Математички факултет
Универзитет у Београду

ДИНАМИЧКО УРАВНОТЕЖАВАЊЕ ОПТЕРЕЋЕЊА ЗА СИМУЛАЦИЈЕ У СТАТИЦИ

- ДИПЛОМСКИ МАСТЕР РАД -

Ментори:
Проф. др Миодраг Живковић
Др Ralf-Peter Mundani

Јована Кнежевић
Број индекса: 1044/2008

Београд, септембар 2009.

Садржај

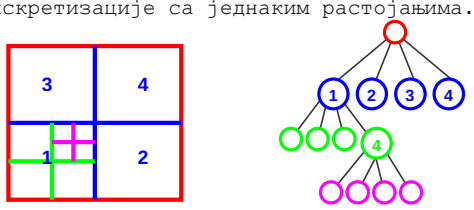
I Увод

1. Мотивација

Стални напредак рачунарске технологије је мотивисао многе који се баве природним наукама и инжењерством да примењују нумеричке методе како би симулирали велике проблеме. Међутим, захтевност у вези са сложеношћу и количином података у нумеричким симулацијама у различитим областима истраживања, као што су структурна динамика, апликације за изградњу протеина, предвиђање временских прилика, моделирање глобалне климе, симулације у математичким финансијама, итд. су такође у сталном порасту. Моћ израчунавања данашњих микропроцесора није довољна да се пратио овај пораст, тако да се развијају методе за паралелна израчунавања.

У инжењерским применама, област структурне динамике представља велики изазов, нарочито када су у питању нумеричка израчунавања. Како рачунања која се изводе у паралелним симулацијама у овој области онемогућавају довољно брзо добијање резултата (на пример, као што је неопходно у апликацијама где се подразумева и корисничка интеракција, тако да је неопходно често освежавање резултата) и уобичајене технике разлагања у већини случајева не дозвољавају да се у потпуности искористе могућности расположивог хардвера, потребне су софистициране структуре података и методе.

Хијерархијске структуре, као што су стабла, су најпогодније због могућности субструктуирања простора. Ова особина је, на пример за октална стабла, искоришћена за сукцесивно половљење коцке (или квадрата у две димензије) која садржи неке геометријске податке за сваку димензију тела, док резултујући делићи не буду лежали у потпуности унутар или изван тог тела. Користећи ову структуру за смештање фигуре, сложеност може бити смањена на $O(N^2)$ делића у три димензије или $O(N)$ делића у две димензије уместо $O(N^3)$ or $O(N^2)$, респективно, у случају дискретизације са једнаким растојањима.



Слика 1.1: Кватернарно стабло (quadtree) – могућност субструктуирања 2D простора.

С друге стране, перформансе паралелног програма се не побољшавају осим ако се, поред ових структура, не користе технике уравнотежавања оптерећења. Циљ истраживања био је да се развије ефикасна стратегија за уравнотежавање код једне симулације структура заснованој на тзв. „P-верзији“ методе коначних елемената (finite element method, FEM) организованих у октално стаблу.

2. Увод у паралелно програмирање

Паралелни рачунари могу бити грубо класификовани према степену у коме хардвер подржава паралелизам у, с једне стране, рачунаре са више језгара и више процесора који имају више обрадних јединица унутар једне машине (обично дељена меморија) и, с друге стране, масивно паралелне процесоре[1] и мреже, који користе више рачунара за извршавање једног посла (системи са дистрибуираном меморијом). Међутим, да би се имало било какве користи од паралелног програмирања, сам програм мора бити *паралелан*, односно написан тако да га симултано извршава више од једног процеса. Примарни циљ паралелног програмирања је побољшање перформанси израчунавања у односу на перформансе добијене серијским извршавањем одговарајућег програма. Имајући ово на уму, свако треба да се определи за

[1] енг. massive parallel processing (MPP) односи се на рачунарске системе са стотинама, ако не и хиљадама, независних аритметичких јединица или читавих микропроцесора.

најбољу паралелну имплементацију неког одређеног проблема узимајући у обзир доступне рачунарске ресурсе и сам проблем (могућност примене одређених метода и структуирања података конкретно на њега).

Што се тиче рачунарских ресурса, у *системима са дељеном меморијом* сви процесори имају једнак приступ једној или више меморијских банки. Све области у меморији су једнако доступне свим процесорима и преноси података са процесора на процесор се обављају користећи дељене области у меморији. Сви процеси имају директан приступ дељеним подацима, тако да, генерално говорећи, морају бити синхронизовани. Када користимо *меморијски повезан* (memory-coupling) паралелни модел у оваквим системима, ми, заправо, пресликавамо програме написане у секвенцијалном језику на паралелни рачунар путем преводилачких директива (углавном за паралелизацију петљи). У случају паралелизма заснованог на нитима, један једини процес извршава код између петљи, али активира скуп процеса који му се придружују у извршавању делова петље симултано. Ово омогућавају ***OpenMP***[2] директиве.

У *системима са дистрибуираном меморијом*, сваки процесор у систему има своју засебну банку меморије. Преноси података са процесора на процесор се обављају преко неке врсте мрежне топологије. У овим системима се користи модел повезивања порукама (message-coupling model) – модел који проширује паралелне програмске језике (C/C++, Fortran) додатним конструкцијама паралелног језика, имплементираних преко процедуралних позива из одговарајућих библиотека (на пример, ***MPI***[3]).

На неким системима, као што су системи са дистрибуираном и дељеном меморијом, најпогодније је комбиновати слање порука са мултитредингом (multithreading) да би се постигле максималне перформансе. Овај приступ се назива *хибридно* или *вишеслојно* (multi-level) *паралелно програмирање*.

Како мреже бивају брже и мултипроцесори слабије повезани, ове раније јасне разлике између коришћења дељене меморије и слања порука као механизама за комуникацију постале су слабије видљиве. Треба поменути три најзначајнија комерцијална типа мултипроцесора:

1) *мултипроцесори са униформним приступом меморији* (uniform memory access multiprocessors, UMA или cc-UMA, који користе хардвер посебне намене да одржавају кохерентни кеш) са приступним временом свакој меморијској локацији скоро независнимод тога који процесор захтева приступ или на ком се меморијском чипу налазе подаци.

2) *мултипроцесори са неуниформним приступом меморији* (non-uniform memory access multiprocessors, NUMA или cc-NUMA), где време приступа меморији зависи од локације меморије у односу на процесор, а процесор може приступити својој локалној меморији брже него другим меморијама (меморијама локалним за друге процесоре, или меморијама дељеним међу процесорима).

3) *мултипроцесори без даљинског приступа меморији* (no remote memory access, NORMA), где мултипроцесори чији се рад заснива на размени порука комуницирају преко унутрашње или спољашње мреже.

За ефективан паралелни приступ израчунавању потребна је следећа комбинација могућности хардвера и софтвера:

1) Веза између процесора и меморије мора омогућавати веома брзу комуникацију између појединачних процеса, као и брз трансфер података до и из меморије.
2) Мора постојати протокол за комуникацију (било да је заснован на портовима (каналима) или на идентификаторима процеса).
3) Мора бити могуће ефективно раздвојити делове алгоритма и улазних података у мање целине (*проблем разлагања*). Грубо говорећи, постоје две врсте разлагања: разлагање домена (паралелизам података) и функционално разлагање (паралелизам задатака).

---

[2] „Open Multi-Processing“ интерфејс за програмирање апликација који се састоји од скупа преводилачких директива, библиотечких функција и променљивих окружења које утичу на ток извршавања програма.

[3] Ипак, MPI (*Message Passing Interface* – спецификација библиотеке за размену порука) је довољно флексибилан да би био коришћен, такође, и у системима са дељеном меморијом.

Код *разлагања домена*, подаци су подељени у делове који су приближно исте величине и мапирани на различите процесе. Сваки процесор ради само са деловима података који су му додељени. Међутим, процеси у неким случајевима имају потребу да размењују податке међусобно (на пример, на ивицама поддомена).

За неке велике и комплексне проблеме, разлагање домена није најефикаснија стратегија код израчунавања. Ово је случај, на пример, када појединачни подскупови података додељени различитим процесима захтевају драстично различито време за обраду. Перформансе извршавања су тада ограничене брзином најспоријег процеса, пошто га преостали процеси чекају за то време, тако да је *функционално разлагање* много ефикаснији приступ. На овај начин, проблем је подељен на више мањих функција, које онда бивају прослеђене процесорима чим њихово извршавање дође на ред тако да процесорима који брже заврше једноставно буде додељен додатни посао.

4) Мора бити омогућено додељивање појединачних потпроблема различитим процесима.
5) *Уравнотежавање оптерећења* (load balancing) система – равномерна подела потребног посла међу свим доступним процесима, која обезбеђује да нити један од њих не чека док други активно раде на својим додељеним задацима. На овај начин, драгоцени рачунарски ресурси се не расипају.

Рад је организован на следећи начин. У Поглављу 2 су описане неопходне нумеричке методе коришћене за симулацију – дискретизација методом коначних елемената и како решавач који користи тзв. „рекурзивну дисекцију“ (nested dissection), а заснован је на хијерархијским структурама, може бити примењен на ове елементе. У Поглављу 3, презентовано је шта је постигнуто у раду на овој тези. Објашњене су технике за уравнотежавање оптерећења које су коришћене у усвојеном паралелном моделу, на који начин је посао равноправно подељен различитим процесима као и како изгледа шаблон за комуникацију. У исто време, приказани су до сада добијени резултати мерења. Међутим, нагласак је на предностима концепта овакве паралелизације, а не конкретни бенчмарк (benchmark) резултати, који ће бити добијени током будућег истраживања. Могућности значајног побољшања ове имплементације нису још увек исцрпљене, тако да су нека побољшања предложена у Поглављу 4.

***

Истраживање за ову тезу је обављено на Technische Universität München, Fakultät für Bauingenieur und Vermessungswesen, на Катедри за рачунарство у инжењерству.

Посебно бих се захвалила др Ralf-Peter Mundani-ју, који је предложио бројне драгоцене идеје, разматрао значај свих нових резултата са мном и пажљиво организовао мој боравак у Минхену. Његова стручност, разумевање и стрпљење су знатно допринели квалитету мог истраживања. Захвалила бих се, такође, и свом номиналном ментору, проф. др Миодрагу Живковићу, за логистичку подршку, помоћ у планирању тезе и будно надгледање на свим нивоима овог истраживачког пројекта. Захвална сам обојици и осталим члановима комисије на коментарима везаним за претходне верзије овог писаног рада.

Писање ове тезе било је јако инспиративно, али без помоћи толико појединаца текст би изгледао нешто другачије. Ипак, као аутор, сама сносим одговорност за коначну верзију и, уз то, гарантујем да сам самостално састављала рад, искључиво се ослањајући на наведене изворе.

У закључку бих споменула да моје истраживање не би било могуће без финансијске подршке International Graduate School of Science and Engineering (IGSSE) и без дозволе проф. др Ernst Rank-а да проведем пет месеци у веома пријатељској и мотивишућој атмосфери на његовој катедри. На крају, желела бих да нагласим да сматрам ову тезу само почетком изазовног истраживања, не и мојим коначним доприносом овој тематици.

II : Методе и структуре података

1. „P“-верзија методе коначних елемената

Метод коначних елемената има своје корене у потреби за решавањем сложених проблема структурне анализе у области грађевинарства. У механици конструкција, потребно је предвидети понашање структуре под дејством неке силе. Применом FEM се на пример могу добити притисак и истезање ако је деформација структуре под дејством силе претходно позната.

Код ове методе, домен дистрибуираног физичког система је подељен на бројне тзв. коначне елементе, што резултује, углавном, у добијању троугаоних или четвороугаоних елемената у дводимензионалном или тетраедарских и хексаедарских елемената у тродимензионалном простору, респективно. За сваки од елемената потом дефинишемо тзв. базну функцију (линеарну или вишег степена) и дефинишемо проблем локално.

Понашање елемента неког одређеног типа је описано преко оптерећења и понашања под њим у појединачним чворовима. Анализом сваког елемента добијамо матрицу малих димензија која повезује вектор променљивих – *померај чвора* (енг. displacement vector) – са вектором примењених сила - *вектором оптерећења* (енг. load vector). Елементи матрице, назване *матрица крутости* (енг. stiffness matrix), који описују еластичну деформацију под дејством силе, се рачунају као интеграли од парцијалних извода функција које зависе од облика и материјала елемената.

Када су израчунати, матрице крутости и вектори оптерећења елемената се комбинују, према принципу суперпозиције, у једну велику матрицу - глобалну матрицу крутости која се односи на цео, комплексни, систем – и вектор, респективно. Како су особине сваког елемента описане према његовом понашању на одређеним дискретним чворовима дуж ивица, комбиновање матрица елемената у глобалну матрицу се заснива на чињеници да нема међусобних утицаја елемената једних на друге осим на поменутим дискретним чворовима, као и на томе да је удео у одговору чвора који је заједнички за два елемента исти за оба елемента. Да резимирамо, алгебарске једначине које описују понашање система су: $K \cdot u = d$, где је K матрица крутости система, u су помераји чворова, а d све акумулиране силе (површинске, запреминске и у чворовима).

Ако је основни начин за повећање степена тачности приближног решења да се додатно издели мрежа (обично имамо алгоритам како да то постигнемо), говоримо о „h“-*методу* (енг. h-method, h је обично пречник највећег елемента у мрежи). Ако уместо да смањујемо h, повећавамо степен полинома коришћених у базним функцијама, говоримо о „p“-*методу* (енг. p-method, p-FEM). Метод који комбинује ова два назива се „hp“-метод (енг. hp-method).

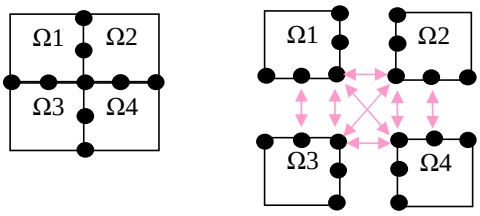
Као што је примећено у [2], обично се нумеричка израчунавања везана за FEM састоје из две целине: прва је рачунање матрице крутости елемента, а друга решавање резултујућег система линеарних једначина. Што се последњег поменутог тиче, када имамо израчунату глобалну матрицу крутости и она је ретка, за очекивати је да постоје методи за решавање система који су ефикаснији од инвертовања матрице. Најчешће су фаворизоване итеративне методе попут конјуговано градијентне (енг. conjugate gradient, CG), али за проблеме који нису нарочито велики, директне методе решавања такође дају добре резултате.

У симулацији је коришћена „пе“-верзија FEM (p-FEM) да би се повећала тачност решења, што је погодно јер дискретизација, тј. елементи остају исти, али за резултујући систем једначина, због његовог лошег кондиционалног броја, софистициране итеративне методе као што је CG више нису ефикасне. Због тога треба размотрити директне методе решавања као што су Гаусова (Gauss) или Чолески декомпозиција (енг. Cholesky decomposition) које су се у овом случају показале ефикаснијим од итеративних метода, мада је још увек сложеност $O(n^3)$ за Гаусов метод. Ове методе нису погодне за паралелизацију.

Дакле, неопходне су другачије методе за решавање добијеног система једначина. Као што је наглашено у [2], примена хијерархијских концепата заснованих на тзв. "нестид дисекшн“ принципу омогућава како дизајнирање софистицираних решавача, тако и напредне стратегије за паралелизацију.

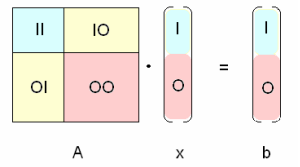
2. Рекурзивна дисекција

Раних седамдесетих, Алан Џорџ (Alan George) је први применио *"нестид дисекшн"* (Nested Dissection, ND) на проблеме са коначним елементима [1]. Као што је описано у [2], његова идеја је била да дели цео домен Ω рекурзивно у поддомене Ω*i* и, након елиминисања свих непознатих које су локалне за Ω*i*, да реши преостали систем једначина, чија је матрица названа *Шуров комплемент* (енг. Schur complement). Сада када су подсистеми ослобођени поменутих утицаја локалних променљивих, они могу бити процесирани независно једни од других.



Слика 2.1: Подела домена у под-домене са заједничким променљивим на ивицама.

Размотримо систем једначина $K \cdot u = d$. Ако га приближно „преполовимо" по свакој димензији, добијамо четири под-матрице K_i, $i = 1,\ldots,4$. Ако преуредимо ове матрице тако да могу бити подељене у унутрашње (енг. inner, II), које садрже само локалне променљиве, спољашње (енг. outer, OO) и унутрашњо-спољашње (IO, OI) делове, системи тада изгледају као што је приказано на Слици 2.2 (A, x и b су преуређени K_i, u_i и d_i респективно, $i \in \{1,2,3,4\}$).



Слика 2.2: "преуређени" систем једначина који описује понашање наше конструкције.

Добија се еквивалентни систем једначина

$$A_{II} \cdot x_I + A_{IO} \cdot x_O = b_I \quad (1)$$
$$A_{OI} \cdot x_I + A_{OO} \cdot x_O = b_O \quad (2)$$

Formatted: Russian (Russia)
Formatted: Russian (Russia)
Formatted: Russian (Russia)
Formatted: Russian (Russia)
Formatted: Russian (Russia)
Formatted: Russian (Russia)
Formatted: Russian (Russia)

Након замене x_I из (1) и (2) и примене очигледних трансформација, добијамо:

$$(A_{OO} - A_{OI} \cdot A_{II}^{-1} \cdot A_{IO}) \cdot x_O = b_O - A_{OI} \cdot A_{II}^{-1} \cdot b_I \quad (3),$$

где је $A_{OO} - A_{OI} \cdot A_{II}^{-1} \cdot A_{IO}$ Шуров комплемент.

Ако уведемо ознаке $\bar{A} = A_{OO} - A_{OI} \cdot A_{II}^{-1} \cdot A_{IO}$ и $b' = b_O - A_{OI} \cdot A_{II}^{-1} \cdot b_I$, имамо следећи систем једначина:

$$\bar{A} \cdot x_O = b' \quad (4).$$

У (4) утицаји свих локалних променљивих су елиминисани. Сличан принцип може бити примењен на било коју подматрицу коју бисмо добили ако бисмо решили да матрицу K и даље делимо рекурзивно. Сада се могу комбиновати сви Шурови комплементи „малих" матрица и вектори на десним странама, и тако добити нови систем једначина, чија је матрица комплетирани Шуров комплемент, и који може бити решен, на пример, коришћењем директних метода. На овај начин смо израчунали вектор непознатих x_O и можемо заменити његове вредности у једнакости (1).

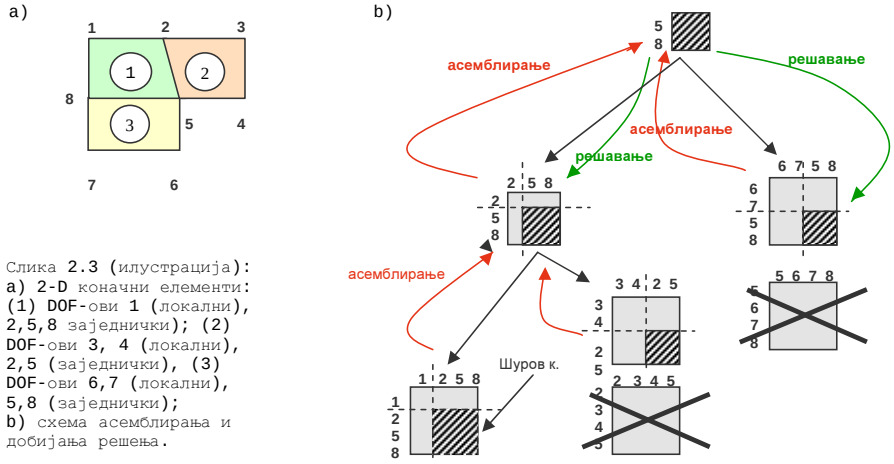
Из (1) сада можемо израчунати и остатак решења - x_I:

$$x_I = A_{II}^{-1} \cdot b_I - A_{II}^{-1} \cdot A_{IO} \cdot x_O \quad (5).$$

3. Рекурзивна дисекција за p-FEM

По завршетку FEM дискретизације и израчунавања матрице крутости, прелази се на решавање. Међутим, ни итеративне и директне методе за решавање глобалног система - као што је Гаус (због велике сложености и ограничених могућности за паралелизацију) - нису пожељне, па је неопходан алтернативни приступ. Погледајмо како ND може бити примењена на ове елементе.

Идеја је да после рекурзивних подела, као што је описано у [2], поставимо мали систем једначина за сваки поддомен, да се системи решавају одоздо навише, док се, уједно, сукцесивно елиминишу локалне променљиве (рачунају се Шурови комплементи). Ово, заправо, значи да смештамо израчунате матрице крутости елемената на дно хијерархије тј. у листове окталног стабла - које је погодна хијерархијска структура због могућности субструктуирања тродимензионалног простора - на тај начин да се у сваком листу или чува један елемент (његови подаци, укључујући матрицу крутости и вектор оптерећења), или он остаје „празан“. Непознате у систему, назване *степени слободе* (енг. degrees of freedom, DOF), су смештене и у листовима и у унутрашњим чворовима на тај начин да родитељски чворови чувају DOF-ове заједничке за своје синове, при чему су неки од њих сада проглашени локалним за тог родитеља (*LDOF-ови*) а остали остају као заједнички. У [2] је напоменуто да је главна предност добијена хијерархијском организацијом ND преко окталних стабала чиста вертикална комуникација међу родитељским чворовима и њиховим синовима (навише за слање Шурових комплемената за асемблирање и наниже за примање израчунатих решења).



Слика 2.3 (илустрација): a) 2-D коначни елементи: (1) DOF-ови 1 (локални), 2,5,8 заједнички); (2) DOF-ови 3, 4 (локални), 2,5 (заједнички), (3) DOF-ови 6,7 (локални), 5,8 (заједнички); b) схема асемблирања и добијања решења.

Штавише, узимајући у обзир сложеност ND [2] која је $O(n^{3/2})$, где је n број непознатих у систему, можемо закључити да је много ефикаснија од претходно поменутих директних метода за решавање. Још једна предност, када вршимо поређење са Гаусом и сличним методама, може бити уочена у случају када се параметри коришћени за рачунање матрице крутости мењају, у ком случају једино рачунања везана за измењена подстабла морају бити обављена испочетка. Међутим, још увек одговор модела за рачунање није довољно брз колико је неопходно у случају веома честих освежавања резултата у интерактивним апликацијама. Због тога се морају предузети даљи кораци - и то не само уобичајена паралелизацију (пошто стабла нису најефикасније решење, као што су Мински (Minsky) и други предочили у [4], а и детаљније је објашњено у следећој глави, већ и стратегију за уравнотежавање оптерећења да би се решио проблем скалабилности. Ово је омогућено посредством техника дистрибуираног програмирања заснованих на TCP/IP протоколу за комуникацију и имплементираних преко позива MPI библиотеке, као и паралелизације петљи путем OpenMP директива. Ови кораци су предмет истраживања и главна тема наредног поглавља.

III : Паралелизација

1. Проблем и идеја

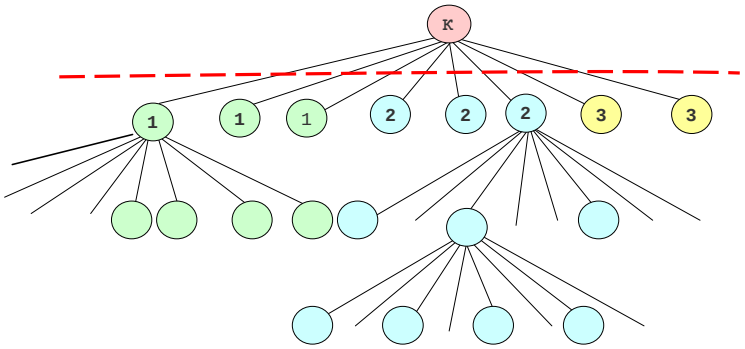
Када смо одлучили да применимо неку стратегију за паралелизацију, прво треба сагледати за апликацију који делови (секвенцијалног) програма могу бити паралелизовани. Ово треба пажљиво сагледати, пошто, очигледно, постоје предуслови за обављање различитих задатака (Шурови комплементи на одређеном хијерархијском нивоу морају бити израчунати пре Шурових комплемената на вишем нивоу дрвоидне структуре, тако да мора постојати начин да одреди редослед извршавања различитих послова. Пре свега, мора бити доступан механизам за препознавање независних послова и, када њихове зависности од других послова буду разрешене, начин да они буду процесирани од стране неких (изабраних) процеса.

Осим чињенице да уколико су задаци генерисани *динамички* (тј. за време извршавања апликације) и дистрибуција задатака процесорима мора да се одигра такође за време извршавања, постоје и други разлози из којих би се могло одлучити за *динамичко уравнотежавање оптерећења*. Наиме, супротно од великих (mainframe) рачунара чији су делови обично хомогени, умрежени персонални рачунари и кластери радних станица се најчешће састоје од хетерогених машина. У том случају, не може се лако предвидети колико су и када поједини процеси на различитим машинама на располагању. Супротно од *статичке дистрибуције оптерећења* (које користи само a priori познате чињенице о могућностима различитих процеса), код динамичког уравнотежавања узимају се у обзир и ове разлике у доступности појединих процеса. Међутим, пошто компоненте које учествују у реализацији динамичког уравнотежавања користе рачунарске ресурсе и тако узрокују кашњења, главни изазов динамичког уравнотежавања је управо проналажење решења са најповољнијим односом између кашњења и самог квалитета уравнотежавања које би требало да убрза извршавање. [7].

Уравнотежавање обима посла међу процесима је потребан, али, нажалост, не довољан захтев да би се смањило време чекања појединих процеса. Након што је завршено са разлагањем, као што је поменуто претходно у тексту, резултујући задаци нису сви спремни за извршавање у исто време. Оно што се подразумева под добром стратегијом је да је обезбеђено да су обрада задатака и интеракција међу процесима у свакој фази извршавања паралелног програма добро избалансирани.

Проблем који се јавља за дрвоидну структуру - симултано асемблирање матрица и рачунање Шурових комплемената на сваком нивоу - слично проблему који су изнели Мински и други у [4] са рачунањем суме 2N бројева помоћу N процеса - је да се број активних процеса смањује осам пута са сваким нивоом вишим за један у хијерархији (у обради стабла одоздо навише), тако да што смо ближе кореном (нултом) нивоу, више иницијално покренутих процеса проводи време чекајући.

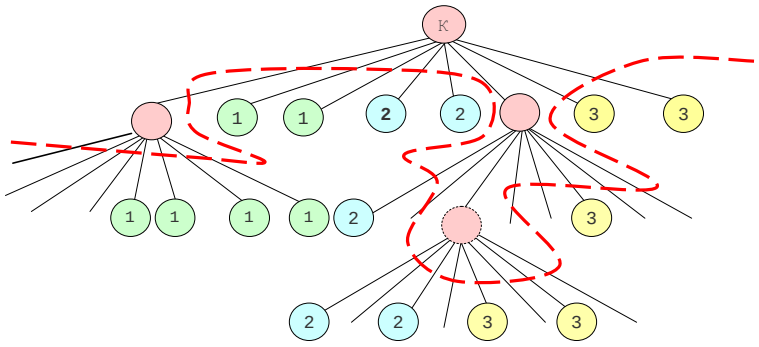
Штавише, ако покушамо да „вертикално“ разложимо стабло, претпоставка да



Слика 3.1: К - корен , 1, 2, 3 - подстабла додељена процесима 1, 2, 3 респективно.

је стабло добро балансирано нас може навести на закључак да је довољно да га „пресечемо“ на одређеном нивоу, доделимо задатке из подстабала која почињу на том нивоу различитим процесима и нумерички обрадимо та подстабла независно једно од другог (Слика 3.1). Напротив, да бисмо постигли жељену скалабилност, морамо да користимо софистицираније стратегије уравнотежавања оптерећења.

На неким архитектурама хардвера, као што су системи са дистрибуираном и дељеном меморијом, најбоље је комбиновати вишенитно (multithreading) и дистрибуирано програмирање, тако да у нашем приступу користимо оба. Идеја је да се процени целокупан посао на почетку и издели стабло на више од једног нивоа, ако је потребно, нпр. као што је приказано на Слици 3.2, да би се што боље избалансирао посао међу различитим процесима.



Слика 3.2: К - корен, 1, 2, 3 - подстабла додељена процесима 1, 2, 3 респективно.

Процена количине посла би требало да буде лака за израчунавање, тј. заснована на некој једноставној формули.

Наредни корак био би одлучивање о начину пресликавања. *Пресликавање* је механизам којим се задаци додељују процесима за извршавање. Добро пресликавање би требало да максимизира конкурентност процеса тако што додели независне послове различитим процесима и, у исто време, минимизира укупно време извршавања тако што обезбеди да су различити процеси доступни за извршавање послова зависних од других послова чим су исти постану разрешени поменутих зависности, као што би требало и да минимизира интеракцију међу процесима тако што додели послове са великим степеном међусобне интеракције истим процесима. Још један од циљева био би да су сви процеси заузети послом све време. Нажалост, испоставља се да су ово, најчешће, конфликтни циљеви.

“На пример, најефикаснија комбинација разлагања и пресликавања јесте један задатак мапиран на један процес. Тако процеси не чекају један други и не губи се време на њихову међусобно интеракцију, али исто тако се не постиже ни најмање убрзање.” [7]

Иако је степен конкурентности одређен разлагањем, заправо пресликавање је оно што одређује који степен те конкурентности је заиста искоришћен и колико ефикасно када се примени на одређени проблем.

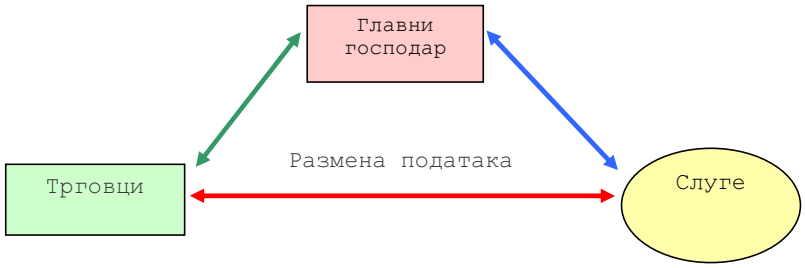
Када говоримо о *паралелном моделу*, заправо мислимо на начин да се структуира паралелни алгоритам тако што се изабере начин разлагања и пресликавања и примене одређене стратегије да би се смањила међусобна интеракција процеса [7].

Да бисмо предупредили уобичајено загушење код опслуживања захтева у случају коришћења само једног господара (енг. master) процеса, коришћен је модификовани централизовани динамички модел за уравнотежавање, где неколико господара одговарају на захтеве слуга (енг. slave) процеса. Наиме, имплементиран је концепт са два нивоа за обраду захтева од слуга, са једним тзв. главним господарем на првом нивоу и неколико под-господара, тзв. трговаца, на другом нивоу.

Овде главни господар на почетку додељује приближно једнаке делове посла трговцима. Када су подскупови задатака (иницијално) дистрибуирани међу господарима са оба нивоа (штавише, ови процеси су у могућности да генеришу сопствене редове (енг. queues) задатака за извршавање), главни господар опслужује захтеве слуга за новим пословима тако што их упућују на доступног, насумично изабраног трговца. Трговци такође постају директан интерфејс слугама и управљају већином потребне размене података са њима. И поред тога, главни господар остаје такође директан интерфејс за слуге, тако да може примати и даље њихове захтеве за новим задацима (Слика 3.3). Такође и управљање разменом података остаје одговорност главног господара у случају оних послова који, због свог обима, нису додељени трговцима, већ су остали у надлежности главног господара. Захтеви за задацима и преносима података могу бити иницирани или од стране процеса примаоца, или процеса пошиљаоца, као што је објашњено у наредним поглављима.

Паралелно извршавање задатака од стране слуга је оптимизовано помоћу OpenMP директива за паралелизацију петљи, како би се убрзала локална израчунавања.

У међувремену, идентификаторе међузависности задатака, тј. низове стринговних идентификатора, чија веза са дрвоидном структуром података је објашњена у тачки 3 овог поглавља, чувају главни господар и трговци за све задатке везане за њима додељене чворове и подстабла, респективно. Када су зависности од других задатака разрешене за задатак који је додељен трговцу, та информација се прослеђује главном господару да би се освежили и његови подаци, све док сви послови у реду који садржи послове главног господара нису обављени.



Слика 3.3: Модификована схема господар-слуга.

2. Структура програма

Реализовани паралелни програм састоји се од више инстанци серијских програма који међусобно комуницирају путем позива MPI библиотеци. Позиви који су коришћени могу бити грубо подељени у оне који се користе да иницијализују, управљају њоме и, напокон, окончају комуникацију (ови позиви се користе за започињање комуникације, одређивање колико процеса се користи, креирање подгрупе процеса и идентификовање који процес извршава коју инстанцу програма), као и оне који реализују саму комуникацију међу процесима (ови позиви, познати као *комуникацијске операције* од тачке до тачке (енг. point-to-point) представљају различито имплементиране операције слања и примања података).

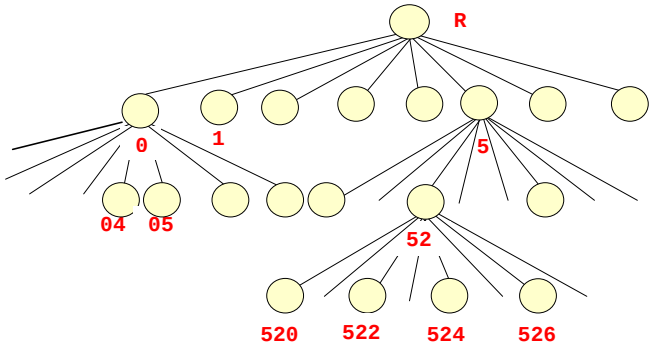
Програм је писан у програмском језику C, тестиран је у Linux-у, компајлиран помоћу *mpicc-а* - Open MPI омотача (енг. wrapper) C преводиоца (са базним gcc преводиоцем). Флег (flag) "-fopenmp" је потребан gcc-у за препознавање OpenMP директива.

У наредним тачкама овог поглавља биће представљена структура свих процеса (главног господара, трговаца и слуга). Нагласак је на томе како се дрвоидна структура иницијализује појединачно за сваки процес у складу са улазним подацима, а онда како се послови праведно дистрибуирају трговцима и, коначно, како је целокупна комуникација имплементирана, водећи брзом добијању коначног резултата – вектора решења система једначина.

3. Обрада улазних података

Подаци за сваки чвор чувају се у структури која садржи поља: size (димензија матрице крутости), lsize (величина вектора локалних степени слободе – *LDOF-ова*), ssize (димензија Шуровог комплемента), op (приближан број операција), показивачи (DOF и LDOF) на вектор DOF-ова and LDOF-ова, показивач на врсте MAT (матрице крутости), показивач на d (вектор оптерећења), ident (стринговни идентификатор чвора) и показивач на синове чвора.

Сваки чвор има свој јединствен стринговни идентификатор. Почевши од синова кореног чвора окталног стабла и обилазећи га наниже до листова, сви чворови имају јединствене бројеве који их разликују од осталих синова истог родитеља (из интервала [1,8], који су им додељени с лева на десно). За све осим кореног чвора (означеног са "R") и његових синова, идентификатор чвора добијен је конкатенацијом идентификатора родитеља са карактерском репрезентацијом претходно поменутог броја (видети пример на Слици 3.4).



Слика 3.4: Додељивање идентификатора (ident) чворовима у окталном стаблу.

Како бисмо иницијализовали октална стабла и попунили их подацима, морамо парсирати улазни фајл, где су смештени сви потребни подаци на следећи начин:

1) За листове:

MAT <идент. чвора> <бр. DOF-ова> <вредности DOF век.> <ел. матрице> <ел. век. опт.>

2) За све чворове (који имају бар један LDOF):

DOF <идент. чвора> <бр. LDOF-ова> <вредности LDOF век.>

Одмах након утврђивања да ли се ред у фајлу односи на "MAT"- или "DOF"-податке, требало би да буде алоцирана меморија за чворове окталног стабла на путањи од корена па до чвора чији је идентификатор node id, тј. све чворове чији су идентификатори префикси стринга node id. Онда се учитавају и остали подаци и попуњавају одговарајућа поља у окталном стаблу. Процес главни господар не чува нити елементе матрице крутости, нити вектора оптерећења за своје стабло. Трговци, након што су им додељена подстабла за даље старање о њима, чувају горе поменуто само ако је то потребно, односно за оне чворове који припадају управо њима додељеним подстаблима.

На тај начин, у случају "MAT" улазних линија и главни господар и трговци чувају: size са бројем степени слободе, вектор DOF са вредностима елемената, а само трговци чувају показивач на врсте матрице MAT са елементима матрице крутости и показивач на елементе вектора оптерећења, d (природно, само уколико

је актуелни node id идентификатор чвора из неког од конкретно њима додељеног подстабла). У случају "DOF"-линија, и главни господар, и трговци чувају lsize са бројем LDOF-ова, а трговци чувају вектор DOF са вредностима DOF-ова ако је node id идентификатор чвора баш из њима додељеног подстабла.

4. Дистрибуирање задатака

4.1 Иницијализација

Генерално, проблем представља проналажење разумног компромиса између, с једне стране, постизања веће тачности и коришћења одређене количине података како бисмо одредили оптерећење појединих процеса и, с друге стране, кашњења које је узроковано тим мерењима, израчунавањима и слањима података о оптерећењима [5]. Најчешће информације које се користе при дистрибуирању посла су оне о сложености потребних израчунавања и цени комуникације за сваки посао [5]. Због тога цена зависи од неколико аспеката, нпр. броја итерација у петљама датог алгоритма или количине процесираних података које треба послати [5].

Због тога бирамо једноставну али смислену формулу за мерење потенцијалног оптерећења – укупан број аритметичких операција у израчунавању Шуровог комплемента (алгоритам парцијалне Гаусове елиминације) за свако подстабло.

Одмах након што попуни своје октално стабло подацима, главни господар анализира ту дрвоидну структуру према ту чуваним димензијама матрица крутости и тако израчунава укупан број операција које се извршавају у Гаусовој елиминацији за сваки чвор и подстабло. Оно што се зна из улазних података је све о DOF-овима и LDOF-овима у листовима октalног стабла, као и све о LDOF-овима за све чворове у стаблу који их имају. Да бисмо извршили асемблирање и израчунали претходно поменути број елиминација, морамо да, респективно, преуредимо постојећи DOF вектор (за листове, такође, и врсте и колоне матрице крутости и вектора оптерећења) као и да одредимо елементе и величине DOF вектора.

За листове, ово друго изводимо тако што прво одредимо пермутације врста и колона матрице крутости (или елемената вектора оптерећења и DOF-а) тако да су уређени на начин представљен на Слици 2.2 и онда према њима заменимо елементе колона и показивача на врсте (или само елементе у случају вектора). Алгоритам за унутрашње чворове је аналоган, осим што вршимо пермутације само на елементима DOF вектора. Пермутације одређујемо тако што поредимо елементе DOF и LDOF вектора, мењамо бит знака елемената DOF-а који су уједно и елементи LDOF-а (тј. локални су за чвор) и уређујемо вектор са пермутацијама DOF-ова (или матрице и вектора) на тај начин да они негативни (тј. „означени", чији је бит знака промењен на 1) долазе на почетак.

Елементи и величине DOF вектора за унутрашње чворове се рачунају рекурзивно, асемблирањем DOF-ова синова, и њиховим преуређивањем на, већ познати, „inner/outer"-начин. Сада када имамо тачне информације о величинама DOF и LDOF вектора (уједно и матрице крутости и вектора оптерећења), можемо проценити посао потребан за сваки чвор.

Као што је приказано у псеудокоду на Слици 3.5, ако бројимо операције у

```
/* Izracunaj Schur-ov komplement */
 for  (sve lokalne DOF-ove, sa indeksima i) {
        for  (j = i+1; j < dimenzije matrice; ++j) {
                coeff = matrix[j][i] / matrix[i][i];                    (1)
                 for  (k = i+1; k < dimenzije matrice; ++k) {
                        matrix[j][k] -= matrix[i][k] * coeff;           (2)
                 }
                 d [j] -= d [i] * coeff;                                (3)
        }
}
```

Слика 3.5: (1) једно дељење у спољашњој петљи, (2) једно множење и једно одузимање у унутрашњој петљи, (3) једно множење и једно одузимање у спољашњој петљи

Гаусовој елиминацији узимајући у обзир само основне аритметичке операције, приметићемо три у спољашњој и две у унутрашњој for-петљи. Тако да би укупан број операција могао бити процењен следећим збиром:

Formatted: Serbian (Cyrillic, Serbia)

Formatted: Serbian (Cyrillic, Serbia)

Formatted: Serbian (Cyrillic, Serbia)

$$br\_op = 2 \cdot \sum_{i=1}^{n} (m-i)^2 + 3 \cdot \sum_{i=1}^{n} (m-i)$$

$$\sum_{i=1}^{n} (m-i)^2 = \sum_{i=1}^{n} m^2 - 2 \cdot \sum_{i=1}^{n} m \cdot i + \sum_{i=1}^{n} i^2 = nm^2 - 2 \cdot m \cdot \frac{n \cdot (n+1)}{2} + \frac{n \cdot (n+1) \cdot (2n+1)}{6} =$$

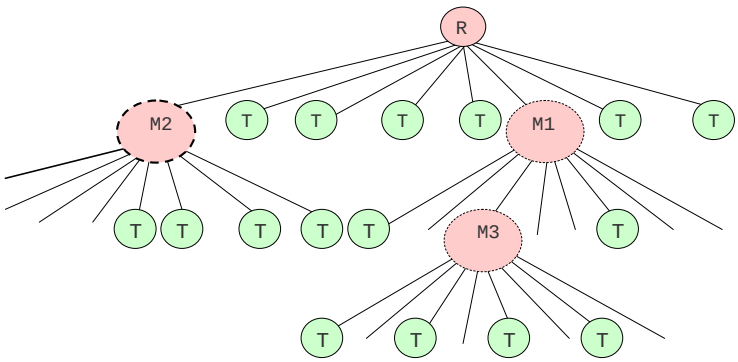
$$= nm^2 - m \cdot n \cdot (n+1) + n \cdot (n+1) \cdot \frac{(2n+1)}{6}$$

$$\sum_{i=1}^{n} (m-i) = \sum_{i=1}^{n} m - \sum_{i=1}^{n} i = nm - \frac{n \cdot (n+1)}{2}$$

$$br\_op = 2nm^2 - 2m \cdot n \cdot (n+1) + n \cdot (n+1) \cdot \frac{(2n+1)}{3} + 3nm - \frac{3n \cdot (n+1)}{2}$$

Укупан број операција у подстаблу се сматра *„тежином“* за сваки корени чвор тог подстабла, односно грубом проценом количине посла који треба обавити. Главни господар одмах складишти корени чвор целог октално стабла у свој ред послова. Касније, обилази чворове са директном граном ка кореном чвору и пореди израчунати број операција за сваког од њих са *граничном вредношћу* [3]. Овде је граница постављена као просечан број операција које треба да изврши један трговац узимајући укупан број операција на овом нивоу хијерархије као укупан посао. Ако максималан број операција за све до сада посећене чворове није већи него овај просек, главни господар је једино одговоран за корени чвор целог стабла [3].

У супротном, он чува идентификатор који одговара чвору са највећим бројем операција у низу задатака које треба сам да обави и испутује следећи ниво хијерархије за подстабло које има тај чвор за корен. У том случају, у следећем кораку узимамо у обзир све чворове из претходног корака (осим оних који су већ у надлежности главног господара) као и додатне, односно синове „најтежег“ чвора из претходног корака. Граница за нови подскуп је, поново, постављена као просечна количина посла коју треба доделити једном трговцу узимајући број операција које треба извршити за нови подскуп као укупан број (погледати пример на Слици 3.6).



Слика 3.6: Обилазак стабла: почевши од корена стабла (*R*) пронађено је неколико чворова (*M1*, *M2* and *M3*) са тежинама које прелазе границу, па вршимо даље разматрање док тежине свих чворова и подстабала респективно не буду испод границе и, самим тим, спремни да буду додељени различитим трговцима (означени са T).

```
/*
        LIMIT = 0 – limit za broj operacija za svakog trgovca
         Q – cvorovi za koje je zaduzen glavni gospodar
         ST - cvorovi (podstabla) koje treba dodeliti trgovcima
 */

spread (struct octree *next)
{
         node = next;

        smesti_u_Q (node);

        if  (dati cvor nije list) {
         for  (sve sinove) {
          if  (sin[i] nije prazan cvor) {
                  smesti_sina[i]_u _ST ( );
                  dodaj_na_LIMIT_broj_operacija_u_podstablu_sina[i]( );
           }
         }
         }

         node = maksimalan (po broju operacija) cvor u ST, indeks mu je j;

         sum = 0;
         if  (br. operacija u podstablu > (LIMIT / broj_trgovaca)) {
                  za ST[j] je oznaceno da nije vise u ST-u;
                  oduzmi_broj_operacija_cvora_ od_LIMIT-a( );
                  spread  (node);
        }
}
```

Слика 3.7: Обилазак у ширину окталног стабла при том чувајући неке од задатака у реду послова главног господара (Q) а остатак у реду ST за даље дистрибуирање трговцима.

Ова процедура се понавља – обилазак у ширину (енг. breadth-first), додељивање чвора са максималном ширином главном господару и затим разматрање његових синова уместо њега – све док за сам чвор са максималном тежином не проценимо да не прелази предложену границу (Слика 3.7). У том тренутку, препознат је подскуп чворова који прелазе границу потребног посла, њихови идентификатори су смештени у низ послова које треба да изврши главни господар, а поскуп идентификатора који представљају подстабла која треба доделити трговцима је спреман за дистрибуирање. На Слици 3.6 је приказано октално стабло које се процесира све док сви чворови и одговарајућа подстабла не буду са тежином испод границе.

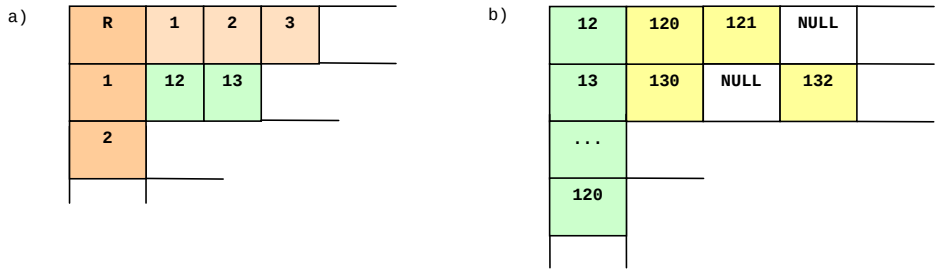
По завршетку обраде стабла према горе описаном алгоритму, главни господар покушава да избалансира посао међу трговцима. Он сортира низ чворова (корених чворова подстабала) у опадајућем поретку према количини посла и, идући кроз њега с лева на десно, додељује задатке (одговарајућа подстабла) појединачним трговцима, водећи рачуна да ниједан од њих не добије више него што граница дозвољава. Ако на крају неки чворови (подстабла) због свој тежине не могу бити додељени ни једном процесу, а да се, при том, не прекорачи лимит, они бивају безусловно, један по један, приписани трговцима са до тада најмањом количином посла. Алгоритам је детаљно изложен на Слици 3.8.

Коначно, главни господар је одговоран за дистрибуирање идентификатора послова (подстабала) трговцима [3]. Сваки трговац анализира све гране својих подстабала како би идентификовао међусобне зависности, односно редослед обраде међу својим чворовима стабла [3].

```
void divide ()
{
        for (sve trgovce) {
                TOTAL = 0;
                total2[i] = 0;
                for (sve cvorove u ST-u) {
                        if (cvor nije markiran kao da nije vise u ST-u) {
                                dodaj_na_TOTAL_broj_operacija_za_podstablo_iz_cvora( );
                                /* tolerancija je 5% od prosecne vrednosti */
                                if (TOTAL > (LIMIT / broj_trgovaca)*1.05) {
                                        oduzmi_od_TOTALa_operacije_podstabla_iz_cvora( );
                                }
                                else {
                                        dodeli_ST[j]_trgovcu( );
                                        ST[j]_oznaci_kao_da_nije_ vise_u_ST( );
                                }
                        }
                }
                 /* Vodimo evidenciju o ukupnom broju operacija za svakog trgovca */
                total2 [i] = TOTAL;
        }
        j = 0;
        for (sve cvorove u ST-u) {
                if (neki od cvorova [N] jos uvek nije dodeljen ni jednom trgovcu) {
                        j = indeks_trgovca_sa minimalnim_opterecenjem( );
                        dodeli_cvor_N_iz_STa_trgovcu[j]( );
                        proglasi_taj_cvor_dodeljenim( );
                        azuriraj_opterecenje_poslom_za_trgovca[j]( );
                }
        }
}
```

Слика 3.8: Дистрибуирање подстабала одговарајућим трговцима.

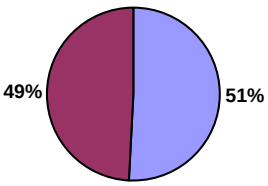
Сви идентификатори задатака су смештени у реду (енг. queue), заједно са информацијама о зависностима међу пословима [3] (погледати Слику 3.9 а) и б)). Очигледно су задаци везани за листове стабла независни од свих осталих, пошто се код приступа рекурзивне дисекције обрада врши одоздо навише и, на тај начин, родитељски чворови увек зависе од резултата елиминације својих директних потомака [3]. Када сви трговци генеришу своје локалне редове задатака, фаза иницијализације је завршена [3].



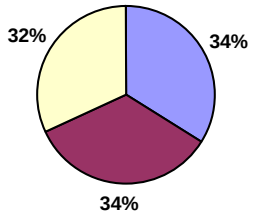
Слика 3.9: а) Ред задатака главног господара (са информацијама о зависностима); б) ред задатака трговца.

4.2 Резултати

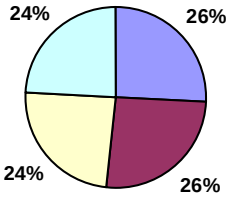
Добијени резултати су веома охрабрујући. Разматрали смо сценарио са дистрибуирањем 4171 задатка додељеног различитом броју трговаца. Иако су појединачни задаци били и већег и мањег обима, коначна расподела је прилично праведна и не очекујемо да би могле бити уочене веће разлике у расподели посла [3]. Детаљнији резултати су приказани на Слици 3.10.



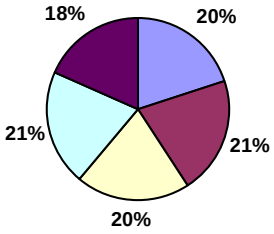
а)



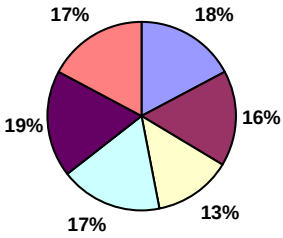
б)



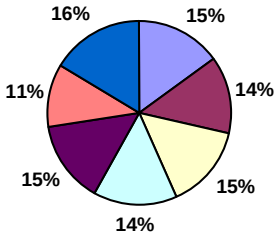
в)



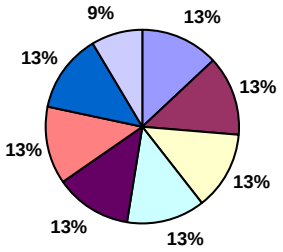
г)



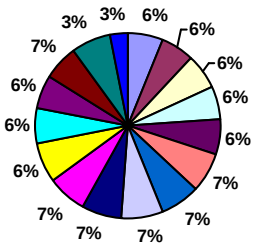
д)



ђ)



е)



ж)

Слика 3.10: Односи делова посла за : а) 2, б) 3, в) 4, г) 5,д) 6, ђ) 7, е) 8, ж) 16 трговаца.

4.3 Побољшања

Нека могућа побољшања разматраног приступа наведена су у [3]. До сада је само било разматрано уравнотежавање засновано на оптерећењу процесора послом. Пошто је за паралелне системе такође пожељно узети у обзир место података који су потребни различитим процесима и истражити утицај уравнотежавања не само количине посла, већ и уравнотежавања везаног за удаљеност података, чини се да је очигледан недостатак овог приступа то што се не придаје значај положају чворова приликом њиховог додељивања различитим процесима.

С једне стране, идеја се може илустровати једноставним примером са четири задатка, Зад1, Зад2, Зад3 and Зад4, које треба извршити на два процесора П1 и П2. За прва два задатка су потребни исти улазни подаци, као што су заједнички улазни подаци и за трећи и четврти. Ако је иницијално Зад1 лоциран на П1 и Зад3 лоциран на П2, могао би бити оптималан избор извршити Зад2 на П1 и Зад4 на П2, чак и ако Зад1 и Зад2 нису временски захтевна израчунавања у поређењу са Зад3 и Зад4. Наша схема за уравнотежавање би, на пример, поставила Зад1 и Зад3 на П1, а Зад2 и Зад4 на П2 да би се избалансирао обим потребних израчунавања. Међутим, било би далеко повољније сместити Зад1 и Зад2 на П1, а Зад3 и Зад4 на П2. Иако би П1 у неком тренутку чекао док би П2 био још увек заузет рачунањем, доделом задатака који имају потребе за сличним подацима истом процесору смањујемо потребну комуникацију (у систему са размењивањем порука) и тако смањујемо укупно време извршавања.

С друге стране, очигледна је зависност од величине блокова података и времена неопходног за њихово премештање, тако да, имајући на уму конфликт овог типа, мора се правити избор заснован на сопственим, специфичним, подацима и захтевима конкретне примене. У нашем случају, обилазећи стабло по ширини и испитујући нижи ниво хијерархије (идући у дубину) само за оне чворове са прекомерном тежином, осигурано је бар да је редукован број потенцијалних различитих подстабала за дистрибуцију, тако да се подаци не „растурају“ више него што је неопходно.

Ипак, могуће побољшање алгоритма могло би бити да се током доношења одлуке проверава и где се смештају подаци и да се избор, поред потребног броја операција, заснива и на цени комуникације међу процесима.”[3]

Оно што је, такође, вредно помена када је у питању контролисање кашњења узроковано међусобним интеракцијама процеса, осим бриге о томе где су смештени подаци, је техника преклапања израчунавања и интеракција. Подршка за ову технику је омогућена MPI *не-блокирајућим* (енг. non-blocking) позивима за размену порука. Наиме, обезбеђене су функције за слање и примање порука којима се враћа контрола корисничком програму пре него што су слање и примање, респективно, завршени. На

тај начин, програм користи ове примитиве да иницира интеракције и, одмах након тога, да настави са израчунавањима. На системима где су изражена кашњења, иницирање слања или примања је често ефективан начин да се маскирају одлагања узрокована комуникацијом. Кашњења имају обичај да буду буду велика на физички дистрибуираним системима, а релативно мала на мултипроцесорима са дељеном меморијом. У оба случаја, ако хардвер којим располажемо подржава конкурентно извршавање рачунања и трансфера порука, смањење кашњења због интеракције процеса може бити значајно. Употреба горе поменутих функција је једноставна, мада ипак захтева додатну бригу о алгоритамским корацима и структури кода.

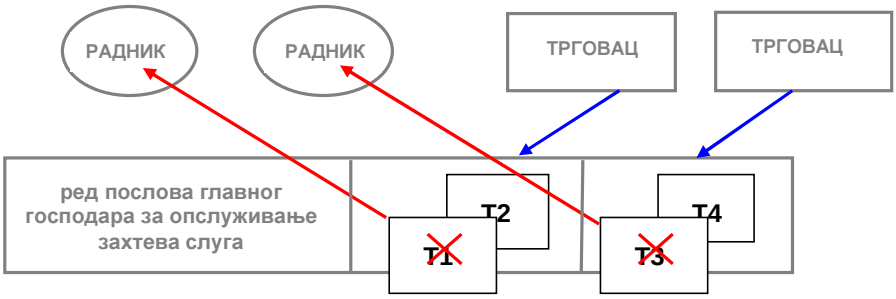
У развијеној апликацији је коришћена MPI-рутина за слање, MPI_Bsend. Она употребљава бафер који обезбеђује сам корисник да сачува све поруке које не могу бити одмах послате. Негативна страна овог приступа је то што је сам програмер одговоран за управљање бафером, односно, ако је у било ком тренутку простор у баферу недовољан да се оконча позив овој рутини, исход није дефинисан. Позивањем функција *MPI_BUFFER_ATTACH* и *MPI_BUFFER_DETACH* у програму, бафер стаје на располагање MPI-ју.[4]

Све операције слања за процесе налазе се у баферованом моду. Синхронизација са операцијама примања је постигнута тако што су коришћене блокирајуће (енг. blocking) рутине за примање (тј. нема враћања из функције пре него што је позвана комуникациона операција завршена). Значење *завршетка* за позив MPI_RECV је интуитивно - промењиве прослеђене MPI_RECV садрже поруку и спремне су да буду употребљене.

Детаљна комуникацијска схема је предмет наредне тачке.

5. Комуникација

Чим је иницијализација окончана, слуге могу тражити послове од главног господара. У исто време, трговци нуде први задатак из свог реда спреман за извршавање (који још увек није био процесиран) из свог реда, тако да је по задатак од сваког од њих доступан главном господару како би један (произвољан) од њих могао да га изабере (Слика 3.11) и, такорећи, упари са захтевом слуге - процес слуга је одмах упућен на изабраног трговца или трговце како би добио потребне податке. Слуга или директно рачуна Шуров комплемент, у случају да је примио матрице и векторе који припадају листовима стабла, или претходно асемблира допремљене матрице и векторе.



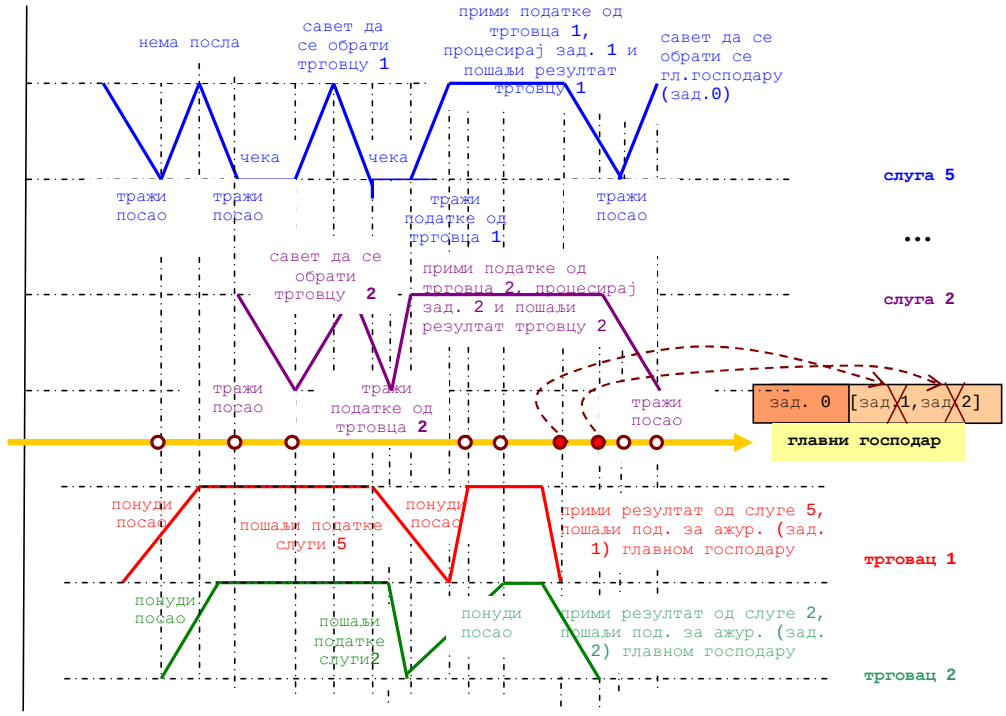
Слика 3.11: Главни господар бира задатак Ti од произвољног трговца и опслужује захтев процеса-слуге.

Када заврши са израчунавањима, слуга шаље натраг резултате трговцу од кога је добио задатак и податке, тако да сада трговац може проверити да ли су неке од зависности од тог задатка разрешене и да ли су неки додатни послови сада спремни

[4] извор: http://ci-tutor.ncsa.illinois.edu/

да буду понуђени. Штавише, он прослеђује ове информације и главном господару тако да и он сам може проверити зависности својих послова од овог.

Захваљујући освежавању зависности у реду послова, у неком тренутку су спремни за извршавање и задаци главног господара (Слика 3.12, слуга 5 је добио зад. 0, за који је одговоран главни господар). Пошто није вероватно да главни господар сам има све податке које је потребно послати за ова израчунавања - штавише, подаци су чувани у меморији трговаца – он је обавезан да пронађе ком трговцу или трговцима припадају ти подаци и да посаветује слугу да траже податке од одговарајућих. Када је завршено са израчунавањима, слуга шаље идентификатор завршеног задатка, овог пута – главном господару, поново ради освежавања његовог реда послова.



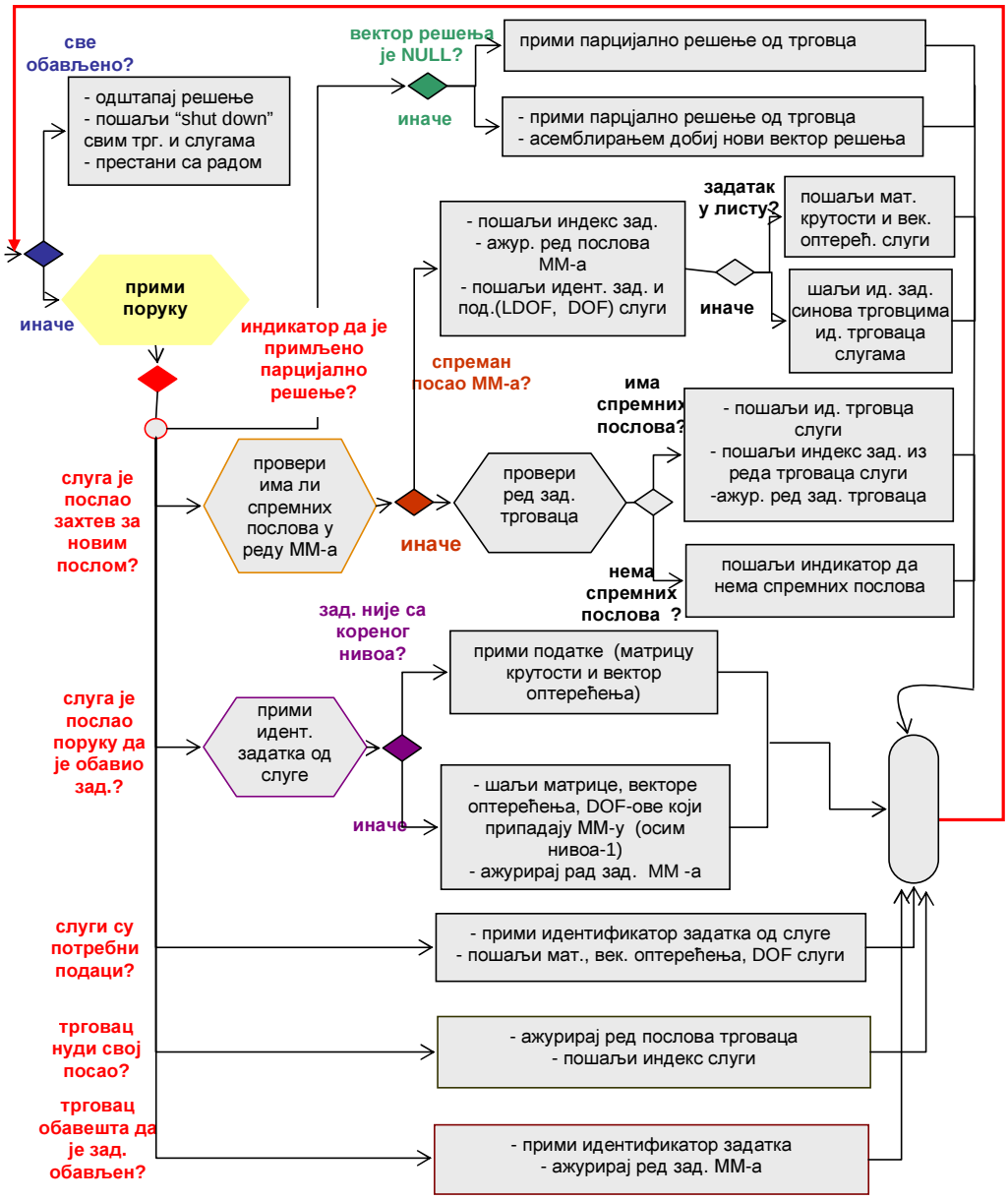
Слика 3.12: Упрошћена илустрација могуће временске скале решења (кружићи на линији главног господара представљају тренутке када други процеси покушавају да га контактирају); чим зависности од задатака 1 и 2 буду разрешене, задатак 0 из реда главног господара је спреман за обраду и послат слуги 5.

Рекапитулација целокупног комуникацијског шаблона би могла изгледати као што је приказано на Слици 3.13, а преглед процеса понаособ следи у наредној тачки.

Слика 3.13. Комуникацијски шаблон

5.1 Главни господар

Главни господар (енг. main master, MM) је имплементиран као процес са идентификатором нула. Као што је представљено на Слици 3.14, он је кључни координатор свих других процеса. Његова иницијална операција је примање: примање захтева за новим послом од слуге, новог доступног задатка или идентификатора завршеног посла од трговца, резултата израчунавања матрице крутости и вектора оптерећења који припадају чвору у његовој надлежности од слуге, инстанцу вектора решења од сваког трговца.

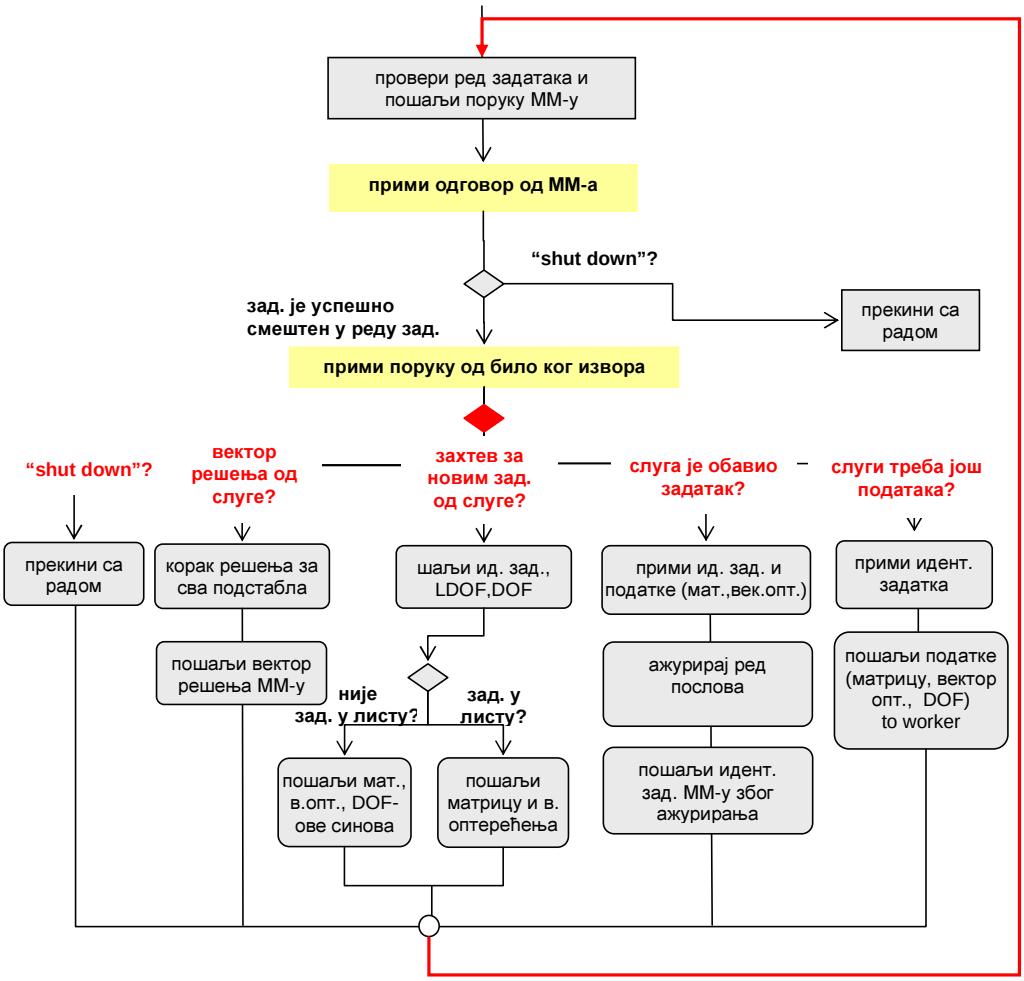


Слика 3.14: Ток комуникације главног господара.

5.2 Трговци

Основна улога *трговаца* је да управљају подацима који су потребни за обраду послова које обављају слуге, како се не би јавиле очекиване баријере код покушаја главног господара да одговори на све захтеве. Они и чувају све потребне податке, пошто су слуге имплементиране тако да су без сопствене меморије, па се резултати израчунавања Шурових комплемената од стране слуга за подстабла која су додељена трговцима тим истим трговцима и шаљу.

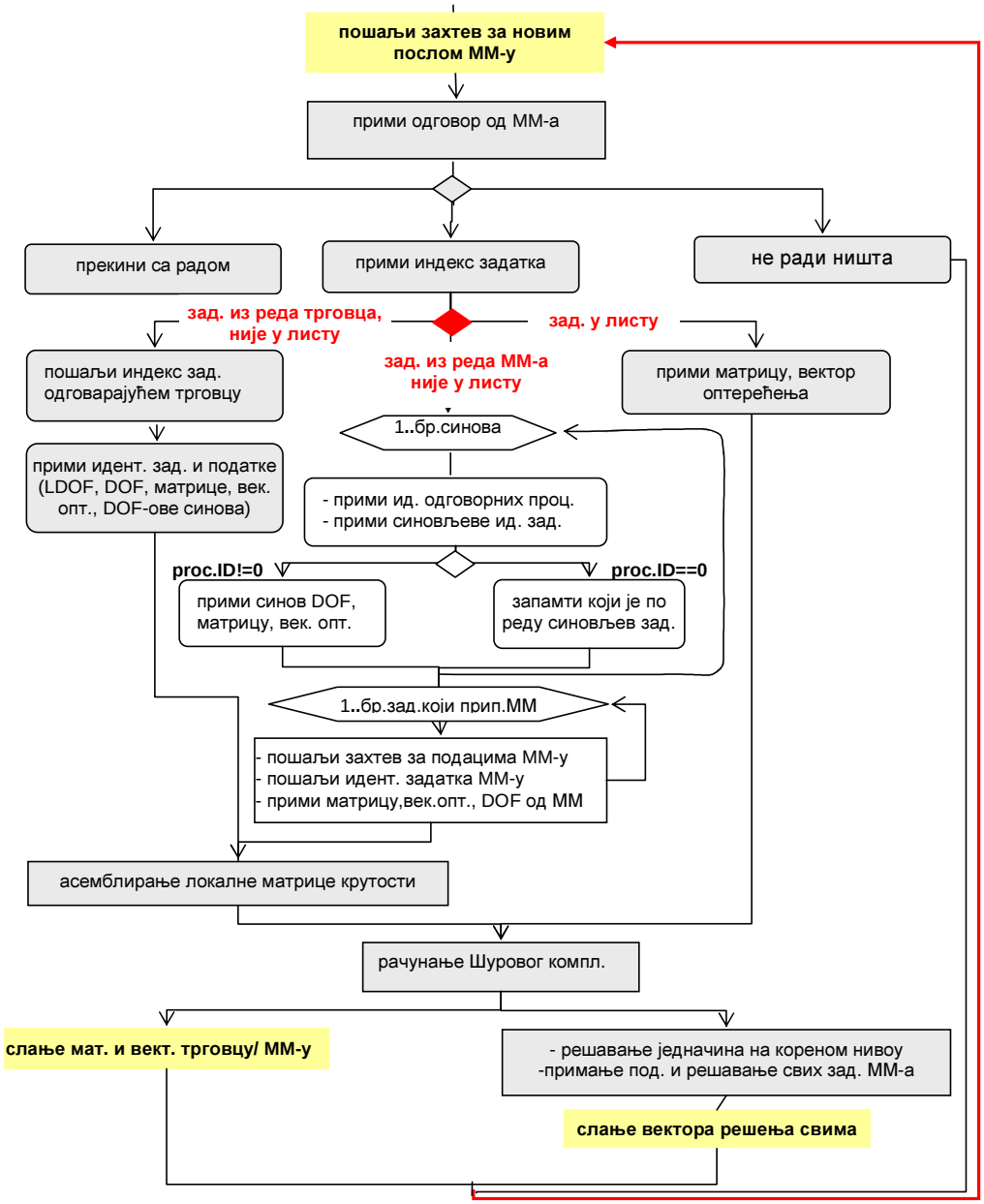
Њихова иницијална операција је примање: примање захтева за послом из његовог подстабла или захтева за додатним подацима (матрицом крутости и вектором оптерећења) у случају да су потребни слуги за обраду посла у надлежности главног господара, тј. задатка који је зависан од задатака трговаца. Даље, примање може обухватити добијање идентификатора посла за освежавање информација, као и резултата израчунавања, парцијална решења (векторе) од слуге (потребне у фази решавања за његова подстабла, као што је објашњено у наредној тачки), или захтев од главног господара да прекине са радом. Структура кода који извршава трговац приказана је на Слици 3.15.



Слика 3.15: Ток комуникације трговца.

5.3 Слуге

Слуга је имплементиран као процес без сопствене меморије, који упућује захтеве за пословима главном господару и прима идентификатор одабраног процеса (или више њих) од којих може добити Шурове комплементе и векторе потребне за корак асемблирања (Слика 3.16). Одабрани процеси укључују било ког од трговаца или самог главног господара.



Слика 3.16: Комуникацијски ток слуге.

Након асемблирања, слуга изводи кораке алгоритма парцијалне елиминације над резултујућим матрицом и вектором и шаље резултате истом процесу од кога је примио податке.

Систем на нултом (кореном) нивоу решавамо Чолески декомпозицијом. У случају да су веће количине података потребне за израчунавања на овом нивоу, може се приметити да, с обзиром на то да сви други процеси проводе време чекајући да се оконча ово израчунавање, слуга изабран да сам обави обраду на врху хијерархије представља препреку за добијање још значајнијег убрзања при извршавању програма.

Када је завршена обрада матрице на кореном нивоу, слуга претходно задужен за ово рачунање прима све матрице крутости и векторе оптерећења из меморије главног господара (међутим, не и матрице и векторе са првог нивоа хијерархије, јер се, због претходних израчунавања, већ налазе у „привременој“ меморији слуге) и изводи корак решавања за све системе за које је задужен главни господар. Исти слуга прослеђује парцијално решење свим трговцима за даља израчунавања инстанци вектора решења, које ће све бити асемблиране на крају од стране главног господара у једно, коначно, решење.

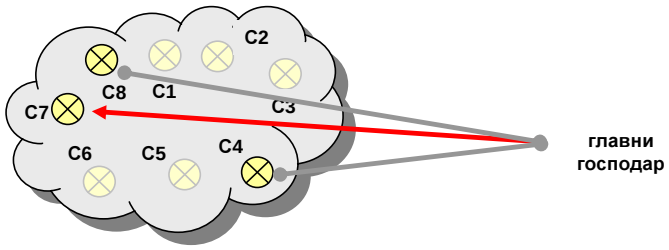
Закључује се да је иницијална операција слуге слање: слање захтева за послом главном господару, захтева за послом трговцу, захтева за подацима (матрицом крутости, вектором оптерећења, итд.) од трговца или више њих, резултата одговарајућем трговцу или главном господару, слање вектора решења свим трговцима.

Као што је већ поменуто, иницијална операција процеса господара на оба нивоа је примање. Делови дијаграма тока који представљају кључне тачке слања и примања за слуге и господаре, респективно, су обојени жуто на Сликама 3.14, 3.15 и 3.16.

Занимљив је резултат у вези са слугама, које или раде (обављају задатак) или не (чекају да им посао буде послат): ако проценимо суму свих временских интервала у којима је слуга активан, поделимо је укупним временом извршавања слуге и прикажемо графички резултате за 4171 елемената које је процесирало 4, 8 и 16 слуга (чије захтеве је опслуживало 4 трговца), исход изгледа као на Сликама 3.18 а), б) и в) респективно. Y-оса представља проценат (изражен у стотинама) укупног времена извршавања који слуге проведу извршавајући задатке.

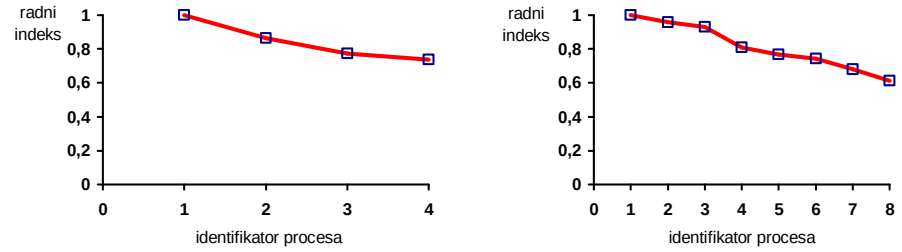
Због проблема са смањивањем броја независних послова и све већих задатака са сваким вишим нивоом у окталном стаблу, резултати мање обећавају него што би то био случај без поменутих зависности и задатака приближно исте величине, где би проценат био близу идеалних 100, за исти број слуга.

Овај, идеалан, проценат би био логична последица тога што главни господар увек бира произвољног трговца. Наиме, наша очекивања, која се, у исто време, поклапају са мерењима, су да ниједан од слуга није фаворизован када дође до одабира једног од њих и упућивања истог на трговца. Када слуге контактирају главног господара, успостави се конекција и сокети (енг. sockets) за ове слуге су отворени, главни господар бира произвољно извор од кога ће примити захтев. TCP-сокети/MPI-конекције између главног господара и осталих слуга, који су упослени у том тренутку, нису „активни“, али, према Open MPI спецификацијама, нису ни затворени (Слика 3.17).

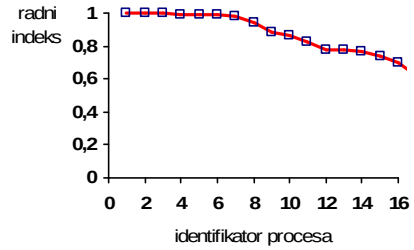


Слика 3.17: Слуга 4 (C4), 7 (C7) и 8 (C8) захтевају нови посао и главни господар бира C7 да од њега прими захтев; “упослене” слуге C1, C2, C3, C5, C6 не покушавају да пошаљу захтев/податке главном господару у овом тренутку.

Слика 3.18 показује, такође, да је отприлике пола процеса активно преко 80% укупног времена, због зависности међу пословима и повећавања величине задатака у обради стабла одоздо навише, што чини да слободни процеси чекају дуже пре него што изнова добију посао.



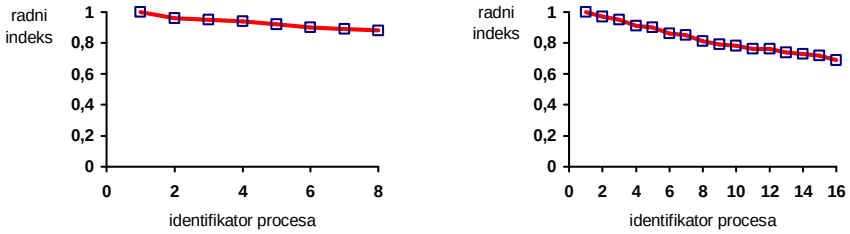
а) б)



в)

Слика 3.18: Тренд „радног" процента (у стотинама) за а) 4, б) 8 и в) 16 слуга (које опслужују 4 трговца) који процесирају 4171 задатак са међусобним зависностима.

Резултати за сценарио са 8 трговаца који опслужују 8 или 16 слуга за исти број процесираних елемената више обећавају. Они показују (Слика 3.19) да је око половине процеса слуга упослено више од 90% укупног времена.



а) б)

Слика 3.19: Тренд „радног" процента (у стотинама) за а) 8, б) 16 слуга (опслужених од стране 8 трговаца) који процесирају 4171 задатак са међусобним зависностима.

Интересантно питање је који је оптималан број слуга за ову апликацију и биће тема будућег истраживања. У вези са питањима скалабилности, када покрећемо велики број процеса (на пример слуга), не би требало да буде проблема ако је комуникација између процеса у паралелној апликацији релативно ретка. Наиме, иако је вероватно да ће Open MPI отворити више TCP сокета, неће отворити сокет за сваки MPI пир (peer) процес током иницијализације. Open MPI отвара сокете на захтев, тј. први пут када процес пошаље поруку пиру и постоји TCP конекција међу њима, аутоматски ће бити отворен нови сокет.

6. Обрада задатака

OpenMP директиве су искоришћене како би се оптимизовали делови кода, нарочито они које извршавају процеси-слуге – асемблирање и рачунање Шуровог комплемента – као и они који се односе на локална решавања система једначина за сваки чвор. У поменутим фрагментима кода употребљено је неколико угњеждених for-петљи, па се чини да је ту оптимална ова врста паралелизма. Ипак, неопходан је опрез са синхронизацијом нити које покушавају да приступе истим меморијским локацијама. OpenMP нуди неколико механизама за ову синхронизацију, нпр. критичне секције (енг. *critical sections*, које дозвољавају да само једна нит извршава спецификовани део изворног кода у одређеном времену) или атрибути као што је приватни (енг. *private*, који може бити коришћена да се спречи да нити деле специфициране променљиве).

Даље, Бернштајнови услови (енг. Bernstein's conditions) помажу да се идентификују могућности за паралелизацију петљи у коду. Нека су $П_1$ и $П_2$ два фрагмента кода. Бернштајнови услови [6] описују када су П1 и П2 независни један од другог и могу бити симултано извршени. За $П_i$, нека су I_i све улазне променљиве и O_i све излазне променљиве, за $i \in \{1,2\}$ и слично за $П_j$. Бернштајнови услови за $П_i$ и $П_j$ гласе:

$I_2 \cap O_1 = \emptyset$
$I_1 \cap O_2 = \emptyset$
$O_1 \cap O_2 = \emptyset$

Сада ће бити показано како све претходно поменуто може бити примењено на фрагменте кода у конкретној апликацији.

Пре свега, Шурови комплементи K_i се асемблирају сабирањем одговарајућих елемената матрица. Разматрањем коришћења неколико нити у овом кораку бисмо закључили да је, пошто двема нитима није дозвољено да ажурира исти елемент матрице K_{ijk} паралелно, неопходна максимална синхронизација. Таква синхронизација би значила заштиту сваког од Шурових комплемената критичном секцијом, која би, међутим, водила да скоро серијског извршавања кода. Псеудокод који илуструје овај сценарио је приказан на Слици 3.20.

```
/* Asembliranje lokalne matrice krutosti i vektora opterecenja */
#pragma omp parallel for private (j, k, perm)
for  (sve matrice krutosti sinova) {
        #pragma omp critical  {
                izracunaj_permutacije( );
                for (j =  lsize_sina[i]; j <  size_sina[i]; ++j) {
                        for (k =  lsize_sina[i]; k <  size_sina[i]; ++k) {
                                M[perm [j]][perm [k]] += K[i][j][k];
                        }
                        d[perm [j]] += d_sina[i][j];
                }
        }
}
```

Слика 3.20: Асемблирање оптимизовано OpenMP директивама (скоро серијска верзија).

Део кода који би требало да се извршава паралелно је тако и обележен, помоћу препроцесорских директива, што резултује у формирању нити пре извршавања те секције кода.

Овај проблем може бити решен употребом неколико критичних секција (по једна за сваку врсту или колону). На слици 3.21, употребљена је критична секција за сваку врсту.

```
/* Asembliranje lokalne matrice krutosti i vektora opterecenja */
#pragma omp parallel for private (j, k,)
for  (sve matrice krutosti sinova)  {

        izracunaj_permutacije( );

        for (j =  lsize_sina[i]; j <  size_sina[i]; ++j) {
                #pragma omp critical (j)
                {
                        for (k =  lsize_sina[i]; k <  size_sina[i]; ++k) {
                                M[perm [j]][perm [k]] += K[i][j][k];
                        }
                        d[perm [j]] += d_sina[i][j];
                }
        }
  }
```

Слика 3.24: Рачунање локалних решења оптимизовано помоћу OpenMP-а.

Друга могућност не укључује синхронизацију, али исход је мањи степен паралелизма због серијске обраде свих Шурових комплемената. Међутим, очекивано је да би и овај приступ дао задовољавајуће резултате у нашој апликацији. Наиме, могле би се користити све нити за обраду само једне K_i (Слика 3.22).

```
/* Asembliranje lokalne matrice krutosti i vektora opterecenja */
 for (sve matrice krutosti sinova)  {

        izracunaj_permutacije( );

        #pragma omp parallel for private (k)
        for (j =  lsize_sina[i]; j <  size_sina[i]; ++j)  {
                for (k =  lsize_sina[i]; k <  size_sina[i]; ++k) {
                        M[perm [j]][perm [k]] += K[i][j][k];
                }
                d[perm [j]] += d_sons[i][j];
        }
}
```

Слика 3.22: Комплетирање оптимизирано помоћу OpenMP (средња петља је паралелизована).

За парцијалну Гаусову елиминацију изабран је приступ са више нити које стартују пре друге од три угњеждене петље, као што је представљено на Слици 3.23. Када заврше са послом, слуге контактирају трговца како би послале Шурове комплементе који су потребни за послове на следећем (вишем) нивоу хијерархије.

```
/* Izracunaj Schur-ov komplement */

for (i = 0; i < lsize; ++i) {
        #pragma omp parallel for private (k, coeff)
        for (j = i+1; j < size; ++j) {
                coeff = M[j][i] / M[i][i];
                for (k = i+1; k < size; ++k) {
                        M[j][k] -= M[i][k] * coeff;
                }
                d [j] -= d [i] * coeff;
        }
}
```

Слика 3.23: Парцијална Гаусова елиминација оптимизована помоћу OpenMP-а.

Део кода који се односи на рачунање локалних решења, а који може бити оптимизован OpenMP директивама је приказан на Слици 3.24. Нити су форковане (енг. forked) пре него што се почне са извршавањем спољашње петље; на тај начин поједине елементе вектора решења рачунају различите нити. Међурезултати, тј. суме производа које треба одузети од елемената вектора десне стране система, су проглашени приватним за сваки интервал индекса који се процесира независно од других. Локална решења се касније шаљу наниже, синовима чвора, на даљу, рекурзивну, обраду.

```
/* Izracunaj lokalno resenje */

#pragma omp parallel for private (j, tmp)
for (i = lsize - 1; i >= 0; --i) {
        tmp = 0;
        for (j = i+1; j < lsize; ++j) {
                tmp += M[i][j] * sol_vec [DOF[j]];
        }
        sol_vec [DOF[i]] = (d[i] - tmp) / M[i][i];
}
```

Слика 3.24: Рачунање локалних решења оптимизовано помоћу OpenMP-а.

Осим описаних приступа паралелизацији петљи, може бити примењено да су могућа и даља побољшања у случају задатака већих по обиму - може се комбиновати OpenMP паралелизација са дистрибуирањем задатака различитим процесима како бисмо дејствовали и на нивоу симултаног извршавања блокова (петљи), и на нивоу паралелних процеса, респективно.

Ова идеја потиче, такође, из чињенице да су програми имплементирани тако да се извршавају само на системима са дељеном меморијом, на пример они који су паралелизовани искључиво помоћу OpenMP-а, ограничени доступним хардвером. Концептуално, парадигме које користе дистрибирану меморију, као што је MPI немају ограничења у вези са бројем процесора. Без сумње, не може се лако одговорити на питање када је дистрибуирано паралелно програмирање ефикасније од вишенитног програмирања. Међутим, када се ради о поменутој апликацији, пошто се OpenMP делови кода односе искључиво на петље које могу бити паралелизоване, коришћење искључиво малтитрединга у целој апликацији би резултовало у релативно великом проценту кода остављеног за извршавање у серијском моду, што би, даље, условило мању ефикасност при паралелном извршавању чак и ако би нешто већи број процесора био на располагању. Из тог разлога, одлучили смо се за описани, хибридни приступ.

IV : Закључак

У овом раду описан је модел динамичког уравнотежавања оптерећења за решавач система линеарних једначина заснован на принципу „рекурзивне дисекције“, чиме се дотакло питање скалабилности у вези са паралелизацијом хијерархијски организованих структура.

Испоставља се да је балансирање количине посла само по себи неопходан, али, нажалост, не и довољан услов за смањење времена чекања појединих процеса, углавном због постојећих зависности међу различитим пословима. Оно у шта смо покушали да се уверимо јесте да је обезбеђено да су обрада послова и интеракције међу процесима добро избалансирани у свакој фази извршавања паралелног програма. У вези са тим, проблем који се јавио за октално стабло - симултано асемблирање матрица и рачунање Шурових комплемената на сваком нивоу - се састојао у томе да се број активних процеса смањивао осам пута за сваки ниво којим смо били ближе корену, тако да је било неопходно коришћење софистициране стратегије за уравнотежавање оптерећења. Такође, трагало се за начином процене оптерећења који не узрокује непотребна кашњења, односно, заснован је на некој једноставној.

Осим тога, пошто управо пресликавање заиста условљава колика ће конкурентност бити и реализована од оне претпостављене након разлагања и, штавише, колико ефикасно за конкретан проблем, а и како је примећено да би у случају само једног процеса господара у централизованом моделу могло доћи до загушења при његовом покушају да одговори на захтеве свих слуга, примењено је „напредније“ господар-слуга пресликавање.

Након одабира начина разлагања и пресликавања, да би се паралелни модел учинио комплетним, потребно је било применити одговарајуће технике смањивања међусобних интеракција процеса. Овде је проучавано неколико начина, укључујући проверу локалитета података током одлучивања при пресликавању, или употребу механизма који омогућава да се израчунавања и комуникација извршавају истовремено (уколико је то исто и хардверски подржано), мада нешто од поменутог само као идеја за даље истраживање.

Поред тога, пошто је на неким архитектурама хардвера најпрофитабилније комбиновати мултитрединг и дистрибуирано програмирање, одлучено је да се користе оба. Наиме, оптимизације делова кода који се односе на обраду задатака су постигнуте помоћу OpenMP директива за паралелизацију петљи. Анализирано је неколико механизама паралелизације овог типа - са и без синхронизације нити. Овде, у случају обимних задатака (углавном оних близу нултом нивоу стабла), може се наметнути питање, да ли је вишенитно програмирање ефикаснији приступ од дистрибуиране паралелне обраде.

***

До сада добијени резултати обећавају много. Међутим, као што је наглашено у Уводу, циљ је више био испитивање предности појединих концепата у вези са паралелизацијом, него конкретни бенчмарк резултати, који ће, свакако, бити добијени током даљег рада на овој теми.

Када је разматрано оптерећење трговаца послом, тестирани су сценарији дистрибуирања 4171 задатка за различите бројеве трговаца у опсегу од 2 до 16. Закључено је да, иако су појединци задаци били различитих обима, коначна дистрибуција је праведна (разлике су 5-6 процената у најгорем случају) и, штавише, јављање већих разлика није очекивано.

По питању времена које покренути процеси проведу извршавајући задатке (не чекајући) у поређењу са својим укупним временом извршавања, закључено је да, због проблема са смањивањем броја међусобно независних послова и пораста обима појединачних задатака при обради одоздо навише, однос радног и времена чекања није увек близу идеалних 1, мада су, ипак, резултати много бољи него што је случај код „уобичајене“ паралелизације стабла (симултано извршавање задатака на сваком нивоу појединачно, редом идући одоздо навише). Видели смо да је, у неким случајевима, отприлике половина од укупног броја процеса активна више од 80%

свог укупног времена извршавања, или да је, што још више обећава, у неким другим случајевима, који укључују више процеса-трговаца, око половине слуга активно више од 90% свог времена.

Ови резултати доводе нас мало ближе дугорочном циљу - добијању рапидног одговора модела конструкције, као што је неопходно у апликацијама са корисничком интеракцијом. Ипак, има још питања која захтевају одговоре и постоје разлози због којих се очекује да би исход појединих поправки биле још боље перформансе.

***

Што се могућих побољшања тиче, прво је разлагање. „Разлагање података је моћан и често коришћен метод за постизање конкурентности у алгоритмима којима се оперише над великим структурама података. Код овог метода, разлагање израчунавања се обавља у два основна корака. У првом кораку, деле се подаци над којим се врше израчунавања, а у другом кораку ово партиционирање података је искоришћено како би се израчунавање изделило на мање задатке.”[7] У апликацији која је развијана, могућа је даља оптимизација у вези са обрадом задатака који су близу корену стабла, тј. оних релативно обимних, тако што се они прво изделе а онда и расподеле између неколико слуга, тако да буду симултано процесирани коришћењем и дистрибуираног рачунарства и механизама са више паралелних нити.

Штавише, за израчунавање на кореном нивоу, могле би се истражити и могућности примене технике разлагања података на неку од итеративних метода за решавање система, рецимо Јакобијеву (енг. Jacobi) методу, и пресликавања мањих проблема на претходно израчунати, оптимални, број слуга. Да би се извело ово паралелно израчунавање, матрица би могла бити или подељена међу процесима логички организованим у једнодимензиону мрежу од p појасева или у дводимензиону (са блоковима величине m x m = p). Може се поћи од тога да је довољно поделити је у p појасева. У том случају, сваки од p слуга био би одговоран за решавање у једном појасу. Ипак, неки од резултата морали би бити и размењивани успут. Ово би захтевало употребу механизма размене порука, како би добијене вредности (из других делова) биле доступне тренутно посматраном појасу. У случају Јакобијеве методе, слуги би могла бити потребна приближна решења других слуга из претходне итерације. У том случају, могле би се искористити предности колективних (енг. collective) MPI рутина као што је MPI_AllGather, којом се шаљу подаци од свих ка свим унутар неког изабраног подскупа активних процеса и којом је имплементиран најефикаснији алгоритам познат за ову радњу.

Након разлагања, друго питање које је остало отворено и предлог за даље истраживање је, као што је и раније поменуто, питање оптималног броја радника узимајући у обзир улазне податке. Тестирање апликације за различит број трговаца који одговарају на захтеве извесног броја слуга би могло показати да би чак и у овој, напредној, поставци, са господарима на два нивоа, могло доћи до загушења код комуникације са главним господаром у случају одговарања на захтеве превеликог броја процеса. У том случају, могло би бити разматрано увођење два „главна“ господара, при чему би сваки био задужен за различиту групу процеса (трговце, односно слуге).

Даље, пошто су дискутована могућа побољшања разлагања и пресликавања, изнова се разматра једна од претходно поменутих стратегија за минимизацију комуникације, а уз то се помињу и неке нове.

Како је, до сада, посматрано само уравнотежавање оптерећења засновано на оптерећењу процесора, а код паралелних система је пожељно проверити где су смештени подаци којима се служе различити процеси и истражити утицај уравнотежавања не само према обиму посла већ и заснованог на удаљености података, предложено је унапређење алгоритма дистрибуирања посла на тај начин да се узме у обзир и положај чворова за време њихове доделе различитим процесима.

Обиласком стабла по ширини и преласком на нижи ниво хијерархије само за оне чворове чија је тежина прекомерна, осигурано је бар да се подаци не „распршују“ више него што је неопходно.

С једне стране, потенцијално побољшање алгоритма и даље остаје проверавање локалитета података и одлучивање према цени и комуникације, и броја потребних операција. С друге стране, треба пазити да сложеност унапређеног алгоритма сама по себи не поништава побољшања иначе добијена провером ових локалитета.

Друга идеја за смањивање кашњења при размени порука (занимљива за неку каснију фазу развоја апликације), а у контексту дистрибуираног рачунарства, јесте смештање података који су потребни и трговцима и слугама на некој удаљеној локацији којој могу приступити оба типа процеса.

Исто тако, пошто у актуелној имплементацији слуге немају своју меморију, они бришу све коришћене податке чим заврше са једним задатком и шаљу натраг целе матрице крутости и векторе оптерећења одговарајућим трговцима. Још једна идеја за имплементацију слуга била би да они чувају све податке потребне за касније кораке решавања и шаљу трговцима само Шурове комплементе и нелокалне делове вектора оптерећења (који су потребни за асемблирање матрица и вектора оптерећења, респективно, на вишим нивоима хијерархије). За сада сматрамо да, иако би ово уштедело време потребно за слање у кораку решавања, условило би круту схему за слање задатака тачно одређеним слугама, па би водило ка могућем успоравању.

Напокон, иако је очигледно (из постигнуте праведне дистрибуције целокупног посла међу трговцима и високог процента времена које слуге проводе обављајући послове) да је значајно убрзање морало бити постигнуто, не може се још увек ништа одређено рећи о профитабилности описане стратегије када су ефикасност и убрзање програма у питању. Због тога ће конкретни бенчмарк резултати бити неизоставан део будућег рада.

Литература

1. George, J.A.: Nested dissection of a regular finite element mesh. SIAMJ. on Numer. Analysis, 10:345–363 (1973)
2. Ralf-Peter Mundani, Alexander Düster, Jovana Knežević, Andreas Niggl и Ernst Rank: Dynamic Load Balancing Strategies for Hierarchical p-FEM Solvers (биће објављено у Proc. of the 16th EuroPVM/MPI Conf., M. Ropo et al., eds., LNCS 5759, Springer, 2009, стр. 305–312)
3. Jovana Knežević, Ralf-Peter Mundani: Applying Dynamic Load Balancing Techniques to the Parallel p-Version FEM using Nested Dissection (биће објављено у Proc. of Forum Bauinformatik 2009)
4. Minsky, M., Papert, S.: On some associative, parallel and analog computations, Associative Information Technologies, Elsevier North Holland (1971)
5. Schnekenburger, Stellner: Dynamic Load Distribution for Parallel Applications, Teubner, Stuttgart (1997)
6. Bernstein, A.J. "Program Analysis for Parallel Processing,' IEEE Trans. on Electronic Computers". EC-15, стр. 757–62. (октобар 1966).
7. Grama, Gupta: Introduction to Parallel Computing, Addison Wesley (2003)