktívne vyvíjaná. Namiesto nej pracujú vývojári na komerčnej verzii PBS Porfessional a stále je dostupná aj open-source vetva projektu pod názvom

Torque. PBS je podporovaný ako plánovací mechanizmus pre rôzne distribuované systémy medzi inými aj modul GRAM projektu Globus Toolkit.

Oracle Grid Engine, pôvodne známy ako The Sun Grid Engine (SGE) je plánovací a dávkovací systém používaný hlavne v homogénnych a heterogénnych počítačových klastroch. Je schopný prijímať úlohy, ktoré mu zadá používateľ spolu s popisom ich požiadaviek, udržiava zoznam týchto úloh a pri uvoľnení niektorého radu určeného na vykonávanie programov na konkrétnych počítačoch vyberie pomocou plánovacieho algoritmu vhodný program [13].

The Load Sharing Facility (LSF) je komerčný systém pôvodne vyvinutý na univerzite v Toronte pod názvom Utopia. Podľa niektorých zdrojov je to v súčasnosti najpoužívanejší komerčný produkt pre správu úloh v distribuovaných prostrediach. Obsahuje v sebe algoritmy vyvažovania záťaže, dávkového spracovania úloh, správy výpočtových prostriedkov ale aj monitorovacie a analytické služby pre sieťové prostredie homogénnych alebo heterogénnych počítačových systémov [14].

2 Skúmanie plánovacích algoritmov v gridovom prostredí

V nasledujúcej časti práce sa venujeme problémom záťaže v gridovom prostredí a jej vplyvu na celkový výpočtový čas potrebný na vykonávanie úloh. Naznačujeme dva základné spôsoby vyvažovania záťaže a dosiahnutia optimálnych časových výsledkov. Ďalej sa podrobne venujeme problémom gridového plánovania ako jedného z kľúčových faktorov pre minimalizáciu výpočtového času. Predstavujeme základné fázy a skúmame výpočtovú náročnosť plánovania. Na základe optimalizačných kritérií navrhujeme riešenia, ktorých cieľom je dosiahnuť efektívne plánovanie v reálnom čase. Prezentujeme použitie optimalizačného heuristického algoritmu založeného na prírodných a sociálno-ekonomických javoch a analyzujeme jeho výkonnosť a náročnosť. Ďalej sme sa rozhodli tento algoritmus porovnať s niekoľkými návrhmi plánovacích algoritmov, ktoré sú v súčasnosti bežne používané v produkčných systémoch. Rozhodli sme sa porovnať účinnosť týchto algoritmov v rôznych úrovniach gridového plánovania.

2.1 Metodológia a ciele

Naším hlavným cieľom je navrhnúť a implementovať plánovací algoritmus, ktorý bude schopný vykonávať základné funkcie potrebné pre efektívne využitie výpočtových prostriedkov na realizovanie náročných sekvenčných, alebo paralelných úloh. Základom plánovacieho algoritmu je pomerne mladá myšlienka podobná evolučným algoritmom. Je založená na optimalizácii pomocou mnohých častíc, ktoré vytvárajú roj. Za čiastkové ciele práce sme si stanovili otestovanie funkčnosti plánovacieho algoritmu pomocou simulácie virtuálneho modelu a preukázanie, že nami navrhovaný algoritmus sa dokáže minimálne vyrovnať alebo predčiť iné plánovacie algoritmy aj v reálnom prostredí.

V prvej časti nášho výskumu sme vytvorili virtuálny model gridu na základe dostupných informácií o výpočtových prostriedkoch, ktoré poskytuje český gridový projekt MetaCentrum. Pomocou gridového simulátora sme v tomto virtuálnom prostredí aplikovali na sériu náhodne vygenerovaných úloh s rôznymi veľkosťami vstupu a dĺžkami výpočtu dva známe a efektívne plánovacie algoritmy a porovnali ich s nami navrhovaným riešením. Základným kritériom, ktoré sme sa snažili minimalizovať bol čas dokončenia výpočtu všetkých úloh čo je možné dosiahnuť správnym vyvážením záťaže výpočtových uzlov.

V druhej časti sa zaoberáme implementáciou stochastického plánovacieho

algoritmu v prostredí lokálneho plánovača Torque [15]. Naprogramovaný plánovač sme nasadili na jeden virtuálny a jeden reálny vysokovýkonný klaster s rôznymi parametrami a odskúšali sme ich efektivitu. Celkové časy potrebné na realizáciu testovacích úloh naplánovaných pomocou nášho plánovača sme porovnali so základným plánovačom *pbs_sched*, distribuovaným spolu so softvérom Torque.

2.2 Záťaž v gridových systémoch

Ako sme spomenuli v úvode, základnou úlohou výpočtových zdrojov v gridovom prostredí je vykonávať programy, ktoré zadajú do systému používatelia. Procesory jednotlivých uzlov ako základné výpočtové jednotky obsluhujú procesy operačného systému, ktoré sú nevyhnutné pre chod počítača a správu jeho diskov, pamätí a periférnych zariadení. Už v tomto momente vzniká v systéme záťaž, aj keď len minimálna. Výber vhodného jadra operačného systému, ako aj doplnkových služieb a modulov ktoré operačný systém ponúka je prvým z faktorov, ktoré ovplyvňujú záťaž v jednotlivých uzloch gridového systému.

V gridovom systéme existuje veľká rôznorodosť výpočtových prostriedkov. Preto aj zdroje systému môžu byť využívané rôznymi spôsobmi. Prostriedky pre správu zdrojov v gridovom systéme obyčajne ponúkajú dve základné možnosti ako využívať dostupné uzly:

- **spoločné využívanie priestoru** (angl. space-sharing) – úloha priradená procesoru využíva celý procesor až pokiaľ nie je ukončený jej výpočet. V prípade viacerých dostupných procesorov sa jednej úlohe vyhradí skupina procesorov a žiadna iná úloha ich už nemôže využívať. Iné úlohy môžu byť priradené ďalším procesorom daného uzla.
- **spoločné využívanie času** (angl. time-sharing) – procesor striedavo vykonáva operácie rôznych programov, čím zdanlivo tieto programy bežia naraz. Ide o tzv. pseudo-paralelizmus a umožňuje vykonávanie viacerých úloh na tom istom procesore v rovnakom časovom období.

Voľba vhodnej politiky pre využívanie procesorov v jednotlivých uzloch tiež výrazne ovplyvňuje záťaž v systéme.

Najvýraznejšia záťaž vzniká samozrejme na základe výpočtov, z ktorých pozostávajú používateľské programy. Dôležitým faktorom sú štruktúry a typy údajov s

ktorými program pracuje. Výraznejšiu záťaž produkujú napríklad operácie s číslami s pohyblivou desatinnou čiarkou, menej zaťažujú počítač celo-číselné operácie. Práve na základe množstva a náročnosti operácií procesora je možné na úrovni operačného systému merať zaťaženie elementárnych výpočtových jednotiek a tým aj celého výpočtového uzla. Úlohou softvérových prostriedkov, ktoré slúžia na správu zdrojov v gridovom prostredí je získavať, udržiavať, porovnávať a poskytovať informácie o tejto záťaži za účelom vykonanie efektívnych rozhodnutí.

Klastre a superpočítače ako súčasti gridového systému nie sú vždy využívané iba gridovými používateľmi. Rovnako sa na nich môžu spúšťať svoje programy aj lokálni používatelia, ktorí nepatria do gridovej infraštruktúry. Títo tiež prispievajú k lokálnej záťaži gridových uzlov. Lokálna záťaž je dôležitým faktorom pre globálne monitorovanie výkonu a vyťaženia systému.

2.2.1 Ako dosiahnuť vyváženú záťaž

Heterogénnosť v gridovom systéme spôsobuje, že výkon jednotlivé výpočtové prostriedky sa od seba odlišujú svojím výkonom. Požiadavkou správcu systému je aby sa výpočtovo náročnejšie úlohy vykonávali na výkonnejších prostriedkoch a menej náročné programy na pomalších počítačoch. Tým sa dosiahne aj optimalizácia časových požiadaviek pre vykonanie úloh a zároveň vyvážená záťaž. Existujú dva základné spôsoby ako dosiahnuť aby sa používateľské programy vykonali na čo najvhodnejších zdrojoch [16]:

Plánovanie a priradenie úlohy na najvhodnejší z dostupných zdrojov systému pred začatím výpočtu. Označuje sa aj ako statické vyvažovanie záťaže. Po naplánovaní sa úloha začne vykonávať na zvolenom zdroji až do jej ukončenia. Pre zvolenie vhodných prostriedkov na ktoré sa má úloha naplánovať existuje niekoľko prístupov a niekoľko algoritmov, ktoré popisujeme v ďalšej časti práce. Pomocou vhodného plánovacieho algoritmu je možné dosiahnuť požadované priradenie úloh na výpočtové prostriedky tak, aby boli dosiahnuté požadované kritériá.

Migrácia úloh v systéme je dynamické vyvažovanie záťaže. Ide o presunutie programu pred jeho spustením počas čakania na vykonanie, alebo počas jeho behu na iné zdroje v prípade, že toto presunutie prinesie očakávané výsledky. Okrem migrácie úloh je možné v prípade použitia virtualizácie uskutočniť migráciu virtuálnych obrazov celých

systémov na iné fyzické prostriedky. Existujú rôzne spôsoby ako rozhodnúť či sa má migrácia uskutočniť a mnoho spôsobov ako určiť najvhodnejšie prostriedky pre migráciu. Základné druhy stratégií migrácie je možné rozdeliť nasledovne [17]:

- podľa iniciátora migrácie (odosielateľ vs. prijímateľ úlohy)
- globálne vs. lokálne stratégie migrácie
- centralizovaná vs. decentralizovaná migrácia
- spolupracujúca a nespolupracujúca migrácia
- adaptívna vs. neadaptívna migrácia
- jednorázová migrácia vs. dynamické migrovanie úloh

Významnou technológiou pre dosiahnutie vyváženej záťaže je aj spätné vypĺňanie rozvrhu (viď ďalej časť 3.4.3).

Nutným predpokladom pre všetky spôsoby rozhodovania je monitorovanie prostriedkov systému a získavania informácií o záťaži a úlohách na jednotlivých zdrojoch systému.

2.3 Plánovanie a rozvrhovanie v gridovom prostredí

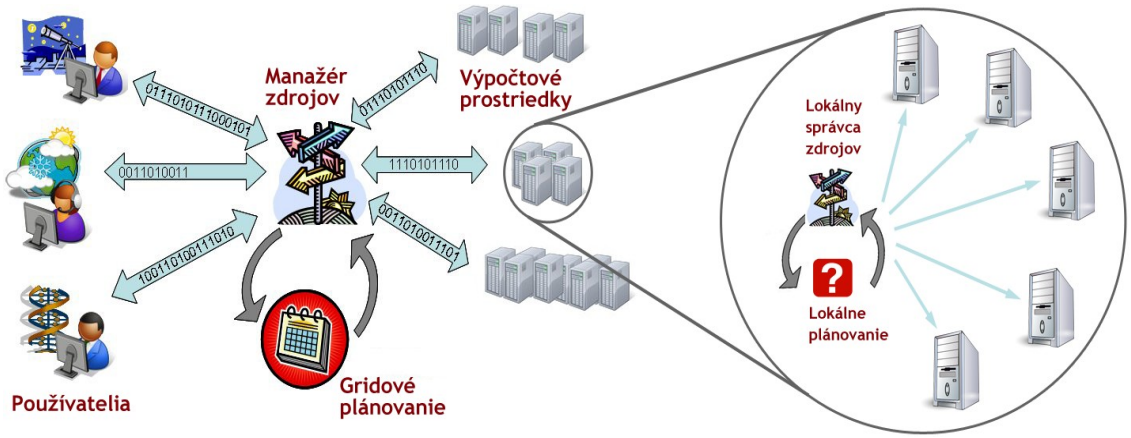
V nasledujúcej časti práce sa zaoberáme práve prvou z dvoch spomenutých techník, ktoré zabezpečujú vhodné priradenie úloh na výpočtové uzly – plánovaním.

Grid je vzájomné prepojenie mnohých výpočtových prostriedkov, výkonných strojov a zariadení. Často medzi ne patria aj menšie podsiete, ktoré tvoria vlastné administratívne domény s vlastnými politikami správy dostupných výpočtových prostriedkov. Tieto súčasti gridu, ktoré označujeme ako uzly (angl. node) majú, pokiaľ je to nutné, vlastné systémové prostriedky na správu zdrojov ktoré obsahujú a plánovanie úloh ktoré sa na nich vykonávajú. Pre superpočítače, alebo klastre, ktoré sú súčasťou gridu je takýto plánovač nevyhnutný. Je potrebné si uvedomiť, že jednotlivé uzly v gride nemusia byť využívané iba gridovými používateľmi, ale aj lokálnymi používateľmi daného uzla.

Plánovanie je proces, ktorého výsledkom je určenie výpočtového prostriedku na ktorom sa má vykonať používateľská úloha – aplikácia. V gridovom prostredí existuje niekoľko úrovní plánovania, ktoré medzi sebou úzko súvisia. Za najnižšiu vrstvu môžeme považovať plánovač operačného systému, ktorý zabezpečuje vykonávanie elementárnych úloh – procesov – na výpočtovej jednotke – procesore. Na správu úloh, ktoré sú

vykonávané v rámci jednej administratívnej domény (napr. počítačového klastra) slúži lokálny plánovač, ktorý väčšinou, na rozdiel od plánovania operačného systému, slúži aj na monitorovanie, správu a prideľovanie prostriedkov a zariadení. Na najvyššej vrstve je gridový plánovač, ktorý v závislosti od hierarchie gridového prostredia môže byť opäť rozdelený na niekoľko úrovní a zabezpečuje gridové plánovanie. Vzťahy globálneho a lokálneho plánovača znázorňuje obrázok č. 6.

Gridové plánovanie je definované ako proces vytvárania plánovacích rozhodnutí, ktoré zahŕňajú využívanie prostriedkov z rôznych administratívnych domén. Tento proces zároveň obsahuje aj prehľadávanie týchto domén za účelom výberu jedného stroja s konkrétnymi parametrami, alebo priradenie jednej úlohy viacerým počítačom v gridovom uzle, alebo dokonca viacerým počítačom vo viacerých uzloch gridového prostredia [18].



Obrázok č. 6: Gridové globálne a lokálne plánovanie a ich vzájomný vzťah.

Hlavný rozdiel medzi gridovým plánovačom a lokálnym plánovačom je to, že gridový plánovač nemá priamy prístup k žiadnym výpočtovým prostriedkom. Lokálny plánovač alebo manažér je väčšinou jediný prístupový bod, cez ktorý používatelia komunikujú s výpočtovými prostriedkami v lokálnom systéme. Má preto priamu kontrolu nad dostupnými prostriedkami. Gridový plánovač môže získať od lokálneho správcu prostriedkov informácie o dostupných zdrojoch a po vykonaní plánovacieho rozhodnutia, ktoré spočíva vo výbere miesta pre spustenie úlohy odošle túto úlohu na spracovanie lokálnemu manažérovi zdrojov. S ním komunikuje pomocou rovnakých protokolov ako obyčajný lokálny používateľ. Gridový plánovač spravidla nemá informácie o úplne všetkých úlohách, ktoré sa v danom systéme nachádzajú. Toto je jeden z nedostatkov, ktoré je potrebné často riešiť.

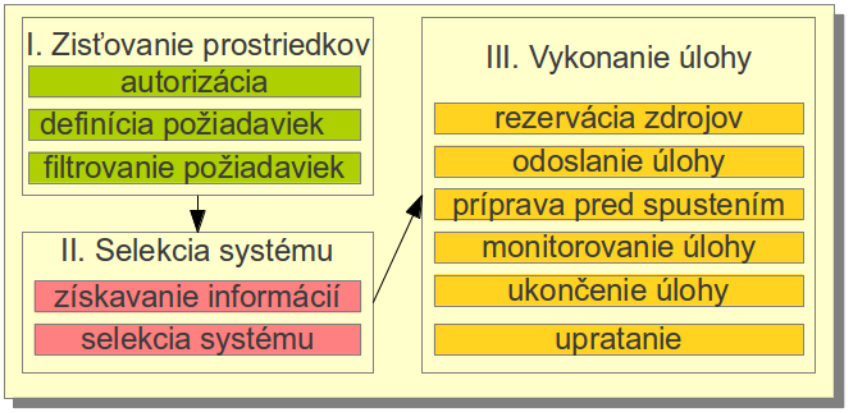
Aby program, ktorý funguje ako gridový plánovač nemusel komunikovať priamo s jednotlivými manažérmi prostriedkov gridových uzlov tieto služby mu poskytuje Gridová Informačná Služba (GIS). Príkladom je služba MDS2 (z angl. Monitoring and Discovery Service), ktorá plní úlohu GIS v projekte Globus. Hoci môžu byť informačné služby implementované rôzne majú isté spoločné charakteristiky. Ich základnou úlohou je zhromažďovanie informácií tak, aby boli jednotné voči vonkajšiemu používateľovi služby. Niektoré informácie sú statické (napr. operačný systém, celková fyzická pamäť, ktorý súborový systém je prístupný a pod.), iné sú dynamické (veľkosť voľného miesta na disku, voľná operačná pamäť). Existujú rôzne spôsoby akými informačná služba pracuje s dynamickými charakteristikami gridového systému (prúdová komunikácia, dočasné vyrovnávacie pamäte). Pre účinnú komunikáciu medzi informačnou službou a gridovým plánovačom je potrebné aby bol vopred dohodnutý jednotný komunikačný protokol. Je diskutabilné, ktoré zo súčasných technológií sú na túto úlohu najvhodnejšie (LDAP, XML, SQL).

2.4 Fázy gridového plánovania

Nájdenie vhodného prostriedku pre vykonanie aplikácie odoslanej do gridu používateľom nie je jednoduché. Proces gridového plánovania teda v sebe zahŕňa nasledovné 3 fázy [18]:

- zisťovanie dostupných prostriedkov (angl. resource discovery)
- získavanie informácií o ich stave (angl. information gathering) a voľba najlepšieho prostriedku
- vykonanie úlohy (angl. job execution) zahŕňa spustenie úlohy na zvolených uzloch

Každá z týchto troch fáz pozostáva z niekoľkých krokov tak ako to znázorňuje obrázok č. 7.



Obrázok č. 7: 3 fázy gridového plánovania.

2.4.1 Zisťovanie dostupných prostriedkov

Pred tým ako sa plánovač pokúsi urobiť akékoľvek plánovacie rozhodnutie je nutné, aby získal zoznam zdrojov, ktoré sú nielen aktuálne dostupné v gridovom prostredí, ale spĺňajú isté nevyhnutné požiadavky. Základnou požiadavkou je aby mal používateľ dostatočné práva na ich využívanie. Toto sa deje v 3 krokoch:

1. **autorizácia** – prvý krok slúži na získania informácií o zdrojoch, ktoré sú dostupné GIS. Súčasťou tohto kroku je filtrovanie zdrojov ku ktorým používateľ nemá prístup. Zrejmým dôvodom je to, že v prípade priradenia úlohy na zdroj bez oprávnení táto úloha nebude spustená. Ďalším dôvodom je, že zdroje ku ktorým nemá vlastník úlohy prístup nemusia byť uvažované v ďalších krokoch plánovania čo efektívne znižuje výpočtovú náročnosť plánovača.
2. **definícia požiadaviek aplikácie** – požiadavky aplikácie sú parametre, ktoré musia byť dodržané aby bolo možné efektívne spustiť aplikáciu v gride. V súčasnosti najbežnejším spôsobom je definovanie týchto požiadaviek pri zadávaní aplikácie do gridu pomocou príkazového riadku, alebo spúšťacieho skriptu. Príkladom týchto požiadaviek je architektúra, operačný systém, požadovaná operačná pamäť a pod. V gridovom systéme sa v súvislosti s jeho heterogénnosťou vynára problém nejednotnosti požiadaviek pre rôzne uzly gridu. Napríklad veľkosť operačnej pamäte potrebnej pre vykonanie aplikácie je rôzna pri spustení aplikácie na počítačoch s rôznymi architektúrami.
3. **filtrovanie minimálnych požiadaviek** – po porovnaní požiadaviek danej aplikácie s informáciami o prostriedkoch, ktoré má používateľ k dispozícii je výsledkom

tohto kroku množina prostriedkov, ktoré sú potenciálne vhodné na vykonanie úlohy. Keďže dynamické informácie sú veľmi premenlivé, plánovače pracujú v tomto kroku len so statickými informáciami, alebo tento krok úplne vynechávajú a vhodné prostriedky filtrujú až v ďalších fázach keď sú dostupné aj dynamické informácie z GIS.

2.4.2 Selekcia systému

Výsledkom prvej fázy plánovania je množina prostriedkov gridu, ktoré sú vhodné pre vykonanie plánovanej úlohy a vlastník úlohy je oprávnený ich používať. Úlohou plánovača je vybrať vhodný zdroj z tejto množiny a priradiť mu úlohu. Selekcia sa dá popísať pomocou dvoch krokov, ktoré na seba veľmi tesne nadväzujú:

1. **získavanie dynamických informácií** – pred tým ako sa vykoná rozhodnutie o výbere zdroja je potrebné získať všetky informácie, ktoré môžu ovplyvniť toto rozhodnutie. Ide prevažne o dynamické informácie, ktorých aktuálnosť je dôležitá, ale niektoré plánovacie systémy spájajú tento krok aj so získaním statických informácií. Informácie sa získavajú komunikáciou s lokálnymi manažérmi zdrojov, alebo pomocou GIS, ktorá zabezpečuje túto komunikáciu. V súvislosti s rozšíriteľnosťou ako charakteristickou vlastnosťou gridu sa objavuje problém dostupnosti gridových prostriedkov. Niektoré prostriedky môžu byť zaneprázdnené, dočasne nedostupné, alebo naopak pridané do systému. Moderné plánovacie systémy obsahujú rôzne algoritmy predikcie a analýzy dostupnosti prostriedkov a stavu systému [19].
2. **výber najvhodnejšieho gridového prostriedku** – z množiny dostupných prostriedkov, ktoré spĺňajú minimálne požiadavky úlohy sa pomocou rôznych techník a kritérií zvolí ten, ktorý bude z hľadiska plánovacej politiky najvhodnejší. Existuje mnoho algoritmov, spôsobov a kritérií podľa ktorých sa toto plánovanie vykonáva. Príkladom sú: dohadzovanie (matchmaking) – používa napr. Condor, plánovanie podľa viacerých kritérií (multi-criteria) a rôzne heuristické plánovacie algoritmy [20].

2.4.3 Vykonanie úlohy

Tretia fáza zabezpečuje vykonanie úlohy na počítači, alebo viacerých počítačoch na ktoré bola naplánovaná v druhej fáze. Pozostáva z piatich krokov:

1. **rezervácia zdrojov** – tento krok je voliteľný a pokiaľ to lokálny systém podporuje zabezpečuje dostupnosť prostriedkov v čase, spustenia úlohy. Výhodou je jednoduchšia predikcia predpokladaného času spustenia a čakania úloh v rade lokálneho plánovača.
2. **odoslanie úlohy** – po zvolení najvhodnejšieho zdroja pre danú úlohu je možné aplikáciu odoslať na lokálnemu správcovi úloh a prostriedkov. Toto odoslanie môže byť vykonané jednoduchým príkazom, alebo sériou zložitých skriptov, ktoré obsahujú informácie potrebné pre spustenie aplikácie. Pre unifikáciu spôsobu odosielania úloh na lokálne zdroje existuje štandard JSDL (z angl. Job Submission Description Language), ktorý popisuje parametre odosielania úlohy lokálnemu plánovaču.
    - Názov a popis aplikácie
    - Požiadavky na prostriedky systému (RAM, swap, rýchlosť CPU, počet procesorov/jadier)
    - Limity výpočtu (čas behu úlohy, čas procesora, pamäťové limity)
    - Vstupné a výstupné súbory a ostatné prostriedky s ktorými aplikácia pracuje
    - Príkaz, ktorým sa aplikácia spúšťa, vrátane argumentov, systémových premenných a presmerovania štandardných a chybových vstupov a výstupov.
3. **príprava** – slúži na vykonanie všetkých procesov, ktoré sú potrebné na spustenie aplikácie (prenos súborov na daný počítač, resp. všetky počítače na ktorých sa bude úloha vykonávať). V tomto kroku sa rezervujú všetky potrebné zdroje na lokálnom uzle gridu pre následné spustenie úlohy. Samotné spustenie zabezpečí lokálny plánovač.
4. **monitorovanie aplikácie** – lokálny správca úloh dokáže odpovedať na požiadavku o stave úlohy a jej vývoji. V prípade, že úloha nespĺňa požiadavky používateľa, čo sa týka jej progresu je možné zmeniť jej umiestnenie a spustiť ju na inom gridovom uzle (vrátiť sa do druhej fázy plánovania). Takáto migrácia úlohy je v gridovom prostredí oveľa náročnejšia ako v obyčajnom paralelnom systéme a aby to bolo efektívne je nutné vyvinúť mechanizmy pre lokálne plánovače, ktoré budú túto úlohy zabezpečovať [18].
5. **ukončenie úlohy** – pozostáva zo zhromaždenia výstupov aplikácie a monitorovacích informácií a z informovania používateľa o úspešnom, alebo

neúspešnom ukončení úlohy. Najčastejšie pomocou e-mailovej komunikácie.

6. **odstránenie dočasných súborov** – používateľ môže tieto úlohy vykonať manuálne po ukončení úlohy, alebo zahrnúť potrebné akcie do spúšťacích a ukončovacích skriptov, ktoré sú súčasťou jeho aplikácie.

2.5 Plánovací algoritmus

V ďalšej časti práce sa budeme zaoberať technikami, ktoré sa používajú v 2. kroku 2. fázy plánovania. Dôležitosť týchto algoritmov a ich praktický vplyv na chod gridového systému si ukážeme na nasledujúcom triviálnom príklade. Princípy na ktorých je príklad založený platia rovnako v globálnom, ako aj v lokálnom meradle gridu.

Príklad – dôležitosť plánovacieho algoritmu

Uvažujme modelový príklad gridu s 3 výpočtovými uzlami, ktoré zahŕňajú 1 výkonný klaster, obsahujúci 300 výpočtových jadier a 2 počítačové klastre, pričom každý obsahuje 100 výpočtových jadier. Teoretický fyzik odošle do systému 30 paralelných úloh, ktoré nevyžadujú presne špecifikovaný počet procesorov ale vykonávajú rovnaký počet elementárnych operácií. Bez náročnejšieho plánovania budú tieto úlohy postupne priraďované klastrom v poradí 1,2,3,1,2,3... atď. V konečnom dôsledku sa na každom klastri vykoná 10 z týchto úloh. Keďže prvý klaster je 3-násobne výkonnejší oproti zvyšným dvom, úlohy ktoré mu boli priradené sa vykonajú 3-krát rýchlejšie. Aby mohol fyzik pokračovať v práci musí čakať na ukončenie poslednej úlohy na pomalšom klastri. Ak jedna úloha na rýchlejšom klastri trvá 8 hodín, na pomalšom klastri trvá 24 hodín, potom fyzik bude čakať 10 dní.

Uveďme do činnosti inteligentnejší plánovací systém, ktorý dokáže zohľadniť výkonnosť klastrov a naplánuje na výkonný klaster 18 úloh a na pomalšie klastre iba po 6 úloh. Keďže prvý klaster je 3-násobne výkonnejší ale spracováva 3-krát viac úloh všetky klastre skončia svoju činnosť v rovnakom čase a fyzik musí čakať iba 6 dní a môže pokračovať vo svojej práci.

Tu sa ale náš príklad nekončí. Chod počítačového klastra je finančne náročná záležitosť, ktorá zahŕňa údržbu, administráciu ale hlavne elektrickú spotrebu na chod jednotlivých počítačov. Výkonnejší klaster používa lepšiu technológiu a jedna hodina práce jedného procesora stojí 1 cent. Oproti tomu práca procesora na pomalšom klastri stojí 1.5

centa na hodinu. V prvom prípade boli náklady prvého klastra 300x10x8x1 + 100x10x24x1.5 + 100x10x24x1.5 = 24 000 + 36 000 + 36 000 = 96 000c = 960 EUR

V druhom prípade boli náklady 300x18x8x1 + 100x6x24x1.5 + 100x6x24x1.5 = 43 200 + 21 600 + 21 600 = 86400 = 864 EUR.

Efektívne použitie plánovača nielen znížilo čas potrebný na ukončenie úloh, ale ušetrilo prevádzkovateľom gridu takmer 100 EUR.

2.6 Teoretické predpoklady, formalizácia vstupných údajov

Ako je uvedené v predchádzajúcej časti, plánovač má v čase vykonávania svojej činnosti k dispozícii dynamické a statické informácie o gridových prostriedkoch, úlohách a ich parametroch. Plánovacie algoritmy pracujú s mnohými teoretickými informáciami, ktoré možno formálne popísať nasledovne:

- úloha *j* – program, ktorý je možné spustiť na počítači,
- množina *J* (z angl. job) – množina *n* úloh *j,* ktoré v danom momente existujú v systéme, $J = \{j_1, j_2, \ldots, j_n\}$
- uzol *i* – gridový výpočtový prostriedok (počítač, klaster), na ktorom je možné spúšťať úlohy.
- množina *M* (z angl. machine) – množina *m* počítačov *i,* ktoré sú súčasťou paralelného systému, $M = \{i_1, i_2, \ldots, i_m\}$
- S_j – čas spustenia úlohy *j*,
- C_j – čas ukončenia úlohy *j*,
- C_{ij} – čas ukončenia úlohy *j* na počítači *i*,
- p_j – čas vykonania úlohy *j,* $p_j = C_j - S_j$
- d_j – časový limit ukončenia úlohy *j* (angl. walltime), $d_j \geq p_j$
- w_j – váha úlohy *j*. Vyjadruje dôležitosť alebo prioritu úlohy.
- s_j – čas potrebný na vykonanie operácií pred samotným spustením úlohy *j*,
- *zdroje(i)* – Množina prostriedkov, ktoré ponúka uzol *i* a možno ich kvantitatívne vyjadriť. Jej najdôležitejšie prvky pre plánovač sú:
  - *mem* – dostupná operačná pamäť (v B),
  - *ncpus* – počet výpočtových jednotiek (procesorov, jadier),

- *ngpus* – počet grafických výpočtových jednotiek,
- *speed* – rýchlosť výpočtovej jednotky (v MIPS – milión inštrukcií za sekundu).

Uzol *i* má aj isté vlastnosti, ktoré sa nedajú vyčísliť ale je možné slovne ich popísať. Najdôležitejšie sú:

- *arch* – architektúra,
- *OS* – operačný systém.

- M_j – popisuje požiadavky, ktoré musia byť splnené, aby mohla byť úloha *j* spustená. Pre teoretický odhad náročnosti aplikácie sú dôležité najmä:
  - *input_size* – veľkosť vstupného súboru pred začatím vykonávania úlohy (v B),
  - *output_size* – veľkosť výstupného súboru po skončení výpočtu (v B),
  - *length* – dĺžka úlohy (v MI – miliónoch inštrukcií),
  - *mem* – požadovaná pamäť (v B),
  - *ncpus* – požadovaný počet procesorov,
  - *ngpus* – požadovaný počet grafických procesorov.
- l_{xy} – rýchlosť sieťového spojenia medzi uzlami *x* a *y.* (v Mbit/s).

Očakávania používateľov a administrátorov, ktoré sa týkajú rýchlosti spustenia a dokončenia úloh a efektívneho využívania gridových prostriedkov môžeme vyjadriť nasledovnými metrikami:

- čas dokončenia (angl. makespan) – určuje čas dokončenia všetkých úloh [21].

  $$C_{max} = max(C_i : i = 1..m)$$

  Pričom C_i je čas dokončenia poslednej úlohy na uzle *i.* $C_i = max(C_j : j = 1..n)$
- využitie prostriedkov (angl. resource utilization) – koľko zdrojov je vyťažených vykonávaním úloh. Pre každý zdroj môžeme určiť ako je celkovo využívaný na základe požiadaviek *r* úlohy *j* na zdroj *z*.

  $$\forall z \in zdroje(i) \rightarrow požiadavky_z = \sum_{j=0}^{n} r_{j,z}$$

  Následne je možné ohodnotiť využitie zdrojov funkciou [20]:

  $$util = \sum_{zdroje(i)}^{z} \left(\frac{max(požiadavky_z, kapacita_z)}{kapacita_z} - 1 \right)$$
- spoľahlivosť služieb (angl. service reliability) [22] – je definovaná ako:

- dostupnosť – prostriedky sú dostupné používateľovi v požadovanom čase
- nepretržitosť – prostriedky sú dostupné počas celej požadovanej doby
- výkonnosť – prostriedky spĺňajú očakávania používateľa z hľadiska požadovaného výkonu.

- spravodlivé rozdeľovanie – každý používateľ má rovnakú príležitosť využívať gridové prostriedky, rovnako ako každý poskytovateľ má rovnakú príležitosť na využívanie jeho prostriedkov a tým dosiahnutie adekvátneho zisku [23].
- úspešné vykonanie úlohy – ukončenie úlohy v rámci požadovaného času [23].
- čas odozvy – dĺžka času, ktorý uplynie od zadania úlohy do jej úspešného ukončenia [22].
- priepustnosť – počet úspešne ukončených úloh za isté časové obdobie.

Niektoré z uvedených metrík sú navzájom protichodné a preto plánovač, ktorý zabezpečuje priraďovanie úloh na jednotlivým uzlom musí zvoliť jednu metriku, ktorou sa bude určovať efektívnosť plánovania, alebo nájsť riešenia, ktoré sú kompromisom v týchto metrikách.

2.7 Prístupy k plánovaniu a rozvrhovaniu úloh

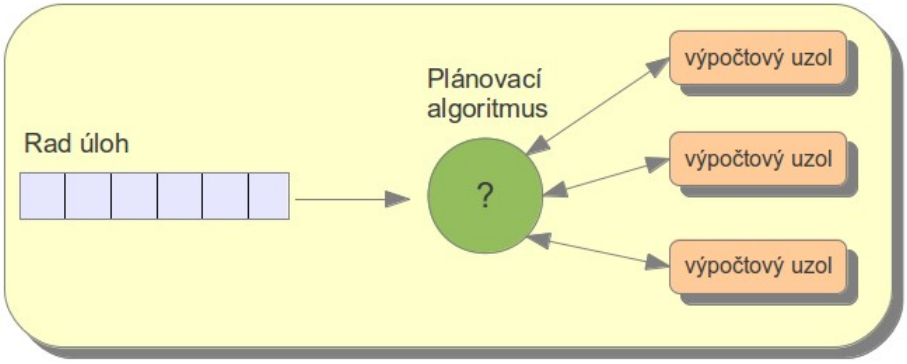
Všeobecne môžeme rozlíšiť dva prístupy k plánovaniu úloh na uzly [24]:

- plánovanie pomocou radov,
- plánovanie pomocou rozvrhu.

2.7.1 Plánovanie pomocou radov

Základnou štruktúrou, ktorú používa plánovač je rad. Rad je zoznam úloh odoslaných do systému používateľmi. Úlohy v rade sú zoradené podľa politiky plánovača, väčšinou podľa poradia príchodu do zoznamu (FCFS). Stratégiu zoradenia môže zvoliť administrátor a bežne používané sú: zoradenie podľa najkratšej úlohy (SJF), najdlhšej úlohy (LJF) atď. Po odoslaní úlohy a jej zaradení do radu sa plánovač pokúsi spustiť všetky úlohy z radu, alebo v prípade použitia viacerých radov, jednu úlohu zo začiatku každého radu. Je možné použiť aj kombináciu týchto prístupov. Mapovanie úloh na jednotlivé uzly a pridelenie zdrojov sa deje na základe plánovacej politiky. Ide o algoritmus, do ktorého vstupujú mnohé parametre ovplyvniteľné administrátorom

(vyťaženie systému, časové obdobie a pod). Základným faktom je, že uzol sa priraďuje úlohe až keď je čas na jej spustenie (obrázok č. 8). V prípade, že sa nenájde taký uzol, ktorý má dostatok zdrojov pre úlohu, táto úloha ostane zaradená v rade a plánovač sa pokúsi spustiť ďalšiu úlohu v rade.



Obrázok č. 8: Plánovanie pomocou radu.

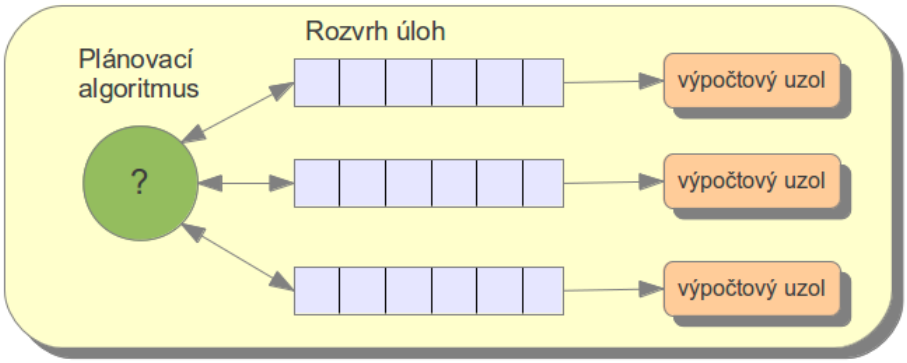
Manažér zdrojov spúšťa plánovač pri rôznych udalostiach: odoslanie novej úlohy, ukončenie bežiacej úlohy, v pravidelnom časovom intervale... Takéto plánovanie využívajú mnohé produkčné systémy ako napr. PBS Pro [12], LSF [14], Sun Grid Engine (SGE) [13], Condor [25], Maui a Moab [26].

2.7.2 Plánovanie pomocou rozvrhu

Protikladom plánovača, ktorý funguje pomocou radu je plánovač, ktorý pracuje s dátovou štruktúrou, ktorú môžeme označiť ako rozvrh. Prakticky ide o zoznam úloh, pričom každá úloha zároveň obsahuje aj informáciu o tom, na ktorom výpočtovom uzle bude spustená (obrázok č. 9). Vytvorenie a udržiavanie takéhoto rozvrhu je možné dvoma spôsobmi:

- Pri prijatí novej úlohy sa vygeneruje rozvrh, ktorý zohľadňuje požiadavky všetkých úloh. Pri prijatí každej ďalšej úlohy sa tento rozvrh vytvára od začiatku. Následne je možné rozvrh optimalizovať, aby sme dosiahli čo najefektívnejšie mapovanie úloh na jednotlivé uzly.
- Druhý spôsob pracuje s rozvrhom, ktorý sa vytvorí na začiatku a každú novú úlohu prichádzajúcu do systému zaradí do existujúceho rozvrhu tak, aby bol opäť výsledný rozvrh čo najefektívnejší.

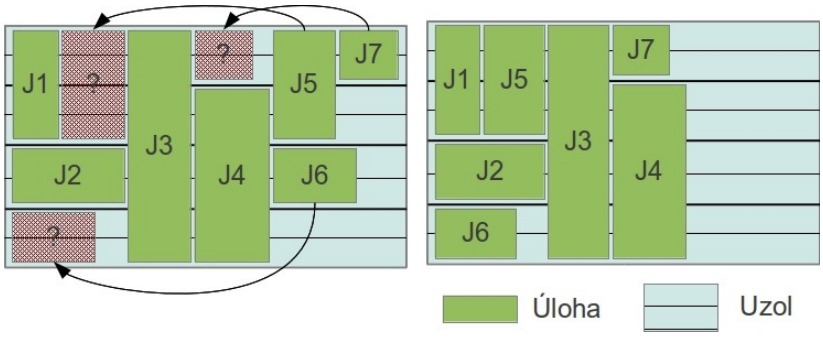
Efektivitu zistíme a porovnáme práve podľa vyššie spomínaných kritérií (priepustnosť, čas dokončenia. čas odozvy atď). Hodnoty týchto metrík vieme z daného rozvrhu zistiť hlavne podľa predpokladaného čau behu programu a predpokladaného času spustenia, ktorý vyplýva z rozvrhu.



Obrázok č. 9: Plánovanie pomocou rozvrhu.

2.7.3 Spätné vypĺňanie rozvrhu

Významnou technikou používanou ako súčasť plánovania pomocou rozvrhu je tzv. spätné vypĺňanie rozvrhu (angl. backfilling). Základnou štruktúrou je rozvrh, ktorá úloha sa má spustiť na ktorom zdroji (zdrojoch) a v akom čase. Na základe počiatočných a koncových časov úloh je možné v rozvrhu nájsť medzery pre vhodne zvolené úlohy tak, aby boli vykonané skôr a celkový čas vykonania všetkých úloh sa nezhoršil. Príklad rozvrhu pred a po aplikovaní techniky spätného plánovania je na obrázku č. 10.



Obrázok č. 10: Technika spätného vypĺňania medzier v rozvrhu

Technika spätného plánovania zabezpečuje lepšie využitie dostupných zdrojov tým, že umožňuje vykonávanie úloh mimo ich pôvodného poradia [27].

3 Algoritmy pre optimalizáciu plánovania

Problém mapovania úloh na systémové prostriedky sa klasifikuje ako NP-úplný [28]. Na riešenie takéhoto problému s rozsiahlejším vstupom nie je dosiaľ známy žiadny algoritmus, ktorý by dokázal v konečnom čase dospieť k dokázateľne najlepšiemu riešeniu takejto úlohy. Preto boli vymyslené rôzne heuristické algoritmy, ktoré sa aplikovaním špecifických postupov snažia čo najviac priblížiť k výsledku. Heuristické algoritmy sú často založené na pozorovaní komplexných prírodných javov, alebo dejov okolo nás. Hoci nepoznáme zákonitosti týchto dejov vieme vytvoriť abstraktný matematický model, ktorým ich dokážeme priblížiť a odpozorovať ich správanie. V našej práci sme sa zaoberali skúmaním optimalizácie založenej na princípe pohybu častíc v roji (PSO z angl. Particle Swarm Optimization), horolezeckým algoritmom (HC z angl. Hill Climbing) a ich možným použitím pri plánovaní úloh v multipočítačovom klastri a gride. Tieto algoritmy sme porovnávali s najbežnejším a najjednoduchším plánovacím algoritmom RR (angl. Round Robin), resp. FIFO (angl. First In First Out). Ďalej sme sa zaoberali ich vzájomným porovnaním a analýzou ich vhodnosti pre paralelný systém. Predpokladom nášho skúmania je, že vopred poznáme stav výpočtových jednotiek, čiže ide o statické plánovanie. Rovnako vieme vopred určiť veľkosť každej úlohy a úlohy prichádzajúce do systému sú navzájom nezávislé a preto nemusíme uvažovať ich vzájomné poradie. Ďalším predpokladom je, že systém je nepreemptívny a teda každá ďalšia úloha začne až po skončení tej predchádzajúcej.

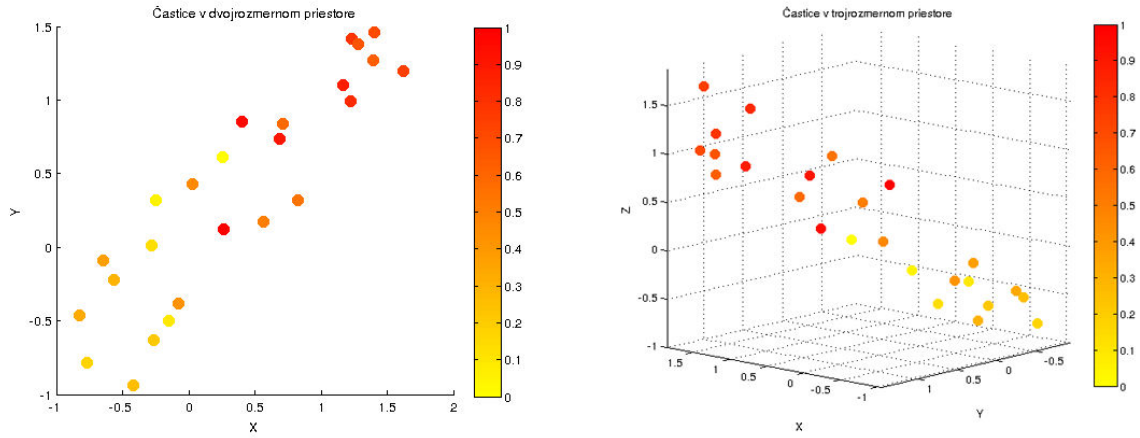
3.1 Optimalizácia časticami roja

Algoritmus optimalizácie časticami roja bol po prvý krát predstavený v roku 1995 a jeho autori sú páni James Kennedy and Russel Eberhart [29]. Je založený na základe pozorovania prírodných javov. Ide o heuristický, dokonca až meta-heuristický algoritmus, ktorý negarantuje 100% správne riešenie problému, ale iteračnými metódami sa snaží čo najviac k nemu priblížiť. Implementuje tzv. stochastickú optimalizáciu, čo znamená, že zahŕňa element náhody na rozdiel od tzv. deterministickej optimalizácie, ktorá sa vyznačuje exaktnosťou a presnou znalosťou postupu výpočtu.

Predlohou algoritmu PSO je roj včiel, vtákov, alebo iných pohybujúcich sa častíc, ktoré smerujú k istému cieľu. Na čele je väčšinou častica, ktorá významne ovplyvňuje smer

letu ostatných častíc. Okrem toho smer každej jednej častice môžu ovplyvniť aj jej najbližší susedia a ich správanie. Pre matematické popísanie zákonitostí, ktorými sa riadi roj častíc je potrebná vysoká miera abstrakcie.

Podstatná informácia pre náš algoritmus je poloha častice. V n-rozmernom priestore ju presne popisuje práve n súradníc. S časticou teda môžeme pracovať ako s jednorozmerným poľom (vektorom) s veľkosťou n, ktorého prvky sú súradnicami, určujúcimi polohu častice v priestore. Poloha častice reprezentuje jedno z možných riešení problému a nevyhnutným predpokladom fungovania algoritmu je možnosť ohodnotiť toto riešenie pomocou zvolenej funkcie, ktorej vstupom je práve n argumentov. Cieľom algoritmu je nájsť globálne minimum, alebo maximum tejto funkcie, ktoré zodpovedá najlepšiemu možnému riešeniu problému. Príklady častíc v 2-rozmernom a 3-rozmernom priestore ukazuje obrázok č. 11.



Obrázok č. 11: Častice v 2-rozmernom a 3-rozmernom priestore. Farba určuje ako blízko sa častica nachádza k najlepšiemu riešeniu.

Pohyb častice v roji znamená, že sa v každom časovom okamihu s istou rýchlosťou mení jej poloha.

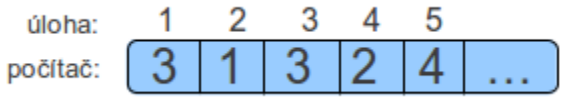
Na popísanie stavu častice z matematicko-fyzikálneho hľadiska teda postačuje jej pozícia a rýchlosť, ktorá má danú veľkosť a smer. Predpokladáme, že častica nemá informáciu o tom, kam smeruje, ale má možnosť vyhodnotiť svoju aktuálnu pozíciu a určiť, ktorým smerom sa bude uberať. Aby toto rozhodnutie nebolo celkom náhodné ovplyvňuje ho niekoľko faktorov:

- váha zotrvačnosti (w),

- najlepšia známa pozícia častice (*pBest*),
- najlepšia známa pozícia v rámci celého roja (*gBest*).

V metafore vtáčieho kŕdľa zodpovedá pozícia *gBest* tomu jedincovi, ktorý je na čele a určuje let všetkých jeho druhov.

Ako aplikovať model roja častíc na plánovanie úloh? Pomerne jednoducho. Našou úlohou je rozmiestniť daný počet úloh na dostupné výpočtové prostriedky gridu. O každej úlohe vieme povedať akú má výpočtovú, alebo časovú náročnosť a o každom výpočtovom zdroji poznáme základné informácie týkajúce sa jeho výkonu. Usporiadanie úloh tak, aby sme každú úlohu pridelili práve jednému zdroju budeme nazývať rozvrh. Rozvrh vieme znázorniť pomocou jednorozmerného poľa tak že i-ty prvok poľa predstavuje i-tu úlohu a jeho hodnota určuje zdroj na ktorom sa daná úloha vykoná.



Obrázok č. 12: Príklad častice, ktorá reprezentuje rozvrh.

Jedna častica reprezentuje jeden takýto rozvrh (obrázok č. 12). Premiestnením tejto častice na inú pozíciu v tomto priestore teda získame iný rozvrh, ktorý môžeme s predchádzajúcim porovnať a určiť, ktorý je lepší. Na určenie relatívnej hodnoty rozvrhu použijeme jednoduchú funkciu, pomocou ktorej dokážeme určiť čas dokončenia poslednej úlohy - makespan. Hodnota tejto funkcie sa v literatúre označuje ako fitness. Fitness hodnotu konkrétnej častice určíme podľa vzorca nasledovne:

$$F = MAX\left(\sum_{j=1}^{n_1} p_{j1}, \sum_{j=1}^{n_2} p_{j2}, \ldots, \sum_{j=1}^{n_m} p_{jm}\right)$$

$$n_1 + n_2 + \ldots + n_m = n$$

n – počet všetkých úloh

m - počet počítačov

n_i – počet úloh naplánovaných na počítač i

p_{ji} – čas behu úlohy j na počítači i. Je to čas od jej začatia po ukončenie.

C_{ij} – čas dokončenia úlohy j na počítači i.

$$\sum_{j=1}^{n_i} p_{ji} \approx MAX\left(C_{i1}, C_{i2}, \ldots, C_{i_j}\right)$$

$\sum_{j=1}^{n_i} p_{ji}$ je súčet časov všetkých úloh na počítači *i*. Zodpovedá zároveň času dokončenia poslednej úlohy na počítači *i,* ktorý ubehol od spustenia prvej úlohy.

Postupne prechádzame rozvrh a pre každý zdroj spočítame jeho výpočtový čas tak, že spočítame dĺžku všetkých úloh (v miliónoch inštrukcií z angl. MI – Millions of Instructions) a vydelíme ju celkovým výkonom daného počítača (v miliónoch inštrukcií za sekundu z angl. MIPS – Millions of Instructions Per Second). Výsledkom je čas, kedy sa na danom počítači skončí vykonávať posledná úloha. Zistíme na ktorom zdroji je táto hodnota najväčšia a toto číslo bude určovať našu fitness. Takto teda zistíme celkový výpočtový čas a našou primárnou snahou bude túto funkciu minimalizovať.

3.1.1 Optimalizácia kritéria pre plánovanie

Okrem vlastnej aktuálnej pozície si každá častica ukladá informáciu o svojej dosiaľ najlepšej dosiahnutej pozícii (*pBest*) a informáciu o najlepšej pozícii v rámci celého roja (*gBest*). Na základe týchto informácií môže upraviť svoju pozíciu a overiť, či došlo k zlepšeniu fitness, alebo nie. Ak áno nová pozícia sa uloží do *pBest.* Po úprave všetkých častíc sa porovná, či nedošlo k zlepšeniu *gBest*. Ak áno uloží sa nová hodnota *gBest*.

Základné vstupné parametre algoritmu sú:

- N – počet úloh, rozmer poľa,
- R – počet dostupných zdrojov,
- S – veľkosť roja (počet častíc v roji),
- I – počet iterácií do skončenia výpočtu,
- f() – funkcia, ktorej vstupom je pole (rozvrh) a výsledkom je fitness,
- c1,c2 – parametre určené programátorom. Určujú váhu *pBest* a *gBest.*

Celý algoritmus prebieha nasledovne:

1. Vytvor S častíc a inicializuj ich tak, aby boli rovnomerne rozmiestnené po celom priestore možných riešení problému. To znamená pre každú časticu treba vytvoriť pole N prvkov a do každého prvku priradiť náhodnú hodnotu z intervalu <1, R>.
2. Urči najlepšiu pozíciu častice ako aktuálnu pozíciu.

3. Ak f(*pBest*) < f(*gBest*) urči *pBest* ako najlepšiu pozíciu roja.
4. Inicializuj rýchlosť častice. Vytvor pole N prvkov, ktorých hodnoty sú náhodne zvolené z intervalu <-R, R>.
5. Opakuj pre c z intervalu <0..I): Pre každú časticu patriacu do roja vykonaj nasledovné:
   - zvoľ náhodné čísla r1,r2 z intervalu <0, 1> a urči váhu zotrvačnosti

     $$w = R - (\frac{c}{I}) * (R - 1)$$

   - uprav rýchlosť častice takto:

     $$v[i] = w * v[i] + c1 * r1 * (pBest[i] - x[i]) + c2 * r2 * (gBest[i] - x[i])$$

   - uprav pozíciu častice nasledovne:

     $$x[i] = x[i] + v[i]$$

   - vypočítaj fitness a ak je lepšia ako fitness *pBest* ulož aktuálnu pozíciu do *pBest*
   - ak je aktuálna fitness lepšia ako fitness *gBest* zaznamenaj novú pozíciu *gBest*

Po skončení algoritmu sa výsledný rozvrh nachádza v premennej *gBest* a je najlepším dosiahnutým riešením pre dané úlohy a prostriedky.

3.1.2 Dekrementácia váhy zotrvačnosti

Konvergencia k požadovanému riešeniu závisí od rýchlosti s akou častice menia svoju polohu. V prípade, že je táto rýchlosť príliš veľká, môže dôjsť k preskočeniu riešenia a tým pádom k pomalšej konvergencii, ba až k divergencii. Preto sme zaviedli tzv. faktor α, $\alpha \in \langle 0,1 \rangle$ ktorý určuje zmenu váhy zotrvačnosti po každej iterácii algoritmu.

$$w = w * \alpha$$

Na začiatku sa častice pohybujú v priestore možných riešení s veľkými zmenami a majú tak možnosť prebádať jeho veľkú časť. Postupne keď sa približujú k riešeniu ich váha zotrvačnosti je nízka a aj ich rýchlosť bude nízka čo umožňuje presnejšie nájdenie vyhovujúceho riešenia.

3.1.3 Rozvrhovanie paralelných úloh

Jednou z vecí, ktoré treba brať do úvahy je to, že používateľská aplikácia môže pre svoj beh požadovať viacero výpočtových zdrojov. V takom prípade je program je tvorený

niekoľkými podúlohami, pričom každá z nich sa vykoná na jednom stroji. Úlohou plánovača nie je len nájsť jeden stroj, na ktorom sa vykoná jedna úloha, ale nájsť množinu strojov, ktoré budú schopné vykonať všetky podúlohy daného programu. Navyše musí tieto rozhodnutia urobiť čo najefektívnejšie. Aby bol plánovač schopný takto pracovať je potrebné navrhnúť štruktúru, ktorá bude reprezentovať rozvrh, ktorý berie do úvahy jednotlivé podúlohy a ich umiestnenie na požadované zdroje systému. Príklad rozvrhu je znázornený na obrázku č. 13.

Obrázok č. 13: Rozvrh pre paralelné úlohy vyžadujúce viac výpočtových uzlov.

3.2 Horolezecký algoritmus

Horolezecký algoritmus (HC) je pomerne rýchly algoritmus, ktorý pracuje s počiatočným riešením a postupnými krokmi sa ho snaží vylepšiť. V každom kroku preskúma okolité riešenia a pokiaľ sú niektoré z nich lepšie (v zmysle optimalizačnej funkcie) tak z nich vyberie to, ktorého zlepšenie je v porovnaní so súčasným riešením najlepšie. Okolité riešenia pre problém plánovania sa vytvárajú zámenou dvoch prvkov v poli, ktoré reprezentuje rozvrh. Existujú aj rôzne iné verzie tohto algoritmu (výber prvého lepšieho riešenia, výber najlepšieho riešenia, algoritmus s náhodným reštartom a pod.) Známou nevýhodou algoritmu je to, že môže uviaznuť v lokálnom extréme hľadanej funkcie. Preto je vhodné pridať do algoritmu nejakú náhodnosť (stochastický horolezecký algoritmus). Ukazuje sa, že v prípade hľadania riešení NP - úplných problémov je efektívnym nástrojom na dosiahnutie výsledku, ktorý je blízky k optimálnemu riešeniu [30].

3.2.1 Horolezecký algoritmus v diskrétnom priestore

Opakom spojitého priestoru, v ktorom medzi dvomi bodmi existuje nekonečne veľa ďalších bodov je diskrétny priestor. V našej práci využívame horolezecký algoritmus, ktorý pracuje v diskrétnom priestore. Každý bod priestoru predstavuje jedno riešenie problému. Algoritmus začína v štartovacom bode, ktorý je zvolený náhodne a postupne preskúma všetky susedné body. Pokiaľ nájde medzi susednými bodmi bod s lepšou hodnotou začne

pracovať s týmto bodom. Pokiaľ sa taký nenájde algoritmus vráti ako výsledok aktuálny_bod.

Pseudo-kód horolezeckého algoritmu

```
aktuálny_bod := štartovací_bod;
opakuj pokiaľ aktuálny_bod != NULL {
  L := SUSEDNÉ_BODY(aktuálny_bod);
  hodnota := -INF;
  nasledujúci_bod := NULL;
  pre všetky x v L {
    if (EVAL(x) > hodnota) {
      nasledujúci_bod := x;
      hodnota = EVAL(x);
    }
    if (hodnota <= EVAL(aktuálny_bod)) {
      //neexistujú lepšie riešenia  vráť aktuálny bod
      return aktuálny_bod;
    }
  }
  aktuálny_bod := nasledujúci_bod;
}
```

4 Návrh hybridného plánovacieho algoritmu PSO + HC

Rozhodli sme sa implementovať hybridný algoritmus, ktorý primárne využíva algoritmus PSO pre usporiadanie úloh a horolezecký algoritmus usporiadaným úlohám na základe počtu ich podúloh priraďuje jednotlivé zdroje. Pre nájdenie lokálneho optimálneho priradenia podúloh na jednotlivé zdroje využíva výhody horolezeckého algoritmu.

Použitím algoritmu PSO pre nájdenie vhodného usporiadania úloh vzniká ďalší problém. Keďže jedna častica reprezentuje práve jedno usporiadanie úloh ide o permutáciu a to znamená, že je potrebné zabezpečiť, aby sa prvky v rámci jednej častice neopakovali. Jedným zo spôsobov ako zabezpečiť aby zmenou niektorej súradnice vznikla opäť platná permutácia rozvrhu je pri úprave jednej súradnice nemeniť jej hodnotu na základe rýchlosti, ale určiť inú súradnicu a ich hodnoty vymeniť [31]. Často sa používa tzv. SPV (Smallest Position Value) pravidlo, ktoré určuje špeciálny postup ako sa dá z vektora súradníc častice v n-rozmernom priestore vytvoriť permutácia predstavujúca rozvrh [33]. Jednotlivé prvky častice sa usporiadajú podľa ich hodnoty vzostupne a podľa tohto zoradenia sa zmení aj poradie úloh. Príklad vytvorenia rozvrhu z častice ukazuje tabuľka č 3. Z častice P = [-0.29, 1.25, 0.46, -1.67, 2.00, 0.13] sa vytvorí nasledovné poradie úloh: S = [J_4, J_1, J_6, J_3, J_2, J_5]. Najmenšiu hodnotu má 4. súradnica preto prvá v rozvrhu bude úloha č. 4. Druhá najmenšia je súradnica 1, preto v rozvrhu nasleduje úloha č. 1 atď. až sa takýmto spôsobom vytvorí celý rozvrh.

Tabuľka č. 1: Príklad vytvorenia rozvrhu z častice

	1	**2**	**3**	**4**	**5**	**6**
x_j	-0.29	1.25	0.46	**-1.67**	2	0.13
s_j	**4**	1	6	3	2	5

Po úprave častice na rozvrh je nutné každej úlohe priradiť dostatočný počet výpočtových uzlov tak, aby sa mohli úspešne vykonať všetky jej podúlohy. V našom algoritme používame štruktúru, ktorú označujeme ako požiadavka na uzol – *node_request*. Táto štruktúra obsahuje všetky informácie o požiadavkách danej úlohy na jej výpočtové prostriedky. Pokiaľ úloha obsahuje *m* podúloh bude obsahovať aj *m* požiadaviek na uzol. Druhým krokom algoritmu je teda pridelenie každej požiadavky jednému konkrétnemu počítaču. To je možné dosiahnuť niekoľkými spôsobmi:

- náhodným výberom
- iteratívne (postupným priraďovaním počítačov spôsobom Round Robin)
- použitím zložitejšieho optimalizačného algoritmu pre nájdenie lokálneho optimálneho riešenia.

Každá z týchto metód má svoje výhody a nevýhody, ktoré sa odzrkadľujú hlavne na celkovom čase generovania rozvrhu a fitness hodnote rozvrhu. V našom algoritme sme sa rozhodli použiť horolezecký algoritmus pre nájdenie optimálneho priradenia podúloh na dostupné stroje. Takto získame rozvrh, ktorému môžeme pomocou hodnotiacej funkcie priradiť fitness. V našom prípade budeme opäť používať ako hodnotiace kritérium čas potrebný na dokončenie všetkých úloh – makespan.

4.1.1 Pseudo-kód navrhovaného algoritmu PSO+HC

Po počiatočnej inicializácii parametrov algoritmu sa pre všetky častice roja opakuje výpočet ich fitness, overenie či nedošlo k zlepšeniu *pBest* a *gBest,* následne sa upraví váha zotrvačnosti, vypočíta rýchlosť a pozícia a zistí sa rozvrh. Pomocou horolezeckého algoritmu sa nájde lokálne optimum – usporiadanie podúloh v rozvrhu.

Pseudo-kód hybridného algoritmu PSO+HC

```
inicializuj parametre algoritmu
náhodne inicializuj všetky častice roja
pre všetky častice i opakuj {
  fitness_i := ohodnoť_časticu(i)
}
pokiaľ nie je splnená ukončovacia podmienka opakuj {
  pre všetky častice i opakuj {
    if (fitness_i < pBest_i) pBest_i := fitness_i
    if (pBest_i < gBest) gBest := pBest_i
    w_i := w * α
    uprav_rýchlosť(i)
    uprav_pozíciu(i)
    rozvrh := nájdi_rozvrh_pomocou_SPV(i)
    nájdi_lokálne_optimum_pomocou_HC(rozvrh, i)
    fitness_i := ohodnoť_časticu(i)
  }
}
```

Ukončovacou podmienkou v našom prípade je počet generácií roja počas ktorých nedôjde k vylepšeniu riešenia *gBest*. V prípade, že by výpočet trval príliš dlho zároveň so spomenutou podmienkou je ohraničený aj časovým intervalom. Keď uplynie vopred definovaný čas, považuje sa za výsledok aktuálne najlepšie riešenie a vytváranie rozvrhu sa ukončí.

4.1.2 Nájdenie lokálneho optima pomocou HC

Za účelom dosiahnutia najlepšieho riešenia plánovač udržuje informácie o predpokladanom čase ukončenia výpočtu na každom uzle. Na začiatku hľadania lokálneho optimálneho riešenia je tento čas pre každý uzol nulový. Po priradení ďalšej podúlohy na uzol sa tento čas zvýši o predpokladaný čas výpočtu p_j úlohy j. Po priradení všetkých podúloh je možné zistiť maximálny čas dokončenia poslednej úlohy v systéme ako

$$makespan = MAX\left(tc_1, tc_2, \ldots, tc_n\right)$$

kde tc_i je čas dokončenia výpočtu čiže čas ukončenia poslednej úlohy na uzle i.

$$tc_i = MAX\left(C_{ij}\right)$$

Týmto získame ohodnotenie potenciálneho rozvrhu, ktorý vznikol z častice roja. Optimalizácia horolezeckým algoritmom spočíva v pokuse vylepšiť náhodné počiatočné priradenie požiadaviek uzlom postupným prehľadávaním susedných riešení, ich ohodnotením a nájdením toho, ktoré je najvýhodnejšie. Toto sa opakuje pokiaľ sa riešenie neprestane vylepšovať. Prehľadávanie susedných riešení spočíva jednoducho v postupnom priraďovaní podúloh všetkým uzlom, ktorým ešte žiadna podúloha danej úlohy nebola priradená.

Pseudo-kód navrhovaného hľadania lokálneho optima

```
náhodne inicializuj všetky podúlohy každej úlohy
opakuj, pokiaľ koniec != TRUE {
  koniec := TRUE
  opakuj pre j z množiny úloh {
    opakuj pre k z množiny počítačov a i z množiny požiadaviek
úlohy j {
      ak žiadna podúloha ešte nebola priradená k a zároveň k spĺňa
požiadavky i {
        premiestni i na počítač k
        ohodnoť toto riešenie
        ak sa našlo zlepšenie {
          zapamätaj si nové riešenie
          koniec := FALSE
        }
      }
    }
  }
}
```

Ukončovacou podmienkou v našom prípade je fakt, že riešenie sa prestáva vylepšovať. To znamená, že pokiaľ sa nájde počas algoritmu zlepšenie, kontrolná premenná **koniec** sa nastaví na FALSE, ináč ostane TRUE a algoritmus sa pri najbližšej kontrole podmienky ukončí,

4.2 Analýza časovej a pamäťovej náročnosti

Na rozdiel od väčšiny deterministických algoritmov nie je jednoduché určiť časovú náročnosť stochastických algoritmov, ktoré zahŕňajú element náhody pretože práve správny náhodný výber niektorého prvku môže viesť k rýchlemu nájdeniu optimálneho, alebo sub-optimálneho riešenia, čo znamená ukončenie výpočtu. Naopak viacnásobné náhodné nesprávne zvolenie konštanty môže neúmerne predĺžiť výpočet bez získania výsledku. Tento fakt sa mimochodom odzrkadľuje aj na rôznych výsledných výstupoch, ktoré je možné dosiahnuť pri tých istých vstupných údajoch. Napriek tomu sa pokúsime odhadnúť časovú zložitosť navrhovaného algoritmu.

Základom je výpočet fitness hodnoty, ktorý prebieha v každej iterácii toľkokrát, koľko častíc obsahuje roj. Dĺžka výpočtu fitness závisí od počtu úloh N a počtu dostupných zdrojov R. Každá úloha môže obsahovať maximálne R podúloh, inak by nebolo povolené jej odoslanie do systému. Počas hľadania lokálneho optima sa algoritmus HC pokúša priradiť postupne všetky podúlohy všetkým voľným zdrojom. Maximálny počet krokov je teda N*R*R. Časová zložitosť výpočtu fitness je $O(NR^2)$.

Tento krok sa musí vykonať pre každú časticu roja. V jednej iterácii teda prebehne S-krát. Časová náročnosť jednej iterácie je $O(SNR^2)$.

Počet iterácií závisí od schopnosti roja priblížiť sa a zotrvať v blízkosti vyhovujúceho riešenia. Konvergenciu roja ovplyvňuje vyššie zmienený faktor α. Pri jeho vhodnom nastavení môžeme bezpečne zaručiť, že algoritmus dospeje k riešeniu v čo najmenšom počte iterácií. Ich počet ale nie je možné presne určiť pretože nevieme ako sa bude riešenie vylepšovať.

Pamäťová náročnosť lineárne závisí od počtu úloh, ktoré sú reprezentované štruktúrou *job_info*, počtu všetkých uzlov – *node_info* a počtu podúloh, ktoré jednotlivé úlohy obsahujú. Ostatné premenné a štruktúry sú konštante pamäťovo náročné. Horný odhad pamäťovej náročnosti je teda $P(NR^2)$.

5 Overenie funkčnosti a testovanie algoritmu

Funkčnosť navrhnutého algoritmu sme otestovali v dvoch rôznych úrovniach gridového plánovania. Prvým prostriedkom pre overenie funkčnosti a vhodnosti PSO plánovacieho algoritmu je vytvorenie simulácie gridového prostredia a implementovanie plánovacieho algoritmu v tomto prostredí. Na základe výsledkov simulácie sme sa rozhodli implementovať algoritmus aj v prostredí lokálneho plánovača a overiť jeho funkčnosť pri plánovaní reálnych úloh v počítačovom klastri.

5.1 Simulátor gridového prostredia

Pre analýzu PSO algoritmu sme sa rozhodli požiť nástroj GridSim – gridový simulačný nástroj pre modelovanie zdrojov a plánovanie aplikácií pre paralelné a distribuované systémy. Nástroj GridSim umožňuje modelovanie a simuláciu entít ako sú používatelia, aplikácie, systémové prostriedky, plánovače (brokery), návrh a vyhodnocovanie plánovacích algoritmov. Prostriedkom môže byť jeden procesor, alebo multiprocesor so zdieľanou, alebo distribuovanou pamäťou, ktorý je spravovaný plánovačom s politikou zdieľania času, alebo priestoru. Broker využíva plánovacie algoritmy pre mapovanie úloh na zdroje so snahou optimalizovať systém, alebo užívateľské ciele s ohľadom na ich požiadavky. GridSim je naprogramovaný v jazyku Java a je voľne dostupný. Súčasná verzia je 5.2 [32].

Jednu výpočtovú úlohu reprezentuje trieda *Gridlet*, ktorá obsahuje niekoľko atribútov popisujúcich vlastnosti úlohy. Najpodstatnejšie pre našu analýzu je atribút *length* – dĺžka výpočtu v MIPS (z angl. Millions of Instruction Per Second). Predpokladáme teda, že poznáme náročnosť úlohy a vieme ju vyjadriť v rovnakých jednotkách ako výkon počítačov.

Zdroje gridu sú reprezentované triedou *GridResource*, ktorá popisuje vlastnosti ako sú architektúra, operačný systém, záťaž systému, prenosová rýchlosť a pod. Ďalej obsahuje objekt *ResourceCharacteristic*, ktorého súčasťou je zoznam všetkých počítačov v klastri, počet ich procesorov a ich výkon. Aby sme mohli porovnávať výkon všetkých počítačov zvolili sme jednotnú mierku, ktorá vyjadruje počet operácií s plávajúcou desatinnou čiarkou za sekundu. Hodnotenie každého procesora je založené na údajoch získaných pomocou benchmarku Geekbench [33].

Cieľom našej simulácie bolo porovnať kvalitu riešení získaných pomocou samotného algoritmu PSO s riešeniami, získanými na základe bežne používaných algoritmov – horolezecký algoritmus (HC) a Round Robin (RR). Pre porovnanie výkonu PSO plánovacieho algoritmu sme použili jeho dve verzie odlišujúce sa vstupnými parametrami.

Verzia PSO-1 pracuje s 10 časticami a počet iterácii je nastavený na 300. Verzia PSO-2 sa líši v počte častíc, ktorých je až 100 a počet iterácií je 500.

Algoritmus Round Robin (RR) s ktorým sme porovnávali navrhovaný plánovací algoritmus PSO predstavuje najjednoduchší spôsob plánovania a funguje nasledovne:

Úlohy spracováva v poradí v akom prichádzajú do systému (alebo v poradí v akom sú usporiadané v rade) a postupne bez dôkladnejšej analýzy im priraďuje jednotlivé zdroje, ktoré spĺňajú minimálne požiadavky. Keď má každý stroj priradenú úlohy začne priraďovať zdroje opäť od začiatku.

V našich testoch sme na všetky testovacie úlohy sme aplikovali algoritmus PSO, HC a RR a porovnali celkový simulovaný čas po použití vygenerovaného rozvrhu. Zároveň sme zaznamenávali čas potrebný na vygenerovanie rozvrhu.

5.2 Implementácia a testovanie algoritmu v počítačovom klastri

Okrem simulácie a modelovania gridového prostredia pomocou nástroja GridSim sme sa rozhodli implementovať algoritmus PSO s miernymi zmenami a doplnením o vyhľadávanie lokálneho optima pomocou horolezeckého algoritmu a otestovať jeho funkčnosť na počítačovom klastri univerzity Mateja Bela v Banskej Bystrici.

Lokálnym správcom prostriedkov klastra je manažér Torque (pôvodne OpenPBS). Je to manažér zdrojov, ktorý obsahuje základný plánovač *pbs_sched*, ale zároveň je ďalej rozšíriteľný externými plánovačmi. Plánovač *pbs_sched* používa jednoduchý algoritmus FIFO (z angl. First In First Out), a jeho plánovacie politiky je možné nastaviť pomocou konfiguračného súboru, ktorý sa načítava vždy pri spustení plánovača. Základné vlastnosti, ktoré je možné konfigurovať sú:

- použitie viacerých radov,
- striktný FIFO algoritmus,
- vyrovnané používanie zdrojov užívateľmi,
- vyrovnaná záťaž uzlov,

- spôsob zotriedenia radu.

Používanie tohto plánovača je postačujúce pre jednoduché počítačové klastre s menším počtom uzlov, ktoré nevyžadujú náročnejšie plánovanie a predikovanie správania sa systému v budúcnosti. Ide hlavne o predpokladané využitie zdrojov systému a ich možné priraďovanie čakajúcim úlohám.

Plánovač, v ktorom algoritmus implementujeme pracuje nasledovne: Základné funkcie, ktoré sú volané hlavným procesom plánovača sú *schedinit()* a *schedule()*. Prvá slúži na inicializáciu algoritmu, vytvoria a naplnia sa v nej statické premenné, ktoré sa používajú počas behu algoritmu. Funkcia *schedule()* spracováva nasledovné príkazy, ktoré získava od manažéra zdrojov:

- SCH_SCHEDULE_FIRST – prvé spustenie plánovača pri zaradení prvej úlohy do radu
- SCH_SCHEDULE_NEW – zaradenie novej úlohy do radu pripravených úloh
- SCH_SCHEDULE_TERM – ukončenie niektorej úlohy
- SCH_SCHEDULE_CMD – príkaz na vykonanie plánovacej iterácie
- SCH_SCHEDULE_TIME – plánovacia iterácia sa vykonáva vo vopred definovanom intervale
- SCH_QUIT – ukončenie činnosti plánovača

Pri zaregistrovaní novej úlohy sa vygeneruje rozvrh pomocou uvedeného hybridného algoritmu PSO + HC. Zdrojový kód funkcie, ktorá generuje rozvrh je uvedený v prílohe A. Tento rozvrh ostáva platný až pokiaľ nie je zaradená do radu nová úloha, alebo plánovač nedostane príkaz SCH_SCHEDULE_TIME. V tom prípade sa vygeneruje nový rozvrh. Po vygenerovaní rozvrhu sa plánovač pokúsi spustiť čo najviac úloh podľa poradia v tomto rozvrhu na pridelené výpočtové uzly. To isté sa vykoná aj v prípade ukončenia výpočtu niektorej úlohy. V tomto prípade sa nevygeneruje nový rozvrh ale plánovač spúšťa ďalšie úlohy z aktuálne platného rozvrhu.

5.2.1 Proces a automatizácia testovania v klastri

Pre testovanie sme vytvorili niekoľko skupín testovacích úloh, ktoré sme postupne spúšťali na 2 klastroch. Všetky tieto skupiny úloh sme testovali s predvoleným plánovačom distribuovaným spolu s manažérom zdrojov – *pbs_sched,* ktorý štandardne používa

algoritmus FIFO. Rovnakou sadou úloh sme potom otestovali plánovač používajúci optimalizáciu s hybridným algoritmom PSO+HC.

Odoslanie úlohy do systému Torque sa realizuje pomocou príkazu *qsub*. Jeho vstupom je buď cesta k programu, ktorý sa má vykonať a jeho parametre, alebo špeciálny odosielací skript v ktorom je možné definovať ďalšie požiadavky pre odosielanú úlohu. Aby sme nemuseli odosielať všetky úlohy do systému ručne vytvorili sme pre každú sériu testov v testovacej skupine skript v jazyku BASH, ktorý odošle do systému niekoľko úloh s rôznymi požiadavkami. Príklad odosielacieho súboru je v prílohe B. Potom spustí plánovač pbs_sched a odošle do systému ešte jednu spúšťaciu úlohu (*trigger*), ktorá spustí v plánovači príkaz SCH_SCHEDULE_FIRST, čím sa začne vytvárať rozvrh. V prípade testovania dynamického plánovania sa tieto úlohy neodošlú naraz, ale v istom časovom intervale.

Po odoslaní všetkých úloh testovací skript zisťuje koľko úloh bolo úspešne vykonaných pomocou príkazu *qstat* a po ukončení poslednej úlohy špeciálnym príkazom zozbiera informácie o čase odoslania, čase začatia a čase ukončenia výpočtu úlohy. Podobný skript tieto výsledky upraví tak, aby bola možná ich jednoduchá interpretácia. Skript, ktorý zabezpečí spustenie a získanie výsledkov jednej testovacej série je v prílohe C.

6 Použité testovacie údaje

Prvým cieľom našich experimentov bolo pomocou simulácie analyzovať a otestovať vhodnosť algoritmu pre reálny systém. Preto sme sa rozhodli overiť jeho funkčnosť na modele skutočného gridového prostredia. Údaje pre simuláciu sú prevzaté z webovej stránky českého projektu MetaCentrum, ktorého cieľom je podpora rozvoja národnej gridovej iniciatívy (NGI), ktorá prepája výskumné tímy a inštitúcie v rôznych regiónoch ČR a umožňuje ich napojenie do medzinárodného výskumného prostredia. Je súčasťou celoeurópskeho projektu European Grid Iniciative (EGI).

Druhou časťou výskumu bolo otestovanie plánovača v prostredí vysokovýkonného klastra pre ktorý sme si sami vopred pripravili úlohy s rôznou náročnosťou. Úlohy sme najprv spustili na testovaných počítačoch každú zvlášť a zaznamenali sme ich čas dokončenia. Táto informácia nám ďalej slúžila ako predpokladaný čas dokončenia úlohy v paralelnom prostredí. Pripravené úlohy sme následne spúšťali a merali ich časy spustenia a dokončenia.

6.1 Testovacie údaje pre simuláciu gridového plánovania

Vybavenie gridu projektu MetaCentrum zahŕňa širokú škálu hardvéru, ktorá pozostáva z približne 20 strojov dostupných pre používateľov, niekoľkých súborových serverov a záložného systému. K dispozícii sú multiprocesorové SMP stroje so zdieľanou pamäťou založené na architektúre MIPS, multipočítačové klastre používajúce 32 a 64-bitové procesory firiem Intel (Xeon) a AMD (Opteron) a osobné počítače so 16, alebo 32 jadrami [34]. Na základe údajov dostupných na webovej stránke projektu MetaCentrum o názve gridových uzlov, počte počítačov a procesorov v každom uzle a ohodnotení jednotlivých strojov pomocou výsledkov benchmarku Geekbench sme určili výkon všetkých strojov, ktoré sme sa rozhodli použiť v simulácii gridového prostredia. Použité údaje o uzloch v simulovanom gridovom prostredí podrobnejšie opisuje tabuľka č. 2.

Tabuľka č. 2: Použité údaje o výkone gridu

Názov	Počítače	Procesory	Výkon (MIPS)	Názov	Počítače	Procesory	Výkon (MIPS)
Tarkil	28	2	17 950	Konos	4	2	1 922
Eru	2	8	15 398	Hermes	12	1	5 132
Manwe	7	8	8 556	Ajax	1	2	17 950
Aule	1	8	3 154	Alea	12	2	11 092
Skirit	1	2	1 648	Wood	1	1	1 310
Skirit	32	2	17 050	Quark	5	2	1 648
Skirit	35	2	6 180	Quark	8	2	6 180
Skirit	1	2	10 650	Quark	3	2	17 950
Loslab	6	2	5 616	Acharon	1	16	5 670
Nympha	20	2	13 746	Glamdring	1	2	1 698
Hydra	1	2	8 570	Perian	8	2	1 698
Hydra	10	1	11 134	Perian	10	2	14 598
Konos	10	2	35 080	Perian	10	2	10 168
Konos	24	2	4 733				

Celkovo sme počas simulácie vykonali 3 série testov s rovnakým počtom úloh. Dĺžka jednotlivých úloh bola generovaná náhodne a naším cieľom bolo overiť správanie plánovača pri rovnakom počte úloh s rôznymi veľkosťami.

Prvá testovacia séria je zameraná na plánovanie úloh s náhodnou veľkosťou a náhodnou variabilitou dĺžky úloh. Rozsah vstupného súboru je 100 úloh. Dĺžka úloh je vygenerovaná v rozmedzí od 10^5 do 10^6 MI.

Druhá testovacia séria obsahuje 100 úloh s nízkou variabilitou (diverzitou) dĺžky úloh. To znamená, že všetky úlohy sú približne rovnako veľké a ich dĺžka je vygenerovaná náhodne z intervalu $d \in \langle 1*10^5, 2*10^5 \rangle$ MI.

Posledná séria obsahuje takisto 100 úloh s veľkou diverzitou, čo znamená, že ich veľkosti sú značne rozdielne. Existuje množstvo malých úloh a približne rovnaký počet veľkých úloh. Malé úlohy majú dĺžku len niekoľko desiatok MI, veľké úlohy až niekoľko tisíc MI. Tabuľka č. 3 znázorňuje prehľad použitých testovacích úloh pre simuláciu plánovania pomocou algoritmu PSO v gridovom prostredí.

Tabuľka č. 3: Použité testovacie údaje

Séria	Počet úloh	veľkosť úloh (MI)	Diverzita
Test 1	100	100000 – 1000000	náhodne
Test 2	100	100000 – 200 000	malá diverzita
Test 3	100	1 – 100000	veľká diverzita

6.2 Testovacie údaje pre výpočtový klaster

Navrhnutý a implementovaný algoritmus plánovania mapovania úloh v paralelnom systéme sme otestovali na dvoch nezávislých vysokovýkonných paralelných klastroch. Prvý klaster pozostáva zo 1 servera a 3 výpočtových uzlov. Ide o 4 virtuálne počítače s architektúrou x86_64, operačným systémom Ubuntu Server, 512MB RAM, 6GB HDD. Na každom pracovnom uzle je nainštalovaná implementácia MPI s podporou manažéra zdrojov – Torque. Tento klaster budeme označovať *cluster1*.

Druhý testovaný klaster – *cluster2* – je súčasťou Slovenskej infraštruktúry pre vysokovýkonné počítanie a nachádza sa na Univerzite Mateja Bela v Banskej Bystrici. Pozostáva z 24 výpočtových uzlov a celkovo ponúka k dispozícii 288 výpočtových jadier s celkovým výkonom viac ako 3 TFLOP. Každý výpočtový uzol je osadený dvomi procesormi Intel Xeon X5670, 48 GB RAM a dvomi pevnými diskami, z ktorých je vytvorené pole RAID 0 s celkovou kapacitou 1,2TB [35]. Tabuľka č. 4 popisuje prehľad vlastností klastrov na ktorých sme algoritmus testovali.

Tabuľka č. 4: Prehľad testovacích klastrov

	Počet uzlov	Architektúra	Operačný systém	Operačná pamäť	Pevný disk
cluster1	4	Intel x86_64	Linux, Debian 4.0	500MB / uzol	6GB / uzol
cluster2	24	Intel x86_64	Scientific Linux	48 GB / uzol	1,2 TB / uzol

Na oboch klastroch sme spustili niekoľko skupín testovacích sérií náhodne usporiadaných úloh s rôznymi dĺžkami času vykonávania a rôznymi požiadavkami na počet uzlov. Testovacie úlohy sme vyberali spomedzi niekoľkých vopred pripravených výpočtovo náročných programov, ktorých výpočtový čas sme vopred odmerali. Prehľad použitých programov, ich dĺžky výpočtu a popisy predstavuje tabuľka č. 5. Okrem náhodného výberu z tejto skupiny úloh sme v testovacích sériách použili úlohy s náhodnou dĺžkou trvania, ktoré sme vygenerovali pomocou príkazu *sleep()*.

Tabuľka č. 5: Testovacie úlohy a ich vlastnosti

Program a parametre	Čas – cluster1	Čas – cluster2	Popis
./ackerman	20s	18s	Výpočet Ackermanovej funkcie
./binary 10000000	3s	3s	Binárne vyhľadávanie
./binary 100000000	33s	30s	Binárne triedenie
./bsort 100000	35s	37s	Binárne triedenie
./bsort 200000	2m 23s	2m 27s	Binárne triedenie
./bsort 300000	5m 43s	5m 32s	Binárne triedenie
./compress	3.3s	3s	Kompresia
./fft 24	16s	15s	Rýchla Fourierova transformácia
./fft 25	36s	32s	Rýchla Fourierova transformácia
./fft 26	1m 19s	1m 7s	Rýchla Fourierova transformácia
./fft 27	3m 16s	2m 21s	Rýchla Fourierova transformácia
./jacobi	39s	2s	Výpočet lineárnych rovníc Jacobiho metódou
./matmult 100	0.008s	0.012s	Násobenie matíc
./matmult 1000	12s	7s	Násobenie matíc
./matmult 2000	1m 43s	1m 8s	Násobenie matíc
./pi	32s	29s	Výpočet čísla PI
./pi2 100000000	3s	3s	Výpočet čísla PI 2. spôsob
./pi2 1000000000	31s	27s	Výpočet čísla PI 2. spôsob
./pi3 100000000	3s	3s	Výpočet čísla PI 3. spôsob
./pi3 1000000000	31s	32s	Výpočet čísla PI 3. spôsob
./pi3 100000000000	3m 22s	3m24s	Výpočet čísla PI 3. spôsob
./qsort 20000000	4s	4s	Qsort
./fibonacci 47	31s	34s	Výpočet fibonacciho postupnosti
./fibonacci 48	51s	56s	Výpočet fibonacciho postupnosti
./fibonacci 49	1m 24s	1m 31s	Výpočet fibonacciho postupnosti
./fibonacci 50	2m 17s	2m 28s	Výpočet fibonacciho postupnosti
./monte_carlo 1000000	10s	12s	Simulácia chemickej reakcie metódou Monte Carlo
./monte_carlo 2000000	21s	24s	Simulácia chemickej reakcie metódou Monte Carlo
./monte_carlo 3000000	32s	37s	Simulácia chemickej reakcie metódou Monte Carlo
./whetstone	51s	1m 58s	Benchmark pre hodnotenie výkonu procesora

Prvú skupinu úloh pre *cluster1* budeme ďalej označovať *suite1*. Spoločným znakom pre každú testovaciu sériu v tejto skupine bolo, že všetky úlohy boli zaradené do radu úloh ešte pred tým ako bol spustený plánovač. V prípade plánovača pracujúceho na princípe rozvrhu to znamená, že rozvrh bolo potrebné vygenerovať iba raz na začiatku a všetky úlohy sa spúšťali podľa tohto rozvrhu. Ide o statické, alebo tzv. dávkové plánovanie. Priemerná dĺžka testovacích úloh je 29 – 36s a počet úloh sa v každej testovacej sérii

zvyšuje od 11 až po 71. Cieľom je zistiť schopnosť plánovača vykonávať efektívne rozhodnutia pri zvyšujúcom sa počte úloh v systéme.

Druhou skupinu úloh pre *cluster1* s podobnými charakteristikami ako *suite1* (označujeme *suite2*) sme testovali schopnosť dynamického plánovania, teda schopnosť plánovača reagovať na pridávanie nových úloh do systému a správne a rýchle vygenerovanie optimálneho rozvrhu. Úlohy sme odosielali do systému v náhodných časových intervaloch (pre oba porovnávané algoritmy boli tieto intervaly rovnaké). Priemerná dĺžka úloha v jednotlivých testovacích sériách je 30 – 40s. Počet úloh sa postupne zvyšuje od 10 do 70.

Treťou skupinou úloh pre *cluster1* (*suite3*) sme testovali schopnosť plánovača priraďovať úlohy na zvlášť vyžiadané prostriedky. (Názov uzla bol špecifikovaný v príkaze odoslania úlohy.) Úlohou plánovača by malo byť sprístupniť vyžiadané prostriedky a ostatné úlohy pri ktorých nezáleží na tom, na ktorom uzle sa budú vykonávať prideliť na ostatné voľné počítače. Priemerný čas úloh v šiestich testovacích sériách je 18 až 20s a počet úloh postupne lineárne narastá od 11 až po 111.

Počet a vlastnosti úloh v každej testovacej sérii pre *cluster1* ukazuje tabuľka č. 6.

Tabuľka č. 6: Prehľad skupín testovacích úloh pre *cluster1*

Skupina	suite1							suite2							suite3					
Testovacia séria	1	2	3	4	5	6	7	1	2	3	4	5	6	7	1	2	3	4	5	6
Počet úloh	11	21	31	41	51	61	71	10	20	30	40	50	60	70	11	31	51	71	91	111
Priemerná dĺžka úlohy (s)	35	31	33	29	34	36	36	37	39	40	32	30	34	33	18	20	19	19	20	19

Podobné tri série úloh sme pripravili a testovali aj pre *cluster2*. Úlohy s náhodne vygenerovanou dĺžkou požadujú rôzny počet uzlov, ktorý je tiež určený náhodne v rozmedzí 2 až 12 uzlov. Reálne úlohy sú všetky vykonávané na jednom uzle.

Prvá skupina (*suite4*) je zameraná na testovanie plánovania dlhších úloh. Dĺžka úloh v 7 testovacích sériách sa pohybuje od 180 do 300s a počet úloh narastá od 20 až do 140. Všetky úlohy boli do systému odoslané v rovnakom čase.

Cieľom druhej skupiny testovacích úloh pre cluster2 bolo opäť otestovať dynamické plánovanie. Pozostáva zo 7 sérií testovacích úloh s dĺžkou 10 – 60s. Počet úloh sa v každej sérii zväčší o 10 oproti predchádzajúcej.

Tretia skupina úloh pre cluster2 (*suite6*) testuje schopnosť plánovača priradiť úlohy

na špecificky vyžiadané zdroje. Pri vygenerovaní odosielacieho skriptu úlohy sme pridali ako požiadavku úlohy náhodne vygenerovanú množinu uzlov na ktorých sa má úloha vykonať. Prehľad úloh, ich počet v jednotlivých testovacích sériách, minimálny a maximálny čas každej úlohy opisuje tabuľka č. 7.

Tabuľka č. 7: Prehľad skupín testovacích úloh pre *cluster2*

Skupina	**suite4**							**suite5**							**suite6**						
Séria	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Min. čas	180	180	180	180	180	180	180	10	10	10	10	10	10	10	30	30	30	30	30	30	30
Max. čas	300	300	300	300	300	300	300	60	60	60	60	60	60	60	60	60	60	60	60	60	60
Počet	20	40	60	80	100	120	140	10	20	30	40	50	60	70	5	10	15	20	25	30	35

7 Reprezentácia výsledkov testovania

Pri plánovaní a prideľovaní jednotlivých úloh na pracovné uzly sme sa v obidvoch experimentoch sústredili hlavne na celkový čas dokončenia všetkých úloh. Počas simulácie sme zaznamenávali aj čas vygenerovania rozvrhu pre danú skupinu testovacích úloh. Ide o dôležitý faktor hlavne pre dynamické plánovanie. Úlohou plánovača je prijímať rýchle rozhodnutia. Niekedy však dlhší čas potrebný na vytvorenie rozvrhu môže priniesť zlepšenie celkového výsledku. Je preto potrebné nájsť kompromis medzi obidvoma krajnými riešeniami.

Pri plánovaní úloh v klastri sme okrem zaznamenávania vývoja funkcie makespan merali aj čas vygenerovania rozvrhu a priemerné a maximálne časy dokončenia jednotlivých úloh. V plánovacích algoritmoch totiž niekedy nastáva situácia, že jedna úloha je odkladaná za účelom využitia zdrojov inou úlohou, ktorá lepšie spĺňa kritériá plánovača. Takáto situácia sa nazýva hladovanie. V našich výsledkoch sme sa zamerali aj na to, či niektoré úlohy nehladujú a na porovnanie časov čakania dosiahnutých pomocou PSO+HC v porovnaní s iným plánovačom.

7.1 Výsledky simulácie gridového plánovača

Na základe simulácie troch skupín úloh v modelovom gridovom prostredí sme analyzovali dva ukazovatele, ktoré môžu byť významné pri rozhodovaní o použití plánovacieho algoritmu. V prvom rade ide o čas, ktorý je potrebný na vytvorenie rozvrhu, podľa ktorého sa následne odosielajú úlohy jednotlivým uzlom gridového prostredia a v druhom rade je to celkový simulovaný čas dokončenia poslednej úlohy.

7.1.1 Generovanie rozvrhu

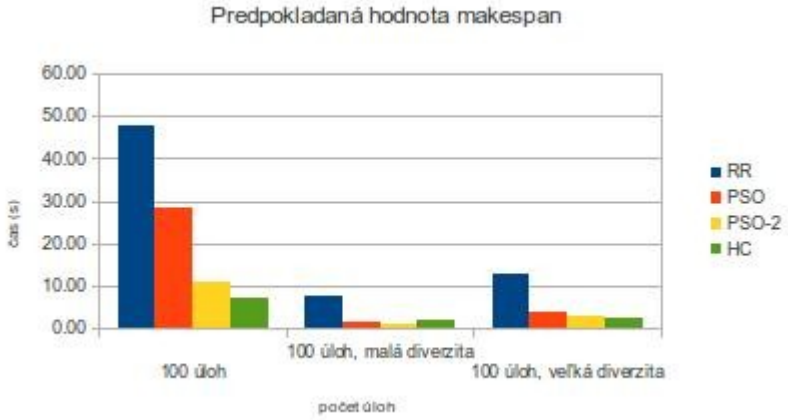
Vygenerovanie rozvrhu je operácia, ktorá značne vplýva na celkovú dobu výpočtu hlavne pri dynamickom plánovaní. Je však rovnako dôležitým faktorom statického plánovania. Najkratší čas potrebný na vygenerovanie rozvrhu je potrebný pre Round Robin (prakticky nulový), pretože počas tohto algoritmu nie je potrebné vykonať žiadne náročné výpočty v momente určenia cieľového uzla úlohy. Jediné čo je potrebné vykonať je porovnať či zvolený uzol spĺňa minimálne požiadavky úlohy. Veľmi krátky čas je potrebný aj pre algoritmus Hill Climbing, pretože jeho gradientová podstata zabezpečuje, že z množiny okolitých riešení sa vždy vyberie to najlepšie a tak sa dá rýchlo dospieť k

prijateľnému výsledku. Najdlhšie trvá vygenerovanie rozvrhu pre PSO-2 hlavne kvôli väčšiemu počtu iterácií v porovnaní s PSO-1. Priemerné časy generovania rozvrhu je možné vidieť na obrázku č. 14. Nulový čas je potrebný pre algoritmus RR, len 20-30ms trvá vytvorenie rozvrhu pomocou algoritmu HC pre všetky testovacie série. Algoritmus PSO-1 si s rozvrhom poradí približne za 200ms pre všetky testovacie série. Viac ako 1000ms trvá vytvorenie rozvrhu pomocou algoritmu PSO-2.

Obrázok č. 14: Čas vygenerovania rozvrhu pri rôznych typoch úloh.

7.1.2 Simulácia výpočtového času

Známou nevýhodou algoritmu HC pri veľkých vstupoch je, že výsledkom nemusí byť globálne, ale lokálne minimum, pričom algoritmus nemá možnosť zistiť o aký extrém ide. Preto sa môže stať, že pri veľkom vstupe sa jeho výsledok nelíši o mnoho viac od algoritmu RR. Pokiaľ sa počet úloh pohybuje okolo 100 dokáže algoritmus HC vytvoriť efektívnejší rozvrh ako by sme získali použitím obyčajného algoritmu RR. Lepšie výsledky v porovnaní s RR ukazuje aj použitie PSO, pričom verzia PSO-2 je ešte o niečo efektívnejšia. V prípade rovnako veľkých úloh získavame pomocou PSO lepší výsledok ako pomocou algoritmu HC. Počas simulácie sme zaznamenávali predpokladaný čas dokončenia úloh pre daný algoritmus. Po ukončení simulácie sme odmerali skutočný čas dokončenia všetkých úloh. Výsledné časy predpokladanej hodnoty makespan a celkového času simulácie je možné vidieť na obrázkoch č. 15 a 16.



Obrázok č. 15: Hodnota makespan určená po vytvorení rozvrhu

Obrázok č. 16: Čas potrebný na dokončenie všetkých simulovaných úloh.

Z obrázku č. 16 vyplýva, že pri veľkej diverzite vstupných úloh sú všetky tri algoritmy návrhu okrem RR takmer rovnocenné.

Je nutné uvedomiť si, že testy, ktoré sme vykonali napriek vysokej úrovni simulácie nezodpovedajú úplne skutočnosti. Simulované prostredie je takmer ideálne. Nedochádza tu k žiadnym výpadkom, lokálnej záťaži prostriedkov, komunikačným poruchám a podobne. Zároveň sme pre zjednodušenie predpokladali, že komunikačné linky medzi jednotlivými zdrojmi majú rovnakú rýchlosť a tak veľkosť prenášaných údajov nezohráva podstatnú úlohu pri plánovaní, čo v reálnom prostredí nie je pravda.

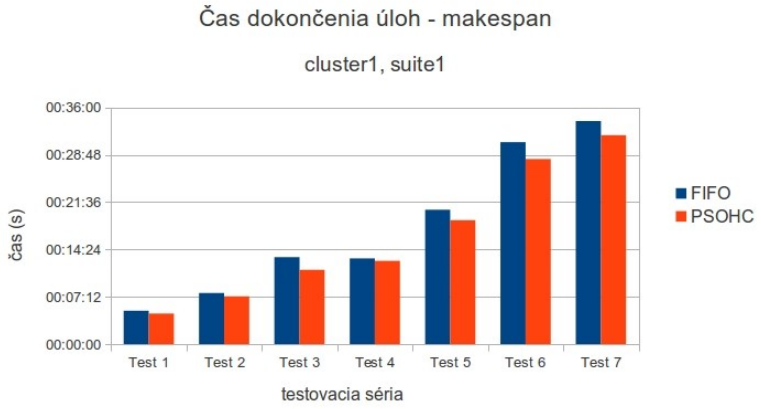
7.2 Výsledky testovania plánovacieho algoritmu v klastri

Výsledky testovania úloh, ktoré sme spúšťali na klastroch automaticky pomocou

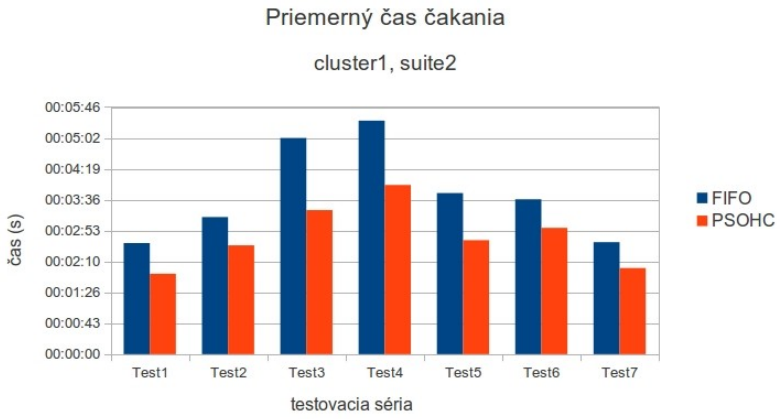
testovacích skriptov sa po vykonaní testovacej skupiny uložia do textového súboru. Následne sa spracujú tak, aby ich bolo možné importovať do tabuľkového kalkulátora, kde je možné ich upraviť a analyzovať.

7.2.1 Testovanie plánovača na virtuálnom klastri

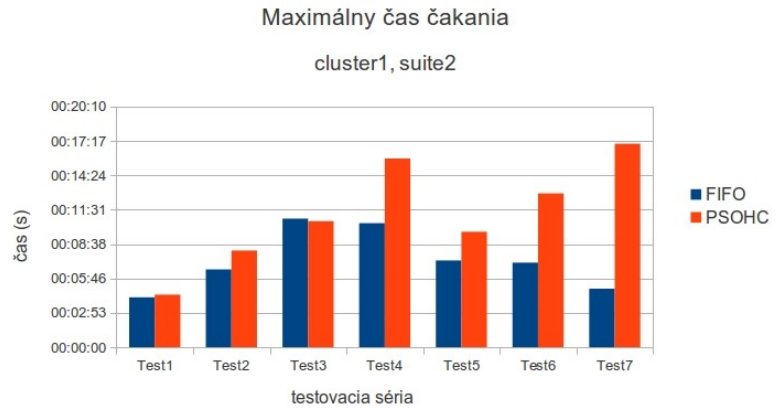
Výsledky ukazujú, že pri použití navrhovaného hybridného algoritmu PSO+HC pri menšom ale aj väčšom počte úloh vo virtuálnom klastri je možné dosiahnuť efektívnejšie plánovanie oproti jednoduchému základnému algoritmu FIFO a to nielen pri statickom, ale aj dynamickom plánovaní. Graf na obrázku č. 17 znázorňuje vývoj funkcie makespan v závislosti od počtu vstupných úloh v testovacej skupine *suite1*, pre ktoré bolo potrebné vygenerovať rozvrh. Navrhovaný algoritmus dokáže urýchliť dobu výpočtu v paralelnom systéme približne o 10%.



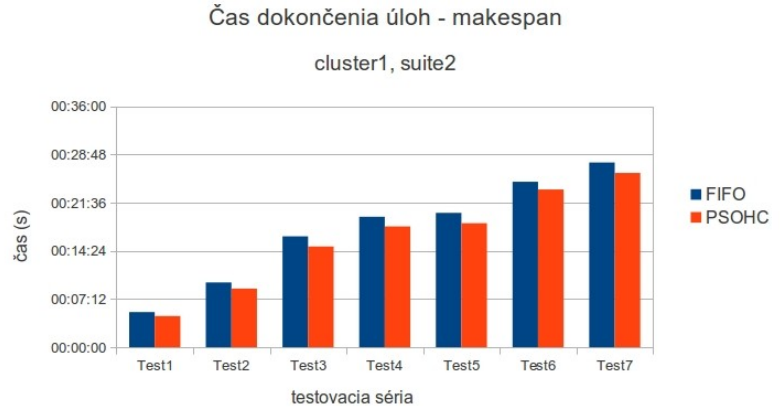
Obrázok č. 17: Čas dokončenia úloh v testovacej skupine suite1.



Obrázok č. 18: Priemerný čas čakania úloh skupiny suite2 v rade.



Obrázok č. 19: Maximálny čas čakania úloh skupiny suite2 v rade.

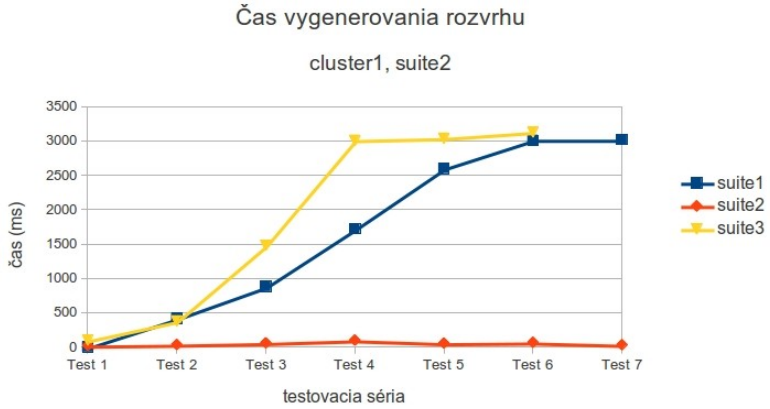


Obrázok č. 20: Čas dokončenia úloh skupiny suite2.

Grafy na obrázkoch č. 18, 19 a 20 znázorňujú efektivitu PSO algoritmu v porovnaní

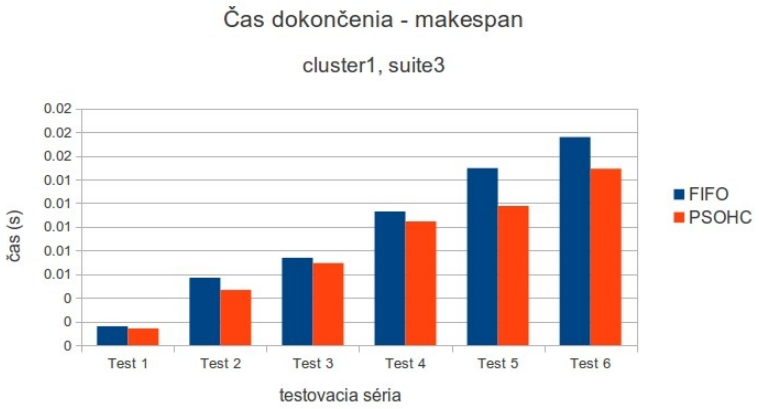
s FIFO pri dynamickom plánovaní. Časy dokončenia úloh pri použití PSO sú opäť o niečo výhodnejšie oproti FIFO. Obrázky č. 18 a 19 znázorňujú priemerný resp. maximálny čas, ktorý uplynie od zaradenia úlohy medzi čakajúce a jej úspešným ukončením. Z údajov vyplýva, že pri použití PSO je priemerný čas čakania úlohy menší ako pri použití algoritmu FIFO, ale maximálny čas je o niečo väčší. Znamená to, že väčšina úloh plánovaných pomocou PSO čakali v rade kratšie ako pri FIFO plánovaní, ale existoval menší počet úloh, ktoré čakali v rade výrazne dlhšie ako pri použití algoritmu FIFO.

Zaujímavým ukazovateľom pri dynamickom plánovaní je čas vygenerovania rozvrhu. Obrázok č. 21 znázorňuje ako sa vyvíja čas vygenerovania rozvrhu pri použití algoritmu PSO+HC a rôznych skupín testovacích úloh. Výhodou plánovania pomocou FIFO je, že nie je potrebný žiadny čas pre vytváranie rozvrhu. Z obrázku ale vidno, že pri dynamickom plánovaní je čas potrebný na vygenerovanie rozvrhu úloh omnoho nižší ako pri statickom plánovaní. Čas vygenerovania rozvrhu rýchlo narastá so zväčšujúcim sa počtom úloh ale je časovo obmedzený a nikdy nepresiahne 3s.

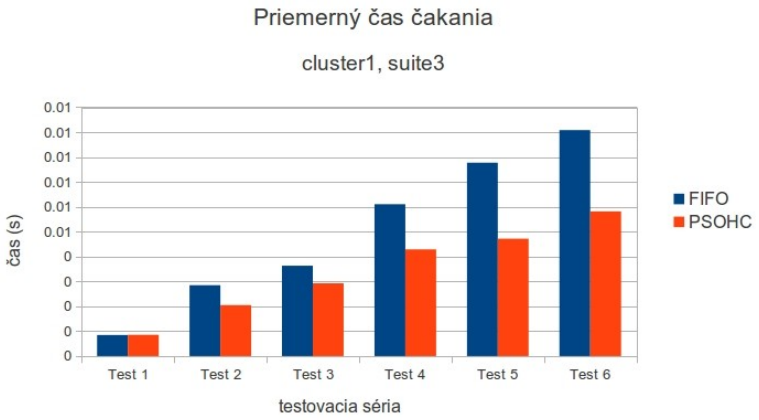


Obrázok č. 21: Čas generovania rozvrhu pri dynamickom plánovaní.

Správanie sa plánovača pri plánovaní úloh so špecifickými požiadavkami na zdroje znázorňujú obrázky č. 22 a 23. Pri použití PSO plánovacieho algoritmu je čas dokončenia úloh v priemere na jednu testovaciu množinu lepší o 15% oproti FIFO plánovaniu. Zároveň pri použití algoritmu PSO úlohy v priemere ostávajú čakať v rade iba 68% času v porovnaní z úlohami plánovanými pomocou FIFO.



Obrázok č. 22: Čas dokončenia úloh skupiny suite3.



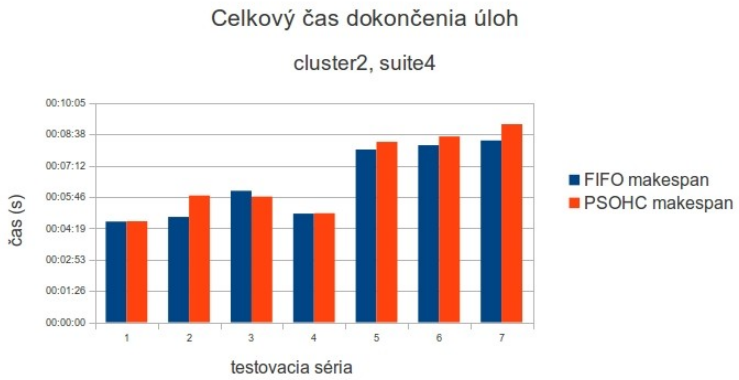
Obrázok č. 23: Priemerný čas čakania úloh skupiny suite3 v rade.

7.2.2 Testovanie plánovača na reálnom klastri

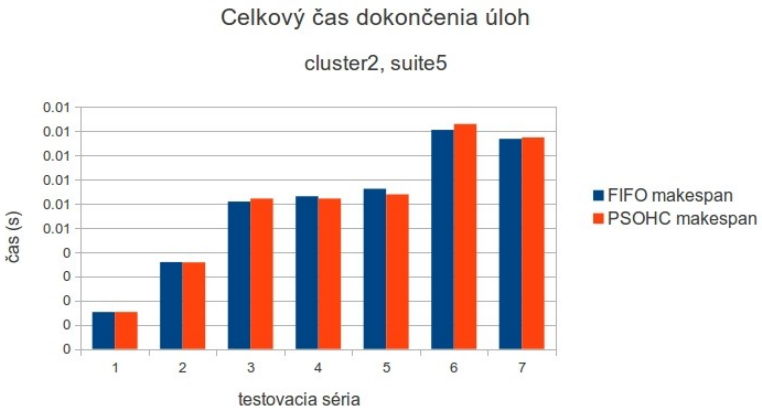
Výsledky experimentov z virtuálneho klastra naznačujú, že navrhovaný algoritmus môže byť vhodnou alternatívou k pôvodnému plánovaciemu algoritmu FIFO. Preto sme sa rozhodli porovnať výsledky plánovania na virtuálnom klastri s výsledkami plánovača na reálnom vysokovýkonnom klastri. Hlavným kritériom pre hodnotenie plánovača boli časy dokončenia úloh v jednotlivých skupinách testov.

Výsledky prvej skupiny *suite4*, ktorá bola zameraná na statické plánovanie dlhších úloh zobrazuje graf na obrázku č. 24. Celkový čas dokončenia úloh v tejto skupine pri použití algoritmu PSO+HC sa však ukázal o niečo dlhší ako pri použití FIFO plánovacieho algoritmu. Priemerný čas dokončenia úloh v sérii pri použití PSO+HC bol 6m 45s. V porovnaní s časom dosiahnutým pomocou FIFO, ktorý bol 6m 26s je to o 5% dlhší čas.

Testovanie dynamického plánovania v skupine *suite5* neprinieslo ani zhoršenie ani zlepšenie celkového času dokončenia úloh, hoci niektoré testovacie série boli pri použití PSO+HC ukončené rýchlejšie iných výpočet trval dlhšie a tak priemerný čas jednej série testovania algoritmu FIFO bol 8m 37s a pri použití algoritmu PSO+HC 8m 38s. Výsledky jednotlivých sérií znázorňuje graf na obrázku č. 25.

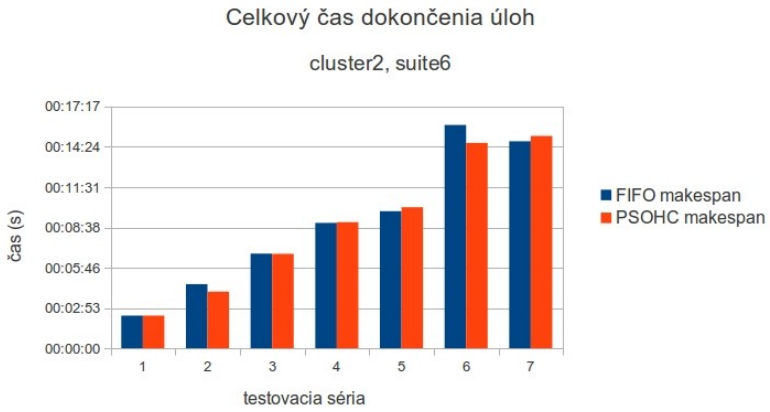


Obrázok č. 24: Čas dokončenia úloh skupiny *suite4*



Obrázok č. 25: Čas dokončenia úloh skupiny *suite5*

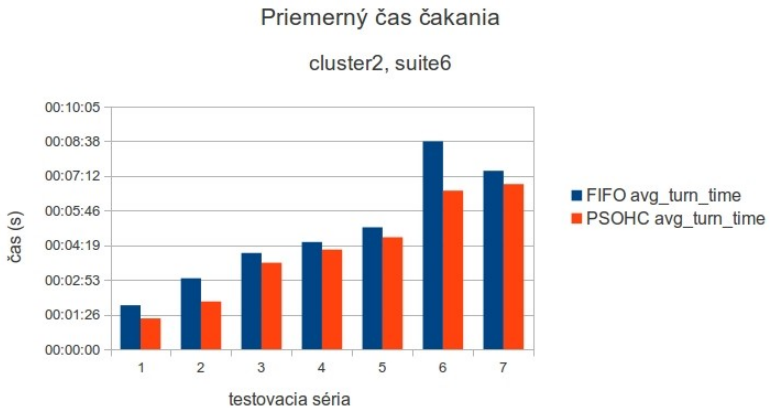
Posledná skupina *suite6,* ktorá bola zameraná na plánovanie úloh so špecifickými požiadavkami prináša len mierne zlepšenie použitím PSO+HC oproti algoritmu FIFO. Tak ako znázorňuje graf na obrázku č. 26 v druhej a šiestej sérii bol čas vykonania úloh výraznejšie lepší, ako zhoršenie času pri ostatných testovacích sériách. Priemerný čas dokončenia úloh pri použití algoritmu FIFO bol 9m 2s, použitím algoritmu PSO+HC sme dokázali skrátiť tento čas na 8m 52s teda približne na 98%.



Obrázok č. 26: Čas dokončenia úloh skupiny *suite6*

Pri krátkych úlohách, ktoré trvajú len niekoľko sekúnd, resp. minút sa zdá, že ušetrený čas je takmer zanedbateľný. V skutočnosti existujú v klastri aj úlohy, ktorý trvanie je omnoho dlhšie a takých úloh je výrazná väčšina. Pri dlhších úlohách sa aj ušetrený čas prejaví výraznejšie.

Dôkazom, že použitím algoritmu PSO+HC je možné skrátiť aj čas čakania úloh v rade sú výsledky skupiny úloh *suite6* zobrazené v grafe na obrázku č. 27. Pri použití algoritmu FIFO je priemerný čas čakania úloh v rade 4m 54s. V porovnaní s algoritmom PSO+HC a časom 4m 9s je to zlepšenie takmer o 16%.



Obrázok č. 27: Priemerný čas čakania úloh skupiny *suite6* v rade

7.3 Využitie navrhovaného algoritmu a pokračovanie práce

Výsledky testovania ukazujú, že použitie navrhovaného plánovacieho algoritmu efektívne znižuje čas potrebný na vykonanie úloh v klastri. Tým šetrí nielen čas používateľov, ale aj prostriedky potrebné na chod a údržbu celého zariadenia. Pre použitie plánovača v stálej prevádzke klastra by bolo vhodné upraviť plánovač tak aby bolo možná konfigurácia parametrov zo vstupného súboru alebo databázy. Ďalej navrhujeme vykonať ďalšie série testov, ktoré budú overovať najvhodnejšie nastavenie vstupných parametrov ako sú veľkosť roja, socializačné parametre c1, c2, počiatočná váha zotrvačnosti w, faktor α, ukončovacia podmienka a iné. Pokračovaním práce by mohla byť optimalizácia algoritmu, ktorá by využila paralelný potenciál počítača na ktorom plánovač pracuje. Keďže v rámci výpočtu jednej iterácie sú výpočty pre každú časticu takmer nezávislé (jediná premenná s ktorou pracujú všetky častice je *pBest*) výpočet pozície častice je vhodný pre vykonávanie v samostatnom vlákne, alebo procese. Takúto paralelizáciu je možné efektívne dosiahnuť pomocou nástrojov ako openMP, alebo MPI.

ZÁVER

Cieľom našej práce bolo teoreticky popísať spôsoby plánovania a vyvažovania záťaže v gridových systémoch a navrhnúť vlastný plánovací algoritmus, ktorý zabezpečuje efektívne rozdeľovanie používateľských úloh medzi výpočtové prostriedky, ktoré ponúka gridová infraštruktúra. Pre pochopenie kontextu v akom sa vyvíjajú a nachádzajú súčasné gridové systémy sme sa rozhodli v prvej časti práce predstaviť históriu gridov a stručne uviesť technologické prostriedky, ktoré najčastejšie bývajú súčasťami gridových systémov. Ide najmä o paralelné počítače so zdieľanou pamäťou tzv. superpočítače, alebo vysokovýkonné počítačové klastre, ktoré vznikajú prepojením mnohých počítačov lokálnou sieťou. Pre správu týchto výpočtových prostriedkov vzniklo niekoľko softvérových systémov. Z nich vyberáme tie, ktoré si získali význam buď svojím prvenstvom v tejto oblasti, alebo svojím výrazným rozšírením v súčasných systémoch.

Už prvotné návrhy ukázali, že neoddeliteľnou súčasťou gridového softvéru musí byť nástroj, ktorý slúži na správu výpočtových prostriedkov, ktoré grid združuje. Nazýva sa správca zdrojov a je nepriamo zodpovedný za vykonávanie úloh, ktoré používatelia zadajú do systému. Aby mohol splniť túto úlohu komunikuje s ďalším programom – plánovačom, ktorého úlohou je nájsť vhodné priradenie úloh výpočtovým prostriedkom. Plánovanie pozostáva z niekoľkých fáz, z ktorých najdôležitejšia je práve výber vhodných prostriedkov na základe plánovacieho algoritmu. Existuje mnoho algoritmov, ktoré sú vhodné na účel plánovania. V praktickej časti práce sa zaoberáme najmä dvomi z nich – Round Robin a Hill Climbing.

Cieľom nášho výskumu bolo navrhnúť vlastný plánovací algoritmus, ktorý by sa nielen vyrovnal týmto algoritmom ale bol by schopný lepšie využiť prostriedky, ktoré ponúka gridový systém. Navrhli sme preto plánovací algoritmus, ktorý využíva spôsoby evolučných algoritmov a je postavený na základe metafory správania sa roja častíc. Uvedený algoritmus sme otestovali pomocou simulácie gridového prostredia s použitím nástroja GridSim. V simulácii sme odoslali do systému istý počet náhodne vygenerovaných úloh a zaznamenávali sme čas potrebný na vytvorenie rozvrhu pre túto skupinu úloh. Zároveň sme zaznamenávali čas potrebný na vykonanie všetkých úloh podľa tohto rozvrhu. Výsledky simulácie a testovania boli pozitívne a ukazujú potenciál použitia podobných heuristických algoritmov pri plánovaní v gridovom systéme. Pomocou nami navrhnutého plánovača sme schopní dosiahnuť kratší výpočtový čas a rovnomerné distribuovanie úloh

podľa výkonu dostupných počítačov a tým zabezpečiť v istom zmysle vyrovnanú záťaž.

Na základe úspešných výsledkov simulácie sme sa rozhodli implementovať navrhnutý algoritmus s menšími úpravami v prostredí lokálneho správcu zdrojov pre vysokovýkonný počítačový klaster. Pre vytvorenie rozvrhu a hľadanie optimálnych lokálnych riešení sme okrem optimalizácie pomocou roja častíc použili horolezecký algoritmus, ktorý je veľmi rýchly a preto je vhodným riešením pre problémy plánovania. To sa odzrkadľuje aj na výsledkoch testovania, ktoré sme vykonali po nasadení plánovača na výpočtový klaster. Výsledky opäť potvrdzujú schopnosť plánovača robiť rozhodnutia, ktorú sú efektívne nielen na globálnej úrovni gridového plánovania, ale aj na úrovni lokálneho plánovania vo vysokovýkonnom paralelnom počítačovom klastri.

REFERENCIE

[1] SCHALLER, R.R. Moore's law: past, present and future. In *Spectrum, IEEE*. 1997. Vol. 34, no. 6, s. 52–59. .

[2] BUYYA, R. *The Design of PARAS Microkernel*. Bangalore: Centre for Development of Advanced Computing, 1998. .

[3] MEUER, H. et al TOP 10 Sites for November 2012. In [online]. 2000. [cit. 2013-04-10]. Dostupné na internete: <http://www.top500.org/>.

[4] RIKEN AICS System Configuration of the K computer. In [online]. [cit. 2013-04-10]. Dostupné na internete: <http://www.aics.riken.jp/en/kcomputer/system-2.html>.

[5] FORTIER, P.J. *Design of distributed operating systems* [online]. [s.l.]: Intertext Publications, 1986. .

[6] BERMAN, F. et al *Grid Computing: Making the Global Infrastructure a Reality* [online]. [s.l.]: Wiley, 2003. ISBN 9780470853191.

[7] RIVEST, R.L. et al A method for obtaining digital signatures and public-key cryptosystems. In *Communications of the ACM*. 1978. Vol. 21, no. 2, s. 120–126. .

[8] ŠKRINÁROVÁ, J. - KRNÁČ, M. Particle Swarm Optimization Model for Grid Scheduling. In *Second International Conference on Computer Modelling and Simulation*. Brno: Brno University of Technology, 2011. s. 146–153. .

[9] FOSTER, I. Globus toolkit version 4: software for service-oriented systems. In *Proceedings of the 2005 IFIP international conference on Network and Parallel Computing* [online]. Berlin, Heidelberg: Springer-Verlag, 2005. s. 2–13. Dostupné na internete: <http://dx.doi.org/10.1007/11577188_2>.

[10] MATEESCU, G. et al Hybrid Computing-Where HPC meets grid and Cloud Computing. In *Future Gener. Comput. Syst.* 2011. Vol. 27, no. 5, s. 440–453. .

[11] NEWHOUSE, S. EGI-InSPIRE Project Presentation. In [online]. 2011. [cit. 2013-04-10]. Dostupné na internete: <https://documents.egi.eu/public/RetrieveFile?docid=506&version=5&filename=EGI-InSPIRE-May2011-V3.pdf>.

[12] HENDERSON, R.L. Job Scheduling Under the Portable Batch System. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing* [online]. London, UK, UK: Springer-Verlag, 1995. s. 279–294. Dostupné na internete: <http://dl.acm.org/citation.cfm?id=646376.689372>.

[13] SHANKAR, U. *Oracle Grid Engine User Guide Release 6.2 Update 7* [online]. 2012. .

[14] ZHOU, S. et al Utopia: a load sharing facility for large, heterogeneous distributed computer systems. In *Softw. Pract. Exper.* 1993. Vol. 23, no. 12, s. 1305–1336. .

[15] TORQUE User's Manual. In [online]. 2001. Dostupné na internete: <http://www.clusterresources.com/torquedocs21/usersmanual.shtml>.

[16] KUMAR, P. *Load Balancing and Job Migration in Grid Environment*. Patiala: Thapar University, 2009. .

[17] GHANEM, J. *Implementation of Load Balancing Policies in Distributed Systems*. Beirut: American University of Beirut, 2004. .

[18] SCHOPF, J.M. Grid resource management. In NABRZYSKI, J. et alEd. [online]. Norwell, MA, USA: Kluwer Academic Publishers, 2004. s. 15–23. ISBN 1-4020-7575-8Dostupné na internete: <http://dl.acm.org/citation.cfm?id=976113.976116>.

[19] WOLSKI, R. et al The network weather service: a distributed resource performance forecasting service for metacomputing. In *Future Gener. Comput. Syst.* 1999. Vol. 15, no. 5-6, s. 757–768. .

[20] GUIM, F. et al Job Scheduling Strategies for Parallel Processing. In FRACHTENBERG, E. - SCHWIEGELSHOHN, U.Ed. [online]. Berlin, Heidelberg: Springer-Verlag, 2009. s. 59–79. ISBN 978-3-642-04632-2Dostupné na internete: <http://dx.doi.org/10.1007/978-3-642-04633-9_4>.

[21] ETMINANI, K. - NAGHIBZADEH, M. A Min-Min Max-Min selective algorihtm for grid task scheduling. In *Internet, 2007. ICI 2007. 3rd IEEE/IFIP International Conference in Central Asia on*. 2007. s. 1–7. .

[22] DAI, Y.-S. - WANG, X.-L. Optimal resource allocation on grid systems for maximizing service reliability using a genetic algorithm. In *Reliability Engineering & System Safety*. 2006. Vol. 91, no. 9, s. 1071 – 1082. .

[23] IZAKIAN, H. et al An auction method for resource allocation in computational grids. In *Future Gener. Comput. Syst.* 2010. Vol. 26, no. 2, s. 228–235. .

[24] CHLUMSKY, V. et al The extension of TORQUE scheduler allowing the use of planning and optimization in grids. In *Computer Science*. 2012. Vol. Vol. 13 (2), s. 5–19. .

[25] FREY, J. et al Condor-G: A Computation Management Agent for Multi-Institutional Grids. In *Cluster Computing*. 2002. Vol. 5, no. 3, s. 237–246. .

[26] *Moab Workload Manager Administrator Guide version 6.1.2* [online]. 2011. .

[27] Maui Scheduler - Backfill. In [online]. Dostupné na internete: <http://docs.adaptivecomputing.com/maui/8.2backfill.php#overview>.

[28] SOTSKOV, Y.N. - SHAKHLEVICH, N.V. NP-hardness of shop-scheduling problems with three jobs. In *Discrete Applied Mathematics*. 1995. Vol. 59, no. 3, s. 237 – 266. .

[29] KENNEDY, J. - EBERHART, R. Particle swarm optimization. In *Neural Networks, 1995. Proceedings., IEEE International Conference on*. 1995. s. 1942–1948 vol.4. .

[30] DUVIVIER, D. et al Stochastic Algorithms for Optimization and Application to Job-Shop-Scheduling. In. 1995. .

[31] HU, X. et al Swarm intelligence for permutation optimization: a case study of n-queens problem. In *Swarm Intelligence Symposium, 2003. SIS '03. Proceedings of the 2003 IEEE*. 2003. s. 243–246. .

[32] BUYYA, R. GridSim: A Grid Simulation Toolkit for Resource Modelling and Application Scheduling for Parallel and Distributed Computing. In [online]. 2002. [cit. 2013-04-10]. Dostupné na internete: <http://www.buyya.com/gridsim/>.

[33] PRIMATE LABS INC. Geekbench2.4.3. In [online]. 2004. [cit. 2013-04-10]. Dostupné na internete: <http://www.primatelabs.com/geekbench/>.

[34] CESNET MetaCentrum NGI. In [online]. 2013. [cit. 2013-04-10]. Dostupné na internete: <http://www.metacentrum.cz/cs/index.html>.

[35] Klaster SIVVP. In [online]. 2013. Dostupné na internete: <http://www.hpcc.umb.sk/sk/klaster-sivvp.html>.

[36] Titan. In [online]. Dostupné na internete: <http://www.olcf.ornl.gov/wp-content/themes/olcf/titan/images/gallery/titan1.jpg>.

[37] NASA 128-processor Beowulf cluster. In [online]. Dostupné na internete: <http://www.cse.mtu.edu/Common/cluster.jpg>.

[38] Klaster SIVVP. In [online]. Dostupné na internete: <http://hpcc.umb.sk/public/filestore/documents/richtext/8/umb3.jpg>.

Príloha A – zdrojový kód vytvorenia rozvrhu pomocou PSO+HC

```
int generate_schedule(server_info *sinfo) {
    int i,j,k;

    /* Initialize jobMap */
    int job_count;
        job_info**  jobMap  =  job_filter(sinfo->jobs,  sinfo->sc.total,
check_job_queued, &job_count);

    int node_count;
    sch_node** nodeMap = init_nodeMap(sinfo, &node_count);

    /** PSO OPTIMIZATION **/

    /** BEGIN initialization **/
    srand((unsigned)time(NULL));

        /** Configurable parameters **/
        int swarmSize = 2 * job_count;
        double xmin = 0.0;
        double xmax = 4.0;
        double vmin = -4.0;
        double vmax = 4.0;
        double w = 0.9;    //initial inertia weight;
        double alpha = 0.975; //decrement factor of inertia weight
        double c1 = 2, c2 = 2;
        double r1 = 0.5, r2 = 0.5;
        int maxgBestAge = 1000;
        /** Configurable parameters **/

    particle** swarm = (particle**)malloc(swarmSize * sizeof(particle*));
    double * gBest = (double*)malloc(job_count * sizeof(double));
    long gBestV = LONG_MAX;
    for (i=0; i<swarmSize; i++) {
        particle *p = new_particle(job_count);
        for (j=0; j<job_count; j++) {
            p->x[j] = xmin + ((xmax - xmin) * rnd());
            p->v[j] = vmin + ((vmax - vmin) * rnd());
            p->sList[j].key = p->x[j];
            p->sList[j].iJob = j;
            p->pBest[j] = p->x[j];
        }

                        qsort(p->sList,   job_count,   sizeof(sort_job_info),
sort_job_by_key);
        for (j=0; j<job_count; j++) {
            p->jList[j] = jobMap[p->sList[j].iJob];
        }

        fill_nodeMap(nodeMap, node_count, p->jList, job_count);
        long value = eval_nodeMap(nodeMap, node_count, sinfo);

        p->pBestV = value;
        if (value < gBestV) {
            printf("Updating initial gBest: %ld\n", value);
            gBestV = value;
            memcpy(gBest, p->pBest, job_count * sizeof(job_info*));
```

```
        }
        swarm[i] = p;
    }
    /** END initialization **/

    int done = 0;
    int gBestAge = 1;
    clock_t c0 = clock();
    while (!done) {
        w = w * alpha;
        for (i=0; i<swarmSize; i++) {
            particle* p = swarm[i];
            for (j=0;j<job_count;j++) {
                p->v[j] = (w * p->v[j]) + (c1*r1*(p->pBest[j] - p->x[j]))
+ (c2*r2*(gBest[j] - p->x[j]));
                p->x[j] = p->x[j] + p->v[j];
                p->sList[j].key = p->x[j];
                p->sList[j].iJob = j;
            }
                    qsort(p->sList,  job_count,  sizeof(sort_job_info),
sort_job_by_key);
            for (j=0; j<job_count; j++) {
                p->jList[j] = jobMap[p->sList[j].iJob];
            }

            fill_nodeMap(nodeMap, node_count, p->jList, job_count);
            long value = eval_nodeMap(nodeMap, node_count, sinfo);

            if (value < p->pBestV) {
                p->pBestV = value;
                memcpy(p->pBest, p->x, job_count * sizeof(double));
            }
            if (value < gBestV) {
                printf("Updating gBest: %ld\n", value);
                gBestAge = 1;
                gBestV = value;
                memcpy(gBest, p->x, job_count * sizeof(job_info*));
            }
        }
        clock_t c1 = clock();
        double timeDiff = (c1 - c0) * 1000 / CLOCKS_PER_SEC;
        if (gBestAge++ >= maxgBestAge || timeDiff > 3000) done = 1;
    }

    //create shareable schedule representation from nodeMap and job list
from gBest
    particle *g = new_particle(job_count);
    for (j=0; j<job_count; j++) {
        g->sList[j].key = gBest[j];
        g->sList[j].iJob = j;
    }
    qsort(g->sList, job_count, sizeof(sort_job_info), sort_job_by_key);
    for (j=0; j<job_count; j++) {
        g->jList[j] = jobMap[g->sList[j].iJob];
    }

    fill_nodeMap(nodeMap, node_count, g->jList, job_count);
```

```
    /* Assign every job its scheduled node_request list */
    for (i=0;i<node_count; i++) {
        node_request *r = nodeMap[i]->nrl.b;
        while (r != NULL) {
            node_request *copy = copy_node_request(r);
            list_push(&r->job->nodeList, copy);
            r = r->next;
        }
    }

    _sched = job_count;
    sched = (sch_item**)malloc(_sched * sizeof(sch_item*));
    for (j=0; j<_sched; j++) {
        sched[j] = (sch_item*)malloc(sizeof(sch_item));
        strcpy(sched[j]->jobName, g->jList[j]->name);
        sched[j]->nodeList = node_request_2_str(g->jList[j]->nodeList);
        DBPRT(("%s: %s\n", sched[j]->jobName, sched[j]->nodeList));
    }

    /** FREE MEMORY AND FINISH  **/
    free_nodeMap(nodeMap, node_count);
    free(gBest);
    free_particle(g);
    for (i=0; i<swarmSize; i++) {
        free_particle(swarm[i]);
    }
    free(swarm);
    free(jobMap);
    return;
    /** FREE MEMORY AND FINISH **/

    return 0;
}
```

Príloha B – príklad odosielacieho súboru testovacej série

```
#!/bin/bash
echo "Submitting job 1 (length: 67, nodes: 1)"
echo "/home/mkrnac2/dipl/prgs/fft 26" | qsub -N test_job_1 -l
nodes=1,walltime=00:01:07 -e /dev/null -o /dev/null


echo "Submitting job 2 (length: 24, nodes: 1)"
echo "/home/mkrnac2/dipl/prgs/monte_carlo 2000000" | qsub -N test_job_2
-l nodes=1,walltime=00:00:24 -e /dev/null -o /dev/null


echo "Submitting job 3 (length: 12, nodes: 1)"
echo "/home/mkrnac2/dipl/prgs/monte_carlo 1000000" | qsub -N test_job_3
-l nodes=1,walltime=00:00:12 -e /dev/null -o /dev/null


echo "Submitting job 4 (length: 91, nodes: 1)"
echo "/home/mkrnac2/dipl/prgs/fibonacci 49" | qsub -N test_job_4 -l
nodes=1,walltime=00:01:31 -e /dev/null -o /dev/null


echo "Submitting job 5 (length: 30, nodes: 1)"
echo "/home/mkrnac2/dipl/prgs/binary 100000000" | qsub -N test_job_5 -l
nodes=1,walltime=00:00:30 -e /dev/null -o /dev/null


echo "Submitting job 6 (length: 241, nodes: 7)"
echo "sleep 241" | qsub -N test_job_6 -l nodes=7,walltime=00:04:01 -e
/dev/null -o /dev/null


echo "Submitting job 7 (length: 118, nodes: 1)"
echo "/home/mkrnac2/dipl/prgs/whetstone 0" | qsub -N test_job_7 -l
nodes=1,walltime=00:01:58 -e /dev/null -o /dev/null


echo "Submitting job 8 (length: 232, nodes: 2)"
echo "sleep 232" | qsub -N test_job_8 -l nodes=2,walltime=00:03:52 -e
/dev/null -o /dev/null


echo "Submitting job 9 (length: 30, nodes: 1)"
echo "/home/mkrnac2/dipl/prgs/binary 100000000" | qsub -N test_job_9 -l
nodes=1,walltime=00:00:30 -e /dev/null -o /dev/null


echo "Submitting job 10 (length: 230, nodes: 5)"
echo "sleep 230" | qsub -N test_job_10 -l nodes=5,walltime=00:03:50 -e
/dev/null -o /dev/null
```

Príloha C – vykonanie testovacej série a získanie výsledkov

```
#!/bin/bash

#submission list folder
STAT_FOLDER="statistics"
PBS_HOME="/var/spool/torque"
VERBOSE=
DYNAMIC=
PREFIX="qsub_"

usage() {
cat << ABCD
        usage: $0 options
        This script runs all the submissions scripts in given folder
        OPTIONS:
        -d                      dynamic scheduling test round
         -f FOLDER       Folder where PBS test round submission files are
located
        -h                      Show this message
        -p prefix       test round submission file prefix
         -s name          statisticsf folder name, if does not exists it
will be created in the same directory as given in -f
        -t FOLDER       PBS_HOME folder
        -v                      Verbose
ABCD
}

while getopts "df:hp:s:t:v" OPTION
do
     case $OPTION in
         d)
             DYNAMIC=1
             ;;
         f)
             SFOLDER=$OPTARG
             ;;
         h)
             usage
             exit 1
             ;;
         p)
             PREFIX=$OPTARG
             ;;
         s)
             STAT_FOLDER=$OPTARG
             ;;
         t)
             PBS_HOME=$OPTARG
             ;;
         v)
             VERBOSE=1
             ;;
         ?)
             usage
             exit
             ;;
     esac
```

```
done

function log () {
    if [[ $VERBOSE -eq 1 ]]; then
        echo "$@"
    fi
}

if [[ -z "$SFOLDER" ]]; then
     usage
     exit 1
fi

if [[ ! -d $SFOLDER ]]; then
    echo "Wrong usage $SFOLDER is not a directory"
        exit
fi

i=1
for test_file in $SFOLDER/$PREFIX*; do

        if [[ $DYNAMIC -eq 1 ]]; then
                SW_DYN="-d"
        else
                SW_DYN=""
        fi

        if [[ $VERBOSE -eq 1 ]]; then
                SW_VER="-v"
        else
                SW_VER=""
        fi

         run_single_test_serie -f $test_file -s $STAT_FOLDER -t $PBS_HOME
-i $i $SW_DYN $SW_VER

        ((i++))

        sleep 60
done

#results preprocessing
TFILE="/tmp/out.tmp.$$"

cd $SFOLDER/$STAT_FOLDER/

for tf in qstat_test*
do
  if [ -f $tf -a -r $tf ]; then
        f="p_$tf"
        cp $tf $f

        sed "s/Job\ //g" "$f" > $TFILE && mv $TFILE "$f"
        sed "s/:\ /=/g" "$f" > $TFILE && mv $TFILE "$f"
        sed "s/\ =\ /=/g" "$f" > $TFILE && mv $TFILE "$f"

        today=`date +"%a\\ %b\\ \\ %-d\\ "`
        yesterday=`date -d '1 day ago' +"%a\\ %b\\ \\ %-d\\ "`
```

```
        year=`date +"%Y"`

        sed "s/$today//g" $f > $TFILE && mv $TFILE $f
        sed "s/$yesterday//g" $f > $TFILE && mv $TFILE $f
        sed "s/\ $year//g" "$f" > $TFILE && mv $TFILE "$f"

         sed "s/-e\ \/dev\/null\ -o\ \/dev\/null//g" "$f" > $TFILE && mv
$TFILE "$f"

        tr '\n' '#' < $f > $TFILE && mv $TFILE "$f"

        sed 's/Id/\
Id/g' "$f" > $TFILE && mv $TFILE "$f"

        sed "s/\s\+//g" "$f" > $TFILE && mv $TFILE "$f"

        cat $f >> process_all
        rm $f

  else
   echo "Error: Cannot read $f"
  fi
done
rm $TFILE
```